"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 guide, we'll 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 guide. Specifically, it is a quality: A program is considered idempotent if, for a set input, running the program once has the same effect as running the program multiple times.

In the context of Airflow, a DAG is considered idempotent if rerunning the same *DAG Run* (with the same inputs) multiple times has the same effect as running it only once. This can be achieved by designing each individual task in your DAG to be idempotent. Designing idempotent DAGs and tasks 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 re-run independently of the others. Said another way, in an atomized a task, a success in 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

By using templated fields in Airflow, you can pull values into DAGs using environment variables and jinja templating. Compared to using Python functions, using templated 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" 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 datetime 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 off of a sequence ID can be a sound way to filter pipeline records without a last modified date.

Avoid Top-Level Code in Your DAG File

In the context of Airflow, we use "top-level code" to mean any code that isn't part of your DAG or operator instantiations, particularly code making requests to external systems.

Airflow executes all code in the `dags_folder` on every `min_file_process_interval`, which defaults to 30 seconds (you can read more about this parameter in the Airflow docs). Because of this, top-level code that makes requests to external systems, like an API or a database, or makes function calls outside of your tasks can cause performance issues since these requests and connections are being made every 30 seconds rather than only when the DAG is scheduled to run.

An example that goes against this best practice is the DAG below, which dynamically generates `PostgresOperator` tasks based on records pulled from a database (i.e. the `hook` and `result` variables which are written outside of an operator instantiation):

```
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_1',
         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 use the `hook` and `result` variables to 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 leverage dynamic task mapping to have a task that gets the required information from the `grocery_list` table and dynamically maps downstream tasks based on the result. Note that dynamic task mapping is available as of Airflow 2.3.

Treat Your DAG File Like a Config File

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. When possible, 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, consider keeping 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 SQL query is provided directly in the `PostgresOperator` `sql` param.

```
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
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_2',
         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 SQL query directly 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 AS a
                JOIN yesterday_covid_data AS b
                ON a.state = b.state
            )
            SELECT a.state, b.state, c.two_day_avg
            FROM yesterday_covid_data AS a
            JOIN today_covid_data AS b
            ON a.state=b.state
            JOIN two_day_rolling_avg AS 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 set the `template_searchpath` in the DAG instantiation and then 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 PostgresOperator
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 + "'"}
        )
```

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]
```

You might also like:

- Dynamic DAGs Webinar
- Best Practices for Writing DAGs in Airflow 2 Webinar
- 7 Common Errors to Check when Debugging Airflow DAGs
- Dynamically Generating DAGs in Airflow

Leverage Airflow Features

The next category of best practices relates to getting the most out of Airflow by leveraging built-in features and the broader Airflow ecosystem, namely provider packages for third-party integrations, to fulfill specific use cases. Using Airflow in this way makes it easier to scale and pull in the right tools based on your needs.

Make Use of Provider Packages

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

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

For easy discovery of all the great provider packages out there, check out the Astronomer Registry.

Decide Where to Run Data Processing Jobs

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

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

recommend the following:

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

Use Intermediary Data Storage

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

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

Use an ELT Framework

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

Other Best Practices

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

Use a Consistent File Structure

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

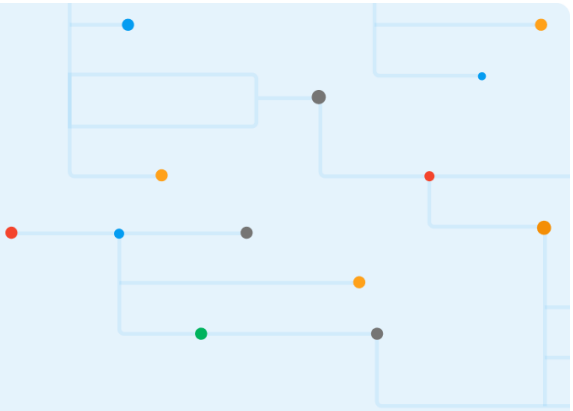
```
├── dags/ # Where your DAGs go
│   ├── example-dag.py # An example dag that comes with the initialized project
├── Dockerfile # For Astronomer's Docker image and runtime overrides
├── include/ # For any other files you'd like to include
├── plugins/ # For any custom or community Airflow plugins
├──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

Additional Resources

<iframe src="https://fast.wistia.net/embed/iframe/hfrzvkb3lk" title="branchpythonoperator Video" allow="autoplay; fullscreen" allowtransparency="true" frameborder="0" scrolling="no" class="wistia_embed" name="wistia_embed" allowfullscreen msallowfullscreen width="100%" height="100%" style="aspect-ratio:16/9"></iframe>
Are you looking for more ways of creating better and more reliable data pipelines? What you should know and do to sleep easily while your tasks run? Well, you've come at the right place! In this course you will learn more about:

- The best practices around DAG authoring
- How to make better tasks
- What configuration settings make a difference

Find out on the Astronomer's Best Practice Course for FREE today!

See you there!