Best Practices

Creating a new DAG is a two-step process:

- writing Python code to create a DAG object,
- testing if the code meets our expectations

This tutorial will introduce you to the best practices for these two steps.

Writing a DAG

Creating a new DAG in Airflow is quite simple. However, there are many things that you need to take care of to ensure the DAG run or failure does not produce unexpected results.

Creating a task

You should treat tasks in Airflow equivalent to transactions in a database. This implies that you should never produce incomplete results from your tasks. An example is not to produce incomplete data in `HDFS` or `S3` at the end of a task.

Airflow can retry a task if it fails. Thus, the tasks should produce the same outcome on every re-run. Some of the ways you can avoid producing a different result -

- Do not use INSERT during a task re-run, an INSERT statement might lead to duplicate rows in your database. Replace it with UPSERT.
- Read and write in a specific partition. Never read the latest available data in a task. Someone may update the input data between re-runs, which results in different outputs. A better way is to read the input data from a specific partition. You can use `execution_date` as a partition. You should follow this partitioning method while writing data in S3/HDFS, as well.
- The python datetime `now()` function gives the current datetime object. This function should never be used inside a task, especially to do the critical computation, as it leads to different outcomes on each run. It's fine to use it, for example, to generate a temporary log.

Tip

You should define repetitive parameters such as `connection_id` or S3 paths in `default_args` rather than declaring them for each task. The `default_args` help to avoid mistakes such as typographical errors.

Deleting a task

Never delete a task from a DAG. In case of deletion, the historical information of the task disappears from the Airflow UI. It is advised to create a new DAG in case the tasks need to be deleted.

Communication

Airflow executes tasks of a DAG on different servers in case you are using Kubernetes executor or Celery executor. Therefore, you should not store any file or config in the local filesystem as the next task is likely to run on a different server without access to it — for example, a task that downloads the data file that the next task processes. In the case of `Local executor`, storing a file on disk can make retries harder e.g., your task requires a config file that is deleted by another task in DAG.

If possible, use `XCom` to communicate small messages between tasks and a good way of passing larger data between tasks is to use a remote storage such as S3/HDFS. For example, if we have a task that stores processed data in S3 that task can push the S3 path for the output data in `Xcom`, and the downstream tasks can pull the path from XCom and use it to read the data.

The tasks should also not store any authentication parameters such as passwords or token inside them. Where at all possible, use Connections to store data securely in Airflow backend and retrieve them using a unique connection id.

Variables

You should avoid usage of Variables outside an operator's `execute()` method or Jinja templates if possible, as Variables create a connection to metadata DB of Airflow to fetch the value, which can slow down parsing and place extra load on the DB.

Airflow parses all the DAGs in the background at a specific period. The default period is set using `processor_poll_interval` config, which is by default 1 second. During parsing, Airflow creates a new connection to the metadata DB for each DAG. This can result in a lot of open connections.

The best way of using variables is via a Jinja template, which will delay reading the value until the task execution. The template syntax to do this is:

```
{{ var.value.<variable_name> }}
```

or if you need to deserialize a json object from the variable :

```
{{ var.json.<variable_name> }}
```

Note

In general, you should not write any code outside the tasks. The code outside the tasks runs every time Airflow parses the DAG, which happens every second by default.

Testing a DAG

Airflow users should treat DAGs as production level code, and DAGs should have various associated tests to ensure that they produce expected results. You can write a wide variety of tests for a DAG. Let's take a look at some of them.

DAG Loader Test

This test should ensure that your DAG does not contain a piece of code that raises error while loading. No additional code needs to be written by the user to run this test.

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

Running the above command without any error ensures your DAG does not contain any uninstalled dependency, syntax errors, etc.

You can look into Testing a DAG for details on how to test individual operators.

Unit tests

Unit tests ensure that there is no incorrect code in your DAG. You can write unit tests for both your tasks and your DAG.

Unit test for loading a DAG:

```
from airflow.models import DagBag
import unittest

class TestHelloWorldDAG(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.dagbag = DagBag()

    def test_dag_loaded(self):
        dag = self.dagbag.get_dag(dag_id='hello_world')
        assert self.dagbag.import_errors == {}
        assert dag is not None
        assert len(dag.tasks) == 1
```

Unit test a DAG structure: This is an example test want to verify the structure of a code-generated DAG against a dict object

```
import unittest
class testClass(unittest.TestCase):
    def assertDagDictEqual(self,source,dag):
        assert dag.task_dict.keys() == source.keys()
        for task_id, downstream_list in source.items():
            assert dag.has_task(task_id)
            task = dag.get_task(task_id)
            assert task.downstream_task_ids == set(downstream_list)
    def test_dag(self):
        self.assertDagDictEqual({
          "DummyInstruction_0": ["DummyInstruction_1"],
          "DummyInstruction_1": ["DummyInstruction_2"],
          "DummyInstruction_2": ["DummyInstruction_3"],
          "DummyInstruction_3": []
        },dag)
```

Unit test for custom operator:

```
import unittest
from airflow.utils.state import State

DEFAULT_DATE = '2019-10-03'
TEST_DAG_ID = 'test_my_custom_operator'

class MyCustomOperatorTest(unittest.TestCase):
   def setUp(self):
       self.dag = DAG(TEST_DAG_ID, schedule_interval='@daily', default_args={'start_date' : DEFAULT_DATE})
       self.op = MyCustomOperator(
           dag=self.dag,
           task_id='test',
           prefix='s3://bucket/some/prefix',
       )
       self.ti = TaskInstance(task=self.op, execution_date=DEFAULT_DATE)

   def test_execute_no_trigger(self):
       self.ti.run(ignore_ti_state=True)
       assert self.ti.state == State.SUCCESS
       # Assert something related to tasks results
```

Self-Checks

You can also implement checks in a DAG to make sure the tasks are producing the results as expected. As an example, if you have a task that pushes data to S3, you can implement a check in the next task. For example, the check could make sure that the partition is created in S3 and perform some simple checks to determine if the data is correct.

Similarly, if you have a task that starts a microservice in Kubernetes or Mesos, you should check if the service has started or not using **`airflow.providers.http.sensors.http.HttpSensor`** .

```
task = PushToS3(...)
check = S3KeySensor(
   task_id='check_parquet_exists',
   bucket_key="s3://bucket/key/foo.parquet",
   poke_interval=0,
   timeout=0
)
task >> check
```

Staging environment

If possible, keep a staging environment to test the complete DAG run before deploying in the production. Make sure your DAG is parameterized to change the variables, e.g., the output path of S3 operation or the database used to read the configuration. Do not hard code values inside the DAG and then change them manually according to the environment.

You can use environment variables to parameterize the DAG.

```
import os

dest = os.environ.get(
   "MY_DAG_DEST_PATH",
   "s3://default-target/path/"
)
```

Mocking variables and connections

When you write tests for code that uses variables or a connection, you must ensure that they exist when you run the tests. The obvious solution is to save these objects to the database so they can be read while your code is executing. However, reading and writing objects to the database are burdened with additional time overhead. In order to speed up the test execution, it is worth simulating the existence of these objects without saving them to the database. For this, you can create environment variables with mocking **`os.environ`** using **`unittest.mock.patch.dict()`** .

For variable, use **`AIRFLOW_VAR_{KEY}`** .

```
with mock.patch.dict('os.environ', AIRFLOW_VAR_KEY="env-value"):
    assert "env-value" == Variable.get("key")
```

For connection, use **`AIRFLOW_CONN_{CONN_ID}`** .

```
conn = Connection(
    conn_type="gcpssh",
    login="cat",
    host="conn-host",
)
conn_uri = conn.get_uri()
with mock.patch.dict("os.environ", AIRFLOW_CONN_MY_CONN=conn_uri):
  assert "cat" == Connection.get("my_conn").login
```

Previous Next

Was this entry helpful?

☆ ☆ ☆ ☆ ☆

Want to be a part of Apache Airflow? Join community

License Donate Thanks
Security
© The Apache Software Foundation 2019

Apache Airflow, Apache, Airflow, the Airflow logo, and the Apache feather logo are either registered trademarks or trademarks of The Apache Software Foundation. All other products or name brands are trademarks of their respective holders, including The Apache Software Foundation.