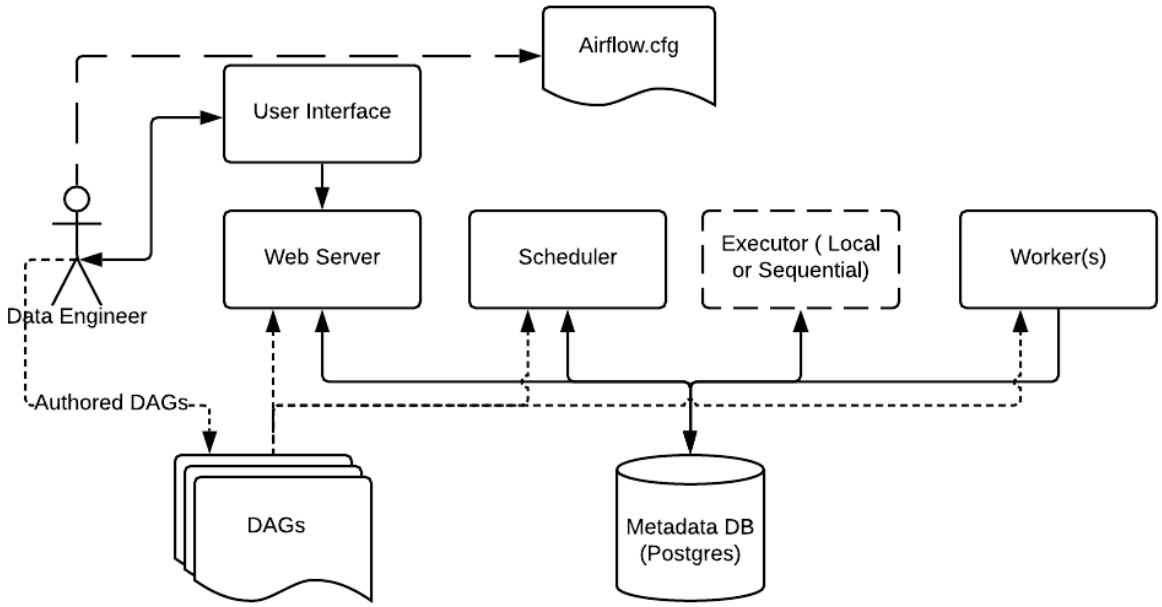
Concepts

The Airflow platform is a tool for describing, executing, and monitoring workflows.

Basic Airflow architecture

Primarily intended for development use, the basic Airflow architecture with the Local and Sequential executors is an excellent starting point for understanding the architecture of Apache Airflow.



There are a few components to note:

- **Metadata Database**: Airflow uses a SQL database to store metadata about the data pipelines being run. In the diagram above, this is represented as Postgres which is extremely popular with Airflow. Alternate databases supported with Airflow include MySQL.
- **Web Server** and **Scheduler**: The Airflow web server and Scheduler are separate processes run (in this case) on the local machine and interact with the database mentioned above.
- The **Executor** is shown separately above, since it is commonly discussed within Airflow and in the documentation, but in reality it is NOT a separate process, but run within the Scheduler.
- The **Worker(s)** are separate processes which also interact with the other components of the Airflow architecture and the metadata repository.
- `airflow.cfg` is the Airflow configuration file which is accessed by the Web Server, Scheduler, and Workers.
- **DAGs** refers to the DAG files containing Python code, representing the data pipelines to be run by Airflow. The location of these files is specified in the Airflow configuration file, but they need to be accessible by the Web Server, Scheduler, and Workers.

Core Ideas

DAGs

In Airflow, a `DAG` – or a Directed Acyclic Graph – is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

A DAG is defined in a Python script, which represents the DAGs structure (tasks and their dependencies) as code.

For example, a simple DAG could consist of three tasks: A, B, and C. It could say that A has to run successfully before B can run, but C can run anytime. It could say that task A times out after 5 minutes, and B can be restarted up to 5 times in case it fails. It might also say that the workflow will run every night at 10pm, but should not start until a certain date.

In this way, a DAG describes *how* you want to carry out your workflow; but notice that we haven't said anything about *what* we actually want to do! A, B, and C could be anything. Maybe A prepares data for B to analyze while C sends an email. Or perhaps A monitors your location so B can open your garage door while C turns on your house lights. The important thing is that the DAG isn't concerned with what its constituent tasks do; its job is to make sure that whatever they do happens at the right time, or in the right order, or with the right handling of any unexpected issues.

DAGs are defined in standard Python files that are placed in Airflow's `DAG_FOLDER`. Airflow will execute the code in each file to dynamically build the `DAG` objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.

Note

When searching for DAGs, Airflow only considers Python files that contain the strings "airflow" and "dag" by default (case-insensitive). To consider all Python files instead, disable the `DAG_DISCOVERY_SAFE_MODE` configuration flag.

Scope

Airflow will load any `DAG` object it can import from a DAGfile. Critically, that means the DAG must appear in `globals()`. Consider the following two DAGs. Only `dag_1` will be loaded; the other one only appears in a local scope.

```
dag_1 = DAG('this_dag_will_be_discovered')

def my_function():
    dag_2 = DAG('but_this_dag_will_not')

my_function()
```

Sometimes this can be put to good use. For example, a common pattern with `SubDagOperator` is to define the subdag inside a function so that Airflow doesn't try to load it as a standalone DAG.

Default Arguments

If a dictionary of `default_args` is passed to a DAG, it will apply them to any of its operators. This makes it easy to apply a common parameter to many operators without having to type it many times.

```
default_args = {
    'start_date': datetime(2016, 1, 1),
    'owner': 'airflow'
}

dag = DAG('my_dag', default_args=default_args)
op = DummyOperator(task_id='dummy', dag=dag)
print(op.owner) # Airflow
```

Context Manager

Added in Airflow 1.8

DAGs can be used as context managers to automatically assign new operators to that DAG.

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    op = DummyOperator('op')

op.dag is dag # True
```

TaskFlow API

New in version 2.0.0.

Airflow 2.0 adds a new style of authoring dags called the TaskFlow API which removes a lot of the boilerplate around creating PythonOperators, managing dependencies between task and accessing XCom values. (During development this feature was called “Functional DAGs”, so if you see or hear any references to that, it’s the same thing)

Outputs and inputs are sent between tasks using XCom values. In addition, you can wrap functions as tasks using the task decorator. Airflow will also automatically add dependencies between tasks to ensure that XCom messages are available when operators are executed.

Example DAG built with the TaskFlow API

```
with DAG(
    'send_server_ip', default_args=default_args, schedule_interval=None
) as dag:

  # Using default connection as it's set to httpbin.org by default
  get_ip = SimpleHttpOperator(
      task_id='get_ip', endpoint='get', method='GET', xcom_push=True
  )

  @dag.task(multiple_outputs=True)
  def prepare_email(raw_json: str) -> Dict[str, str]:
    external_ip = json.loads(raw_json)['origin']
    return {
      'subject':f'Server connected from {external_ip}',
      'body': f'Seems like today your server executing Airflow is connected from the external IP {external_ip}<br>'
    }

  email_info = prepare_email(get_ip.output)

  send_email = EmailOperator(
      task_id='send_email',
      to='example@example.com',
      subject=email_info['subject'],
      html_content=email_info['body']
  )
```

DAG decorator

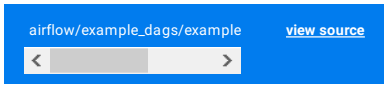
New in version 2.0.0.

In addition to creating DAGs using context manager, in Airflow 2.0 you can also create DAGs from a function. DAG decorator creates a DAG generator function. Any function decorated with `@dag` returns a DAG object.

DAG decorator also sets up the parameters you have in the function as DAG params. This allows you to parameterize your DAGs and set the parameters when triggering the DAG manually. See Passing Parameters when triggering dags to learn how to pass parameters when triggering DAGs.

You can also use the parameters on jinja templates by using the `{{context.params}}` dictionary.

Example DAG with decorator:



```
@dag(default_args=DEFAULT_ARGS, sche
def example_dag_decorator(email: str
    """
    DAG to send server IP to email.

    :param email: Email to send IP t
    :type email: str
    """
    get_ip = GetRequestOperator(task

    @task(multiple_outputs=True)
    def prepare_email(raw_json: Dict
        external_ip = raw_json['orig
        return {
            'subject': f'Server conn
            'body': f'Seems like toc
        }

    email_info = prepare_email(get_i

    EmailOperator(
        task_id='send_email', to=ema
    )


dag = example_dag_decorator()
```

Note

Note that Airflow will only load DAGs that appear in `globals()` as noted in scope section. This means you need to make sure to have a variable for your returned DAG is in the module scope. Otherwise Airflow won't detect your decorated DAG.

`executor_config`

The `executor_config` is an argument placed into operators that allow airflow users to override tasks before launch. Currently this is primarily used by the **`KubernetesExecutor`**, but will soon be available for other overrides.

DAG Runs

A DAG run is an instantiation of a DAG, containing task instances that run for a specific `execution_date`.

A DAG run is usually created by the Airflow scheduler, but can also be created by an external trigger. Multiple DAG runs may be running at once for a particular DAG, each of them having a different `execution_date`. For example, we might currently have two DAG runs that are in progress for 2016-01-01 and 2016-01-02 respectively.

execution_date

The `execution_date` is the *logical* date and time which the DAG Run, and its task instances, are running for.

This allows task instances to process data for the desired *logical* date & time. While a task_instance or DAG run might have an *actual* start date of now, their *logical* date might be 3 months ago because we are busy reloading something.

In the prior example the `execution_date` was 2016-01-01 for the first DAG Run and 2016-01-02 for the second.

A DAG run and all task instances created within it are instanced with the same `execution_date`, so that logically you can think of a DAG run as simulating the DAG running all of its tasks at some previous date & time specified by the `execution_date`.

Tasks

A Task defines a unit of work within a DAG; it is represented as a node in the DAG graph, and it is written in Python.

Each task is an implementation of an Operator, for example a `PythonOperator` to execute some Python code, or a `BashOperator` to run a Bash command.

The task implements an operator by defining specific values for that operator, such as a Python callable in the case of `PythonOperator` or a Bash command in the case of `BashOperator`.

Relations between Tasks

Consider the following DAG with two tasks. Each task is a node in our DAG, and there is a dependency from task_1 to task_2:

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    task_1 = DummyOperator('task_1')
    task_2 = DummyOperator('task_2')
    task_1 >> task_2 # Define dependencies
```

We can say that task_1 is *upstream* of task_2, and conversely task_2 is *downstream* of task_1. When a DAG Run is created, task_1 will start running and task_2 waits for task_1 to complete successfully before it may start.

Python task decorator

New in version 2.0.0.

Airflow `task` decorator converts any Python function to an Airflow operator. The decorated function can be called once to set the arguments and key arguments for operator execution.

```
with DAG('my_dag', start_date=datetime(2020, 5, 15)) as dag:
    @dag.task
    def hello_world():
        print('hello world!')


    # Also...
    from airflow.decorators import task


    @task
    def hello_name(name: str):
        print(f'hello {name}!')


    hello_name('Airflow users')
```

Task decorator captures returned values and sends them to the XCom backend. By default, the returned value is saved as a single XCom value. You can set `multiple_outputs` key argument to `True` to unroll dictionaries, lists or tuples into separate XCom values. This can be used with regular operators to create DAGs with Task Flow API.

Calling a decorated function returns an `XComArg` instance. You can use it to set templated fields on downstream operators.

You can call a decorated function more than once in a DAG. The decorated function will automatically generate a unique `task_id` for each generated operator.

```
with DAG('my_dag', start_date=datetime(2020, 5, 15)) as dag:

  @dag.task
  def update_user(user_id: int):
    ...

  # Avoid generating this list dynamically to keep DAG topology stable between DAG runs
  for user_id in user_ids:
    update_user(user_id)

  # This will generate an operator for each user_id
```

Task ids are generated by appending a number at the end of the original task id. For the above example, the DAG will have the following task ids: `[update_user, update_user__1, update_user__2, ... update_user__n]` .

Due to dynamic nature of the ids generations users should be aware that changing a DAG by adding or removing additional invocations of task-decorated

function may change `task_id` of other task of the same type within a single DAG.

For example, if there are many task-decorated tasks without explicitly given task_id. Their `task_id` will be generated sequentially: `task__1`, `task__2`, `task__3`, etc. After the DAG goes into production, one day someone inserts a new task before `task__2`. The `task_id` after that will all be shifted forward by one place. This is going to produce `task__1`, `task__2`, `task__3`, `task__4`. But at this point the `task__3` is no longer the same `task__3` as before. This may create confusion when analyzing history logs / DagRuns of a DAG that changed over time.

Accessing current context

To retrieve current execution context you can use `get_current_context` method. In this way you can gain access to context dictionary from within your operators. This is especially helpful when using `@task` decorator.

```
from airflow.operators.python import task, get_current_context

@task
def my_task():
    context = get_current_context()
    ti = context["ti"]
```

Current context is accessible only during the task execution. The context is not accessible during `pre_execute` or `post_execute`. Calling this method outside execution context will raise an error.

Task Instances

A task instance represents a specific run of a task and is characterized as the combination of a DAG, a task, and a point in time (`execution_date`). Task instances also have an indicative state, which could be “running”, “success”, “failed”, “skipped”, “up for retry”, etc.

Tasks are defined in DAGs, and both are written in Python code to define what you want to do. Task Instances belong to DAG Runs, have an associated `execution_date`, and are instantiated, runnable entities.

Relations between Task Instances

Again consider the following tasks, defined for some DAG:

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    task_1 = DummyOperator('task_1')
    task_2 = DummyOperator('task_2')
    task_1 >> task_2 # Define dependencies
```

When we enable this DAG, the scheduler creates several DAG Runs - one with `execution_date` of 2016-01-01, one with `execution_date` of 2016-01-02, and so on up to the current date.

Each DAG Run will contain a task_1 Task Instance and a task_2 Task instance. Both Task Instances will have `execution_date` equal to the DAG Run’s `execution_date`, and each task_2 will be *downstream* of (depends on) its task_1.

We can also say that task_1 for 2016-01-01 is the *previous* task instance of the task_1 for 2016-01-02. Or that the DAG Run for 2016-01-01 is the *previous* DAG Run to the DAG Run of 2016-01-02. Here, *previous* refers to the logical past/prior `execution_date`, that runs independently of other runs, and *upstream* refers to a dependency within the same run and having the same `execution_date`.

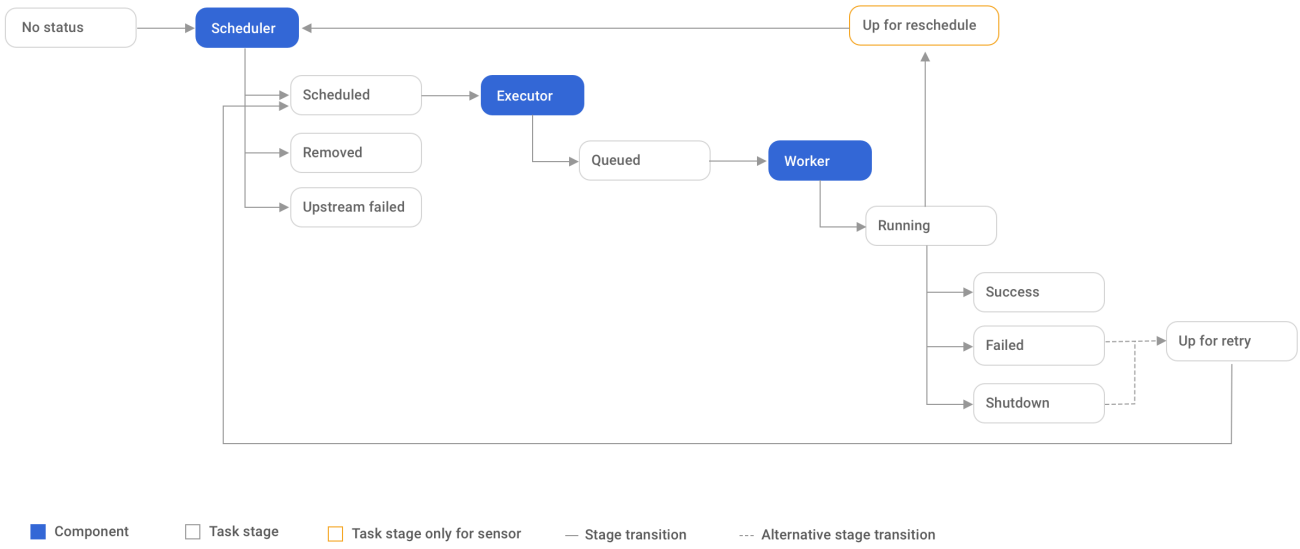
Note

The Airflow documentation sometimes refers to *previous* instead of *upstream* in places, and vice-versa. If you find any occurrences of this, please help us improve by contributing some corrections!

Task Lifecycle

A task goes through various stages from start to completion. In the Airflow UI (graph and tree views), these stages are displayed by a color representing each stage:

The complete lifecycle of the task looks like this:



The happy flow consists of the following stages:

1. No status (scheduler created empty task instance)
2. Scheduled (scheduler determined task instance needs to run)
3. Queued (scheduler sent task to executor to run on the queue)
4. Running (worker picked up a task and is now running it)
5. Success (task completed)

Operators

While DAGs describe *how* to run a workflow, `Operators` determine what actually gets done by a task.

An operator describes a single task in a workflow. Operators are usually (but not always) atomic, meaning they can stand on their own and don't need to share resources with any other operators. The DAG will make sure that operators run in the correct order; other than those dependencies, operators generally run independently. In fact, they may run on two completely different machines.

This is a subtle but very important point: in general, if two operators need to share information, like a filename or small amount of data, you should consider combining them into a single operator. If it absolutely can't be avoided, Airflow does have a feature for operator cross-communication called XCom that is described in the section XComs

Airflow provides many built-in operators for many common tasks, including:

- **`BashOperator`** - executes a bash command
- **`PythonOperator`** - calls an arbitrary Python function
- **`EmailOperator`** - sends an email

There are also other, commonly used operators that are installed together with airflow automatically, by pre-installing some Provider packages packages (they are always available no matter which extras you chose when installing Apache Airflow:

- **`SimpleHttpOperator`** - sends an HTTP request
- **`SqliteOperator`** - SQLite DB operator

In addition to these basic building blocks, there are many more specific operators developed by the community that you can install additionally by installing community-maintained provider packages. You can install them by adding an extra (for example (`[mysql]`) when installing Airflow or by installing additional packages manually (for example `apache-airflow-providers-mysql` package).

Some examples of popular operators are:

- `MySqlOperator`
- `PostgresOperator`
- `MsSqlOperator`
- `OracleOperator`
- `JdbcOperator`
- `DockerOperator`
- `HiveOperator`
- `S3FileTransformOperator`
- `PrestoToMySqlOperator` ,
- `SlackAPIOperator`

But there are many, many more - you can see the list of those by following the providers documentation at Provider packages.

Operators are only loaded by Airflow if they are assigned to a DAG.

See also

- List Airflow operators
- How-to guides for some Airflow operators

Sensors

`Sensor` is an Operator that waits (polls) for a certain time, file, database row, S3 key, another DAG/task, etc…

There are currently 3 different modes for how a sensor operates:

Schedule Mode	Description	Use case
`poke` (default)	The sensor is taking up a worker slot for its whole execution time and sleeps between pokes.	Use this mode if the expected runtime of the sensor is short or if a short poke interval is required. Note that the sensor will hold onto a worker slot and a pool slot for the duration of the sensor's runtime in this mode.
`reschedule`	The sensor task frees the worker slot when the criteria is not yet met and it's rescheduled at a later time.	Use this mode if the time before the criteria is met is expected to be quite long. The poke interval should be more than one minute to prevent too much load on the scheduler.
`smart sensor`	smart sensor is a service (run by a builtin DAG) which consolidate the execution of sensors in batches. Instead of holding a long running process for each sensor and poking periodically, a sensor will only store poke context at `sensor_instance` table and then exits with a 'sensing' state.	Use this mode if you have a large amount of sensor tasks running in your airflow cluster. This can largely reduce airflow's infrastructure cost and improve cluster stability - reduce meta database load.

How to use:

For `poke|schedule` mode, you can configure them at the task level by supplying the `mode` parameter, i.e. `S3KeySensor(task_id='check-bucket', mode='reschedule', ...)` .

For `smart sensor` , you need to configure it in `airflow.cfg` , for example:

```
[smart_sensor]
use_smart_sensor = true
shard_code_upper_limit = 10000

# Users can change the following config based on their requirements
shards = 5
sensors_enabled = NamedHivePartitionSensor, MetastorePartitionSensor
```

For more information on how to configure `smart-sensor` and its architecture, see: Smart Sensor Architecture and Configuration

DAG Assignment

Added in Airflow 1.8

Operators do not have to be assigned to DAGs immediately (previously `dag` was a required argument). However, once an operator is assigned to a DAG, it can not be transferred or unassigned. DAG assignment can be done explicitly when the operator is created, through deferred assignment, or even inferred from other operators.

```
dag = DAG('my_dag', start_date=datetime(2016, 1, 1))

# sets the DAG explicitly
explicit_op = DummyOperator(task_id='op1', dag=dag)

# deferred DAG assignment
deferred_op = DummyOperator(task_id='op2')
deferred_op.dag = dag

# inferred DAG assignment (linked operators must be in the same DAG)
inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)
```

Bitshift Composition

Added in Airflow 1.8

We recommend you setting operator relationships with bitshift operators rather than `set_upstream()` and `set_downstream()`.

Traditionally, operator relationships are set with the `set_upstream()` and `set_downstream()` methods. In Airflow 1.8, this can be done with the Python bitshift operators `>>` and `<<`. The following four statements are all functionally equivalent:

```
op1 >> op2
op1.set_downstream(op2)

op2 << op1
op2.set_upstream(op1)
```

When using the bitshift to compose operators, the relationship is set in the direction that the bitshift operator points. For example, `op1 >> op2` means that `op1` runs first and `op2` runs second. Multiple operators can be composed – keep in mind the chain is executed left-to-right and the rightmost object is always returned. For example:

```
op1 >> op2 >> op3 << op4
```

is equivalent to:

```
op1.set_downstream(op2)
op2.set_downstream(op3)
op3.set_upstream(op4)
```

We can put this all together to build a simple pipeline:

```
with DAG('my_dag', start_date=datetime(2016, 1, 1)) as dag:
    (
        DummyOperator(task_id='dummy_1')
        >> BashOperator(
            task_id='bash_1',
            bash_command='echo "HELLO!"')
        >> PythonOperator(
            task_id='python_1',
            python_callable=lambda: print("GOODBYE!"))
    )
```

Bitshift can also be used with lists. For example:

```
op1 >> [op2, op3] >> op4
```

is equivalent to:

```
op1 >> op2 >> op4
op1 >> op3 >> op4
```

and equivalent to:

```
op1.set_downstream([op2, op3])
op4.set_upstream([op2, op3])
```

Relationship Builders

Moved in Airflow 2.0

In Airflow 2.0 those two methods moved from `airflow.utils.helpers` to `airflow.models.baseoperator`.

`chain` and `cross_downstream` function provide easier ways to set relationships between operators in specific situation.

When setting a relationship between two lists, if we want all operators in one list to be upstream to all operators in the other, we cannot use a single bitshift composition. Instead we have to split one of the lists:

```
[op1, op2, op3] >> op4
[op1, op2, op3] >> op5
[op1, op2, op3] >> op6
```

`cross_downstream` could handle list relationships easier.

```
cross_downstream([op1, op2, op3], [op4, op5, op6])
```

When setting single direction relationships to many operators, we could concat them with bitshift composition.

```
op1 >> op2 >> op3 >> op4 >> op5
```

This can be accomplished using `chain`

```
chain(op1, op2, op3, op4, op5)
```

even without operator's name

```
chain([DummyOperator(task_id='op' + i, dag=dag) for i in range(1, 6)])
```

`chain` can handle a list of operators

```
chain(op1, [op2, op3], op4)
```

is equivalent to:

```
op1 >> [op2, op3] >> op4
```

When `chain` sets relationships between two lists of operators, they must have the same size.

```
chain(op1, [op2, op3], [op4, op5], op6)
```

is equivalent to:

```
op1 >> [op2, op3]
op2 >> op4
op3 >> op5
[op4, op5] >> op6
```

Workflows

You're now familiar with the core building blocks of Airflow. Some of the concepts may sound very similar, but the vocabulary can be conceptualized like this:

- DAG: The work (tasks), and the order in which work should take place (dependencies), written in Python.
- DAG Run: An instance of a DAG for a particular logical date and time.
- Operator: A class that acts as a template for carrying out some work.
- Task: Defines work by implementing an operator, written in Python.
- Task Instance: An instance of a task - that has been assigned to a DAG and has a state associated with a specific DAG run (i.e for a specific execution_date).
- execution_date: The logical date and time for a DAG Run and its Task Instances.

By combining `DAGs` and `Operators` to create `TaskInstances`, you can build complex workflows.

Additional Functionality

In addition to the core Airflow objects, there are a number of more complex features that enable behaviors like limiting simultaneous access to resources, cross-communication, conditional execution, and more.

Hooks

Hooks are interfaces to external platforms and databases like Hive, S3, MySQL, Postgres, HDFS, and Pig. Hooks implement a common interface when possible, and act as a building block for operators. They also use the `airflow.models.connection.Connection` model to retrieve hostnames and

authentication information. Hooks keep authentication code and information out of pipelines, centralized in the metadata database.

Hooks are also very useful on their own to use in Python scripts, Airflow airflow.operators.PythonOperator, and in interactive environments like iPython or Jupyter Notebook.

See also

List Airflow hooks

Pools

Some systems can get overwhelmed when too many processes hit them at the same time. Airflow pools can be used to **limit the execution parallelism** on arbitrary sets of tasks. The list of pools is managed in the UI (`Menu -> Admin -> Pools`) by giving the pools a name and assigning it a number of worker slots. Tasks can then be associated with one of the existing pools by using the `pool` parameter when creating tasks (i.e., instantiating operators).

```
aggregate_db_message_job = BashOperator(
    task_id='aggregate_db_message_job',
    execution_timeout=timedelta(hours=3),
    pool='ep_data_pipeline_db_msg_agg',
    bash_command=aggregate_db_message_job_cmd,
    dag=dag)
aggregate_db_message_job.set_upstream(wait_for_empty_queue)
```

The `pool` parameter can be used in conjunction with `priority_weight` to define priorities in the queue, and which tasks get executed first as slots open up in the pool. The default `priority_weight` is `1` , and can be bumped to any number. When sorting the queue to evaluate which task should be executed next, we use the `priority_weight` , summed up with all of the `priority_weight` values from tasks downstream from this task. You can use this to bump a specific important task and the whole path to that task gets prioritized accordingly.

Tasks will be scheduled as usual while the slots fill up. Once capacity is reached, runnable tasks get queued and their state will show as such in the UI. As slots free up, queued tasks start running based on the `priority_weight` (of the task and its descendants).

Note that if tasks are not given a pool, they are assigned to a default pool `default_pool` . `default_pool` is initialized with 128 slots and can changed through the UI or CLI (though it cannot be removed).

To combine Pools with SubDAGs see the SubDAGs section.

Connections

The information needed to connect to external systems is stored in the Airflow metastore database and can be managed in the UI (`Menu -> Admin -> Connections`). A `conn_id` is defined there, and hostname / login / password / schema information attached to it. Airflow pipelines retrieve centrally-managed connections information by specifying the relevant `conn_id` .

Airflow also provides a mechanism to store connections outside the database, e.g. in environment variables. Additional sources may be enabled, e.g. AWS SSM Parameter Store, or you may roll your own secrets backend.

Many hooks have a default `conn_id` , where operators using that hook do not need to supply an explicit connection ID. For example, the default `conn_id` for the **`PostgresHook`** is `postgres_default` .

See Managing Connections for details on creating and managing connections.

XComs

XComs let tasks exchange messages, allowing more nuanced forms of control and shared state. The name is an abbreviation of “cross-communication”. XComs are principally defined by a key, value, and timestamp, but also track attributes like the task/DAG that created the XCom and when it should become visible. Any object that can be pickled can be used as an XCom value, so users should make sure to use objects of appropriate size.

XComs can be “pushed” (sent) or “pulled” (received). When a task pushes an XCom, it makes it generally available to other tasks. Tasks can push XComs at any time by calling the `xcom_push()` method. In addition, if a task returns a value (either from its Operator’s `execute()` method, or from a PythonOperator’s `python_callable` function), then an XCom containing that value is automatically pushed.

Tasks call `xcom_pull()` to retrieve XComs, optionally applying filters based on criteria like `key` , source `task_ids` , and source `dag_id` . By default,

`xcom_pull()` filters for the keys that are automatically given to XComs when they are pushed by being returned from execute functions (as opposed to XComs that are pushed manually).

If `xcom_pull` is passed a single string for `task_ids`, then the most recent XCom value from that task is returned; if a list of `task_ids` is passed, then a corresponding list of XCom values is returned.

```
# inside a PythonOperator called 'pushing_task'
def push_function():
    return value

# inside another PythonOperator
def pull_function(task_instance):
    value = task_instance.xcom_pull(task_ids='pushing_task')
```

When specifying arguments that are part of the context, they will be automatically passed to the function.

It is also possible to pull XCom directly in a template, here's an example of what this may look like:

```
SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
```

Note that XComs are similar to Variables, but are specifically designed for inter-task communication rather than global settings.

Custom XCom backend

It is possible to change `XCom` behaviour of serialization and deserialization of tasks' result. To do this one have to change `xcom_backend` parameter in Airflow config. Provided value should point to a class that is subclass of **`BaseXCom`**. To alter the serialization / deserialization mechanism the custom class should override `serialize_value` and `deserialize_value` methods.

It is also possible to override the `orm_deserialize_value` method which is used for deserialization when recreating ORM XCom object. This happens every time we query the XCom table, for example when we want to populate XCom list view in webserver. If your XCom backend performs expensive operations, or has large values that are not useful to show in such a view, override this method to provide an alternative representation. By default Airflow will use `BaseXCom.orm_deserialize_value` method which returns the value stored in Airflow database.

See Modules Management for details on how Python and Airflow manage modules.

Variables

Variables are a generic way to store and retrieve arbitrary content or settings as a simple key value store within Airflow. Variables can be listed, created, updated and deleted from the UI (`Admin -> Variables`), code or CLI. In addition, json settings files can be bulk uploaded through the UI. While your pipeline code definition and most of your constants and variables should be defined in code and stored in source control, it can be useful to have some variables or configuration items accessible and modifiable through the UI.

```
from airflow.models import Variable
foo = Variable.get("foo")
bar = Variable.get("bar", deserialize_json=True)
baz = Variable.get("baz", default_var=None)
```

The second call assumes `json` content and will be deserialized into `bar`. Note that `Variable` is a sqlalchemy model and can be used as such. The third call uses the `default_var` parameter with the value `None`, which either returns an existing value or `None` if the variable isn't defined. The get function will throw a `KeyError` if the variable doesn't exist and no default is provided.

You can use a variable from a jinja template with the syntax :

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

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

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

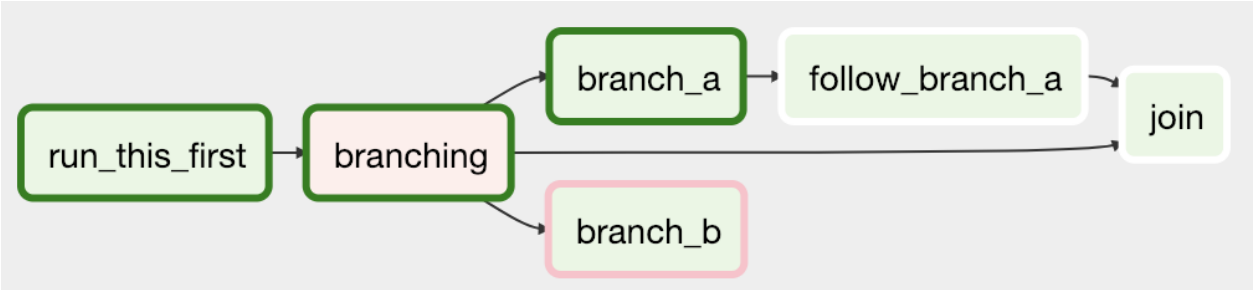
See Managing Variables for details on managing variables.

Branching

Sometimes you need a workflow to branch, or only go down a certain path based on an arbitrary condition which is typically related to something that happened in an upstream task. One way to do this is by using the `BranchPythonOperator` .

The `BranchPythonOperator` is much like the PythonOperator except that it expects a `python_callable` that returns a task_id (or list of task_ids). The task_id returned is followed, and all of the other paths are skipped. The task_id returned by the Python function has to reference a task directly downstream from the BranchPythonOperator task.

Note that when a path is a downstream task of the returned task (list), it will not be skipped:



Paths of the branching task are `branch_a` , `join` and `branch_b` . Since `join` is a downstream task of `branch_a` , it will be excluded from the skipped tasks when `branch_a` is returned by the Python callable.

The `BranchPythonOperator` can also be used with XComs allowing branching context to dynamically decide what branch to follow based on upstream tasks. For example:

```
def branch_func(ti):
    xcom_value = int(ti.xcom_pull(task_ids='start_task'))
    if xcom_value >= 5:
        return 'continue_task'
    else:
        return 'stop_task'

start_op = BashOperator(
    task_id='start_task',
    bash_command="echo 5",
    xcom_push=True,
    dag=dag)

branch_op = BranchPythonOperator(
    task_id='branch_task',
    python_callable=branch_func,
    dag=dag)

continue_op = DummyOperator(task_id='continue_task', dag=dag)
stop_op = DummyOperator(task_id='stop_task', dag=dag)

start_op >> branch_op >> [continue_op, stop_op]
```

If you wish to implement your own operators with branching functionality, you can inherit from **`BaseBranchOperator`** , which behaves similarly to `BranchPythonOperator` but expects you to provide an implementation of the method `choose_branch` . As with the callable for `BranchPythonOperator` , this method should return the ID of a downstream task, or a list of task IDs, which will be run, and all others will be skipped.

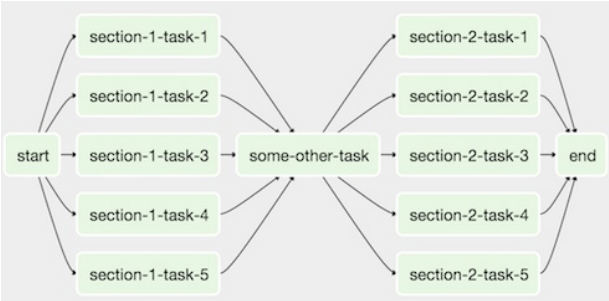
```
class MyBranchOperator(BaseBranchOperator):
    def choose_branch(self, context):
        """
        Run an extra branch on the first day of the month
        """
        if context['execution_date'].day == 1:
            return ['daily_task_id', 'monthly_task_id']
        else:
            return 'daily_task_id'
```

SubDAGs

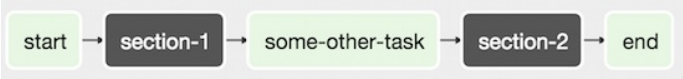
SubDAGs are perfect for repeating patterns. Defining a function that returns a DAG object is a nice design pattern when using Airflow.

Airbnb uses the *stage-check-exchange* pattern when loading data. Data is staged in a temporary table, after which data quality checks are performed against that table. Once the checks all pass the partition is moved into the production table.

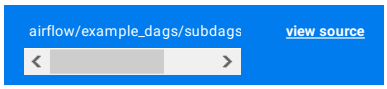
As another example, consider the following DAG:



We can combine all of the parallel `task-*` operators into a single SubDAG, so that the resulting DAG resembles the following:



Note that SubDAG operators should contain a factory method that returns a DAG object. This will prevent the SubDAG from being treated like a separate DAG in the main UI. For example:



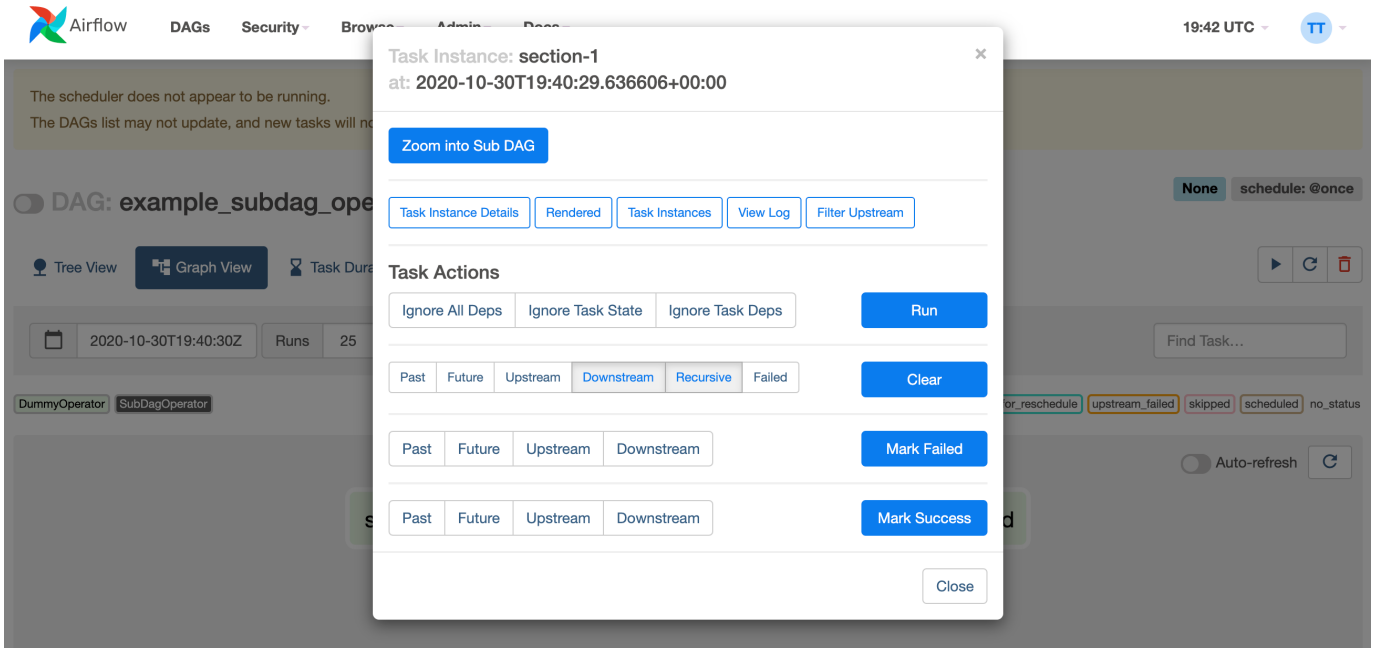
```
from airflow import DAG
from airflow.operators.dummy import
from airflow.utils.dates import days


def subdag(parent_dag_name, child_da
    """
    Generate a DAG to be used as a s

    :param str parent_dag_name: Id o
    :param str child_dag_name: Id of
    :param dict args: Default argume
    :return: DAG to use as a subdag
    :rtype: airflow.models.DAG
    """
    dag_subdag = DAG(
        dag_id=f'{parent_dag_name}.
        default_args=args,
        start_date=days_ago(2),
        schedule_interval="@daily",
    )

    for i in range(5):
        DummyOperator(
            task_id='{}-task-{}'.for
            default_args=args,
            dag=dag_subdag,
        )

    return dag_subdag
```

This SubDAG can then be referenced in your main DAG file:

airflow/example_dags/example_subdag_operat

```
from airflow import DAG
from airflow.example_dags.subdags.su
from airflow.operators.dummy import
from airflow.operators.subdag import
from airflow.utils.dates import days

DAG_NAME = 'example_subdag_operator'

args = {
    'owner': 'airflow',
}

dag = DAG(
    dag_id=DAG_NAME, default_args=ar
)

start = DummyOperator(
    task_id='start',
    dag=dag,
)

section_1 = SubDagOperator(
    task_id='section-1',
    subdag=subdag(DAG_NAME, 'section
    dag=dag,
)

some_other_task = DummyOperator(
    task_id='some-other-task',
    dag=dag,
)

section_2 = SubDagOperator(
    task_id='section-2',
    subdag=subdag(DAG_NAME, 'section
    dag=dag,
)

end = DummyOperator(
    task_id='end',
    dag=dag,
)

start >> section_1 >> some_other_tas
```

You can zoom into a `SubDagOperator` from the graph view of the main DAG to show the tasks contained within the SubDAG:



Some other tips when using SubDAGs:

- by convention, a SubDAG's `dag_id` should be prefixed by its parent and a dot. As in `parent.child`
- share arguments between the main DAG and the SubDAG by passing arguments to the SubDAG operator (as demonstrated above)
- SubDAGs must have a schedule and be enabled. If the SubDAG's schedule is set to `None` or `@once`, the SubDAG will succeed without having done anything
- clearing a **`SubDagOperator`** also clears the state of the tasks within
- marking success on a **`SubDagOperator`** does not affect the state of the tasks within
- refrain from using `depends_on_past=True` in tasks within the SubDAG as this can be confusing
- it is possible to specify an executor for the SubDAG. It is common to use the SequentialExecutor if you want to run the SubDAG in-process and effectively limit its parallelism to one. Using LocalExecutor can be problematic as it may over-subscribe your worker, running multiple tasks in a single slot

See `airflow/example_dags` for a demonstration.

Note that airflow pool is not honored by **`SubDagOperator`**. Hence resources could be consumed by SubdagOperators.

TaskGroup

TaskGroup can be used to organize tasks into hierarchical groups in Graph View. It is useful for creating repeating patterns and cutting down visual clutter. Unlike **`SubDagOperator`**, TaskGroup is a UI grouping concept. Tasks in TaskGroups live on the same original DAG. They honor all the pool configurations.

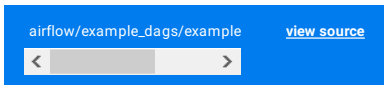
Dependency relationships can be applied across all tasks in a TaskGroup with the `>>` and `<<` operators. For example, the following code puts `task1` and `task2` in TaskGroup `group1` and then puts both tasks upstream of `task3`:

```
with TaskGroup("group1") as group1:
    task1 = DummyOperator(task_id="task1")
    task2 = DummyOperator(task_id="task2")

task3 = DummyOperator(task_id="task3")

group1 >> task3
```

Note

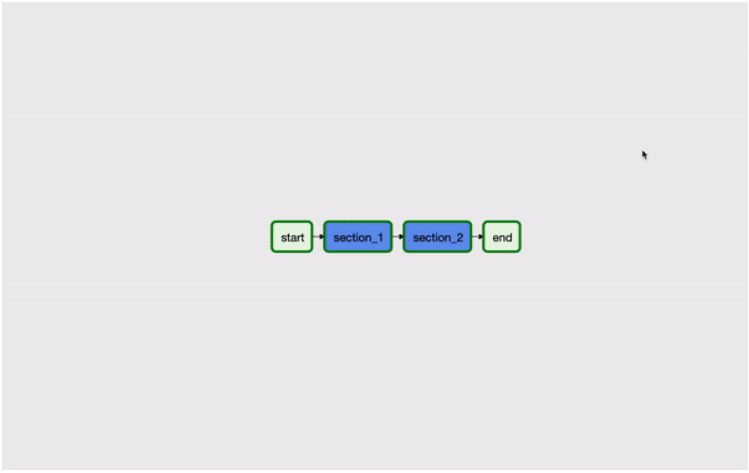
By default, child tasks and TaskGroups have their task_id and group_id prefixed with the group_id of their parent TaskGroup. This ensures uniqueness of group_id and task_id throughout the DAG. To disable the prefixing, pass `prefix_group_id=False` when creating the TaskGroup. This then gives the user full control over the actual group_id and task_id. They have to ensure group_id and task_id are unique throughout the DAG. The option `prefix_group_id=False` is mainly useful for putting tasks on existing DAGs into TaskGroup without altering their task_id.

Here is a more complicated example DAG with multiple levels of nested TaskGroups:



```
with DAG(dag_id="example_task_group'
    start = DummyOperator(task_id="s

    # [START howto_task_group_sectio
    with TaskGroup("section_1", tool
        task_1 = DummyOperator(task_
        task_2 = BashOperator(task_i
        task_3 = DummyOperator(task_

        task_1 >> [task_2, task_3]
    # [END howto_task_group_section_

    # [START howto_task_group_sectio
    with TaskGroup("section_2", tool
        task_1 = DummyOperator(task_

        # [START howto_task_group_in
        with TaskGroup("inner_sectio
            task_2 = BashOperator(ta
            task_3 = DummyOperator(t
            task_4 = DummyOperator(t

            [task_2, task_3] >> task
        # [END howto_task_group_inne

    # [END howto_task_group_section_

    end = DummyOperator(task_id='end

    start >> section_1 >> section_2
```

This animated gif shows the UI interactions. TaskGroups are expanded or collapsed when clicked:



SLAs

Service Level Agreements, or time by which a task or DAG should have succeeded, can be set at a task level as a `timedelta`. If one or many instances have not succeeded by that time, an alert email is sent detailing the list of tasks that missed their SLA. The event is also recorded in the database and made available in the web UI under `Browse->SLA Misses` where events can be analyzed and documented.

SLAs can be configured for scheduled tasks by using the `sla` parameter. In addition to sending alerts to the addresses specified in a task's `email` parameter, the `sla_miss_callback` specifies an additional `Callable` object to be invoked when the SLA is not met.

If you don't want to check SLAs, you can disable globally (all the DAGs) by setting `check_slas=False` under `[core]` section in `airflow.cfg` file:

```
[core]
check_slas = False
```

Note

For information on the email configuration, see Email Configuration

Trigger Rules

Though the normal workflow behavior is to trigger tasks when all their directly upstream tasks have succeeded, Airflow allows for more complex dependency settings.

All operators have a `trigger_rule` argument which defines the rule by which the generated task get triggered. The default value for `trigger_rule` is `all_success` and can be defined as "trigger this task when all directly upstream tasks have succeeded". All other rules described here are based on direct parent tasks and are values that can be passed to any operator while creating tasks:

- `all_success` : (default) all parents have succeeded
- `all_failed` : all parents are in a `failed` or `upstream_failed` state
- `all_done` : all parents are done with their execution
- `one_failed` : fires as soon as at least one parent has failed, it does not wait for all parents to be done
- `one_success` : fires as soon as at least one parent succeeds, it does not wait for all parents to be done
- `none_failed` : all parents have not failed (`failed` or `upstream_failed`) i.e. all parents have succeeded or been skipped
- `none_failed_or_skipped` : all parents have not failed (`failed` or `upstream_failed`) and at least one parent has succeeded.
- `none_skipped` : no parent is in a `skipped` state, i.e. all parents are in a `success` , `failed` , or `upstream_failed` state
- `dummy` : dependencies are just for show, trigger at will

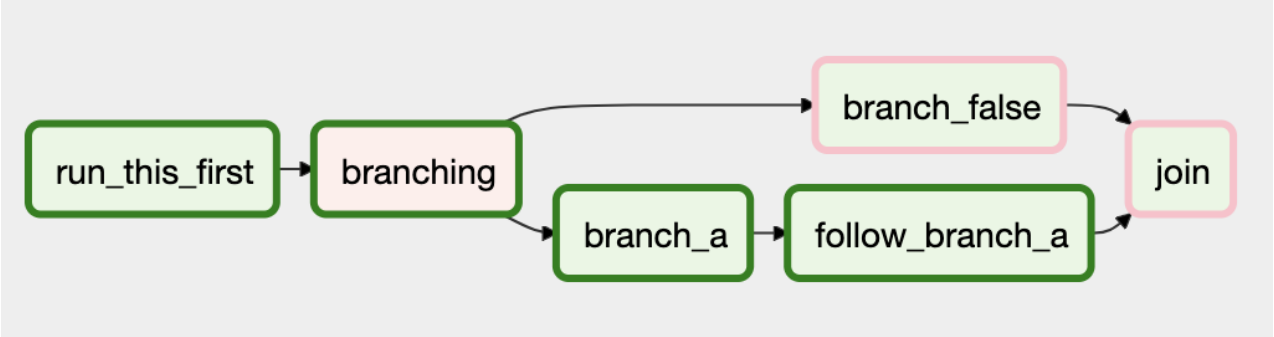
Note that these can be used in conjunction with `depends_on_past` (boolean) that, when set to `True` , keeps a task from getting triggered if the previous schedule for the task hasn't succeeded.

One must be aware of the interaction between trigger rules and skipped tasks in schedule level. Skipped tasks will cascade through trigger rules `all_success` and `all_failed` but not `all_done` , `one_failed` , `one_success` , `none_failed` , `none_failed_or_skipped` , `none_skipped` and `dummy` .

For example, consider the following DAG:

```
#dags/branch_without_trigger.py
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy import DummyOperator
from airflow.operators.python import BranchPythonOperator

dag = DAG(
    dag_id='branch_without_trigger',
    schedule_interval='@once',
    start_date=dt.datetime(2019, 2, 28)
)

run_this_first = DummyOperator(task_id='run_this_first', dag=dag)
branching = BranchPythonOperator(
    task_id='branching', dag=dag,
    python_callable=lambda: 'branch_a'
)

branch_a = DummyOperator(task_id='branch_a', dag=dag)
follow_branch_a = DummyOperator(task_id='follow_branch_a', dag=dag)

branch_false = DummyOperator(task_id='branch_false', dag=dag)

join = DummyOperator(task_id='join', dag=dag)

run_this_first >> branching
branching >> branch_a >> follow_branch_a >> join
branching >> branch_false >> join
```

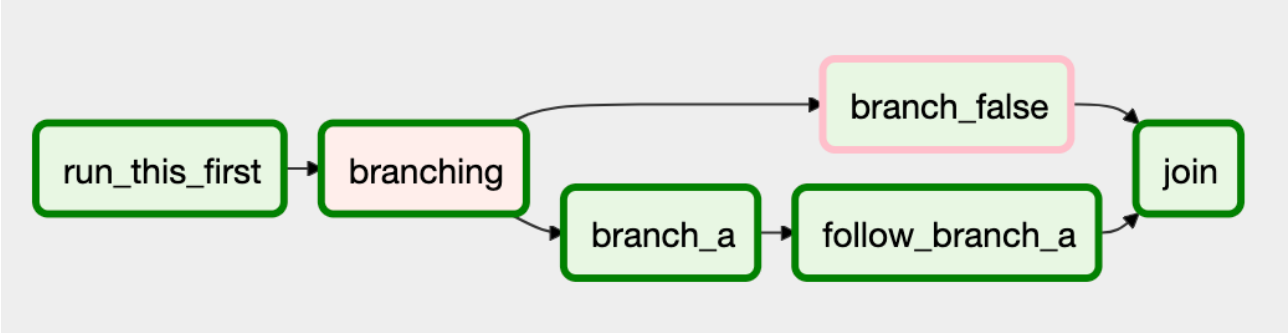
In the case of this DAG, `join` is downstream of `follow_branch_a` and `branch_false` . The `join` task will show up as skipped because its `trigger_rule` is set to `all_success` by default and skipped tasks will cascade through `all_success` .



By setting `trigger_rule` to `none_failed_or_skipped` in `join` task,

```
#dags/branch_with_trigger.py
...
join = DummyOperator(task_id='join', dag=dag, trigger_rule='none_failed_or_skipped')
...
```

The `join` task will be triggered as soon as `branch_false` has been skipped (a valid completion state) and `follow_branch_a` has succeeded. Because skipped tasks **will not** cascade through `none_failed_or_skipped` .

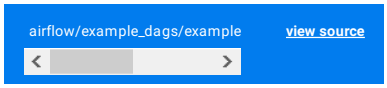


Latest Run Only

Standard workflow behavior involves running a series of tasks for a particular date/time range. Some workflows, however, perform tasks that are independent of run time but need to be run on a schedule, much like a standard cron job. In these cases, backfills or running jobs missed during a pause just wastes CPU cycles.

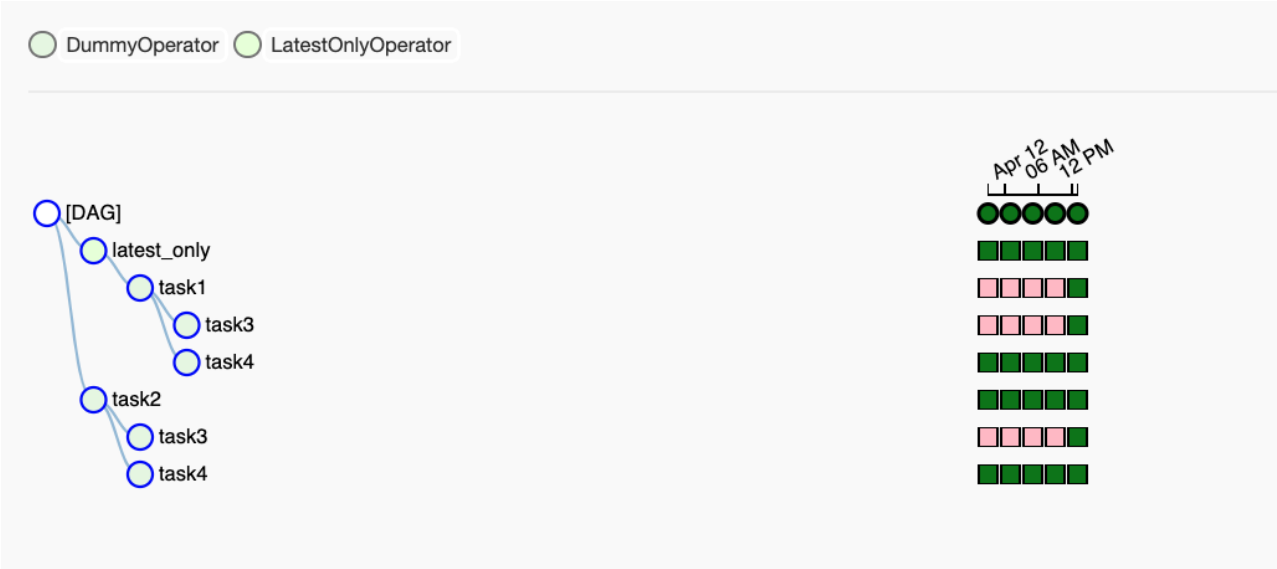
For situations like this, you can use the `LatestOnlyOperator` to skip tasks that are not being run during the most recent scheduled run for a DAG. The `LatestOnlyOperator` skips all direct downstream tasks, if the time right now is not between its `execution_time` and the next scheduled `execution_time` or the DagRun has been externally triggered.

For example, consider the following DAG:



```
import datetime as dt

from airflow import DAG
from airflow.operators.dummy import
from airflow.operators.latest_only i
from airflow.utils.dates import days
from airflow.utils.trigger_rule impo

dag = DAG(
    dag_id='latest_only_with_trigger
    schedule_interval=dt.timedelta(h
    start_date=days_ago(2),
    tags=['example3'],
)

latest_only = LatestOnlyOperator(tas
task1 = DummyOperator(task_id='task1
task2 = DummyOperator(task_id='task2
task3 = DummyOperator(task_id='task3
task4 = DummyOperator(task_id='task4

latest_only >> task1 >> [task3, task
task2 >> [task3, task4]
```

In the case of this DAG, the task `task1` is directly downstream of `latest_only` and will be skipped for all runs except the latest. `task2` is entirely independent of `latest_only` and will run in all scheduled periods. `task3` is downstream of `task1` and `task2` and because of the default `trigger_rule` being `all_success` will receive a cascaded skip from `task1`. `task4` is downstream of `task1` and `task2`, but it will not be skipped, since its `trigger_rule` is set to `all_done`.



Zombies & Undeads

Task instances die all the time, usually as part of their normal life cycle, but sometimes unexpectedly.

Zombie tasks are characterized by the absence of a heartbeat (emitted by the job periodically) and a `running` status in the database. They can occur when a worker node can't reach the database, when Airflow processes are killed externally, or when a node gets rebooted for instance. Zombie killing is performed periodically by the scheduler's process.

Undead processes are characterized by the existence of a process and a matching heartbeat, but Airflow isn't aware of this task as `running` in the database. This mismatch typically occurs as the state of the database is altered, most likely by deleting rows in the "Task Instances" view in the UI. Tasks are instructed to verify their state as part of the heartbeat routine, and terminate themselves upon figuring out that they are in this "undead" state.

Cluster Policy

Cluster policies provide an interface for taking action on every Airflow task or DAG either at DAG load time or just before task execution. In this way users are able to do the following:

- set default arguments on each DAG/task
- checks that DAG/task meets required standards
- perform custom logic of routing task to a queue

And many other options. To use cluster-wide policies users can define in their `airflow_local_settings` the following functions

- `dag_policy` - which as an input takes `dag` argument of **`DAG`** type. This function allows users to define dag-level policy which is executed for every DAG at loading time.
- `task_policy` - which as an input takes `task` argument of **`BaseOperator`** type. This function allows users to define task-level policy which is executed for every task at DAG loading time.
- `task_instance_mutation_hook` - which as an input takes `task_instance` argument of **`TaskInstance`** type. This function allows users to define task-level policy that is executed right before the task execution.

In case of DAG and task policies users may raise **`AirflowClusterPolicyViolation`** to prevent a DAG from being imported or prevent a task from being executed if the task is not compliant with users' check.

Please note, cluster policy will have precedence over task attributes defined in DAG meaning if `task.sla` is defined in dag and also mutated via cluster policy then later will have precedence.

In next sections we show examples of each type of cluster policy.

Where to put `airflow_local_settings.py` ?

Add a `airflow_local_settings.py` file to your `$PYTHONPATH` or to `$AIRFLOW_HOME/config` folder.

See Modules Management for details on how Python and Airflow manage modules.

DAG level cluster policy

In this example we check if each DAG has at least one tag defined. Here is what it may look like:

```
def dag_policy(dag: DAG):
    """Ensure that DAG has at least one tag"""
    if not dag.tags:
        raise AirflowClusterPolicyViolation(
            f"DAG {dag.dag_id} has no tags. At least one tag required. File path: {dag.filepath}"
        )
```

Task level cluster policy

For example, this function could apply a specific queue property when using a specific operator, or enforce a task timeout policy, making sure that no tasks run for more than 48 hours. Here's an example of what this may look like:

```
def task_policy(task: BaseOperator):
    if task.task_type == 'HivePartitionSensor':
        task.queue = "sensor_queue"
    if task.timeout > timedelta(hours=48):
        task.timeout = timedelta(hours=48)
```

As a more advanced example we may consider implementing checks that are intended to help teams using Airflow to protect against common beginner errors that may get past a code reviewer, rather than as technical security controls.

For example, don't run tasks without airflow owners:

```python
def task_must_have_owners(task: BaseOperator):
    if not task.owner or task.owner.lower() == conf.get('operators', 'default_owner'):
        raise AirflowClusterPolicyViolation(
            f'''Task must have non-None non-default owner. Current value: {task.owner}'''
        )
```

If you have multiple checks to apply, it is best practice to curate these rules in a separate python module and have a single policy / task mutation hook that performs multiple of these custom checks and aggregates the various error messages so that a single `AirflowClusterPolicyViolation` can be reported in the UI (and import errors table in the database).

For Example in `airflow_local_settings.py`:

```python
TASK_RULES: List[Callable[[BaseOperator], None]] = [
    task_must_have_owners,
]


def _check_task_rules(current_task: BaseOperator):
    """Check task rules for given task."""
    notices = []
    for rule in TASK_RULES:
        try:
            rule(current_task)
        except AirflowClusterPolicyViolation as ex:
            notices.append(str(ex))
    if notices:
        notices_list = " * " + "\n * ".join(notices)
        raise AirflowClusterPolicyViolation(
            f"DAG policy violation (DAG ID: {current_task.dag_id}, Path: {current_task.dag.filepath}):\n"
            f"Notices:\n"
            f"{notices_list}"
        )


def cluster_policy(task: BaseOperator):
    """Ensure Tasks have non-default owners."""
    _check_task_rules(task)
```

Task instance mutation hook

Task instance mutation hook can be used for example to re-routes the task to execute in a different queue during retries:

```python
def task_instance_mutation_hook(task_instance: TaskInstance):
    if task_instance.try_number >= 1:
        task_instance.queue = 'retry_queue'
```

Documentation & Notes

It's possible to add documentation or notes to your DAGs & task objects that become visible in the web interface ("Graph View" & "Tree View" for DAGs, "Task Details" for tasks). There are a set of special task attributes that get rendered as rich content if defined:

attribute	rendered to
doc	monospace
doc_json	json

attribute	rendered to
doc_yaml	yaml
doc_md	markdown
doc_rst	reStructuredText

Please note that for DAGs, doc_md is the only attribute interpreted.

This is especially useful if your tasks are built dynamically from configuration files, it allows you to expose the configuration that led to the related tasks in Airflow.

```
"""
### My great DAG
"""

dag = DAG('my_dag', default_args=default_args)
dag.doc_md = __doc__

t = BashOperator("foo", dag=dag)
t.doc_md = """\
#Title"
Here's a [url](www.airbnb.com)
"""
```

This content will get rendered as markdown respectively in the “Graph View” and “Task Details” pages.

Jinja Templating

Airflow leverages the power of Jinja Templating and this can be a powerful tool to use in combination with macros (see the Macros reference section).

For example, say you want to pass the execution date as an environment variable to a Bash script using the `BashOperator` .

```
# The execution date as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id='test_env',
    bash_command='/tmp/test.sh ',
    dag=dag,
    env={'EXECUTION_DATE': date})
```

Here, `{{ ds }}` is a macro, and because the `env` parameter of the `BashOperator` is templated with Jinja, the execution date will be available as an environment variable named `EXECUTION_DATE` in your Bash script.

You can use Jinja templating with every parameter that is marked as “templated” in the documentation. Template substitution occurs just before the pre_execute function of your operator is called.

You can also use Jinja templating with nested fields, as long as these nested fields are marked as templated in the structure they belong to: fields registered in `template_fields` property will be submitted to template substitution, like the `path` field in the example below:

```
class MyDataReader:
  template_fields = ['path']

  def __init__(self, my_path):
    self.path = my_path

  # [additional code here...]

t = PythonOperator(
    task_id='transform_data',
    python_callable=transform_data
    op_args=[
      MyDataReader('/tmp/{{ ds }}/my_file')
    ],
    dag=dag)
```

Note

`template_fields` property can equally be a class variable or an instance variable.

Deep nested fields can also be substituted, as long as all intermediate fields are marked as template fields:

```
class MyDataTransformer:
  template_fields = ['reader']

  def __init__(self, my_reader):
    self.reader = my_reader

  # [additional code here...]

class MyDataReader:
  template_fields = ['path']

  def __init__(self, my_path):
    self.path = my_path

  # [additional code here...]

t = PythonOperator(
    task_id='transform_data',
    python_callable=transform_data
    op_args=[
      MyDataTransformer(MyDataReader('/tmp/{{ ds }}/my_file'))
    ],
    dag=dag)
```

You can pass custom options to the Jinja `Environment` when creating your DAG. One common usage is to avoid Jinja from dropping a trailing newline from a template string:

```
my_dag = DAG(dag_id='my-dag',
             jinja_environment_kwargs={
                  'keep_trailing_newline': True,
                  # some other jinja2 Environment options here
             })
```

See Jinja documentation to find all available options.

Exceptions

Airflow defines a number of exceptions; most of these are used internally, but a few are relevant to authors of custom operators or Python callables called from `PythonOperator` tasks. Normally any exception raised from an `execute` method or Python callable will either cause a task instance to fail if it is not configured to retry or has reached its limit on retry attempts, or to be marked as “up for retry”. A few exceptions can be used when different behavior is desired:

- `AirflowSkipException` can be raised to set the state of the current task instance to “skipped”
- `AirflowFailException` can be raised to set the state of the current task to “failed” regardless of whether there are any retry attempts remaining.

This example illustrates some possibilities

```
from airflow.exceptions import AirflowFailException, AirflowSkipException

def fetch_data():
    try:
        data = get_some_data(get_api_key())
        if not data:
            # Set state to skipped and do not retry
            # Downstream task behavior will be determined by trigger rules
            raise AirflowSkipException("No data available.")
    except Unauthorized:
        # If we retry, our api key will still be bad, so don't waste time retrying!
        # Set state to failed and move on
        raise AirflowFailException("Our api key is bad!")
    except TransientError:
        print("Looks like there was a blip.")
        # Raise the exception and let the task retry unless max attempts were reached
        raise
    handle(data)

task = PythonOperator(task_id="fetch_data", python_callable=fetch_data, retries=10)
```

See also

- List of Airflow exceptions

Packaged DAGs

While often you will specify DAGs in a single `.py` file it might sometimes be required to combine a DAG and its dependencies. For example, you might want to combine several DAGs together to version them together or you might want to manage them together or you might need an extra module that is not available by default on the system you are running Airflow on. To allow this you can create a zip file that contains the DAG(s) in the root of the zip file and have the extra modules unpacked in directories.

For instance you can create a zip file that looks like this:

```
my_dag1.py
my_dag2.py
package1/__init__.py
package1/functions.py
```

Airflow will scan the zip file and try to load `my_dag1.py` and `my_dag2.py`. It will not go into subdirectories as these are considered to be potential packages.

In case you would like to add module dependencies to your DAG you basically would do the same, but then it is more suitable to use a virtualenv and pip.

```
virtualenv zip_dag
source zip_dag/bin/activate

mkdir zip_dag_contents
cd zip_dag_contents

pip install --install-option="--install-lib=$PWD" my_useful_package
cp ~/my_dag.py .

zip -r zip_dag.zip *
```

Note

the zip file will be inserted at the beginning of module search list (sys.path) and as such it will be available to any other code that resides within the same interpreter.

Note

packaged dags cannot be used with pickling turned on.

Note

packaged dags cannot contain dynamic libraries (eg. libz.so) these need to be available on the system if a module needs those. In other words only pure Python modules can be packaged.

.airflowignore

A `.airflowignore` file specifies the directories or files in `DAG_FOLDER` or `PLUGINS_FOLDER` that Airflow should intentionally ignore. Each line in `.airflowignore` specifies a regular expression pattern, and directories or files whose names (not DAG id) match any of the patterns would be ignored (under the hood,``Pattern.search()`` is used to match the pattern). Overall it works like a `.gitignore` file. Use the `#` character to indicate a comment; all characters on a line following a `#` will be ignored.

`.airflowignore` file should be put in your `DAG_FOLDER`. For example, you can prepare a `.airflowignore` file with contents

```
project_a
tenant_[\d]
```

Then files like `project_a_dag_1.py`, `TESTING_project_a.py`, `tenant_1.py`, `project_a/dag_1.py`, and `tenant_1/dag_1.py` in your `DAG_FOLDER` would be ignored (If a directory's name matches any of the patterns, this directory and all its subfolders would not be scanned by Airflow at all. This improves efficiency of DAG finding).

The scope of a `.airflowignore` file is the directory it is in plus all its subfolders. You can also prepare `.airflowignore` file for a subfolder in `DAG_FOLDER` and it would only be applicable for that subfolder.

Previous Next