FAQ

Why isn't my task getting scheduled?

There are very many reasons why your task might not be getting scheduled. Here are some of the common causes:

- Does your script "compile", can the Airflow engine parse it and find your DAG object? To test this, you can run `airflow dags list` and confirm that your DAG shows up in the list. You can also run `airflow tasks list foo_dag_id --tree` and confirm that your task shows up in the list as expected. If you use the CeleryExecutor, you may want to confirm that this works both where the scheduler runs as well as where the worker runs.
- Does the file containing your DAG contain the string "airflow" and "DAG" somewhere in the contents? When searching the DAG directory, Airflow ignores files not containing "airflow" and "DAG" in order to prevent the DagBag parsing from importing all python files collocated with user's DAGs.
- Is your `start_date` set properly? The Airflow scheduler triggers the task soon after the `start_date + schedule_interval` is passed.
- Is your `schedule_interval` set properly? The default `schedule_interval` is one day (`datetime.timedelta(1)`). You must specify a different `schedule_interval` directly to the DAG object you instantiate, not as a `default_param` , as task instances do not override their parent DAG's `schedule_interval` .
- Is your `start_date` beyond where you can see it in the UI? If you set your `start_date` to some time say 3 months ago, you won't be able to see it in the main view in the UI, but you should be able to see it in the `Menu -> Browse ->Task Instances` .
- Are the dependencies for the task met? The task instances directly upstream from the task need to be in a `success` state. Also, if you have set `depends_on_past=True` , the previous task instance needs to have succeeded (except if it is the first run for that task). Also, if `wait_for_downstream=True` , make sure you understand what it means - all tasks *immediately* downstream of the *previous* task instance must have succeeded. You can view how these properties are set from the `Task Instance Details` page for your task.
- Are the DagRuns you need created and active? A DagRun represents a specific execution of an entire DAG and has a state (running, success, failed, ...). The scheduler creates new DagRun as it moves forward, but never goes back in time to create new ones. The scheduler only evaluates `running` DagRuns to see what task instances it can trigger. Note that clearing tasks instances (from the UI or CLI) does set the state of a DagRun back to running. You can bulk view the list of DagRuns and alter states by clicking on the schedule tag for a DAG.
- Is the `concurrency` parameter of your DAG reached? `concurrency` defines how many `running` task instances a DAG is allowed to have, beyond which point things get queued.
- Is the `max_active_runs` parameter of your DAG reached? `max_active_runs` defines how many `running` concurrent instances of a DAG there are allowed to be.

You may also want to read the Scheduler section of the docs and make sure you fully understand how it proceeds.

How do I trigger tasks based on another task's failure?

Check out the Trigger Rules.

What's the deal with `start_date` ?

`start_date` is partly legacy from the pre-DagRun era, but it is still relevant in many ways. When creating a new DAG, you probably want to set a global `start_date` for your tasks using `default_args` . The first DagRun to be created will be based on the `min(start_date)` for all your tasks. From that point on, the scheduler creates new DagRuns based on your `schedule_interval` and the corresponding task instances run as your dependencies are met. When introducing new tasks to your DAG, you need to pay special attention to `start_date` , and may want to reactivate inactive DagRuns to get the new task onboarded properly.

We recommend against using dynamic values as `start_date` , especially `datetime.now()` as it can be quite confusing. The task is triggered once the period closes, and in theory an `@hourly` DAG would never get to an hour after now as `now()` moves along.

Previously, we also recommended using rounded `start_date` in relation to your `schedule_interval` . This meant an `@hourly` would be at `00:00` minutes:seconds, a `@daily` job at midnight, a `@monthly` job on the first of the month. This is no longer required. Airflow will now auto align the `start_date` and the `schedule_interval` , by using the `start_date` as the moment to start looking.

You can use any sensor or a `TimeDeltaSensor` to delay the execution of tasks within the schedule interval. While `schedule_interval` does allow specifying a `datetime.timedelta` object, we recommend using the macros or cron expressions instead, as it enforces this idea of rounded schedules.

When using `depends_on_past=True` , it's important to pay special attention to `start_date` , as the past dependency is not enforced only on the specific schedule of the `start_date` specified for the task. It's also important to watch DagRun activity status in time when introducing new `depends_on_past=True` , unless you are planning on running a backfill for the new task(s).

It is also important to note that the task's `start_date`, in the context of a backfill CLI command, gets overridden by the backfill's `start_date` commands. This allows for a backfill on tasks that have `depends_on_past=True` to actually start. If this were not the case, the backfill just would not start.

How can I create DAGs dynamically?

Airflow looks in your `DAGS_FOLDER` for modules that contain `DAG` objects in their global namespace and adds the objects it finds in the `DagBag`. Knowing this, all we need is a way to dynamically assign variable in the global namespace. This is easily done in python using the `globals()` function for the standard library, which behaves like a simple dictionary.

```
def create_dag(dag_id):
    """
    A function returning a DAG object.
    """

    return DAG(dag_id)


for i in range(10):
    dag_id = f'foo_{i}'
    globals()[dag_id] = DAG(dag_id)

    # or better, call a function that returns a DAG object!
    other_dag_id = f'bar_{i}'
    globals()[other_dag_id] = create_dag(other_dag_id)
```

What are all the `airflow tasks run` commands in my process list?

There are many layers of `airflow tasks run` commands, meaning it can call itself.

- Basic `airflow tasks run`: fires up an executor, and tell it to run an `airflow tasks run --local` command. If using Celery, this means it puts a command in the queue for it to run remotely on the worker. If using LocalExecutor, that translates into running it in a subprocess pool.
- Local `airflow tasks run --local`: starts an `airflow tasks run --raw` command (described below) as a subprocess and is in charge of emitting heartbeats, listening for external kill signals and ensures some cleanup takes place if the subprocess fails.
- Raw `airflow tasks run --raw` runs the actual operator's execute method and performs the actual work.

How can my airflow dag run faster?

There are a few variables we can control to improve airflow dag performance:

- `parallelism`: This variable controls the number of task instances that runs simultaneously across the whole Airflow cluster. User could increase the `parallelism` variable in the `airflow.cfg`.
- `concurrency`: The Airflow scheduler will run no more than `concurrency` task instances for your DAG at any given time. Concurrency is defined in your Airflow DAG. If you do not set the concurrency on your DAG, the scheduler will use the default value from the `dag_concurrency` entry in your `airflow.cfg`.
- `task_concurrency`: This variable controls the number of concurrent running task instances across `dag_runs` per task.
- `max_active_runs`: the Airflow scheduler will run no more than `max_active_runs` DagRuns of your DAG at a given time. If you do not set the `max_active_runs` in your DAG, the scheduler will use the default value from the `max_active_runs_per_dag` entry in your `airflow.cfg`.
- `pool`: This variable controls the number of concurrent running task instances assigned to the pool.

How can we reduce the airflow UI page load time?

If your dag takes long time to load, you could reduce the value of `default_dag_run_display_number` configuration in `airflow.cfg` to a smaller value. This configurable controls the number of dag run to show in UI with default value 25.

How to fix Exception: Global variable explicit_defaults_for_timestamp needs to be on (1)?

This means `explicit_defaults_for_timestamp` is disabled in your mysql server and you need to enable it by:

1. Set `explicit_defaults_for_timestamp = 1` under the `mysqld` section in your `my.cnf` file.
2. Restart the Mysql server.

How to reduce airflow dag scheduling latency in production?

Airflow 2 has low DAG scheduling latency out of the box (particularly when compared with Airflow 1.10.x), however if you need more throughput you can start multiple schedulers.

Why next_ds or prev_ds might not contain expected values?

- When scheduling DAG, the `next_ds` `next_ds_nodash` `prev_ds` `prev_ds_nodash` are calculated using `execution_date` and `schedule_interval`. If you set `schedule_interval` as `None` or `@once`, the `next_ds`, `next_ds_nodash`, `prev_ds`, `prev_ds_nodash` values will be set to `None`.
- When manually triggering DAG, the schedule will be ignored, and `prev_ds == next_ds == ds`

How do I stop the sync perms happening multiple times per webserver?

Set the value of `update_fab_perms` configuration in `airflow.cfg` to `False`.

Why did the pause dag toggle turn red?

If pausing or unpausing a dag fails for any reason, the dag toggle will revert to its previous state and turn red. If you observe this behavior, try pausing the dag again, or check the console or server logs if the issue recurs.

Previous Next

Was this entry helpful?