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. Here we’ve gathered the most popular DAG content from our website—everything you need to know as a data engineer. From a walkthrough on DAG building blocks, design, and best practices, to tips on dynamically generating DAGs, testing, debugging, and more! This ebook was created by practitioners for practitioners.

Enjoy!

P.S. Don’t miss the info about our brand new Certification for Apache Airflow DAG Authoring!

Like what you’re reading? Share it with others!

Table of Contents

1. DAGs Where to Begin?

✦

What Exactly is a DAG?

> *If you work in the world of data engineering, you're probably familiar with one of two terms: Data Pipeline (as an analyst/marketer/business-focused person) or DAG (as an engineer who favors term accuracy over branding). Though used in different circles, these terms both represent the same mechanism.*

"Data pipeline" describes the general process by which data moves from one system into another. By using the metaphor of plumbing, it helps illuminate an otherwise complex process. But data isn't literally in a single tube starting on one side and coming out of the other.

Instead, the plumbing metaphor is useful in illustrating three important qualities:

1. When data is moving through pipelines, it is isolated from other data.

2. When a lot of data is moving through a pipeline, it can create stress on the servers it runs on, much like water pressure.

3. The data needs to be treated at various stages before it reaches its destination system, just like chemicals or waste moving through an industrial pipeline.

So what exactly is a DAG, and what does it tell us that's missing from the "pipeline" term?

Let's start with breaking down the acronym:

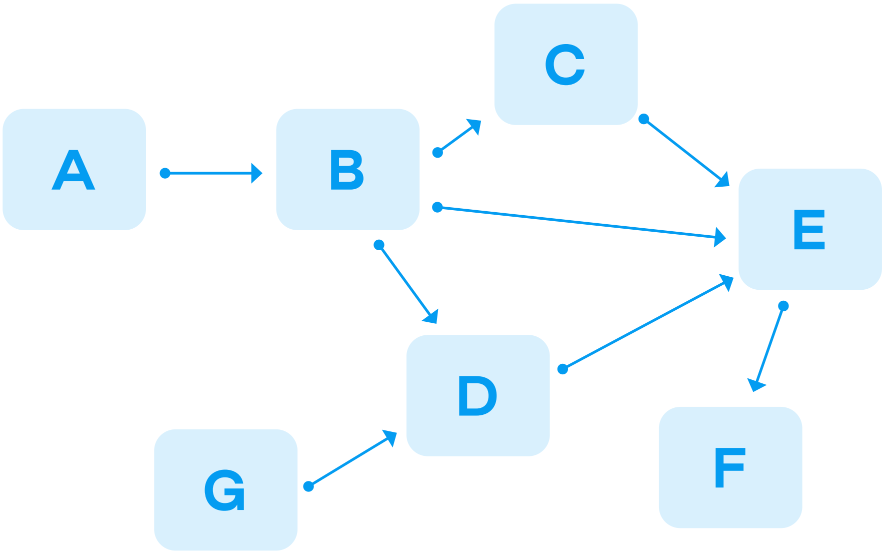
DIRECTED
ACYCLIC
GRAPH

Now let's work through each individual word of the acronym:

> *In mathematics, a graph is a finite set of nodes, with vertices connecting the nodes to each other. In the context of data engineering, each node in a graph represents a data processing task.*

For example, consider the directed acyclic **graph** below. We can create a data engineering story out of these simple nodes and vertices:

- **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.

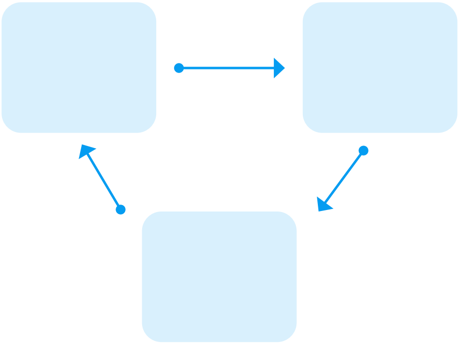


It's pretty neat, but… it doesn't look like a pipeline at all! Tasks branch out and come back together, with new inputs being brought in along the way. So why does this graph represent a pipeline? Because it's both **directed** and **acyclic**!

Have a look at the above graph again – 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* the direction of the vertex. For our 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 the direction as 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**.

Notice that 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, this loop is a waste of data flow!



Data moving through a cyclical graph becomes self-referential, which means that you can't have final, reliable results.

To recap, DAGs are:

1. Directed

If multiple tasks exist, each must have at least one defined upstream (previous) or downstream (subsequent) task, although they could easily have both.

2. Acyclic

No task can create data that goes on to reference itself. It could cause an infinite loop, and that could cause a problem or two.

3. Graph

All tasks are laid out in a clear structure, with discrete processes occurring at set points and transparent relationships with other tasks.

DAGs
as Functional Programming

Best practices emerging around data engineering — specifically, extract, transform, and load (ETL) jobs — often match concepts straight out of functional programming. These include practices around immutability, isolation, and idempotency. In this chapter, we'll explore how DAGs embody these best practices in both their structure and usage.

Idempotency, idempotency, idempotency

This concept will be emphasized throughout the rest of this book. Idempotency is one of, if not the most, essential characteristic of good ETL architecture. The word may sound scary, but the concept is simple.

Something is idempotent if it will produce the same result regardless of how many times it is run. In mathematical terms:

`$f(x) = f(f(x)) = f(f(f(x)))...$`

Idempotency is usually inextricable from reproducibility – a set of inputs always produces the same set of outputs.

Making ETL tasks idempotent makes them easier to execute. If it's just a SQL load for a day's data, implementing upsert logic is pretty easy. If, however, a file is dropped on an externally controlled FTP that won't be there for long, things get a little more tricky.

However, the initial investment is usually worth it for safety, operability, and modularity.

Direct Association

There should be a clear and intuitive association between tables, intermediate files, and all other levels of your data. DAGs are a natural fit here, as every task can have an exclusive target that does not propagate into the target of another task without a direct dependency being set.

Furthermore, this association should filter down into all metadata, such as logs and runtimes.

Mathematically, this idea of clarity and directness is intuitive:
`$f(x) = y$`

✦

From Operators to DagRuns: Implementing DAGs in Airflow

While DAGs are simple structures in and of themselves, defining them in code requires some more complex structures 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 wrappers around types of work.**
  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.

- **An instance of an Operator is a task.**
  Operators are reusable task templates, and a task is an instantiation of an operator. Tasks take the parameters and process data within the context of the DAG.

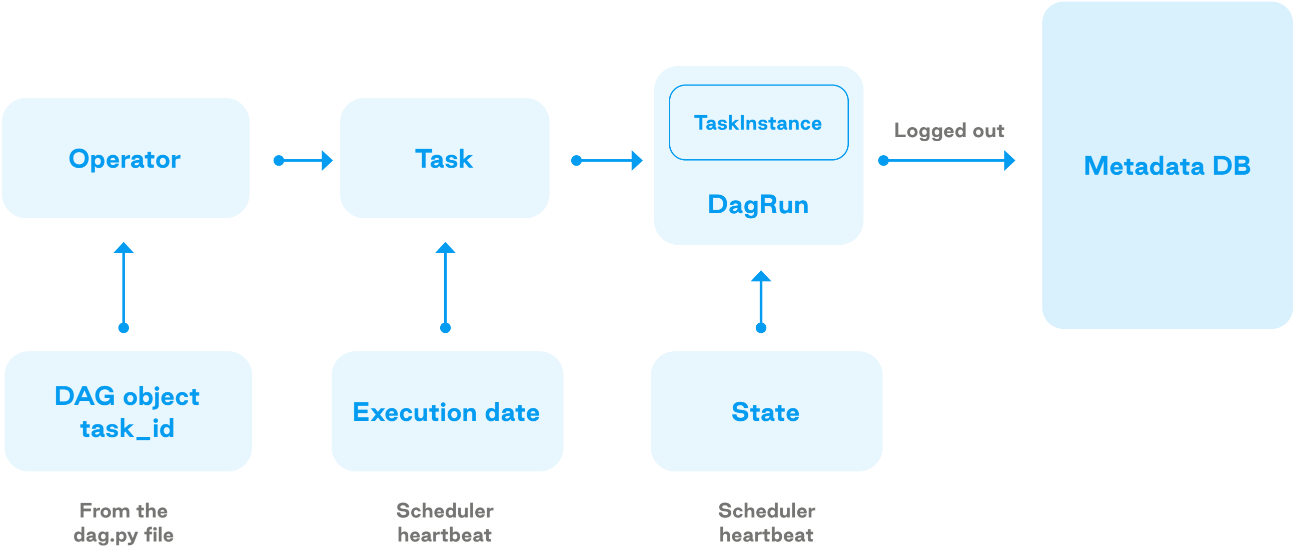
- **Tasks are nodes in a DAG.**

  Once you define tasks within a **DAG** object, a **DAG** can be executed based on its schedule. A built-in Scheduler in Airflow "taps" the **DAG** and begins to execute the **tasks** based on their dependencies.

  A real-time run of a task is called a **task instance**. These task instances get logged in the metadata database with information about when and how they ran.

- **DAGs become DAG Runs**

  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 website with more in-depth Apache Airflow guides.

Explore guides

2. DAG Building Blocks

Operators 101

Operators are the main building blocks of Airflow DAGs. They are classes that encapsulate logic to do a unit of work.
When you create an instance of an operator in a DAG and provide it with its required parameters, it becomes a task. Many tasks can be added to a DAG along with their dependencies. When Airflow executes that task for a given `execution_date`, it becomes a task instance.

To browse and search all of the available Operators in Airflow, visit the **Astronomer Registry**. The following are examples of Operators that are frequently used in Airflow projects.

BashOperator

```
t1 = BashOperator(
        task_id='bash_hello_world',
        dag=dag,
        bash_command='echo "Hello World"'
        )
```

This **BashOperator** simply runs a bash command and echos `"Hello World"`.

Github: **BashOperator Code**

Python Operator

```
def hello(**kwargs):
    print(‘Hello from {kw}’.format(kw=kwargs[‘my_keyword’]))

t2 = PythonOperator(
        task_id=‘python_hello’,
        dag=dag,
        python_callable=hello,
        op_kwargs={‘my_keyword’: ‘Airflow’}
        )
```

The **PythonOperator** calls a python function defined earlier in our code. You can pass parameters to the function via the `op_kwargs` parameter. This task will print “Hello from Airflow” when it runs.

Github: **PythonOperator Code**

Postgres Operator

```
t3 = PostgresOperator(
        task_id=‘PythonOperator’,
        sql=‘CREATE TABLE my_table (my_column varchar(10));’,
        postgres_conn_id=‘my_postgres_connection’,
        autocommit=False
    )
```

This Operator issues a SQL statement against a Postgres database. Credentials for the database are stored in an Airflow connection called `my_postgres_connection`. If you look at the code for the **PostgresOperator**, it uses a **PostgresHook** to actually interact with the database.

Github: **PostgresOperator**

SSH Operator

```
t4 = SSHOperator(
        task_id=’SSHOperator’,
        ssh_conn_id=’my_ssh_connection’,
        command=’echo “Hello from SSH Operator”’
    )
```

Like the `BashOperator`, the **SSHOperator** allows you to run a bash command, but has built-in support to SSH into a remote machine to run commands there.

The private key to authenticate to the remote server is stored in Airflow Connections as `my_ssh_conenction`. This key can be referred to in all DAGs, so the Operator itself only needs the command you want to run. This Operator uses an **SSHHook** to establish the ssh connection and run the command. Github: **SSHOperator Code**

S3 To Redshift Operator

```
t5 = S3ToRedshiftOperator(
        task_id=’S3ToRedshift’,
        schema=’public’,
        table=’my_table’,
        s3_bucket=’my_s3_bucket’,
        s3_key=’{{ ds_nodash }}/my_file.csv’,
        redshift_conn_id=’my_redshift_connection’,
        aws_conn_id=’my_aws_connection’
    )
```

The **S3ToRedshiftOperator** loads data from S3 to Redshift via Redshift's COPY command. This is in a family of Operators called `Transfer Operators` – Operators designed to move data from one system (S3) to another (Redshift). Notice it has two Airflow connections in the parameters, one for Redshift and one for S3.

This also uses another concept – **macros and templates**. In the `s3_key` parameter, Jinja template notation is used to pass the execution date for this DAG Run formatted as a string with no dashes (`ds_nodash` – a pre-defined macro in Airflow). It will look for a key formatted similarly to `my_s3_bucket/20190711/my_file.csv`, with the timestamp dependent on when the file ran.

Templates can be used to determine runtime parameters (e.g. the range of data for an API call) and also make your code idempotent (each intermediary file is named for the data range it contains).
Github: **S3ToRedshiftOperator Code**

✦ Hooks 101

Operators are the main building blocks of Airflow, but Operators rely heavily upon Hooks to interact with all of their source and destination systems.

Hooks are used as a way to abstract the methods you would use against a source system. Hooks should always be used to connect with any external system. The **S3Hook** below shows how a hook can import a standard library (in this case, boto3) and expose some of the most common methods.

```
class S3Hook(AwsHook):
    """
    Interact with AWS S3, using the boto3 library.
    """

    def get_conn(self):
        return self.get_client_type('s3')

    @staticmethod
    def parse_s3_url(s3url):
            return bucket_name, key

    def check_for_bucket(self, bucket_name):
            return False

    def get_bucket(self, bucket_name):
        return s3.Bucket(bucket_name)

    def create_bucket(self, bucket_name, region_name=None):
        return None

    def check_for_prefix(self, bucket_name, prefix, delimiter):
        return False if plist is None else prefix in plist

    def list_prefixes(self, bucket_name, prefix=’’, delimiter=’’,
page_size=None, max_items=None):
            return prefixes

    def list_keys(self, bucket_name, prefix=’’, delimiter=’’, page_
size=None, max_items=None):
            return keys

    def check_for_key(self, key, bucket_name=None):
        return obj

    def read_key(self, key, bucket_name=None):
        return obj.get()[‘Body’].read().decode(‘utf-8’)

    def select_key(self, key, bucket_name=None,
                   expression=‘SELECT * FROM S3Object’,
                   expression_type=‘SQL’,
                   input_serialization=None,
                   output_serialization=None):
        return ’’.join(event[‘Records’][‘Payload’].decode(‘utf-8’)
                       for event in response[‘Payload’]
                       if ‘Records’ in event)

    def load_file(self, filename, key, bucket_name=None, replace=-
False, encrypt=False):
        return None

    def load_string(self, string_data, key, bucket_name=None, re-
place=False, encrypt=False,
                    encoding='utf-8'):
        return None

    def load_bytes(self, bytes_data, key, bucket_name=None, re-
place=False, encrypt=False):
       return None

    def load_file_obj(self, file_obj, key, bucket_name=None, re-
place=False, encrypt=False):
       return None

    def copy_object(self, source_bucket_key, dest_bucket_key,
source_bucket_name=None, dest_bucket_name=None, source_version_
id=None):
        return response

    def delete_objects(self, bucket, keys):
        return response
```

This Hook inherits from the **AwsBaseHook**, which inherits from the **BaseHook**. All Hooks inherit from the BaseHook which contains the logic for how hooks interact with Airflow `Connections`. `Connections` are Airflow's built-in credential-store for your source/destination systems. Hooks are designed to handle these in a clean, reusable way. Tasks that use a hook will have an input parameter for the `conn_id` of the connection you wish to use.

To browse and search all of the available Hooks in Airflow, visit the **Registry.**

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:

S3 Key Sensor

```
s1 = S3KeySensor(
        task_id='s3_key_sensor',
        bucket_key='{{ ds_nodash }}/my_file.csv',
        bucket_name='my_s3_bucket',
        aws_conn_id='my_aws_connection',
    )
```

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.

3. DAG Design

✦

DAG Writing Best Practices in Apache Airflow

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

For an in-depth walk-through and examples of some of the concepts covered here, check out our DAG writing best practices **webinar recording**, as well as our **Github repo** with good and bad example DAGs.

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 automized 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:

```
# Variables used by tasks
# Bad example - Define today’s and yesterday’s date using date-
time module
today = datetime.today()
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:

```
# Variables used by tasks
# Good example - Define yesterday’s date with an Airflow variable
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 on 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.

Because Airflow executes all code in the `DAGS_Folder` on every scheduler heartbeat, 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:

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import Post-
gresOperator
from datetime import datetime, timedelta

#Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: False,
    ‘email_on_retry’: False,
    ‘retries’: 1,
    ‘retry_delay’: timedelta(minutes=1)
}
```

```
#Instantiate DAG
with DAG(’bad_practices_dag_1’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval=’@daily’,
         default_args=default_args,
         catchup=False
         ) as dag:

    t0 = DummyOperator(task_id=’start’)

    #Bad example with top level SQL code in the DAG file
    query_1 = PostgresOperator(
        task_id=’covid_query_wa’,
        postgres_conn_id=’postgres_default’,
        sql=’’’with yesterday_covid_data as (
                SELECT *
                FROM covid_state_data
                WHERE date = {{ params.today }}
                AND state = ‘WA’
            ),
            today_covid_data as (
                SELECT *
                FROM covid_state_data
                WHERE date = {{ params.yesterday }}
                AND state = ‘WA’
            ),
two_day_rolling_avg as (

SELECT AVG(a.state, b.state) as two_day_avg
                FROM yesterday_covid_data a
                JOIN yesterday_covid_data b
                ON a.state = b.state
            )
            SELECT a.state, b.state, c.two_day_avg
            FROM yesterday_covid_data a
            JOIN today_covid_data b
            ON a.state=b.state
            JOIN two_day_rolling_avg c
            ON a.state=b.two_day_avg;’’’,
            params={‘today’: today, ‘yesterday’:yesterday}
    )
```

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:

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import Post-
gresOperator
from datetime import datetime, timedelta

#Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: False,
    ‘email_on_retry’: False,
    ‘retries’: 1,
    ‘retry_delay’: timedelta(minutes=1)
}

#Instantiate DAG
with DAG(‘good_practices_dag_1’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval=‘@daily’,
         default_args=default_args,
         catchup=False,
         template_searchpath=‘/usr/local/airflow/include’ #include
path to look for external files
         ) as dag:

        query = PostgresOperator(
            task_id=‘covid_query_{0}’.format(state),
            postgres_conn_id=‘postgres_default’,
            sql=‘covid_state_query.sql’, #reference query kept in
separate file
            params={‘state’: “‘” + state + “’”}
        )
```

Another example that goes against this best practice is the DAG below, which dynamically generates PostgresOperator tasks based on records pulled from a database.

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
from airflow.providers.postgres.hooks.postgres import PostgresHook
from datetime import datetime, timedelta

hook = PostgresHook(‘database_conn’)
result = hook.get_records(”SELECT * FROM grocery_list;”)

with DAG(‘bad_practices_dag_2’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval=’@daily’,
         default_args=default_args,
         catchup=False
         ) as dag:

    for grocery_item in result:
        query = PostgresOperator(
            task_id=’query_{0}’.format(result),
            postgres_conn_id=’postgres_default’,
            sql=”INSERT INTO purchase_order VALUES (value1, value2, value3);”
        )
```

When the scheduler parses this DAG, it will query the `grocery_list` table to construct the Operators in the DAG. This query will be run on every scheduler heartbeat, which could cause performance issues. A better implementation would be to have a separate DAG or task that gets the required information from the `grocery_list` table and saves it to an XCom or an Airflow variable, which can be used by the dynamic 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:

```
task_1.set_downstream(task_2)
task_3.set_upstream(task_2)
task_3 >> task_4
```

Try to be consistent with something like this:

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

Use Airflow as an Orchestrator

> *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 way you can use Airflow as an orchestrator is by making use of existing **provider packages**, which orchestrate jobs using third party tools. One of the best aspects of Airflow is its robust and active community, which has resulted in many integrations between Airflow and other tools in the data ecosystem. Wherever possible, it's best practice to make use of these integrations. This makes it easier for teams using existing tools to adopt Airflow, and it means you get to write less code since many existing hooks and Operators have taken care of that for you.

Don't Use Airflow as a Processing Framework

Conversely, Airflow was not designed to be a processing framework. Since DAGs are written in Python, it can be tempting to make use of data processing libraries like Pandas. However, processing large amounts of data within your Airflow tasks does not scale and should only be used in cases where the data are limited in size. A better option for scalability is to offload any heavy-duty processing to a framework like **Apache Spark**, then use Airflow to orchestrate those jobs.

If you must process small data within Airflow, 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 avoids using Airflow as a processing framework. 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:

```
├── dags/ # Where your DAGs go
│   ├── example-dag.py # An example dag that comes with the ini-
tialized project
├── Dockerfile # For Astronomer's Docker image and runtime over-
rides
├── include/ # For any other files you'd like to include
├── plugins/ # For any custom or community Airflow plugins
├── packages.txt # For OS-level packages
└── 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.

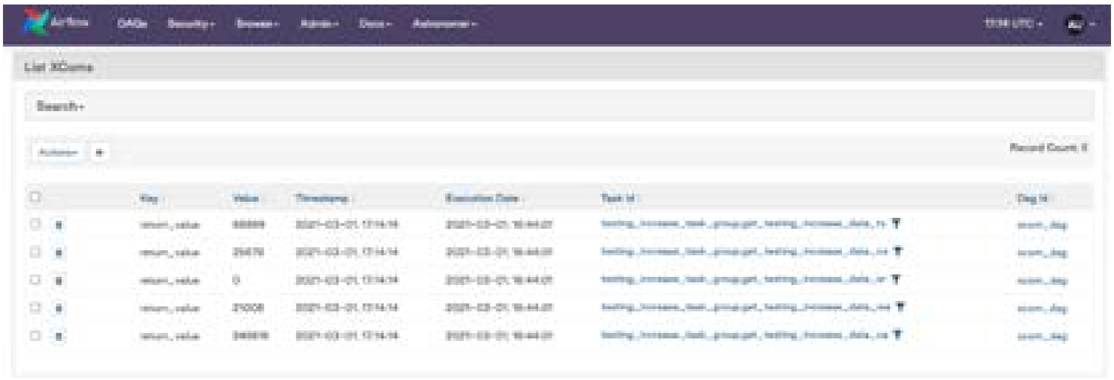
XCom

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 value (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:



List XComs

Search

Actions

Record Count: 5

	Key	Value	Timestamp	Execution Date	Task Id	Dag Id
□	return_value	88889	2021-03-01, 17:14:14	2021-03-01, 16:44:07	testing_increase_task_group.get_testing_increase_data_[illegible]	xcom_dag
□	return_value	25670	2021-03-01, 17:14:14	2021-03-01, 16:44:07	testing_increase_task_group.get_testing_increase_data_[illegible]	xcom_dag
□	return_value	0	2021-03-01, 17:14:14	2021-03-01, 16:44:07	testing_increase_task_group.get_testing_increase_data_[illegible]	xcom_dag
□	return_value	21008	2021-03-01, 17:14:14	2021-03-01, 16:44:07	testing_increase_task_group.get_testing_increase_data_[illegible]	xcom_dag
□	return_value	248818	2021-03-01, 17:14:14	2021-03-01, 16:44:07	testing_increase_task_group.get_testing_increase_data_[illegible]	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.

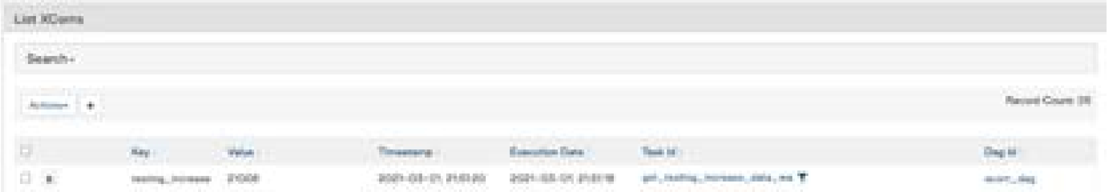
```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

import requests
import json

url = 'https://covidtracking.com/api/v1/states/'
state = 'wa'

def get_testing_increase(state, ti):
    """
    Gets totalTestResultsIncrease field from Covid API for given state and returns value
    """
    res = requests.get(url+'{0}/current.json'.format(state))
    testing_increase = json.loads(res.text)['totalTestResultsIncrease']

    ti.xcom_push(key='testing_increase', value=testing_increase)

def analyze_testing_increases(state, ti):
    """
    Evaluates testing increase results
    """
    testing_increases=ti.xcom_pull(key='testing_increase', task_ids='get_testing_increase_data_{0}'.format(state))
    print('Testing increases for {0}:'.format(state), testing_increases)
    #run some analysis here

```

```
# Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: False,
    ‘email_on_retry’: False,
    ‘retries’: 1,
    ‘retry_delay’: timedelta(minutes=5)
}

with DAG(‘xcom_dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=2,
         schedule_interval=timedelta(minutes=30),
         default_args=default_args,
         catchup=False
         ) as dag:

    opr_get_covid_data = PythonOperator(
        task_id = ‘get_testing_increase_data_{0}’.format(state),
        python_callable=get_testing_increase,
        op_kwargs={‘state’:state}
    )

    opr_analyze_testing_data = PythonOperator(
        task_id = ‘analyze_data’,
        python_callable=analyze_testing_increases,
                      op_kwargs={‘state’:state}
    )

    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.



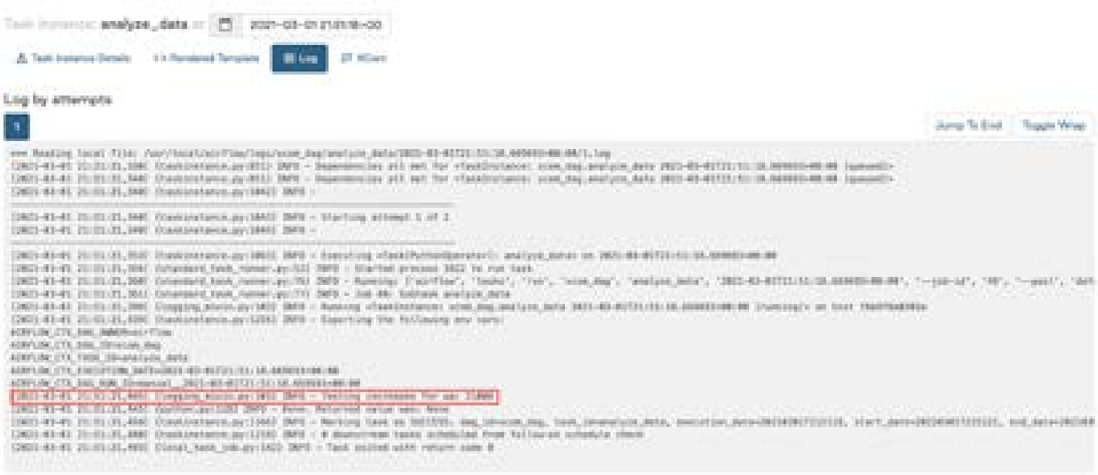
List XComs

Search▾

Actions▾ + Record Count: [illegible]

□		Key	Value	Timestamp	Execution Date	Task Id	Dag Id
□	✎	testing_increase	21008	2021-03-01 21:03:20	2021-03-01 21:01:18	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.



```
*** Reading local file: /usr/local/airflow/logs/xcom_dag/analyze_data/2021-03-01T21:51:18.[illegible]+00:00/1.log
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:851} INFO - Dependencies all met for <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.[illegible]+00:00 [queued]>
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:851} INFO - Dependencies all met for <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.[illegible]+00:00 [queued]>
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1042} INFO -
--------------------------------------------------------------------------------
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1043} INFO - Starting attempt 1 of 1
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1044} INFO -
--------------------------------------------------------------------------------
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1063} INFO - Executing <Task(PythonOperator): analyze_data> on 2021-03-01T21:51:18.[illegible]+00:00
[2021-03-01 21:51:21,[illegible]] {standard_task_runner.py:52} INFO - Started process [illegible] to run task
[2021-03-01 21:51:21,[illegible]] {standard_task_runner.py:76} INFO - Running: ['airflow', 'tasks', 'run', 'xcom_dag', 'analyze_data', '2021-03-01T21:51:18.[illegible]+00:00', '--job-id', '[illegible]', '--pool', 'def
[2021-03-01 21:51:21,[illegible]] {standard_task_runner.py:77} INFO - Job [illegible]: Subtask analyze_data
[2021-03-01 21:51:21,[illegible]] {logging_mixin.py:103} INFO - Running <TaskInstance: xcom_dag.analyze_data 2021-03-01T21:51:18.[illegible]+00:00 [running]> on host [illegible]
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1255} 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.[illegible]+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2021-03-01T21:51:18.[illegible]+00:00
[2021-03-01 21:51:21,[illegible]] {logging_mixin.py:103} INFO - Testing increases for wa: [illegible]
[2021-03-01 21:51:21,[illegible]] {python.py:118} INFO - Done. Returned value was: None
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1166} INFO - Marking task as SUCCESS. dag_id=xcom_dag, task_id=analyze_data, execution_date=[illegible], start_date=[illegible], end_date=[illegible]
[2021-03-01 21:51:21,[illegible]] {taskinstance.py:1220} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2021-03-01 21:51:21,[illegible]] {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:

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

import requests
import json

url = 'https://covidtracking.com/api/v1/states/'
state = 'wa'

default_args = {
    'start_date': datetime(2021, 1, 1)
}
```

```
@dag(’xcom_taskflow_dag’, schedule_interval=’@daily’, default_args=default_args, catchup=False)
def taskflow():

    @task
    def get_testing_increase(state):
        """
        Gets totalTestResultsIncrease field from Covid API for given state and returns value
        """
        res = requests.get(url+’{0}/current.json’.format(state))
        return{’testing_increase’: json.loads(res.text)[’totalTestResultsIncrease’]}

    @task
    def analyze_testing_increases(testing_increase: int):
        """
        Evaluates testing increase results
        """
        print(’Testing increases for {0}:’.format(state), testing_increase)
        #run some analysis here

    analyze_testing_increases(get_testing_increase(state))

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

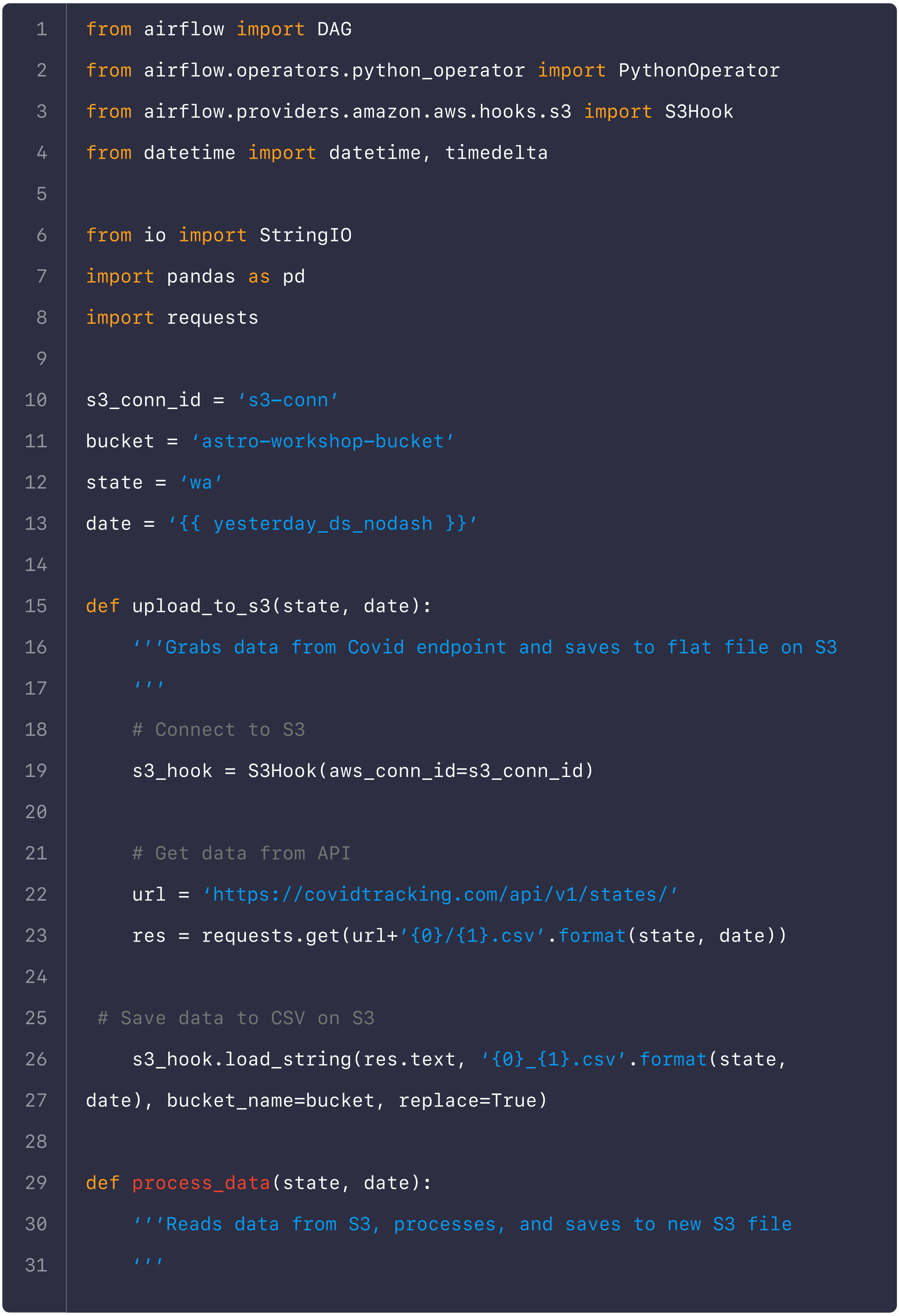
As mentioned above, XCom can be a great option for sharing data between tasks because it doesn't rely on any tools external to Airflow itself. However, it is only designed to be used for very small amounts of data. What if the data you need to pass is a little bit larger, for example, a small dataframe?

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.



```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from datetime import datetime, timedelta

from io import StringIO
import pandas as pd
import requests

s3_conn_id = ‘s3-conn’
bucket = ‘astro-workshop-bucket’
state = ‘wa’
date = ‘{{ yesterday_ds_nodash }}’

def upload_to_s3(state, date):
    '''Grabs data from Covid endpoint and saves to flat file on S3
    '''
    # Connect to S3
    s3_hook = S3Hook(aws_conn_id=s3_conn_id)

    # Get data from API
    url = ‘https://covidtracking.com/api/v1/states/’
    res = requests.get(url+‘{0}/{1}.csv’.format(state, date))

 # Save data to CSV on S3
    s3_hook.load_string(res.text, ‘{0}_{1}.csv’.format(state,
date), bucket_name=bucket, replace=True)

def process_data(state, date):
    '''Reads data from S3, processes, and saves to new S3 file
    '''
    # Connect to S3
    s3_hook = S3Hook(aws_conn_id=s3_conn_id)

    # Read data
    data = StringIO(s3_hook.read_key(key=’{0}_{1}.csv’.format(state, date), bucket_name=bucket))
    df = pd.read_csv(data, sep=’,’)

    # Process data
    processed_data = df[[’date’, ’state’, ’positive’, ’negative’]]

    # Save processed data to CSV on S3
    s3_hook.load_string(processed_data.to_string(), ’{0}_{1}_processed.csv’.format(state, date), bucket_name=bucket, replace=True)

# Default settings applied to all tasks
default_args = {
    ’owner’: ’airflow’,
    ’depends_on_past’: False,
    ’email_on_failure’: False,
    ’email_on_retry’: False,
    ’retries’: 1,
    ’retry_delay’: timedelta(minutes=1)
}

with DAG(’intermediary_data_storage_dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=’@daily’,
         default_args=default_args,
         catchup=False
         ) as dag:

    generate_file = PythonOperator(
        task_id=’generate_file_{0}’.format(state),
        python_callable=upload_to_s3,
        op_kwargs={’state’: state, ‘date’: date}
    )

    process_data = PythonOperator(
        task_id=’process_data_{0}’.format(state),
        python_callable=process_data,
        op_kwargs={’state’: state, ‘date’: date}
    )

    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.

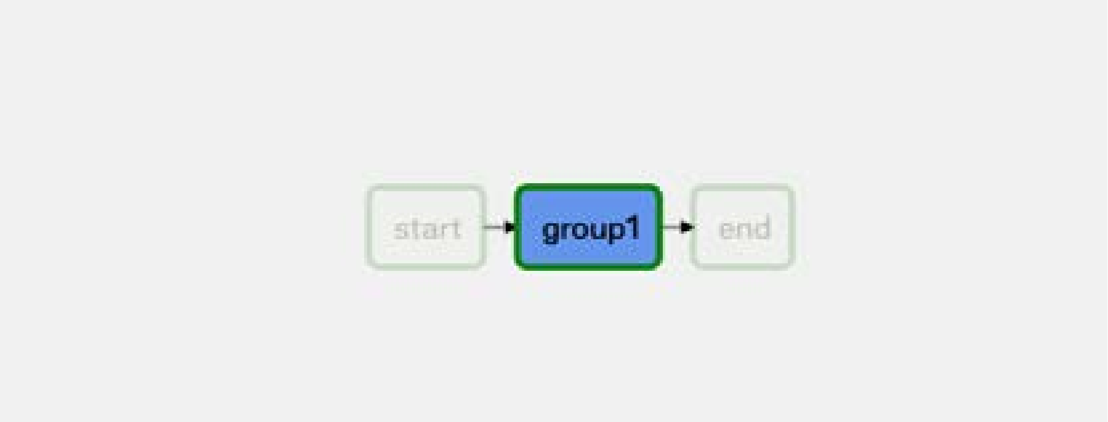
```python
from airflow.utils.task_group import TaskGroup
```

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`:

```python
t0 = DummyOperator(task_id='start')

# Start Task Group definition
with TaskGroup(group_id='group1') as tg1:
    t1 = DummyOperator(task_id='task1')
    t2 = DummyOperator(task_id='task2')

    t1 >> t2
# End Task Group definition

t3 = DummyOperator(task_id='end')

# Set Task Group's (tg1) dependencies
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.

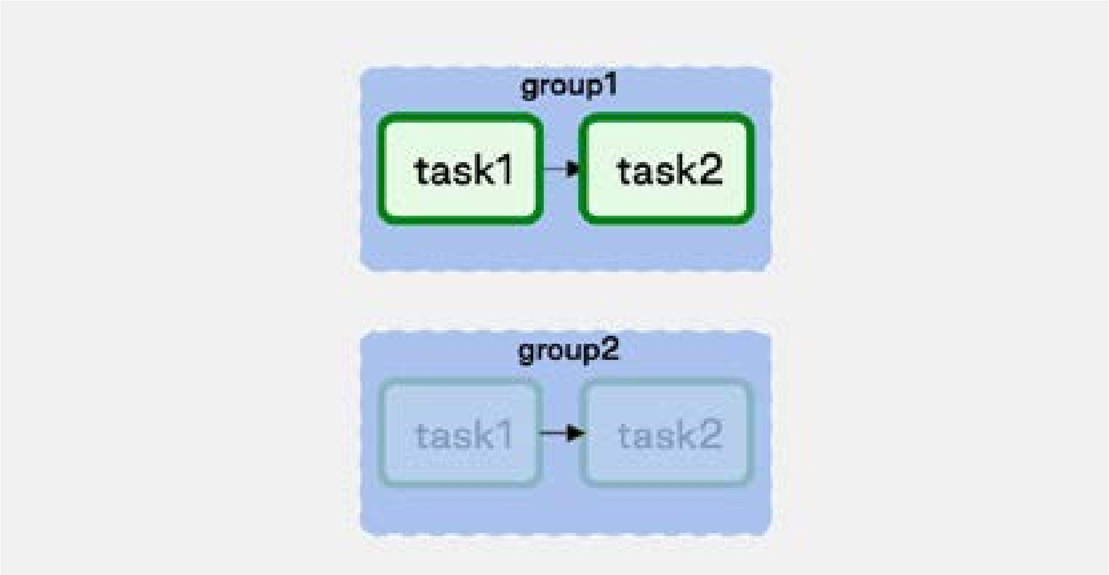
Dynamically Generating Task Groups

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`:

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

        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.

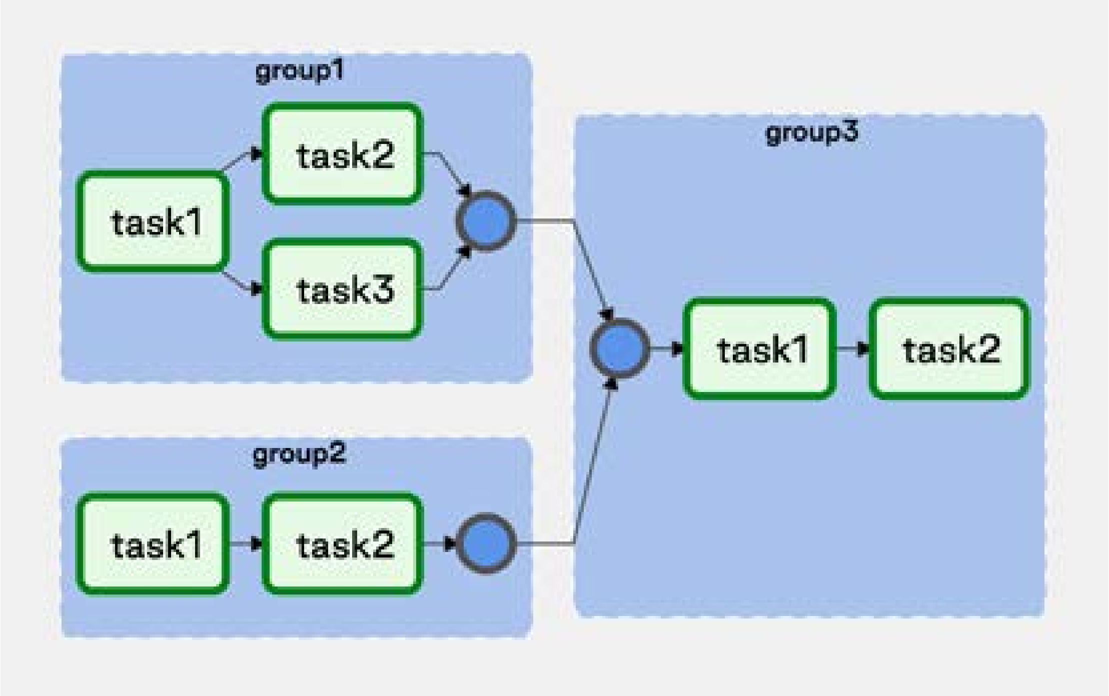
Ordering Task Groups

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:

```
groups = []
for g_id in range(1,4):
    tg_id = f'group{g_id}'
    with TaskGroup(group_id=tg_id) as tg1:
        t1 = DummyOperator(task_id='task1')
        t2 = DummyOperator(task_id='task2')

        t1 >> t2

        if tg_id == 'group1':
            t3 = DummyOperator(task_id='task3')
            t1 >> t3

        groups.append(tg1)

[groups[0] , groups[1]] >> groups[2]
```

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



Conditioning on Task Groups

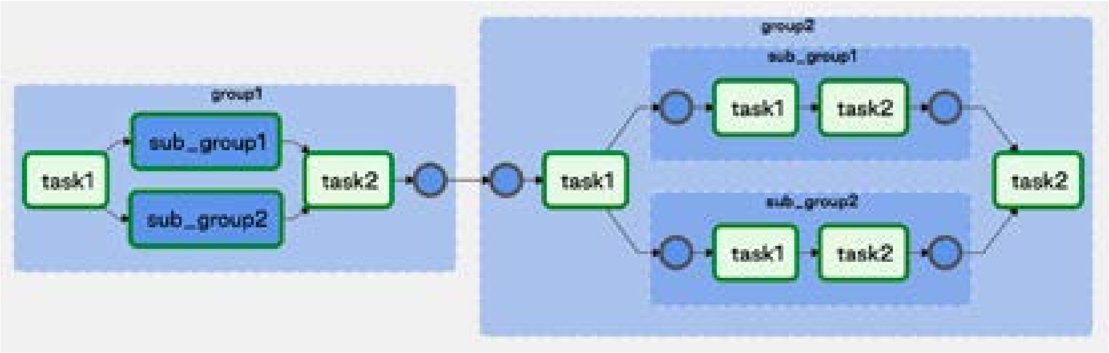
In the above example, we added an additional task to `group1` based on our `group_id`. This was to demonstrate that even though we are dynamically creating Task Groups to take advantage of patterns, we can still introduce variations to the pattern while avoiding code redundancies from building each Task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

```
groups = []
for g_id in range(1,3):
    with TaskGroup(group_id=f’group{g_id}’) as tg1:
        t1 = DummyOperator(task_id=’task1’)
        t2 = DummyOperator(task_id=’task2’)

        sub_groups = []
        for s_id in range(1,3):
            with TaskGroup(group_id=f’sub_group{s_id}’) as tg2:
                st1 = DummyOperator(task_id=’task1’)
                st2 = DummyOperator(task_id=’task2’)

                st1 >> st2
                sub_groups.append(tg2)

        t1 >> sub_groups >> t2
        groups.append(tg1)

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

Overview

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 will 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:

- **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 section, we will 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.**

Implementing Cross-DAG Dependencies

There are multiple ways to implement cross-DAG dependencies in Airflow, including the `TriggerDagRunOperator`, `ExternalTaskSensor`, and the Airflow API. You'll want to use different implementations depending on your DAG dependencies and Airflow setup. 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.

TriggerDagRunOperator

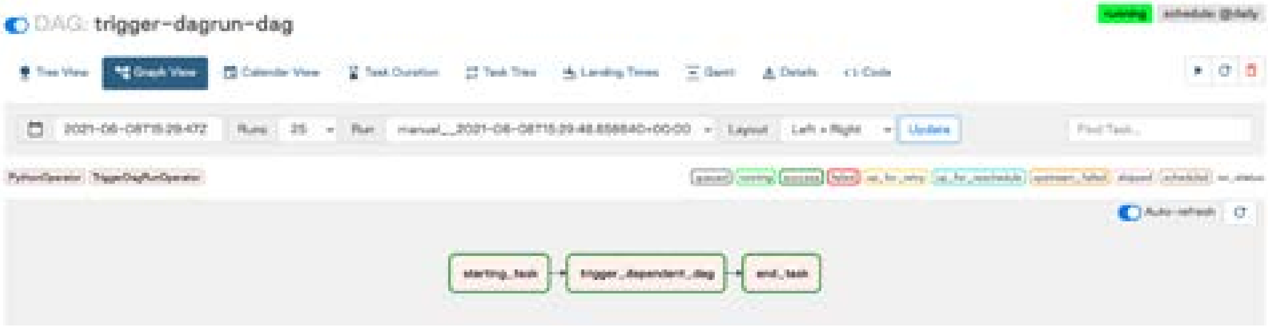
The `TriggerDagRunOperator` is an easy way to implement cross-DAG dependencies. This Operator allows you to have a task in one DAG that triggers another DAG in the same Airflow environment.

The `TriggerDagRunOperator` is ideal in situations where you have one upstream DAG that needs to trigger one or more downstream DAGs. It can also work if you have dependent DAGs that have both upstream and downstream tasks in the upstream DAG (i.e. the dependent DAG is in the middle of tasks in the upstream DAG). Because you can use this Operator for any task in your DAG, it is highly flexible. It's also an ideal replacement for SubDAGs.

Below is an example DAG that implements the `TriggerDagRunOperato`r to trigger the `dependent-dag` between two other tasks.

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.trigger_dagrun import TriggerDagRunOperator
from datetime import datetime, timedelta

def print_task_type(**kwargs):
    """
    Dummy function to call before and after dependent DAG.
    """
    print(f”The {kwargs[‘task_type’]} task has completed.”)

# Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: False,
    ‘email_on_retry’: False,
    ‘retries’: 1,
    ‘retry_delay’: timedelta(minutes=5)
}

with DAG(‘trigger-dagrun-dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=‘@daily’,
         default_args=default_args,
         catchup=False
         ) as dag:

    start_task = PythonOperator(
        task_id=‘starting_task’,
        python_callable=print_task_type,
        op_kwargs={‘task_type’: ‘starting’}
    )

   trigger_dependent_dag = TriggerDagRunOperator(
       task_id=”trigger_dependent_dag”,
       trigger_dag_id=”dependent-dag”,
       wait_for_completion=True
   )

   end_task = PythonOperator(
       task_id=’end_task’,
       python_callable=print_task_type,
       op_kwargs={’task_type’: ’ending’}
   )

   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`.



There are a couple of things to note when using this Operator:

- 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.

- If your upstream DAG has downstream tasks that require the downstream DAG to finish first, you should set the `wait_for_completion` param to `True` as shown in the example above. This param defaults to `False`, meaning once the downstream DAG has started, the upstream DAG will mark the task as a success and move on to any downstream tasks.

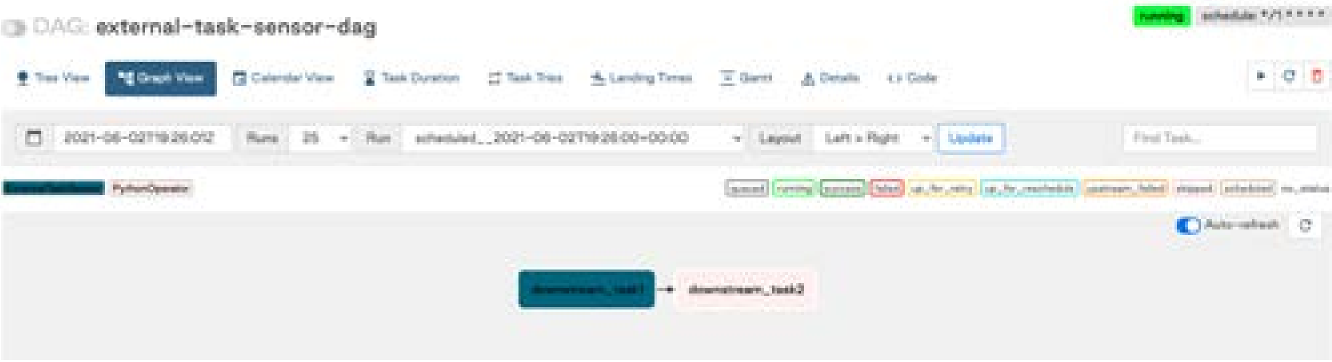
ExternalTaskSensor

The next method for creating cross-DAG dependencies is to add an `ExternalTaskSensor` to your downstream DAG. The downstream DAG will wait until a task is completed in the upstream DAG before moving on to the rest of the DAG. You can find more info on this sensor on the Astronomer Registry.

This method is not as flexible as the `TriggerDagRunOperator`, since the dependency is implemented in the downstream DAG. It is ideal in situations where you have a downstream DAG that is dependent on multiple upstream DAGs. An example DAG using the `ExternalTaskSensor` is shown below:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.sensors.external_task import ExternalTaskSensor
from datetime import datetime, timedelta

def downstream_fuction():
    """
    Downstream function with print statement.
    """
    print(‘Upstream DAG has completed. Starting other tasks.’)

default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: False,
    ‘email_on_retry’: False,
    ‘retries’: 1,
    ‘retry_delay’: timedelta(minutes=5)
}

with DAG(‘external-task-sensor-dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval=‘*/1 * * * *’,
         catchup=False
         ) as dag:

    downstream_task1 = ExternalTaskSensor(
        task_id=”downstream_task1”,
        external_dag_id=‘example_dag’,
        external_task_id=‘bash_print_date2’,
 allowed_states=['success'],
        failed_states=['failed', 'skipped']
    )

    downstream_task2 = PythonOperator(
        task_id='downstream_task2',
        python_callable=downstream_fuction,
        provide_context=True
    )

    downstream_task1 >> downstream_task2
```

In this DAG, `downstream_task1` waits for the `bash_print_date2` task of `example_dag` to complete before moving on to executing the rest of the downstream tasks (`downstream_task2`). The graph view of the DAG looks like this:



2021-06-02T19:26:01Z Runs 25 Run scheduled__2021-06-02T19:26:00+00:00 Layout Left > Right

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 this case, 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, 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 the completion of the specified task or DAG at the same `execution_date`. To look for the 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 full 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 documentation**.

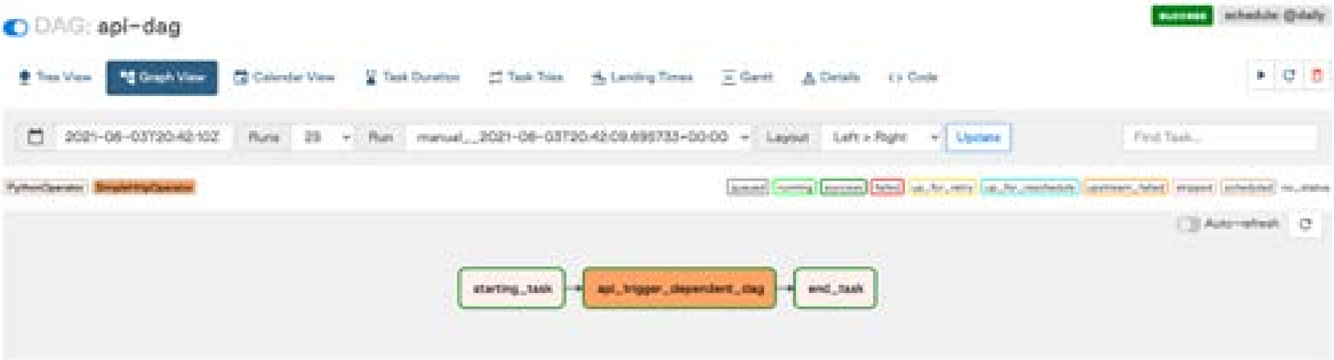
This method is useful if your dependent DAGs live in different Airflow environments (more on this in the Cross-Deployment Dependencies section below), or if the downstream DAG does not have any downstream dependencies in the upstream DAG (e.g. the downstream DAG is the last task in the upstream DAG). One drawback to this method is that the task triggering the downstream DAG will complete once the API call is complete, rather than when the downstream DAG 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:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.http.operators.http import SimpleHttpOperator
from datetime import datetime, timedelta
import json

# Define body of POST request for the API call to trigger another DAG
date = ’{{ execution_date }}’
request_body = {
  ”execution_date”: date
}
json_body = json.dumps(request_body)

def print_task_type(**kwargs):
    ”””
    Dummy function to call before and after downstream DAG.
    ”””
    print(f”The {kwargs[’task_type’]} task has completed.”)
    print(request_body)

default_args = {
    ’owner’: ’airflow’,
    ’depends_on_past’: False,
    ’email_on_failure’: False,
    ’email_on_retry’: False,
    ’retries’: 1,
    ’retry_delay’: timedelta(minutes=5)
}

```

```
with DAG(’api-dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=’@daily’,
         catchup=False
         ) as dag:

    start_task = PythonOperator(
        task_id=’starting_task’,
        python_callable=print_task_type,
        op_kwargs={’task_type’: ‘starting’}
    )

    api_trigger_dependent_dag = SimpleHttpOperator(
        task_id=”api_trigger_dependent_dag”,
        http_conn_id=’airflow-api’,
        endpoint=’/api/v1/dags/dependent-dag/dagRuns’,
        method=’POST’,
        headers={’Content-Type’: ‘application/json’},
        data=json_body
    )

    end_task = PythonOperator(
        task_id=’end_task’,
        python_callable=print_task_type,
        op_kwargs={’task_type’: ‘ending’}
    )

    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:

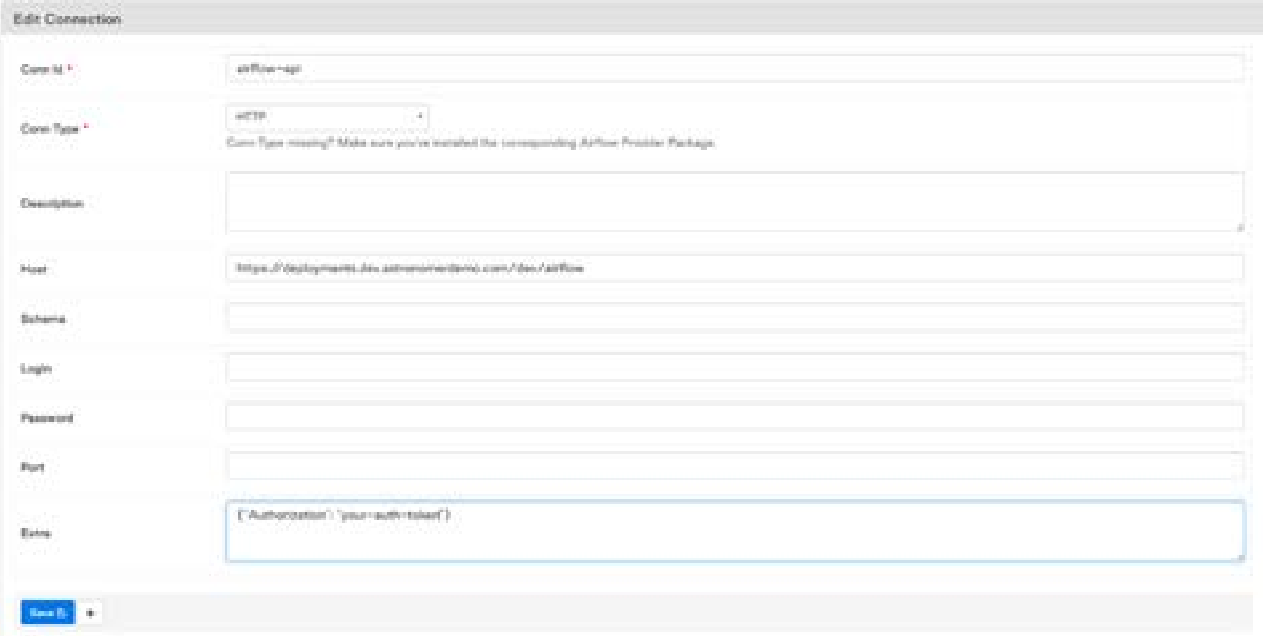


In order to use the `SimpleHttpOperator` to trigger another DAG, you need to define the following:

- `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.



Edit Connection

Conn Id * airflow-api

Conn Type * HTTP

Conn Type missing? Make sure you've installed the corresponding Airflow Provider Package.

Description

Host https://deployments.dev.astronomerdemo.com/dev/airflow

Schema

Login

Password

Port

Extra {"Authorization": "your-auth-token"}

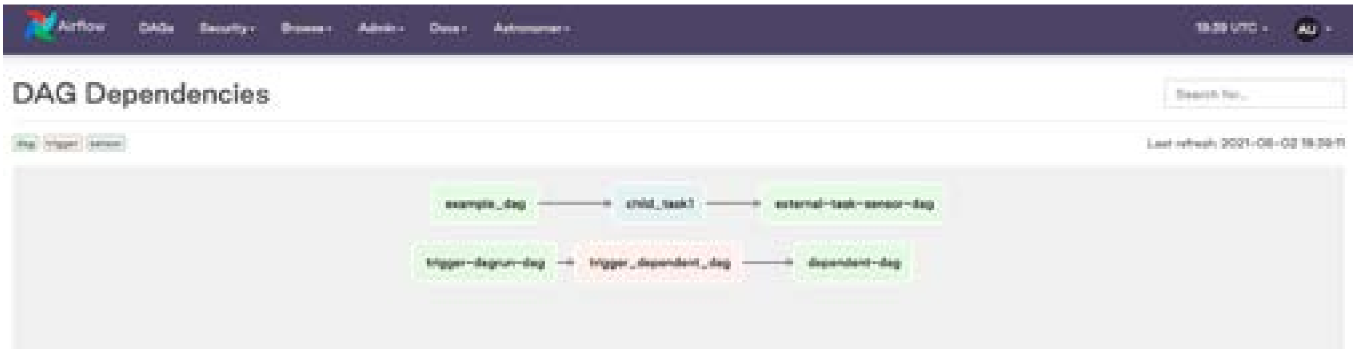
DAG Dependencies View

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 a `TriggerDagRunOperator`
- Using an `ExternalTaskSensor`

Dependencies can be viewed in the UI by going to `Browse` → DAG Dependencies.

The screenshot below shows the dependencies created by the `TriggerDagRunOperator` and ExternalTaskSensor example DAGs in the sections above.



Cross-Deployment Dependencies

To implement cross-DAG dependencies on two different Airflow environments on the Astronomer platform, 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

Note: All code in this section can be found in this Github repo.

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.

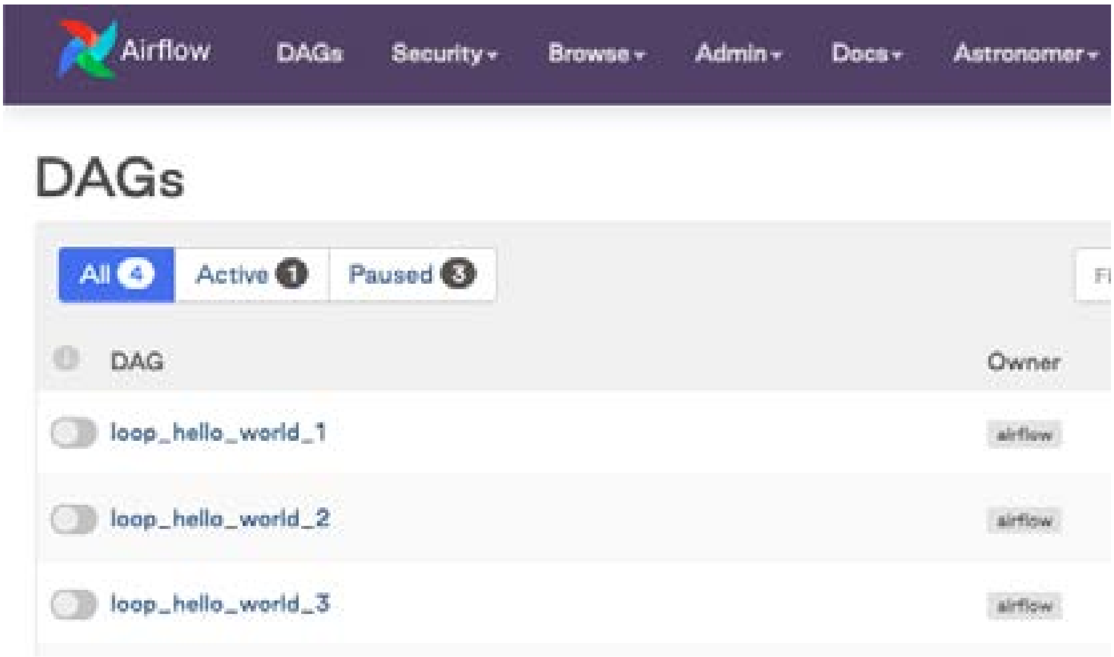
```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime


def create_dag(dag_id,
               schedule,
               dag_number,
               default_args):

    def hello_world_py(*args):
        print(‘Hello World’)
        print(‘This is DAG: {}’.format(str(dag_number)))

    dag = DAG(dag_id,
              schedule_interval=schedule,
              default_args=default_args)

    with dag:
        t1 = PythonOperator(
            task_id=‘hello_world’,
            python_callable=hello_world_py,
            dag_number=dag_number)

    return dag
```

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:

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime


def create_dag(dag_id,
               schedule,
               dag_number,
               default_args):

   def hello_world_py(*args):
        print(‘Hello World’)
        print(‘This is DAG: {}’.format(str(dag_number)))

    dag = DAG(dag_id,
              schedule_interval=schedule,
```

```python
              default_args=default_args)

    with dag:
        t1 = PythonOperator(
            task_id='hello_world',
            python_callable=hello_world_py)

    return dag


# build a dag for each number in range(10)
for n in range(1, 4):
    dag_id = 'loop_hello_world_{}'.format(str(n))

    default_args = {'owner': 'airflow',
                    'start_date': datetime(2021, 1, 1)
                    }

    schedule = '@daily'
    dag_number = n

    globals()[dag_id] = create_dag(dag_id,
                                  schedule,
                                  dag_number,
                                  default_args)
```

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

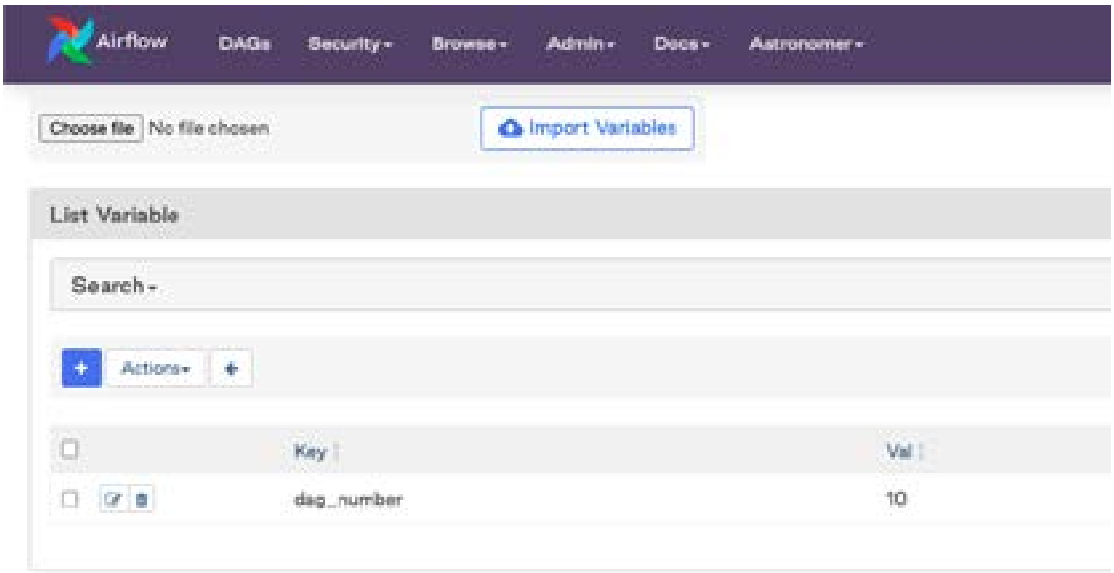


DAG	Owner
loop_hello_world_1	airflow
loop_hello_world_2	airflow
loop_hello_world_3	airflow

EXAMPLE:

Generate DAGs From Variables

As aforementioned, 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.



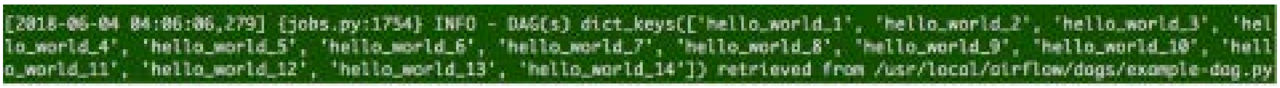
Key	Val
dag_number	10

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.

```python
from airflow import DAG
from airflow.models import Variable
from airflow.operators.python_operator import PythonOperator
from datetime import datetime


def create_dag(dag_id,
               schedule,
               dag_number,
               default_args):

    def hello_world_py(*args):
        print(‘Hello World’)
        print(‘This is DAG: {}’.format(str(dag_number)))

    dag = DAG(dag_id,
              schedule_interval=schedule,
              default_args=default_args)

    with dag:
        t1 = PythonOperator(
            task_id=‘hello_world’,
            python_callable=hello_world_py)

    return dag

```

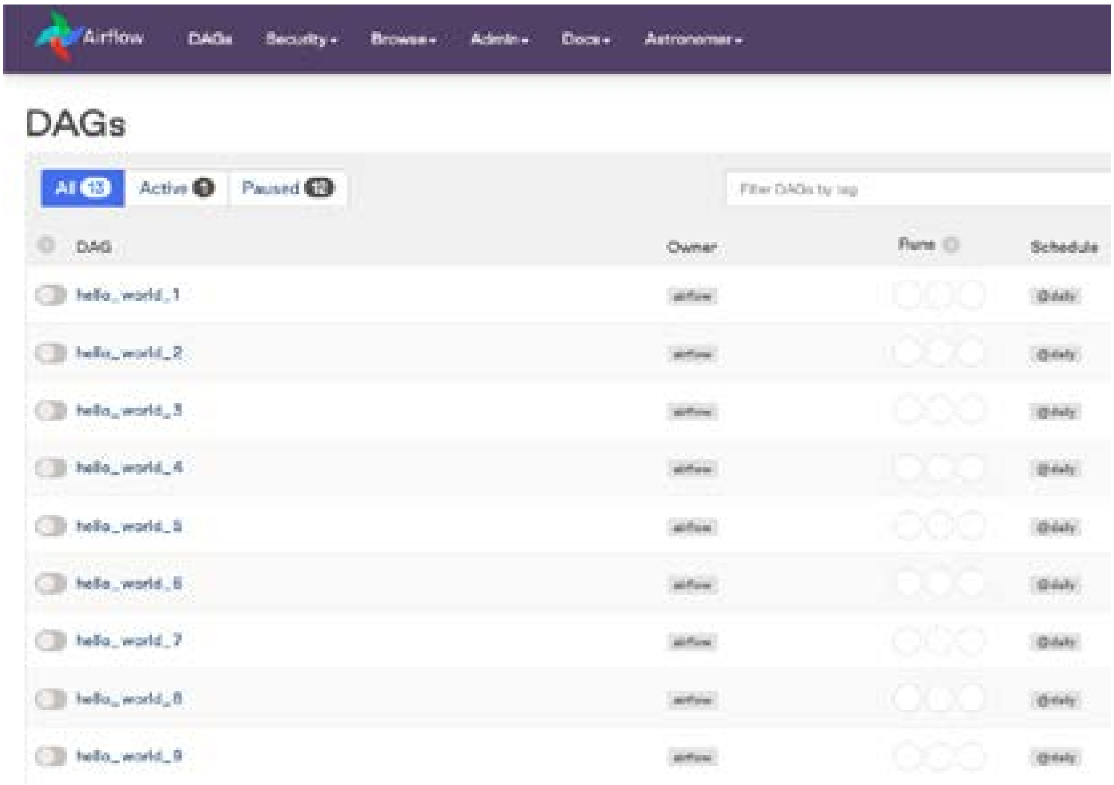
```python
number_of_dags = Variable.get('dag_number', default_var=3)
number_of_dags = int(number_of_dags)

for n in range(1, number_of_dags):
    dag_id = 'hello_world_{}'.format(str(n))

    default_args = {'owner': 'airflow',
                    'start_date': datetime(2021, 1, 1)
                    }

    schedule = '@daily'
    dag_number = n
    globals()[dag_id] = create_dag(dag_id,
                                  schedule,
                                  dag_number,
                                  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.



```
[2018-06-04 04:06:06,279] {jobs.py:1754} INFO - DAG(s) dict_keys(['hello_world_1', 'hello_world_2', 'hello_world_3', 'hel
lo_world_4', 'hello_world_5', 'hello_world_6', 'hello_world_7', 'hello_world_8', 'hello_world_9', 'hello_world_10', 'hell
o_world_11', 'hello_world_12', 'hello_world_13', 'hello_world_14']) retrieved from /usr/local/airflow/dags/example-dag.py
```

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



DAGs

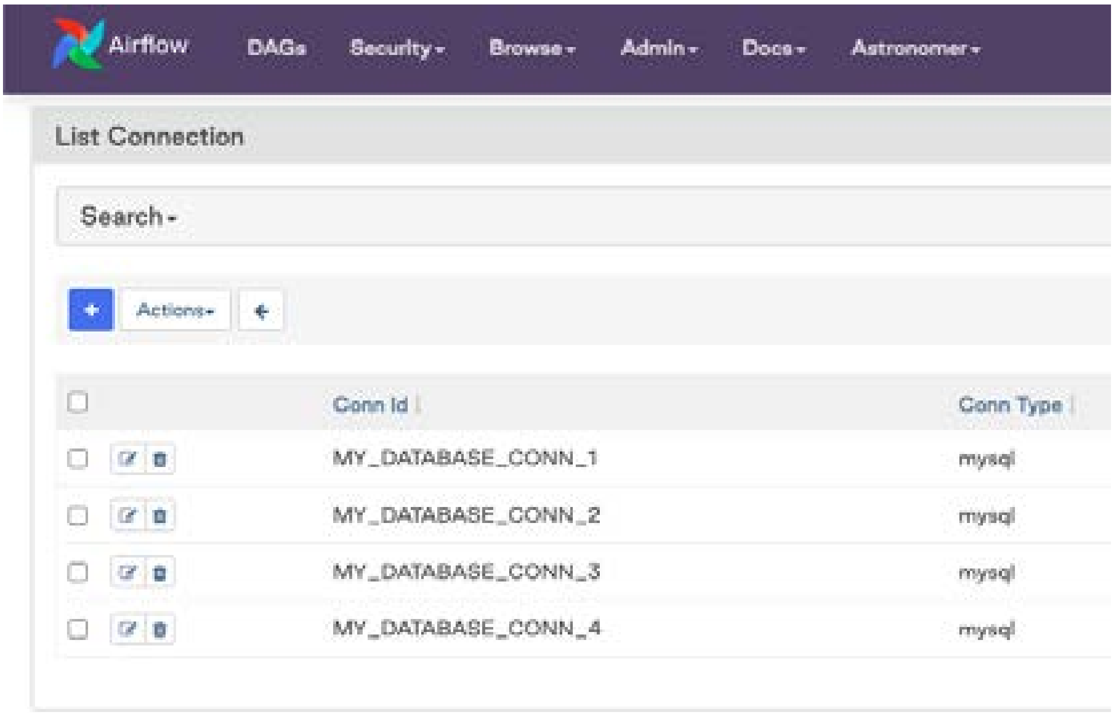
DAG	Owner	Runs	Schedule
hello_world_1	airflow		@daily
hello_world_2	airflow		@daily
hello_world_3	airflow		@daily
hello_world_4	airflow		@daily
hello_world_5	airflow		@daily
hello_world_6	airflow		@daily
hello_world_7	airflow		@daily
hello_world_8	airflow		@daily
hello_world_9	airflow		@daily

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.

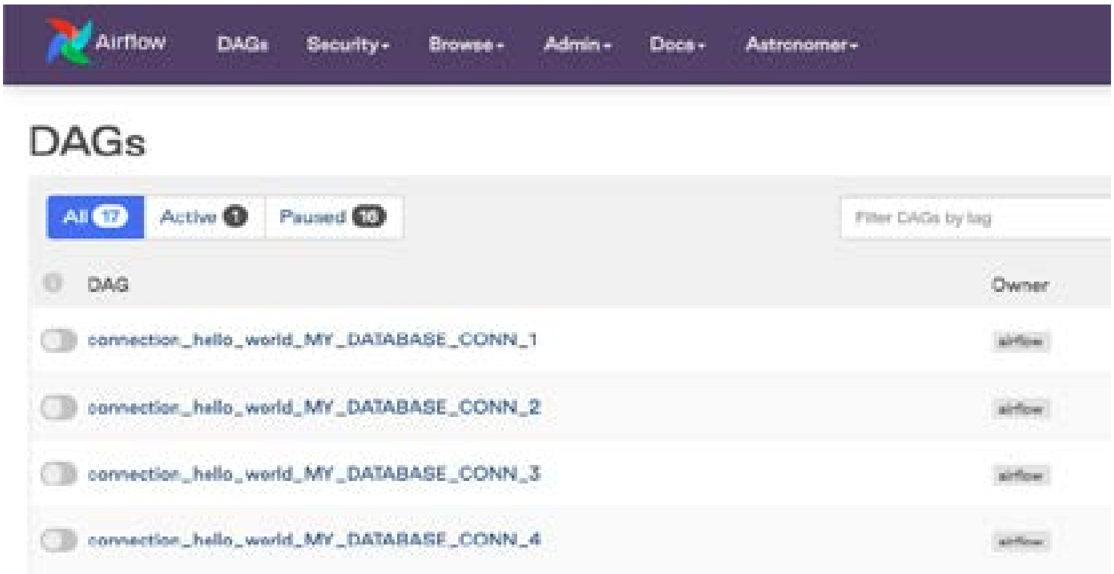
To implement this method, we can pull the connections we have in our Airflow metadata database by instantiating the "Session" and querying the "Connection" table. We can also filter this query so that it only pulls connections that match specific criteria.



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

```python
from airflow import DAG, settings
from airflow.models import Connection
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

def create_dag(dag_id,
               schedule,
               dag_number,
               default_args):

    def hello_world_py(*args):
        print('Hello World')
        print('This is DAG: {}'.format(str(dag_number)))

dag = DAG(dag_id,
              schedule_interval=schedule,
              default_args=default_args)

    with dag:
        t1 = PythonOperator(
            task_id=’hello_world’,
            python_callable=hello_world_py)

    return dag


session = settings.Session()
conns = (session.query(Connection.conn_id)
               .filter(Connection.conn_id.ilike(’%MY_DATABASE_CONN%’))
                .all())

for conn in conns:
    dag_id = ‘connection_hello_world_{}’.format(conn[0])

    default_args = {’owner’: ‘airflow’,
                    ’start_date’: datetime(2018, 1, 1)
                    }

    schedule = ‘@daily’
    dag_number = conn

    globals()[dag_id] = create_dag(dag_id,
                                  schedule,
                                  dag_number,
                                  default_args)
```

Notice that, like 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.



DAG	Owner
connection_hello_world_MY_DATABASE_CONN_1	airflow
connection_hello_world_MY_DATABASE_CONN_2	airflow
connection_hello_world_MY_DATABASE_CONN_3	airflow
connection_hello_world_MY_DATABASE_CONN_4	airflow

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`, `scheduletoreplace`, and `querytoreplace`.

```
from airflow import DAG
from airflow.operators.postgres_operator import PostgresOperator
from datetime import datetime

default_args = {‘owner’: ‘airflow’,
                ‘start_date’: datetime(2021, 1, 1)
                }

dag = DAG(dag_id,
            schedule_interval=scheduletoreplace,
            default_args=default_args,
            catchup=False)

```

```
with dag:
    t1 = PostgresOperator(
        task_id=’postgres_query’,
        postgres_conn_id=connection_id
        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.

```
{
    “DagId”: “dag_file_1”,
    “Schedule”: “’@daily’”,
    “Query”:”’SELECT * FROM table1;’”
}
```

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).

```
import json
import os
import shutil
import fileinput

config_filepath = ‘include/dag-config/’
dag_template_filename = ‘include/dag-template.py’

```

```python
for filename in os.listdir(config_filepath):
    f = open(filepath + filename)
    config = json.load(f)

    new_filename = 'dags/'+config['DagId']+'.py'
    shutil.copyfile(dag_template_filename, new_filename)


    for line in fileinput.input(new_filename, inplace=True):
        line.replace("dag_id", "'"+config['DagId']+"'")
        line.replace("scheduletoreplace", config['Schedule'])
        line.replace("querytoreplace", config['Query'])
        print(line, end="")
```

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:

```
dags/
├── dag_file_1.py
├── dag_file_2.py
include/
├── dag-template.py
├── generate-dag-files.py
└── dag-config
    ├── dag1-config.json
    └── 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:

```python
from airflow import DAG
import dagfactory

dag_factory = dagfactory.DagFactory(“/path/to/dags/config_file.yml”)

dag_factory.clean_dags(globals())
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:

```
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:

```python
import pytest
from airflow.models import DagBag

def test_no_import_errors():
  dag_bag = DagBag()
  assert len(dag_bag.import_errors) == 0, “No Import Failures”
```

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:

```
def test_retries_present():
  dag_bag = DagBag()
  for dag in dag_bag.dags:
      retries = dag_bag.dags[dag].default_args.get(‘retries’, [])
      error_msg = ‘Retries not set to 2 for DAG {id}’.format(id=dag)
      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:

```python
def test_function_returns_5():
    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:

```python
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults

class EvenNumberCheckOperator(BaseOperator):
    @apply_defaults
    def __init__(self, my_operator_param, *args, **kwargs):
        self.operator_param = my_operator_param
        super(EvenNumberCheckOperator, self).__init__(*args, **kwargs)

    def execute(self, context):
                      if self.operator_param % 2:
                                 return True
        else:
                                 return False
```

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

```
import unittest
import pytest
from datetime import datetime
from airflow import DAG
from airflow.models import TaskInstance
from airflow.operators import EvenNumberCheckOperator

DEFAULT_DATE = datetime(2021, 1, 1)

class EvenNumberCheckOperator(unittest.TestCase):

    def setUp(self):
      super().setUp()
      self.dag = DAG(‘test_dag’, default_args={‘owner’: ‘airflow’,
‘start_date’: DEFAULT_DATE})
                self.even = 10
                self.odd = 11

    def test_even(self):
      """Tests that the EvenNumberCheckOperator returns True for 10."""
```

```
        task = EvenNumberCheckOperator(my_operator_param=self.even,
task_id='even', dag=self.dag)
        ti = TaskInstance(task=task, execution_date=datetime.now())
        result = task.execute(ti.get_template_context())
        assert result is True

    def test_odd(self):
        """Tests that the EvenNumberCheckOperator returns False for 11."""
        task = EvenNumberCheckOperator(my_operator_param=self.odd,
task_id='odd', dag=self.dag)
        ti = TaskInstance(task=task, execution_date=datetime.now())
        result = task.execute(ti.get_template_context())
        assert result is False
```

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.

```
from airflow import DAG
from datetime import datetime, timedelta
from airflow.operators.email_operator import EmailOperator
from airflow.operators.python_operator import PythonOperator
from airflow.providers.microsoft.azure.hooks.azure_data_factory
import AzureDataFactoryHook
from airflow.providers.microsoft.azure.hooks.wasb import WasbHook
from great_expectations_provider.operators.great_expectations im-
port GreatExpectationsOperator

```

```
#Get yesterday’s date, in the correct format
yesterday_date = ‘{{ yesterday_ds_nodash }}’

#Define Great Expectations file paths
data_dir = ‘/usr/local/airflow/include/data/’
data_file_path = ‘/usr/local/airflow/include/data/’
ge_root_dir = ‘/usr/local/airflow/include/great_expectations’

def run_adf_pipeline(pipeline_name, date):
    ‘’’Runs an Azure Data Factory pipeline using the AzureDataFactory–
Hook and passes in a date parameter
    ’’’

    #Create a dictionary with date parameter
    params = {}
    params[“date”] = date

    #Make connection to ADF, and run pipeline with parameter
    hook = AzureDataFactoryHook(‘azure_data_factory_conn’)
    hook.run_pipeline(pipeline_name, parameters=params)

def get_azure_blob_files(blobname, output_filename):
    ‘’’Downloads file from Azure blob storage
    ’’’
    azure = WasbHook(wasb_conn_id=’azure_blob’)
    azure.get_file(output_filename, container_name=’covid–data’,
blob_name=blobname)



default_args = {
    ’owner’: ’airflow’,
    ’depends_on_past’: False,
    ’email_on_failure’: False,
    ’email_on_retry’: False,
    ’retries’: 0,
    ’retry_delay’: timedelta(minutes=5)
}

with DAG(’adf_great_expectations’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=’@daily’,
         default_args=default_args,
         catchup=False
         ) as dag:

         run_pipeline = PythonOperator(
            task_id=’run_pipeline’,
            python_callable=run_adf_pipeline,
            op_kwargs={’pipeline_name’: ’pipeline1’, ’date’: yesterday_date}
         )

         download_data = PythonOperator(
            task_id=’download_data’,
            python_callable=get_azure_blob_files,
            op_kwargs={’blobname’: ’or/’+ yesterday_date +’.csv’,
’output_filename’: data_file_path+’or_’+yesterday_date+’.csv’}
         )

```

```
        ge_check = GreatExpectationsOperator(
            task_id=’ge_checkpoint’,
            expectation_suite_name=’azure.demo’,
            batch_kwargs={
               ’path’: data_file_path+’or_’+yesterday_date+’.csv’,
                ’datasource’: ’data__dir’
            },
            data_context_root_dir=ge_root_dir
        )

        send_email = EmailOperator(
            task_id=’send_email’,
            to=’noreply@astronomer.io’,
            subject=’Covid to S3 DAG’,
            html_content=’<p>The great expectations checks passed
successfully. <p>’
        )

        run_pipeline >> download_data >> ge_check >> send_email
```

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.

6. Debugging DAGs

✦

7 Common Errors to Check when Debugging Airflow DAGs

Apache Airflow is the industry standard for workflow orchestration, handling use cases that range from machine learning model training to traditional ETL at scale. It's an incredibly flexible tool that powers mission-critical projects for startups and Fortune 500 teams alike.

However, Airflow's breadth and extensibility can make it challenging to adopt – especially for those looking for guidance beyond day-one operations. 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: Airflow 2.0 was released in December of 2020 and addresses a significant number of pain points commonly reported by users running previous versions. We strongly encourage all teams to upgrade to Airflow 2.0+ as soon as they're able.

For quick guidelines on how to run Airflow 2.0 locally, refer to **Get Started with Airflow 2.0**.

For detailed instructions on the migration process, refer to **Upgrading to Airflow 2.0+** from the Apache Airflow Project.

1. Your DAG isn't running at the time you expect it to.

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 2 pm, setting a reminder to check back in a couple of hours. You hop on at 3:30 pm to find that while your DAG did run, your logs indicate that there is only one recorded execution date for 2 pm. Huh – what happened to the 3 pm 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.

Two things:

- By design, an Airflow DAG will execute at the completion 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. Why? Airflow can't ensure that all data corresponding to the 2:00 PM interval 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**.

Note: **A proposal** to improve scheduling logic and terminology is currently being evaluated in the Airflow community.

Airflow Time Zones

Airflow stores datetime information in UTC internally and in the database. Many databases and APIs share this behavior, but it's worth clarifying.

You should not expect your DAG executions to correspond to your local timezone. 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 DAG, deployment level, or task level. When you look at corresponding logs, the `run_id` will show `manual__` instead of `scheduled__`.

3. You see a 503 Error on your Deployment.

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

If you've 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 core 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 (at Astronomer, you might hear this referenced as a `CrashLoopBackOff` state). If you push up a deploy and your Webserver for any reason takes longer than a few seconds to start, it might hit a timeout period (10 secs by default) that "crashes" it before it has time to spin up. That triggers a retry, which crashes again, 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.

Airflow 1.10 is a bit greedier than Airflow 1.9 regarding CPU (memory usage), so we've seen a recent uptick in users reporting 503s. Often, a quick

bump in resources allocated to your Webserver does the trick.
If you're using Astronomer, we generally recommend keeping the Webserver at a minimum of 5 AU (**Astronomer Units**). Even if you're not running anything particularly heavy, keeping your Webserver below 5 will more likely than not return some funky behavior all around.

Increase the Webserver Timeout period.

If upping the 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 Deployment is in fact, 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 called outside of an Operator is called every heartbeat, which can be pretty taxing.
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.

This leads us to a general best practice we've come to adopt.

Be careful when using Sensors.

If you're running Airflow 1.10.1 or earlier, **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**.

Note: Airflow v1.10.2's new sensor `mode=reschedule` feature addresses this issue. If you have more sensors than worker slots, the sensor will now get thrown into a new `up_for_reschedule` state, thus unblocking a worker slot.

If you're running Airflow 2.0+, consider enabling **Smart Sensors**. They significantly improve sensor functionality and are built to optimize resources and reduce cost.

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

Environment variables and concurrency configurations.

The potential root cause for a bottleneck and what exactly these values should be set at 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, however, setting a few **environment variables** can help improve performance. For most users, these environment variables are set in Airflow's `airflow.cfg` file. If you're running on Astronomer, you can also set these **via the Astronomer UI or your project's Dockerfile**. For all default values, **refer here**.

Parallelism

Defined as `AIRFLOW__CORE__PARALLELISM`, **Parallelism** determines how many task instances can be actively running in parallel across DAGs given the resources available at any given time at the Deployment level. 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

Defined as `ENV AIRFLOW__CORE__DAG_CONCURRENCY=`, **dag_concurrency** determines how many task instances your Scheduler can schedule at once per DAG. Think of this as "maximum tasks that can be scheduled at once, per DAG." The default is 16.

Max Active Runs per DAG

Defined as `AIRFLOW__CORE__MAX_ACTIVE_RUNS=`, **maxactiverunsperdag** determines the maximum number of active DAG runs per DAG. The default value is 16.

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 using **"pools"** instead – 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.

Suppose tasks are getting bottlenecked and your concurrency configurations are already optimized. In that case, the issue might be that your Scheduler is underpowered or that your Deployment could use another Celery Worker, assuming you're using the **Celery Executor**.

If you're running on Astronomer, we generally recommend 5 AU as the default minimum for the Scheduler and 10 AU for Celery 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 task 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 more than 5 seconds (default). Defined in seconds, this is 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 Astronomer, you can do this in the Configure tab of the Astronomer UI.
- Are you looking for a log from over 15 days ago? If you’re using Astronomer, 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 run 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.

7. Tasks are slow to schedule and/or have stopped being scheduled altogether

If your tasks are slower than usual to get scheduled, you'll want to check how often you've set your Scheduler to restart. Airflow 1.10 has an unfortunately well-known problem by which the Scheduler's performance degrades over time and requires a quick restart to ensure optimal performance.

Set automatic Scheduler restarts.

The frequency of restarts is defined in your `airflow.cfg` file as `run_duration`. A `run_duration` of -1 indicates that you never want your Scheduler to restart, whereas a `run_duration` of 3600 will restart your Scheduler every hour. Check out **this forum post** for more information. We generally recommend restarting your Scheduler once every 24 hours, but optimal frequency largely depends on your use case.

If you're running Airflow on Astronomer, you can restart your Scheduler by either:

- Inserting `AIRFLOW__SCHEDULER__RUN_DURATION={num_seconds_between_restarts}` as an Environment Variable to set a recurring restart.
- Running `astro deploy` via the Astronomer CLI to restart all Airflow components. If you're running the Celery Executor, the **Worker Termination Grace Period** can minimize task disruption.

Note: Scheduler performance was a critical part of the **Airflow 2.0 release** and has seen significant improvements since December of 2020. For more information, read **The Airflow 2.0 Scheduler**.

Ready for more?

Check out our lineup of webinars – from introductory content to super advanced tutorials, we cover the most common topics around Apache Airflow and data management!

You can expect practitioners from our team to share their insights, best practices and know-how!

Register today!

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.

```
from datetime import datetime
from airflow import DAG

default_args = {
    ’owner’: ’airflow’,
    ’start_date’: datetime(2018, 1, 30),
    ’email’: [’noreply@astronomer.io’],
    ’email_on_failure’: True
}

with DAG(’sample_dag’,
    default_args=default_args,
    schedule_interval=’@daily’,
    catchup=False) as dag:

...
```

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.

```python
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator

default_args = {
    ’owner’: ’airflow’,
    ’start_date’: datetime(2018, 1, 30),
    ’email_on_failure’: False,
    ’email’: [’noreply@astronomer.io’],
    ’retries’: 1
}

with DAG(’sample_dag’,
    default_args=default_args,
    schedule_interval=’@daily’,
    catchup=False) as dag:

    wont_email = DummyOperator(
          task_id=’wont_email’
    )

    will_email = DummyOperator(
          task_id=’will_email’,
          email_on_failure=True
    )
```

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.

```
from datetime import datetime, timedelta
from airflow import DAG

default_args = {
    ’owner’: ’airflow’,
    ’start_date’: datetime(2018, 1, 30),
    ’email’: [’noreply@astronomer.io’],
    ’email_on_failure’: True,
    ’email_on_retry’: True,
    ’retry_exponential_backoff’: True,
    ’retry_delay’ = timedelta(seconds=300)
    ’retries’: 3
}

with DAG(’sample_dag’,
    default_args=default_args,
    schedule_interval=’@daily’,
    catchup=False) as dag:

...
```

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`.

```python
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator

def custom_failure_function(context):
    “Define custom failure notification behavior”
    dag_run = context.get(‘dag_run’)
    task_instances = dag_run.get_task_instances()
    print(“These task instances failed:”, task_instances)

def custom_success_function(context):
    “Define custom success notification behavior”
    dag_run = context.get(‘dag_run’)
    task_instances = dag_run.get_task_instances()
    print(“These task instances succeeded:”, task_instances)

default_args = {
    ‘owner’: ‘airflow’,
    ‘start_date’: datetime(2018, 1, 30),
    ‘on_failure_callback’: custom_failure_function
    ‘retries’: 1
}

with DAG(‘sample_dag’,
          default_args=default_args,
          schedule_interval=‘@daily’,
          catchup=False) as dag:

    failure_task = DummyOperator(
          task_id=‘failure_task’
    )

    success_task = DummyOperator(
          task_id=‘success_task’,
          on_success_callback=custom_success_function
    )
```

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:

```
from datetime import datetime, timedelta
from airflow import DAG

default_args = {
    ‘owner’: ‘airflow’,
    ‘start_date’: datetime(2018, 1, 30),
    ‘email’: [‘noreply@astronomer.io’],
    ‘email_on_failure’: True,
    ‘email_on_retry’: True,
    ‘retry_delay’ = timedelta(seconds=300)
    ‘retries’: 1
}

with DAG(‘sample_dag’,
    default_args=default_args,
    schedule_interval=‘@daily’,
    catchup=False) as dag:

...
```

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:

```
[smtp]
# If you want airflow to send emails on retries, failure, and you want to use
# the airflow.utils.email.send_email_smtp function, you have to configure an
# smtp server here
smtp_host = your-smtp-host.com
smtp_starttls = True
smtp_ssl = False
# Uncomment and set the user/pass settings if you want to use SMTP AUTH
# smtp_user =
# smtp_password =
smtp_port = 587
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:

```
default_subject = ‘Airflow alert: {{ti}}’
# For reporting purposes, we report based on 1-indexed,
# not 0-indexed lists (i.e. Try 1 instead of
# Try 0 for the first attempt).
default_html_content = (
    ‘Try {{try_number}} out of {{max_tries + 1}}<br>’
    ‘Exception:<br>{{exception_html}}<br>’
    ‘Log: <a href="{{ti.log_url}}">Link</a><br>’
    ‘Host: {{ti.hostname}}<br>’
    ‘Mark success: <a href=”{{ti.mark_success_url}}”>Link</a><br>’
)
```

To see the full method, check out the source code **here**.

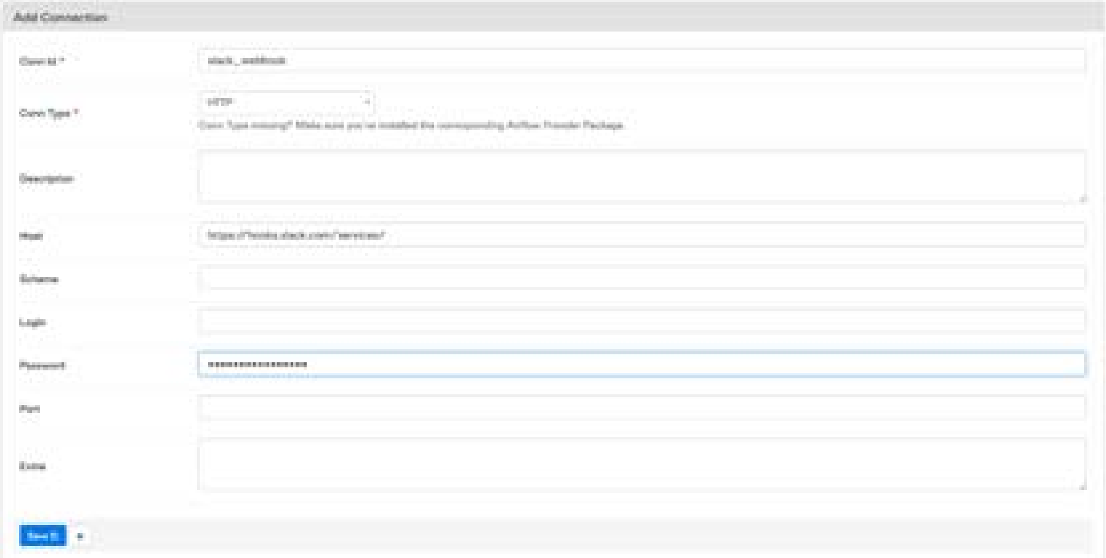
You can overwrite this default with your custom content by setting the `subject_template` and/or `html_content_template` variables in your `airflow.cfg` with the path to your jinja template files for subject and content respectively.

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/XXXXXXXXXXXXXXXXXXXXXXXX`).



Add Connection

Conn Id *	slack_webhook
Conn Type *	HTTP Conn Type missing? Make sure you've installed the corresponding Airflow Provider Package.
Description	
Host	https://hooks.slack.com/services/
Schema	
Login	
Password	••••••••••••••••
Port	
Extra	

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:

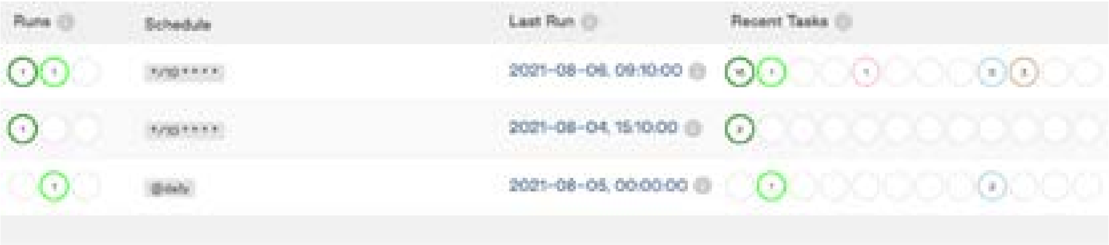
```
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator

def slack_notification(context):
    slack_msg = ”””
            :red_circle: Task Failed.
            *Task*: {task}
            *Dag*: {dag}
            *Execution Time*: {exec_date}
            *Log Url*: {log_url}
            ”””.format(
            task=context.get(’task_instance’).task_id,
            dag=context.get(’task_instance’).dag_id,
            ti=context.get(’task_instance’),
            exec_date=context.get(’execution_date’),
            log_url=context.get(’task_instance’).log_url,
        )
    failed_alert = SlackWebhookOperator(
        task_id=’slack_notification’,
        http_conn_id=’slack_webhook’,
        message=slack_msg)
    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:



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

DAG Runs and tasks can have the following states:

- **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:

```
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
import time

def my_custom_function(ts,**kwargs):
    print(“task is sleeping”)
    time.sleep(40)

# Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: True,
    ‘email’: ‘noreply@astronomer.io’,
    ‘email_on_retry’: False,
    ‘sla’: timedelta(seconds=30)
}

# Using a DAG context manager, you don’t have to specify the dag
property of each task
with DAG(‘sla-dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=timedelta(minutes=2),
         default_args=default_args,
         catchup=False
         ) as dag:

```

```
    t0 = DummyOperator(
        task_id='start'
    )

    t1 = DummyOperator(
        task_id='end'
    )

    sla_task = PythonOperator(
        task_id='sla_task',
        python_callable=my_custom_function
    )
t0 >> sla_task >> t1
```

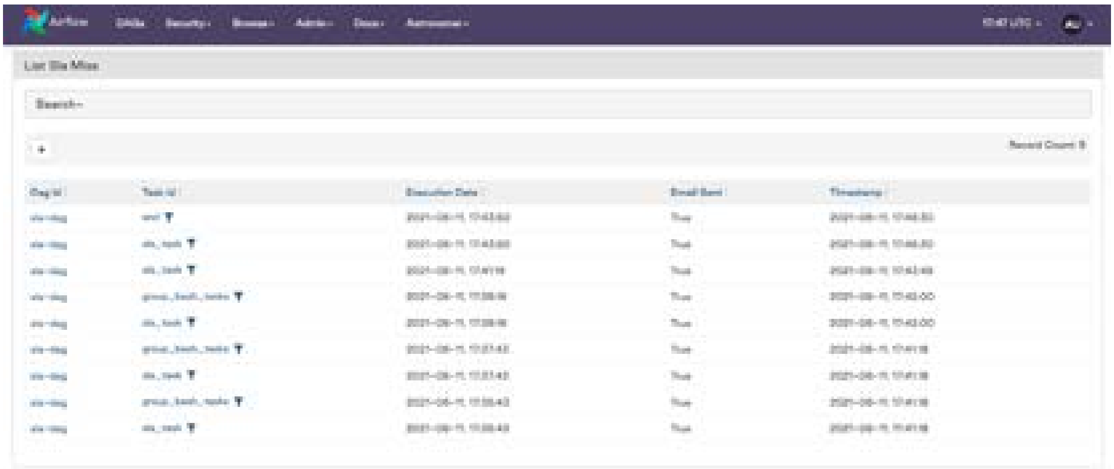
SLAs have some unique behaviors that you should consider before implementing them:

- 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 to execute.

```python
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta
import time

def my_custom_function(ts,**kwargs):
    print(“task is sleeping”)
    time.sleep(40)

# Default settings applied to all tasks
default_args = {
    ‘owner’: ‘airflow’,
    ‘depends_on_past’: False,
    ‘email_on_failure’: True,
    ‘email’: ‘noreply@astronomer.io’,
    ‘email_on_retry’: False
}

# Using a DAG context manager, you don’t have to specify the dag
property of each task
with DAG(‘sla-dag’,
         start_date=datetime(2021, 1, 1),
         max_active_runs=1,
         schedule_interval=timedelta(minutes=2),
         default_args=default_args,
         catchup=False
         ) as dag:

```

```
    t0 = DummyOperator(
        task_id='start',
        sla=timedelta(seconds=50)
    )

    t1 = DummyOperator(
        task_id='end',
        sla=timedelta(seconds=500)
    )

    sla_task = PythonOperator(
        task_id='sla_task',
        python_callable=my_custom_function,
        sla=timedelta(seconds=5)
    )

    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:



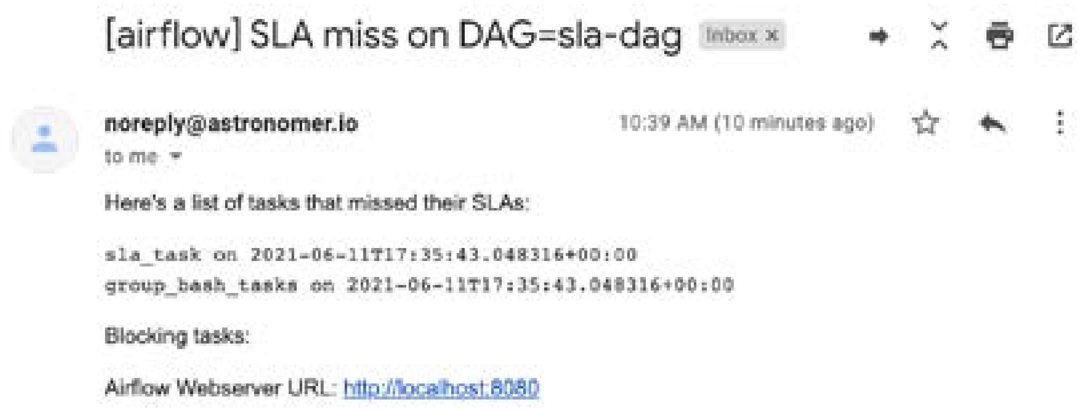
List Sla Miss

Search

Record Count: 9

Dag Id	Task Id	Execution Date	Email Sent	Timestamp
sla-dag	end	2021-06-11, 17:43:50	True	2021-06-11, 17:46:50
sla-dag	sla_task	2021-06-11, 17:43:00	True	2021-06-11, 17:46:50
sla-dag	sla_task	2021-06-11, 17:41:18	True	2021-06-11, 17:42:49
sla-dag	group_bash_tasks	2021-06-11, 17:28:18	True	2021-06-11, 17:42:00
sla-dag	sla_task	2021-06-11, 17:28:18	True	2021-06-11, 17:42:00
sla-dag	group_bash_tasks	2021-06-11, 17:27:43	True	2021-06-11, 17:41:18
sla-dag	sla_task	2021-06-11, 17:27:43	True	2021-06-11, 17:41:18
sla-dag	group_bash_tasks	2021-06-11, 17:20:43	True	2021-06-11, 17:41:18
sla-dag	sla_task	2021-06-11, 17:20:43	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.



[airflow] SLA miss on DAG=sla-dag

noreply@astronomer.io

10:39 AM (10 minutes ago)

to me

Here's a list of tasks that missed their SLAs:

```
sla_task on 2021-06-11T17:35:43.048316+00:00
group_bash_tasks on 2021-06-11T17:35:43.048316+00:00
```

Blocking tasks:

Airflow Webserver URL: http://localhost:8080

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.

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

7. Airflow in Business

How Herman Miller Eliminated Data Silos With a Centralized Data Strategy

Astronomer product values:

- **DataOps productivity**

  eliminating data silos through data centralization and automation.

- **Business impact**

  helping the client to deliver the right data, to the right team, at the right time.

- **Day-2 operations**

  adding scalability, diagnostics, automation, and observability.

- **Continuous innovation**

  allowing the client to stay on top of Apache Airflow advancements.

Herman Miller is a global company that places great importance on design, the environment, community service, and the health and well-being of its customers and employees. Herman Miller puts furniture in a bigger context of social well-being and sustainable living. Its mission is to "leverage the power of design to improve people's lives".

Data & Analytics is a new organization at Herman Miller, responsible for managing the data needed to drive analytics and insights in the company. In March of 2020 they kicked off planning sessions focused on developing a vision and roadmap for how they want to manage data in the organization. The development of a new modern Data Platform was key to their success. Using tools like Apache Airflow, Astronomer and Snowflake as part of the data platform they intended to provide value to the business and its employees. They have been continuing to build the data environment prioritizing areas of the business where they can have the largest impact.

“My favorite thing about data and analytics is that it allows you to have a direct impact on the business. I love looking at data and making informed decisions that can be tied to any action and once implemented leads to change. It also allows us to help people in the organization and improve and streamline the way they work— from very manual and tedious to automated and scalable,”

Mark Gergess
VP of Data & Analytics at Herman Miller

Challenge

Herman Miller was facing two main issues:

1. Disparate data sources

with core information about their customer, products and sales spread across multiple systems, bringing the data together and normalizing it to perform analysis was a very manual and difficult task for team members. Employees would have to run multiple reports, download them to Excel, and then figure out the business logic across the data sets to deliver a global report. Because of the differences in product, time zones, standards across the globe, figuring out the common language for that data involved many meetings with international teams and took a lot of time.

2. Data Quality, Alerting & Monitoring

the client's existing data environments didn't allow them to be proactive in communicating and resolving data issues. Most of the time the business side of Herman Miller would notify the Data & Analytics team of missing or bad data, or when they failed to receive an automated report. Investigating, finding and fixing these issues would take hours.

Why Airflow

Before Airflow, **Herman Miller** had been using a few similar ETL tools. However, they had an issue with problem detection. Investigation into problems would take half a day, as the pipelines are written in a complex way.

> *"We need to know about the problem as early as possible. We would know there was an issue, but to pinpoint it would require a few-hour investigation," states Mark.*

They were looking for a solution that would offer data accuracy, strong alerting and monitoring, as well as easy developing and deploying of CI/CD. In the development of their data platform, they've partnered with Raybeam — a company specializing in data and analytics. Raybeam, being a big fan of Astronomer and Airflow, recommended the tool to Herman Miller.

> *"When it comes to our data platform, Herman Miller wants to be the best. We try to have a strong relationship with organizations we are cooperating with, like Raybeam, Astronomer, and Snowflake," says Mark, "It's more than just using a tool, it's about developing a true business partnership."*

After comparing Airflow with other tools it turned out that Airflow ticked all the boxes from data quality and a data monitoring perspective. The decision was a no–brainer.

Snowflake partnership

Before Mark joined the team, Herman Miller had already been using—and loving—**Snowflake**. The data warehousing tool fits all of their needs perfectly: speed, zero to no maintenance required, capabilities to support data privacy etc.

Snowflake Data Marketplace is another feature Herman Miller is excited about. It allows them to easily access external **datasets** and use them to enrich Herman Miller data to provide more informative insights. For example, looking at the Starbucks foot traffic in large cities in North America can inform them on the return-to-work behavior in certain markets. They can use that as an input to understand how it correlates to their B2B and B2C business performance and inform their partners of what is happening in a proactive manner.

> *"We are at the very beginning of our journey of data and analytics but to be kicking things off with these strong partners, I think it's really helping us have a very solid foundation," says Mark*

Solution

The design of corporate systems is often a copy of the communication structure within a company. Herman Miller had to break that pattern to develop a data platform that would champion the "One Herman Miller" strategy. In partnership with **Raybeam**, Snowflake and Astronomer, Herman Miller brought together data, governance rules and business logic that had previously been scattered throughout different parts of the organization.

Work that used to take months can now be done in minutes. Analysts can quickly pull up a global view of product sales across brands and locations. Marketing data can be stitched together across partners to develop a true, customer-first approach to communication. The editorial team is no longer held back by the data science team.

While ease of use is the most visible **benefit of this centralized platform**, the underlying process changes are what will continue to drive benefits far into the future. DataOps at Herman Miller means deploying high-quality code quickly and monitoring that code closely. The process is more similar to agile software development than it is to older DBA-style workflows. The move away from IT-driven processes to more modern DataOps patterns ensures that the data team can adapt quickly while keeping up the quick pace of development required by a blue chip company like Herman Miller.

Weekly stats then and now

September 2020

- 2–3 sources
- Multiple weeks to months to add a new data source
- Unknown number to fixes
- **No ticketing system**
- **No release system**
- **No alerting system**

Today

- 18 sources and counting
- Hours to days to add a new data
- **19 tickets closed**
- **4 production releases**
- **1 production alert ⁄ not downstream impact**

Finally, the more team-oriented approach eliminates silos of information. Historically, each data source or transformation would be understood mainly by the person who built it. The current process ensures more than one person will see, review and collaborate on code before it makes it into production. No more useless "knowledge transfer" sessions when someone leaves for a holiday. The team is built to operate truly as a team.

> *"We are not focusing on the technology anymore—Airflow has allowed us to focus on using the data and making the most of it," explains Mark. "The platform we've created in partnership with Raybeam, Snowflake, and Astronomer is best-in-class. I would definitely recommend using Airflow—all the best tech companies in the market do."*

Why Astronomer

Before Astronomer, Mark's team would have to wait up to 3 weeks to get the data they needed from the data engineering team. Now, with Apache Airflow and Astronomer, they can achieve a lot in a short amount of time and the platform allows them to make sure they are bringing high–quality data.

> *"I love how my data science team has become self-sufficient and effective. Airflow made it very easy for them to get the data they need and manage it in a way that allows them to do their job quickly and efficiently" — Mark.*

As this is their first data platform, there's still a lot of learning involved. Luckily, Apache Airflow allows them to quickly adjust to changes.

> *"Without Astronomer, there would have been a lot more tweaking and adjusting to achieve the result we want. Our work as data scientists would have been a lot slower and more difficult," says Mark.*
>
> *"Astronomer has ticked a lot of boxes for us," Mark continues, "it's easy to use, plus there's no silos of information with other platforms and tools. I don't think there's any company I wouldn't recommend Astronomer to."*

The Data & Analytics team has ambitious plans for the future: they want the whole company to be introduced to the modern way of managing data—making sure they are building governance rules and business processes in their workflows that are beneficial to them. The goal is to give Herman Miller business teams more responsibility and ownership by keeping them connected to the data.

ASTRONOMER

Thank you!

Hope you had a great time reading our DAG guide. We would love to hear your feedback, so make sure to follow us on Twitter and Linkedin!

Start building your next-generation data platform today.

Get Astronomer

Experts behind:

Marc Lamberti | Head of Customer Training at Astronomer
Kenten Danas | Field Engineer at Astronomer
Jake Witz | Technical Writer at Astronomer