DATA ENGINEERING 101

Apache Airflow Tips and Best Practices

The intermediate guide to building reliable data pipelines with Airflow.

Xinran Waibel Dec 9, 2019 · 6 min read ★

Photo by JJ Ying on Unsplash

When I first started building ETL pipelines with Airflow, I had so many memorable "aha" moments after figuring out why my pipelines didn't run. As the tech

I will share many tips and best practices for Airflow along with behind-the-scenes mechanisms to help you build more reliable and scalable data pipelines.

(New to Airflow? Check out the beginner's guide to Airflow first.)

(Interested in ways to efficiently learn a tech stack? Check out the Systematic Learning Method.)

DAG Schedule

In Airflow, a DAG is triggered by the Airflow scheduler periodically based on the `start_date` and `schedule_interval` parameters specified in the DAG file. It is very common for beginners to get confused by Airflow's job scheduling mechanism because it is unintuitive at first that **the Airflow scheduler triggers a DAG run at the end of its schedule period, rather than at the beginning of it.**

When a new DAG is created and picked up by Airflow, the Airflow scheduler materializes many DAG run entries along with corresponding schedule periods based on `start_date` and `schedule_interval` of the DAG, and each DAG run is triggered when its time dependency is met. For example, consider this sample DAG that runs daily at 7 am UTC:

```
default_args = {
  'owner': 'xinran.waibel',
  'start_date': datetime(2019, 12, 5),
}

dag = DAG('sample_dag', default_args=default_args,
schedule_interval='0 7 * * *')
```

This DAG will have the following DAG runs created by the Airflow scheduler:

DAG Run ID	Schedule Period (UTC)	Execution Time (UTC)	Trigger At (UTC)
scheduled__2019-12-05T07:00:00+00:00	2019-12-05 07:00:00 -- 2019-12-06 07:00:00	2019-12-05 7:00:00	2019-12-06 7:00:00
scheduled__2019-12-06T07:00:00+00:00	2019-12-06 07:00:00 -- 2019-12-07 07:00:00	2019-12-06 7:00:00	2019-12-07 7:00:00
scheduled__2019-12-07T07:00:00+00:00	2019-12-07 07:00:00 -- 2019-12-08 07:00:00	2019-12-07 7:00:00	2019-12-08 7:00:00
scheduled__2019-12-08T07:00:00+00:00	2019-12-08 07:00:00 -- 2019-12-09 07:00:00	2019-12-08 7:00:00	2019-12-09 7:00:00
scheduled__2019-12-09T07:00:00+00:00	2019-12-09 07:00:00 -- 2019-12-10 07:00:00	2019-12-09 7:00:00	2019-12-10 7:00:00

The first DAG run would be triggered after 7 am on 2019–12–06, at the end of its schedule period, instead of on the start date. Similarly, the rest of DAG runs would be executed every day at 7 am after that.

The *execution time* in Airflow is not the actual run time, but rather the start timestamp of its schedule period. For example, the execution time of the first DAG run is 2019–12–05 7:00:00, though it is executed on 2019–12–06. However, if a DAG run is manually started by users, the execution time of this manual DAG run would be exactly when it was triggered. (To tell whether a DAG run is scheduled or manually triggered, you can look at the prefix of its DAG run ID: *scheduled__* or *manual__*).

Based on Airflow's scheduling mechanism described above, you should always use a static `start_date` for your DAGs to make sure DAG runs are populated as expected. Keep in mind the `start_date` is *not* necessarily when the first DAG run would be triggered.

Catchup and Idempotent DAG

Photo by Matt Popovich on Unsplash

whose time dependency has been met if catchup is enabled. If catchup in turned off, then only the latest DAG run will be executed and those before it will not even show up in the DAG history. For example, assuming the sample DAG is picked up by Airflow at 8 am on 2019–12–08, three DAG runs will run if catch up is enabled. However, if catchup is turned off, only `scheduled__2019-12-07T07:00:00+00:00` will be triggered.

There are 2 ways to configure the catchup setting in Airflow:

- Airflow cluster level: Set `catchup_by_default = True` (by default) or `False` under the `scheduler` section in the Airflow configuration file *airflow.cfg*. This setting is applied to all DAGs unless a DAG-level catchup setting is specified.
- DAG level: Set `dag.catchup = True` or `False` in the DAG file:

```
dag = DAG('sample_dag', catchup=False, default_args=default_args)
```

Because Airflow can backfill past DAG runs when catchup is enabled and each DAG run can be re-run manually at any time, **it is important to make sure DAGs are idempotent and each DAG run is independent of each other and the actual run date.** DAG idempotence means the result of running the same DAG run multiple times should be the same as the result of running it once.

Now let's consider this primitive DAG that runs a Python function every day to retrieve daily marketing ads' performance data from an API and load the data to a database:

```
with DAG('ads_api', catchup=True, default_args=default_args, schedule_interval='@daily') as dag

  def export_api_data(api_token):

    # Call API to extract yesterday's data from API
    yesterday = date.today() - timedelta(days=1)
    data = download_api_data(api_token, date = yesterday)

    # Load data to database
    insert_to_database(data)
```

```
    op_kwargs={'api_token': secret_api_token}
  )
```

bad_airflow_python_task.py hosted with ♡ by GitHub view raw

The `export_api_data` Python function uses `datetime` library to dynamically get yesterday's date, download yesterday's ads performance data from API, and then insert downloaded data into the destination database. However, this DAG will have correct results if no past DAG runs are backfilled and all DAG runs are executed exactly once. This is because there are two big issues in the DAG design:

1. If `start_date` is set to 2019–12–01 and the DAG is uploaded to Airflow bucket on 2019–12–08, then seven past DAG runs would run on 2019–12–08. Since yesterday's date is obtained dynamically in `export_api_data` function, all the backfilled DAG runs will have `yesterday = 2019-12-07` and therefore download and upload the same day's data into the database.
2. When a DAG run is executed more than once, multiple copies of the same day's ads data will be inserted into the database, causing unwanted duplicates in the database.

We can make a few improvements to the DAG file to solve these issues:

1. Instead of using `datetime` library, now we use `{{ ds }}` , one of Airflow's built-in template variables, to get the execution date of the DAG run, which is independent of its actual DAG run date.
2. Before the previous day's ads data is inserted to the database, delete the corresponding partition in the database, if any, to avoid duplicates.

```
with DAG('ads_api', catchup=True, default_args=default_args, schedule_interval='@daily') as dag

  def export_api_data(api_token, yesterday):

    # Call API to extract yesterday's data from API
    data = download_api_data(api_token, date = yesterday)

    # IMPROVEMENT: Delete yesterday's partition from database
    delete_partition_database(yesterday)
```

```
13
14    py_task = PythonOperator(
15      task_id='load_yesterday_data',
16      python_callable=export_api_data,
17      op_kwargs={
18        'api_token': secret_api_token,
19        'yesterday': '{{ ds }}'  # IMPROVEMENT
20      }
21    )
```

improved_airflow_python_task.py hosted with ♡ by GitHub view raw

Airflow Metadata

Airflow is powered by two key components:

- **Metadata database:** maintains information on DAG and task states.
- **Scheduler**: processes DAG files and utilizes information stored in the metadata database to decide when tasks should be executed.

The scheduler will scan and compile all qualified DAG files in the Airflow bucket every a couple of seconds to detect DAG changes and check whether a task can be triggered. **It is critical to keep DAG files very light (like a configuration file) so that it takes less time and resources for the Airflow scheduler to process them at each heartbeat**. No actual data processing should happen in DAG files.

Changing the DAG ID of an existing DAG is equivalent to creating a brand new DAG since Airflow will actually add a new entry in the metadata database without deleting the old one. This might cause extra trouble because you will lose all the DAG run history and Airflow will attempt to backfill all the past DAG runs again if catchup is turned on. Do not rename DAGs unless it is totally necessary.

Deleting a DAG file from the Airflow bucket does not erase its DAG run history and other metadata. You need to use either the Delete button in Airflow UI or `airflow delete_dag` to explicitly delete the metadata. If you upload the same DAG again after all previous metadata is deleted, it will be treated as a brand new DAG again (which comes very handy if you want to rerun all the past DAG runs at once).

(**TL;DR**) Here is a summary of the main takeaways:

- The `start_date` is *not* necessarily when the first DAG run would be executed, as a DAG run is triggered at the end of its schedule period.
- Always use a static `start_date` for your DAGs to make sure DAG runs are populated as expected.
- Utilize Airflow's template variable and macros to ensure your DAG runs are independent of each other and actual run time.
- Make sure your DAGs are idempotent to ensure running the same DAG run multiple times is the same as the result of running it once.
- Keep DAG files light and quick-to-process like a configuration file, as the Airflow scheduler processes all DAG files at each heartbeat.
- Renaming an existing DAG will introduce a brand new DAG.
- In order to completely erase a DAG, you need to remove DAG files from the Airflow bucket and explicitly delete DAG metadata.

Want to learn more about Data Engineering? Check out my Data Engineering 101 column on Towards Data Science:

Data Engineering 101 - Towards Data Science

Read writing about Data Engineering 101 in Towards Data Science. A Medium publication sharing concepts, ideas, and...

towardsdatascience.com