
apache-airflow

Release 3.1.0

Author name not set

May 11, 2025

CONTENT

Apache Airflow® is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. Airflow's extensible Python framework enables you to build workflows connecting with virtually any technology. A web-based UI helps you visualize, manage, and debug your workflows. You can run Airflow in a variety of configurations — from a single process on your laptop to a distributed system capable of handling massive workloads.

WORKFLOWS AS CODE

Airflow workflows are defined entirely in Python. This “workflows as code” approach brings several advantages:

- **Dynamic:** Pipelines are defined in code, enabling dynamic dag generation and parameterization.
- **Extensible:** The Airflow framework includes a wide range of built-in operators and can be extended to fit your needs.
- **Flexible:** Airflow leverages the [Jinja](#) templating engine, allowing rich customizations.

1.1 Dags

A DAG is a model that encapsulates everything needed to execute a workflow. Some DAG attributes include the following:

- **Schedule:** When the workflow should run.
- **Tasks:** *tasks* are discrete units of work that are run on workers.
- **Task Dependencies:** The order and conditions under which *tasks* execute.
- **Callbacks:** Actions to take when the entire workflow completes.
- **Additional Parameters:** And many other operational details.

Let's look at a code snippet that defines a simple dag:

```
from datetime import datetime

from airflow.sdk import DAG, task
from airflow.providers.standard.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

Here you see:

- A dag named "demo", scheduled to run daily starting on January 1st, 2022. A dag is how Airflow represents a workflow.
- Two tasks: One using a BashOperator to run a shell script, and another using the @task decorator to define a Python function.
- The >> operator defines a dependency between the two tasks and controls execution order.

Airflow parses the script, schedules the tasks, and executes them in the defined order. The status of the "demo" dag is displayed in the web interface:

```

1  from datetime import datetime
2
3  from airflow.sdk import DAG, task
4  from airflow.providers.standard.operators.bash import BashOperator
5
6  # A DAG represents a workflow, a collection of tasks
7  with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="@0 0 * * *") as dag:
8      # Tasks are represented as operators
9      hello = BashOperator(task_id="hello", bash_command="echo hello")
10
11     @task()
12     def airflow():
13         print("airflow")
14
15
16     # Set dependencies between tasks
17     hello >> airflow()
18
19
20

```

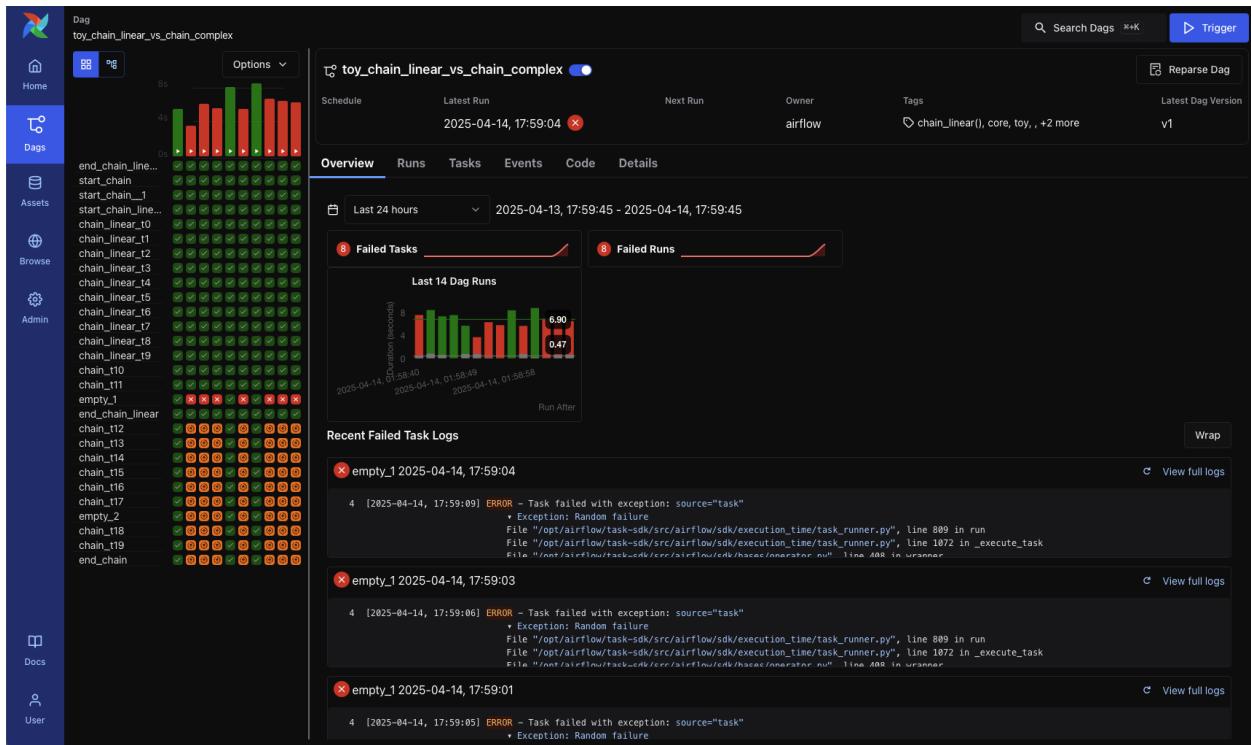
This example uses a simple Bash command and Python function, but Airflow tasks can run virtually any code. You might use tasks to run a Spark job, move files between storage buckets, or send a notification email. Here's what that same dag looks like over time, with multiple runs:

```

Log message source.details: sources=["/root/airflow/logs/dag_id=demo/run_id=manual_2025-04-14T17:56:37.631252+00:00/task_id=hello/attempt=1.log"]
2 [2025-04-14, 17:56:38] INFO - DAG bundles loaded: dag_folder: source="airflow.dag_processing.bundles.manager.DagBundlesManager"
3 [2025-04-14, 17:56:38] INFO - Filling up the DagBag from /files/dags/hello_world.py: source="airflow.models.dagbag.DagBag"
4 [2025-04-14, 17:56:38] INFO - Tmp dir root location: /tmp: source="airflow.task.hooks.airflow.providers.standard.hooks.subprocess.SubprocessHook"
5 [2025-04-14, 17:56:38] INFO - Running command: ['/usr/bin/bash', '-c', 'echo hello'];
source="airflow.task.hooks.airflow.providers.standard.hooks.subprocess.SubprocessHook"
6 [2025-04-14, 17:56:38] INFO - Output: source="airflow.task.hooks.airflow.providers.standard.hooks.subprocess.SubprocessHook"
7 [2025-04-14, 17:56:38] INFO - hello source="airflow.task.hooks.airflow.providers.standard.hooks.subprocess.SubprocessHook"
8 [2025-04-14, 17:56:38] INFO - Command exited with return code 0: source="airflow.task.hooks.airflow.providers.standard.hooks.subprocess.SubprocessHook"
9 [2025-04-14, 17:56:38] INFO - Pushing xcom: ti="RuntimeTaskInstance(id='01983572-2e8c-7908-9590-e3ebcb0f0519')", task_id='hello', dag_id='demo'
run_id='manual_2025-04-14T17:56:37.631252+00:00', try_number=1, map_index=1, hostname='66cd9a53d3', context_carrier=None, task_id_task[BashOperator]=hello,
bundle_instance=LocalDagBundle(name=dags-f0de1f), max_tries=6, start_date=datetime.datetime(2025, 4, 14, 17, 56, 38, 195746, tzinfo=TzInfo(UTC)), end_date=None,
is_mapped=False": source="task"

```

Each column in the grid represents a single dag run. While the graph and grid views are most commonly used, Airflow provides several other views to help you monitor and troubleshoot workflows — such as the [DAG Overview](#) view:



Note

The term “DAG” comes from the mathematical concept “directed acyclic graph”, but the meaning in Airflow has evolved well beyond just the literal data structure associated with the mathematical DAG concept.

CHAPTER
TWO

WHY AIRFLOW®?

Airflow is a platform for orchestrating batch workflows. It offers a flexible framework with a wide range of built-in operators and makes it easy to integrate with new technologies.

If your workflows have a clear start and end and run on a schedule, they're a great fit for Airflow dags.

If you prefer coding over clicking, Airflow is built for you. Defining workflows as Python code provides several key benefits:

- **Version control:** Track changes, roll back to previous versions, and collaborate with your team.
- **Team collaboration:** Multiple developers can work on the same workflow codebase.
- **Testing:** Validate pipeline logic through unit and integration tests.
- **Extensibility:** Customize workflows using a large ecosystem of existing components — or build your own.

Airflow's rich scheduling and execution semantics make it easy to define complex, recurring pipelines. From the web interface, you can manually trigger dags, inspect logs, and monitor task status. You can also backfill dag runs to process historical data, or rerun only failed tasks to minimize cost and time.

The Airflow platform is highly customizable. With the *Public Interface of Airflow* you can extend and adapt nearly every part of the system — from operators to UI plugins to execution logic.

Because Airflow is open source, you're building on components developed, tested, and maintained by a global community. You'll find a wealth of learning resources, including blog posts, books, and conference talks — and you can connect with others via the [community](#), Slack, and mailing lists.

WHY NOT AIRFLOW®?

Airflow® is designed for finite, batch-oriented workflows. While you can trigger dags using the CLI or REST API, Airflow is not intended for continuously running, event-driven, or streaming workloads. That said, Airflow often complements streaming systems like Apache Kafka. Kafka handles real-time ingestion, writing data to storage. Airflow can then periodically pick up that data and process it in batch.

If you prefer clicking over coding, Airflow might not be the best fit. The web UI simplifies workflow management, and the developer experience is continuously improving, but defining workflows as code is central to how Airflow works — so some coding is always required.

3.1 Quick Start

This quick start guide will help you bootstrap an Airflow standalone instance on your local machine.

 **Note**

Successful installation requires a Python 3 environment. Starting with Airflow 2.7.0, Airflow supports Python 3.9, 3.10, 3.11, and 3.12.

Officially supported installation methods is with ``pip``.

Run `pip install apache-airflow[EXTRAS]==AIRFLOW_VERSION --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-AIRFLOW_VERSION/constraints-PYTHON_VERSION.txt"`, for example `pip install "apache-airflow[celery]==3.0.0" --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-3.0.0/constraints-3.9.txt"` to install Airflow in a reproducible way.

While there have been successes with using other tools like `poetry` or `pip-tools`, they do not share the same workflow as `pip` or `uv` - especially when it comes to constraint vs. requirements management. Installing via `Poetry` or `pip-tools` is not currently supported.

If you wish to install Airflow using those tools you should use the constraint files and convert them to appropriate format and workflow that your tool requires.

This guide will help you quickly set up Apache Airflow using `uv`, a fast and modern tool for managing Python environments and dependencies. `uv` makes the installation process easy and provides a smooth setup experience.

1. Set Airflow Home (optional):

Airflow requires a home directory, and uses `~/airflow` by default, but you can set a different location if you prefer. The `AIRFLOW_HOME` environment variable is used to inform Airflow of the desired location. This step of setting the environment variable should be done before installing Airflow so that the installation process knows where to store the necessary files.

```
export AIRFLOW_HOME=~/airflow
```

2. Install Airflow Using uv:

Install uv: [uv Installation Guide](#)

For creating virtual environment with uv, refer to the documentation here: [Creating and Maintaining Local virtual environment with uv](#)

3. Install Airflow using the constraints file, which is determined based on the URL we pass:

```
AIRFLOW_VERSION=3.0.0

# Extract the version of Python you have installed. If you're currently using a
# Python version that is not supported by Airflow, you may want to set this
# manually.
# See above for supported versions.
PYTHON_VERSION="$(python -c 'import sys; print(f"{sys.version_info.major}.{sys.
#version_info.minor}")')"

CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-$
# {AIRFLOW_VERSION}/constraints-${PYTHON_VERSION}.txt"
# For example this would install 3.0.0 with python 3.9: https://raw.
#githubusercontent.com/apache/airflow/constraints-3.1.0/constraints-3.9.txt

uv pip install "apache-airflow==${AIRFLOW_VERSION}" --constraint "${CONSTRAINT_URL}"
```

4. Run Airflow Standalone:

The `airflow standalone` command initializes the database, creates a user, and starts all components.

```
airflow standalone
```

5. Access the Airflow UI:

Visit `localhost:8080` in your browser and log in with the admin account details shown in the terminal. Enable the `example_bash_operator` DAG in the home page.

Upon running these commands, Airflow will create the `$AIRFLOW_HOME` folder and create the “`airflow.cfg`” file with defaults that will get you going fast. You can override defaults using environment variables, see [Configuration Reference](#). You can inspect the file either in `$AIRFLOW_HOME/airflow.cfg`, or through the UI in the Admin->Configuration menu. The PID file for the webserver will be stored in `$AIRFLOW_HOME/airflow-api-server.pid` or in `/run/airflow/airflow-webserver.pid` if started by systemd.

As you grow and deploy Airflow to production, you will also want to move away from the `standalone` command we use here to running the components separately. You can read more in [Production Deployment](#).

Here are a few commands that will trigger a few task instances. You should be able to see the status of the jobs change in the `example_bash_operator` DAG as you run the commands below.

```
# run your first task instance
airflow tasks test example_bash_operator runme_0 2015-01-01
# run a backfill over 2 days
airflow backfill create --dag-id example_bash_operator \
--from-date 2015-01-01 \
--to-date 2015-01-02
```

If you want to run the individual parts of Airflow manually rather than using the all-in-one `standalone` command, you can instead run:

```
airflow db migrate

airflow users create \
    --username admin \
    --firstname Peter \
    --lastname Parker \
    --role Admin \
    --email spiderman@superhero.org

airflow api-server --port 8080

airflow scheduler

airflow dag-processor

airflow triggerer
```

Note

`airflow users` command is only available when `apache-airflow-providers-fab:auth-manager/index` is enabled.

3.1.1 What's Next?

From this point, you can head to the *Tutorials* section for further examples or the *How-to Guides* section if you're ready to get your hands dirty.

3.2 Installation of Airflow®

- *Using released sources*
- *Using PyPI*
- *Using Production Docker Images*
- *Using Official Airflow Helm Chart*
- *Using Managed Airflow Services*
- *Using 3rd-party images, charts, deployments*
- *Notes about minimum requirements*

3.2.1 Prerequisites

Airflow® is tested with:

- Python: 3.9, 3.10, 3.11, 3.12
- Databases:
 - PostgreSQL: 12, 13, 14, 15, 16
 - MySQL: 8.0, Innovation
 - SQLite: 3.15.0+
- Kubernetes: 1.26, 1.27, 1.28, 1.29, 1.30

While we recommend a minimum of 4GB of memory for Airflow, the actual requirements heavily depend on your chosen deployment.

Warning

Despite significant similarities between MariaDB and MySQL, we DO NOT support MariaDB as a backend for Airflow. There are known problems (for example index handling) between MariaDB and MySQL and we do not test our migration scripts nor application execution on Maria DB. We know there were people who used MariaDB for Airflow and that cause a lot of operational headache for them so we strongly discourage attempts to use MariaDB as a backend and users cannot expect any community support for it because the number of users who tried to use MariaDB for Airflow is very small.

Warning

SQLite is used in Airflow tests. DO NOT use it in production. We recommend using the latest stable version of SQLite for local development.

Warning

Airflow® currently can be run on POSIX-compliant Operating Systems. For development it is regularly tested on fairly modern Linux distributions that our contributors use and recent versions of MacOS. On Windows you can run it via WSL2 (Windows Subsystem for Linux 2) or via Linux Containers. The work to add Windows support is tracked via [#10388](#) but it is not a high priority. You should only use Linux-based distributions as “Production environment” as this is the only environment that is supported. The only distribution that is used in our CI tests and that is used in the [Community managed DockerHub image](#) is [Debian Bookworm](#).

3.2.2 Dependencies

Airflow extra dependencies

The apache-airflow PyPI basic package only installs what's needed to get started. Additional packages can be installed depending on what will be useful in your environment. For instance, if you don't need connectivity with PostgreSQL, you won't have to go through the trouble of installing the `postgres-devel` yum package, or whatever equivalent applies on the distribution you are using.

Most of the extra dependencies are linked to a corresponding provider package. For example “amazon” extra has a corresponding `apache-airflow-providers-amazon` provider package to be installed. When you install Airflow

with such extras, the necessary providers are installed automatically (latest versions from PyPI for those packages). However, you can freely upgrade and install providers independently from the main Airflow installation.

For the list of the extras and what they enable, see: [Reference for package extras](#).

Provider distributions

Unlike Apache Airflow 1.10, the Airflow 2.0 is delivered in multiple, separate, but connected packages. The core of Airflow scheduling system is delivered as `apache-airflow` package and there are around 60 providers which can be installed separately as so called `Airflow providers`. The default Airflow installation doesn't have many integrations and you have to install them yourself.

You can even develop and install your own providers for Airflow. For more information, see: [apache-airflow-providers:index](#)

For the list of the providers and what they enable, see: [apache-airflow-providers:packages-ref](#).

Differences between extras and providers

Just to prevent confusion of extras versus providers: Extras and providers are different things, though many extras are leading to installing providers.

Extras are standard Python setuptools feature that allows to add additional set of dependencies as optional features to “core” Apache Airflow. One type of such optional features is providers packages, but not all optional features of Apache Airflow have corresponding providers.

We are using the `extras` setuptools features to also install providers. Most of the extras are also linked (same name) with providers - for example adding `[google]` extra also adds `apache-airflow-providers-google` as dependency. However, there are some extras that do not install providers (examples `github_enterprise`, `kerberos`, `async` - they add some extra dependencies which are needed for those `extra` features of Airflow mentioned. The three examples above add respectively GitHub Enterprise OAuth authentication, Kerberos integration or asynchronous workers for Gunicorn. None of those have providers, they are just extending Apache Airflow “core” package with new functionalities.

System dependencies

You need certain system level requirements in order to install Airflow. Those are requirements that are known to be needed for Linux Debian distributions:

Debian Bookworm (12)

Debian Bookworm is our platform of choice for development and testing. It is the most up-to-date Debian distribution and it is the one we use for our CI/CD system. It is also the one we recommend for development and testing as well as production use.

```
sudo apt install -y --no-install-recommends apt-utils ca-certificates \
curl dumb-init freetds-bin krb5-user libgeos-dev \
ldap-utils libsasl2-2 libsasl2-modules libxmlsec1 locales libffi8 libldap-2.5-0 \
libssl3 netcat-openbsd \
lsb-release openssh-client python3-selinux rsync sasl2-bin sqlite3 sudo unixodbc
```

3.2.3 Supported versions

Version Life Cycle

Apache Airflow® version life cycle:

| Version | Current Patch/Minor | State | First Release | Limited Maintenance | EOL/Terminated |
|---------|---------------------|-----------|---------------|---------------------|----------------|
| 3 | 3.0.0 | Supported | Apr 22, 2025 | TBD | TBD |
| 2 | 2.10.5 | Supported | Dec 17, 2020 | TBD | TBD |
| 1.10 | 1.10.15 | EOL | Aug 27, 2018 | Dec 17, 2020 | June 17, 2021 |
| 1.9 | 1.9.0 | EOL | Jan 03, 2018 | Aug 27, 2018 | Aug 27, 2018 |
| 1.8 | 1.8.2 | EOL | Mar 19, 2017 | Jan 03, 2018 | Jan 03, 2018 |
| 1.7 | 1.7.1.2 | EOL | Mar 28, 2016 | Mar 19, 2017 | Mar 19, 2017 |

Limited support versions will be supported with security and critical bug fixes only. EOL versions will not get any fixes or support. We **highly** recommend installing the latest Airflow release which has richer features.

Support for Python and Kubernetes versions

For Airflow 2.0+ versions, we agreed to certain rules we follow for Python and Kubernetes support. They are based on the official release schedule of Python and Kubernetes, nicely summarized in the [Python Developer's Guide](#) and [Kubernetes version skew policy](#).

1. We drop support for Python and Kubernetes versions when they reach EOL. We drop support for those EOL versions in main right after the EOL date, and it is effectively removed when we release the first new MINOR (Or MAJOR if there is no new MINOR version) of Airflow. For example for Python 3.6 it means that we drop support in main right after 23.12.2021, and the first MAJOR or MINOR version of Airflow released after will not have it.
2. The “oldest” supported version of Python/Kubernetes is the default one. “Default” is only meaningful in terms of “smoke tests” in CI PRs which are run using this default version and default reference image available in Docker Hub. Currently the `apache/airflow:latest` and `apache/airflow:2.10.2` images are Python 3.8 images, however, in the first MINOR/MAJOR release of Airflow released after 2024-10-14, they will become Python 3.9 images.
3. We support a new version of Python/Kubernetes in main after they are officially released, as soon as we make them work in our CI pipeline (which might not be immediate due to dependencies catching up with new versions of Python mostly) we release a new images/support in Airflow based on the working CI setup.

3.2.4 Installing from Sources

Released packages

This page describes downloading and verifying Airflow® version 3.1.0 using officially released packages. You can also install Apache Airflow - as most Python packages - via [PyPI](#). You can choose different version of Airflow by selecting a different version from the drop-down at the top-left of the page.

The `source`, `sdist` and `whl` packages released are the “official” sources of installation that you can use if you want to verify the origin of the packages and want to verify checksums and signatures of the packages. The packages are available via the [Official Apache Software Foundation Downloads](#)

As of version 2.8 Airflow follows PEP 517/518 and uses `pyproject.toml` file to define build dependencies and build process and it requires relatively modern versions of packaging tools to get Airflow built from local sources or `sdist` packages, as PEP 517 compliant build hooks are used to determine dynamic build dependencies. In case of `pip`, it means that at least version 22.1.0 is needed (released at the beginning of 2022) to build or install Airflow from sources. This does not affect the ability of installing Airflow from released wheel packages.

The 3.1.0 downloads of Airflow® are available at:

- Sources package for airflow (asc, sha512)

- Sdist package for airflow meta package (asc, sha512)
- Whl package for airflow meta package (asc, sha512)
- Sdist package for airflow core package (asc, sha512)
- Whl package for airflow core package (asc, sha512)

If you want to install from the source code, you can download from the sources link above, it will contain a `INSTALL` file containing details on how you can build and install Airflow.

Release integrity

PGP signatures KEYS

It is essential that you verify the integrity of the downloaded files using the PGP or SHA signatures. The PGP signatures can be verified using GPG or PGP. Please download the KEYS as well as the asc signature files for relevant distribution. It is recommended to get these files from the main distribution directory and not from the mirrors.

```
gpg -i KEYS
```

or

```
pgpk -a KEYS
```

or

```
pgp -ka KEYS
```

To verify the binaries/sources you can download the relevant asc files for it from main distribution directory and follow the below guide.

```
gpg --verify apache-airflow-*****.asc apache-airflow-*****
```

or

```
pgpv apache-airflow-*****.asc
```

or

```
pgp apache-airflow-*****.asc
```

Example:

```
$ gpg --verify apache-airflow-3.1.0-source.tar.gz.asc apache-airflow-3.1.0-source.tar.gz
gpg: Signature made Sat 11 Sep 12:49:54 2021 BST
gpg:           using RSA key CDE15C6E4D3A8EC4ECF4BA4B6674E08AD7DE406F
gpg:           issuer "kaxilnaik@apache.org"
gpg: Good signature from "Kaxil Naik <kaxilnaik@apache.org>" [unknown]
gpg:           aka "Kaxil Naik <kaxilnaik@gmail.com>" [unknown]
gpg: WARNING: The key's User ID is not certified with a trusted signature!
gpg:           There is no indication that the signature belongs to the owner.
Primary key fingerprint: CDE1 5C6E 4D3A 8EC4 ECF4 BA4B 6674 E08A D7DE 406F
```

The “Good signature from …” is indication that the signatures are correct. Do not worry about the “not certified with a trusted signature” warning. Most of the certificates used by release managers are self-signed, that’s why you get this warning. By importing the server in the previous step and importing it via ID from KEYS page, you know that this is a valid Key already.

For SHA512 sum check, download the relevant sha512 and run the following:

```
shasum -a 512 apache-airflow--***** | diff - apache-airflow--*****.sha512
```

The SHASUM of the file should match the one provided in .sha512 file.

Example:

```
shasum -a 512 apache-airflow-3.1.0-source.tar.gz | diff - apache-airflow-3.1.0-source.  
tar.gz.sha512
```

Verifying PyPI releases

You can verify the Airflow .whl packages from PyPI by locally downloading the package and signature and SHA sum files with the script below:

```
#!/bin/bash  
AIRFLOW_VERSION="3.1.0"  
airflow_download_dir="$(mktemp -d)"  
pip download --no-deps "apache-airflow==$AIRFLOW_VERSION" --dest "${airflow_download_  
dir}"  
curl "https://downloads.apache.org/airflow/${AIRFLOW_VERSION}/apache_airflow-${AIRFLOW_  
VERSION}-py3-none-any.whl.asc" \  
-L -o "${airflow_download_dir}/apache_airflow-${AIRFLOW_VERSION}-py3-none-any.whl.asc"  
curl "https://downloads.apache.org/airflow/${AIRFLOW_VERSION}/apache_airflow-${AIRFLOW_  
VERSION}-py3-none-any.whl.sha512" \  
-L -o "${airflow_download_dir}/apache_airflow-${AIRFLOW_VERSION}-py3-none-any.whl.  
sha512"  
echo  
echo "Please verify files downloaded to ${airflow_download_dir}"  
ls -la "${airflow_download_dir}"  
echo
```

Once you verify the files following the instructions from previous section, you can remove the temporary folder created.

3.2.5 Installation from PyPI

This page describes installations using the `apache-airflow` package published in PyPI.

Installation tools

Only pip installation is currently officially supported.

Note

While there are some successes with using other tools like `poetry` or `pip-tools`, they do not share the same workflow as pip - especially when it comes to constraint vs. requirements management. Installing via Poetry or pip-tools is not currently supported. If you wish to install Airflow using those tools you should use the constraints and convert them to appropriate format and workflow that your tool requires.

Typical command to install Airflow from scratch in a reproducible way from PyPI looks like below:

```
pip install "apache-airflow[celery]==|version|" --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-|version|/constraints-3.9.txt"
```

Typically, you can add other dependencies and providers as separate command after the reproducible installation - this way you can upgrade or downgrade the dependencies as you see fit, without limiting them to constraints. Good practice for those is to extend such `pip install` command with the `apache-airflow` pinned to the version you have already installed to make sure it is not accidentally upgraded or downgraded by `pip`.

```
pip install "apache-airflow==|version|" apache-airflow-providers-google==10.1.0
```

Those are just examples, see further for more explanation why those are the best practices.

Note

Generally speaking, Python community established practice is to perform application installation in a virtual environment created with `virtualenv` or `venv` tools. You can also use `uv` or `pipx` to install Airflow in application dedicated virtual environment created for you. There are also other tools that can be used to manage your virtual environment installation and you are free to choose how you are managing the environments. Airflow has no limitation regarding the tool of your choice when it comes to virtual environment.

The only exception where you might consider not using virtual environment is when you are building a container image with only Airflow installed - this is for example how Airflow is installed in the official Container image.

Constraints files

Why we need constraints

Airflow® installation can be tricky because Airflow is both a library and an application.

Libraries usually keep their dependencies open and applications usually pin them, but we should do neither and both at the same time. We decided to keep our dependencies as open as possible (in `pyproject.toml`) so users can install different versions of libraries if needed. This means that from time to time plain `pip install apache-airflow` will not work or will produce an unusable Airflow installation.

Reproducible Airflow installation

In order to have a reproducible installation, we also keep a set of constraint files in the `constraints-main`, `constraints-2-0`, `constraints-2-1` etc. orphan branches and then we create a tag for each released version e.g. `constraints-3.1.0`.

This way, we keep a tested set of dependencies at the moment of release. This provides you with the ability of having the exact same installation of Airflow + providers + dependencies as was known to be working at the moment of release - frozen set of dependencies for that version of Airflow. There is a separate constraints file for each version of Python that Airflow supports.

You can create the URL to the file substituting the variables in the template below.

```
https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_VERSION}/  
constraints-${PYTHON_VERSION}.txt
```

where:

- `AIRFLOW_VERSION` - Airflow version (e.g. `3.1.0`) or `main`, `2-0`, for latest development version
- `PYTHON_VERSION` Python version e.g. `3.9`, `3.10`

The examples below assume that you want to use install Airflow in a reproducible way with the `celery` extra, but you can pick your own set of extras and providers to install.

```
pip install "apache-airflow[celery]==|version|" --constraint "https://raw.  
→githubusercontent.com/apache/airflow/constraints-|version|/constraints-3.9.txt"
```

Note

The reproducible installation guarantees that this initial installation steps will always work for you - providing that you use the right Python version and that you have appropriate Operating System dependencies installed for the providers to be installed. Some of the providers require additional OS dependencies to be installed such as `build-essential` in order to compile libraries, or for example database client libraries in case you install a database provider, etc.. You need to figure out which system dependencies you need when your installation fails and install them before retrying the installation.

Upgrading and installing dependencies (including providers)

The reproducible installation above should not prevent you from being able to upgrade or downgrade providers and other dependencies to other versions

You can, for example, install new versions of providers and dependencies after the release to use the latest version and up-to-date with latest security fixes - even if you do not want upgrade Airflow core version. Or you can downgrade some dependencies or providers if you want to keep previous versions for compatibility reasons. Installing such dependencies should be done without constraints as a separate pip command.

When you do such an upgrade, you should make sure to also add the `apache-airflow` package to the list of packages to install and pin it to the version that you have, otherwise you might end up with a different version of Airflow than you expect because pip can upgrade/downgrade it automatically when performing dependency resolution.

```
pip install "apache-airflow[celery]==|version|" --constraint "https://raw.  
→githubusercontent.com/apache/airflow/constraints-|version|/constraints-3.9.txt"  
pip install "apache-airflow==|version|" apache-airflow-providers-google==10.1.1
```

You can also downgrade or upgrade other dependencies this way - even if they are not compatible with those dependencies that are stored in the original constraints file:

```
pip install "apache-airflow[celery]==|version|" --constraint "https://raw.  
→githubusercontent.com/apache/airflow/constraints-|version|/constraints-3.9.txt"  
pip install "apache-airflow[celery]==|version|" dbt-core==0.20.0
```

Warning

Not all dependencies can be installed this way - you might have dependencies conflicting with basic requirements of Airflow or other dependencies installed in your system. However, by skipping constraints when you install or upgrade dependencies, you give pip a chance to resolve the conflicts for you, while keeping dependencies within the limits that Apache Airflow, providers and other dependencies require. The resulting combination of those dependencies and the set of dependencies that come with the constraints might not be tested before, but it should work in most cases as we usually add requirements, when Airflow depends on particular versions of some dependencies. In cases you cannot install some dependencies in the same environment as Airflow - you can attempt to use other approaches. See *best practices for handling conflicting/complex Python dependencies*

Verifying installed dependencies

You can also always run the `pip check` command to test if the set of your Python packages is consistent and not conflicting.

```
> pip check  
No broken requirements found.
```

When you see such message and the exit code from `pip check` is 0, you can be sure that there are no conflicting dependencies in your environment.

Using your own constraints

When you decide to install your own dependencies, or want to upgrade or downgrade providers, you might want to continue being able to keep reproducible installation of Airflow and those dependencies via a single command. In order to do that, you can produce your own constraints file and use it to install Airflow instead of the one provided by the community.

```
pip install "apache-airflow[celery]==|version|" --constraint "https://raw.  
←githubusercontent.com/apache/airflow/constraints-|version|/constraints-3.9.txt"  
pip install "apache-airflow==|version|" dbt-core==0.20.0  
pip freeze > my-constraints.txt
```

Then you can use it to create reproducible installations of your environment in a single operation via a local constraints file:

```
pip install "apache-airflow[celery]==|version|" --constraint "my-constraints.txt"
```

Similarly as in the case of Airflow original constraints, you can also host your constraints at your own repository or server and use it remotely from there.

Fixing Constraints at release time

The released “versioned” constraints are mostly **fixed** when we release Airflow version and we only update them in exceptional circumstances. For example when we find out that the released constraints might prevent Airflow from being installed consistently from scratch.

In normal circumstances, the constraint files are not going to change if new version of Airflow dependencies are released - not even when those versions contain critical security fixes. The process of Airflow releases is designed around upgrading dependencies automatically where applicable but only when we release a new version of Airflow, not for already released versions.

Between the releases you can upgrade dependencies on your own and you can keep your own constraints updated as described in the previous section.

The easiest way to keep-up with the latest released dependencies is to upgrade to the latest released Airflow version. Whenever we release a new version of Airflow, we upgrade all dependencies to the latest applicable versions and test them together, so if you want to keep up with those tests - staying up-to-date with latest version of Airflow is the easiest way to update those dependencies.

Installation and upgrade scenarios

In order to simplify the installation, we have prepared examples of how to upgrade Airflow and providers.

Installing Airflow® with extras and providers

If you need to install extra dependencies of Airflow®, you can use the script below to make an installation a one-liner (the example below installs Postgres and Google providers, as well as `async` extra).

Note, that it will install the versions of providers that were available at the moment this version of Airflow has been released. You need to run separate pip commands without constraints, if you want to upgrade providers in case they were released afterwards.

Upgrading Airflow together with providers

You can upgrade Airflow together with extras (providers available at the time of the release of Airflow being installed). This will bring `apache-airflow` and all providers to the versions that were released and tested together when the version of Airflow you are installing was released.

Managing providers separately from Airflow core

In order to add new features, implement bug-fixes or simply maintain backwards compatibility, you might need to install, upgrade or downgrade any of the providers - separately from the Airflow Core package. We release providers independently from the core of Airflow, so often new versions of providers are released before Airflow is, also if you do not want yet to upgrade Airflow to the latest version, you might want to install just some (or all) newly released providers separately.

As you saw above, when installing the providers separately, you should not use any constraint files.

If you build your environment automatically, You should run provider's installation as a separate command after Airflow has been installed (usually with constraints). Constraints are only effective during the `pip install` command they were used with.

It is the best practice to install apache-airflow in the same version as the one that comes from the original image. This way you can be sure that pip will not try to downgrade or upgrade apache Airflow while installing other requirements, which might happen in case you try to add a dependency that conflicts with the version of apache-airflow that you are using:

```
pip install "apache-airflow==|version|" "apache-airflow-providers-google==8.0.0"
```

Note

Installing, upgrading, downgrading providers separately is not guaranteed to work with all Airflow versions or other providers. Some providers have minimum-required version of Airflow and some versions of providers might have limits on dependencies that are conflicting with limits of other providers or other dependencies installed. For example google provider before 10.1.0 version had limit of protobuf library $\leq 3.20.0$ while for example google-ads library that is supported by google has requirement for protobuf library ≥ 4 . In such cases installing those two dependencies alongside in a single environment will not work. In such cases you can attempt to use other approaches. See *best practices for handling conflicting/complex Python dependencies*

Managing just Airflow core without providers

If you don't want to install any providers you have, just install or upgrade Apache Airflow, you can simply install Airflow in the version you need. You can use the special `constraints-no-providers` constraints file, which is smaller and limits the dependencies to the core of Airflow only, however this can lead to conflicts if your environment already has some of the dependencies installed in different versions and in case you have other providers installed. This command, however, gives you the latest versions of dependencies compatible with just Airflow core at the moment Airflow was released.

```
AIRFLOW_VERSION=3.1.0
PYTHON_VERSION="$(python -c 'import sys; print(f'{sys.version_info.major}.{sys.version_
info.minor}')')"
# For example: 3.9
CONSTRAINT_URL="https://raw.githubusercontent.com/apache/airflow/constraints-${AIRFLOW_-
VERSION}/constraints-no-providers-${PYTHON_VERSION}.txt"
# For example: https://raw.githubusercontent.com/apache/airflow/constraints-3.1.0/-
constraints-no-providers-3.9.txt
pip install "apache-airflow==$AIRFLOW_VERSION" --constraint "${CONSTRAINT_URL}"
```

Troubleshooting

This section describes how to troubleshoot installation issues with PyPI installation.

The ‘airflow’ command is not recognized

If the `airflow` command is not getting recognized (can happen on Windows when using WSL), then ensure that `~/local/bin` is in your PATH environment variable, and add it in if necessary:

```
PATH=$PATH:~/local/bin
```

You can also start Airflow with `python -m airflow`

Symbol not found: `_Py_GetArgcArgv`

If you see `Symbol not found: _Py_GetArgcArgv` while starting or importing `airflow`, this may mean that you are using an incompatible version of Python. For a homebrew installed version of Python, this is generally caused by using Python in `/usr/local/opt/bin` rather than the Frameworks installation (e.g. for `python 3.9`: `/usr/local/opt/python@3.9/Frameworks/Python.framework/Versions/3.9`).

The crux of the issue is that a library Airflow depends on, `setproctitle`, uses a non-public Python API which is not available from the standard installation `/usr/local/opt/` (which symlinks to a path under `/usr/local/Cellar`).

An easy fix is just to ensure you use a version of Python that has a dylib of the Python library available. For example:

```
# Note: these instructions are for python3.9 but can be loosely modified for other
˓→versions
brew install python@3.9
virtualenv -p /usr/local/opt/python@3.9/Frameworks/Python.framework/Versions/3.9/bin/
˓→python3 .toy-venv
source .toy-venv/bin/activate
pip install apache-airflow
python
>>> import setproctitle
# Success!
```

Alternatively, you can download and install Python directly from the [Python website](#).

3.2.6 Setting up the database

Apache Airflow® requires a database. If you're just experimenting and learning Airflow, you can stick with the default SQLite option. If you don't want to use SQLite, then take a look at *Set up a Database Backend* to setup a different database.

Usually, you need to run `airflow db migrate` in order to create the database schema if it does not exist or migrate to the latest version if it does. You should make sure that Airflow components are not running while the database migration is being executed.

⚠ Warning

Prior to Airflow version 2.7.0, `airflow db upgrade` was used to apply migrations, however, it has been deprecated in favor of `airflow db migrate`.

In some deployments, such as `helm-chart:index`, both initializing and running the database migration is executed automatically when Airflow is upgraded.

Sometimes, after the upgrade, you are also supposed to do some post-migration actions. See [*Upgrading Airflow® to a newer version*](#) for more details about upgrading and doing post-migration actions.

3.2.7 Upgrading Airflow® to a newer version

Why you need to upgrade

Newer Airflow versions can contain database migrations so you must run `airflow db migrate` to migrate your database with the schema changes in the Airflow version you are upgrading to. Don't worry, it's safe to run even if there are no migrations to perform.

What are the changes between Airflow version x and y?

The *release notes* lists the changes that were included in any given Airflow release.

Upgrade preparation - make a backup of DB

It is highly recommended to make a backup of your metadata DB before any migration. If you do not have a "hot backup" capability for your DB, you should do it after shutting down your Airflow instances, so that the backup of your database will be consistent. If you did not make a backup and your migration fails, you might end-up in a half-migrated state and restoring DB from backup and repeating the migration might be the only easy way out. This can for example

be caused by a broken network connection between your CLI and the database while the migration happens, so taking a backup is an important precaution to avoid problems like this.

When you need to upgrade

If you have a custom deployment based on virtualenv or Docker Containers, you usually need to run the DB migrate manually as part of the upgrade process.

In some cases the upgrade happens automatically - it depends if in your deployment, the upgrade is built-in as post-install action. For example when you are using helm-chart:index with post-upgrade hooks enabled, the database upgrade happens automatically right after the new software is installed. Similarly all Airflow-As-A-Service solutions perform the upgrade automatically for you, when you choose to upgrade Airflow via their UI.

How to upgrade

Reinstall Apache Airflow®, specifying the desired new version.

To upgrade a bootstrapped local instance, you can set the AIRFLOW_VERSION environment variable to the intended version prior to rerunning the installation command. Upgrade incrementally by patch version: e.g., if upgrading from version 2.8.2 to 2.8.4, upgrade first to 2.8.3. For more detailed guidance, see [Quick Start](#).

To upgrade a PyPI package, rerun the `pip install` command in your environment using the desired version as a constraint. For more detailed guidance, see [Installation from PyPI](#).

In order to manually migrate the database you should run the `airflow db migrate` command in your environment. It can be run either in your virtual environment or in the containers that give you access to Airflow CLI [Using the Command Line Interface](#) and the database.

Offline SQL migration scripts

If you want to run the upgrade script offline, you can use the `-s` or `--show-sql-only` flag to get the SQL statements that would be executed. You may also specify the starting Airflow version with the `--from-version` flag and the ending Airflow version with the `-n` or `--to-version` flag. This feature is supported in Postgres and MySQL from Airflow 2.0.0 onward.

Sample usage for Airflow version 2.7.0 or greater:

```
airflow db migrate -s --from-version "2.4.3" -n "2.7.3"           airflow db migrate
--show-sql-only --from-version "2.4.3" --to-version "2.7.3"
```

Warning

`airflow db upgrade` has been replaced by `airflow db migrate` since Airflow version 2.7.0 and former has been deprecated.

Handling migration problems

Wrong Encoding in MySQL database

If you are using old Airflow 1.10 as a database created initially either manually or with previous version of MySQL, depending on the original character set of your database, you might have problems with migrating to a newer version of Airflow and your migration might fail with strange errors (“key size too big”, “missing indexes” etc). The next chapter describes how to fix the problem manually.

Why you might get the error? The recommended character set/collation for MySQL 8 database is `utf8mb4` and `utf8mb4_bin` respectively. However, this has been changing in different versions of MySQL and you could have custom created database with a different character set. If your database was created with an old version of Airflow or MySQL, the encoding could have been wrong when the database was created or broken during migration.

Unfortunately, MySQL limits the index key size and with utf8mb4, Airflow index key sizes might be too big for MySQL to handle. Therefore in Airflow we force all the “ID” keys to use utf8 character set (which is equivalent to utf8mb3 in MySQL 8). This limits the size of indexes so that MySQL can handle them.

Here are the steps you can follow to fix it BEFORE you attempt to migrate (but you might also choose to do it your way if you know what you are doing).

Get familiar with the internal Database structure of Airflow which you might find at *ERD Schema of the Database* and list of migrations that you might find in *Reference for Database Migrations*.

1. Make a backup of your database so that you can restore it in case of a mistake.
2. Check which of the tables of yours need fixing. Look at those tables:

```
SHOW CREATE TABLE task_reschedule;
SHOW CREATE TABLE xcom;
SHOW CREATE TABLE task_fail;
SHOW CREATE TABLE rendered_task_instance_fields;
SHOW CREATE TABLE task_instance;
```

Make sure to copy the output. You will need it in the last step. Your dag_id, run_id, task_id and key columns should have utf8 or utf8mb3 character set set explicitly, similar to:

```
``task_id`` varchar(250) CHARACTER SET utf8 COLLATE utf8_bin NOT NULL, # correct
```

or

```
``task_id`` varchar(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_bin NOT NULL, # correct
```

The problem is if your fields have no encoding:

```
``task_id`` varchar(250), # wrong !!
```

or just collation set to utf8mb4:

```
``task_id`` varchar(250) COLLATE utf8mb4_unicode_ci DEFAULT NULL, # wrong !!
```

or character set and collation set to utf8mb4

```
``task_id`` varchar(250) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NOT NULL, # wrong !!
```

You need to fix those fields that have wrong character set/collation set.

3. Drop foreign key indexes for tables you need to modify (you do not need to drop all of them - do it just for those tables that you need to modify). You will need to recreate them in the last step (that's why you need to keep the SHOW CREATE TABLE output from step 2).

```
ALTER TABLE task_reschedule DROP FOREIGN KEY task_reschedule_ti_fkey;
ALTER TABLE xcom DROP FOREIGN KEY xcom_task_instance_fkey;
ALTER TABLE task_fail DROP FOREIGN KEY task_fail_ti_fkey;
ALTER TABLE rendered_task_instance_fields DROP FOREIGN KEY rtif_ti_fkey;
```

4. Modify your ID fields to have correct character set/encoding. Only do that for fields that have wrong encoding (here are all potential commands you might need to use):

```
ALTER TABLE task_instance MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE
˓→utf8mb3_bin;
```

(continues on next page)

(continued from previous page)

```

ALTER TABLE task_reschedule MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE_
↪utf8mb3_bin;

ALTER TABLE rendered_task_instance_fields MODIFY task_id VARCHAR(250) CHARACTER SET_
↪utf8mb3 COLLATE utf8mb3_bin;
ALTER TABLE rendered_task_instance_fields MODIFY dag_id VARCHAR(250) CHARACTER SET_
↪utf8mb3 COLLATE utf8mb3_bin;

ALTER TABLE task_fail MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;
ALTER TABLE task_fail MODIFY dag_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;

ALTER TABLE sla_miss MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;
ALTER TABLE sla_miss MODIFY dag_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;

ALTER TABLE task_map MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;
ALTER TABLE task_map MODIFY dag_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;
ALTER TABLE task_map MODIFY run_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_
↪bin;

ALTER TABLE xcom MODIFY task_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_bin;
ALTER TABLE xcom MODIFY dag_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_bin;
ALTER TABLE xcom MODIFY run_id VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_bin;
ALTER TABLE xcom MODIFY key VARCHAR(250) CHARACTER SET utf8mb3 COLLATE utf8mb3_bin;

```

5. Recreate the foreign keys dropped in step 3.

Repeat this one for all the indexes you dropped. Note that depending on the version of Airflow you Have, the indexes might be slightly different (for example `map_index` was added in 2.3.0) but if you keep the `SHOW CREATE TABLE` output prepared in step 2., you will find the right `CONSTRAINT_NAME` and `CONSTRAINT` to use.

```
# Here you have to copy the statements from SHOW CREATE TABLE output
ALTER TABLE <TABLE> ADD CONSTRAINT `<CONSTRAINT_NAME>` <CONSTRAINT>
```

This should bring the database to the state where you will be able to run the migration to the new Airflow version.

Post-upgrade warnings

Typically you just need to successfully run `airflow db migrate` command and this is all. However, in some cases, the migration might find some old, stale and probably wrong data in your database and moves it aside to a separate table. In this case you might get warning in your webserver UI about the data found.

Typical message that you might see:

Airflow found incompatible data in the <original table> table in the metadatabase, and has moved them to <new table> during the database migration to upgrade. Please inspect the moved data to decide whether you need to keep them, and manually drop the <new table> table to dismiss this warning.

When you see such message, it means that some of your data was corrupted and you should inspect it to determine whether you would like to keep or delete some of that data. Most likely the data was corrupted and left-over from some

bugs and can be safely deleted - because this data would not be anyhow visible and useful in Airflow. However, if you have particular need for auditing or historical reasons you might choose to store it somewhere. Unless you have specific reasons to keep the data most likely deleting it is your best option.

There are various ways you can inspect and delete the data - if you have direct access to the database using your own tools (often graphical tools showing the database objects), you can drop such table or rename it or move it to another database using those tools. If you don't have such tools you can use the `airflow db shell` command - this will drop you in the db shell tool for your database and you will be able to both inspect and delete the table.

How to drop the table using Kubernetes:

1. Exec into any of the Airflow pods - webserver or scheduler: `kubectl exec -it <your-webserver-pod> python`
2. Run the following commands in the python shell:

```
from airflow.settings import Session

session = Session()
session.execute("DROP TABLE _airflow_moved__2_2__task_instance")
session.commit()
```

Please replace `<table>` in the examples with the actual table name as printed in the warning message.

Inspecting a table:

```
SELECT * FROM <table>;
```

Deleting a table:

```
DROP TABLE <table>;
```

Migration best practices

Depending on the size of your database and the actual migration it might take quite some time to migrate it, so if you have long history and big database, it is recommended to make a copy of the database first and perform a test migration to assess how long the migration will take. Typically “Major” upgrades might take longer as adding new features require sometimes restructuring of the database.

3.2.8 Upgrading to Airflow 3

Apache Airflow 3 is a major release and contains *breaking changes*. This guide walks you through the steps required to upgrade from Airflow 2.x to Airflow 3.0.

Step 1: Take care of prerequisites

- Make sure that you are on Airflow 2.7 or later. It is recommended to upgrade to latest 2.x and then to Airflow 3.
- Make sure that your Python version is in the supported list. Airflow 3.0.0 supports the following Python versions: Python 3.9, 3.10, 3.11 and 3.12.
- Ensure that you are not using any features or functionality that have been *removed in Airflow 3*.

Step 2: Clean and back up your existing Airflow Instance

- It is highly recommended that you make a backup of your Airflow instance, specifically your Airflow metadata database before starting the migration process.
 - If you do not have a “hot backup” capability for your database, you should do it after shutting down your Airflow instances, so that the backup of your database will be consistent. For example, if you don’t turn off your Airflow instance, the backup of the database will not include all TaskInstances or DagRuns.
 - If you did not make a backup and your migration fails, you might end up in a half-migrated state. This can be caused by, for example, a broken network connection between your Airflow CLI and the database during the migration. Having a backup is an important precaution to avoid problems like this.
- A long running Airflow instance can accumulate a substantial amount of data that are no longer required (for example, old XCom data). Schema changes will be a part of the Airflow 3 upgrade process. These schema changes can take a long time if the database is large. For a faster, safer migration, we recommend that you clean up your Airflow meta-database before the upgrade. You can use the `airflow db clean` Airflow CLI command to trim your Airflow database.
- Ensure that there are no errors related to dag processing, such as `AirflowDagDuplicatedIdException`. You should be able to run `airflow dags reserialize` with no errors. If you have have to resolve errors from dag processing, ensure you deploy your changes to your old instance prior to upgrade, and wait until your dags have all been reprocessed (and all errors gone) before you proceed with upgrade.

Step 3: DAG Authors - Check your Airflow DAGs for compatibility

To minimize friction for users upgrading from prior versions of Airflow, we have created a dag upgrade check utility using [Ruff](#).

The latest available `ruff` version will have the most up-to-date rules, but be sure to use at least version `0.11.6`. The below example demonstrates how to check for dag incompatibilities that will need to be fixed before they will work as expected on Airflow 3.

```
ruff check dag/ --select AIR301 --preview
```

To preview the recommended fixes, run the following command:

```
ruff check dag/ --select AIR301 --show-fixes --preview
```

Some changes can be automatically fixed. To do so, run the following command:

```
ruff check dag/ --select AIR301 --fix --preview
```

You can also configure these flags through configuration files. See [Configuring Ruff](#) for details.

Step 4: Install the Standard Providers

- Some of the commonly used Operators which were bundled as part of the `airflow-core` package (for example `BashOperator` and `PythonOperator`) have now been split out into a separate package: `apache-airflow-providers-standard`.
- For convenience, this package can also be installed on Airflow 2.x versions, so that DAGs can be modified to reference these Operators from the standard provider package instead of Airflow Core.

Step 5: Deployment Managers - Upgrade your Airflow Instance

For an easier and safer upgrade process, we have also created a utility to upgrade your Airflow instance configuration. The first step is to run this configuration check utility as shown below:

```
airflow config update
```

This configuration utility can also update your configuration to automatically be compatible with Airflow 3. This can be done as shown below:

```
airflow config update --fix
```

The biggest part of an Airflow upgrade is the database upgrade. The database upgrade process for Airflow 3 is the same as for Airflow 2.7 or later:

```
airflow db migrate
```

If you have plugins that use Flask-AppBuilder views (`appbuilder_views`), Flask-AppBuilder menu items (`appbuilder_menu_items`), or Flask blueprints (`flask_blueprints`), you will either need to convert them to FastAPI apps or ensure you install the FAB provider which provides a backwards compatibility layer for Airflow 3. Ideally, you should convert your plugins to FastAPI apps (`fastapi_apps`), as the compatibility layer in the FAB provider is deprecated.

Step 6: Changes to your startup scripts

In Airflow 3, the Webserver has become a generic API server. The API server can be started up using the following command:

```
airflow api-server
```

The dag processor must now be started independently, even for local or development setups:

```
airflow dag-processor
```

You should now be able to start up your Airflow 3 instance.

3.2.9 Breaking Changes

Some capabilities which were deprecated in Airflow 2.x are not available in Airflow 3. These include:

- **SubDAGs**: Replaced by TaskGroups, Assets, and Data Aware Scheduling.
- **Sequential Executor**: Replaced by LocalExecutor, which can be used with SQLite for local development use cases.
- **CeleryKubernetesExecutor and LocalKubernetesExecutor**: Replaced by [Multiple Executor Configuration](#)
- **SLAs**: Deprecated and removed; Will be replaced by forthcoming [Deadline Alerts](#).
- **Subdir**: Used as an argument on many CLI commands, `--subdir` or `-S` has been superseded by *DAG bundles*.
- **REST API (/api/v1)** replaced: Use the modern FastAPI-based stable `/api/v2` instead; see *Airflow API v2* for details.
- **Some Airflow context variables**: The following keys are no longer available in a *task instance's context*. If not replaced, will cause dag errors:
- `tomorrow_ds` - `tomorrow_ds_nodash`
- `yesterday_ds` - `yesterday_ds_nodash` - `prev_ds` - `prev_ds_nodash` - `prev_execution_date`
- `prev_execution_date_success` - `next_execution_date` - `next_ds_nodash` - `next_ds` - `execution_date`

- The `catchup_by_default` dag parameter is now `False` by default.
- The `create_cron_data_intervals` configuration is now `False` by default. This means that the `CronTriggerTimetable` will be used by default instead of the `CronDataIntervalTimetable`
- **Simple Auth** is now default `auth_manager`. To continue using FAB as the Auth Manager, please install the FAB provider and set `auth_manager` to `FabAuthManager`:

`airflow.providers.fab.auth_manager.fab_auth_manager.FabAuthManager`

This page describes installation options that you might use when considering how to install Airflow®. Airflow consists of many components, often distributed among many physical or virtual machines, therefore installation of Airflow might be quite complex, depending on the options you choose.

You should also check out the [Prerequisites](#) that must be fulfilled when installing Airflow as well as [Supported versions](#) to know what are the policies for the supporting Airflow, Python and Kubernetes.

Airflow requires additional [Dependencies](#) to be installed - which can be done via extras and providers.

When you install Airflow, you need to [setup the database](#) which must also be kept updated when Airflow is upgraded.

3.2.10 Using released sources

More details: [Installing from Sources](#)

When this option works best

- This option is best if you expect to build all your software from sources.
- Apache Airflow is one of the projects that belong to the [Apache Software Foundation](#). It is a requirement for all ASF projects that they can be installed using official sources released via [Official Apache Downloads](#).
- This is the best choice if you have a strong need to [verify the integrity](#) and provenance of the software

Intended users

- Users who are familiar with installing and building software from sources and are conscious about integrity and provenance of the software they use down to the lowest level possible.

What are you expected to handle

- You are expected to build and install Airflow and its components on your own.
- You should develop and handle the deployment for all components of Airflow.
- You are responsible for setting up the database, creating and managing database schema with `airflow db` commands, automated startup and recovery, maintenance, cleanup and upgrades of Airflow and the Airflow Providers.
- You need to setup monitoring of your system allowing you to observe resources and react to problems.
- You are expected to configure and manage appropriate resources for the installation (memory, CPU, etc) based on the monitoring of your installation and feedback loop. See the notes about requirements.

What Apache Airflow Community provides for that method

- You have [instructions](#) on how to build the software but due to various environments and tools you might want to use, you might expect that there will be problems which are specific to your deployment and environment you will have to diagnose and solve.

Where to ask for help

- The `#user-troubleshooting` channel on Slack can be used for quick general troubleshooting questions. The [GitHub discussions](#) if you look for longer discussion and have more information to share.

- The `#user-best-practices` channel on Slack can be used to ask for and share best practices on using and deploying Airflow.
- If you can provide description of a reproducible problem with Airflow software, you can open issue at [GitHub issues](#)
- If you want to contribute back to Airflow, the `#contributors` Slack channel for building the Airflow itself

3.2.11 Using PyPI

More details: [*Installation from PyPI*](#)

When this option works best

- This installation method is useful when you are not familiar with Containers and Docker and want to install Apache Airflow on physical or virtual machines and you are used to installing and running software using custom deployment mechanisms.
- The only officially supported mechanism of installation is via `pip` using constraint mechanisms. The constraint files are managed by Apache Airflow release managers to make sure that you can repeatedly install Airflow from PyPI with all Providers and required dependencies.
- In case of PyPI installation you could also verify integrity and provenance of the packages downloaded from PyPI as described at the installation page, but software you download from PyPI is pre-built for you so that you can install it without building, and you do not build the software from sources.

Intended users

- Users who are familiar with installing and configuring Python applications, managing Python environments, dependencies and running software with their custom deployment mechanisms.

What are you expected to handle

- You are expected to install Airflow - all components of it - on your own.
- You should develop and handle the deployment for all components of Airflow.
- You are responsible for setting up the database, creating and managing database schema with `airflow db` commands, automated startup and recovery, maintenance, cleanup and upgrades of Airflow and Airflow Providers.
- You need to setup monitoring of your system allowing you to observe resources and react to problems.
- You are expected to configure and manage appropriate resources for the installation (memory, CPU, etc) based on the monitoring of your installation and feedback loop.

What Apache Airflow Community provides for that method

- You have [*Installation from PyPI*](#) on how to install the software but due to various environments and tools you might want to use, you might expect that there will be problems which are specific to your deployment and environment you will have to diagnose and solve.
- You have [*Quick Start*](#) where you can see an example of Quick Start with running Airflow locally which you can use to start Airflow quickly for local testing and development. However, this is just for inspiration. Do not expect [*Quick Start*](#) is ready for production installation, you need to build your own production-ready deployment if you follow this approach.

Where to ask for help

- The `#user-troubleshooting` channel on Airflow Slack for quick general troubleshooting questions. The [GitHub discussions](#) if you look for longer discussion and have more information to share.
- The `#user-best-practices` channel on Slack can be used to ask for and share best practices on using and deploying Airflow.

- If you can provide description of a reproducible problem with Airflow software, you can open issue at [GitHub issues](#)

3.2.12 Using Production Docker Images

More details: docker-stack:index

When this option works best

This installation method is useful when you are familiar with Container/Docker stack. It provides a capability of running Airflow components in isolation from other software running on the same physical or virtual machines with easy maintenance of dependencies.

The images are built by Apache Airflow release managers and they use officially released packages from PyPI and official constraint files - same that are used for installing Airflow from PyPI.

Intended users

- Users who are familiar with Containers and Docker stack and understand how to build their own container images.
- Users who understand how to install providers and dependencies from PyPI with constraints if they want to extend or customize the image.
- Users who know how to create deployments using Docker by linking together multiple Docker containers and maintaining such deployments.

What are you expected to handle

- You are expected to be able to customize or extend Container/Docker images if you want to add extra dependencies. You are expected to put together a deployment built of several containers (for example using `docker-compose`) and to make sure that they are linked together.
- You are responsible for setting up the database, creating and managing database schema with `airflow db` commands, automated startup and recovery, maintenance, cleanup and upgrades of Airflow and the Airflow Providers.
- You are responsible to manage your own customizations and extensions for your custom dependencies. With the Official Airflow Docker Images, upgrades of Airflow and Airflow Providers which are part of the reference image are handled by the community - you need to make sure to pick up those changes when released by upgrading the base image. However, you are responsible for creating a pipeline of building your own custom images with your own added dependencies and Providers and need to repeat the customization step and building your own image when new version of Airflow image is released.
- You should choose the right deployment mechanism. There are a number of available options of deployments of containers. You can use your own custom mechanism, custom Kubernetes deployments, custom Docker Compose, custom Helm charts etc., and you should choose it based on your experience and expectations.
- You need to setup monitoring of your system allowing you to observe resources and react to problems.
- You are expected to configure and manage appropriate resources for the installation (memory, CPU, etc) based on the monitoring of your installation and feedback loop.

What Apache Airflow Community provides for that method

- You have instructions: docker-stack:build on how to build and customize your image.
- You have *Running Airflow in Docker* where you can see an example of Quick Start which you can use to start Airflow quickly for local testing and development. However, this is just for inspiration. Do not expect to use this `docker-compose.yml` file for production installation, you need to get familiar with Docker Compose and its capabilities and build your own production-ready deployment with it if you choose Docker Compose for your deployment.
- The Docker Image is managed by the same people who build Airflow, and they are committed to keep it updated whenever new features and capabilities of Airflow are released.

Where to ask for help

- For quick questions with the Official Docker Image there is the [#production-docker-image](#) channel in Airflow Slack.
- The [#user-troubleshooting](#) channel on Airflow Slack for quick general troubleshooting questions. The [GitHub discussions](#) if you look for longer discussion and have more information to share.
- The [#user-best-practices](#) channel on Slack can be used to ask for and share best practices on using and deploying Airflow.
- If you can provide description of a reproducible problem with Airflow software, you can open issue at [GitHub issues](#)

3.2.13 Using Official Airflow Helm Chart

More details: [helm-chart:index](#)

When this option works best

- This installation method is useful when you are not only familiar with Container/Docker stack but also when you use Kubernetes and want to install and maintain Airflow using the community-managed Kubernetes installation mechanism via Helm chart.
- It provides not only a capability of running Airflow components in isolation from other software running on the same physical or virtual machines and managing dependencies, but also it provides capabilities of easier maintaining, configuring and upgrading Airflow in the way that is standardized and will be maintained by the community.
- The Chart uses the Official Airflow Production Docker Images to run Airflow.

Intended users

- Users who are familiar with Containers and Docker stack and understand how to build their own container images.
- Users who understand how to install providers and dependencies from PyPI with constraints if they want to extend or customize the image.
- Users who manage their infrastructure using Kubernetes and manage their applications on Kubernetes using Helm Charts.

What are you expected to handle

- You are expected to be able to customize or extend Container/Docker images if you want to add extra dependencies. You are expected to put together a deployment built of several containers (for example using Docker Compose) and to make sure that they are linked together.
- You are responsible for setting up database.
- The Helm Chart manages your database schema, automates startup, recovery and restarts of the components of the application and linking them together, so you do not have to worry about that.
- You are responsible to manage your own customizations and extensions for your custom dependencies. With the Official Airflow Docker Images, upgrades of Airflow and Airflow Providers which are part of the reference image are handled by the community - you need to make sure to pick up those changes when released by upgrading the base image. However, you are responsible for creating a pipeline of building your own custom images with your own added dependencies and Providers and need to repeat the customization step and building your own image when new version of Airflow image is released.
- You need to setup monitoring of your system allowing you to observe resources and react to problems.
- You are expected to configure and manage appropriate resources for the installation (memory, CPU, etc) based on the monitoring of your installation and feedback loop.

What Apache Airflow Community provides for that method

- You have instructions: docker-stack:build on how to build and customize your image.
- You have helm-chart:index - full documentation on how to configure and install the Helm Chart.
- The Helm Chart is managed by the same people who build Airflow, and they are committed to keep it updated whenever new features and capabilities of Airflow are released.

Where to ask for help

- For quick questions with the Official Docker Image there is the #production-docker-image channel in Airflow Slack.
- For quick questions with the official Helm Chart there is the #helm-chart-official channel in Slack.
- The #user-troubleshooting channel on Airflow Slack for quick general troubleshooting questions. The [GitHub discussions](#) if you look for longer discussion and have more information to share.
- The #user-best-practices channel on Slack can be used to ask for and share best practices on using and deploying Airflow.
- If you can provide description of a reproducible problem with Airflow software, you can open issue at [GitHub issues](#)

3.2.14 Using Managed Airflow Services

Follow the [Ecosystem](#) page to find all Managed Services for Airflow.

When this option works best

- When you prefer to have someone else manage Airflow installation for you, there are Managed Airflow Services that you can use.

Intended users

- Users who prefer to get Airflow managed for them and want to pay for it.

What are you expected to handle

- The Managed Services usually provide everything you need to run Airflow. Please refer to documentation of the Managed Services for details.

What Apache Airflow Community provides for that method

- Airflow Community does not provide any specific documentation for managed services. Please refer to the documentation of the Managed Services for details.

Where to ask for help

- Your first choice should be support that is provided by the Managed services. There are a few channels in the Apache Airflow Slack that are dedicated to different groups of users and if you have come to conclusion the question is more related to Airflow than the managed service, you can use those channels.

3.2.15 Using 3rd-party images, charts, deployments

Follow the [Ecosystem](#) page to find all 3rd-party deployment options.

When this option works best

- Those installation methods are useful in case none of the official methods mentioned before work for you, or you have historically used those. It is recommended though that whenever you consider any change, you should consider switching to one of the methods that are officially supported by the Apache Airflow Community or Managed Services.

Intended users

- Users who historically used other installation methods or find the official methods not sufficient for other reasons.

What are you expected to handle

- Depends on what the 3rd-party provides. Look at the documentation of the 3rd-party.

What Apache Airflow Community provides for that method

- Airflow Community does not provide any specific documentation for 3rd-party methods. Please refer to the documentation of the Managed Services for details.

Where to ask for help

- Depends on what the 3rd-party provides. Look at the documentation of the 3rd-party deployment you use.

3.2.16 Notes about minimum requirements

There are often questions about minimum requirements for Airflow for production systems, but it is not possible to give a simple answer to that question.

The requirements that Airflow might need depend on many factors, including (but not limited to):

- The deployment your Airflow is installed with (see above ways of installing Airflow)
- The requirements of the deployment environment (for example Kubernetes, Docker, Helm, etc.) that are completely independent from Airflow (for example DNS resources, sharing the nodes/resources) with more (or less) pods and containers that are needed that might depend on particular choice of the technology/cloud/integration of monitoring etc.
- Technical details of database, hardware, network, etc. that your deployment is running on
- The complexity of the code you add to your DAGS, configuration, plugins, settings etc. (note, that Airflow runs the code that DAG author and Deployment Manager provide)
- The number and choice of providers you install and use (Airflow has more than 80 providers) that can be installed by choice of the Deployment Manager and using them might require more resources.
- The choice of parameters that you use when tuning Airflow. Airflow has many configuration parameters that can fine-tuned to your needs
- The number of DagRuns and tasks instances you run with parallel instances of each in consideration
- How complex are the tasks you run

The above “DAG” characteristics will change over time and even will change depending on the time of the day or week, so you have to be prepared to continuously monitor the system and adjust the parameters to make it works smoothly.

While we can provide some specific minimum requirements for some development “quick start” - such as in case of our *Running Airflow in Docker* quick-start guide, it is not possible to provide any minimum requirements for production systems.

The best way to think of resource allocation for Airflow instance is to think of it in terms of process control theory - where there are two types of systems:

1. Fully predictable, with few knobs and variables, where you can reliably set the values for the knobs and have an easy way to determine the behaviour of the system
2. Complex systems with multiple variables, that are hard to predict and where you need to monitor the system and adjust the knobs continuously to make sure the system is running smoothly.

Airflow (and generally any modern systems running usually on cloud services, with multiple layers responsible for resources as well multiple parameters to control their behaviour) is a complex system and it fall much more in the second

category. If you decide to run Airflow in production on your own, you should be prepared for the monitor/observe/adjust feedback loop to make sure the system is running smoothly.

Having a good monitoring system that will allow you to monitor the system and adjust the parameters is a must to put that in practice.

There are a few guidelines that you can use for optimizing your resource usage as well. The *Fine-tuning your Scheduler performance* is a good starting point to fine-tune your scheduler, you can also follow the *Best Practices* guide to make sure you are using Airflow in the most efficient way.

Also, one of the important things that Managed Services for Airflow provide is that they make a lot of opinionated choices and fine-tune the system for you, so you don't have to worry about it too much. With such managed services, there are usually far less number of knobs to turn and choices to make and one of the things you pay for is that the Managed Service provider manages the system for you and provides paid support and allows you to scale the system as needed and allocate the right resources - following the choices made there when it comes to the kinds of deployment you might have.

3.3 Security

This section of the documentation covers security-related topics.

Make sure to get familiar with the [Airflow Security Model](#) if you want to understand the different user types of Apache Airflow®, what they have access to, and the role Deployment Managers have in deploying Airflow in a secure way.

Also, if you want to understand how Airflow releases security patches and what to expect from them, head over to [Releasing security patches](#).

Follow the below topics as well to understand other aspects of Airflow security:

3.3.1 Public API

Airflow public API authentication

The Airflow public API uses JWT (JSON Web Token) for authenticating API requests. Each request made to the Airflow API must include a valid JWT token in the `Authorization` header to verify the identity and permissions of the client.

Generate a JWT token

To interact with the Airflow API, clients must first authenticate and obtain a JWT token. The token can be generated by making a POST request to the `/auth/token` endpoint, passing the necessary credentials (e.g., username and password). The `/auth/token` endpoint is provided by the auth manager, therefore, please read the documentation of the auth manager configured in your environment for more details.

- *Generate JWT token with simple auth manager*
- apache-airflow-providers-fab:auth-manager/token

Example

Request

```
ENDPOINT_URL="http://localhost:8080/"
curl -X POST ${ENDPOINT_URL}/auth/token \
```

(continues on next page)

(continued from previous page)

```
-H "Content-Type: application/json" \
-d '{
    "username": "your-username",
    "password": "your-password"
}'
```

Response

```
{  
    "access_token": "<JWT-TOKEN>"  
}
```

Use the JWT token to call Airflow public API

```
ENDPOINT_URL="http://localhost:8080/"
curl -X GET ${ENDPOINT_URL}/api/v2/dags \
-H "Authorization: Bearer <JWT-TOKEN>"
```

Enabling CORS

Cross-origin resource sharing (CORS) is a browser security feature that restricts HTTP requests that are initiated from scripts running in the browser.

`Access-Control-Allow-Headers`, `Access-Control-Allow-Methods`, and `Access-Control-Allow-Origin` headers can be added by setting values for `access_control_allow_headers`, `access_control_allow_methods`, and `access_control_allow_origins` options in the `[api]` section of the `airflow.cfg` file.

```
[api]
access_control_allow_headers = origin, content-type, accept
access_control_allow_methods = POST, GET, OPTIONS, DELETE
access_control_allow_origins = https://exampleclientapp1.com https://exampleclientapp2.
    ↪.com
```

Page size limit

To protect against requests that may lead to application instability, the stable API has a limit of items in response. The default is 100 items, but you can change it using `maximum_page_limit` option in `[api]` section in the `airflow.cfg` file.

3.3.2 Audit Logs in Airflow

Overview

Audit logs are a critical component of any system that needs to maintain a high level of security and compliance. They provide a way to track user actions and system events, which can be used to troubleshoot issues, detect security breaches, and ensure regulatory compliance.

In Airflow, audit logs are used to track user actions and system events that occur during the execution of dags and tasks. They are stored in a database and can be accessed through the Airflow UI.

To be able to see audit logs, a user needs to have the `Audit Logs.can_read` permission. Such user will be able to see all audit logs, independently of the dags permissions applied.

Level of Audit Logs

Audit logs exist at the task level and the user level.

- Task Level: At the task level, audit logs capture information related to the execution of a task, such as the start time, end time, and status of the task.
- User Level: At the user level, audit logs capture information related to user actions, such as creating, modifying, or deleting a DAG or task.

Location of Audit Logs

Audit logs can be accessed through the Airflow UI. They are located under the “Browse” tab, and can be viewed by selecting “Audit Logs” from the dropdown menu.

Types of Events

Airflow provides a set of predefined events that can be tracked in audit logs. These events include, but aren't limited to:

- `trigger`: Triggering a DAG
- `[variable,connection].create`: A user created a Connection or Variable
- `[variable,connection].edit`: A user modified a Connection or Variable
- `[variable,connection].delete`: A user deleted a Connection or Variable
- `delete`: A user deleted a DAG or task
- `failed`: Airflow or a user set a task as failed
- `success`: Airflow or a user set a task as success
- `retry`: Airflow or a user retried a task instance
- `clear`: A user cleared a task's state
- `cli_task_run`: Airflow triggered a task instance

In addition to these predefined events, Airflow allows you to define custom events that can be tracked in audit logs. This can be done by calling the `log` method of the `TaskInstance` object.

3.3.3 Flower

Flower is a web based tool for monitoring and administrating Celery clusters. This topic describes how to configure Airflow to secure your flower instance.

This is an optional component that is disabled by default in Community deployments and you need to configure it on your own if you want to use it.

Flower Authentication

Basic authentication for Celery Flower is supported.

You can specify the details either as an optional argument in the Flower process launching command, or as a configuration item in your `airflow.cfg`. For both cases, please provide `user:password` pairs separated by a comma.

```
airflow celery flower --basic-auth=user1:password1,user2:password2
```

[celery]

```
flower_basic_auth = user1:password1,user2:password2
```

Flower URL Prefix

Enables deploying Celery Flower on non-root URL

For example to access Flower on <http://example.com/flower> run it with:

```
airflow celery flower --url-prefix=flower
```

[celery]

```
flower_url_prefix = flower
```

NOTE: The old nginx rewrite is no longer needed

3.3.4 Kerberos

Airflow has initial support for Kerberos. This means that Airflow can renew Kerberos tickets for itself and store it in the ticket cache. The hooks and dags can make use of ticket to authenticate against kerberized services.

Limitations

Please note that at this time, not all hooks have been adjusted to make use of this functionality. Also it does not integrate Kerberos into the web interface and you will have to rely on network level security for now to make sure your service remains secure.

Celery integration has not been tried and tested yet. However, if you generate a key tab for every host and launch a ticket renewer next to every worker it will most likely work.

Enabling Kerberos

Airflow

To enable Kerberos you will need to generate a (service) key tab.

```
# in the kadmin.local or kadmin shell, create the airflow principal
kadmin: addprinc -randkey airflow/fully.qualified.domain.name@YOUR-REALM.COM

# Create the Airflow keytab file that will contain the Airflow principal
kadmin: xst -norandkey -k airflow.keytab airflow/fully.qualified.domain.name
```

Now store this file in a location where the Airflow user can read it (chmod 600). And then add the following to your airflow.cfg

[core]

```
security = kerberos
```

[kerberos]

```
keytab = /etc/airflow/airflow.keytab
reinit_frequency = 3600
principal = airflow
```

In case you are using Airflow in a docker container based environment, you can set the below environment variables in the Dockerfile instead of modifying airflow.cfg

```
ENV AIRFLOW__CORE__SECURITY kerberos
ENV AIRFLOW__KERBEROS__KEYTAB /etc/airflow/airflow.keytab
ENV AIRFLOW__KERBEROS__INCLUDE_IP False
```

If you need more granular options for your Kerberos ticket the following options are available with the following default values:

```
[kerberos]
# Location of your ccache file once kinit has been performed
ccache = /tmp/airflow_krb5_ccache
# principal gets augmented with fqdn
principal = airflow
reinit_frequency = 3600
kinit_path = kinit
keytab = airflow.keytab

# Allow kerberos token to be flag forwardable or not
forwardable = True

# Allow to include or remove local IP from kerberos token.
# This is particularly useful if you use Airflow inside a VM NATted behind host system
→ IP.
include_ip = True
```

Keep in mind that Kerberos ticket are generated via `kinit` and will use your local `krb5.conf` by default.

Launch the ticket renewer by

```
# run ticket renewer
airflow kerberos
```

To support more advanced deployment models for using kerberos in standard or one-time fashion, you can specify the mode while running the `airflow kerberos` by using the `--one-time` flag.

a) standard: The Airflow kerberos command will run endlessly. The ticket renewer process runs continuously every few seconds and refreshes the ticket if it has expired. b) one-time: The Airflow kerberos will run once and exit. In case of failure the main task won't spin up.

The default mode is standard.

Example usages:

For standard mode:

```
airflow kerberos
```

For one time mode:

```
airflow kerberos --one-time
```

Hadoop

If want to use impersonation this needs to be enabled in `core-site.xml` of your hadoop config.

```
<property>
  <name>hadoop.proxyuser.airflow.groups</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.users</name>
  <value>*</value>
</property>

<property>
  <name>hadoop.proxyuser.airflow.hosts</name>
  <value>*</value>
</property>
```

Of course if you need to tighten your security replace the asterisk with something more appropriate.

Using Kerberos authentication

The Hive hook has been updated to take advantage of Kerberos authentication. To allow your dags to use it, simply update the connection details with, for example:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM"}
```

Adjust the principal to your settings. The _HOST part will be replaced by the fully qualified domain name of the server.

You can specify if you would like to use the DAG owner as the user for the connection or the user specified in the login section of the connection. For the login user, specify the following as extra:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "login"}
```

For the DAG owner use:

```
{ "use_beeline": true, "principal": "hive/_HOST@EXAMPLE.COM", "proxy_user": "owner"}
```

and in your DAG, when initializing the HiveOperator, specify:

```
run_as_owner=True
```

To use kerberos authentication, you must install Airflow with the `kerberos` extras group:

```
pip install 'apache-airflow[kerberos]'
```

You can read about some production aspects of Kerberos deployment at *Kerberos-authenticated workers*

3.3.5 Releasing security patches

Apache Airflow® uses a consistent and predictable approach for releasing security patches - both for the Apache Airflow package and Apache Airflow providers (security patches in providers are treated separately from security patches in Airflow core package).

Releasing Airflow with security patches

Apache Airflow uses a strict [SemVer](#) versioning policy, which means that we strive for any release of a given MAJOR Version (version “2” currently) to be backwards compatible. When we release a MINOR version, the development continues in the main branch where we prepare the next MINOR version, but we release PATCHLEVEL releases with selected bugfixes (including security bugfixes) cherry-picked to the latest released MINOR line of Apache Airflow. At the moment, when we release a new MINOR version, we stop releasing PATCHLEVEL releases for the previous MINOR version.

For example, once we released `2.6.0` version on April 30, 2023 all the security patches will be cherry-picked and released in `2.6.*` versions until we release `2.7.0` version. There will be no `2.5.*` versions released after `2.6.0` has been released.

This means that in order to apply security fixes in Apache Airflow, you MUST upgrade to the latest MINOR and PATCHLEVEL version of Airflow.

Releasing Airflow providers with security patches

Similarly to Airflow, providers uses a strict [SemVer](#) versioning policy, and the same policies apply for providers as for Airflow itself. This means that you need to upgrade to the latest MINOR and PATCHLEVEL version of the provider to get the latest security fixes. Airflow providers are released independently from Airflow itself and the information about vulnerabilities is published separately. You can upgrade providers independently from Airflow itself, following the instructions found in [Managing providers separately from Airflow core](#).

3.3.6 SBOM

Software Bill Of Materials (SBOM) files are critical assets used for transparency, providing a clear inventory of all the components used in Apache Airflow (name, version, supplier and transitive dependencies). They are an exhaustive representation of the software dependencies.

The general use case for such files is to help assess and manage risks. For instance a quick lookup against your SBOM files can help identify if a CVE (Common Vulnerabilities and Exposures) in a library is affecting you.

By default, Apache Airflow SBOM files are generated for Airflow core with all providers. In the near future we aim at generating SBOM files per provider and also provide them for docker standard images.

Each Airflow version has its own SBOM files, one for each supported python version. You can find them [here](#).

3.3.7 Airflow Security Model

This document describes Airflow’s security model from the perspective of the Airflow user. It is intended to help users understand the security model and make informed decisions about how to develop and manage Airflow.

If you would like to know how to report security vulnerabilities and how security reports are handled by the security team of Airflow, head to [Airflow’s Security Policy](#).

Airflow security model - user types

The Airflow security model involves different types of users with varying access and capabilities:

While - in smaller installations - all the actions related to Airflow can be performed by a single user, in larger installations it is apparent that there different responsibilities, roles and capabilities that need to be separated.

This is why Airflow has the following user types:

- Deployment Managers - overall responsible for the Airflow installation, security and configuration

- Authenticated UI users - users that can access Airflow UI and API and interact with it
- DAG Authors - responsible for creating dags and submitting them to Airflow

You can see more on how the user types influence Airflow's architecture in *Architecture Overview*, including, seeing the diagrams of less and more complex deployments.

Deployment Managers

They have the highest level of access and control. They install and configure Airflow, and make decisions about technologies and permissions. They can potentially delete the entire installation and have access to all credentials. Deployment Managers can also decide to keep audits, backups and copies of information outside of Airflow, which are not covered by Airflow's security model.

DAG Authors

They can create, modify, and delete DAG files. The code in DAG files is executed on workers and in the DAG Processor. Therefore, DAG authors can create and change code executed on workers and the DAG Processor and potentially access the credentials that the DAG code uses to access external systems. DAG Authors have full access to the metadata database.

Authenticated UI users

They have access to the UI and API. See below for more details on the capabilities authenticated UI users may have.

Non-authenticated UI users

Airflow doesn't support unauthenticated users by default. If allowed, potential vulnerabilities must be assessed and addressed by the Deployment Manager. However, there are exceptions to this. The `/health` endpoint responsible to get health check updates should be publicly accessible. This is because other systems would want to retrieve that information. Another exception is the `/login` endpoint, as the users are expected to be unauthenticated to use it.

Capabilities of authenticated UI users

The capabilities of **Authenticated UI users** can vary depending on what roles have been configured by the Deployment Manager or Admin users as well as what permissions those roles have. Permissions on roles can be scoped as tightly as a single DAG, for example, or as broad as Admin. Below are four general categories to help conceptualize some of the capabilities authenticated users may have:

Admin users

They manage and grant permissions to other users, with full access to all UI capabilities. They can potentially execute code on workers by configuring connections and need to be trusted not to abuse these privileges. They have access to sensitive credentials and can modify them. By default, they don't have access to system-level configuration. They should be trusted not to misuse sensitive information accessible through connection configuration. They also have the ability to create a API Server Denial of Service situation and should be trusted not to misuse this capability.

Only admin users have access to audit logs.

Operations users

The primary difference between an operator and admin is the ability to manage and grant permissions to other users, and access audit logs - only admins are able to do this. Otherwise assume they have the same access as an admin.

Connection configuration users

They configure connections and potentially execute code on workers during DAG execution. Trust is required to prevent misuse of these privileges. They have full access to sensitive credentials stored in connections and can modify them. Access to sensitive information through connection configuration should be trusted not to be abused. They also have the ability to configure connections wrongly that might create API Server Denial of Service situations and specify insecure connection options which might create situations where executing dags will lead to arbitrary Remote Code Execution for some providers - either community released or custom ones.

Those users should be highly trusted not to misuse this capability.

Audit log users

They can view audit events for the whole Airflow installation.

Regular users

They can view and interact with the UI and API. They are able to view and edit dags, task instances, and DAG runs, and view task logs.

Viewer users

They can view information related to dags, in a read only fashion, task logs, and other relevant details. This role is suitable for users who require read-only access without the ability to trigger or modify dags.

Viewers also do not have permission to access audit logs.

For more information on the capabilities of authenticated UI users, see [apache-airflow-providers-fab:auth-manager/access-control](#).

Capabilities of DAG Authors

DAG authors are able to create or edit code - via Python files placed in a dag bundle - that will be executed in a number of circumstances. The code to execute is neither verified, checked nor sand-boxed by Airflow (that would be very difficult if not impossible to do), so effectively DAG authors can execute arbitrary code on the workers (part of Celery Workers for Celery Executor, local processes run by scheduler in case of Local Executor, Task Kubernetes POD in case of Kubernetes Executor), in the DAG Processor and in the Triggerer.

There are several consequences of this model chosen by Airflow, that deployment managers need to be aware of:

Local executor

In case of Local Executor, DAG authors can execute arbitrary code on the machine where scheduler is running. This means that they can affect the scheduler process itself, and potentially affect the whole Airflow installation - including modifying cluster-wide policies and changing Airflow configuration. If you are running Airflow with Local Executor, the Deployment Manager must trust the DAG authors not to abuse this capability.

Celery Executor

In case of Celery Executor, DAG authors can execute arbitrary code on the Celery Workers. This means that they can potentially influence all the tasks executed on the same worker. If you are running Airflow with Celery Executor, the Deployment Manager must trust the DAG authors not to abuse this capability and unless Deployment Manager separates task execution by queues by Cluster Policies, they should assume, there is no isolation between tasks.

Kubernetes Executor

In case of Kubernetes Executor, DAG authors can execute arbitrary code on the Kubernetes POD they run. Each task is executed in a separate POD, so there is already isolation between tasks as generally speaking Kubernetes provides isolation between PODs.

Triggerer

In case of Triggerer, DAG authors can execute arbitrary code in Triggerer. Currently there are no enforcement mechanisms that would allow to isolate tasks that are using deferrable functionality from each other and arbitrary code from various tasks can be executed in the same process/machine. Deployment Manager must trust that DAG authors will not abuse this capability.

DAG files not needed for Scheduler and API Server

The Deployment Manager might isolate the code execution provided by DAG authors - particularly in Scheduler and API Server by making sure that the Scheduler and API Server don't even have access to the DAG Files. Generally speaking - no DAG author provided code should ever be executed in the Scheduler or API Server process. This means the deployment manager can exclude credentials needed for dag bundles on the Scheduler and API Server - but the bundles must still be configured on those components.

Allowing DAG authors to execute selected code in Scheduler and API Server

There are a number of functionalities that allow the DAG author to use pre-registered custom code to be executed in the Scheduler or API Server process - for example they can choose custom Timetables, UI plugins, Connection UI Fields, Operator extra links, macros, listeners - all of those functionalities allow the DAG author to choose the code that will be executed in the Scheduler or API Server process. However this should not be arbitrary code that DAG author can add dag bundles. All those functionalities are only available via `plugins` and `providers` mechanisms where the code that is executed can only be provided by installed packages (or in case of plugins it can also be added to `PLUGINS` folder where DAG authors should not have write access to). `PLUGINS_FOLDER` is a legacy mechanism coming from Airflow 1.10 - but we recommend using entrypoint mechanism that allows the Deployment Manager to - effectively - choose and register the code that will be executed in those contexts. DAG Author has no access to install or modify packages installed in Scheduler and API Server, and this is the way to prevent the DAG Author to execute arbitrary code in those processes.

Additionally, if you decide to utilize and configure the `PLUGINS_FOLDER`, it is essential for the Deployment Manager to ensure that the DAG author does not have write access to this folder.

The Deployment Manager might decide to introduce additional control mechanisms to prevent DAG authors from executing arbitrary code. This is all fully in hands of the Deployment Manager and it is discussed in the following chapter.

Access to all dags

All dag authors have access to all dags in the Airflow deployment. This means that they can view, modify, and update any dag without restrictions at any time.

Responsibilities of Deployment Managers

As a Deployment Manager, you should be aware of the capabilities of DAG authors and make sure that you trust them not to abuse the capabilities they have. You should also make sure that you have properly configured the Airflow installation to prevent DAG authors from executing arbitrary code in the Scheduler and API Server processes.

Deploying and protecting Airflow installation

Deployment Managers are also responsible for deploying Airflow and make it accessible to the users in the way that follows best practices of secure deployment applicable to the organization where Airflow is deployed. This includes but is not limited to:

- protecting communication using TLS/VPC and whatever network security is required by the organization that is deploying Airflow
- applying rate-limiting and other forms of protections that is usually applied to web applications
- applying authentication and authorization to the web application so that only known and authorized users can have access to Airflow
- any kind of detection of unusual activity and protection against it
- choosing the right session backend and configuring it properly including timeouts for the session

Limiting DAG Author capabilities

The Deployment Manager might also use additional mechanisms to prevent DAG authors from executing arbitrary code - for example they might introduce tooling around DAG submission that would allow to review the code before it is deployed, statically-check it and add other ways to prevent malicious code to be submitted. The way submitting code to a DAG bundle is done and protected is completely up to the Deployment Manager - Airflow does not provide any tooling or mechanisms around it and it expects that the Deployment Manager will provide the tooling to protect access to DAG bundles and make sure that only trusted code is submitted there.

Airflow does not implement any of those feature natively, and delegates it to the deployment managers to deploy all the necessary infrastructure to protect the deployment - as external infrastructure components.

Limiting access for authenticated UI users

Deployment Managers also determine access levels and must understand the potential damage users can cause. Some Deployment Managers may further limit access through fine-grained privileges for the **Authenticated UI users**. However, these limitations are outside the basic Airflow's security model and are at the discretion of Deployment Managers.

Examples of fine-grained access control include (but are not limited to):

- Limiting login permissions: Restricting the accounts that users can log in with, allowing only specific accounts or roles belonging to access the Airflow system.
- Access restrictions to views or dags: Controlling user access to certain views or specific dags, ensuring that users can only view or interact with authorized components.

Future: multi-tenancy isolation

These examples showcase ways in which Deployment Managers can refine and limit user privileges within Airflow, providing tighter control and ensuring that users have access only to the necessary components and functionalities based on their roles and responsibilities. However, fine-grained access control does not provide full isolation and separation of access to allow isolation of different user groups in a multi-tenant fashion yet. In future versions of Airflow, some fine-grained access control features could become part of the Airflow security model, as the Airflow community is working on a multi-tenant model currently.

3.3.8 SQL Injection

Previously, Airflow issued CVE like [CVE-2025-27018 SQL injection in MySQL provider core function](#). The CVE were about the ability to inject SQL without considering the actor performing it. Airflow will no longer issue CVE for cases of SQL Injection unless the reporter can demonstrate a scenario of exploitation. For example, if in a security report the only actor that can operate the injection is Actor who has access to DAGs folder the report will be rejected. When submitting a security report of SQL injection the reporter must explain who is the user that can utilize the injection and how the user gained access to be able to perform it. In simple words, if a user has legit access to write and access the specific DAG, there is no risk that this user will do SQL injection.

3.3.9 Workload

This topic describes how to configure Airflow to secure your workload.

Impersonation

Airflow has the ability to impersonate a unix user while running task instances based on the task's `run_as_user` parameter, which takes a user's name.

NOTE: For impersonations to work, Airflow requires sudo as subtasks are run with `sudo -u` and permissions of files are changed. Furthermore, the unix user needs to exist on the worker. Here is what a simple sudoers file entry could look like to achieve this, assuming Airflow is running as the `airflow` user. This means the Airflow user must be trusted and treated the same way as the root user.

```
airflow ALL=(ALL) NOPASSWD: ALL
```

Subtasks with impersonation will still log to the same folder, except that the files they log to will have permissions changed such that only the unix user can write to it.

Default Impersonation

To prevent tasks that don't use impersonation to be run with `sudo` privileges, you can set the `core:default_impersonation` config which sets a default user impersonate if `run_as_user` is not set.

```
[core]
default_impersonation = airflow
```

3.3.10 Secrets

During Airflow operation, variables or configurations are used that contain particularly sensitive information. This guide provides ways to protect this data.

The following are particularly protected:

- Variables. See the *Variables Concepts* documentation for more information.
- Connections. See the *Connections Concepts* documentation for more information.

Fernet

Airflow uses [Fernet](#) to encrypt passwords in the connection configuration and the variable configuration. It guarantees that a password encrypted using it cannot be manipulated or read without the key. Fernet is an implementation of symmetric (also known as “secret key”) authenticated cryptography.

The first time Airflow is started, the `airflow.cfg` file is generated with the default configuration and the unique Fernet key. The key is saved to option `fernet_key` of section `[core]`.

You can also configure a fernet key using environment variables. This will overwrite the value from the `airflow.cfg` file

```
# Note the double underscores
export AIRFLOW__CORE__FERNET_KEY=your_fernet_key
```

Generating Fernet key

If you need to generate a new fernet key you can use the following code snippet.

```
from cryptography.fernet import Fernet
fernet_key = Fernet.generate_key()
print(fernet_key.decode()) # your fernet_key, keep it in secured place!
```

Rotating encryption keys

Once connection credentials and variables have been encrypted using a fernet key, changing the key will cause decryption of existing credentials to fail. To rotate the fernet key without invalidating existing encrypted values, prepend the new key to the `fernet_key` setting, run `airflow rotate-fernet-key`, and then drop the original key from `fernet_key`:

1. Set `fernet_key` to `new_fernet_key,old_fernet_key`
2. Run `airflow rotate-fernet-key` to re-encrypt existing credentials with the new fernet key
3. Set `fernet_key` to `new_fernet_key`

Secrets Backend

Added in version 1.10.10.

In addition to retrieving connections & variables from environment variables or the metastore database, you can also enable alternative secrets backend to retrieve Airflow connections or Airflow variables via [Apache Airflow Community provided backends](#) in `apache-airflow-providers:core-extensions/secrets-backends`.

Note

The Airflow UI only shows connections and variables stored in the Metadata DB and not via any other method. If you use an alternative secrets backend, check inside your backend to view the values of your variables and connections.

You can also get Airflow configurations with sensitive data from the Secrets Backend. See [Setting Configuration Options](#) for more details.

Search path

When looking up a connection/variable, by default Airflow will search environment variables first and metastore database second.

If you enable an alternative secrets backend, it will be searched first, followed by environment variables, then metastore. This search ordering is not configurable. Though, in some alternative secrets backend you might have the option to filter which connection/variable/config is searched in the secret backend. Please look at the documentation of the secret backend you are using to see if such option is available.

On the other hand, if a workers secrets backend is defined, the order of lookup has higher priority for the workers secrets backend and then the secrets backend.

⚠ Warning

When using environment variables or an alternative secrets backend to store secrets or variables, it is possible to create key collisions. In the event of a duplicated key between backends, all write operations will update the value in the metastore, but all read operations will return the first match for the requested key starting with the custom backend, then the environment variables and finally the metastore.

Configuration

The [secrets] section has the following options:

```
[secrets]
backend =
backend_kwargs =
```

Set `backend` to the fully qualified class name of the backend you want to enable.

You can provide `backend_kwargs` with json and it will be passed as kwargs to the `__init__` method of your secrets backend.

If you want to check which secret backend is currently set, you can use `airflow config get-value secrets backend` command as in the example below.

```
$ airflow config get-value secrets backend
airflow.providers.google.cloud.secrets.secret_manager.CloudSecretManagerBackend
```

Worker Specific Configuration

The above section covers a general configuration option for all Airflow components. But with Airflow 3, if you want to configure separate secrets backend for workers, you can do that using:

```
[workers]
secrets_backend =
secrets_backend_kwargs =
```

Set `secrets_backend` to the fully qualified class name of the backend you want to enable.

You can provide `secrets_backend_kwargs` with json and it will be passed as kwargs to the `__init__` method of your secrets backend for the workers.

If you want to check which secret backend is currently set, you can use `airflow config get-value workers secrets_backend` command as in the example below.

```
$ airflow config get-value workers secrets_backend
airflow.providers.google.cloud.secrets.secret_manager.CloudSecretManagerBackend
```

Supported core backends

Local Filesystem Secrets Backend

This backend is especially useful in the following use cases:

- **Development:** It ensures data synchronization between all terminal windows (same as databases), and at the same time the values are retained after database restart (same as environment variable)
- **Kubernetes:** It allows you to store secrets in [Kubernetes Secrets](#) or you can synchronize values using the sidecar container and [a shared volume](#)

To use variable and connection from local file, specify `LocalFilesystemBackend` as the backend in `[secrets]` section of `airflow.cfg`.

Available parameters to `backend_kwargs`:

- `variables_file_path`: File location with variables data.
- `connections_file_path`: File location with connections data.

Here is a sample configuration:

```
[secrets]
backend = airflow.secrets.local_filesystem.LocalFilesystemBackend
backend_kwargs = {"variables_file_path": "/files/var.json", "connections_file_path": "/
˓→files/conn.json"}
```

JSON, YAML and .env files are supported. All parameters are optional. If the file path is not passed, the backend returns an empty collection.

Storing and Retrieving Connections

If you have set `connections_file_path` as `/files/my_conn.json`, then the backend will read the file `/files/my_conn.json` when it looks for connections.

The file can be defined in JSON, YAML or env format. Depending on the format, the data should be saved as a URL or as a connection object. Any extra json parameters can be provided using keys like `extra_dejson` and `extra`. The key `extra_dejson` can be used to provide parameters as JSON object where as the key `extra` can be used in case of a JSON string. The keys `extra` and `extra_dejson` are mutually exclusive.

The JSON file must contain an object where the key contains the connection ID and the value contains the definition of one connection. The connection can be defined as a URI (string) or JSON object. For a guide about defining a connection as a URI, see [Generating a connection URI](#). For a description of the connection object parameters see [Connection](#). The following is a sample JSON file.

```
{
    "CONN_A": "mysql://host_a",
    "CONN_B": {
        "conn_type": "scheme",
        "host": "host",
        "schema": "schema",
```

(continues on next page)

(continued from previous page)

```
"login": "Login",
"password": "None",
"port": "1234"
}
}
```

The YAML file structure is similar to that of a JSON. The key-value pair of connection ID and the definitions of one or more connections. In this format, the connection can be defined as a URI (string) or JSON object.

```
CONN_A: 'mysql://host_a'
```

```
CONN_B:
```

- 'mysql://host_a'
- 'mysql://host_b'

```
CONN_C:
```

```
  conn_type: scheme
  host: host
  schema: lschema
  login: Login
  password: None
  port: 1234
  extra_dejson:
    a: b
  nestedblock_dict:
    x: y
```

You can also define connections using a .env file. Then the key is the connection ID, and the value should describe the connection using the URL. Connection ID should not be repeated, it will raise an exception. The following is a sample file.

```
mysql_conn_id=mysql://log:password@13.1.21.1:3306/mysql_db
google_custom_key=google-cloud-platform://?key_path=%2Fkeys%2Fkey.json
```

Storing and Retrieving Variables

If you have set `variables_file_path` as `/files/my_var.json`, then the backend will read the file `/files/my_var.json` when it looks for variables.

The file can be defined in JSON, YAML or env format.

The JSON file must contain an object where the key contains the variable key and the value contains the variable value. The following is a sample JSON file.

```
{
  "VAR_A": "some_value",
  "var_b": "different_value"
}
```

The YAML file structure is similar to that of JSON, with key containing the variable key and the value containing the variable value. The following is a sample YAML file.

```
VAR_A: some_value
VAR_B: different_value
```

You can also define variable using a `.env` file. Then the key is the variable key, and variable should describe the variable value. The following is a sample file.

```
VAR_A=some_value
var_B=different_value
```

Apache Airflow Community provided secret backends

Apache Airflow Community also releases community developed providers (`apache-airflow-providers:index`) and some of them also provide handlers that extend secret backends capability of Apache Airflow. You can see all those providers in `apache-airflow-providers:core-extensions/secrets-backends`.

Roll your own secrets backend

A secrets backend is a subclass of `airflow.secrets.base_secrets.BaseSecretsBackend` and must implement either `get_connection()` or `get_conn_value()` for retrieving connections, `get_variable()` for retrieving variables and `get_config()` for retrieving Airflow configurations.

After writing your backend class, provide the fully qualified class name in the `backend` key in the `[secrets]` section of `airflow.cfg`.

Additional arguments to your `SecretsBackend` can be configured in `airflow.cfg` by supplying a JSON string to `backend_kwargs`, which will be passed to the `__init__` of your `SecretsBackend`. See [Configuration](#) for more details, and SSM Parameter Store for an example.

Adapt to non-Airflow compatible secret formats for connections

The default implementation of Secret backend requires use of an Airflow-specific format of storing secrets for connections. Currently most community provided implementations require the connections to be stored as JSON or the Airflow Connection URI format (see `apache-airflow-providers:core-extensions/secrets-backends`). However, some organizations may need to store the credentials (passwords/tokens etc) in some other way. For example, if the same credentials store needs to be used for multiple data platforms, or if you are using a service with a built-in mechanism of rotating the credentials that does not work with the Airflow-specific format. In this case you will need to roll your own secret backend as described in the previous chapter, possibly extending an existing secrets backend and adapting it to the scheme used by your organization.

Masking sensitive data

Airflow will by default mask Connection passwords and sensitive Variables and keys from a Connection's extra (JSON) field when they appear in Task logs, in the Variable and in the Rendered fields views of the UI.

It does this by looking for the specific *value* appearing anywhere in your output. This means that if you have a connection with a password of `a`, then every instance of the letter `a` in your logs will be replaced with `***`.

To disable masking you can set `hide_sensitive_var_conn_fields` to false.

The automatic masking is triggered by Connection or Variable access. This means that if you pass a sensitive value via XCom or any other side-channel it will not be masked when printed in the downstream task.

Sensitive field names

When masking is enabled, Airflow will always mask the password field of every Connection that is accessed by a task.

It will also mask the value of a Variable, rendered template dictionaries, XCom dictionaries or the field of a Connection's extra JSON blob if the name is in the list of known-sensitive fields (i.e. `'access_token'`, `'api_key'`, `'apikey'`, `'authorization'`, `'passphrase'`, `'passwd'`, `'password'`, `'private_key'`, `'secret'` or `'token'`). This list can also be extended:

[core]

```
sensitive_var_conn_names = comma,separated,sensitive,names
```

Adding your own masks

If you want to mask an additional secret that is not already masked by one of the above methods, you can do it in your DAG file or operator's `execute` function using the `mask_secret` function. For example:

```
@task
def my_func():
    from airflow.sdk.execution_time.secrets_masker import mask_secret

    mask_secret("custom_value")

    ...
```

or

```
class MyOperator(BaseOperator):
    def execute(self, context):
        from airflow.sdk.execution_time.secrets_masker import mask_secret

        mask_secret("custom_value")

        ...
```

The mask must be set before any log/output is produced to have any effect.

NOT masking when using environment variables

When you are using some operators - for example `airflow.providers.cncf.kubernetes.operators.pod.KubernetesPodOperator`, you might be tempted to pass secrets via environment variables. This is very bad practice because the environment variables are visible to anyone who has access to see the environment of the process - such secrets passed by environment variables will NOT be masked by Airflow.

If you need to pass secrets to the `KubernetesPodOperator`, you should use native Kubernetes secrets or use Airflow Connection or Variables to retrieve the secrets dynamically.

3.4 Tutorials

Once you have Airflow up and running with the [Quick Start](#), these tutorials are a great way to get a sense for how Airflow works.

3.4.1 Airflow 101: Building Your First Workflow

Welcome to world of Apache Airflow! In this tutorial, we'll guide you through the essential concepts of Airflow, helping you understand how to write your first DAG. Whether you're familiar with Python or just starting out, we'll make the journey enjoyable and straightforward.

What is a DAG?

At its core, a DAG is a collection of tasks organized in a way that reflects their relationships and dependencies. It's like a roadmap for your workflow, showing how each task connects to the others. Don't worry if this sounds a bit complex; we'll break it down step by step.

Example Pipeline definition

Let's start with a simple example of a pipeline definition. Although it might seem overwhelming at first, we'll explain each line in detail.

`airflow/example_dags/tutorial.py`

```
import textwrap
from datetime import datetime, timedelta

# Operators; we need this to operate!
from airflow.providers.standard.operators.bash import BashOperator

# The DAG object; we'll need this to instantiate a DAG
from airflow.sdk import DAG
with DAG(
    "tutorial",
    # These args will get passed on to each operator
    # You can override them on a per-task basis during operator initialization
    default_args={
        "depends_on_past": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
        # 'queue': 'bash_queue',
        # 'pool': 'backfill',
        # 'priority_weight': 10,
        # 'end_date': datetime(2016, 1, 1),
        # 'wait_for_downstream': False,
        # 'execution_timeout': timedelta(seconds=300),
        # 'on_failure_callback': some_function, # or list of functions
        # 'on_success_callback': some_other_function, # or list of functions
        # 'on_retry_callback': another_function, # or list of functions
        # 'sla_miss_callback': yet_another_function, # or list of functions
        # 'on_skipped_callback': another_function, #or list of functions
        # 'trigger_rule': 'all_success'
    },
    description="A simple tutorial DAG",
    schedule=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=["example"],
) as dag:

    # t1, t2 and t3 are examples of tasks created by instantiating operators
    t1 = BashOperator(
        task_id="print_date",
        bash_command="date",
    )
```

(continues on next page)

(continued from previous page)

```

t2 = BashOperator(
    task_id="sleep",
    depends_on_past=False,
    bash_command="sleep 5",
    retries=3,
)
t1.doc_md = textwrap.dedent(
    """
    #### Task Documentation
    You can document your task using the attributes `doc_md` (markdown),
    `doc` (plain text), `doc_rst`, `doc_json`, `doc_yaml` which gets
    rendered in the UI's Task Instance Details page.
    ! [img] (https://imgs.xkcd.com/comics/fixing_problems.png)
    **Image Credit:** Randall Munroe, [XKCD] (https://xkcd.com/license.html)
    """
)
dag.doc_md = __doc__ # providing that you have a docstring at the beginning of the
→DAG; OR
dag.doc_md = """
This is a documentation placed anywhere
"" # otherwise, type it like this
templated_command = textwrap.dedent(
    """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7) }}"
    {% endfor %}
    """
)
t3 = BashOperator(
    task_id="templated",
    depends_on_past=False,
    bash_command=templated_command,
)
t1 >> [t2, t3]

```

Understanding the DAG Definition File

Think of the Airflow Python script as a configuration file that lays out the structure of your DAG in code. The actual tasks you define here run in a different environment, which means this script isn't meant for data processing. It's main job is to define the DAG object, and it needs to evaluate quickly since the DAG File Processor checks it regularly for any changes.

Importing Modules

To get started, we need to import the necessary libraries. This is a typical first step in any Python script.

airflow/example_dags/tutorial.py

```
import textwrap
from datetime import datetime, timedelta

# Operators; we need this to operate!
from airflow.providers.standard.operators.bash import BashOperator

# The DAG object; we'll need this to instantiate a DAG
from airflow.sdk import DAG
```

For more details on how Python and Airflow handle modules, check out *Modules Management*.

Setting Default Arguments

When creating a DAG and its tasks, you can either pass arguments directly to each task or define a set of default parameters in a dictionary. The latter approach is usually more efficient and cleaner.

airflow/example_dags/tutorial.py

```
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args={

    "depends_on_past": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function, # or list of functions
    # 'on_success_callback': some_other_function, # or list of functions
    # 'on_retry_callback': another_function, # or list of functions
    # 'sla_miss_callback': yet_another_function, # or list of functions
    # 'on_skipped_callback': another_function, #or list of functions
    # 'trigger_rule': 'all_success'
},
```

If you want to dive deeper into the parameters of the BaseOperator, take a look at the documentation for `airflow.sdk.BaseOperator` documentation.

Creating a DAG

Next, we'll need to create a DAG object to house our tasks. We'll provide a unique identifier for the DAG, known as the `dag_id`, and specify the default arguments we just defined. We'll also set a schedule for our DAG to run every day.

airflow/example_dags/tutorial.py

```
with DAG(
    "tutorial",
    # These args will get passed on to each operator
    # You can override them on a per-task basis during operator initialization
    default_args={
        "depends_on_past": False,
        "retries": 1,
        "retry_delay": timedelta(minutes=5),
        # 'queue': 'bash_queue',
        # 'pool': 'backfill',
        # 'priority_weight': 10,
        # 'end_date': datetime(2016, 1, 1),
        # 'wait_for_downstream': False,
        # 'execution_timeout': timedelta(seconds=300),
        # 'on_failure_callback': some_function, # or list of functions
        # 'on_success_callback': some_other_function, # or list of functions
        # 'on_retry_callback': another_function, # or list of functions
        # 'sla_miss_callback': yet_another_function, # or list of functions
        # 'on_skipped_callback': another_function, #or list of functions
        # 'trigger_rule': 'all_success'
    },
    description="A simple tutorial DAG",
    schedule=timedelta(days=1),
    start_date=datetime(2021, 1, 1),
    catchup=False,
    tags=["example"],
) as dag:
```

Understanding Operators

An operator represents a unit of work in Airflow. They are the building blocks of your workflows, allowing you to define what tasks will be executed. While we can use operators for many tasks, Airflow also offers the [Taskflow API](#) for a more Pythonic way to define workflows, which we'll touch on later.

All operators derive from the `BaseOperator`, which includes the essential arguments needed to run tasks in Airflow. Some popular operators include the `PythonOperator`, `BashOperator`, and `KubernetesPodOperator`. In this tutorial, we'll focus on the `BashOperator` to execute some simple bash commands.

Defining Tasks

To use an operator, you must instantiate it as a task. Tasks dictate how the operator will perform its work within the DAG's context. In the example below, we instantiate the `BashOperator` twice to run two different bash scripts. The `task_id` serves as a unique identifier for each task.

airflow/example_dags/tutorial.py

```
t1 = BashOperator(
    task_id="print_date",
    bash_command="date",
```

(continues on next page)

(continued from previous page)

```
)  
  
t2 = BashOperator(  
    task_id="sleep",  
    depends_on_past=False,  
    bash_command="sleep 5",  
    retries=3,  
)
```

Notice how we mix operator-specific arguments (like `bash_command`) with common arguments (like `retries`) inherited from `BaseOperator`. This approach simplifies our code. In the second task, we even override the `retries` parameter to set it to 3.

The precedence for task arguments is as follows:

1. Explicitly passed arguments
2. Values from the `default_args` dictionary
3. The operator's default values, if available

Note

Remember, every task must include or inherit the arguments `task_id` and `owner`. Otherwise, Airflow will raise an error. Fortunately, a fresh Airflow installation defaults the `owner` to `airflow`, so you mainly need to ensure `task_id` is set.

Using Jinja for Templating

Airflow harnesses the power of [Jinja Templating](#), giving you access to built-in parameters and macros to enhance your workflows. This section will introduce you to the basics of templating in Airflow, focusing on the commonly used template variable: `{{ ds }}`, which represents today's date stamp.

`airflow/example_dags/tutorial.py`

```
templated_command = textwrap.dedent(  
    """  
    {% for i in range(5) %}  
        echo "{{ ds }}"  
        echo "{{ macros.ds_add(ds, 7)}}"  
    {% endfor %}  
    """  
)  
  
t3 = BashOperator(  
)
```

(continues on next page)

(continued from previous page)

```
task_id="templated",
depends_on_past=False,
bash_command=templated_command,
)
```

You'll notice that the `templated_command` includes logic in `{% %}` blocks and references parameters like `{{ ds }}`. You can also pass files to the `bash_command`, such as `bash_command='templated_command.sh'`, allowing for better organization of your code. You can even define `user_defined_macros` and `user_defined_filters` to create your own variables and filters for use in templates. For more on custom filters, refer to the [Jinja Documentation](#).

For more information on the variables and macros that can be referenced in templates, please read through the *Templates reference*.

Adding DAG and Tasks documentation

You can add documentation to your DAG or individual tasks. While DAG documentation currently supports markdown, task documentation can be in plain text, markdown reStructuredText, JSON, or YAML. It's a good practice to include documentation at the start of your DAG file.

`airflow/example_dags/tutorial.py`

```
t1.doc_md = textwrap.dedent(
    """\
    #### Task Documentation
    You can document your task using the attributes `doc_md` (markdown),
    `doc` (plain text), `doc_rst`, `doc_json`, `doc_yaml` which gets
    rendered in the UI's Task Instance Details page.
    
    **Image Credit:** Randall Munroe, [XKCD](https://xkcd.com/license.html)
    """
)

dag.doc_md = __doc__ # providing that you have a docstring at the beginning of the DAG; ↵
    OR
dag.doc_md = """
This is a documentation placed anywhere
    # otherwise, type it like this
```

The screenshot shows the Apache Airflow UI with the 'Dag' dropdown set to 'tutorial'. In the center, a modal window titled 'Task Documentation' is open for the task 'print_date'. The documentation area contains a list of supported formats: doc_md, (markdown), doc, doc_text, doc_rst, doc_json, and doc_yaml. It also states that this gets rendered in the UI's Task Instance Details page. Below this, there is a small XKCD comic strip featuring a person at a desk with the text: "TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...". At the bottom of the modal, it says 'Image Credit: Randall Munroe, XKCD'.

The screenshot shows the Apache Airflow UI with the 'Dag' dropdown set to 'tutorial'. In the center, a modal window titled 'Dag Documentation' is open. The documentation area contains the text 'This is a documentation placed anywhere'. On the right side of the screen, the DAG Overview is visible, showing the DAG 'tutorial' with a schedule of '1 day, 0:00:00'. Below the overview is a chart titled 'Last 24 hours' showing 'Last 2 Dag Runs' with two bars: one for 2025-04-10, 03:16:58 with a duration of 7.18 seconds, and another for 2025-04-10, 03:17:21 with a duration of 0.54 seconds. The right side of the screen also shows the 'Run Backfill' and 'Reparse Dag' buttons, and the 'Owner' field is set to 'airflow'.

Setting up Dependencies

In Airflow, tasks can depend on one another. For instance, if you have tasks `t1`, `t2`, and `t3`, you can define their dependencies in several ways:

```
t1.set_downstream(t2)

# This means that t2 will depend on t1
# running successfully to run.
# It is equivalent to:
t2.set_upstream(t1)
```

(continues on next page)

(continued from previous page)

```
# The bit shift operator can also be
# used to chain operations:
t1 >> t2

# And the upstream dependency with the
# bit shift operator:
t2 << t1

# Chaining multiple dependencies becomes
# concise with the bit shift operator:
t1 >> t2 >> t3

# A list of tasks can also be set as
# dependencies. These operations
# all have the same effect:
t1.set_downstream([t2, t3])
t1 >> [t2, t3]
[t2, t3] << t1
```

Be mindful that Airflow will raise errors if it detects cycles in your DAG or if a dependency is referenced multiple times.

Working with Time Zones

Creating a time zone aware DAG is straightforward. Just ensure you use time zone aware dates with `pendulum`. Avoid using the standard library `timezone` as they have known limitations.

Recap

Congratulations! By now, you should have a basic understanding of how to create a DAG, define tasks and their dependencies, and use templating in Airflow. Your code should resemble the following:

`airflow/example_dags/tutorial.py`

```
import textwrap
from datetime import datetime, timedelta

# Operators; we need this to operate!
from airflow.providers.standard.operators.bash import BashOperator

# The DAG object; we'll need this to instantiate a DAG
from airflow.sdk import DAG
with DAG(
    "tutorial",
    # These args will get passed on to each operator
    # You can override them on a per-task basis during operator initialization
    default_args={
```

"depends_on_past": False,
"retries": 1,
"retry_delay": timedelta(minutes=5),
'queue': 'bash_queue',
'pool': 'backfill',

(continues on next page)

(continued from previous page)

```

# 'priority_weight': 10,
# 'end_date': datetime(2016, 1, 1),
# 'wait_for_downstream': False,
# 'execution_timeout': timedelta(seconds=300),
# 'on_failure_callback': some_function, # or list of functions
# 'on_success_callback': some_other_function, # or list of functions
# 'on_retry_callback': another_function, # or list of functions
# 'sla_miss_callback': yet_another_function, # or list of functions
# 'on_skipped_callback': another_function, #or list of functions
# 'trigger_rule': 'all_success'
},
description="A simple tutorial DAG",
schedule=timedelta(days=1),
start_date=datetime(2021, 1, 1),
catchup=False,
tags=["example"],
) as dag:

# t1, t2 and t3 are examples of tasks created by instantiating operators
t1 = BashOperator(
    task_id="print_date",
    bash_command="date",
)

t2 = BashOperator(
    task_id="sleep",
    depends_on_past=False,
    bash_command="sleep 5",
    retries=3,
)
t1.doc_md = textwrap.dedent(
    """
    #### Task Documentation
    You can document your task using the attributes `doc_md` (markdown),
    `doc` (plain text), `doc_rst`, `doc_json`, `doc_yaml` which gets
    rendered in the UI's Task Instance Details page.
    
    **Image Credit:** Randall Munroe, [XKCD](https://xkcd.com/license.html)
    """
)

dag.doc_md = __doc__ # providing that you have a docstring at the beginning of the
DAG; OR
dag.doc_md = """
This is a documentation placed anywhere
"""\# otherwise, type it like this
templated_command = textwrap.dedent(
    """
    {% for i in range(5) %}
        echo "{{ ds }}"
        echo "{{ macros.ds_add(ds, 7) }}"
    {% endfor %}
"""
)

```

(continues on next page)

(continued from previous page)

```
    ....  
    )  
  
    t3 = BashOperator(  
        task_id="templated",  
        depends_on_past=False,  
        bash_command=templated_command,  
    )  
  
    t1 >> [t2, t3]
```

Testing Your Pipeline

Now it's time to test your pipeline! First, ensure that your script parses successfully. If you saved your code in `tutorial.py` within the `dags` folder specified in your `airflow.cfg`, you can run:

```
python ~/airflow/dags/tutorial.py
```

If the script runs without errors, congratulations! Your DAG is set up correctly.

Command Line Metadata Validation

Let's validate your script further by running a few commands:

```
# initialize the database tables  
airflow db migrate  
  
# print the list of active dags  
airflow dags list  
  
# prints the list of tasks in the "tutorial" DAG  
airflow tasks list tutorial  
  
# prints the graphviz representation of "tutorial" DAG  
airflow dags show tutorial
```

Testing Task Instances and DAG Runs

You can test specific task instances for a designated *logical date*. This simulates the scheduler running your task for a particular date and time.

Note

Notice that the scheduler runs your task *for* a specific date and time, not necessarily *at* that date or time. The *logical date* is the timestamp that a DAG run is **named after**, and it typically corresponds to the **end** of the time period your workflow is operating on — or the time at which the DAG run was manually triggered.

Airflow uses this logical date to organize and track each run; it's how you refer to a specific execution in the UI, logs, and code. When triggering a DAG via the UI or API, you can supply your own logical date to run the workflow *as of* a specific point in time.

```
# command layout: command subcommand [dag_id] [task_id] [(optional) date]

# testing print_date
airflow tasks test tutorial print_date 2015-06-01

# testing sleep
airflow tasks test tutorial sleep 2015-06-01
```

You can also see how your templates get rendered by running:

```
# testing templated
airflow tasks test tutorial templated 2015-06-01
```

This command will provide detailed logs and execute your bash command.

Keep in mind that the `airflow tasks test` command runs task instances locally, outputs their logs to stdout, and doesn't track state in the database. This is a handy way to test individual task instances.

Similarly, `airflow dags test` runs a single DAG run without registering any state in the database, which is useful for testing your entire DAG locally.

What's Next?

That's a wrap! You've successfully written and tested your first Airflow pipeline. As you continue your journey, consider merging your code into a repository with a Scheduler running against it, which will allow your DAG to be triggered and executed daily.

Here are a few suggestions for your next steps:

See also

- Continue to the next step of the tutorial: [Pythonic DAGs with the TaskFlow API](#)
- Explore the *Core Concepts* section for detailed explanation of Airflow concepts such as DAGs, Tasks, Operators, and more.

3.4.2 Pythonic DAGs with the TaskFlow API

In the first tutorial, you built your first Airflow DAG using traditional Operators like `BashOperator`. Now let's look at a more modern and Pythonic way to write workflows using the **TaskFlow API** — introduced in Airflow 2.0.

The TaskFlow API is designed to make your code simpler, cleaner, and easier to maintain. You write plain Python functions, decorate them, and Airflow handles the rest — including task creation, dependency wiring, and passing data between tasks.

In this tutorial, we'll create a simple ETL pipeline — Extract → Transform → Load using the TaskFlow API. Let's dive in!

The Big Picture: A TaskFlow Pipeline

Here's what the full pipeline looks like using TaskFlow. Don't worry if some of it looks unfamiliar — we'll break it down step-by-step.

`airflow/example_dags/tutorial_taskflow_api.py`

```
import json

import pendulum

from airflow.sdk import dag, task
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def tutorial_taskflow_api():
    """
    ### TaskFlow API Tutorial Documentation
    This is a simple data pipeline example which demonstrates the use of
    the TaskFlow API using three simple tasks for Extract, Transform, and Load.
    Documentation that goes along with the Airflow TaskFlow API tutorial is
    located
    [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html)
    """

    @task()
    def extract():
        """
        #### Extract task
        A simple Extract task to get data ready for the rest of the data
        pipeline. In this case, getting data is simulated by reading from a
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'

        order_data_dict = json.loads(data_string)
        return order_data_dict
    @task(multiple_outputs=True)
    def transform(order_data_dict: dict):
        """
        #### Transform task
        A simple Transform task which takes in the collection of order data and
        computes the total order value.
        """
        total_order_value = 0

        for value in order_data_dict.values():
            total_order_value += value

        return {"total_order_value": total_order_value}
    @task()
    def load(total_order_value: float):
        """
        #### Load task
        A simple Load task which takes in the result of the Transform task and
        instead of saving it to end user review, just prints it out.
        """


```

(continues on next page)

(continued from previous page)

```
"""
    print(f"Total order value is: {total_order_value:.2f}")
order_data = extract()
order_summary = transform(order_data)
load(order_summary["total_order_value"])
tutorial_taskflow_api()
```

Step 1: Define the DAG

Just like before, your DAG is a Python script that Airflow loads and parses. But this time, we're using the `@dag` decorator to define it.

`airflow/example_dags/tutorial_taskflow_api.py`

```
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def tutorial_taskflow_api():
    """
    ## TaskFlow API Tutorial Documentation
    This is a simple data pipeline example which demonstrates the use of
    the TaskFlow API using three simple tasks for Extract, Transform, and Load.
    Documentation that goes along with the Airflow TaskFlow API tutorial is
    located
    [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.
    html)
    """
```

To make this DAG discoverable by Airflow, we can call the Python function that was decorated with `@dag`:

`airflow/example_dags/tutorial_taskflow_api.py`

```
tutorial_taskflow_api()
```

Changed in version 2.4: If you're using the `@dag` decorator or defining your DAG in a `with` block, you no longer need to assign it to a global variable. Airflow will find it automatically.

You can visualize your DAG in the Airflow UI! Once your DAG is loaded, navigate to the Graph View to see how tasks are connected.

Step 2: Write Your Tasks with @task

With Taskflow, each task is just a regular Python function. You can use the `@task` decorator to turn it into a task that Airflow can schedule and run. Here's the `extract` task:

`airflow/example_dags/tutorial_taskflow_api.py`

```
@task()
def extract():
    """
    #### Extract task
    A simple Extract task to get data ready for the rest of the data
    pipeline. In this case, getting data is simulated by reading from a
    hardcoded JSON string.
    """
    data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'

    order_data_dict = json.loads(data_string)
    return order_data_dict
```

The function's return value is passed to the next task — no manual use of XComs required. Under the hood, TaskFlow uses XComs to manage data passing automatically, abstracting away the complexity of manual XCom management from the previous methods. You'll define `transform` and `load` tasks using the same pattern.

Notice the use of `@task(multiple_outputs=True)` above — this tells Airflow that the function returns a dictionary of values that should be split into individual XComs. Each key in the returned dictionary becomes its own XCom entry, which makes it easy to reference specific values in downstream tasks. If you omit `multiple_outputs=True`, the entire dictionary is stored as a single XCom instead, and must be accessed as a whole.

Step 3: Build the Flow

Once the tasks are defined, you can build the pipeline by simply calling them like Python functions. Airflow uses this functional invocation to set task dependencies and manage data passing.

`airflow/example_dags/tutorial_taskflow_api.py`

```
order_data = extract()
order_summary = transform(order_data)
load(order_summary["total_order_value"])
```

That's it! Airflow knows how to schedule and orchestrate your pipeline from this code alone.

Running Your DAG

To enable and trigger your DAG:

1. Navigate to the Airflow UI.
2. Find your DAG in the list and click the toggle to enable it.
3. You can trigger it manually by clicking the “Trigger DAG” button, or wait for it to run on its schedule.

What’s Happening Behind the Scenes?

If you’ve used Airflow 1.x, this probably feels like magic. Let’s compare what’s happening under the hood.

The “Old Way”: Manual Wiring and XComs

Before the TaskFlow API, you had to use Operators like `PythonOperator` and pass data manually between tasks using `XComs`.

Here’s what the same DAG might have looked like using the traditional approach:

```
import json
import pendulum
from airflow.sdk import DAG, PythonOperator

def extract():
    # Old way: simulate extracting data from a JSON string
    data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'
    return json.loads(data_string)

def transform(ti):
    # Old way: manually pull from XCom
    order_data_dict = ti.xcom_pull(task_ids="extract")
    total_order_value = sum(order_data_dict.values())
    return {"total_order_value": total_order_value}

def load(ti):
    # Old way: manually pull from XCom
    total = ti.xcom_pull(task_ids="transform")["total_order_value"]
    print(f"Total order value is: {total:.2f}")

with DAG(
    dag_id="legacy_etl_pipeline",
    schedule_interval=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
) as dag:
    extract_task = PythonOperator(task_id="extract", python_callable=extract)
    transform_task = PythonOperator(task_id="transform", python_callable=transform)
    load_task = PythonOperator(task_id="load", python_callable=load)
```

(continues on next page)

(continued from previous page)

```
extract_task >> transform_task >> load_task
```

Note

This version produces the same result as the TaskFlow API example, but requires explicit management of XComs and task dependencies.

The Taskflow Way

Using TaskFlow, all of this is handled automatically.

```
airflow/example_dags/tutorial_taskflow_api.py
```

```
import json

import pendulum

from airflow.sdk import dag, task
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def tutorial_taskflow_api():
    """
    ## TaskFlow API Tutorial Documentation
    This is a simple data pipeline example which demonstrates the use of
    the TaskFlow API using three simple tasks for Extract, Transform, and Load.
    Documentation that goes along with the Airflow TaskFlow API tutorial is
    located
    [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html)
    """

    @task()
    def extract():
        """
        #### Extract task
        A simple Extract task to get data ready for the rest of the data
        pipeline. In this case, getting data is simulated by reading from a
        hardcoded JSON string.
        """
        data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'

        order_data_dict = json.loads(data_string)
        return order_data_dict
    @task(multiple_outputs=True)
    def transform(order_data_dict: dict):
        """
        
```

(continues on next page)

(continued from previous page)

```

#### Transform task
A simple Transform task which takes in the collection of order data and
computes the total order value.
"""
total_order_value = 0

for value in order_data_dict.values():
    total_order_value += value

return {"total_order_value": total_order_value}
@task()
def load(total_order_value: float):
    """
    #### Load task
    A simple Load task which takes in the result of the Transform task and
    instead of saving it to end user review, just prints it out.
    """

    print(f"Total order value is: {total_order_value:.2f}")
order_data = extract()
order_summary = transform(order_data)
load(order_summary["total_order_value"])
tutorial_taskflow_api()

```

Airflow still uses XComs and builds a dependency graph — it's just abstracted away so you can focus on your business logic.

How XComs Work

TaskFlow return values are stored as XComs automatically. These values can be inspected in the UI under the “XCom” tab. Manual `xcom_pull()` is still possible for traditional operators.

Error Handling and Retries

You can easily configure retries for your tasks using decorators. For example, you can set a maximum number of retries directly in the task decorator:

```
@task(retries=3)
def my_task(): ...
```

This helps ensure that transient failures do not lead to task failure.

Task Parameterization

You can reuse decorated tasks in multiple DAGs and override parameters like `task_id` or `retries`.

```
start = add_task.override(task_id="start")(1, 2)
```

You can even import decorated tasks from a shared module.

What to Explore Next

Nice work! You've now written your first pipeline using the TaskFlow API. Curious where to go from here?

- Add a new task to the DAG – maybe a filter or validation step
- Modify return values and pass multiple outputs
- Explore retries and overrides with `.override(task_id="...")`
- Open the Airflow UI and inspect how the data flows between tasks, including task logs and dependencies

See also

- Continue to the next step: [Building a Simple Data Pipeline](#)
- Learn more in the [TaskFlow API docs](#) or continue below for [Advanced Taskflow Patterns](#)
- Read about Airflow concepts in [Core Concepts](#)

Advanced Taskflow Patterns

Once you're comfortable with the basics, here are a few powerful techniques you can try.

Reusing Decorated Tasks

You can reuse decorated tasks across multiple DAGs or DAG runs. This is especially useful for common logic like reusable utilities or shared business rules. Use `.override()` to customize task metadata like `task_id` or `retries`.

```
start = add_task.override(task_id="start")(1, 2)
```

You can even import decorated tasks from a shared module.

Handling Conflicting Dependencies

Sometimes tasks require different Python dependencies than the rest of your DAG — for example, specialized libraries or system-level packages. TaskFlow supports multiple execution environments to isolate those dependencies.

Dynamically Created Virtualenv

Creates a temporary virtualenv at task runtime. Great for experimental or dynamic tasks, but may have cold start overhead.

```
/build/src/repos/airflow/providers/standard/tests/system/standard/example_python_decorator.py
```

```
@task.virtualenv(
    task_id="virtualenv_python", requirements=["colorama==0.4.0"], system_site_
    packages=False
)
def callable_virtualenv():
    """
    Example function that will be performed in a virtual environment.

```

(continues on next page)

(continued from previous page)

```

Importing at the module level ensures that it will not attempt to import the
library before it is installed.
"""
from time import sleep

from colorama import Back, Fore, Style

print(Fore.RED + "some red text")
print(Back.GREEN + "and with a green background")
print(Style.DIM + "and in dim text")
print(Style.RESET_ALL)
for _ in range(4):
    print(Style.DIM + "Please wait...", flush=True)
    sleep(1)
print("Finished")

virtualenv_task = callable_virtualenv()

```

External Python Environment

Executes the task using a pre-installed Python interpreter — ideal for consistent environments or shared virtualenvs.

/build/src/repos/airflow/providers/standard/tests/system/standard/example_python_decorator.py

```

@task.external_python(task_id="external_python", python=PATH_TO_PYTHON_BINARY)
def callable_external_python():
"""
Example function that will be performed in a virtual environment.

Importing at the module level ensures that it will not attempt to import the
library before it is installed.
"""
import sys
from time import sleep

print(f"Running task via {sys.executable}")
print("Sleeping")
for _ in range(4):
    print("Please wait...", flush=True)
    sleep(1)
print("Finished")

external_python_task = callable_external_python()

```

Docker Environment

Runs your task in a Docker container. Useful for packaging everything the task needs — but requires Docker to be available on your worker.

/build/src/repos/airflow/providers/docker/tests/system/docker/example_taskflow_api_docker_virtualenv.py

```
@task.docker(image="python:3.9-slim-bookworm", multiple_outputs=True)
def transform(order_data_dict: dict):
    """
    #### Transform task
    A simple Transform task which takes in the collection of order data and
    computes the total order value.
    """
    total_order_value = 0

    for value in order_data_dict.values():
        total_order_value += value

    return {"total_order_value": total_order_value}
```

 Note

Requires Airflow 2.2 and the Docker provider.

KubernetesPodOperator

Runs your task inside a Kubernetes pod, fully isolated from the main Airflow environment. Ideal for large tasks or tasks requiring custom runtimes.

/build/src/repos/airflow/providers/cncf/kubernetes/tests/system/cncf/kubernetes/example_kubernetes_decorator.py

```
@task.kubernetes(
    image="python:3.9-slim-buster",
    name="k8s_test",
    namespace="default",
    in_cluster=False,
    config_file="/path/to/.kube/config",
)
def execute_in_k8s_pod():
    import time

    print("Hello from k8s pod")
    time.sleep(2)

@task.kubernetes(image="python:3.9-slim-buster", namespace="default", in_cluster=False)
def print_pattern():
    n = 5
    for i in range(n):
        # inner loop to handle number of columns
```

(continues on next page)

(continued from previous page)

```

# values changing acc. to outer loop
for _ in range(i + 1):
    # printing stars
    print("* ", end="")

# ending line after each row
print("\r")

execute_in_k8s_pod_instance = execute_in_k8s_pod()
print_pattern_instance = print_pattern()

execute_in_k8s_pod_instance >> print_pattern_instance

```

Note

Requires Airflow 2.4 and the Kubernetes provider.

Using Sensors

Use `@task.sensor` to build lightweight, reusable sensors using Python functions. These support both poke and reschedule modes.

/build/src/repos/airflow/providers/standard/tests/system/standard/example_sensor_decorator.py

```

import pendulum

from airflow.sdk import PokeReturnValue, dag, task
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def example_sensor_decorator():
    # Using a sensor operator to wait for the upstream data to be ready.
    @task.sensor(poke_interval=60, timeout=3600, mode="reschedule")
    def wait_for_upstream() -> PokeReturnValue:
        return PokeReturnValue(is_done=True, xcom_value="xcom_value")
    @task
    def dummy_operator() -> None:
        pass
    wait_for_upstream() >> dummy_operator()
tutorial_etl_dag = example_sensor_decorator()

```

Mixing with Traditional Tasks

You can combine decorated tasks with classic Operators. This is helpful when using community providers or when migrating incrementally to TaskFlow.

You can chain Taskflow and traditional tasks using `>>` or pass data using the `.output` attribute.

Templating in TaskFlow

Like traditional tasks, decorated TaskFlow functions support templated arguments — including loading content from files or using runtime parameters.

When running your callable, Airflow will pass a set of keyword arguments that can be used in your function. This set of kwargs correspond exactly to what you can use in your Jinja templates. For this to work, you can add context keys you would like to receive in the function as keyword arguments.

For example, the callable in the code block below will get values of the `ti` and `next_ds` context variables:

```
@task
def my_python_callable(*, ti, next_ds):
    pass
```

You can also choose to receive the entire context with `**kwargs`. Note that this can incur a slight performance penalty since Airflow will need to expand the entire context that likely contains many things you don't actually need. It is therefore more recommended for you to use explicit arguments, as demonstrated in the previous paragraph.

```
@task
def my_python_callable(**kwargs):
    ti = kwargs["ti"]
    next_ds = kwargs["next_ds"]
```

Also, sometimes you might want to access the context somewhere deep in the stack, but you do not want to pass the context variables from the task callable. You can still access execution context via the `get_current_context` method.

```
from airflow.sdk import get_current_context

def some_function_in_your_library():
    context = get_current_context()
    ti = context["ti"]
```

Arguments passed to decorated functions are automatically templated. You can also template file using `templates_exts`:

```
@task(templates_exts=[".sql"])
def read_sql(sql): ...
```

Conditional Execution

Use `@task.run_if()` or `@task.skip_if()` to control whether a task runs based on dynamic conditions at runtime — without altering your DAG structure.

```
@task.run_if(lambda ctx: ctx["task_instance"].task_id == "run")
@task.bash()
def echo():
    return "echo 'run'"
```

What's Next

Now that you've seen how to build clean, maintainable DAGs using the TaskFlow API, here are some good next steps:

- Explore asset-aware workflows in *Asset-Aware Scheduling*
- Dive into scheduling patterns in *Scheduling Options*
- Move to the next tutorial: [Building a Simple Data Pipeline](#)

3.4.3 Building a Simple Data Pipeline

Welcome to the third tutorial in our series! At this point, you've already written your first DAG and used some basic operators. Now it's time to build a small but meaningful data pipeline – one that retrieves data from an external source, loads it into a database, and cleans it up along the way.

This tutorial introduces the `SQLExecuteQueryOperator`, a flexible and modern way to execute SQL in Airflow. We'll use it to interact with a local Postgres database, which we'll configure in the Airflow UI.

By the end of this tutorial, you'll have a working pipeline that:

- Downloads a CSV file
- Loads the data into a staging table
- Cleans the data and upserts it into a target table

Along the way, you'll gain hands-on experience with Airflow's UI, connection system, SQL execution, and DAG authoring patterns.

Want to go deeper as you go? Here are two helpful references:

- The [SQLExecuteQueryOperator](#) documentation
- The [Postgres provider](#) documentation

Let's get started!

Initial setup

Caution

You'll need Docker installed to run this tutorial. We'll be using Docker Compose to launch Airflow locally. If you need help setting it up, check out the [Docker Compose quickstart guide](#).

To run our pipeline, we need a working Airflow environment. Docker Compose makes this easy and safe – no system-wide installs required. Just open your terminal and run the following:

```
# Download the docker-compose.yaml file
curl -Lfo 'https://airflow.apache.org/docs/apache-airflow/stable/docker-compose.yaml'

# Make expected directories and set an expected environment variable
mkdir -p ./dags ./logs ./plugins
echo -e "AIRFLOW_UID=$(id -u)" > .env

# Initialize the database
docker compose up airflow-init
```

(continues on next page)

(continued from previous page)

```
# Start up all services
docker compose up
```

Once Airflow is up and running, visit the UI at <http://localhost:8080>.

Log in with:

- **Username:** airflow
- **Password:** airflow

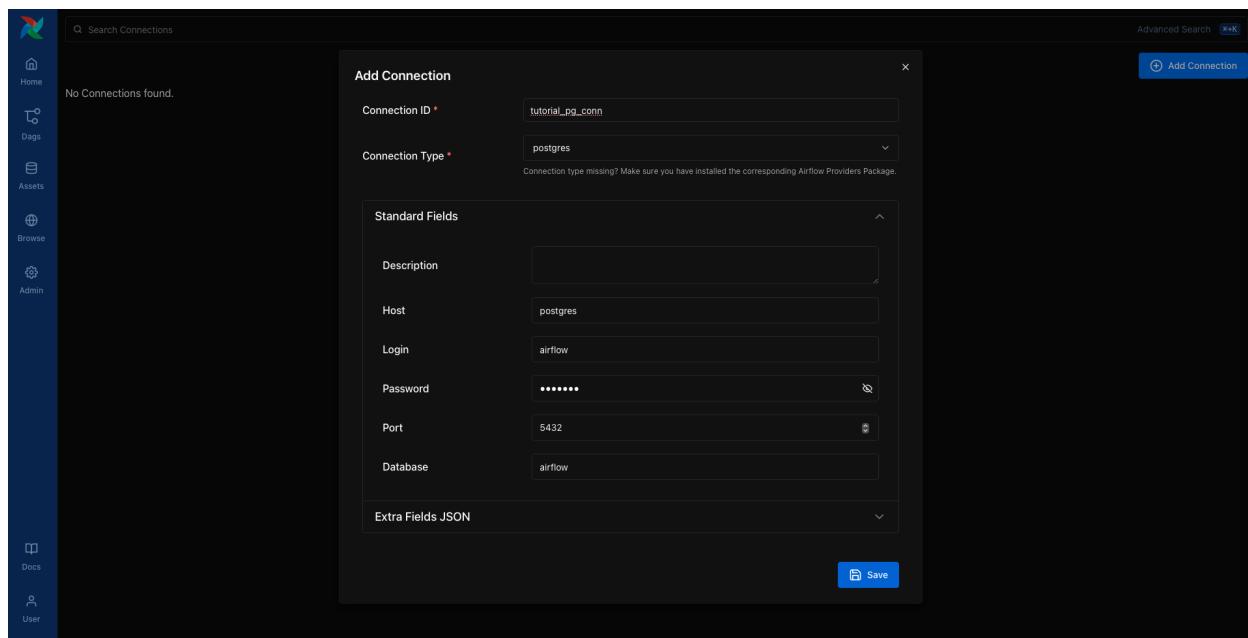
You'll land in the Airflow dashboard, where you can trigger DAGs, explore logs, and manage your environment.

Create a Postgres Connection

Before our pipeline can write to Postgres, we need to tell Airflow how to connect to it. In the UI, open the **Admin > Connections** page and click the + button to add a new connection.

Fill in the following details:

- Connection ID: `tutorial_pg_conn`
- Connection Type: `postgres`
- Host: `postgres`
- Database: `airflow` (this is the default database in our container)
- Login: `airflow`
- Password: `airflow`
- Port: `5432`



Save the connection. This tells Airflow how to reach the Postgres database running in your Docker environment.

Next, we'll start building the pipeline that uses this connection.

Create tables for staging and final data

Let's begin with table creation. We'll create two tables:

- `employees_temp`: a staging table used for raw data
- `employees`: the cleaned and deduplicated destination

We'll use the `SQLExecuteQueryOperator` to run the SQL statements needed to create these tables.

```
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

create_employees_table = SQLExecuteQueryOperator(
    task_id="create_employees_table",
    conn_id="tutorial_pg_conn",
    sql="""
        CREATE TABLE IF NOT EXISTS employees (
            "Serial Number" NUMERIC PRIMARY KEY,
            "Company Name" TEXT,
            "Employee Markme" TEXT,
            "Description" TEXT,
            "Leave" INTEGER
        );""",
)

create_employees_temp_table = SQLExecuteQueryOperator(
    task_id="create_employees_temp_table",
    conn_id="tutorial_pg_conn",
    sql="""
        DROP TABLE IF EXISTS employees_temp;
        CREATE TABLE employees_temp (
            "Serial Number" NUMERIC PRIMARY KEY,
            "Company Name" TEXT,
            "Employee Markme" TEXT,
            "Description" TEXT,
            "Leave" INTEGER
        );""",
)
```

You can optionally place these SQL statements in `.sql` files inside your `dags/` folder and pass the file path to the `sql=` argument. This can be a great way to keep your DAG code clean.

Load data into the staging table

Next, we'll download a CSV file, save it locally, and load it into `employees_temp` using the `PostgresHook`.

```
import os
import requests
from airflow.sdk import task
from airflow.providers.postgres.hooks.postgres import PostgresHook

@task
def get_data():
```

(continues on next page)

(continued from previous page)

```
# NOTE: configure this as appropriate for your Airflow environment
data_path = "/opt/airflow/dags/files/employees.csv"
os.makedirs(os.path.dirname(data_path), exist_ok=True)

url = "https://raw.githubusercontent.com/apache/airflow/main/airflow-core/docs/tutorial/pipeline_example.csv"

response = requests.request("GET", url)

with open(data_path, "w") as file:
    file.write(response.text)

postgres_hook = PostgresHook(postgres_conn_id="tutorial_pg_conn")
conn = postgres_hook.get_conn()
cur = conn.cursor()
with open(data_path, "r") as file:
    cur.copy_expert(
        "COPY employees_temp FROM STDIN WITH CSV HEADER DELIMITER AS ',' QUOTE '\"',
        file,
    )
conn.commit()
```

This task gives you a taste of combining Airflow with native Python and SQL hooks – a common pattern in real-world pipelines.

Merge and clean the data

Now let's deduplicate the data and merge it into our final table. We'll write a task that runs a SQL *INSERT ... ON CONFLICT DO UPDATE*.

```
from airflow.sdk import task
from airflow.providers.postgres.hooks.postgres import PostgresHook

@task
def merge_data():
    query = """
        INSERT INTO employees
        SELECT *
        FROM (
            SELECT DISTINCT *
            FROM employees_temp
        ) t
        ON CONFLICT ("Serial Number") DO UPDATE
        SET
            "Employee Markme" = excluded."Employee Markme",
            "Description" = excluded."Description",
            "Leave" = excluded."Leave";
    """
    try:
        postgres_hook = PostgresHook(postgres_conn_id="tutorial_pg_conn")
        conn = postgres_hook.get_conn()
```

(continues on next page)

(continued from previous page)

```

    cur = conn.cursor()
    cur.execute(query)
    conn.commit()
    return 0
except Exception as e:
    return 1

```

Defining the DAG

Now that we've defined all our tasks, it's time to put them together into a DAG.

```

import datetime
import pendulum
import os

import requests
from airflow.sdk import dag, task
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator


@dag(
    dag_id="process_employees",
    schedule="@0 0 * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    dagrun_timeout=datetime.timedelta(minutes=60),
)
def ProcessEmployees():
    create_employees_table = SQLExecuteQueryOperator(
        task_id="create_employees_table",
        conn_id="tutorial_pg_conn",
        sql="""
            CREATE TABLE IF NOT EXISTS employees (
                "Serial Number" NUMERIC PRIMARY KEY,
                "Company Name" TEXT,
                "Employee Markme" TEXT,
                "Description" TEXT,
                "Leave" INTEGER
            );"""
    )
    create_employees_temp_table = SQLExecuteQueryOperator(
        task_id="create_employees_temp_table",
        conn_id="tutorial_pg_conn",
        sql="""
            DROP TABLE IF EXISTS employees_temp;
            CREATE TABLE employees_temp (
                "Serial Number" NUMERIC PRIMARY KEY,
                "Company Name" TEXT,
                "Employee Markme" TEXT,
                "Description" TEXT,
            """

```

(continues on next page)

(continued from previous page)

```

        "Leave" INTEGER
    );""",

)

@task
def get_data():
    # NOTE: configure this as appropriate for your Airflow environment
    data_path = "/opt/airflow/dags/files/employees.csv"
    os.makedirs(os.path.dirname(data_path), exist_ok=True)

    url = "https://raw.githubusercontent.com/apache/airflow/main/airflow-core/docs/tutorial/pipeline_example.csv"

    response = requests.request("GET", url)

    with open(data_path, "w") as file:
        file.write(response.text)

    postgres_hook = PostgresHook(postgres_conn_id="tutorial_pg_conn")
    conn = postgres_hook.get_conn()
    cur = conn.cursor()
    with open(data_path, "r") as file:
        cur.copy_expert(
            "COPY employees_temp FROM STDIN WITH CSV HEADER DELIMITER AS ',' QUOTE '\\\"',",
            file,
        )
    conn.commit()

@task
def merge_data():
    query = """
        INSERT INTO employees
        SELECT *
        FROM (
            SELECT DISTINCT *
            FROM employees_temp
        ) t
        ON CONFLICT ("Serial Number") DO UPDATE
        SET
            "Employee Markme" = excluded."Employee Markme",
            "Description" = excluded."Description",
            "Leave" = excluded."Leave";
    """
    try:
        postgres_hook = PostgresHook(postgres_conn_id="tutorial_pg_conn")
        conn = postgres_hook.get_conn()
        cur = conn.cursor()
        cur.execute(query)
        conn.commit()
        return 0
    except Exception as e:

```

(continues on next page)

(continued from previous page)

```
return 1

[create_employees_table, create_employees_temp_table] >> get_data() >> merge_data()

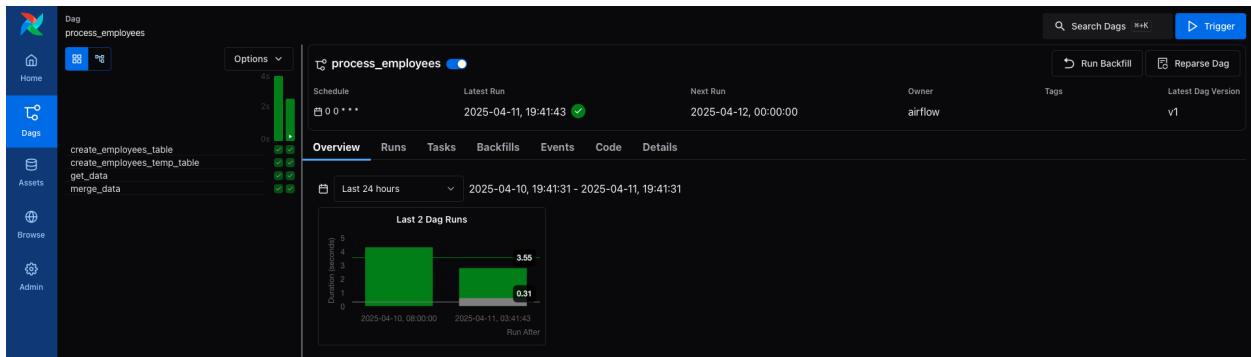
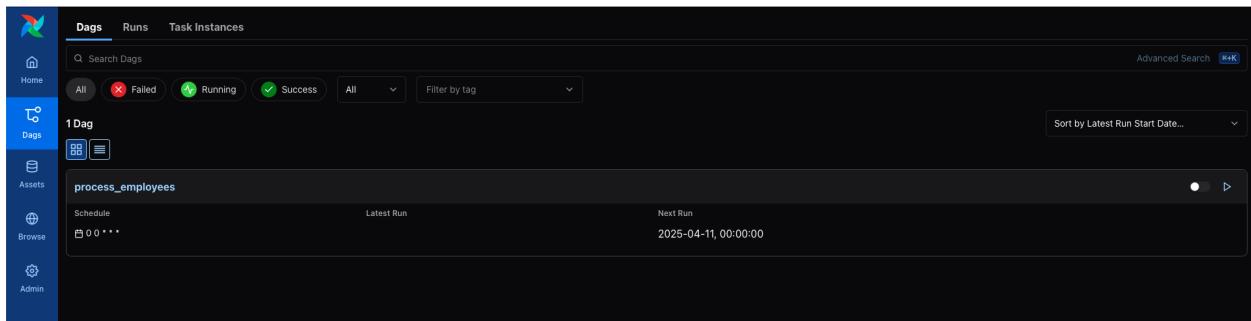
dag = ProcessEmployees()
```

Save this DAG as `dags/process_employees.py`. After a short delay, it will show up in the UI.

Trigger and explore your DAG

Open the Airflow UI and find the `process_employees` DAG in the list. Toggle it “on” using the slider, then trigger a run using the play button.

You can watch each task as it runs in the **Grid** view, and explore logs for each step.



Once it succeeds, you’ll have a fully working pipeline that integrates data from the outside world, loads it into Postgres, and keeps it clean.

What's Next?

Nice work! You've now built a real pipeline using Airflow's core patterns and tools. Here are a few ideas for where to go next:

- Try swapping in a different SQL provider, like MySQL or SQLite.
- Split your DAG into TaskGroups or refactor into a more usable pattern.
- Add an alerting step or send a notification when data is processed.

See also

- Browse more how-to guides in the *Airflow documentation*
- Explore the [SQL provider reference](#)
- Learn how to *write your own custom operator*

3.4.4 Cloud-Native Workflows with Object Storage

Added in version 2.8.

Welcome to the final tutorial in our Airflow series! By now, you've built DAGs with Python and the Taskflow API, passed data with XComs, and chained tasks together into clear, reusable workflows.

In this tutorial we'll take it a step further by introducing the **Object Storage API**. This API makes it easier to read from and write to cloud storage – like Amazon S3, Google Cloud Storage (GCS), or Azure Blob Storage – without having to worry about provider-specific SDKs or low-level credentials management.

We'll walk you through a real-world use case:

1. Pulling data from a public API
2. Saving that data to object storage in Parquet format
3. Analyzing it using SQL with DuckDB

Along the way, we'll highlight the new `ObjectStoragePath` abstraction, explain how Airflow handles cloud credentials via connections, and show how this enables portable, cloud-agnostic pipelines.

Why This Matters

Many data workflows depend on files – whether it's raw CSVs, intermediate Parquet files, or model artifacts. Traditionally, you'd need to write S3-specific or GCS-specific code for this. Now, with `ObjectStoragePath`, you can write generic code that works across providers, as long as you've configured the right Airflow connection.

Let's get started!

Prerequisites

Before diving in, make sure you have the following:

- **DuckDB**, an in-process SQL database: Install with `pip install duckdb`
- **Amazon S3 access and Amazon Provider with s3fs**: `pip install apache-airflow-providers-amazon[s3fs]` (You can substitute your preferred provider by changing the storage URL protocol and installing the relevant provider.)
- **Pandas** for working with tabular data: `pip install pandas`

Creating an ObjectStoragePath

At the heart of this tutorial is `ObjectStoragePath`, a new abstraction for handling paths on cloud object stores. Think of it like `pathlib.Path`, but for buckets instead of filesystems.

`airflow/example_dags/tutorial_objectstorage.py`

```
base = ObjectStoragePath("s3://aws_default@airflow-tutorial-data/")
```

The URL syntax is simple: `protocol://bucket/path/to/file`

- The `protocol` (like `s3`, `gs` or `azure`) determines the backend
- The “username” part of the URL can be a `conn_id`, telling Airflow how to authenticate
- If the `conn_id` is omitted, Airflow will fall back to the default connection for that backend

You can also provide the `conn_id` as keyword argument for clarity:

```
ObjectStoragePath("s3://airflow-tutorial-data/", conn_id="aws_default")
```

This is especially handy when reusing a path defined elsewhere (like in an Asset), or when the connection isn’t baked into the URL. The keyword argument always takes precedence.



Tip

You can safely create an `ObjectStoragePath` in your global DAG scope. Connections are resolved only when the path is used, not when it’s created.

Saving Data to Object Storage

Let’s fetch some data and save it to the cloud.

`airflow/example_dags/tutorial_objectstorage.py`

```
@task
def get_air_quality_data(**kwargs) -> ObjectStoragePath:
    """
    #### Get Air Quality Data
    This task gets air quality data from the Finnish Meteorological Institute's
    open data API. The data is saved as parquet.
    """
    import pandas as pd

    logical_date = kwargs["logical_date"]
    start_time = kwargs["data_interval_start"]

    params = {
        "format": "json",
        "precision": "double",
        "groupareas": "0",
        "producer": "airquality_urban",
    }
```

(continues on next page)

(continued from previous page)

```

    "area": "Uusimaa",
    "param": ",".join(aq_fields.keys()),
    "starttime": start_time.isoformat(timespec="seconds"),
    "endtime": logical_date.isoformat(timespec="seconds"),
    "tz": "UTC",
}

response = requests.get(API, params=params)
response.raise_for_status()

# ensure the bucket exists
base.mkdir(exist_ok=True)

formatted_date = logical_date.format("YYYYMMDD")
path = base / f"air_quality_{formatted_date}.parquet"

df = pd.DataFrame(response.json()).astype(aq_fields)
with path.open("wb") as file:
    df.to_parquet(file)

return path

```

Here's what's happening:

- We call a public API from the Finnish Meteorological Institute for Helsinki air quality data
- The JSON response is parsed into a pandas DataFrame
- We generate a filename based on the task's logical date
- Using `ObjectStoragePath`, we write the data directly to cloud storage as Parquet

This is a classic Taskflow pattern. The object key changes each day, allowing us to run this daily and build a dataset over time. We return the final object path to be used in the next task.

Why this is cool: No boto3, no GCS client setup, no credentials juggling. Just simple file semantics that work across storage backends.

Analyzing the Data with DuckDB

Now let's analyze that data using SQL with DuckDB.

`airflow/example_dags/tutorial_objectstorage.py`

```

@task
def analyze(path: ObjectStoragePath, **kwargs):
    """
    #### Analyze
    This task analyzes the air quality data, prints the results
    """

```

(continues on next page)

(continued from previous page)

```
import duckdb

conn = duckdb.connect(database=":memory:")
conn.register_filesystem(path.fs)
conn.execute(f"CREATE OR REPLACE TABLE airquality_urban AS SELECT * FROM read_
˓→parquet('{path}')")

df2 = conn.execute("SELECT * FROM airquality_urban").fetchhdf()

print(df2.head())
```

A few key things to note:

- DuckDB supports reading Parquet natively
- DuckDB and ObjectStoragePath both rely on `fsspec`, which makes it easy to register the object storage backend
- We use `path.fs` to grab the right filesystem object and register it with DuckDB
- Finally, we query the Parquet file using SQL and return a pandas DataFrame

Notice that the function doesn't recreate the path manually – it gets the full path from the previous task using Xcom. This makes the task portable and decoupled from earlier logic.

Bringing It All Together

Here's the full DAG that ties everything together:

`airflow/example_dags/tutorial_objectstorage.py`

```
import pendulum
import requests

from airflow.sdk import ObjectStoragePath, dag, task

API = "https://opendata.fmi.fi/timeseries"

aq_fields = {
    "fmisid": "int32",
    "time": "datetime64[ns]",
    "AQINDEX_PT1H_avg": "float64",
    "PM10_PT1H_avg": "float64",
    "PM25_PT1H_avg": "float64",
    "O3_PT1H_avg": "float64",
    "CO_PT1H_avg": "float64",
    "SO2_PT1H_avg": "float64",
    "NO2_PT1H_avg": "float64",
    "TRSC_PT1H_avg": "float64",
}
```

(continues on next page)

(continued from previous page)

```

base = ObjectStoragePath("s3://aws_default@airflow-tutorial-data/")

@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def tutorial_objectstorage():
    """
    ## Object Storage Tutorial Documentation
    This is a tutorial DAG to showcase the usage of the Object Storage API.
    Documentation that goes along with the Airflow Object Storage tutorial is
    located
    [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial/objectstorage.
    html)
    """

    @task
    def get_air_quality_data(**kwargs) -> ObjectStoragePath:
        """
        #### Get Air Quality Data
        This task gets air quality data from the Finnish Meteorological Institute's
        open data API. The data is saved as parquet.
        """

        import pandas as pd

        logical_date = kwargs["logical_date"]
        start_time = kwargs["data_interval_start"]

        params = {
            "format": "json",
            "precision": "double",
            "groupareas": "0",
            "producer": "airquality_urban",
            "area": "Uusimaa",
            "param": ",".join(aq_fields.keys()),
            "starttime": start_time.isoformat(timespec="seconds"),
            "endtime": logical_date.isoformat(timespec="seconds"),
            "tz": "UTC",
        }

        response = requests.get(API, params=params)
        response.raise_for_status()

        # ensure the bucket exists
        base.mkdir(exist_ok=True)

        formatted_date = logical_date.format("YYYYMMDD")
        path = base / f"air_quality_{formatted_date}.parquet"

        df = pd.DataFrame(response.json()).astype(aq_fields)
    
```

(continues on next page)

(continued from previous page)

```

with path.open("wb") as file:
    df.to_parquet(file)

    return path
@task
def analyze(path: ObjectStoragePath, **kwargs):
    """
    #### Analyze
    This task analyzes the air quality data, prints the results
    """
    import duckdb

    conn = duckdb.connect(database=":memory:")
    conn.register_filesystem(path.fs)
    conn.execute(f"CREATE OR REPLACE TABLE airquality_urban AS SELECT * FROM read_
    ↪parquet('{path}')")

    df2 = conn.execute("SELECT * FROM airquality_urban").fetchdf()

    print(df2.head())
    obj_path = get_air_quality_data()
    analyze(obj_path)
tutorial_objectstorage()

```

You can trigger this DAG and view it in the Graph View in the Airflow UI. Each task logs its inputs and outputs clearly, and you can inspect returned paths in the Xcom tab.

What to Explore Next

Here are some ways to take this further:

- Use object sensors (like `S3KeySensor`) to wait for files uploaded by external systems
- Orchestrate S3-to-GCS transfers or cross-region data syncs
- Add branching logic to handle missing or malformed files
- Experiment with different formats like CSV or JSON

See Also

- Learn how to securely access cloud services by configuring Airflow connections in the *Managing Connections guide*
- Build event-driven pipelines that respond to file uploads or external triggers using the *Event-Driven Scheduling framework*
- Reinforce your understanding of decorators, return values, and task chaining with the *TaskFlow API guide*

3.5 How-to Guides

Setting up the sandbox in the [Quick Start](#) section was easy; building a production-grade environment requires a bit more work!

These how-to guides will step you through common tasks in using and configuring an Airflow environment.

3.5.1 Using the Command Line Interface

This document is meant to give an overview of all common tasks while using the CLI.

Note

For more information on CLI commands, see [Command Line Interface and Environment Variables Reference](#)

Set Up Bash/Zsh Completion

When using bash (or zsh) as your shell, airflow can use `argcomplete` for auto-completion.

For global activation of all argcomplete enabled python applications run:

```
sudo activate-global-python-argcomplete
```

For permanent (but not global) airflow activation, use:

```
register-python-argcomplete airflow >> ~/.bashrc
```

For one-time activation of argcomplete for Airflow only, use:

```
eval "$(register-python-argcomplete airflow)"
```

If you're using zsh, add the following to your `.zshrc`:

```
autoload bashcompinit  
bashcompinit  
eval "$(register-python-argcomplete airflow)"
```

Creating a Connection

For information on creating connections using the CLI, see [Creating a Connection from the CLI](#)

Exporting DAG structure as an image

Airflow lets you print or save your DAG structure as an image. This is useful for documenting or sharing your DAG structure. You'll need to have `Graphviz` installed.

For example, to print the `example_complex` DAG to the terminal:

```
airflow dags show example_complex
```

This will print the rendered DAG structure to the screen in DOT format.

Multiple file formats are supported. To use them, add the argument `--save [filename]. [format]`.

To save the `example_complex` DAG as a PNG file:

```
airflow dags show example_complex --save example_complex.png
```

This will save the following image as a file:

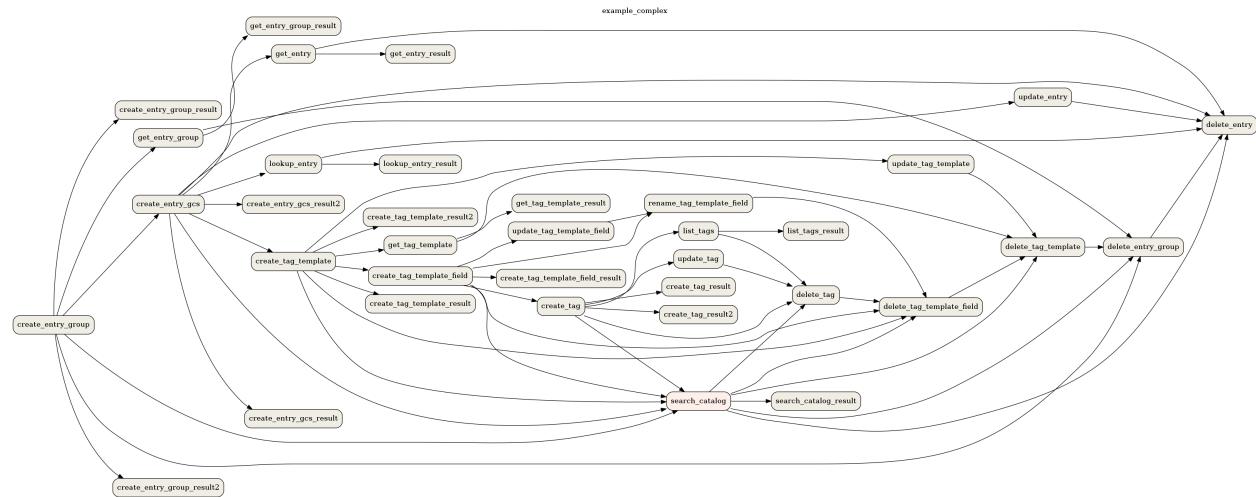


Fig. 1: Example DAG representation

The following file formats are supported:

- `bmp`
- `canon, dot, gv, xdot, xdot1.2, xdot1.4`
- `cimage`
- `cmap`
- `eps`
- `exr`
- `fig`
- `gd, gd2`
- `gif`
- `gtk`
- `ico`
- `imap, cmapx`
- `imap_np, cmapx_np`
- `ismap`
- `jp2`
- `jpg, jpeg, jpe`
- `json, json0, dot_json, xdot_json`
- `pct, pict`
- `pdf`
- `pic`

- plain, plain-ext
- png
- pov
- ps
- ps2
- psd
- sgi
- svg, svgz
- tga
- tif, tiff
- tk
- vml, vmlz
- vrml
- wbmp
- webp
- xlib
- x11

By default, Airflow looks for dags in the directory specified by the `dags_folder` option in the `[core]` section of the `airflow.cfg` file. You can select a new directory with the `--subdir` argument.

Display dag structure

Sometimes you will work on dags that contain complex dependencies. It is helpful then to preview the DAG to see if it is correct.

If you have macOS, you can use [iTerm2](#) together with the `imgcat` script to display the DAG structure in the console. You also need to have [Graphviz](#) installed.

Other terminals do not support the display of high-quality graphics. You can convert the image to a text form, but its resolution will prevent you from reading it.

To do this, you should use the `--imgcat` switch in the `airflow dags show` command. For example, if you want to display `example_bash_operator` DAG then you can use the following command:

```
airflow dags show example_bash_operator --imgcat
```

You will see a similar result as in the screenshot below.

Formatting commands output

Some Airflow commands like `airflow dags list` or `airflow tasks states-for-dag-run` support `--output` flag which allow users to change the formatting of command's output. Possible options:

- `table` - renders the information as a plain text table
- `simple` - renders the information as simple table which can be parsed by standard linux utilities
- `json` - renders the information in form of json string
- `yaml` - render the information in form of valid yaml

```
root@a55d15d6e263:/opt/airflow# airflow dags show example_bash_operator --imgcat
/opt/airflow/airflow/models/dagbag.py:21: DeprecationWarning: the imp module is deprecated
in favour of importlib; see the module's documentation for alternative uses
import imp
[2020-01-03 01:54:57,866] {settings.py:179} INFO - settings.configure_orm(): Using pool se
ttings. pool_size=5, max_overflow=10, pool_recycle=1800, pid=1155
[2020-01-03 01:54:58,548] {executor_loader.py:59} INFO - Using executor SequentialExecutor
[2020-01-03 01:54:58,549] {dagbag.py:411} INFO - Filling up the DagBag from /opt/airflow/a
irflow/example_dags
graph TD
    runme_0[runme_0] --> run_after_loop[run_after_loop]
    runme_1[runme_1] --> run_after_loop
    runme_2[runme_2] --> run_after_loop
    runme_2 --> also_run_this[also_run_this]
    also_run_this --> run_this_last[run_this_last]
    run_after_loop --> run_this_last
root@a55d15d6e263:/opt/airflow#
```

Fig. 2: Preview of DAG in iTerm2

Both json and yaml formats make it easier to manipulate the data using command line tools like jq or yq. For example:

```
airflow tasks states-for-dag-run example_complex 2020-11-13T00:00:00+00:00 --output json
→ | jq ".[] | {sd: .start_date, ed: .end_date}"
{
  "sd": "2020-11-29T14:53:46.811030+00:00",
  "ed": "2020-11-29T14:53:46.974545+00:00"
}
{
  "sd": "2020-11-29T14:53:56.926441+00:00",
  "ed": "2020-11-29T14:53:57.118781+00:00"
}
{
  "sd": "2020-11-29T14:53:56.915802+00:00",
  "ed": "2020-11-29T14:53:57.125230+00:00"
}
{
  "sd": "2020-11-29T14:53:56.922131+00:00",
  "ed": "2020-11-29T14:53:57.129091+00:00"
}
{
  "sd": "2020-11-29T14:53:56.931243+00:00",
  "ed": "2020-11-29T14:53:57.126306+00:00"
}
```

Purge history from metadata database

Note

It's strongly recommended that you backup the metadata database before running the db clean command.

The db clean command works by deleting from each table the records older than the provided --clean-before-timestamp.

You can optionally provide a list of tables to perform deletes on. If no list of tables is supplied, all tables will be included.

You can use the --dry-run option to print the row counts in the primary tables to be cleaned.

By default, db clean will archive purged rows in tables of the form _airflow_deleted__<table>__<timestamp>. If you don't want the data preserved in this way, you may supply argument --skip-archive.

When you encounter an error without using --skip-archive, _airflow_deleted__<table>__<timestamp> would still exist in the DB. You can use db drop-archived command to manually drop these tables.

Export the purged records from the archive tables

The db export-archived command exports the contents of the archived tables, created by the db clean command, to a specified format, by default to a CSV file. The exported file will contain the records that were purged from the primary tables during the db clean process.

You can specify the export format using --export-format option. The default format is csv and is also the only supported format at the moment.

You must also specify the location of the path to which you want to export the data using --output-path option. This location must exist.

Other options include: `--tables` to specify the tables to export, `--drop-archives` to drop the archive tables after exporting.

Dropping the archived tables

If during the `db clean` process, you did not use the `--skip-archive` option which drops the archived table, you can still drop the archive tables using the `db drop-archived` command. This operation is irreversible and you are encouraged to use the `db export-archived` command to backup the tables to disk before dropping them.

You can specify the tables to drop using the `--tables` option. If no tables are specified, all archive tables will be dropped.

Beware cascading deletes

Keep in mind that some tables have foreign key relationships defined with `ON DELETE CASCADE` so deletes in one table may trigger deletes in others. For example, the `task_instance` table keys to the `dag_run` table, so if a `DagRun` record is deleted, all of its associated task instances will also be deleted.

Special handling for DAG runs

Commonly, Airflow determines which `DagRun` to run next by looking up the latest `DagRun`. If you delete all DAG runs, Airflow may schedule an old DAG run that was already completed, e.g. if you have set `catchup=True`. So the `db clean` command will preserve the latest non-manually-triggered DAG run to preserve continuity in scheduling.

Considerations for backfillable dags

Not all dags are designed for use with Airflow's backfill command. But for those which are, special care is warranted. If you delete DAG runs, and if you run backfill over a range of dates that includes the deleted DAG runs, those runs will be recreated and run again. For this reason, if you have dags that fall into this category you may want to refrain from deleting DAG runs and only clean other large tables such as `task instance` and `log` etc.

Upgrading Airflow

Run `airflow db migrate --help` for usage details.

Running migrations manually

If desired, you can generate the sql statements for an upgrade and apply each upgrade migration manually, one at a time. To do so you may use either the `--range` (for Airflow version) or `--revision-range` (for Alembic revision) option with `db migrate`. Do *not* skip running the Alembic revision id update commands; this is how Airflow will know where you are upgrading from the next time you need to. See *Reference for Database Migrations* for a mapping between revision and version.

Downgrading Airflow

Note

It's recommended that you backup your database before running `db downgrade` or any other database operation.

You can downgrade to a particular Airflow version with the `db downgrade` command. Alternatively you may provide an Alembic revision id to downgrade to.

If you want to preview the commands but not execute them, use option `--show-sql-only`.

Options `--from-revision` and `--from-version` may only be used in conjunction with the `--show-sql-only` option, because when actually *running* migrations we should always downgrade from current revision.

For a mapping between Airflow version and Alembic revision see *Reference for Database Migrations*.

 **Note**

It's highly recommended that you reserialize your dags with `dags reserialize` after you finish downgrading your Airflow environment (meaning, after you've downgraded the Airflow version installed in your Python environment, not immediately after you've downgraded the database). This is to ensure that the serialized dags are compatible with the downgraded version of Airflow.

Exporting Connections

You may export connections from the database using the CLI. The supported file formats are `json`, `yaml` and `env`.

You may specify the target file as the parameter:

```
airflow connections export connections.json
```

Alternatively you may specify `file-format` parameter for overriding the file format:

```
airflow connections export /tmp/connections --file-format yaml
```

You may also specify `-` for `STDOUT`:

```
airflow connections export -
```

The JSON format contains an object where the key contains the connection ID and the value contains the definition of the connection. In this format, the connection is defined as a JSON object. The following is a sample JSON file.

```
{
    "airflow_db": {
        "conn_type": "mysql",
        "host": "mysql",
        "login": "root",
        "password": "plainpassword",
        "schema": "airflow",
        "port": null,
        "extra": null
    },
    "druid_broker_default": {
        "conn_type": "druid",
        "host": "druid-broker",
        "login": null,
        "password": null,
        "schema": null,
        "port": 8082,
        "extra": "{\"endpoint\": \"druid/v2/sql\"}"
    }
}
```

The YAML file structure is similar to that of a JSON. The key-value pair of connection ID and the definitions of one or more connections. In this format, the connection is defined as a YAML object. The following is a sample YAML file.

```

airflow_db:
  conn_type: mysql
  extra: null
  host: mysql
  login: root
  password: plainpassword
  port: null
  schema: airflow
druid_broker_default:
  conn_type: druid
  extra: '{"endpoint": "druid/v2/sql"}'
  host: druid-broker
  login: null
  password: null
  port: 8082
  schema: null

```

You may also export connections in .env format. The key is the connection ID, and the value is the serialized representation of the connection, using either Airflow's Connection URI format or JSON. To use JSON provide option --serialization-format=json otherwise the Airflow Connection URI format will be used. The following are sample .env files using the two formats.

URI example:

```

airflow_db=mysql://root:plainpassword@mysql/airflow
druid_broker_default=druid://druid-broker:8082?endpoint=druid%2Fv2%2Fsql

```

JSON example output:

```

airflow_db={"conn_type": "mysql", "login": "root", "password": "plainpassword", "host": "mysql", "schema": "airflow"}
druid_broker_default={"conn_type": "druid", "host": "druid-broker", "port": 8082, "extra": {"\\"endpoint\\": \"druid/v2/sql\""}}

```

Testing for DAG Import Errors

The CLI can be used to check whether any discovered dags have import errors via the `list-import-errors` subcommand. It is possible to create an automation step which fails if any dags cannot be imported by checking the command output, particularly when used with `--output` to generate a standard file format. For example, the default output when there are no errors is `No data found`, and the json output is `[]`. The check can then be run in CI or pre-commit to speed up the review process and testing.

Example command that fails if there are any errors, using `jq` to parse the output:

```

airflow dags list-import-errors --output=json | jq -e 'select(type=="array" and length== 0)'

```

The line can be added to automation as-is, or if you want to print the output you can use `tee`:

```

airflow dags list-import-errors | tee import_errors.txt && jq -e 'select(type=="array" and length == 0)' import_errors.txt

```

Example in a Jenkins pipeline:

```
stage('All dags are loadable') {
    steps {
        sh 'airflow dags list-import-errors | tee import_errors.txt && jq -e \
→ "select(type=="array" and length == 0)\" import_errors.txt'
    }
}
```

Note

For this to work accurately, you must ensure Airflow does not log any additional text to stdout. For example, you may need to fix any deprecation warnings, add `2>/dev/null` to your command, or set `lazy_load_plugins = True` in the Airflow config if you have a plugin that generates logs when loaded.

3.5.2 Add tags to dags and use it for filtering in the UI

Added in version 1.10.8.

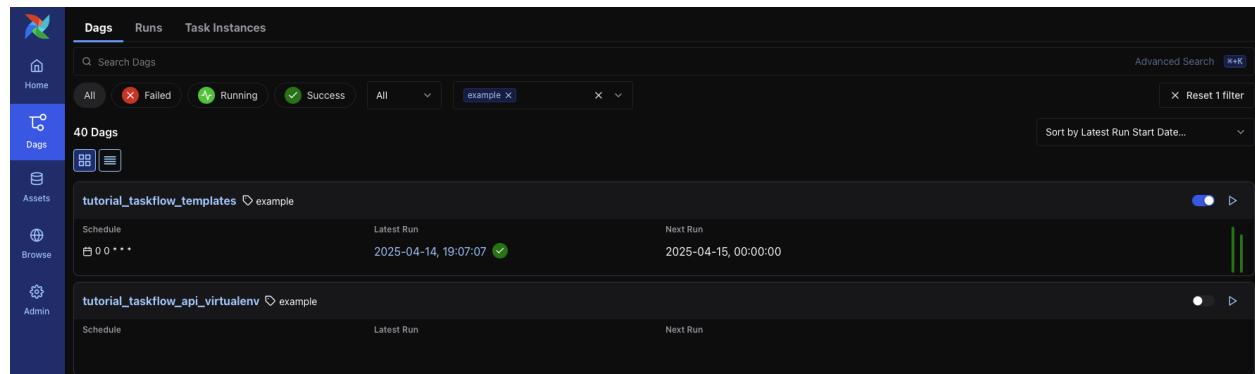
In order to filter dags (e.g by team), you can add tags in each DAG. The filter is saved in a cookie and can be reset by the reset button.

For example:

In your DAG file, pass a list of tags you want to add to the DAG object:

```
dag = DAG(dag_id="example_dag_tag", schedule="0 0 * * *", tags=["example"])
```

Screenshot:



Tags are registered as part of DAG parsing. In case of stale tags, you can purge old data with the Airflow CLI command `airflow db clean`. See `db clean usage` for more details.

3.5.3 Creating a notifier

The `BaseNotifier` is an abstract class that provides a basic structure for sending notifications in Airflow using the various `on_*__callback`. It is intended for providers to extend and customize for their specific needs.

To extend the `BaseNotifier` class, you will need to create a new class that inherits from it. In this new class, you should override the `notify` method with your own implementation that sends the notification. The `notify` method takes in a single parameter, the Airflow context, which contains information about the current task and execution.

You can also set the `template_fields` attribute to specify which attributes should be rendered as templates.

Here's an example of how you can create a Notifier class:

```
from airflow.sdk import BaseNotifier
from my_provider import send_message

class MyNotifier(BaseNotifier):
    template_fields = ("message",)

    def __init__(self, message):
        self.message = message

    def notify(self, context):
        # Send notification here, below is an example
        title = f"Task {context['task_instance'].task_id} failed"
        send_message(title, self.message)
```

Using a notifier

Once you have a notifier implementation, you can use it in your DAG definition by passing it as an argument to the `on_*_callbacks`. For example, you can use it with `on_success_callback` or `on_failure_callback` to send notifications based on the status of a task or a DAG run.

Here's an example of using the above notifier:

```
from datetime import datetime

from airflow.sdk import DAG
from airflow.providers.standard.operators.bash import BashOperator

from myprovider.notifier import MyNotifier

with DAG(
    dag_id="example_notifier",
    start_date=datetime(2022, 1, 1),
    schedule=None,
    on_success_callback=MyNotifier(message="Success!"),
    on_failure_callback=MyNotifier(message="Failure!"),
):
    task = BashOperator(
        task_id="example_task",
        bash_command="exit 1",
        on_success_callback=MyNotifier(message="Task Succeeded!"),
    )
```

For a list of community-managed notifiers, see [apache-airflow-providers:core-extensions/notifications](#).

3.5.4 Setting Configuration Options

The first time you run Airflow, it will create a file called `airflow.cfg` in your `$AIRFLOW_HOME` directory (`~/airflow` by default). This is in order to make it easy to “play” with airflow configuration.

However, for production case you are advised to generate the configuration using command line:

```
airflow config list --defaults
```

This command will produce the output that you can copy to your configuration file and edit.

It will contain all the default configuration options, with examples, nicely commented out so you need only un-comment and modify those that you want to change. This way you can easily keep track of all the configuration options that you changed from default and you can also easily upgrade your installation to new versions of Airflow when they come out and automatically use the defaults for existing options if they changed there.

You can redirect it to your configuration file and edit it:

```
airflow config list --defaults > "${AIRFLOW_HOME}/airflow.cfg"
```

You can also set options with environment variables by using this format: `AIRFLOW__{SECTION}__{KEY}` (note the double underscores).

For example, the metadata database connection string can either be set in `airflow.cfg` like this:

```
[database]
sql_alchemy_conn = my_conn_string
```

or by creating a corresponding environment variable:

```
export AIRFLOW__DATABASE__SQLALCHEMY_CONN=my_conn_string
```

Note that when the section name has a dot in it, you must replace it with an underscore when setting the env var. For example consider the pretend section `providers.some_provider`:

```
[providers.some_provider]
this_param = true
```

```
export AIRFLOW__PROVIDERS__SOME_PROVIDER__THIS_PARAM=true
```

You can also derive the connection string at run time by appending `_cmd` to the key like this:

```
[database]
sql_alchemy_conn_cmd = bash_command_to_run
```

You can also derive the connection string at run time by appending `_secret` to the key like this:

```
[database]
sql_alchemy_conn_secret = sql_alchemy_conn
# You can also add a nested path
# example:
# sql_alchemy_conn_secret = database/sqlalchemy_conn
```

This will retrieve config option from Secret Backends e.g Hashicorp Vault. See [Secrets Backends](#) for more details.

The following config options support this `_cmd` and `_secret` version:

- `sql_alchemy_conn` in `[database]` section

- fernet_key in [core] section
- broker_url in [celery] section
- flower_basic_auth in [celery] section
- result_backend in [celery] section
- password in [atlas] section
- smtp_password in [smtp] section
- secret_key in [webserver] section

The _cmd config options can also be set using a corresponding environment variable the same way the usual config options can. For example:

```
export AIRFLOW__DATABASE__SQLALCHEMY_CONN_CMD=bash_command_to_run
```

Similarly, _secret config options can also be set using a corresponding environment variable. For example:

```
export AIRFLOW__DATABASE__SQLALCHEMY_CONN_SECRET=sql_alchemy_conn
```

Note

The config options must follow the config prefix naming convention defined within the secrets backend. This means that `sql_alchemy_conn` is not defined with a connection prefix, but with config prefix. For example it should be named as `airflow/config/sqlalchemy_conn`

The idea behind this is to not store passwords on boxes in plain text files.

The universal order of precedence for all configuration options is as follows:

1. set as an environment variable (`AIRFLOW__DATABASE__SQLALCHEMY_CONN`)
2. set as a command environment variable (`AIRFLOW__DATABASE__SQLALCHEMY_CONN_CMD`)
3. set as a secret environment variable (`AIRFLOW__DATABASE__SQLALCHEMY_CONN_SECRET`)
4. set in `airflow.cfg`
5. command in `airflow.cfg`
6. secret key in `airflow.cfg`
7. Airflow's built in defaults

Note

For Airflow versions $\geq 2.2.1, < 2.3.0$ Airflow's built in defaults took precedence over command and secret key in `airflow.cfg` in some circumstances.

You can check the current configuration with the `airflow config list` command.

If you only want to see the value for one option, you can use `airflow config get-value` command as in the example below.

```
$ airflow config get-value core executor
LocalExecutor
```

 Note

For more information on configuration options, see *Configuration Reference*

 Note

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

 Note

Use the same configuration across all the Airflow components. While each component does not require all, some configurations need to be same otherwise they would not work as expected. A good example for that is *secret_key* which should be same on the Webserver and Worker to allow Webserver to fetch logs from Worker.

The webserver key is also used to authorize requests to Celery workers when logs are retrieved. The token generated using the secret key has a short expiry time though - make sure that time on ALL the machines that you run Airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors when the logs are accessed.

3.5.5 Configuring local settings

Some Airflow configuration is configured via local setting, because they require changes in the code that is executed when Airflow is initialized. Usually it is mentioned in the detailed documentation where you can configure such local settings - This is usually done in the `airflow_local_settings.py` file.

You should create a `airflow_local_settings.py` file and put it in a directory in `sys.path` or in the `$AIRFLOW_HOME/config` folder. (Airflow adds `$AIRFLOW_HOME/config` to `sys.path` when Airflow is initialized) Starting from Airflow 2.10.1, the `$AIRFLOW_HOME/dags` folder is no longer included in `sys.path` at initialization, so any local settings in that folder will not be imported. Ensure that `airflow_local_settings.py` is located in a path that is part of `sys.path` during initialization, like `$AIRFLOW_HOME/config`. For more context about this change, see the [mailing list announcement](#).

You can see the example of such local settings here: Example settings you can configure this way:

- *Cluster Policies*
- *Advanced logging configuration*
- *Dag serialization*
- *Pod mutation hook in Kubernetes Executor*
- *Control DAG parsing time*
- *Customize your UI*
- *Configure more variables to export*
- *Customize your DB configuration*

3.5.6 Set up a Database Backend

Airflow was built to interact with its metadata using [SqlAlchemy](#).

The document below describes the database engine configurations, the necessary changes to their configuration to be used with Airflow, as well as changes to the Airflow configurations to connect to these databases.

Choosing database backend

If you want to take a real test drive of Airflow, you should consider setting up a database backend to [PostgreSQL](#) or [MySQL](#). By default, Airflow uses [SQLite](#), which is intended for development purposes only.

Airflow supports the following database engine versions, so make sure which version you have. Old versions may not support all SQL statements.

- PostgreSQL: 12, 13, 14, 15, 16
- MySQL: 8.0, Innovation
- SQLite: 3.15.0+

If you plan on running more than one scheduler, you have to meet additional requirements. For details, see [Scheduler HA Database Requirements](#).

Warning

Despite big similarities between MariaDB and MySQL, we DO NOT support MariaDB as a backend for Airflow. There are known problems (for example index handling) between MariaDB and MySQL and we do not test our migration scripts nor application execution on Maria DB. We know there were people who used MariaDB for Airflow and that cause a lot of operational headache for them so we strongly discourage attempts of using MariaDB as a backend and users cannot expect any community support for it because the number of users who tried to use MariaDB for Airflow is very small.

Database URI

Airflow uses SQLAlchemy to connect to the database, which requires you to configure the Database URL. You can do this in option `sql_alchemy_conn` in section `[database]`. It is also common to configure this option with `AIRFLOW__DATABASE__SQLALCHEMY_CONN` environment variable.

Note

For more information on setting the configuration, see [Setting Configuration Options](#).

If you want to check the current value, you can use `airflow config get-value database sql_alchemy_conn` command as in the example below.

```
$ airflow config get-value database sql_alchemy_conn
sqlite:///tmp/airflow/airflow.db
```

The exact format description is described in the SQLAlchemy documentation, see [Database URLs](#). We will also show you some examples below.

Setting up a SQLite Database

SQLite database can be used to run Airflow for development purpose as it does not require any database server (the database is stored in a local file). There are plenty of limitations of using the SQLite database which you can easily find online, and it should NEVER be used for production.

There is a minimum version of sqlite3 required to run Airflow 2.0+ - minimum version is 3.15.0. Some of the older systems have an earlier version of sqlite installed by default and for those system you need to manually upgrade SQLite to use version newer than 3.15.0. Note, that this is not a `python library` version, it's the SQLite system-level application that needs to be upgraded. There are different ways how SQLite might be installed, you can find some information about that at the [official website of SQLite](#) and in the documentation specific to distribution of your Operating System.

Troubleshooting

Sometimes even if you upgrade SQLite to higher version and your local python reports higher version, the python interpreter used by Airflow might still use the older version available in the `LD_LIBRARY_PATH` set for the python interpreter that is used to start Airflow.

You can make sure which version is used by the interpreter by running this check:

```
root@b8a8e73caa2c:/opt/airflow# python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.27.2'
>>>
```

But be aware that setting environment variables for your Airflow deployment might change which SQLite library is found first, so you might want to make sure that the “high-enough” version of SQLite is the only version installed in your system.

An example URI for the sqlite database:

```
sqlite:///home/airflow/airflow.db
```

Upgrading SQLite on AmazonLinux AMI or Container Image

AmazonLinux SQLite can only be upgraded to v3.7 using the source repos. Airflow requires v3.15 or higher. Use the following instructions to setup the base image (or AMI) with latest SQLite3

Pre-requisite: You will need `wget`, `tar`, `gzip`, `gcc`, `make`, and `expect` to get the upgrade process working.

```
yum -y install wget tar gzip gcc make expect
```

Download source from <https://sqlite.org/>, make and install locally.

(continues on next page)

(continued from previous page)

```

-DSQLITE_ENABLE_STAT4 \
-DSQLITE_ENABLE_UPDATE_DELETE_LIMIT \
-DSQLITE_SOUNDEX \
-DSQLITE_TEMP_STORE=3 \
-DSQLITE_USE_URI \
-O2 \
-fPIC"
export PREFIX="/usr/local"
LIBS="-lm" ./configure --disable-tcl --enable-shared --enable-tempstore=always --prefix="$PREFIX"
make
make install

```

Post install add /usr/local/lib to library path

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

Setting up a PostgreSQL Database

You need to create a database and a database user that Airflow will use to access this database. In the example below, a database `airflow_db` and user with username `airflow_user` with password `airflow_pass` will be created

```

CREATE DATABASE airflow_db;
CREATE USER airflow_user WITH PASSWORD 'airflow_pass';
GRANT ALL PRIVILEGES ON DATABASE airflow_db TO airflow_user;

-- PostgreSQL 15 requires additional privileges:
-- Note: Connect to the airflow_db database before running the following GRANT statement
-- You can do this in psql with: \c airflow_db
GRANT ALL ON SCHEMA public TO airflow_user;

```

Note

The database must use a UTF-8 character set

You may need to update your Postgres `pg_hba.conf` to add the `airflow` user to the database access control list; and to reload the database configuration to load your change. See [The pg_hba.conf File](#) in the Postgres documentation to learn more.

Warning

When you use SQLAlchemy 1.4.0+, you need to use `postgresql://` as the database in the `sqlalchemy_conn`. In the previous versions of SQLAlchemy it was possible to use `postgres://`, but using it in SQLAlchemy 1.4.0+ results in:

```

>      raise exc.NoSuchModuleError(
        "Can't load plugin: %s:%s" % (self.group, name)
    )
E      sqlalchemy.exc.NoSuchModuleError: Can't load plugin: sqlalchemy.
→dialects:postgres

```

If you cannot change the prefix of your URL immediately, Airflow continues to work with SQLAlchemy 1.3 and

you can downgrade SQLAlchemy, but we recommend to update the prefix.

Details in the [SQLAlchemy Changelog](#).

We recommend using the `psycopg2` driver and specifying it in your SQLAlchemy connection string.

```
postgresql+psycopg2://<user>:<password>@<host>/<db>
```

Also note that since SQLAlchemy does not expose a way to target a specific schema in the database URI, you need to ensure schema `public` is in your Postgres user's search_path.

If you created a new Postgres account for Airflow:

- The default search_path for new Postgres user is: `"$user", public`, no change is needed.

If you use a current Postgres user with custom search_path, search_path can be changed by the command:

```
ALTER USER airflow_user SET search_path = public;
```

For more information regarding setup of the PostgreSQL connection, see [PostgreSQL dialect](#) in SQLAlchemy documentation.

Note

Airflow is known - especially in high-performance setup - to open many connections to metadata database. This might cause problems for Postgres resource usage, because in Postgres, each connection creates a new process and it makes Postgres resource-hungry when a lot of connections are opened. Therefore we recommend to use [PGBouncer](#) as database proxy for all Postgres production installations. PGBouncer can handle connection pooling from multiple components, but also in case you have remote database with potentially unstable connectivity, it will make your DB connectivity much more resilient to temporary network problems. Example implementation of PGBouncer deployment can be found in the `helm-chart:index` where you can enable pre-configured PGBouncer instance with flipping a boolean flag. You can take a look at the approach we have taken there and use it as an inspiration, when you prepare your own Deployment, even if you do not use the Official Helm Chart.

See also Helm Chart production guide

Note

For managed Postgres such as Azure Postgresql, CloudSQL, Amazon RDS, you should use `keepalives_idle` in the connection parameters and set it to less than the idle time because those services will close idle connections after some time of inactivity (typically 300 seconds), which results with error `The error: psycopg2.OperationalError: SSL SYSCALL error: EOF detected`. The `keepalive` settings can be changed via `sqlalchemy_connect_args` configuration parameter [Configuration Reference](#) in `[database]` section. You can configure the args for example in your `local_settings.py` and the `sqlalchemy_connect_args` should be a full import path to the dictionary that stores the configuration parameters. You can read about [Postgres Keepalives](#). An example setup for `keepalives` that has been observed to fix the problem might be:

```
keepalive_kwargs = {
    "keepalives": 1,
    "keepalives_idle": 30,
    "keepalives_interval": 5,
    "keepalives_count": 5,
}
```

Then, if it were placed in `airflow_local_settings.py`, the config import path would be:

```
sql_alchemy_connect_args = airflow_local_settings.keepalive_kwargs
```

See *Configuring local settings* for details on how to configure local settings.

Setting up a MySQL Database

You need to create a database and a database user that Airflow will use to access this database. In the example below, a database `airflow_db` and user with username `airflow_user` with password `airflow_pass` will be created

```
CREATE DATABASE airflow_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
CREATE USER 'airflow_user' IDENTIFIED BY 'airflow_pass';
GRANT ALL PRIVILEGES ON airflow_db.* TO 'airflow_user';
```

Note

The database must use a UTF-8 character set. A small caveat that you must be aware of is that `utf8` in newer versions of MySQL is really `utf8mb4` which causes Airflow indexes to grow too large (see <https://github.com/apache/airflow/pull/17603#issuecomment-901121618>). Therefore as of Airflow 2.2 all MySQL databases have `sql_engine_collation_for_ids` set automatically to `utf8mb3_bin` (unless you override it). This might lead to a mixture of collation ids for id fields in Airflow Database, but it has no negative consequences since all relevant IDs in Airflow use ASCII characters only.

We rely on more strict ANSI SQL settings for MySQL in order to have sane defaults. Make sure to have specified `explicit_defaults_for_timestamp=1` option under `[mysqld]` section in your `my.cnf` file. You can also activate these options with the `--explicit-defaults-for-timestamp` switch passed to `mysqld` executable

We recommend using the `mysqlclient` driver and specifying it in your SQLAlchemy connection string.

```
mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
```

Important

The integration of MySQL backend has only been validated using the `mysqlclient` driver during Apache Airflow's continuous integration (CI) process.

If you want to use other drivers visit the [MySQL Dialect](#) in SQLAlchemy documentation for more information regarding download and setup of the SQLAlchemy connection.

In addition, you also should pay particular attention to MySQL's encoding. Although the `utf8mb4` character set is more and more popular for MySQL (actually, `utf8mb4` becomes default character set in MySQL8.0), using the `utf8mb4` encoding requires additional setting in Airflow 2+ (See more details in [#7570](#).). If you use `utf8mb4` as character set, you should also set `sql_engine_collation_for_ids=utf8mb3_bin`.

Note

In strict mode, MySQL doesn't allow `0000-00-00` as a valid date. Then you might get errors like "Invalid default value for 'end_date'" in some cases (some Airflow tables use `0000-00-00 00:00:00` as timestamp field default value). To avoid this error, you could disable `NO_ZERO_DATE` mode on your MySQL server.

Read <https://stackoverflow.com/questions/9192027/invalid-default-value-for-create-date-timestamp-field> for how to disable it. See **SQL Mode - NO_ZERO_DATE** for more information.

MsSQL Database

⚠ Warning

After discussion and a voting process, the Airflow's PMC members and Committers have reached a resolution to no longer maintain MsSQL as a supported Database Backend.

As of Airflow 2.9.0 support of MsSQL has been removed for Airflow Database Backend. This does not affect the existing providers (operators and hooks), dags can still access and process data from MsSQL. However, further usage may throw errors making Airflow's core functionality unusable.

Migrating off MsSQL Server

As with Airflow 2.9.0 the support of MSSQL has ended, a migration script can help with Airflow version 2.7.x or 2.8.x to migrate off SQL-Server. The migration script is available in [airflow-mssql-migration](#) repo on Github.

Note that the migration script is provided without support and warranty.

Other configuration options

There are more configuration options for configuring SQLAlchemy behavior. For details, see *reference documentation* for `sqlalchemy_*` option in [database] section.

For instance, you can specify a database schema where Airflow will create its required tables. If you want Airflow to install its tables in the `airflow` schema of a PostgreSQL database, specify these environment variables:

```
export AIRFLOW__DATABASE__SQLALCHEMY_CONN="postgresql://postgres@localhost:5432/my_
˓→database?options=-csearch_path%3Dairflow"
export AIRFLOW__DATABASE__SQLALCHEMY_SCHEMA="airflow"
```

Note the `search_path` at the end of the `SQLALCHEMY_CONN` database URL.

Initialize the database

After configuring the database and connecting to it in Airflow configuration, you should create the database schema.

```
airflow db migrate
```

Database Monitoring and Maintenance in Airflow

Airflow extensively utilizes a relational metadata database for task scheduling and execution. Monitoring and proper configuration of this database are crucial for optimal Airflow performance.

Key Concerns

1. **Performance Impact:** Long or excessive queries can significantly affect Airflow's functionality. These may arise due to workflow specifics, lack of optimizations, or code bugs.
2. **Database Statistics:** Incorrect optimization decisions by the database engine, often due to outdated data statistics, can degrade performance.

Responsibilities

The responsibilities for database monitoring and maintenance in Airflow environments vary depending on whether you're using self-managed databases and Airflow instances or opting for managed services.

Self-Managed Environments:

In the setups where both the database and Airflow are self-managed, the Deployment Manager is responsible for setting up, configuring, and maintaining the database. This includes monitoring its performance, managing backups, periodic cleanups and ensuring its optimal operation with Airflow.

Managed Services:

- **Managed Database Services:** When using managed DB services, many maintenance tasks (like backups, patching, and basic monitoring) are handled by the provider. However, the Deployment Manager still needs to oversee the configuration of Airflow and optimize performance settings specific to their workflows, manages periodic cleanups and monitor their DB to ensure optimal operations with Airflow.
- **Managed Airflow Services:** With managed Airflow services, those service provider take responsibility for the configuration and maintenance of Airflow and its database. However, the Deployment Manager needs to collaborate with the service configuration to ensure that the sizing and workflow requirements are matching the sizing and configuration of the managed service.

Monitoring Aspects

Regular monitoring should include:

- CPU, I/O, and memory usage.
- Query frequency and number.
- Identification and logging of slow or long-running queries.
- Detection of inefficient query execution plans.
- Analysis of disk swap versus memory usage and cache swapping frequency.

Tools and Strategies

- Airflow doesn't provide direct tooling for database monitoring.
- Use server-side monitoring and logging to obtain metrics.
- Enable tracking of long-running queries based on defined thresholds.
- Regularly run house-keeping tasks (like `ANALYZE` SQL command) for maintenance.

Database Cleaning Tools

- **Airflow DB Clean Command:** Utilize the `airflow db clean` command to help manage and clean up your database.
- **Python Methods in ``airflow.utils.db_cleanup``:** This module provides additional Python methods for database cleanup and maintenance, offering more fine-grained control and customization for specific needs.

Recommendations

- **Proactive Monitoring:** Implement monitoring and logging in production without significantly impacting performance.
- **Database-Specific Guidance:** Consult the chosen database's documentation for specific monitoring setup instructions.

- **Managed Database Services:** Check if automatic maintenance tasks are available with your database provider.

SQLAlchemy Logging

For detailed query analysis, enable SQLAlchemy client logging (echo=True in SQLAlchemy engine configuration).

- This method is more intrusive and can affect Airflow's client-side performance.
- It generates a lot of logs, especially in a busy Airflow environment.
- Suitable for non-production environments like staging systems.

You can do it with echo=True as sqlalchemy engine configuration as explained in the [SQLAlchemy logging documentation](#).

Use `sqlalchemy_engine_args` configuration parameter to set echo arg to True.

Caution

- Be mindful of the impact on Airflow's performance and system resources when enabling extensive logging.
- Prefer server-side monitoring over client-side logging for production environments to minimize performance interference.

What's next?

By default, Airflow uses LocalExecutor. You should consider configuring a different *executor* for better performance.

3.5.7 Using Operators

An operator represents a single, ideally idempotent, task. Operators determine what actually executes when your DAG runs.

Note

See the *Operators Concepts* documentation.

3.5.8 Customizing DAG Scheduling with Timetables

For our example, let's say a company wants to run a job after each weekday to process data collected during the work day. The first intuitive answer to this would be `schedule="0 0 * * 1-5"` (midnight on Monday to Friday), but this means data collected on Friday will *not* be processed right after Friday ends, but on the next Monday, and that run's interval would be from midnight Friday to midnight *Monday*. Further, the above schedule string cannot skip processing on holidays. What we want is:

- Schedule a run for each Monday, Tuesday, Wednesday, Thursday, and Friday. The run's data interval would cover from midnight of each day, to midnight of the next day (e.g. 2021-01-01 00:00:00 to 2021-01-02 00:00:00).
- Each run would be created right after the data interval ends. The run covering Monday happens on midnight Tuesday and so on. The run covering Friday happens on midnight Saturday. No runs happen on midnights Sunday and Monday.
- Do not schedule a run on defined holidays.

For simplicity, we will only deal with UTC datetimes in this example.

Note

All datetime values returned by a custom timetable **MUST** be “aware”, i.e. contains timezone information. Furthermore, they must use pendulum’s datetime and timezone types.

Timetable Registration

A timetable must be a subclass of `Timetable`, and be registered as a part of a *plugin*. The following is a skeleton for us to implement a new timetable:

```
from airflow.plugins_manager import AirflowPlugin
from airflow.timetables.base import Timetable

class AfterWorkdayTimetable(Timetable):
    pass

class WorkdayTimetablePlugin(AirflowPlugin):
    name = "workday_timetable_plugin"
    timetables = [AfterWorkdayTimetable]
```

Next, we’ll start putting code into `AfterWorkdayTimetable`. After the implementation is finished, we should be able to use the timetable in our DAG file:

```
import pendulum

from airflow.sdk import DAG
from airflow.example_dags.plugins.workday import AfterWorkdayTimetable

with DAG(
    dag_id="example_after_workday_timetable_dag",
    start_date=pendulum.datetime(2021, 3, 10, tz="UTC"),
    schedule=AfterWorkdayTimetable(),
    tags=["example", "timetable"],
):
    ...
```

Define Scheduling Logic

When Airflow’s scheduler encounters a DAG, it calls one of the two methods to know when to schedule the DAG’s next run.

- `next_dagrun_info`: The scheduler uses this to learn the timetable’s regular schedule, i.e. the “one for every workday, run at the end of it” part in our example.
- `infer_manual_data_interval`: When a DAG run is manually triggered (from the web UI, for example), the scheduler uses this method to learn about how to reverse-infer the out-of-schedule run’s data interval.

We’ll start with `infer_manual_data_interval` since it’s the easier of the two:

`airflow/example_dags/plugins/workday.py`

```
def infer_manual_data_interval(self, run_after: DateTime) -> DataInterval:
    start = DateTime.combine((run_after - timedelta(days=1)).date(), Time.min).
    ↪replace(tzinfo=UTC)
    # Skip backwards over weekends and holidays to find last run
    start = self.get_next_workday(start, incr=-1)
    return DataInterval(start=start, end=(start + timedelta(days=1)))
```

The method accepts one argument `run_after`, a `pendulum.DateTime` object that indicates when the DAG is externally triggered. Since our timetable creates a data interval for each complete work day, the data interval inferred here should usually start at the midnight one day prior to `run_after`, but if `run_after` falls on a Sunday or Monday (i.e. the prior day is Saturday or Sunday), it should be pushed further back to the previous Friday. Once we know the start of the interval, the end is simply one full day after it. We then create a `DataInterval` object to describe this interval.

Next is the implementation of `next_dagrun_info`:

airflow/example_dags/plugins/workday.py

```
def next_dagrun_info(
    self,
    *,
    last_automated_data_interval: DataInterval | None,
    restriction: TimeRestriction,
) -> DagRunInfo | None:
    if last_automated_data_interval is not None: # There was a previous run on the
    ↪regular schedule.
        last_start = last_automated_data_interval.start
        next_start = DateTime.combine((last_start + timedelta(days=1)).date(), Time.min)
    # Otherwise this is the first ever run on the regular schedule...
    elif (earliest := restriction.earliest) is None:
        return None # No start_date. Don't schedule.
    elif not restriction.catchup:
        # If the DAG has catchup=False, today is the earliest to consider.
        next_start = max(earliest, DateTime.combine(Date.today(), Time.min, tzinfo=UTC))
    elif earliest.time() != Time.min:
        # If earliest does not fall on midnight, skip to the next day.
        next_start = DateTime.combine(earliest.date() + timedelta(days=1), Time.min)
    else:
        next_start = earliest
    # Skip weekends and holidays
    next_start = self.get_next_workday(next_start.replace(tzinfo=UTC))

    if restriction.latest is not None and next_start > restriction.latest:
        return None # Over the DAG's scheduled end; don't schedule.
    return DagRunInfo.interval(start=next_start, end=(next_start + timedelta(days=1)))
```

This method accepts two arguments. `last_automated_data_interval` is a `DataInterval` instance indicating the data interval of this DAG's previous non-manually-triggered run, or `None` if this is the first time ever the DAG is being scheduled. `restriction` encapsulates how the DAG and its tasks specify the schedule, and contains three attributes:

- `earliest`: The earliest time the DAG may be scheduled. This is a `pendulum.DateTime` calculated from all the `start_date` arguments from the DAG and its tasks, or `None` if there are no `start_date` arguments found at all.
- `latest`: Similar to `earliest`, this is the latest time the DAG may be scheduled, calculated from `end_date`

arguments.

- **catchup**: A boolean reflecting the DAG’s `catchup` argument. Defaults to `False`.

Note

Both `earliest` and `latest` apply to the DAG run’s logical date (the `start` of the data interval), not when the run will be scheduled (usually after the end of the data interval).

If there was a run scheduled previously, we should now schedule for the next non-holiday weekday by looping through subsequent days to find one that is not a Saturday, Sunday, or US holiday. If there was not a previous scheduled run, however, we pick the next non-holiday workday’s midnight after `restriction.earliest`. `restriction.catchup` also needs to be considered—if it’s `False`, we can’t schedule before the current time, even if `start_date` values are in the past. Finally, if our calculated data interval is later than `restriction.latest`, we must respect it and not schedule a run by returning `None`.

If we decide to schedule a run, we need to describe it with a `DagRunInfo`. This type has two arguments and attributes:

- `data_interval`: A `DataInterval` instance describing the next run’s data interval.
- `run_after`: A `pendulum.DateTime` instance that tells the scheduler when the DAG run can be scheduled.

A `DagRunInfo` can be created like this:

```
info = DagRunInfo(
    data_interval=DataInterval(start=start, end=end),
    run_after=run_after,
)
```

Since we typically want to schedule a run as soon as the data interval ends, `end` and `run_after` above are generally the same. `DagRunInfo` therefore provides a shortcut for this:

```
info = DagRunInfo.interval(start=start, end=end)
assert info.data_interval.end == info.run_after # Always True.
```

For reference, here’s our plugin and DAG files in their entirety:

airflow/example_dags/plugins/workday.py

```
from pendulum import UTC, Date, DateTime, Time

from airflow.plugins_manager import AirflowPlugin
from airflow.timetables.base import DagRunInfo, DataInterval, Timetable

if TYPE_CHECKING:
    from airflow.timetables.base import TimeRestriction

log = logging.getLogger(__name__)
try:
    from pandas.tseries.holiday import USFederalHolidayCalendar

    holiday_calendar = USFederalHolidayCalendar()
except ImportError:
    log.warning("Could not import pandas. Holidays will not be considered.")
    holiday_calendar = None # type: ignore[assignment]
```

(continues on next page)

(continued from previous page)

```

class AfterWorkdayTimetable(Timetable):
    def get_next_workday(self, d: DateTime, incr=1) -> DateTime:
        next_start = d
        while True:
            if next_start.weekday() not in (5, 6): # not on weekend
                if holiday_calendar is None:
                    holidays = set()
                else:
                    holidays = holiday_calendar.holidays(start=next_start, end=next_
→start).to_pydatetime()
                if next_start not in holidays:
                    break
            next_start = next_start.add(days=incr)
        return next_start
    def infer_manual_data_interval(self, run_after: DateTime) -> DataInterval:
        start = DateTime.combine((run_after - timedelta(days=1)).date(), Time.min).
→replace(tzinfo=UTC)
        # Skip backwards over weekends and holidays to find last run
        start = self.get_next_workday(start, incr=-1)
        return DataInterval(start=start, end=(start + timedelta(days=1)))
    def next_dagrun_info(
        self,
        *,
        last_automated_data_interval: DataInterval | None,
        restriction: TimeRestriction,
    ) -> DagRunInfo | None:
        if last_automated_data_interval is not None: # There was a previous run on the_
→regular schedule.
            last_start = last_automated_data_interval.start
            next_start = DateTime.combine((last_start + timedelta(days=1)).date(), Time.
→min)
            # Otherwise this is the first ever run on the regular schedule...
            elif (earliest := restriction.earliest) is None:
                return None # No start_date. Don't schedule.
            elif not restriction.catchup:
                # If the DAG has catchup=False, today is the earliest to consider.
                next_start = max(earliest, DateTime.combine(Date.today(), Time.min,_
→tzinfo=UTC))
            elif earliest.time() != Time.min:
                # If earliest does not fall on midnight, skip to the next day.
                next_start = DateTime.combine(earliest.date() + timedelta(days=1), Time.min)
            else:
                next_start = earliest
            # Skip weekends and holidays
            next_start = self.get_next_workday(next_start.replace(tzinfo=UTC))

            if restriction.latest is not None and next_start > restriction.latest:
                return None # Over the DAG's scheduled end; don't schedule.
            return DagRunInfo.interval(start=next_start, end=(next_start +_
→timedelta(days=1)))

```

(continues on next page)

(continued from previous page)

```
class WorkdayTimetablePlugin(AirflowPlugin):
    name = "workday_timetable_plugin"
    timetables = [AfterWorkdayTimetable]
```

```
import pendulum

from airflow.sdk import DAG
from airflow.example_dags.plugins.workday import AfterWorkdayTimetable
from airflow.providers.standard.operators.empty import EmptyOperator

with DAG(
    dag_id="example_workday_timetable",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule=AfterWorkdayTimetable(),
    tags=["example", "timetable"],
):
    EmptyOperator(task_id="run_this")
```

Parameterized Timetables

Sometimes we need to pass some run-time arguments to the timetable. Continuing with our `AfterWorkdayTimetable` example, maybe we have dags running on different timezones, and we want to schedule some dags at 8am the next day, instead of on midnight. Instead of creating a separate timetable for each purpose, we'd want to do something like:

```
class SometimeAfterWorkdayTimetable(Timetable):
    def __init__(self, schedule_at: Time) -> None:
        self._schedule_at = schedule_at

    def next_dagrun_info(self, last_automated_dagrun, restriction):
        ...
        end = start + timedelta(days=1)
        return DagRunInfo(
            data_interval=DataInterval(start=start, end=end),
            run_after=DateTime.combine(end.date(), self._schedule_at).
        ↪replace(tzinfo=UTC),
        )
```

However, since the timetable is a part of the DAG, we need to tell Airflow how to serialize it with the context we provide in `__init__`. This is done by implementing two additional methods on our timetable class:

```
class SometimeAfterWorkdayTimetable(Timetable):
    ...

    def serialize(self) -> dict[str, Any]:
        return {"schedule_at": self._schedule_at.isoformat()}

    @classmethod
    def deserialize(cls, value: dict[str, Any]) -> Timetable:
```

(continues on next page)

(continued from previous page)

```
return cls(Time.fromisoformat(value["schedule_at"]))
```

When the DAG is being serialized, `serialize` is called to obtain a JSON-serializable value. That value is passed to `deserialize` when the serialized DAG is accessed by the scheduler to reconstruct the timetable.

Timetable Display in UI

By default, a custom timetable is displayed by their class name in the UI (e.g. the `Schedule` column in the “dags” table). It is possible to customize this by overriding the `summary` property. This is especially useful for parameterized timetables to include arguments provided in `__init__`. For our `SometimeAfterWorkdayTimetable` class, for example, we could have:

```
@property
def summary(self) -> str:
    return f"after each workday, at {self._schedule_at}"
```

So for a DAG declared like this:

```
with DAG(
    schedule=SometimeAfterWorkdayTimetable(Time(8)), # 8am.
    ...,
):
    ...
```

The `Schedule` column would say `after each workday, at 08:00:00`.

See also

Module `airflow.timetables.base`

The public interface is heavily documented to explain what should be implemented by subclasses.

Timetable Description Display in UI

You can also provide a description for your Timetable Implementation by overriding the `description` property. This is especially useful for providing comprehensive description for your implementation in UI. For our `SometimeAfterWorkdayTimetable` class, for example, we could have:

```
description = "Schedule: after each workday"
```

You can also wrap this inside `__init__`, if you want to derive description.

```
def __init__(self) -> None:
    self.description = "Schedule: after each workday, at f{self._schedule_at}"
```

This is specially useful when you want to provide comprehensive description which is different from `summary` property.

So for a DAG declared like this:

```
with DAG(
    schedule=SometimeAfterWorkdayTimetable(Time(8)), # 8am.
    ...,
):
    ...
```

The *i* icon would show, Schedule: after each workday, at 08:00:00.

See also

Module `airflow.timetables.interval`

check CronDataIntervalTimetable description implementation which provides comprehensive cron description in UI.

Changing generated run_id

Added in version 2.4.

Since Airflow 2.4, Timetables are also responsible for generating the `run_id` for DagRuns.

For example to have the Run ID show a “human friendly” date of when the run started (that is, the end of the data interval, rather than the start which is the date currently used) you could add a method like this to a custom timetable:

```
def generate_run_id(
    self,
    *,
    run_type: DagRunType,
    logical_date: DateTime,
    data_interval: DataInterval | None,
    **extra,
) -> str:
    if run_type == DagRunType.SCHEDULED and data_interval:
        return data_interval.end.format("YYYY-MM-DD dddd")
    return super().generate_run_id(
        run_type=run_type, logical_date=logical_date, data_interval=data_interval,
    **extra
)
```

Remember that the RunID is limited to 250 characters, and must be unique within a DAG.

3.5.9 Customize view of Apache from Airflow web UI

Airflow has feature that allows to integrate a custom UI along with its core UI using the Plugin manager.

Plugins integrate with the Airflow core RestAPI. In this plugin, two object references are derived from the base class `airflow.plugins_manager.AirflowPlugin`. They are `fastapi_apps` and `fastapi_root_middlewares`.

Using `fastapi_apps` in Airflow plugin, the core RestAPI can be extended to support extra endpoints to serve custom static file or any other json/application responses. In this object reference, the list of dictionaries with FastAPI application and metadata information like the name and the url prefix are passed on.

Using `fastapi_root_middlewares` in Airflow plugin, allows to register custom middleware at the root of the FastAPI application. This middleware can be used to add custom headers, logging, or any other functionality to the entire FastAPI application, including core endpoints. In this object reference, the list of dictionaries with Middleware factories object, initialization parameters and some metadata information like the name are passed on.

Information and code samples to register `fastapi_apps` and `fastapi_root_middlewares` are available in *plugin*.

3.5.10 Support for Airflow 2 plugins

Airflow 2 plugins are still supported with some limitations. More information on such plugins can be found in the Airflow 2 documentation.

Adding Rest endpoints through the blueprints is still supported, those endpoints will be integrated in the FastAPI application via the WSGI Middleware and accessible under /pluginsv2.

It is not possible to extend the core UI, for instance by extending the base template, nonetheless extra menu items of the auth managers are added to the core UI security tab and their href are rendered in iframes. This is how the fab provider integrates users, roles, actions, resources and permissions custom views in the Airflow 3 UI.

Airflow 3 plugins will be improved to allow UI customization for the entire react app, it is recommended to upgrade your plugins to Airflow 3 plugins when possible. Until then for a temporary or custom needs it is possible to use a Middleware to inject custom javascript or css to the core UI index request.

3.5.11 Listener Plugin of Airflow

Airflow has feature that allows to add listener for monitoring and tracking the task state using Plugins.

This is a simple example listener plugin of Airflow that helps to track the task state and collect useful metadata information about the task, dag run and dag.

This is an example plugin for Airflow that allows to create listener plugin of Airflow. This plugin works by using SQLAlchemy's event mechanism. It watches the task instance state change in the table level and triggers event. This will be notified for all the tasks across all the dags.

In this plugin, an object reference is derived from the base class `airflow.plugins_manager.AirflowPlugin`.

Listener plugin uses pluggy app under the hood. Pluggy is an app built for plugin management and hook calling for Pytest. Pluggy enables function hooking so it allows building “pluggable” systems with your own customization over that hooking.

Using this plugin, following events can be listened:

- task instance is in running state.
- task instance is in success state.
- task instance is in failure state.
- dag run is in running state.
- dag run is in success state.
- dag run is in failure state.
- on start before event like Airflow job, scheduler
- before stop for event like Airflow job, scheduler

Listener Registration

A listener plugin with object reference to listener object is registered as part of Airflow plugin. The following is a skeleton for us to implement a new listener:

```
from airflow.plugins_manager import AirflowPlugin

# This is the listener file created where custom code to monitor is added over hookimpl
import listener
```

(continues on next page)

(continued from previous page)

```
class MetadataCollectionPlugin(AirflowPlugin):
    name = "MetadataCollectionPlugin"
    listeners = [listener]
```

Next, we can check code added into `listener` and see implementation methods for each of those listeners. After the implementation, the listener part gets executed during all the task execution across all the dags

For reference, here's the plugin code within `listener.py` class that shows list of tables in the database:

This example listens when the task instance is in running state

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_task_instance_running(previous_state: TaskInstanceState, task_instance: RuntimeTaskInstance):
    """
    Called when task state changes to RUNNING.

    previous_task_state and task_instance object can be used to retrieve more
    information about current
    task_instance that is running, its dag_run, task and dag information.
    """

    print("Task instance is in running state")
    print(f" Previous state of the Task instance:", previous_state)

    name: str = task_instance.task_id

    context = task_instance.get_template_context()

    task = context["task"]

    if TYPE_CHECKING:
        assert task

    dag = task.dag
    dag_name = None
    if dag:
        dag_name = dag.dag_id
    print(f"Current task name:{name}")
    print(f"Dag name:{dag_name}")
```

Similarly, code to listen after `task_instance` success and failure can be implemented.

This example listens when the dag run is change to failed state

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_dag_run_failed(dag_run: DagRun, msg: str):
    """
    This method is called when dag run state changes to FAILED.
```

(continues on next page)

(continued from previous page)

```
"""
print("Dag run in failure state")
dag_id = dag_run.dag_id
run_id = dag_run.run_id
run_type = dag_run.run_type

print(f"Dag information: {dag_id} Run id: {run_id} Run type: {run_type}")
print(f"Failed with message: {msg}")
```

Similarly, code to listen after dag_run success and during running state can be implemented.

The listener plugin files required to add the listener implementation is added as part of the Airflow plugin into \$AIRFLOW_HOME/plugins/ folder and loaded during Airflow startup.

3.5.12 Customizing the UI

Customizing DAG UI Header and Airflow Page Titles

Airflow now allows you to customize the DAG home page header and page title. This will help distinguish between various installations of Airflow or simply amend the page text.

Note

The custom title will be applied to both the page header and the page title.

To make this change, simply:

1. Add the configuration option of `instance_name` under the `[webserver]` section inside `airflow.cfg`:

```
[webserver]
instance_name = "DevEnv"
```

2. Alternatively, you can set a custom title using the environment variable:

```
AIRFLOW__WEBSERVER__INSTANCE_NAME = "DevEnv"
```

Screenshots

Before

DAGs

| All 59 | Active 6 | Paused 53 | Running 0 | Failed 0 | Filter DAGs by tag |
|--|----------|-----------|---------------------|----------------------|--------------------|
| <i>DAG</i> | Owner | Runs | Schedule | Last Run | |
| conditional_dataset_and_time_based_timetable | airflow | ○○○○ | Dataset or 0 1 ** 3 | | |
| consume_1_and_2_with_dataset_expressions | airflow | ○○○○ | Dataset | | |
| consume_1_or_2_with_dataset_expressions | airflow | ○○○○ | Dataset | 2024-03-24, 11:53:13 | |

After

DevEnv

| All 59 | Active 6 | Paused 53 | Running 0 | Failed 0 | Filter DAGs by tag |
|--|----------|-----------|---------------------|----------------------|--------------------|
| <i>DAG</i> | Owner | Runs | Schedule | Last Run | |
| conditional_dataset_and_time_based_timetable | airflow | ○○○○ | Dataset or 0 1 ** 3 | | |
| consume_1_and_2_with_dataset_expressions | airflow | ○○○○ | Dataset | | |
| consume_1_or_2_with_dataset_expressions | airflow | ○○○○ | Dataset | 2024-03-24, 11:53:13 | |

Note

From version 2.3.0 you can include markup in `instance_name` variable for further customization. To enable, set `instance_name_has_markup` under the `[webserver]` section inside `airflow.cfg` to `True`.

Add custom alert messages on the dashboard

Extra alert messages can be shown on the UI dashboard. This can be useful for warning about setup issues or announcing changes to end users. The following example shows how to add alert messages:

1. Add the following contents to `airflow_local_settings.py` file under `$AIRFLOW_HOME/config`. Each alert message should specify a severity level (`info`, `warning`, `error`) using `category`.

```
from airflow.api.fastapi.common.types import UIAlert

DASHBOARD_UIALERTS = [
    UIAlert(text="Welcome to Airflow.", category="info"),
    UIAlert(text="Airflow server downtime scheduled for tomorrow at 10:00 AM.", category="warning")
]

```

(continues on next page)

(continued from previous page)

```
↳category="warning"),
UIAlert(text="Critical error detected!", category="error"),
]
```

See *Configuring local settings* for details on how to configure local settings.

2. Restart Airflow Webserver, and you should now see:

The screenshot shows the Airflow web interface with a sidebar on the left containing 'Home', 'Dags', 'Assets', and 'Browse' buttons. The main area has a light blue header bar with the text 'Welcome to Airflow.'. Below it is an orange bar with the text 'Airflow server downtime scheduled for tomorrow at 10:00 AM.'. At the bottom of the screen is a pink bar with the text 'Critical error detected!'. The central part of the screen is titled 'Welcome' and includes sections for 'Stats', 'Failed dags' (0), 'Running dags' (0), and 'Active dags' (0).

Alert messages also support Markdown. In the following example, we show an alert message of heading 2 with a link included.

```
DASHBOARD_UIALERTS = [
    UIAlert(text="## Visit [airflow.apache.org](https://airflow.apache.org)", ↳
    ↳category="info"),
]
```

The screenshot shows the Airflow web interface with a sidebar on the left containing 'Home', 'Dags', 'Assets', and 'Browse' buttons. The main area has a light blue header bar with the text 'Visit airflow.apache.org'. Below it is a green bar with the text 'Welcome'. The central part of the screen includes sections for 'Stats', 'Failed dags' (0), 'Running dags' (0), and 'Active dags' (0). At the bottom, there is a green progress bar labeled 'Manage Pools' with the value '✓ 128'.

3.5.13 Creating a custom Operator

Airflow allows you to create new operators to suit the requirements of you or your team. This extensibility is one of the many features which make Apache Airflow powerful.

You can create any operator you want by extending the `airflow.models.baseoperator.BaseOperator`

There are two methods that you need to override in a derived class:

- Constructor - Define the parameters required for the operator. You only need to specify the arguments specific to your operator. You can specify the `default_args` in the DAG file. See *Default args* for more details.
- Execute - The code to execute when the runner calls the operator. The method contains the Airflow context as a parameter that can be used to read config values.

Note

When implementing custom operators, do not make any expensive operations in the `__init__` method. The operators will be instantiated once per scheduler cycle per task using them, and making database calls can significantly

slow down scheduling and waste resources.

Let's implement an example `HelloOperator` in a new file `hello_operator.py`:

```
from airflow.sdk import BaseOperator

class HelloOperator(BaseOperator):
    def __init__(self, name: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.name = name

    def execute(self, context):
        message = f"Hello {self.name}"
        print(message)
        return message
```

Note

For imports to work, you should place the file in a directory that is present in the `PYTHONPATH` env. Airflow adds `dags/`, `plugins/`, and `config/` directories in the Airflow home to `PYTHONPATH` by default. e.g., In our example, the file is placed in the `custom_operator/` directory. See *Modules Management* for details on how Python and Airflow manage modules.

You can now use the derived custom operator as follows:

```
from custom_operator.hello_operator import HelloOperator

with dag:
    hello_task = HelloOperator(task_id="sample-task", name="foo_bar")
```

You also can keep using your plugins folder for storing your custom operators. If you have the file `hello_operator.py` within the `plugins` folder, you can import the operator as follows:

```
from hello_operator import HelloOperator
```

If an operator communicates with an external service (API, database, etc) it's a good idea to implement the communication layer using a *Hooks*. In this way the implemented logic can be reused by other users in different operators. Such approach provides better decoupling and utilization of added integration than using `CustomServiceBaseOperator` for each external service.

Other consideration is the temporary state. If an operation requires an in-memory state (for example a job id that should be used in `on_kill` method to cancel a request) then the state should be kept in the operator not in a hook. In this way the service hook can be completely state-less and whole logic of an operation is in one place - in the operator.

Hooks

Hooks act as an interface to communicate with the external shared resources in a DAG. For example, multiple tasks in a DAG can require access to a MySQL database. Instead of creating a connection per task, you can retrieve a connection from the hook and utilize it. Hook also helps to avoid storing connection auth parameters in a DAG. See *Managing Connections* for how to create and manage connections and `apache-airflow-providers:index` for details of how to add your custom connection types via providers.

Let's extend our previous example to fetch name from MySQL:

```
class HelloDBOperator(BaseOperator):
    def __init__(self, name: str, mysql_conn_id: str, database: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.name = name
        self.mysql_conn_id = mysql_conn_id
        self.database = database

    def execute(self, context):
        hook = MySqlHook(mysql_conn_id=self.mysql_conn_id, schema=self.database)
        sql = "select name from user"
        result = hook.get_first(sql)
        message = f"Hello {result['name']}"
        print(message)
        return message
```

When the operator invokes the query on the hook object, a new connection gets created if it doesn't exist. The hook retrieves the auth parameters such as username and password from Airflow backend and passes the params to the `airflow.hooks.base.BaseHook.get_connection()`. You should create hook only in the `execute` method or any method which is called from `execute`. The constructor gets called whenever Airflow parses a DAG which happens frequently. And instantiating a hook there will result in many unnecessary database connections. The `execute` gets called only during a DAG run.

User interface

Airflow also allows the developer to control how the operator shows up in the DAG UI. Override `ui_color` to change the background color of the operator in UI. Override `ui_fgcolor` to change the color of the label. Override `custom_operator_name` to change the displayed name to something other than the classname.

```
class HelloOperator(BaseOperator):
    ui_color = "#ff0000"
    ui_fgcolor = "#000000"
    custom_operator_name = "Howdy"
    # ...
```

Templating

You can use *Jinja templates* to parameterize your operator. Airflow considers the field names present in `template_fields` for templating while rendering the operator.

```
class HelloOperator(BaseOperator):
    template_fields: Sequence[str] = ("name",)

    def __init__(self, name: str, world: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.name = name
        self.world = world

    def execute(self, context):
        message = f"Hello {self.world} it's {self.name}!"
        print(message)
        return message
```

You can use the template as follows:

```
with dag:
    hello_task = HelloOperator(
        task_id="task_id_1",
        name="{{ task_instance.task_id }}",
        world="Earth",
    )
```

In this example, Jinja looks for the `name` parameter and substitutes `{{ task_instance.task_id }}` with `task_id_1`.

The parameter can also contain a file name, for example, a bash script or a SQL file. You need to add the extension of your file in `template_ext`. If a `template_field` contains a string ending with the extension mentioned in `template_ext`, Jinja reads the content of the file and replace the templates with actual value. Note that Jinja substitutes the operator attributes and not the args.

```
class HelloOperator(BaseOperator):
    template_fields: Sequence[str] = ("guest_name",)
    template_ext = ".sql"

    def __init__(self, name: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.guest_name = name
```

In the example, the `template_fields` should be `['guest_name']` and not `['name']`

Additionally you may provide `template_fields_renderers` a dictionary which defines in what style the value from template field renders in Web UI. For example:

```
class MyRequestOperator(BaseOperator):
    template_fields: Sequence[str] = ("request_body",)
    template_fields_renderers = {"request_body": "json"}

    def __init__(self, request_body: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.request_body = request_body
```

In the situation where `template_field` is itself a dictionary, it is also possible to specify a dot-separated key path to extract and render individual elements appropriately. For example:

```
class MyConfigOperator(BaseOperator):
    template_fields: Sequence[str] = ("configuration",)
    template_fields_renderers = {
        "configuration": "json",
        "configuration.query.sql": "sql",
    }

    def __init__(self, configuration: dict, **kwargs) -> None:
        super().__init__(**kwargs)
        self.configuration = configuration
```

Then using this template as follows:

```
with dag:
    config_task = MyConfigOperator(
```

(continues on next page)

(continued from previous page)

```
task_id="task_id_1",
configuration={"query": {"job_id": "123", "sql": "select * from my_table"}},  
)
```

This will result in the UI rendering configuration as json in addition to the value contained in the configuration at query.sql to be rendered with the SQL lexer.

Rendered Template

configuration

```
1 {
2     "query": {
3         "job_id": "123",
4         "sql": "select * from my_table"
5     }
6 }
```

configuration.query.sql

```
1 select * from my_table
```

Currently available lexers:

- bash
- bash_command
- doc
- doc_json
- doc_md
- doc_rst
- doc_yaml
- doc_md
- hql
- html
- jinja
- json
- md
- mysql
- postgresql
- powershell
- py
- python_callable

- rst
- sql
- tsql
- yaml

If you use a non-existing lexer then the value of the template field will be rendered as a pretty-printed object.

Limitations

To prevent misuse, the following limitations must be observed when defining and assigning templated fields in the operator's constructor (when such exists, otherwise - see below):

1. Tempered fields' corresponding parameters passed into the constructor must be named exactly as the fields. The following example is invalid, as the parameter passed into the constructor is not the same as the templated field:

```
class HelloOperator(BaseOperator):
    template_fields = "foo"

    def __init__(self, foo_id) -> None: # should be def __init__(self, foo) -> None
        self.foo = foo_id # should be self.foo = foo
```

2. Tempered fields' instance members must be assigned with their corresponding parameter from the constructor, either by a direct assignment or by calling the parent's constructor (in which these fields are defined as `template_fields`) with explicit an assignment of the parameter. The following example is invalid, as the instance member `self.foo` is not assigned at all, despite being a tempered field:

```
class HelloOperator(BaseOperator):
    template_fields = ("foo", "bar")

    def __init__(self, foo, bar) -> None:
        self.bar = bar
```

The following example is also invalid, as the instance member `self.foo` of `MyHelloOperator` is initialized implicitly as part of the `kwargs` passed to its parent constructor:

```
class HelloOperator(BaseOperator):
    template_fields = "foo"

    def __init__(self, foo) -> None:
        self.foo = foo

class MyHelloOperator(HelloOperator):
    template_fields = ("foo", "bar")

    def __init__(self, bar, **kwargs) -> None: # should be def __init__(self, foo, bar, **kwargs)
        super().__init__(**kwargs) # should be super().__init__(foo=foo, **kwargs)
        self.bar = bar
```

3. Applying actions on the parameter during the assignment in the constructor is not allowed. Any action on the value should be applied in the `execute()` method. Therefore, the following example is invalid:

```
class HelloOperator(BaseOperator):
    template_fields = "foo"

    def __init__(self, foo) -> None:
        self.foo = foo.lower() # assignment should be only self.foo = foo
```

When an operator inherits from a base operator and does not have a constructor defined on its own, the limitations above do not apply. However, the templated fields must be set properly in the parent according to those limitations.

Thus, the following example is valid:

```
class HelloOperator(BaseOperator):
    template_fields = "foo"

    def __init__(self, foo) -> None:
        self.foo = foo

class MyHelloOperator(HelloOperator):
    template_fields = "foo"
```

The limitations above are enforced by a pre-commit named ‘validate-operators-init’.

Add template fields with subclassing

A common use case for creating a custom operator is for simply augmenting existing `template_fields`. There might be a situation in which an operator you wish to use doesn’t define certain parameters as templated, but you’d like to be able to dynamically pass an argument as a Jinja expression. This can easily be achieved with a quick subclassing of the existing operator.

Let’s assume you want to use the `HelloOperator` defined earlier:

```
class HelloOperator(BaseOperator):
    template_fields: Sequence[str] = ("name",)

    def __init__(self, name: str, world: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.name = name
        self.world = world

    def execute(self, context):
        message = f"Hello {self.world} it's {self.name}!"
        print(message)
        return message
```

However, you’d like to dynamically parameterize `world` arguments. Because the `template_fields` property is guaranteed to be a `Sequence[str]` type (i.e. a list or tuple of strings), you can subclass the `HelloOperator` to modify the `template_fields` as desired easily.

```
class MyHelloOperator(HelloOperator):
    template_fields: Sequence[str] = (*HelloOperator.template_fields, "world")
```

Now you can use `MyHelloOperator` like this:

```
with dag:
    hello_task = MyHelloOperator(
        task_id="task_id_1",
        name="{{ task_instance.task_id }}",
        world="{{ var.value.my_world }}",
    )
```

In this example, the `world` argument will be dynamically set to the value of an Airflow Variable named “`my_world`” via a Ninja expression.

Define an operator extra link

For your operator, you can *Define an extra link* that can redirect users to external systems. For example, you can add a link that redirects the user to the operator’s manual.

Sensors

Airflow provides a primitive for a special kind of operator, whose purpose is to poll some state (e.g. presence of a file) on a regular interval until a success criteria is met.

You can create any sensor you want by extending the `airflow.sensors.base.BaseSensorOperator` defining a `poke` method to poll your external state and evaluate the success criteria.

Sensors have a powerful feature called ‘reschedule’ mode which allows the sensor to task to be rescheduled, rather than blocking a worker slot between pokes. This is useful when you can tolerate a longer poll interval and expect to be polling for a long time.

Reschedule mode comes with a caveat that your sensor cannot maintain internal state between rescheduled executions. In this case you should decorate your sensor with `airflow.sensors.base.poke_mode_only()`. This will let users know that your sensor is not suitable for use with reschedule mode.

An example of a sensor that keeps internal state and cannot be used with reschedule mode is `airflow.providers.google.cloud.sensors.gcs.GCSUploadSessionCompleteSensor`. It polls the number of objects at a prefix (this number is the internal state of the sensor) and succeeds when there a certain amount of time has passed without the number of objects changing.

3.5.14 Creating Custom @task Decorators

As of Airflow 2.2 it is possible add custom decorators to the TaskFlow interface from within a provider package and have those decorators appear natively as part of the `@task._____` design.

For an example. Let’s say you were trying to create an easier mechanism to run python functions as “foo” tasks. The steps to create and register `@task.foo` are:

1. Create a `FooDecoratedOperator`

In this case, we are assuming that you have an existing `FooOperator` that takes a python function as an argument. By creating a `FooDecoratedOperator` that inherits from `FooOperator` and `airflow.decorators.base.DecoratedOperator`, Airflow will supply much of the needed functionality required to treat your new class as a taskflow native class.

You should also override the `custom_operator_name` attribute to provide a custom name for the task. For example, `_DockerDecoratedOperator` in the `apache-airflow-providers-docker` provider sets this to `@task.docker` to indicate the decorator name it implements.

2. Create a `foo_task` function

Once you have your decorated class, create a function like this, to convert the new `FooDecoratedOperator` into a TaskFlow function decorator!

```
from typing import TYPE_CHECKING
from airflow.sdk.bases.decorator import task_decorator_factory

if TYPE_CHECKING:
    from airflow.sdk.bases.decorator import TaskDecorator


def foo_task(
    python_callable: Callable | None = None,
    multiple_outputs: bool | None = None,
    **kwargs,
) -> "TaskDecorator":
    return task_decorator_factory(
        python_callable=python_callable,
        multiple_outputs=multiple_outputs,
        decorated_operator_class=FooDecoratedOperator,
        **kwargs,
    )
```

3. Register your new decorator in `get_provider_info` of your provider

Finally, add a key-value `task-decorators` to the dict returned from the provider entrypoint as described in apache-airflow-providers:howto/create-custom-providers. This should be a list with each item containing `name` and `class-name` keys. When Airflow starts, the `ProviderManager` class will automatically import this value and `task.foo` will work as a new decorator!

```
def get_provider_info():
    return {
        "package-name": "foo-provider-airflow",
        "name": "Foo",
        "task-decorators": [
            {
                "name": "foo",
                # "Import path" and function name of the `foo_task`
                "class-name": "name.of.python.package.foo_task",
            }
        ],
        # ...
    }
```

Please note that the `name` must be a valid python identifier.

3.5.15 (Optional) Adding IDE auto-completion support

 Note

This section mostly applies to the apache-airflow managed providers. We have not decided if we will allow third-party providers to register auto-completion in this way.

For better or worse, Python IDEs can not auto-complete dynamically generated methods (see JetBrains's write up on the subject).

To hack around this problem, a type stub `airflow/sdk/definitions/decorators/__init__.pyi` is provided to statically declare the type signature of each task decorator. A newly added task decorator should declare its signature stub like this:

`/build/src/repos/airflow/task-sdk/src/airflow/sdk/definitions/decorators/__init__.pyi`

```
def docker(
    self,
    *,
    multiple_outputs: bool | None = None,
    python_command: str = "python3",
    serializer: Literal["pickle", "cloudpickle", "dill"] | None = None,
    use_dill: bool = False, # Added by _DockerDecoratedOperator.
    # 'command', 'retrieve_output', and 'retrieve_output_path' are filled by
    # _DockerDecoratedOperator.
    image: str,
    api_version: str | None = None,
    container_name: str | None = None,
    cpus: float = 1.0,
    docker_url: str | None = None,
    environment: dict[str, str] | None = None,
    private_environment: dict[str, str] | None = None,
    env_file: str | None = None,
    force_pull: bool = False,
    mem_limit: float | str | None = None,
    host_tmp_dir: str | None = None,
    network_mode: str | None = None,
    tls_ca_cert: str | None = None,
    tls_client_cert: str | None = None,
    tls_client_key: str | None = None,
    tls_verify: bool = True,
    tls_hostname: str | bool | None = None,
    tls_ssl_version: str | None = None,
    mount_tmp_dir: bool = True,
    tmp_dir: str = "/tmp/airflow",
    user: str | int | None = None,
    mounts: list[Mount] | None = None,
    entrypoint: str | list[str] | None = None,
    working_dir: str | None = None,
    xcom_all: bool = False,
    docker_conn_id: str | None = None,
    dns: list[str] | None = None,
    dns_search: list[str] | None = None,
    auto_remove: Literal["never", "success", "force"] = "never",
    shm_size: int | None = None,
    tty: bool = False,
    hostname: str | None = None,
    privileged: bool = False,
    cap_add: str | None = None,
    extra_hosts: dict[str, str] | None = None,
    timeout: int = 60,
    device_requests: list[dict] | None = None,
    log_opts_max_size: str | None = None,
    log_opts_max_file: str | None = None,
```

(continues on next page)

(continued from previous page)

```

ipc_mode: str | None = None,
skip_on_exit_code: int | Container[int] | None = None,
port_bindings: dict | None = None,
ulimits: list[dict] | None = None,
labels: dict[str, str] | list[str] | None = None,
**kwargs,
) -> TaskDecorator:
    """Create a decorator to convert the decorated callable to a Docker task.

:param multiple_outputs: If set, function return value will be unrolled to
multiple XCom values.
    Dict will unroll to XCom values with keys as XCom keys. Defaults to False.
:param python_command: Python command for executing functions, Default: python3
:param serializer: Which serializer use to serialize the args and result. It can
be one of the following:
    - ``"pickle"``: (default) Use pickle for serialization. Included in the
Python Standard Library.
    - ``"cloudpickle"``: Use cloudpickle for serialize more complex types,
this requires to include cloudpickle in your requirements.
    - ``"dill"``: Use dill for serialize more complex types,
this requires to include dill in your requirements.
:param use_dill: Deprecated, use ``serializer`` instead. Whether to use dill to
serialize
    the args and result (pickle is default). This allows more complex types
but requires you to include dill in your requirements.
:param image: Docker image from which to create the container.
    If image tag is omitted, "latest" will be used.
:param api_version: Remote API version. Set to ``auto`` to automatically
detect the server's version.
:param container_name: Name of the container. Optional (templated)
:param cpus: Number of CPUs to assign to the container.
    This value gets multiplied with 1024. See
    https://docs.docker.com/engine/reference/run/#cpu-share-constraint
:param docker_url: URL of the host running the docker daemon.
    Default is the value of the ``DOCKER_HOST`` environment variable or unix://
var/run/docker.sock
    if it is unset.
:param environment: Environment variables to set in the container. (templated)
:param private_environment: Private environment variables to set in the
container.
    These are not templated, and hidden from the website.
:param env_file: Relative path to the ``.env`` file with environment variables to
set in the container.
    Overridden by variables in the environment parameter.
:param force_pull: Pull the docker image on every run. Default is False.
:param mem_limit: Maximum amount of memory the container can use.
    Either a float value, which represents the limit in bytes,
or a string like ``128m`` or ``1g``.
:param host_tmp_dir: Specify the location of the temporary directory on the host,
which will
    be mapped to tmp_dir. If not provided defaults to using the standard system

```

(continues on next page)

(continued from previous page)

```

→ temp directory.
    :param network_mode: Network mode for the container. It can be one of the
→ following:
        - ``"bridge"``: Create new network stack for the container with default
→ docker bridge network
        - ``"none"``: No networking for this container
        - ``"container:<name/id>"``: Use the network stack of another container
→ specified via <name/id>
        - ``"host"``: Use the host network stack. Incompatible with `port_bindings`
        - ``"<network-name>/<network-id>"``: Connects the container to user created
→ network
            (using ``docker network create`` command)
    :param tls_ca_cert: Path to a PEM-encoded certificate authority
        to secure the docker connection.
    :param tls_client_cert: Path to the PEM-encoded certificate
        used to authenticate docker client.
    :param tls_client_key: Path to the PEM-encoded key used to authenticate docker
→ client.
    :param tls_verify: Set ``True`` to verify the validity of the provided
→ certificate.
    :param tls_hostname: Hostname to match against
        the docker server certificate or False to disable the check.
    :param tls_ssl_version: Version of SSL to use when communicating with docker
→ daemon.
    :param mount_tmp_dir: Specify whether the temporary directory should be bind-
→ mounted
        from the host to the container. Defaults to True
    :param tmp_dir: Mount point inside the container to
        a temporary directory created on the host by the operator.
        The path is also made available via the environment variable
        ``AIRFLOW_TMP_DIR`` inside the container.
    :param user: Default user inside the docker container.
    :param mounts: List of mounts to mount into the container, e.g.
        ``['/host/path:/container/path', '/host/path2:/container/path2:ro']``.
    :param entrypoint: Overwrite the default ENTRYPPOINT of the image
    :param working_dir: Working directory to
        set on the container (equivalent to the -w switch the docker client)
    :param xcom_all: Push all the stdout or just the last line.
        The default is False (last line).
    :param docker_conn_id: The :ref:`Docker connection id <howto/connection:docker>`  

    :param dns: Docker custom DNS servers
    :param dns_search: Docker custom DNS search domain
    :param auto_remove: Enable removal of the container when the container's process
→ exits. Possible values:
        - ``never``: (default) do not remove container
        - ``success``: remove on success
        - ``force``: always remove container
    :param shm_size: Size of ``/dev/shm`` in bytes. The size must be
        greater than 0. If omitted uses system default.
    :param tty: Allocate pseudo-TTY to the container

```

(continues on next page)

(continued from previous page)

This needs to be set see logs of the Docker container.

`:param hostname: Optional hostname for the container.`

`:param privileged: Give extended privileges to this container.`

`:param cap_add: Include container capabilities`

`:param extra_hosts: Additional hostnames to resolve inside the container, as a mapping of hostname to IP address.`

`:param device_requests: Expose host resources such as GPUs to the container.`

`:param log_opts_max_size: The maximum size of the log before it is rolled.`

A positive integer plus a modifier representing the unit of measure (k, m, or g).

Eg: 10m or 1g Defaults to -1 (unlimited).

`:param log_opts_max_file: The maximum number of log files that can be present.`

If rolling the logs creates excess files, the oldest file is removed.

Only effective when max-size is also set. A positive integer. Defaults to 1.

`:param ipc_mode: Set the IPC mode for the container.`

`:param skip_on_exit_code: If task exits with this exit code, leave the task in ``skipped`` state (default: None). If set to ``None``, any non-zero exit code will be treated as a failure.`

`:param port_bindings: Publish a container's port(s) to the host. It is a dictionary of value where the key indicates the port to open inside the container`

and value indicates the host port that binds to the container port.

Incompatible with ``host`` in ``network_mode``.

`:param ulimits: List of ulimit options to set for the container. Each item should be a :py:class:`docker.types.Ulimit` instance.`

`:param labels: A dictionary of name-value labels (e.g. `{"label1": "value1", "label2": "value2"}`)`

or a list of names of labels to set with empty values (e.g. `["label1", "label2"]`)

"""

The signature should allow only keyword-only arguments, including one named `multiple_outputs` that's automatically provided by default. All other arguments should be copied directly from the real `FooOperator`, and we recommend adding a comment to explain what arguments are filled automatically by `FooDecoratedOperator` and thus not included.

If the new decorator can be used without arguments (e.g. `@task.python` instead of `@task.python()`), You should also add an overload that takes a single callable immediately after the “real” definition so mypy can recognize the function as a “bare decorator”:

```
/build/src/repos/airflow/task-sdk/src/airflow/sdk/definitions/decorators/_init__.pyi
```

```
@overload
def python(self, python_callable: Callable[FParams, FReturn]) -> Task[FParams, FReturn]: ...
```

Once the change is merged and the next Airflow (minor or patch) release comes out, users will be able to see your decorator in IDE auto-complete. This auto-complete will change based on the version of the provider that the user has installed.

Please note that this step is not required to create a working decorator, but does create a better experience for users of the provider.

3.5.16 Export dynamic environment variables available for operators to use

The key value pairs returned in `get_airflow_context_vars` defined in `airflow_local_settings.py` are injected to default Airflow context environment variables, which are available as environment variables when running tasks. Note, both key and value are must be string.

`dag_id`, `task_id`, `execution_date`, `dag_run_id`, `dag_owner`, `dag_email` are reserved keys.

For example, in your `airflow_local_settings.py` file:

```
def get_airflow_context_vars(context) -> dict[str, str]:
    """
    :param context: The context for the task_instance of interest.
    """
    # more env vars
    return {"airflow_cluster": "main"}
```

See *Configuring local settings* for details on how to configure local settings.

3.5.17 Managing Connections

See also

For an overview of hooks and connections, see *Connections & Hooks*.

Airflow's `Connection` object is used for storing credentials and other information necessary for connecting to external services.

Connections may be defined in the following ways:

- in *environment variables*
- in an external *Secrets Backend*
- in the *Airflow metadata database* (using the *CLI* or *web UI*)

Storing connections in environment variables

Airflow connections may be defined in environment variables.

The naming convention is `AIRFLOW_CONN_{CONN_ID}`, all uppercase (note the single underscores surrounding CONN). So if your connection id is `my_prod_db` then the variable name should be `AIRFLOW_CONN_MY_PROD_DB`.

The value can be either JSON or Airflow's URI format.

JSON format example

Added in version 2.3.0.

If serializing with JSON:

```
export AIRFLOW_CONN_MY_PROD_DATABASE='{
    "conn_type": "my-conn-type",
    "login": "my-login",
    "password": "my-password",
    "host": "my-host",
```

(continues on next page)

(continued from previous page)

```

"port": 1234,
"schema": "my-schema",
"extra": {
    "param1": "val1",
    "param2": "val2"
}
}

```

Generating a JSON connection representation

Added in version 2.8.0.

To make connection JSON generation easier, the `Connection` class has a convenience property `as_json()`. It can be used like so:

```

>>> from airflow.models.connection import Connection
>>> c = Connection(
...     conn_id="some_conn",
...     conn_type="mysql",
...     description="connection description",
...     host="myhost.com",
...     login="myname",
...     password="mypassword",
...     extra={"this_param": "some val", "that_param": "other val*"},
... )
>>> print(f"AIRFLOW_CONN_{c.conn_id.upper()}={c.as_json()}")
AIRFLOW_CONN_SOME_CONN={"conn_type": "mysql", "description": "connection description",
    "host": "myhost.com", "login": "myname", "password": "mypassword", "extra": {"this_
    param": "some val", "that_param": "other val*"}}

```

In addition, same approach could be used to convert Connection from URI format to JSON format

```

>>> from airflow.models.connection import Connection
>>> c = Connection(
...     conn_id="awesome_conn",
...     description="Example Connection",
...     uri="aws://AKIAIOSFODNN7EXAMPLE:wJalrXUtnFEMI%2Fk7MDENG%2FbPxRfiCYEXAMPLEKEY@/?_
...     extra__region_name%23A+%22eu-central-1%22%2C+%22config_kwarg%22%3A+%7B
...     %22retries%22%3A+%7B%22mode%22%3A+%22standard%22%2C+%22max_attempts%22%3A+10%7D%7D%7D
...     ",
... )
>>> print(f"AIRFLOW_CONN_{c.conn_id.upper()}={c.as_json()}")
AIRFLOW_CONN_AWESOME_CONN={"conn_type": "aws", "description": "Example Connection",
    "host": "", "login": "AKIAIOSFODNN7EXAMPLE", "password": "wJalrXUtnFEMI/K7MDENG/
    bPxRfiCYEXAMPLEKEY", "schema": "", "extra": {"region_name": "eu-central-1", "config_
    kwargs": {"retries": {"mode": "standard", "max_attempts": 10}}}}

```

URI format example

If serializing with Airflow URI:

```

export AIRFLOW_CONN_MY_PROD_DATABASE='my-conn-type://login:password@host:port/schema?
    param1=val1&param2=val2'

```

See [Connection URI format](#) for more details on how to generate a valid URI.

Note

Connections defined in environment variables will not show up in the Airflow UI or using `airflow connections list`.

Storing connections in a Secrets Backend

You can store Airflow connections in external secrets backends like HashiCorp Vault, AWS SSM Parameter Store, and other such services. For more details see [Secrets Backend](#).

Storing connections in the database

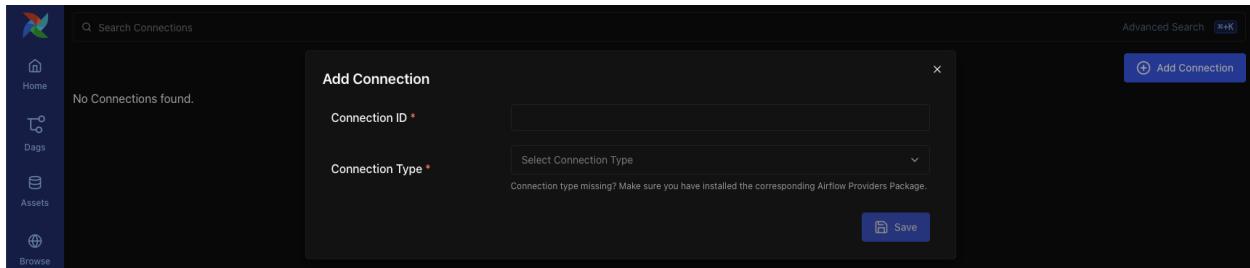
See also

Connections can alternatively be stored in *environment variables* or an *external secrets backend* such as HashiCorp Vault, AWS SSM Parameter Store, etc.

When storing connections in the database, you may manage them using either the web UI or the Airflow CLI.

Creating a Connection with the UI

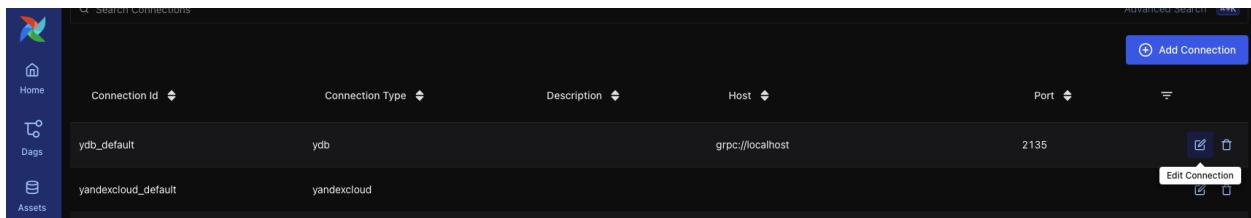
Open the Admin->Connections section of the UI. Click the Add Connection link to create a new connection.



1. Fill in the Connection Id field with the desired connection ID. It is recommended that you use lower-case characters and separate words with underscores.
2. Choose the connection type with the Connection Type field.
3. Fill in the remaining fields. See [Handling of arbitrary dict in extra](#) for a description of the fields belonging to the different connection types.
4. Click the Save button to create the connection.

Editing a Connection with the UI

Open the Admin->Connections section of the UI. Click the pencil icon next to the connection you wish to edit in the connection list.



Modify the connection properties and click the Save button to save your changes.

Creating a Connection from the CLI

You may add a connection to the database from the CLI.

You can add a connection using JSON format (from version 2.3.0):

```
airflow connections add 'my_prod_db' \
--conn-json '{
    "conn_type": "my-conn-type",
    "login": "my-login",
    "password": "my-password",
    "host": "my-host",
    "port": 1234,
    "schema": "my-schema",
    "extra": {
        "param1": "val1",
        "param2": "val2"
    }
}'
```

Alternatively you may use Airflow's Connection URI format (see *Generating a Connection URI*).

```
airflow connections add 'my_prod_db' \
--conn-uri '<conn-type>://<login>:<password>@<host>:<port>/<schema>?param1=val1&
˓→param2=val2&...'
```

Lastly, you may also specify each parameter individually:

```
airflow connections add 'my_prod_db' \
--conn-type 'my-conn-type' \
--conn-login 'login' \
--conn-password 'password' \
--conn-host 'host' \
--conn-port 'port' \
--conn-schema 'schema' \
...
```

Exporting connections to file

You can export to file connections stored in the database (e.g. for migrating connections from one environment to another). See *Exporting Connections* for usage.

Security of connections in the database

For connections stored in the Airflow metadata database, Airflow uses [Fernet](#) to encrypt password and other potentially sensitive data. It guarantees that without the encryption password, Connection Passwords cannot be manipulated or read without the key. For information on configuring Fernet, look at [Fernet](#).

Testing Connections

For security reasons, the test connection functionality is disabled by default across Airflow UI, API and CLI.

For more information on capabilities of users, see the documentation: https://airflow.apache.org/docs/apache-airflow/stable/security/security_model.html#capabilities-of-authenticated-ui-users. It is strongly advised to not enable the feature until you make sure that only highly trusted UI/API users have “edit connection” permissions.

The availability of the functionality can be controlled by the `test_connection` flag in the core section of the Airflow configuration (`airflow.cfg`). It can also be controlled by the environment variable `AIRFLOW__CORE__TEST_CONNECTION`.

The following values are accepted for this config param:

- Disabled: Disables the test connection functionality and disables the Test Connection button in the UI. This is also the default value set in the Airflow configuration.
- Enabled: Enables the test connection functionality and activates the Test Connection button in the UI.
- Hidden: Disables the test connection functionality and hides the Test Connection button in UI.

After enabling Test Connection, it can be used from the `create` or `edit` connection page in the UI, through calling `Connections REST API`, or running the `airflow connections test` CLI command.

⚠ Warning

This feature won't be available for the connections residing in external secrets backends when using the Airflow UI or REST API.

To test a connection, Airflow calls the `test_connection` method from the associated hook class and reports the results. It may happen that the connection type does not have any associated hook or the hook doesn't have the `test_connection` method implementation, in either case an error message will be displayed or functionality will be disabled (if you are testing in the UI).

ℹ Note

When testing in the Airflow UI, the test executes from the webserver so this feature is subject to network egress rules setup for your webserver.

ℹ Note

If webserver & worker machines (if testing via the Airflow UI) or machines/pods (if testing via the Airflow CLI) have different libs or providers installed, test results *might* differ.

Custom connection types

Airflow allows the definition of custom connection types – including modifications of the add/edit form for the connections. Custom connection types are defined in community maintained providers, but you can also add a custom

provider that adds custom connection types. See apache-airflow-providers:index for description on how to add custom providers.

The custom connection types are defined via Hooks delivered by the providers. The Hooks can implement methods defined in the protocol class `DiscoverableHook`. Note that your custom Hook should not derive from this class, this class is an example to document expectations regarding about class fields and methods that your Hook might define. Another good example is `JdbcHook`.

By implementing those methods in your hooks and exposing them via `connection-types` array (and deprecated `hook-class-names`) in the provider meta-data, you can customize Airflow by:

- Adding custom connection types
- Adding automated Hook creation from the connection type
- Adding custom form widget to display and edit custom “extra” parameters in your connection URL
- Hiding fields that are not used for your connection
- Adding placeholders showing examples of how fields should be formatted

You can read more about details how to add custom providers in the apache-airflow-providers:index

Custom connection fields

It is possible to add custom form fields in the connection add / edit views in the Airflow webserver. Custom fields are stored in the `Connection.extra` field as JSON. To add a custom field, implement method `get_connection_form_widgets()`. This method should return a dictionary. The keys should be the string name of the field as it should be stored in the `extra` dict. The values should be inheritors of `wtforms.fields.core.Field`.

Here's an example:

```
@staticmethod
def get_connection_form_widgets() -> dict[str, Any]:
    """Returns connection widgets to add to connection form"""
    from flask_appbuilder.fieldwidgets import BS3TextFieldWidget
    from flask_babel import lazy_gettext
    from wtforms import StringField

    return {
        "workspace": StringField(lazy_gettext("Workspace"), widget=BS3TextFieldWidget()),
        "project": StringField(lazy_gettext("Project"), widget=BS3TextFieldWidget()),
    }
```

Note

Custom fields no longer need the `extra__<conn_type>__` prefix

Prior to Airflow 2.3, if you wanted a custom field in the UI, you had to prefix it with `extra__<conn_type>__`, and this is how its value would be stored in the `extra` dict. From 2.3 onward, you no longer need to do this.

Method `get_ui_field_behaviour()` lets you customize behavior of both . For example you can hide or relabel a field (e.g. if it's unused or re-purposed) and you can add placeholder text.

An example:

```
@staticmethod
def get_ui_field_behaviour() -> dict[str, Any]:
```

(continues on next page)

(continued from previous page)

```
"""Returns custom field behaviour"""
return {
    "hidden_fields": ["port", "host", "login", "schema"],
    "relabeling": {},
    "placeholders": {
        "password": "Asana personal access token",
        "workspace": "My workspace gid",
        "project": "My project gid",
    },
}
```

Note

If you want to add a form placeholder for an `extra` field whose name conflicts with a standard connection attribute (i.e. login, password, host, scheme, port, extra) then you must prefix it with `extra_<conn_type>_`. E.g. `extra_myself_password`.

Take a look at providers for examples of what you can do, for example `JdbcHook` and `AsanaHook` both make use of this feature.

Note

Deprecated hook-class-names

Prior to Airflow 2.2.0, the connections in providers have been exposed via `hook-class-names` array in provider's meta-data. However, this has proven to be inefficient when using individual hooks in workers, and the `hook-class-names` array is now replaced by the `connection-types` array. Until provider supports Airflow below 2.2.0, both `connection-types` and `hook-class-names` should be present. Automated checks during CI build will verify consistency of those two arrays.

URI format**Note**

From version 2.3.0 you can serialize connections with JSON instead. See *example*.

For historical reasons, Airflow has a special URI format that can be used for serializing a Connection object to a string value.

In general, Airflow's URI format looks like the following:

```
my-conn-type://my-login:my-password@my-host:5432/my-schema?param1=val1&param2=val2
```

The above URI would produce a Connection object equivalent to the following:

```
Connection(
    conn_id="",
    conn_type="my_conn_type",
    description=None,
    login="my-login",
```

(continues on next page)

(continued from previous page)

```
password="my-password",
host="my-host",
port=5432,
schema="my-schema",
extra=json.dumps(dict(param1="val1", param2="val2")),
)
```

Generating a connection URI

To make connection URI generation easier, the `Connection` class has a convenience method `get_uri()`. It can be used like so:

```
>>> import json
>>> from airflow.sdk import Connection
>>> c = Connection(
...     conn_id="some_conn",
...     conn_type="mysql",
...     description="connection description",
...     host="myhost.com",
...     login="myname",
...     password="mypassword",
...     extra=json.dumps(dict(this_param="some val", that_param="other val*")),
... )
>>> print(f"AIRFLOW_CONN_{c.conn_id.upper()}={c.get_uri()}")
AIRFLOW_CONN_SOME_CONN=mysql://myname:mypassword@myhost.com?this_param=some+val&that_
param=other+val%2A'
```

Note

The `get_uri()` method return the connection URI in Airflow format, **not** a SQLAlchemy-compatible URI. if you need a SQLAlchemy-compatible URI for database connections, use `sqlalchemy_url` property instead.

Additionally, if you have created a connection, you can use `airflow connections get` command.

```
$ airflow connections get sqlite_default
Id: 40
Connection Id: sqlite_default
Connection Type: sqlite
Host: /tmp/sqlite_default.db
Schema: null
Login: null
Password: null
Port: null
Is Encrypted: false
Is Extra Encrypted: false
Extra: {}
URI: sqlite:///%2Ftmp%2Fsqlite_default.db
```

Handling of arbitrary dict in extra

Some JSON structures cannot be urlencoded without loss. For such JSON, `get_uri` will store the entire string under the url query param `__extra__`.

For example:

```
>>> extra_dict = {"my_val": ["list", "of", "values"], "extra": {"nested": {"json": "val"}}}
```

```
>>> c = Connection(
...     conn_type="scheme",
...     host="host/location",
...     schema="schema",
...     login="user",
...     password="password",
...     port=1234,
...     extra=json.dumps(extra_dict),
... )
>>> uri = c.get_uri()
>>> uri
'scheme://user:password@host%2Flocation:1234/schema?__extra__=%7B%22my_val%22%3A+%5B
%22list%22%2C+%22of%22%2C+%22values%22%5D%2C+%22extra%22%3A+%7B%22nested%22%3A+%7B
%22json%22%3A+%22val%22%7D%7D%7D'
```

And we can verify that it returns the same dictionary:

```
>>> new_c = Connection(uri=uri)
>>> new_c.extra_dejson == extra_dict
True
```

But for the most common case of storing only key-value pairs, plain url encoding is used.

You can verify a URI is parsed correctly like so:

```
>>> from airflow.sdk import Connection

>>> c = Connection(uri="my-conn-type://my-login:my-password@my-host:5432/my-schema?
param1=val1&param2=val2")
>>> print(c.login)
my-login
>>> print(c.password)
my-password
```

Handling of special characters in connection params

Note

Use the convenience method `Connection.get_uri` when generating a connection as described in section *Generating a Connection URI*. This section for informational purposes only.

Special handling is required for certain characters when building a URI manually.

For example if your password has a /, this fails:

```
>>> c = Connection(uri="my-conn-type://my-login:my-pa/ssword@my-host:5432/my-schema?  
↳ param1=val1&param2=val2")  
ValueError: invalid literal for int() with base 10: 'my-pa'
```

To fix this, you can encode with `quote_plus()`:

```
>>> c = Connection(uri="my-conn-type://my-login:my-pa%2Fssword@my-host:5432/my-schema?  
↳ param1=val1&param2=val2")  
>>> print(c.password)  
my-pa/ssword
```

3.5.18 Managing 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.

| Key | Description | Is Encrypted |
|--------------------|-----------------|--------------|
| snowflake_password | *** | true |
| postgres_env | prod | true |
| pipedrive_env | pipedrive | true |
| environment | prod | true |
| airtable_base_key | appzasdasdasdas | true |
| airtable_api_key | *** | true |

See the *Variables Concepts* documentation for more information.

Storing Variables in Environment Variables

Added in version 1.10.10.

Airflow Variables can also be created and managed using Environment Variables. The environment variable naming convention is `AIRFLOW_VAR_{VARIABLE_NAME}`, all uppercase. So if your variable key is `foo` then the variable name should be `AIRFLOW_VAR_FOO`.

For example,

```
export AIRFLOW_VAR_FOO=BAR  
  
# To use JSON, store them as JSON strings  
export AIRFLOW_VAR_FOO_BAZ='{"hello": "world"}'
```

You can use them in your dags as:

```
from airflow.sdk import Variable  
  
foo = Variable.get("foo")  
foo_json = Variable.get("foo_baz", deserialize_json=True)
```

Note

Single underscores surround VAR. This is in contrast with the way `airflow.cfg` parameters are stored, where double underscores surround the config section name. Variables set using Environment Variables would not appear in the Airflow UI but you will be able to use them in your DAG file. Variables set using Environment Variables will also take precedence over variables defined in the Airflow UI.

Securing Variables

Airflow uses [Fernet](#) to encrypt variables stored in the metastore database. It guarantees that without the encryption password, content cannot be manipulated or read without the key. For information on configuring Fernet, look at [Fernet](#).

In addition to retrieving variables from environment variables or the metastore database, you can enable a secrets backend to retrieve variables. For more details see [Secrets Backend](#).

3.5.19 Setup and Teardown

In data workflows it's common to create a resource (such as a compute resource), use it to do some work, and then tear it down. Airflow provides setup and teardown tasks to support this need.

Key features of setup and teardown tasks:

- If you clear a task, its setups and teardowns will be cleared.
- By default, teardown tasks are ignored for the purpose of evaluating dag run state.
- A teardown task will run if its setup was successful, even if its work tasks failed. But it will skip if the setup was skipped.
- Teardown tasks are ignored when setting dependencies against task groups.
- Teardown will also be carried out if the DAG run is manually set to “failed” or “success” to ensure resources will be cleaned-up.

How setup and teardown works

Basic usage

Suppose you have a dag that creates a cluster, runs a query, and deletes the cluster. Without using setup and teardown tasks you might set these relationships:

```
create_cluster >> run_query >> delete_cluster
```

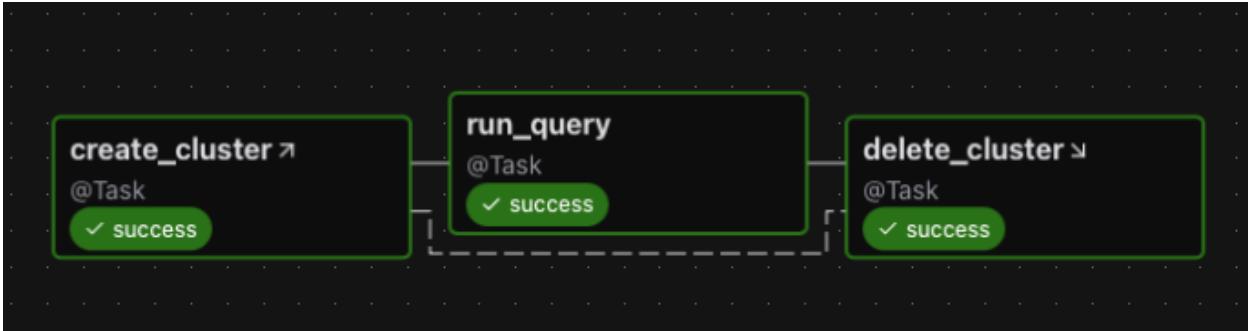
To enable `create_cluster` and `delete_cluster` as setup and teardown tasks, we mark them as such methods `as_setup` and `as_teardown` and add an upstream / downstream relationship between them:

```
create_cluster.as_setup() >> run_query >> delete_cluster.as_teardown()
create_cluster >> delete_cluster
```

For convenience we can do this in one line by passing `create_cluster` to the `as_teardown` method:

```
create_cluster >> run_query >> delete_cluster.as_teardown(setups=create_cluster)
```

Here's the graph for this dag:



Observations:

- If you clear `run_query` to run it again, then both `create_cluster` and `delete_cluster` will be cleared.
- If `run_query` fails, then `delete_cluster` will still run.
- The success of the dag run will depend *only* on the success of `run_query`.

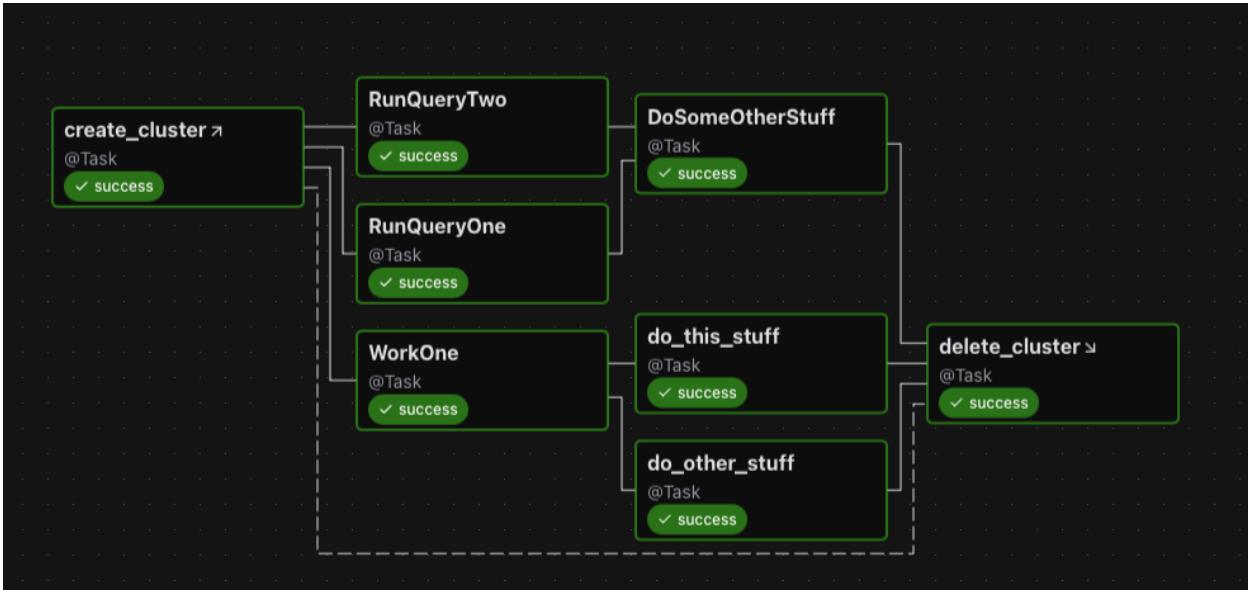
Additionally, if we have multiple tasks to wrap, we can use the teardown as a context manager:

```

with delete_cluster().as_teardown(setups=create_cluster()):
    [RunQueryOne(), RunQueryTwo()] >> DoSomeOtherStuff()
    WorkOne() >> [do_this_stuff(), do_other_stuff()]
  
```

This will set `create_cluster` to run before the tasks in the context, and `delete_cluster` after them.

Here it is, shown in the graph:



Note that if you are attempting to add an already-instantiated task to a setup context you need to do it explicitly:

```

with my_teardown_task as scope:
    scope.add_task(work_task) # work_task was already instantiated elsewhere
  
```

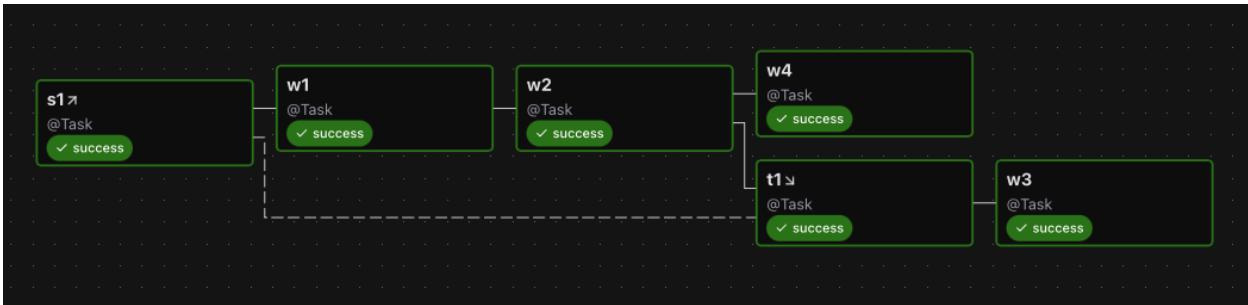
Setup “scope”

Tasks between a setup and its teardown are in the “scope” of the setup / teardown pair.

Let’s look at an example:

```
s1 >> w1 >> w2 >> t1.as_teardown(setups=s1) >> w3
w2 >> w4
```

And the graph:



In the above example, `w1` and `w2` are “between” `s1` and `t1` and therefore are assumed to require `s1`. Thus if `w1` or `w2` is cleared, so too will be `s1` and `t1`. But if `w3` or `w4` is cleared, neither `s1` nor `t1` will be cleared.

You can have multiple setup tasks wired to a single teardown. The teardown will run if at least one of the setups completed successfully.

You can have a setup without a teardown:

```
create_cluster >> run_query >> other_task
```

In this case, everything downstream of `create_cluster` is assumed to require it. So if you clear `other_task`, it will also clear `create_cluster`. Suppose that we add a teardown for `create_cluster` after `run_query`:

```
create_cluster >> run_query >> other_task
run_query >> delete_cluster.as_teardown(setups=create_cluster)
```

Now, Airflow can infer that `other_task` does not require `create_cluster`, so if we clear `other_task`, `create_cluster` will not also be cleared.

In that example, we (in our pretend docs land) actually wanted to delete the cluster. But supposing we did not, and we just wanted to say “`other_task` does not require `create_cluster`”, then we could use an `EmptyOperator` to limit the setup’s scope:

```
create_cluster >> run_query >> other_task
run_query >> EmptyOperator(task_id="cluster_teardown").as_teardown(setups=create_cluster)
```

Implicit ALL_SUCCESS constraint

Any task in the scope of a setup has an implicit “`all_success`” constraint on its setups. This is necessary to ensure that if a task with indirect setups is cleared, it will wait for them to complete. If a setup fails or is skipped, the work tasks which depend on them will be marked as failures or skips. We also require that any non-teardown directly downstream of a setup must have trigger rule `ALL_SUCCESS`.

Controlling dag run state

Another feature of setup / teardown tasks is you can choose whether or not the teardown task should have an impact on dag run state. Perhaps you don't care if the "cleanup" work performed by your teardown task fails, and you only consider the dag run a failure if the "work" tasks fail. By default, teardown tasks are not considered for dag run state.

Continuing with the example above, if you want the run's success to depend on `delete_cluster`, then set `on_failure_fail_dagrun=True` when setting `delete_cluster` as teardown. For example:

```
create_cluster >> run_query >> delete_cluster.as_teardown(setups=create_cluster, on_
failure_fail_dagrun=True)
```

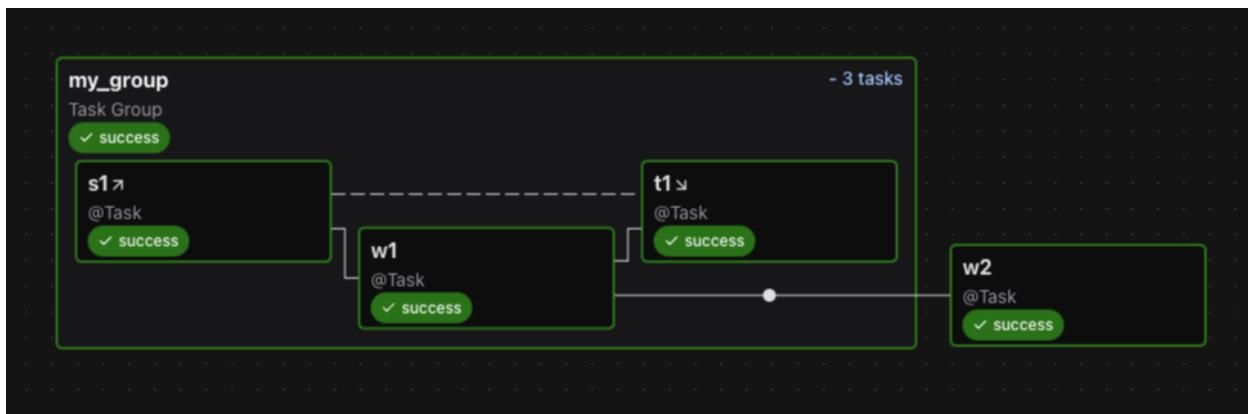
Authoring with task groups

When adding dependency from task group to task group, or from task group to *task*, we ignore teardowns. This allows teardowns to run in parallel, and allows dag execution to proceed even if teardown tasks fail.

Consider this example:

```
with TaskGroup("my_group") as tg:
    s1 = s1()
    w1 = w1()
    t1 = t1()
    s1 >> w1 >> t1.as_teardown(setups=s1)
    w2 = w2()
    tg >> w2
```

Graph:



If `t1` were not a teardown task, then this dag would effectively be `s1 >> w1 >> t1 >> w2`. But since we have marked `t1` as a teardown, it's ignored in `tg >> w2`. So the dag is equivalent to the following:

```
s1 >> w1 >> [t1.as_teardown(setups=s1), w2]
```

Now let's consider an example with nesting:

```
with TaskGroup("my_group") as tg:
    s1 = s1()
    w1 = w1()
    t1 = t1()
    s1 >> w1 >> t1.as_teardown(setups=s1)
    w2 = w2()
```

(continues on next page)

(continued from previous page)

```
tg >> w2
dag_s1 = dag_s1()
dag_t1 = dag_t1()
dag_s1 >> [tg, w2] >> dag_t1.as_teardown(setups=dag_s1)
```

Graph:



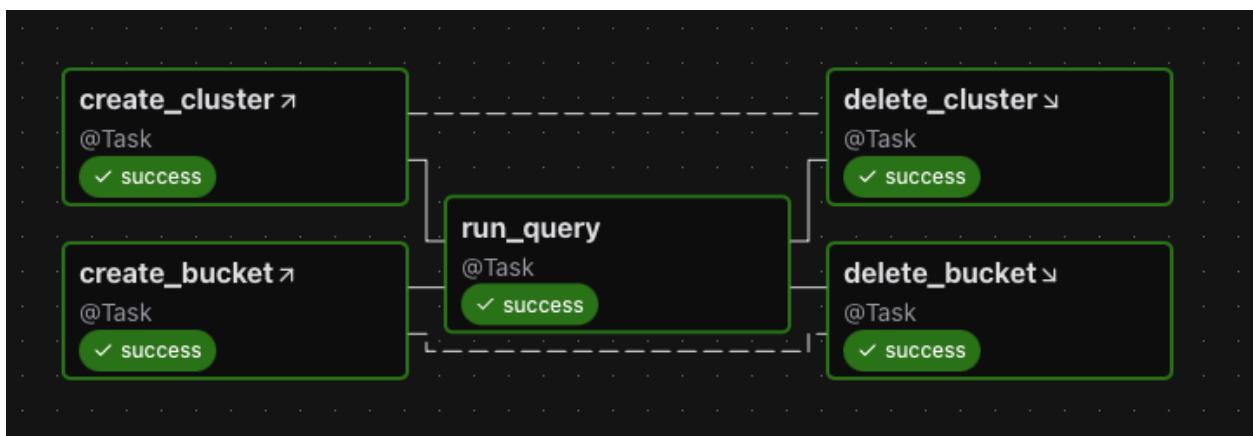
In this example `s1` is downstream of `dag_s1`, so it must wait for `dag_s1` to complete successfully. But `t1` and `dag_t1` can run concurrently, because `t1` is ignored in the expression `tg >> dag_t1`. If you clear `w2`, it will clear `dag_s1` and `dag_t1`, but not anything in the task group.

Running setups and teardowns in parallel

You can run setup tasks in parallel:

```
(  
    [create_cluster, create_bucket]  
    >> run_query  
    >> [delete_cluster.as_teardown(setups=create_cluster), delete_bucket.as_  
    ↪teardown(setups=create_bucket)]  
)
```

Graph:



It can be nice visually to put them in a group:

```
with TaskGroup("setup") as tg_s:  
    create_cluster = create_cluster()
```

(continues on next page)

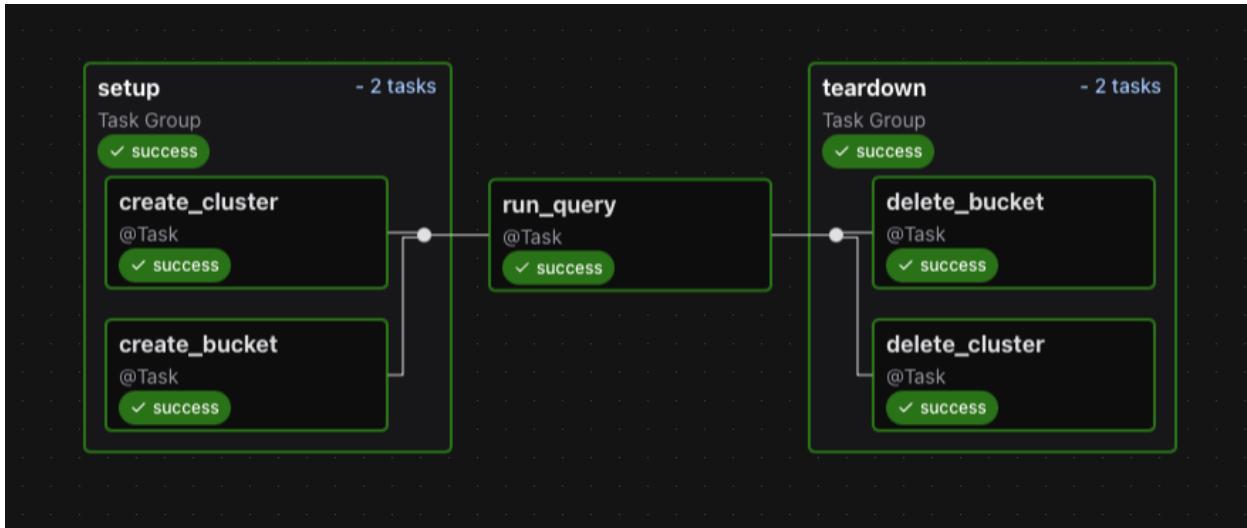
(continued from previous page)

```

create_bucket = create_bucket()
run_query = run_query()
with TaskGroup("teardown") as tg_t:
    delete_cluster = delete_cluster().as_teardown(setups=create_cluster)
    delete_bucket = delete_bucket().as_teardown(setups=create_bucket)
tg_s >> run_query >> tg_t

```

And the graph:



Trigger rule behavior for teardowns

Tear downs use a (non-configurable) trigger rule called ALL_DONE_SETUP_SUCCESS. With this rule, as long as all upstreams are done and at least one directly connected setup is successful, the teardown will run. If all of a teardown's setups were skipped or failed, those states will propagate to the teardown.

Side-effect on manual DAG state changes

As teardown tasks are often used to clean-up resources they need to run also if the DAG is manually terminated. For the purpose of early termination a user can manually mark the DAG run as “success” or “failed” which kills all tasks before completion. If the DAG contains teardown tasks, they will still be executed. Therefore as a side effect allowing teardown tasks to be scheduled, a DAG will not be immediately set to a terminal state if the user requests so.

3.5.20 Running Airflow behind a reverse proxy

Airflow can be set up behind a reverse proxy, with the ability to set its endpoint with great flexibility.

For example, you can configure your reverse proxy to get:

```
https://lab.mycompany.com/myorg/airflow/
```

To do so, you need to set the following setting in your `airflow.cfg`:

```
base_url = http://my_host/myorg/airflow
```

- Configure your reverse proxy (e.g. nginx) to pass the url and http header as it for the Airflow webserver, without any rewrite, for example:

```

server {
    listen 80;
    server_name lab.mycompany.com;

    location /myorg/airflow/ {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $http_host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;
        proxy_redirect off;
        proxy_http_version 1.1;
    }
}

```

- Some parts of the UI are rendered inside iframes (Auth managers security links for instance), you need to make sure that you are not setting a restricted CSP for iframe rendering such as `frame-ancestors 'none'`. You can set the CSP header in your reverse proxy configuration, for example:

```
add_header Content-Security-Policy "frame-ancestors 'self';";
```

- Use `--proxy-headers` CLI flag to tell Uvicorn to respect these headers: `airflow api-server --proxy-headers`
- If your proxy server is not on the same host (or in the same docker container) as Airflow, then you will need to set the `FORWARDED_ALLOW_IPS` environment variable so Uvicorn knows who to trust this header from. See [Uvicorn's docs](#). For the full options you can pass here. (Please note the `--forwarded-allow-ips` CLI option does not exist in Airflow.)

3.5.21 Running Airflow with systemd

Airflow can integrate with systemd based systems. This makes watching your daemons easy as `systemd` can take care of restarting a daemon on failures.

In the `scripts/systemd` directory, you can find unit files that have been tested on Redhat based systems. These files can be used as-is by copying them over to `/usr/lib/systemd/system`.

The following **assumptions** have been made while creating these unit files:

1. Airflow runs as the following `user:group airflow:airflow`.
2. Airflow runs on a Redhat based system.

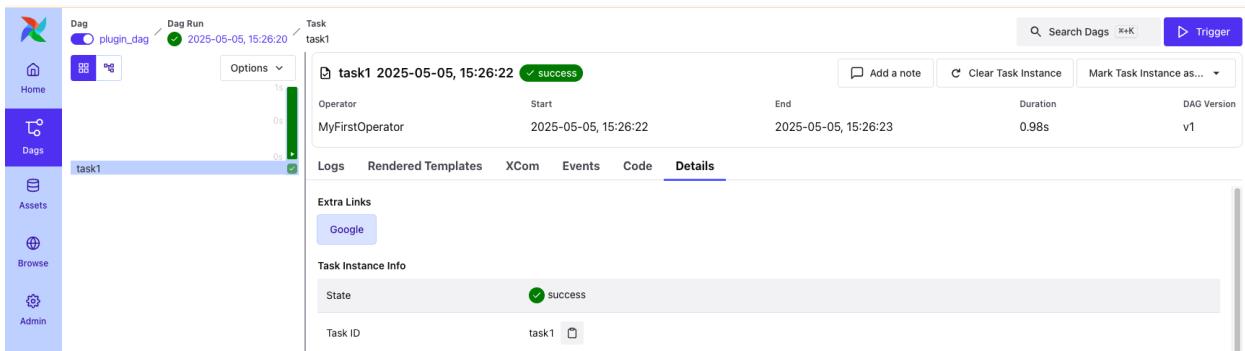
If this is not the case, appropriate changes will need to be made.

Please **note** that environment configuration is picked up from `/etc/sysconfig/airflow`.

An example file is supplied within `scripts/systemd`. You can also define configuration at `AIRFLOW_HOME` or `AIRFLOW_CONFIG`.

3.5.22 Define an operator extra link

If you want to add extra links to operators you can define them via a plugin or provider package. Extra links will be displayed in task details page in Grid view.



The following code shows how to add extra links to an operator via Plugins:

```
from airflow.sdk import BaseOperator
from airflow.sdk import BaseOperatorLink
from airflow.models.taskinstancekey import TaskInstanceKey
from airflow.plugins_manager import AirflowPlugin

class GoogleLink(BaseOperatorLink):
    name = "Google"

    def get_link(self, operator: BaseOperator, *, ti_key: TaskInstanceKey):
        return "https://www.google.com"

class MyFirstOperator(BaseOperator):
    operator_extra_links = (GoogleLink(),)

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def execute(self, context):
        self.log.info("Hello World!")

# Defining the plugin class
class AirflowExtraLinkPlugin(AirflowPlugin):
    name = "extra_link_plugin"
    operator_extra_links = [
        GoogleLink(),
    ]
```

The extra links defined via custom Airflow Provider or Airflow operators will be pushed as an xcom to the XCom table in metadata DB during task execution. During display in the grid view, this xcom is retrieved and displayed.

You can also add a global operator extra link that will be available to all the operators through an Airflow plugin or through Airflow providers. You can learn more about it in the *plugin interface* and in apache-airflow-providers:index.

You can see all the extra links available via community-managed providers in apache-airflow-providers:core-

extensions/extralinks.

Add or override Links to Existing Operators

You can also add (or override) an extra link to an existing operators through an Airflow plugin or custom provider.

For example, the following Airflow plugin will add an Operator Link on all tasks using GCSToS3Operator operator.

Adding Operator Links to Existing Operators `plugins/extralink.py`:

```
from airflow.sdk import BaseOperator, BaseOperatorLink
from airflow.models.taskinstancekey import TaskInstanceKey
from airflow.plugins_manager import AirflowPlugin
from airflow.providers.amazon.aws.transfers.gcs_to_s3 import GCSToS3Operator

class S3LogLink(BaseOperatorLink):
    name = "S3"

    # Add list of all the operators to which you want to add this extra link
    # Example: operators = [GCSToS3operator, GCSToBigQueryOperator]
    operators = [GCSToS3operator]

    def get_link(self, operator: BaseOperator, *, ti_key: TaskInstanceKey):
        # Invalid bucket name because upper case letters and underscores are used
        # This will not be a valid bucket in any region
        bucket_name = "Invalid_Bucket_Name"
        return "https://s3.amazonaws.com/airflow-logs/{bucket_name}/{dag_id}/{task_id}/
→{run_id}".format(
            bucket_name=bucket_name,
            dag_id=operator.dag_id,
            task_id=operator.task_id,
            run_id=ti_key.run_id,
        )

# Defining the plugin class
class AirflowExtraLinkPlugin(AirflowPlugin):
    name = "extra_link_plugin"
    operator_extra_links = [
        S3LogLink(),
    ]
```

Overriding Operator Links of Existing Operators:

It is also possible to replace a built-in link on an operator via a Plugin. For example `BigQueryExecuteQueryOperator` includes a link to the Google Cloud Console, but if we wanted to change that link we could do:

```
from airflow.sdk import BaseOperator, BaseOperatorLink
from airflow.models.taskinstancekey import TaskInstanceKey
from airflow.plugins_manager import AirflowPlugin
from airflow.providers.google.cloud.operators.bigquery import BigQueryOperator

# Change from https to http just to display the override
```

(continues on next page)

(continued from previous page)

```
BIGQUERY_JOB_DETAILS_LINK_FMT = "http://console.cloud.google.com/bigquery?j={job_id}"
```

```
class BigQueryDatasetLink(BaseGoogleLink):
    """
    Helper class for constructing BigQuery Dataset Link.
    """

    name = "BigQuery Dataset"
    key = "bigquery_dataset"
    format_str = BIGQUERY_DATASET_LINK

    @staticmethod
    def persist(
        context: Context,
        task_instance: BaseOperator,
        dataset_id: str,
        project_id: str,
    ):
        task_instance.xcom_push(
            context,
            key=BigQueryDatasetLink.key,
            value={"dataset_id": dataset_id, "project_id": project_id},
        )

# Defining the plugin class
class AirflowExtraLinkPlugin(AirflowPlugin):
    name = "extra_link_plugin"
    operator_extra_links = [
        BigQueryDatasetLink(),
    ]
```

Adding Operator Links via Providers

As explained in apache-airflow-providers:index, when you create your own Airflow Provider, you can specify the list of operators that provide extra link capability. This happens by including the operator class name in the provider-info information stored in your Provider's package meta-data:

Example meta-data required in your provider-info dictionary (this is part of the meta-data returned by apache-airflow-providers-google provider currently):

```
extra-links:
  - airflow.providers.google.cloud.links.bigquery.BigQueryDatasetLink
  - airflow.providers.google.cloud.links.bigquery.BigQueryTableLink
```

You can include as many operators with extra links as you want.

3.5.23 Email Configuration

You can configure the email that is being sent in your `airflow.cfg` by setting a `subject_template` and/or a `html_content_template` in the `[email]` section.

[email]

```
email_backend = airflow.utils.email.send_email_smtp
subject_template = /path/to/my_subject_template_file
html_content_template = /path/to/my_html_content_template_file
```

Equivalent environment variables look like:

```
AIRFLOW__EMAIL__EMAIL_BACKEND=airflow.utils.email.send_email_smtp
AIRFLOW__EMAIL__SUBJECT_TEMPLATE=/path/to/my_subject_template_file
AIRFLOW__EMAIL__HTML_CONTENT_TEMPLATE=/path/to/my_html_content_template_file
```

You can configure a sender's email address by setting `from_email` in the `[email]` section like:

[email]

```
from_email = "John Doe <johndoe@example.com>"
```

Equivalent environment variables look like:

```
AIRFLOW__EMAIL__FROM_EMAIL="John Doe <johndoe@example.com>"
```

To configure SMTP settings, checkout the `SMTP` section in the standard configuration. If you do not want to store the SMTP credentials in the config or in the environment variables, you can create a connection called `smtp_default` of `Email` type, or choose a custom connection name and set the `email_conn_id` with its name in the configuration & store SMTP username-password in it. Other SMTP settings like host, port etc always gets picked up from the configuration only. The connection can be of any type (for example ‘HTTP connection’).

If you want to check which email backend is currently set, you can use `airflow config get-value email email_backend` command as in the example below.

```
$ airflow config get-value email email_backend
airflow.utils.email.send_email_smtp
```

To access the task's information you use [Jinja Templating](#) in your template files.

For example a `html_content_template` file could look like this:

```
Try {{try_number}} out of {{max_tries + 1}}<br>
Exception:<br>{{exception_html}}<br>
Log: <a href="{{ti.log_url}}>Link</a><br>
Host: {{ti.hostname}}<br>
Mark success: <a href="{{ti.mark_success_url}}>Link</a><br>
```

Note

For more information on setting the configuration, see [Setting Configuration Options](#)

Send email using SendGrid

Using Default SMTP

You can use the default Airflow SMTP backend to send email with SendGrid

```
[smtp]
smtp_host=smtp.sendgrid.net
smtp_starttls=False
smtp_ssl=False
smtp_port=587
smtp_mail_from=<your-from-email>
```

Equivalent environment variables looks like

```
AIRFLOW__SMTP__SMTP_HOST=smtp.sendgrid.net
AIRFLOW__SMTP__SMTP_STARTTLS=False
AIRFLOW__SMTP__SMTP_SSL=False
AIRFLOW__SMTP__SMTP_PORT=587
AIRFLOW__SMTP__SMTP_MAIL_FROM=<your-from-email>
```

Using SendGrid Provider

Airflow can be configured to send e-mail using [SendGrid](#).

Follow the steps below to enable it:

1. Setup your SendGrid account, The SMTP and copy username and API Key.
2. Include `sendgrid` provider as part of your Airflow installation, e.g.,

```
pip install 'apache-airflow[sendgrid]' --constraint ...
```

or

```
pip install 'apache-airflow-providers-sendgrid' --constraint ...
```

3. Update `email_backend` property in `[email]` section in `airflow.cfg`, i.e.

```
[email]
email_backend = airflow.providers.sendgrid.utils.emailer.send_email
email_conn_id = sendgrid_default
from_email = "hello@eg.com"
```

Equivalent environment variables looks like

```
AIRFLOW__EMAIL__EMAIL_BACKEND=airflow.providers.sendgrid.utils.emailer.send_email
AIRFLOW__EMAIL__EMAIL_CONN_ID=sendgrid_default
SENDGRID_MAIL_FROM=hello@thelearning.dev
```

4. Create a connection called `sendgrid_default`, or choose a custom connection name and set it in `email_conn_id` of 'Email' type. Only login and password are used from the connection.

Add Connection

Connection ID * sendgrid_default

Connection Type * email
Connection type missing? Make sure you have installed the corresponding Airflow Providers Package.

Standard Fields

Description

Host

Login

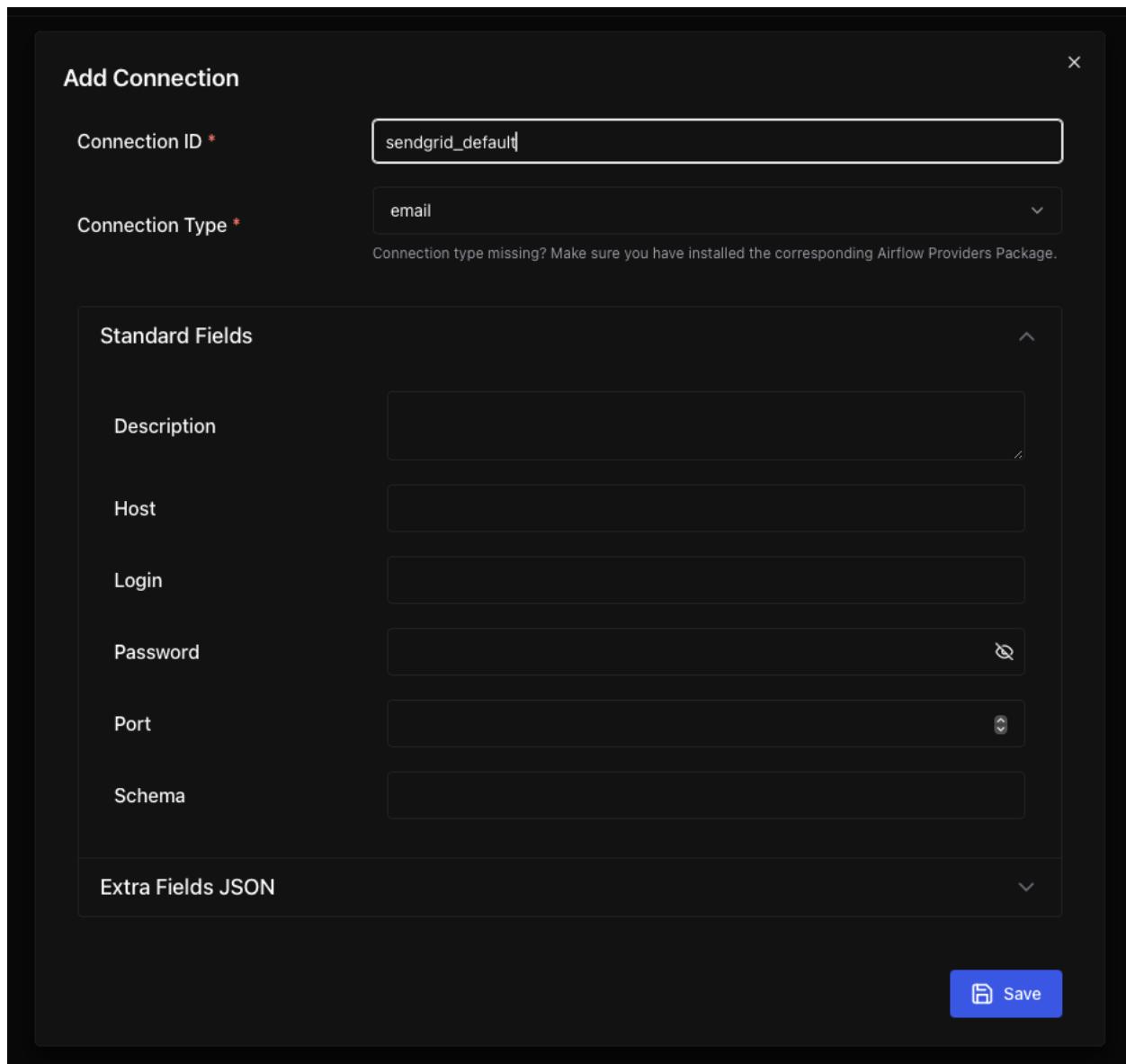
Password

Port

Schema

Extra Fields JSON

 Save



 Note

The callbacks for success, failure and retry will use the same configuration to send the email

Send email using AWS SES

Airflow can be configured to send e-mail using [AWS SES](#).

Follow the steps below to enable it:

1. Include `amazon` subpackage as part of your Airflow installation:

```
pip install 'apache-airflow[amazon]'
```

2. Update `email_backend` property in `[email]` section in `airflow.cfg`:

[email]

```
email_backend = airflow.providers.amazon.aws.utils.emailer.send_email
email_conn_id = aws_default
from_email = From email <email@example.com>
```

Note that for SES, you must configure from_email to the valid email that can send messages from SES.

3. Create a connection called aws_default, or choose a custom connection name and set it in email_conn_id. The type of connection should be Amazon Web Services.

3.5.24 Dynamic DAG Generation

This document describes creation of dags that have a structure generated dynamically, but where the number of tasks in the DAG does not change between DAG Runs. If you want to implement a DAG where number of Tasks (or Task Groups as of Airflow 2.6) can change based on the output/result of previous tasks, see *Dynamic Task Mapping*.

 Note

Consistent sequence of generating tasks and task groups

In all cases where you generate dags dynamically, you should make sure that Tasks and Task Groups are generated with consistent sequence every time the DAG is generated, otherwise you might end up with Tasks and Task Groups changing their sequence in the Grid View every time you refresh the page. This can be achieved for example by using a stable sorting mechanism in your Database queries or by using sorted() function in Python.

Dynamic dags with environment variables

If you want to use variables to configure your code, you should always use [environment variables](#) in your top-level code rather than [Airflow Variables](#). Using Airflow Variables in top-level code creates a connection to the metadata DB of Airflow to fetch the value, which can slow down parsing and place extra load on the DB. See the [best practices on Airflow Variables](#) to make the best use of Airflow Variables in your dags using Jinja templates.

For example you could set DEPLOYMENT variable differently for your production and development environments. The variable DEPLOYMENT could be set to PROD in your production environment and to DEV in your development environment. Then you could build your DAG differently in production and development environment, depending on the value of the environment variable.

```
deployment = os.environ.get("DEPLOYMENT", "PROD")
if deployment == "PROD":
    task = Operator(param="prod-param")
elif deployment == "DEV":
    task = Operator(param="dev-param")
```

Generating Python code with embedded meta-data

You can externally generate Python code containing the meta-data as importable constants. Such constant can then be imported directly by your DAG and used to construct the object and build the dependencies. This makes it easy to import such code from multiple dags without the need to find, load and parse the meta-data stored in the constant - this is done automatically by Python interpreter when it processes the “import” statement. This sounds strange at first, but it is surprisingly easy to generate such code and make sure this is a valid Python code that you can import from your dags.

For example assume you dynamically generate (in your DAG folder), the my_company_utils/common.py file:

```
# This file is generated automatically !
ALL_TASKS = ["task1", "task2", "task3"]
```

Then you can import and use the ALL_TASKS constant in all your dags like that:

```
from my_company_utils.common import ALL_TASKS

with DAG(
    dag_id="my_dag",
    schedule=None,
    start_date=datetime(2021, 1, 1),
    catchup=False,
):
    for task in ALL_TASKS:
        # create your operators and relations here
    ...
```

Don't forget that in this case you need to add empty `__init__.py` file in the `my_company_utils` folder and you should add the `my_company_utils/*` line to `.airflowignore` file (using the default glob syntax), so that the whole folder is ignored by the scheduler when it looks for dags.

Dynamic dags with external configuration from a structured data file

If you need to use a more complex meta-data to prepare your DAG structure and you would prefer to keep the data in a structured non-python format, you should export the data to the DAG folder in a file and push it to the DAG folder, rather than try to pull the data by the DAG's top-level code - for the reasons explained in the parent *Top level Python Code*.

The meta-data should be exported and stored together with the dags in a convenient file format (JSON, YAML formats are good candidates) in DAG folder. Ideally, the meta-data should be published in the same package/folder as the module of the DAG file you load it from, because then you can find location of the meta-data file in your DAG easily. The location of the file to read can be found using the `__file__` attribute of the module containing the DAG:

```
my_dir = os.path.dirname(os.path.abspath(__file__))
configuration_file_path = os.path.join(my_dir, "config.yaml")
with open(configuration_file_path) as yaml_file:
    configuration = yaml.safe_load(yaml_file)
# Configuration dict is available here
```

Registering dynamic dags

You can dynamically generate dags when using the `@dag` decorator or the `with DAG(..)` context manager and Airflow will automatically register them.

```
from datetime import datetime
from airflow.sdk import dag, task

configs = {
    "config1": {"message": "first DAG will receive this message"},
    "config2": {"message": "second DAG will receive this message"},
}

for config_name, config in configs.items():
    dag_id = f"dynamic_generated_dag_{config_name}"
```

(continues on next page)

(continued from previous page)

```

@dag(dag_id=dag_id, start_date=datetime(2022, 2, 1))
def dynamic_generated_dag():
    @task
    def print_message(message):
        print(message)

    print_message(config["message"])

dynamic_generated_dag()

```

The code below will generate a DAG for each config: `dynamic_generated_dag_config1` and `dynamic_generated_dag_config2`. Each of them can run separately with related configuration.

If you do not wish to have dags auto-registered, you can disable the behavior by setting `auto_register=False` on your DAG.

Changed in version 2.4: As of version 2.4 dags that are created by calling a `@dag` decorated function (or that are used in the `with DAG(...)` context manager) are automatically registered, and no longer need to be stored in a global variable.

Optimizing DAG parsing delays during execution

Added in version 2.4.

This is an *experimental feature*.

Sometimes when you generate a lot of Dynamic dags from a single DAG file, it might cause unnecessary delays when the DAG file is parsed during task execution. The impact is a delay before a task starts.

Why is this happening? You might not be aware but just before your task is executed, Airflow parses the Python file the DAG comes from.

The Airflow Scheduler (or rather DAG File Processor) requires loading of a complete DAG file to process all metadata. However, task execution requires only a single DAG object to execute a task. Knowing this, we can skip the generation of unnecessary DAG objects when a task is executed, shortening the parsing time. This optimization is most effective when the number of generated dags is high.

There is an experimental approach that you can take to optimize this behaviour. Note that it is not always possible to use (for example when generation of subsequent dags depends on the previous dags) or when there are some side-effects of your dags generation. Also the code snippet below is pretty complex and while we tested it and it works in most circumstances, there might be cases where detection of the currently parsed DAG will fail and it will revert to creating all the dags or fail. Use this solution with care and test it thoroughly.

A nice example of performance improvements you can gain is shown in the [Airflow's Magic Loop](#) blog post that describes how parsing during task execution was reduced from 120 seconds to 200 ms. (The example was written before Airflow 2.4 so it uses undocumented behaviour of Airflow.)

In Airflow 2.4 instead you can use `get_parsing_context()` method to retrieve the current context in documented and predictable way.

Upon iterating over the collection of things to generate dags for, you can use the context to determine whether you need to generate all DAG objects (when parsing in the DAG File processor), or to generate only a single DAG object (when executing the task).

The `get_parsing_context()` return the current parsing context. The context is of `AirflowParsingContext` and in case only single DAG/task is needed, it contains `dag_id` and `task_id` fields set. In case “full” parsing is needed (for example in DAG File Processor), `dag_id` and `task_id` of the context are set to `None`.

```

from airflow.sdk import DAG
from airflow.sdk import get_parsing_context

current_dag_id = get_parsing_context().dag_id

for thing in list_of_things:
    dag_id = f"generated_dag_{thing}"
    if current_dag_id is not None and current_dag_id != dag_id:
        continue # skip generation of non-selected DAG

    with DAG(dag_id=dag_id, ...):
        ...

```

3.5.25 Running Airflow in Docker

This quick-start guide will allow you to quickly get Airflow up and running with the CeleryExecutor in Docker.

Caution

This procedure can be useful for learning and exploration. However, adapting it for use in real-world situations can be complicated and the docker compose file does not provide any security guarantees required for production system. Making changes to this procedure will require specialized expertise in Docker & Docker Compose, and the Airflow community may not be able to help you.

For that reason, we recommend using Kubernetes with the Official Airflow Community Helm Chart when you are ready to run Airflow in production.

Before you begin

This procedure assumes familiarity with Docker and Docker Compose. If you haven't worked with these tools before, you should take a moment to run through the [Docker Quick Start](#) (especially the section on [Docker Compose](#)) so you are familiar with how they work.

Follow these steps to install the necessary tools, if you have not already done so.

1. Install [Docker Community Edition \(CE\)](#) on your workstation. Depending on your OS, you may need to configure Docker to use at least 4.00 GB of memory for the Airflow containers to run properly. Please refer to the Resources section in the [Docker for Windows](#) or [Docker for Mac](#) documentation for more information.
2. Install [Docker Compose](#) v2.14.0 or newer on your workstation.

Older versions of `docker-compose` do not support all the features required by the Airflow `docker-compose.yaml` file, so double check that your version meets the minimum version requirements.

Tip

The default amount of memory available for Docker on macOS is often not enough to get Airflow up and running. If enough memory is not allocated, it might lead to the webserver continuously restarting. You should allocate at least 4GB memory for the Docker Engine (ideally 8GB).

You can check if you have enough memory by running this command:

```
docker run --rm "debian:bookworm-slim" bash -c 'numfmt --to iec $(echo $((($getconf _PHYS_PAGES) * $(getconf PAGE_SIZE))))'
```

⚠ Warning

Some operating systems (Fedora, ArchLinux, RHEL, Rocky) have recently introduced Kernel changes that result in Airflow in Docker Compose consuming 100% memory when run inside the community Docker implementation maintained by the OS teams.

This is an issue with backwards-incompatible containerd configuration that some of Airflow dependencies have problems with and is tracked in a few issues:

- [Moby issue](#)
- [Containerd issue](#)

There is no solution yet from the containerd team, but seems that installing [Docker Desktop on Linux](#) solves the problem as stated in [This comment](#) and allows to run Breeze with no problems.

Fetching docker-compose.yaml

To deploy Airflow on Docker Compose, you should fetch `docker-compose.yaml`.

```
curl -Lf0 'https://airflow.apache.org/docs/apache-airflow/3.1.0/docker-compose.yaml'
```

❗ Important

From July 2023 Compose V1 stopped receiving updates. We strongly advise upgrading to a newer version of Docker Compose, supplied `docker-compose.yaml` may not function accurately within Compose V1.

This file contains several service definitions:

- `airflow-scheduler` - The *scheduler* monitors all tasks and dags, then triggers the task instances once their dependencies are complete.
- `airflow-dag-processor` - The DAG processor parses DAG files.
- `airflow-api-server` - The api server is available at `http://localhost:8080`.
- `airflow-worker` - The worker that executes the tasks given by the scheduler.
- `airflow-triggerer` - The triggerer runs an event loop for deferrable tasks.
- `airflow-init` - The initialization service.
- `postgres` - The database.
- `redis` - The [redis](#) broker that forwards messages from scheduler to worker.

Optionally, you can enable flower by adding `--profile flower` option, e.g. `docker compose --profile flower up`, or by explicitly specifying it on the command line e.g. `docker compose up flower`.

- `flower` - [The flower app](#) for monitoring the environment. It is available at `http://localhost:5555`.

All these services allow you to run Airflow with CeleryExecutor. For more information, see [Architecture Overview](#).

Some directories in the container are mounted, which means that their contents are synchronized between your computer and the container.

- `./dags` - you can put your DAG files here.

- ./logs - contains logs from task execution and scheduler.
- ./config - you can add custom log parser or add `airflow_local_settings.py` to configure cluster policy.
- ./plugins - you can put your *custom plugins* here.

This file uses the latest Airflow image (`apache/airflow`). If you need to install a new Python library or system library, you can build your image.

Initializing Environment

Before starting Airflow for the first time, you need to prepare your environment, i.e. create the necessary files, directories and initialize the database.

Setting the right Airflow user

On **Linux**, the quick-start needs to know your host user id and needs to have group id set to `0`. Otherwise the files created in `dags`, `logs`, `config` and `plugins` will be created with `root` user ownership. You have to make sure to configure them for the docker-compose:

```
mkdir -p ./dags ./logs ./plugins ./config
echo -e "AIRFLOW_UID=$(id -u)" > .env
```

See *Docker Compose environment variables*

For other operating systems, you may get a warning that `AIRFLOW_UID` is not set, but you can safely ignore it. You can also manually create an `.env` file in the same folder as `docker-compose.yaml` with this content to get rid of the warning:

```
AIRFLOW_UID=50000
```

Initialize airflow.cfg (Optional)

If you want to initialize `airflow.cfg` with default values before launching the airflow service, run.

```
docker compose run airflow-cli airflow config list
```

This will seed `airflow.cfg` with default values in `config` folder.

Initialize the database

On **all operating systems**, you need to run database migrations and create the first user account. To do this, run.

```
docker compose up airflow-init
```

After initialization is complete, you should see a message like this:

```
airflow-init_1      | Upgrades done
airflow-init_1      | Admin user airflow created
airflow-init_1      | 3.1.0
start_airflow-init_1 exited with code 0
```

The account created has the login `airflow` and the password `airflow`.

Cleaning-up the environment

The docker-compose environment we have prepared is a “quick-start” one. It was not designed to be used in production and it has a number of caveats - one of them being that the best way to recover from any problem is to clean it up and restart from scratch.

The best way to do this is to:

- Run `docker compose down --volumes --remove-orphans` command in the directory you downloaded the `docker-compose.yaml` file
- Remove the entire directory where you downloaded the `docker-compose.yaml` file `rm -rf '<DIRECTORY>'`
- Run through this guide from the very beginning, starting by re-downloading the `docker-compose.yaml` file

Running Airflow

Now you can start all services:

```
docker compose up
```

Note

docker-compose is old syntax. Please check Stackoverflow.

In a second terminal you can check the condition of the containers and make sure that no containers are in an unhealthy condition:

```
$ docker ps
CONTAINER ID   IMAGE          |version-spacepad| COMMAND                  CREATED      ▾
→   STATUS        PORTS          NAMES
247ebe6cf87a   apache/airflow:3.1.0   "/usr/bin/dumb-init ..."   3 minutes ago   Up 3 minutes (healthy)   8080/tcp       compose_airflow-worker_1
ed9b09fc84b1   apache/airflow:3.1.0   "/usr/bin/dumb-init ..."   3 minutes ago   Up 3 minutes (healthy)   8080/tcp       compose_airflow-scheduler_1
7cb1fb603a98   apache/airflow:3.1.0   "/usr/bin/dumb-init ..."   3 minutes ago   Up 3 minutes (healthy)   0.0.0.0:8080->8080/tcp   compose_airflow-api_server_1
74f3bbe506eb   postgres:13           |version-spacepad| "docker-entrypoint.s..."   18 minutes ago   Up 17 minutes (healthy)   5432/tcp       compose_postgres_1
0bd6576d23cb   redis:latest         |version-spacepad| "docker-entrypoint.s..."   10 hours ago   Up 17 minutes (healthy)   0.0.0.0:6379->6379/tcp   compose_redis_1
```

Accessing the environment

After starting Airflow, you can interact with it in 3 ways:

- by running *CLI commands*.
- via a browser using *the web interface*.
- using *the REST API*.

Running the CLI commands

You can also run *CLI commands*, but you have to do it in one of the defined `airflow-*` services. For example, to run `airflow info`, run the following command:

```
docker compose run airflow-worker airflow info
```

If you have Linux or Mac OS, you can make your work easier and download a optional wrapper scripts that will allow you to run commands with a simpler command.

```
curl -Lfo 'https://airflow.apache.org/docs/apache-airflow/3.1.0/airflow.sh'  
chmod +x airflow.sh
```

Now you can run commands easier.

```
./airflow.sh info
```

You can also use `bash` as parameter to enter interactive bash shell in the container or `python` to enter python container.

```
./airflow.sh bash
```

```
./airflow.sh python
```

Accessing the web interface

Once the cluster has started up, you can log in to the web interface and begin experimenting with dags.

The webserver is available at: `http://localhost:8080`. The default account has the login `airflow` and the password `airflow`.

Sending requests to the REST API

Basic username password authentication is currently supported for the REST API, which means you can use common tools to send requests to the API.

The webserver is available at: `http://localhost:8080`. The default account has the login `airflow` and the password `airflow`.

Here is a sample `curl` command, which sends a request to retrieve a pool list:

```
ENDPOINT_URL="http://localhost:8080/"  
curl -X GET \  
--user "airflow:airflow" \  
"${ENDPOINT_URL}/api/v1/pools"
```

Cleaning up

To stop and delete containers, delete volumes with database data and download images, run:

```
docker compose down --volumes --rmi all
```

Using custom images

When you want to run Airflow locally, you might want to use an extended image, containing some additional dependencies - for example you might add new python packages, or upgrade airflow providers to a later version. This can be done very easily by specifying `build: .` in your `docker-compose.yaml` and placing a custom Dockerfile alongside your `docker-compose.yaml`. Then you can use `docker compose build` command to build your image (you need to do it only once). You can also add the `--build` flag to your `docker compose` commands to rebuild the images on-the-fly when you run other `docker compose` commands.

Examples of how you can extend the image with custom providers, python packages, apt packages and more can be found in [Building the image](#).

Note

Creating custom images means that you need to maintain also a level of automation as you need to re-create the images when either the packages you want to install or Airflow is upgraded. Please do not forget about keeping these scripts. Also keep in mind, that in cases when you run pure Python tasks, you can use the [Python Virtualenv functions](#) which will dynamically source and install python dependencies during runtime. With Airflow 2.8.0 Virtualenvs can also be cached.

Special case - adding dependencies via requirements.txt file

Usual case for custom images, is when you want to add a set of requirements to it - usually stored in `requirements.txt` file. For development, you might be tempted to add it dynamically when you are starting the original airflow image, but this has a number of side effects (for example your containers will start much slower - each additional dependency will further delay your containers start up time). Also it is completely unnecessary, because docker compose has the development workflow built-in. You can - following the previous chapter, automatically build and use your custom image when you iterate with docker compose locally. Specifically when you want to add your own requirement file, you should do those steps:

- 1) Comment out the `image: ...` line and remove comment from the `build: .` line in the `docker-compose.yaml` file. The relevant part of the docker-compose file of yours should look similar to (use correct image tag):

```
#image: ${AIRFLOW_IMAGE_NAME:-apache/airflow:|version|}  
build: .
```

- 2) Create Dockerfile in the same folder your `docker-compose.yaml` file is with content similar to:

```
FROM apache/airflow:|version|  
ADD requirements.txt .  
RUN pip install apache-airflow==${AIRFLOW_VERSION} -r requirements.txt
```

It is the best practice to install apache-airflow in the same version as the one that comes from the original image. This way you can be sure that `pip` will not try to downgrade or upgrade apache airflow while installing other requirements, which might happen in case you try to add a dependency that conflicts with the version of apache-airflow that you are using.

- 3) Place `requirements.txt` file in the same directory.

Run `docker compose build` to build the image, or add `--build` flag to `docker compose up` or `docker compose run` commands to build the image automatically as needed.

Special case - Adding a custom config file

If you have a custom config file and wish to use it in your Airflow instance, you need to perform the following steps:

- 1) Replace the auto-generated `airflow.cfg` file in the local config folder with your custom config file.
- 2) If your config file has a different name than `airflow.cfg`, adjust the filename in `AIRFLOW_CONFIG: '/opt/airflow/config/airflow.cfg'`

Networking

In general, if you want to use Airflow locally, your dags may try to connect to servers which are running on the host. In order to achieve that, an extra configuration must be added in `docker-compose.yaml`. For example, on Linux the configuration must be in the section `services: airflow-worker` adding `extra_hosts: - "host.docker.internal:host-gateway"`; and use `host.docker.internal` instead of `localhost`. This configuration vary in different platforms. Please check the Docker documentation for [Windows](#) and [Mac](#) for further information.

Debug Airflow inside docker container using PyCharm

Prerequisites: Create a project in **PyCharm** and download the (`docker-compose.yaml`).

Steps:

- 1) Modify `docker-compose.yaml`

Add the following section under the `services` section:

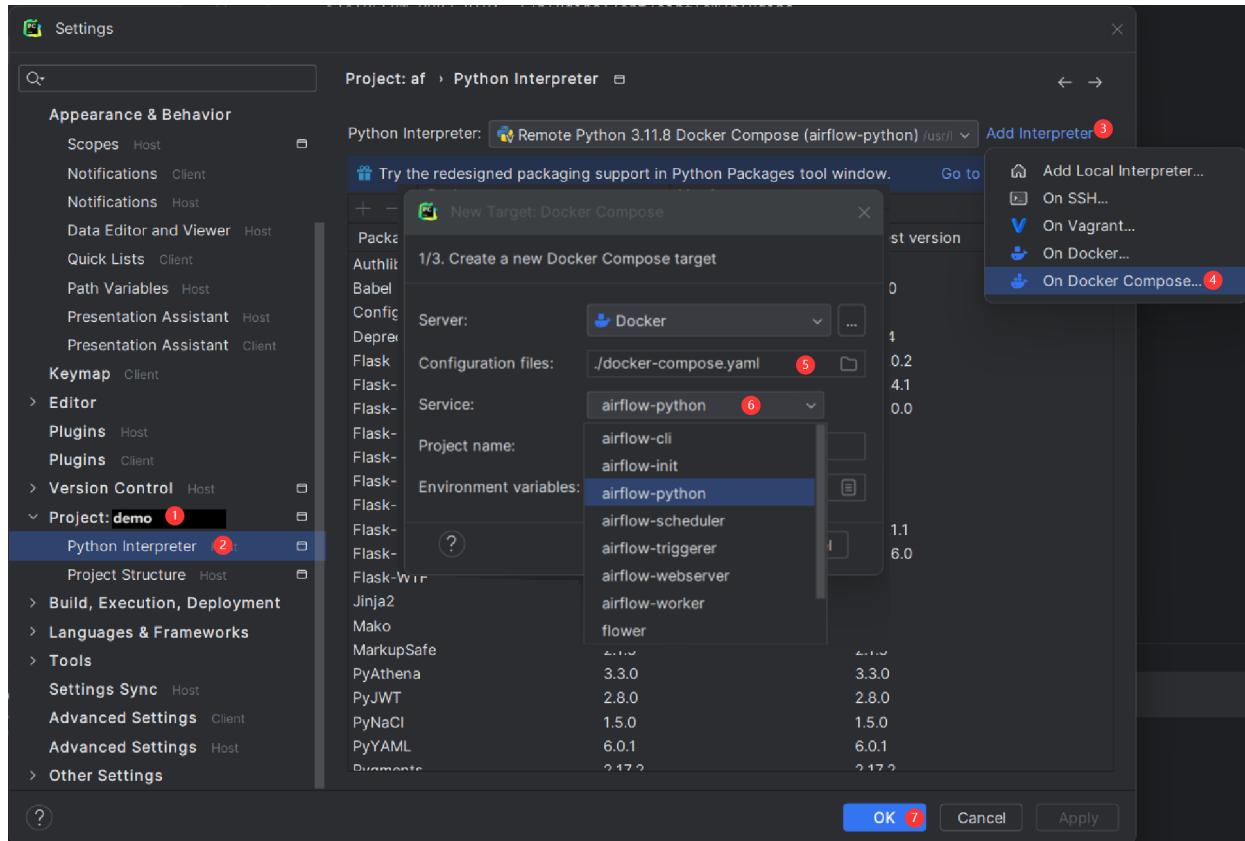
```
airflow-python:
  <<: *airflow-common
  profiles:
    - debug
  environment:
    <<: *airflow-common-env
  user: "50000:0"
  entrypoint: [ "/bin/bash", "-c" ]
```

Note

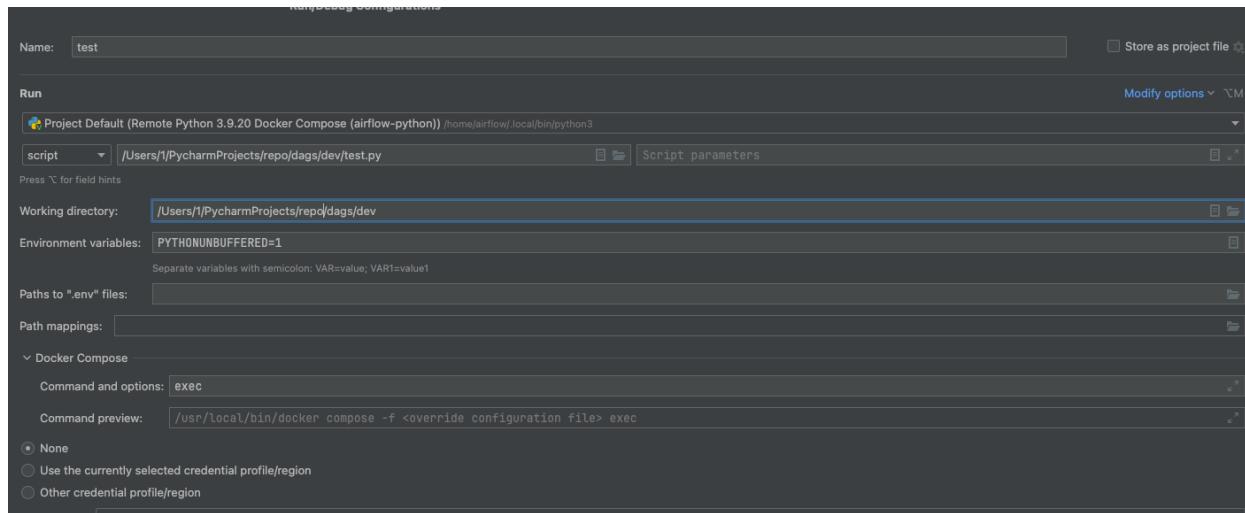
This code snippet creates a new service named “`airflow-python`” specifically for PyCharm’s Python interpreter. On a Linux system, if you have executed the command `echo -e "AIRFLOW_UID=$(id -u)" > .env`, you need to set `user: "50000:0"` in `airflow-python` service to avoid PyCharm’s Unresolved reference ‘`airflow`’ error.

- 2) Configure PyCharm Interpreter

- Open PyCharm and navigate to **Settings > Project: <Your Project Name> > Python Interpreter**.
- Click the “**Add Interpreter**” button and choose “**On Docker Compose**”.
- In the **Configuration file** field, select your `docker-compose.yaml` file.
- In the **Service** field, choose the newly added `airflow-python` service.
- Click “**Next**” and follow the prompts to complete the configuration.



Building the interpreter index might take some time. 3) Add `exec` to docker-compose/command and actions in python service



Once configured, you can debug your Airflow code within the container environment, mimicking your local setup.

FAQ: Frequently asked questions

ModuleNotFoundError: No module named 'XYZ'

The Docker Compose file uses the latest Airflow image (apache/airflow). If you need to install a new Python library or system library, you can customize and extend it.

What's Next?

From this point, you can head to the [Tutorials](#) section for further examples or the [How-to Guides](#) section if you're ready to get your hands dirty.

Environment variables supported by Docker Compose

Do not confuse the variable names here with the build arguments set when image is built. The `AIRFLOW_UID` build arg defaults to `50000` when the image is built, so it is “baked” into the image. On the other hand, the environment variables below can be set when the container is running, using - for example - result of `id -u` command, which allows to use the dynamic host runtime user id which is unknown at the time of building the image.

| Variable | Description | Default |
|--------------------|--|-----------------------------------|
| <code>AIRFL</code> | Airflow Image to use. | <code>apache/airflow:3.1.0</code> |
| <code>AIRFL</code> | UID of the user to run Airflow containers as. Override if you want to use non-default Airflow UID (for example when you map folders from host, it should be set to result of <code>id -u</code> call. When it is changed, a user with the UID is created with <code>default</code> name inside the container and home of the user is set to <code>/airflow/home/</code> in order to share Python libraries installed there. This is in order to achieve the OpenShift compatibility. See more in the Arbitrary Docker User | <code>50000</code> |

Note

Before Airflow 2.2, the Docker Compose also had `AIRFLOW_GID` parameter, but it did not provide any additional functionality - only added confusion - so it has been removed.

Those additional variables are useful in case you are trying out/testing Airflow installation via Docker Compose. They are not intended to be used in production, but they make the environment faster to bootstrap for first time users with the most common customizations.

| Variable | Description | Default |
|---------------------------|--|----------------------|
| <code>_AIRFLOW_WWW</code> | Username for the administrator UI account. If this value is specified, admin UI user gets created automatically. This is only useful when you want to run Airflow for a test-drive and want to start a container with embedded development database. | <code>airflow</code> |
| <code>_AIRFLOW_WWW</code> | Password for the administrator UI account. Only used when <code>_AIRFLOW_WWW_USER_USERNAME</code> set. | <code>airflow</code> |
| <code>_PIP_ADDITI</code> | If not empty, airflow containers will attempt to install requirements specified in the variable. example: <code>lxml==4.6.3 charset-normalizer==1.4.1</code> . Available in Airflow image 2.1.1 and above. | |

3.6 UI Overview

The Airflow UI provides a powerful way to monitor, manage, and troubleshoot your data pipelines and data assets. As of Airflow 3, the UI has been refreshed with a modern look, support for dark and light themes, and a redesigned navigation experience.

This guide offers a reference-style walkthrough of key UI components, with annotated screenshots to help new and experienced users alike.

Note

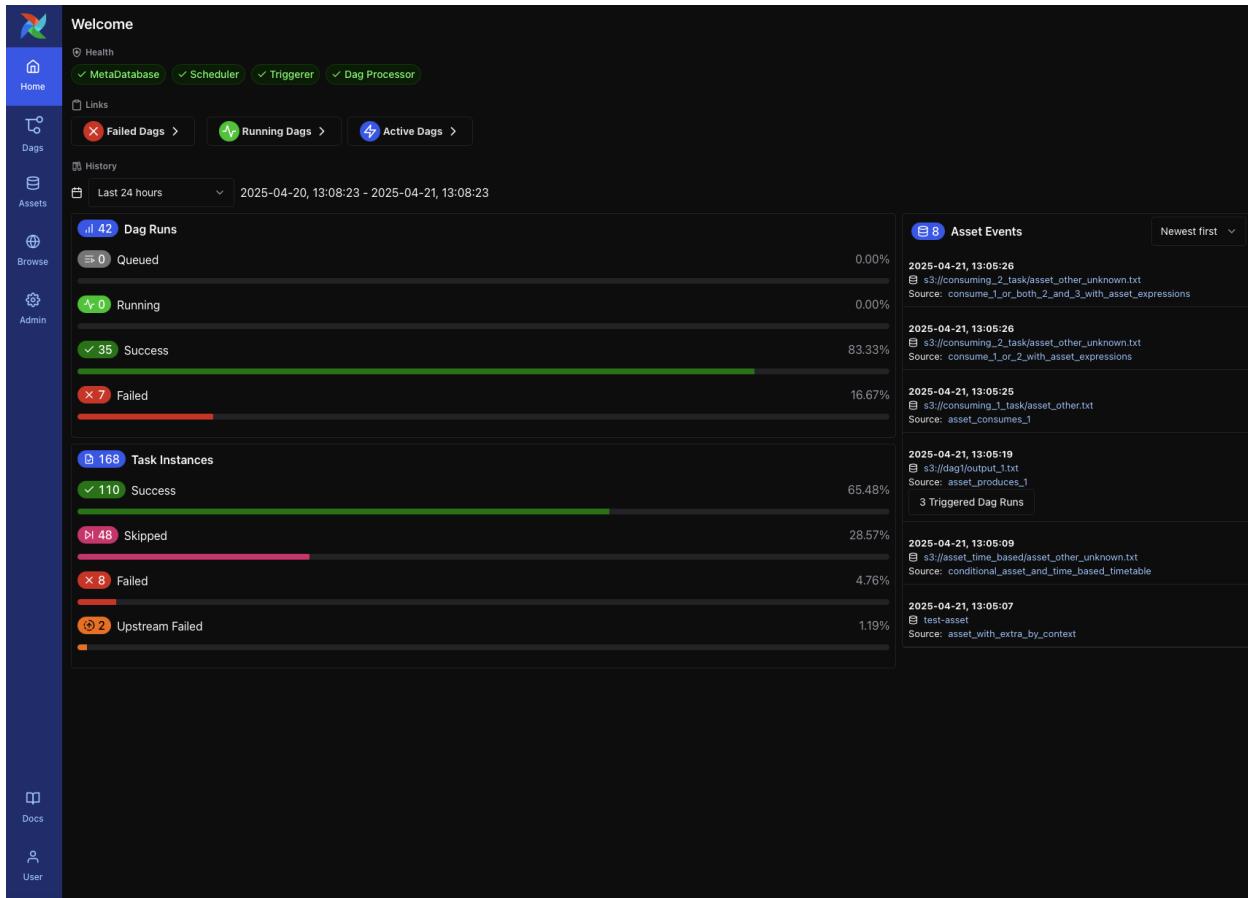
Screenshots in this guide use **dark theme** by default. Select views are also shown in **light theme** for comparison. You can toggle themes from user settings located at the bottom corner of the Airflow UI.

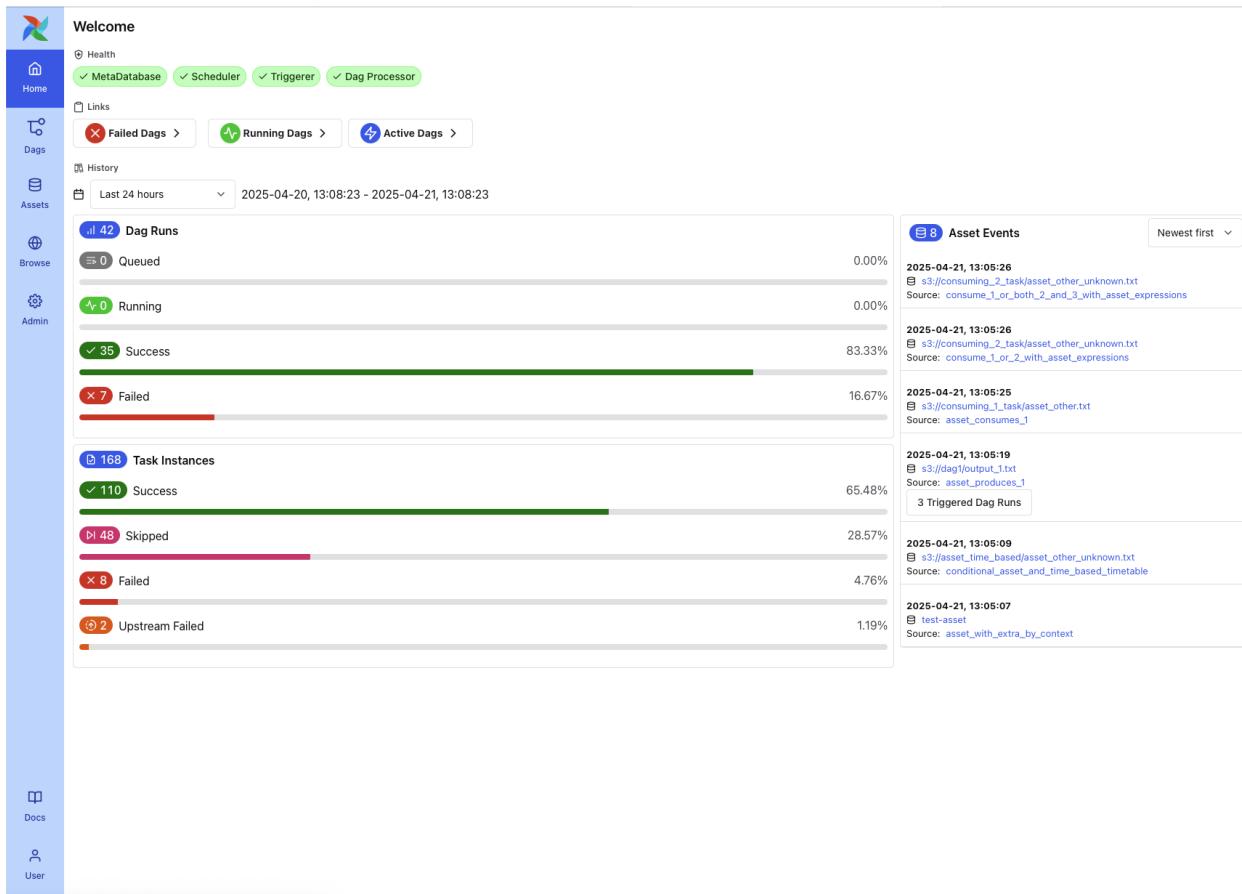
3.6.1 Home Page

The Home Page provides a high-level overview of the system state and recent activity. It is the default landing page in Airflow 3 and includes:

- **Health indicators** for system components such as the MetaDatabase, Scheduler, Triggerer, and Dag Processor
- **Quick links** to DAGs filtered by status (e.g., Failed DAGs, Running DAGs, Active DAGs)
- **DAG and Task Instance history**, showing counts and success/failure rates over a selectable time range
- **Recent asset events**, including materializations and triggered DAGs

This page is useful for quickly assessing the health of your environment and identifying recent issues or asset-triggered events.





3.6.2 DAG List View

The DAG List View appears when you click the **DAGs** tab in the main navigation bar. It displays all DAGs available in your environment, with a clear summary of their status, recent runs, and configuration.

Each row includes:

- **DAG ID**
- **Schedule** and next run time
- **Status of the latest DAG run**
- **Bar chart of recent runs**
- **Tags**, which can be used for grouping or filtering DAGs (e.g., example, produces)
- **Pause/resume toggle**
- Links to access DAG-level views

At the top of the view, you can:

- Use **filters** for DAG status, schedule state, and tags
- Use **search or advanced search (+K)** to find specific DAGs
- Sort the list using the dropdown (e.g., Latest Run Start Date)

The screenshot shows the Apache Airflow web interface with the 'Dags' tab selected. The main area displays a list of 84 dags, each represented by a card. Each card includes the dag's name, its schedule, the latest run status, and the next run scheduled. A green checkmark indicates a successful run, while a red X indicates a failed run. To the right of each card is a vertical timeline bar showing the execution history of the dag. The interface also features a search bar at the top, filters for failed, running, and success states, and a sorting option for 'Latest Run Start Date'. The left sidebar contains links for Home, Dags, Assets, Browse, Admin, Docs, and User.

| Dag Name | Schedule | Latest Run | Next Run |
|--|--|------------------------|------------------------|
| toy_chain_linear_vs_chain_complex | chain(), dependency_functions, core, , +2 more | 2025-04-21, 13:11:17 ✓ | |
| example_branch_dop_operator_v3 | example | 2025-04-21, 13:11:00 ✓ | 2025-04-21, 13:12:00 |
| example_passing_params_via_test_command | example | 2025-04-21, 13:11:00 ✘ | 2025-04-21, 13:12:00 |
| asset_consumes_1 | asset-scheduled, consumes | s3://dag1/output_1.txt | 2025-04-21, 13:10:18 ✓ |
| consume_1_or_both_2_and_3_with_asset_expressions | | 0 of 3 assets updated | 2025-04-21, 13:10:18 ✓ |
| consume_1_or_2_with_asset_expressions | | 0 of 2 assets updated | 2025-04-21, 13:10:18 ✓ |
| asset_produces_1 | produces, asset-scheduled | 0 0 * * * | 2025-04-21, 13:10:10 ✓ |

Dags Runs Task Instances

Search Dags Advanced Search

All Failed Running Success Enabled Filter by tag Sort by Latest Run Start Date...

84 Dags

toy_chain_linear_vs_chain_complex chain(), dependency_functions, core, , +2 more

Schedule Latest Run Next Run
2025-04-21, 13:11:17 ✓

example_branch_dop_operator_v3 example

Schedule Latest Run Next Run
*/1***** 2025-04-21, 13:11:00 ✓ 2025-04-21, 13:12:00

example_passing_params_via_test_command example

Schedule Latest Run Next Run
*/1***** 2025-04-21, 13:11:00 ✘ 2025-04-21, 13:12:00

asset_consumes_1 asset-scheduled, consumes

Schedule Latest Run Next Run
s3://dag1/output_1.txt 2025-04-21, 13:10:18 ✓

consume_1_or_both_2_and_3_with_asset_expressions

Schedule Latest Run Next Run
0 of 3 assets updated 2025-04-21, 13:10:18 ✓

consume_1_or_2_with_asset_expressions

Schedule Latest Run Next Run
0 of 2 assets updated 2025-04-21, 13:10:18 ✓

asset_produces_1 produces, asset-scheduled

Schedule Latest Run Next Run
0 0 * * * 2025-04-21, 13:10:10 ✓ 2025-04-21, 20:00:00

Some DAGs in this list may interact with data assets. For example, DAGs that are triggered by asset conditions may display popups showing upstream asset inputs.

consume_1_or_2_with_asset_expressions

Schedule Latest Run Next Run
0 of 2 assets updated 2025-04-21, 13:10:18 ✓

asset_with_extra_by_context produces

Schedule Latest Run Next Run
0 0 * * * 2025-04-21, 13:09:53 ✓ 2025-04-21, 20:00:00

The screenshot shows the DAG Details Page with three DAGs listed:

- consume_1_or_2_with_asset_expressions**: Schedule: 0 of 2 assets updated. Latest Run: 2025-04-21, 13:10:18. Next Run: 2025-04-21, 20:00:00. Contains tasks: s3://dag1/output_1.txt and s3://dag2/output_1.txt connected by an OR operator.
- asset_with_extra_by_context**: Schedule: 0 of 0 ***. Latest Run: 2025-04-21, 13:09:53. Next Run: 2025-04-21, 20:00:00. Contains task: asset_with_extra_by_context produces.

3.6.3 DAG Details Page

Clicking a DAG from the list opens the DAG Details Page. This view offers centralized access to a DAG's metadata, recent activity, and task-level diagnostics.

Key elements include:

- **DAG metadata**, including ID, owner, tags, schedule, and latest DAG version
- **Action buttons** to trigger the DAG, reparse it, or pause/resume
- **Tabbed interface**: Overview (recent failures, run counts, task logs); Grid View (status heatmap); Graph View (task dependencies); Runs (full run history); Tasks (aggregated stats); Events (system- or asset-triggered); Code (DAG source); and Details (extended metadata)

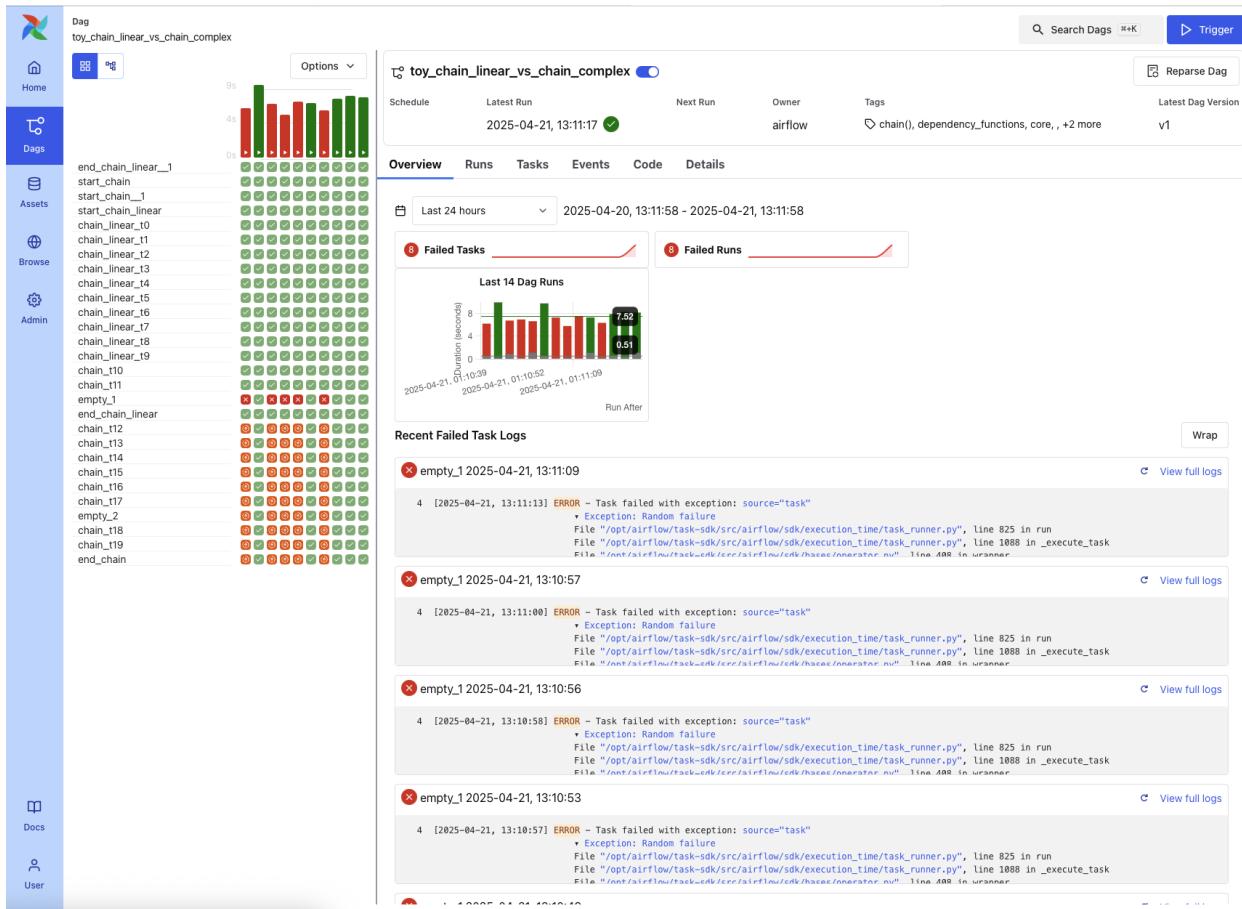
This page also includes a visual **timeline of recent DAG runs** and a **log preview for failures**, helping users quickly identify issues across runs.

The screenshot shows the DAG Details Page for the 'toy_chain_linear_vs_chain_complex' DAG. The top navigation bar includes a search bar, a 'Trigger' button, and a 'Reparse Dag' button. The main area displays the DAG structure, recent runs, and failed tasks.

Recent Failed Task Logs:

- empty_1 2025-04-21, 13:11:09
- empty_1 2025-04-21, 13:10:57
- empty_1 2025-04-21, 13:10:56
- empty_1 2025-04-21, 13:10:53

Each log entry shows an error message indicating a task failure due to a random exception. The logs provide the timestamp, task name, and a detailed traceback of the error.



Grid View

The Grid View is the primary interface for inspecting DAG runs and task states. It offers an interactive way to debug, retry, or monitor workflows over time.

Use Grid View to:

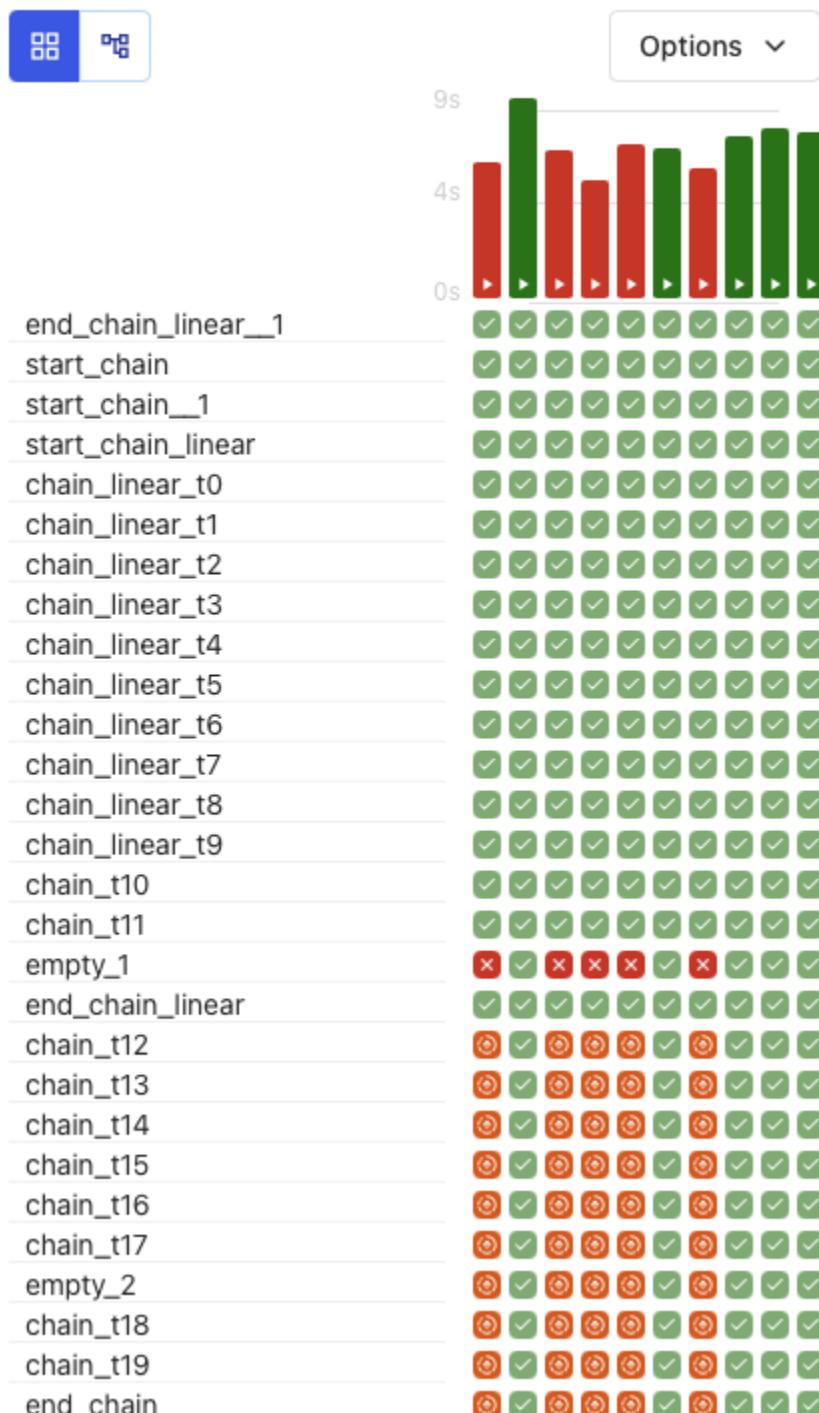
- **Understand the status of recent DAG runs** at a glance
- **Identify failed or retried tasks** by color and tooltip
- **Take action** by clicking a task cell to view logs or mark task instances as successful, failed, or cleared
- **Filter tasks** by name or partial ID
- **Select a run range**, like “last 25 runs” using the dropdown above the grid

Each row represents a task, and each column represents a DAG run. You can hover over any task instance for more detail, or click to drill down into logs and metadata.



Dag

toy_chain_linear_vs_chain_complex



Graph View

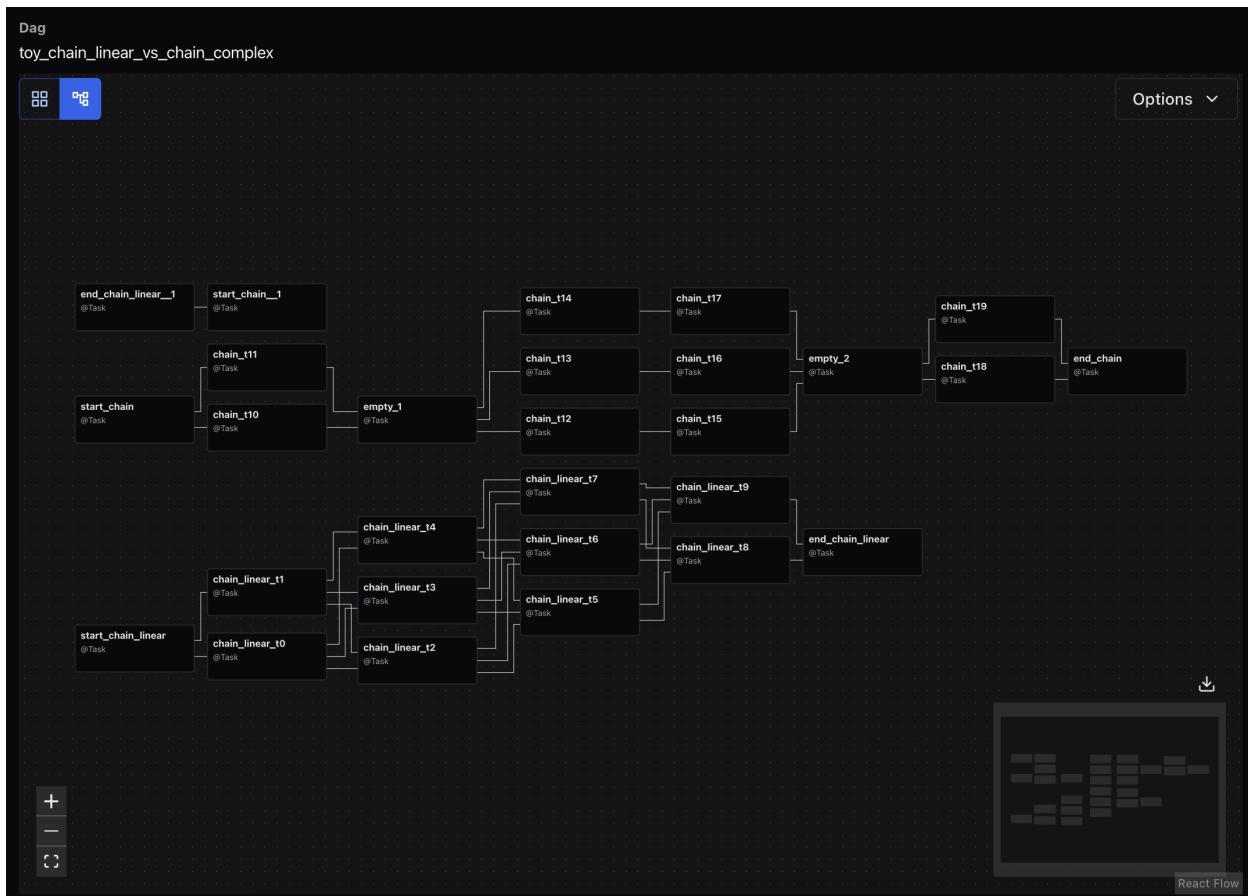
The Graph View shows the logical structure of your DAG—how tasks are connected, what order they run in, and how branching or retries are configured.

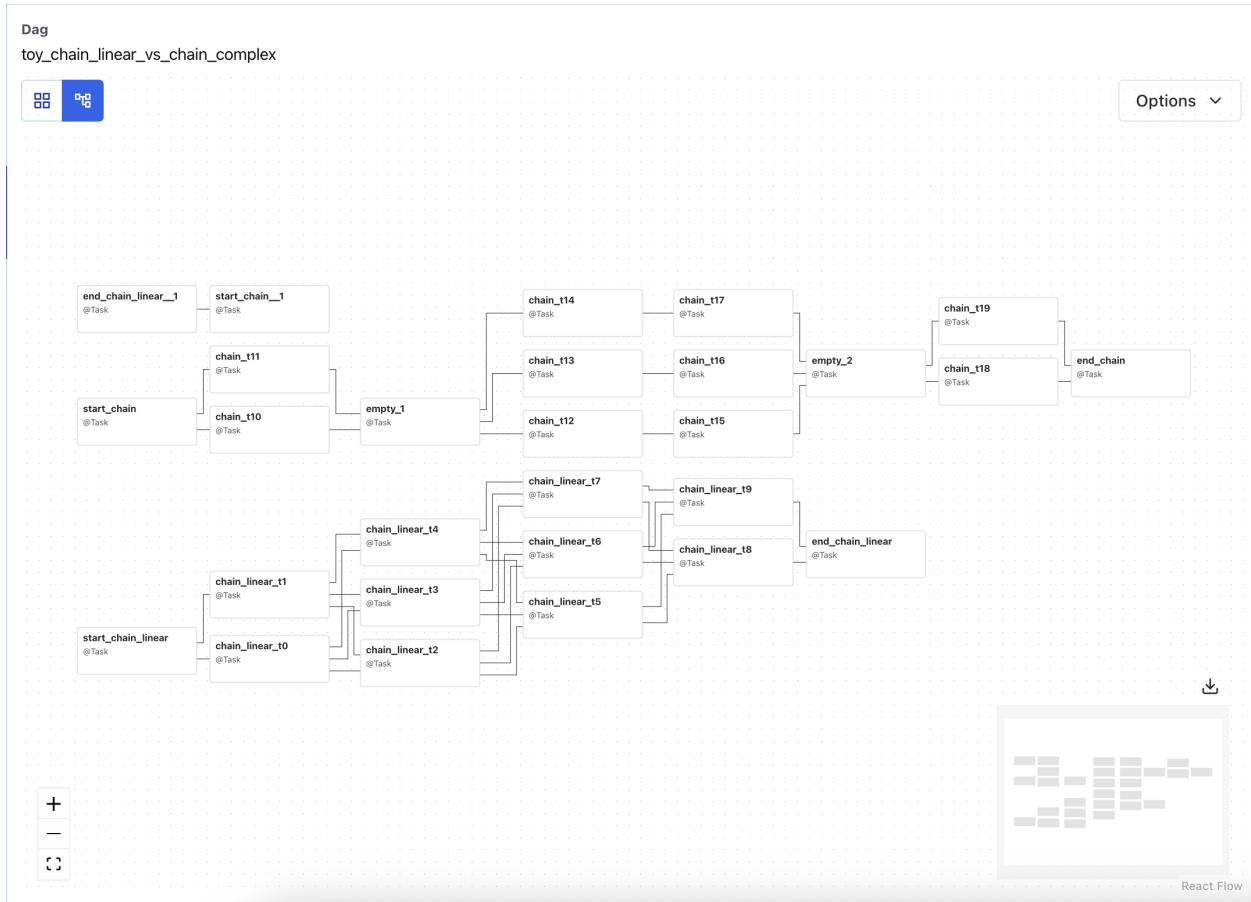
This view is helpful when:

- **Debugging why a task didn't run** (e.g., skipped due to a trigger rule)
- **Understanding task dependencies** across complex pipelines
- **Inspecting run-specific task states** (e.g., success, failed, upstream failed)

Each node represents a task. Edges show the dependencies between them. You can click any task to view its metadata and recent run history.

Use the dropdown at the top to switch between DAG runs and see how task states changed across executions.





3.6.4 DAG Tabs

In addition to the interactive views like Grid and Graph, the DAG Details page includes several other tabs that provide deeper insights and metadata:

Runs Tab

The **Runs** tab displays a sortable table of all DAG runs, along with their status, execution duration, run type, and DAG version.

Dag: toy_chain_linear_vs_chain_complex

Trigger

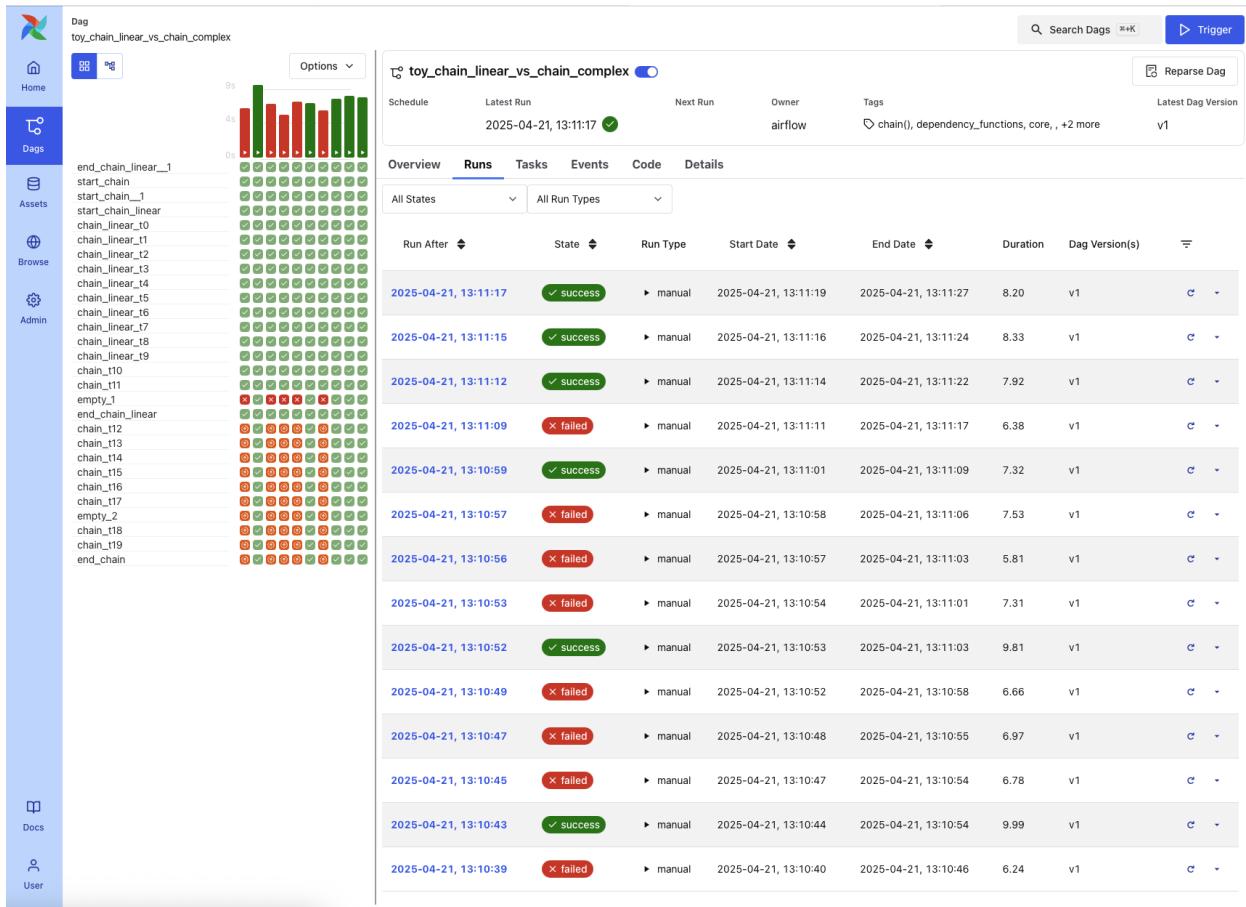
Reparse Dag

Schedule: Latest Run: 2025-04-21, 13:11:17 ✓ Owner: airflow Tags: chain(), dependency_functions, core, , +2 more Latest Dag Version: v1

Overview Runs Tasks Events Code Details

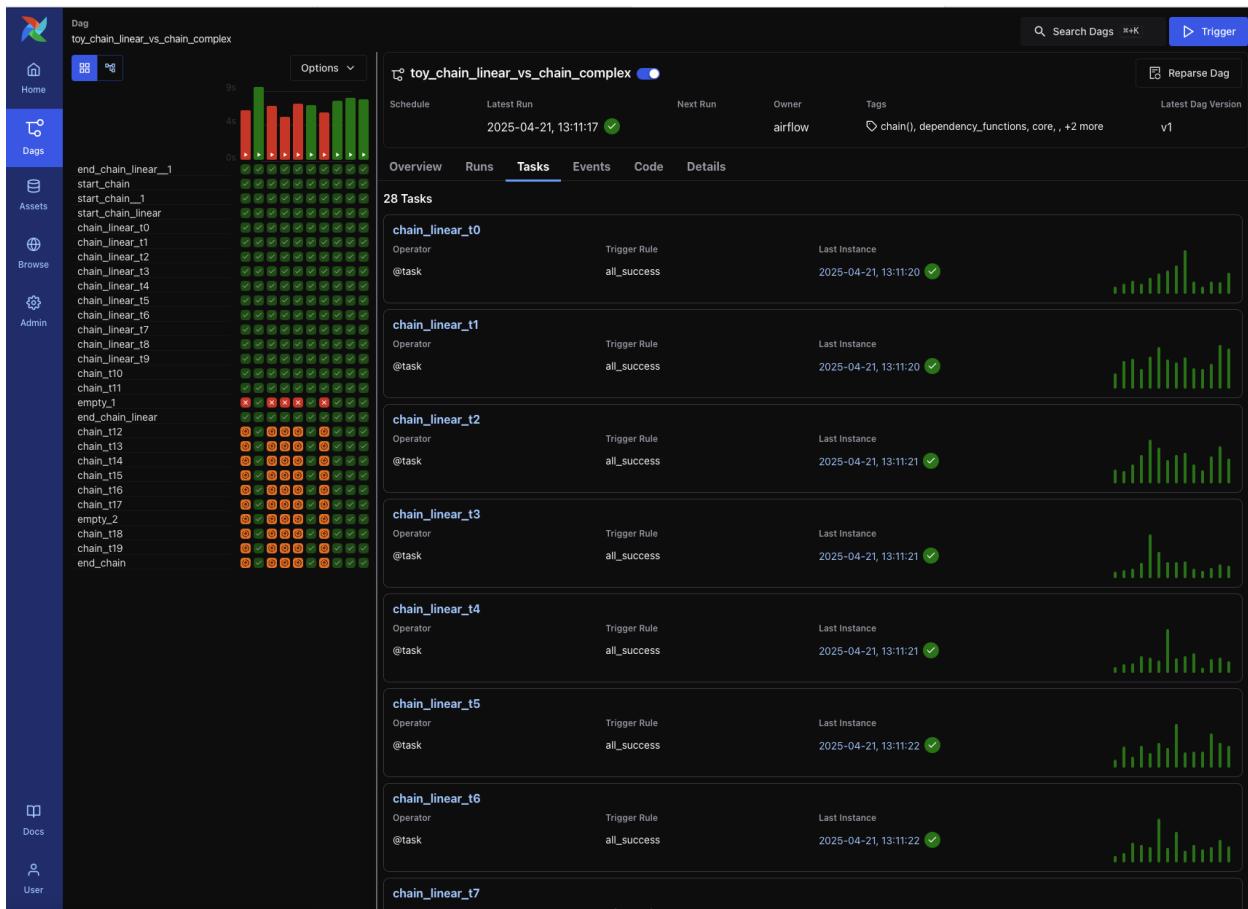
All States All Run Types

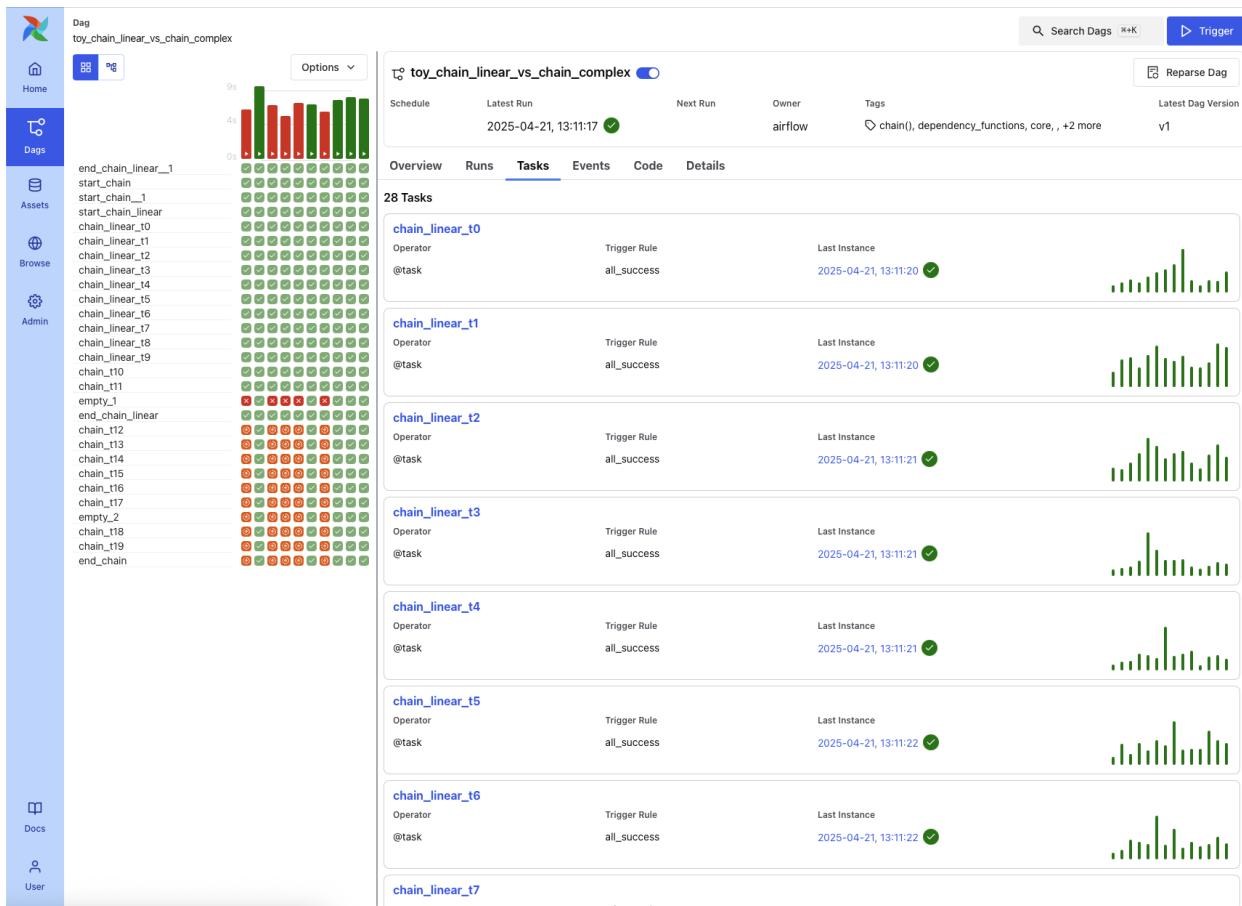
| Run After | State | Run Type | Start Date | End Date | Duration | Dag Version(s) |
|----------------------|-----------|----------|----------------------|----------------------|----------|----------------|
| 2025-04-21, 13:11:17 | ✓ success | ▶ manual | 2025-04-21, 13:11:19 | 2025-04-21, 13:11:27 | 8.20 | v1 |
| 2025-04-21, 13:11:15 | ✓ success | ▶ manual | 2025-04-21, 13:11:16 | 2025-04-21, 13:11:24 | 8.33 | v1 |
| 2025-04-21, 13:11:12 | ✓ success | ▶ manual | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:22 | 7.92 | v1 |
| 2025-04-21, 13:11:09 | ✗ failed | ▶ manual | 2025-04-21, 13:11:11 | 2025-04-21, 13:11:17 | 6.38 | v1 |
| 2025-04-21, 13:10:59 | ✓ success | ▶ manual | 2025-04-21, 13:11:01 | 2025-04-21, 13:11:09 | 7.32 | v1 |
| 2025-04-21, 13:10:57 | ✗ failed | ▶ manual | 2025-04-21, 13:10:58 | 2025-04-21, 13:11:06 | 7.53 | v1 |
| 2025-04-21, 13:10:56 | ✗ failed | ▶ manual | 2025-04-21, 13:10:57 | 2025-04-21, 13:11:03 | 5.81 | v1 |
| 2025-04-21, 13:10:53 | ✗ failed | ▶ manual | 2025-04-21, 13:10:54 | 2025-04-21, 13:11:01 | 7.31 | v1 |
| 2025-04-21, 13:10:52 | ✓ success | ▶ manual | 2025-04-21, 13:10:53 | 2025-04-21, 13:11:03 | 9.81 | v1 |
| 2025-04-21, 13:10:49 | ✗ failed | ▶ manual | 2025-04-21, 13:10:52 | 2025-04-21, 13:10:58 | 6.66 | v1 |
| 2025-04-21, 13:10:47 | ✗ failed | ▶ manual | 2025-04-21, 13:10:48 | 2025-04-21, 13:10:55 | 6.97 | v1 |
| 2025-04-21, 13:10:45 | ✗ failed | ▶ manual | 2025-04-21, 13:10:47 | 2025-04-21, 13:10:54 | 6.78 | v1 |
| 2025-04-21, 13:10:43 | ✓ success | ▶ manual | 2025-04-21, 13:10:44 | 2025-04-21, 13:10:54 | 9.99 | v1 |
| 2025-04-21, 13:10:39 | ✗ failed | ▶ manual | 2025-04-21, 13:10:40 | 2025-04-21, 13:10:46 | 6.24 | v1 |



Tasks Tab

The **Tasks** tab shows metadata for each task in the DAG, including operator type, trigger rule, most recent run status, and run history.





Events Tab

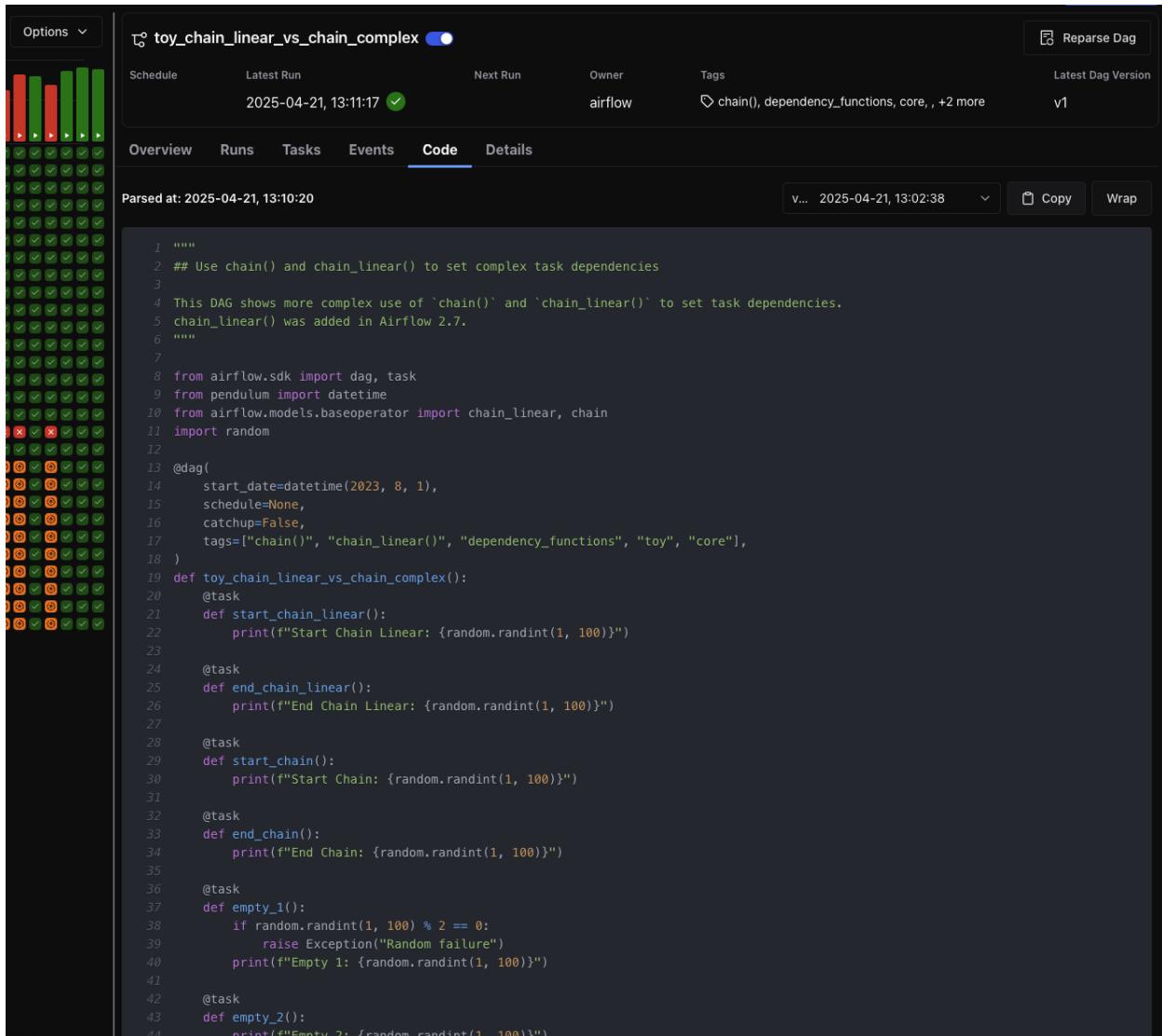
The **Events** tab surfaces structured events related to the DAG, such as DAG triggers and version patches. This tab is especially useful for DAG versioning and troubleshooting changes.

| toy_chain_linear_vs_chain_complex | | Reparse Dag | | | |
|-----------------------------------|------------|----------------------|-----------|---|--------------------|
| Schedule | Latest Run | Next Run | Owner | Tags | Latest Dag Version |
| | | 2025-04-21, 13:11:17 | airflow | chain(), dependency_functions, core,, +2 more | v1 |
| Overview | Runs | Tasks | Events | Code | Details |
| When | Run ID | Task ID | Map Index | Try Number | Event |
| 2025-04-21, 13:11:18 | | | | | trigger_dag_run |
| 2025-04-21, 13:11:16 | | | | | trigger_dag_run |
| 2025-04-21, 13:11:13 | | | | | trigger_dag_run |
| 2025-04-21, 13:11:10 | | | | | trigger_dag_run |
| 2025-04-21, 13:11:00 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:58 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:57 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:54 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:52 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:50 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:48 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:46 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:44 | | | | | trigger_dag_run |
| 2025-04-21, 13:10:40 | | | | | trigger_dag_run |
| 2025-04-21, 13:03:26 | | | | | patch_dag |

| When | Run ID | Task ID | Map Index | Try Number | Event | User |
|----------------------|--------|---------|-----------|------------|-----------------|-------|
| 2025-04-21, 13:11:18 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:11:16 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:11:13 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:11:10 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:11:00 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:58 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:57 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:54 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:52 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:50 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:48 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:46 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:44 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:10:40 | | | | | trigger_dag_run | admin |
| 2025-04-21, 13:03:26 | | | | | patch_dag | admin |

Code Tab

The **Code** tab displays the current version of the DAG definition, including the timestamp of the last parse. Users can view the code for any specific DAG version.



The screenshot shows the Apache Airflow web interface for a DAG named 'toy_chain_linear_vs_chain_complex'. The 'Code' tab is selected, displaying the Python code for the DAG. The code demonstrates the use of `chain()` and `chain_linear()` to manage complex task dependencies. It includes imports for `dag`, `task`, `pendulum`, and specific operators from `airflow.models.baseoperator`. The DAG has a start date of August 2023, no schedule, and no catchup. It features tasks for starting and ending a chain linearly, and two empty tasks that raise exceptions under certain conditions. The code is annotated with line numbers from 1 to 44.

```

1 """
2 ## Use chain() and chain_linear() to set complex task dependencies
3
4 This DAG shows more complex use of `chain()` and `chain_linear()` to set task dependencies.
5 chain_linear() was added in Airflow 2.7.
6 """
7
8 from airflow.sdk import dag, task
9 from pendulum import datetime
10 from airflow.models.baseoperator import chain_linear, chain
11 import random
12
13 @dag(
14     start_date=datetime(2023, 8, 1),
15     schedule=None,
16     catchup=False,
17     tags=["chain()", "chain_linear()", "dependency_functions", "toy", "core"],
18 )
19 def toy_chain_linear_vs_chain_complex():
20     @task
21     def start_chain_linear():
22         print(f"Start Chain Linear: {random.randint(1, 100)}")
23
24     @task
25     def end_chain_linear():
26         print(f"End Chain Linear: {random.randint(1, 100)}")
27
28     @task
29     def start_chain():
30         print(f"Start Chain: {random.randint(1, 100)}")
31
32     @task
33     def end_chain():
34         print(f"End Chain: {random.randint(1, 100)}")
35
36     @task
37     def empty_1():
38         if random.randint(1, 100) % 2 == 0:
39             raise Exception("Random failure")
40         print(f"Empty 1: {random.randint(1, 100)}")
41
42     @task
43     def empty_2():
44         print(f"Empty 2: {random.randint(1, 100)}")

```

```

1 """
2 ## Use chain() and chain_linear() to set complex task dependencies
3
4 This DAG shows more complex use of `chain()` and `chain_linear()` to set task dependencies.
5 chain_linear() was added in Airflow 2.7.
6 """
7
8 from airflow.sdk import dag, task
9 from pendulum import datetime
10 from airflow.models.baseoperator import chain_linear, chain
11 import random
12
13 @dag(
14     start_date=datetime(2023, 8, 1),
15     schedule=None,
16     catchup=False,
17     tags=['chain()', "chain_linear()", "dependency_functions", "toy", "core"],
18 )
19 def toy_chain_linear_vs_chain_complex():
20     @task
21     def start_chain_linear():
22         print(f"Start Chain Linear: {random.randint(1, 100)}")
23
24     @task
25     def end_chain_linear():
26         print(f"End Chain Linear: {random.randint(1, 100)}")
27
28     @task
29     def start_chain():
30         print(f"Start Chain: {random.randint(1, 100)}")
31
32     @task
33     def end_chain():
34         print(f"End Chain: {random.randint(1, 100)}")
35
36     @task
37     def empty_1():
38         if random.randint(1, 100) % 2 == 0:
39             raise Exception("Random failure")
40         print(f"Empty 1: {random.randint(1, 100)}")
41
42     @task
43     def empty_2():
44         print(f"Empty 2: {random.randint(1, 100)}")

```

Details Tab

The **Details** tab provides configuration details and metadata for the DAG, including schedule, file location, concurrency limits, and version identifiers.

toy_chain_linear_vs_chain_complex

Schedule Latest Run Next Run Owner Tags Latest Dag Version

2025-04-21, 13:11:17 ✓ airflow chain(), dependency_functions, core, , +2 more v1

[Reparse Dag](#)

Overview **Runs** **Tasks** **Events** **Code** **Details**

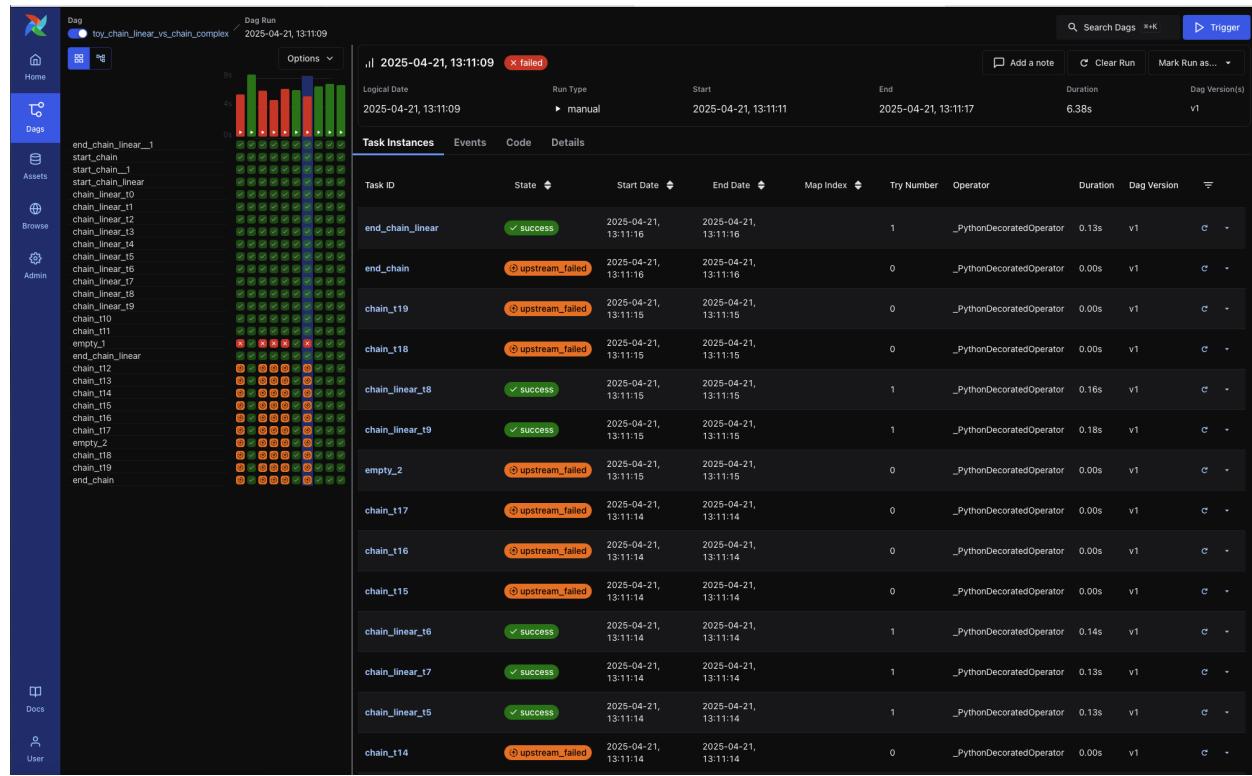
| | |
|---------------------------------|--------------------------------------|
| Dag ID | toy_chain_linear_vs_chain_complex |
| Description | |
| Timezone | UTC |
| File Location | /files/dags/complex.py |
| Last Parsed | 2025-04-21, 13:10:39 |
| Version ID | 0196594d-4412-7142-8109-3fa5abb24078 |
| Latest Dag Version | bundle_name: dags-folder |
| Created At | 2025-04-21, 13:02:38 |
| Start Date | 2023-07-31, 20:00:00 |
| End Date | |
| Last Expired | |
| Concurrency | 16 |
| Has Task Concurrency Limits | false |
| Dag Run Timeout | |
| Max Active Runs | 16 |
| Max Active Tasks | 16 |
| Max Consecutive Failed Dag Runs | 0 |
| Catchup | false |
| Params | {} 0 items |

The screenshot shows the 'Details' tab of a DAG configuration page. At the top, there's a header with the DAG ID 'toy_chain_linear_vs_chain_complex' and a 'Reparse Dag' button. Below the header, there are sections for 'Schedule' (Latest Run: 2025-04-21, 13:11:17), 'Owner' (airflow), 'Tags' (chain(), dependency_functions, core, , +2 more), and 'Latest Dag Version' (v1). A navigation bar below the header includes 'Overview', 'Runs', 'Tasks', 'Events', 'Code', and 'Details'.

| Dag ID | toy_chain_linear_vs_chain_complex |
|---------------------------------|--------------------------------------|
| Description | |
| Timezone | UTC |
| File Location | /files/dags/complex.py |
| Last Parsed | 2025-04-21, 13:10:39 |
| Version ID | 0196594d-4412-7142-8109-3fa5abb24078 |
| Latest Dag Version | Bundle Name: dags-folder |
| | Created At: 2025-04-21, 13:02:38 |
| Start Date | 2023-07-31, 20:00:00 |
| End Date | |
| Last Expired | |
| Concurrency | 16 |
| Has Task Concurrency Limits | false |
| Dag Run Timeout | |
| Max Active Runs | 16 |
| Max Active Tasks | 16 |
| Max Consecutive Failed Dag Runs | 0 |
| Catchup | false |
| Params | <code>{}</code> 0 items |

3.6.5 DAG Run View

Each DAG Run has its own view, accessible by selecting a specific row in the DAG's **Runs** tab. The DAG Run view displays metadata about the selected run, as well as task-level details, rendered code, and more.



Key elements include:

- **DAG Run metadata**, including logical date, run type, duration, DAG version, and parsed time
- **Action buttons** to clear or mark the run, or add a note
- A persistent **Grid View sidebar**, which shows task durations and states across recent DAG runs. This helps spot recurring issues or performance trends at a glance.

3.6.6 DAG Run Tabs

Task Instances

Displays the status and metadata for each task instance within the DAG Run. Columns include:

- Task ID
- State
- Start and End Dates
- Try Number
- Operator Type
- Duration
- DAG Version

Each row also includes a mini Gantt-style timeline that visually represents the task's duration.

| Task ID | State | Start Date | End Date | Map Index | Try Number | Operator | Duration | Dag Version |
|------------------|-------------------|----------------------|----------------------|-----------|------------|--------------------------|----------|-------------|
| end_chain_linear | ✓ success | 2025-04-21, 13:11:16 | 2025-04-21, 13:11:16 | | 1 | _PythonDecoratedOperator | 0.13s | v1 |
| end_chain | ⌚ upstream_failed | 2025-04-21, 13:11:16 | 2025-04-21, 13:11:16 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_t19 | ⌚ upstream_failed | 2025-04-21, 13:11:15 | 2025-04-21, 13:11:15 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_t18 | ⌚ upstream_failed | 2025-04-21, 13:11:15 | 2025-04-21, 13:11:15 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_linear_18 | ✓ success | 2025-04-21, 13:11:15 | 2025-04-21, 13:11:15 | | 1 | _PythonDecoratedOperator | 0.16s | v1 |
| chain_linear_t9 | ✓ success | 2025-04-21, 13:11:15 | 2025-04-21, 13:11:15 | | 1 | _PythonDecoratedOperator | 0.18s | v1 |
| empty_2 | ⌚ upstream_failed | 2025-04-21, 13:11:15 | 2025-04-21, 13:11:15 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_t17 | ⌚ upstream_failed | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_t16 | ⌚ upstream_failed | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_t15 | ⌚ upstream_failed | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |
| chain_linear_t6 | ✓ success | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 1 | _PythonDecoratedOperator | 0.14s | v1 |
| chain_linear_t7 | ✓ success | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 1 | _PythonDecoratedOperator | 0.13s | v1 |
| chain_linear_t5 | ✓ success | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 1 | _PythonDecoratedOperator | 0.13s | v1 |
| chain_t14 | ⌚ upstream_failed | 2025-04-21, 13:11:14 | 2025-04-21, 13:11:14 | | 0 | _PythonDecoratedOperator | 0.00s | v1 |

Events

If available, this tab lists system-level or asset-triggered events that contributed to this DAG Run's execution.

Code

Displays the DAG source code as it was at the time this run was parsed. This view is helpful for debugging version drift or comparing behavior across DAG Runs that used different code.

DAG Run code for `hello >> airflow()`:

Dag Run: demo / Dag Run: 2025-04-20, 20:00:00

Logical Date: 2025-04-20, 20:00:00 | Run Type: scheduled | Start: 2025-04-21, 13:04:56 | End: 2025-04-21, 13:04:57 | Duration: 0.73s | Dag Version(s): v1

Code

```

1 from datetime import datetime
2
3 from airflow.sdk import DAG, task
4 from airflow.providers.standard.operators.bash import BashOperator
5
6 # A DAG represents a workflow, a collection of tasks
7 with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
8     # Tasks are represented as operators
9     hello = BashOperator(task_id="hello", bash_command="echo hello")
10
11     @task()
12     def airflow():
13         print("airflow")
14
15
16     # Set dependencies between tasks
17     hello >> airflow()
18
19
20

```

Dag Run: demo / Dag Run: 2025-04-20, 20:00:00

Logical Date: 2025-04-20, 20:00:00 | Run Type: scheduled | Start: 2025-04-21, 13:04:56 | End: 2025-04-21, 13:04:57 | Duration: 0.73s | Dag Version(s): v1

Code

```

1 from datetime import datetime
2
3 from airflow.sdk import DAG, task
4 from airflow.providers.standard.operators.bash import BashOperator
5
6 # A DAG represents a workflow, a collection of tasks
7 with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
8     # Tasks are represented as operators
9     hello = BashOperator(task_id="hello", bash_command="echo hello")
10
11     @task()
12     def airflow():
13         print("airflow")
14
15
16     # Set dependencies between tasks
17     hello >> airflow()
18
19
20

```

DAG Run code for `hello >> world()`:

Dag demo Dag Run 2025-04-21, 19:17:14

Logical Date: 2025-04-21, 19:17:14 Run Type: manual Start: 2025-04-21, 19:17:16 End: 2025-04-21, 19:17:17 Duration: 1.73s Dag Version(s): v2

Code

```

1 from datetime import datetime
2
3 from airflow.sdk import DAG, task
4 from airflow.providers.standard.operators.bash import BashOperator
5
6 # A DAG represents a workflow, a collection of tasks
7 with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
8     # Tasks are represented as operators
9     hello = BashOperator(task_id="hello", bash_command="echo hello")
10
11     @task()
12     def world():
13         print("world")
14
15
16     # Set dependencies between tasks
17     hello >> world()
18
19
20

```

Dag demo Dag Run 2025-04-21, 19:17:14

Logical Date: 2025-04-21, 19:17:14 Run Type: manual Start: 2025-04-21, 19:17:16 End: 2025-04-21, 19:17:17 Duration: 1.73s Dag Version(s): v2

Code

```

1 from datetime import datetime
2
3 from airflow.sdk import DAG, task
4 from airflow.providers.standard.operators.bash import BashOperator
5
6 # A DAG represents a workflow, a collection of tasks
7 with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
8     # Tasks are represented as operators
9     hello = BashOperator(task_id="hello", bash_command="echo hello")
10
11     @task()
12     def world():
13         print("world")
14
15
16     # Set dependencies between tasks
17     hello >> world()
18
19
20

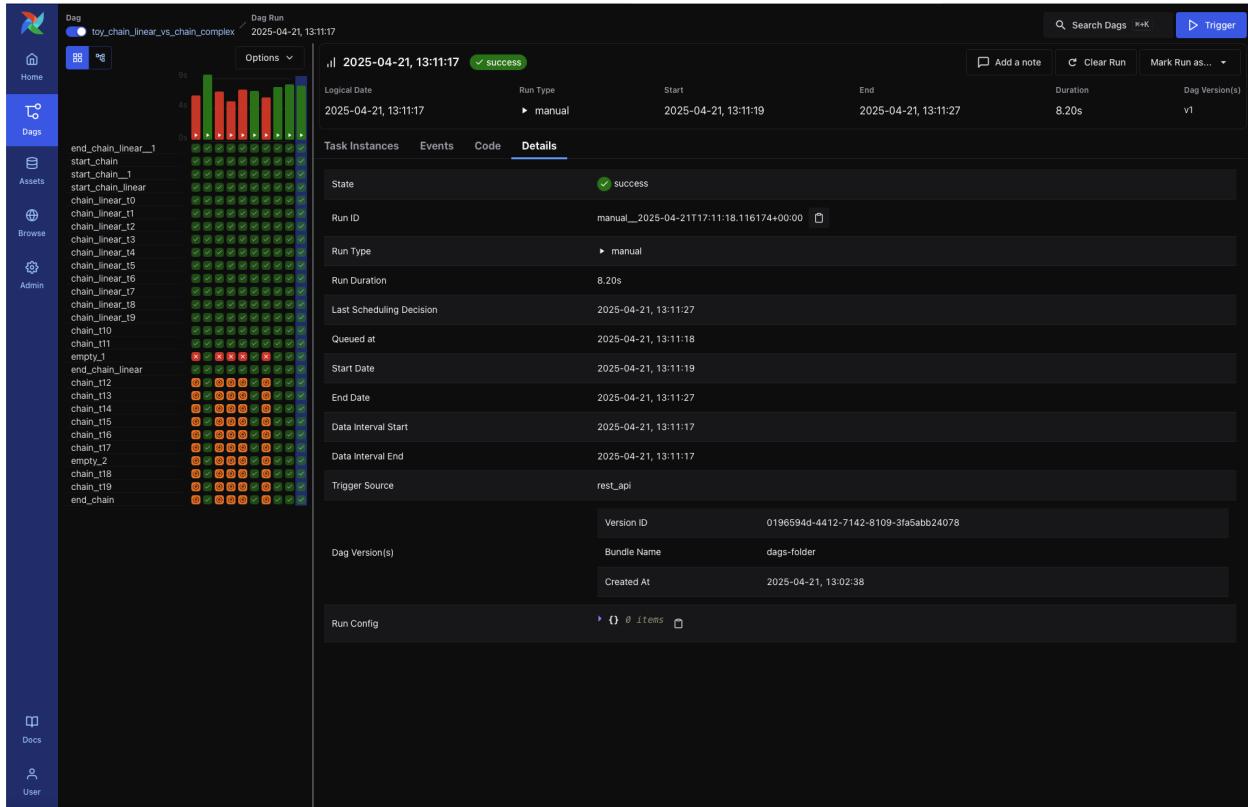
```

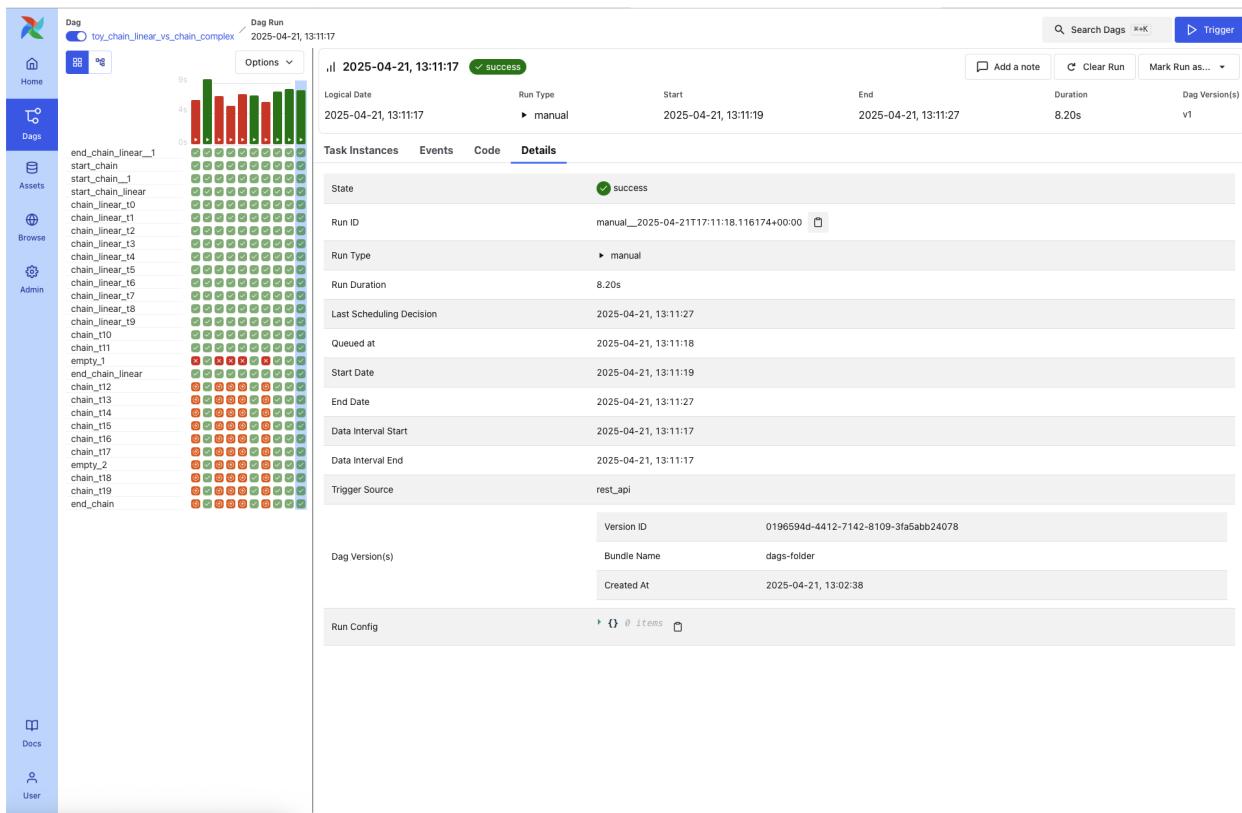
Details

Provides extended metadata for the DAG Run, including:

- Run ID and Trigger Type
- Queued At, Start and End Time, and Duration

- Data Interval boundaries
- Trigger Source and Run Config
- DAG Version ID and Bundle Name

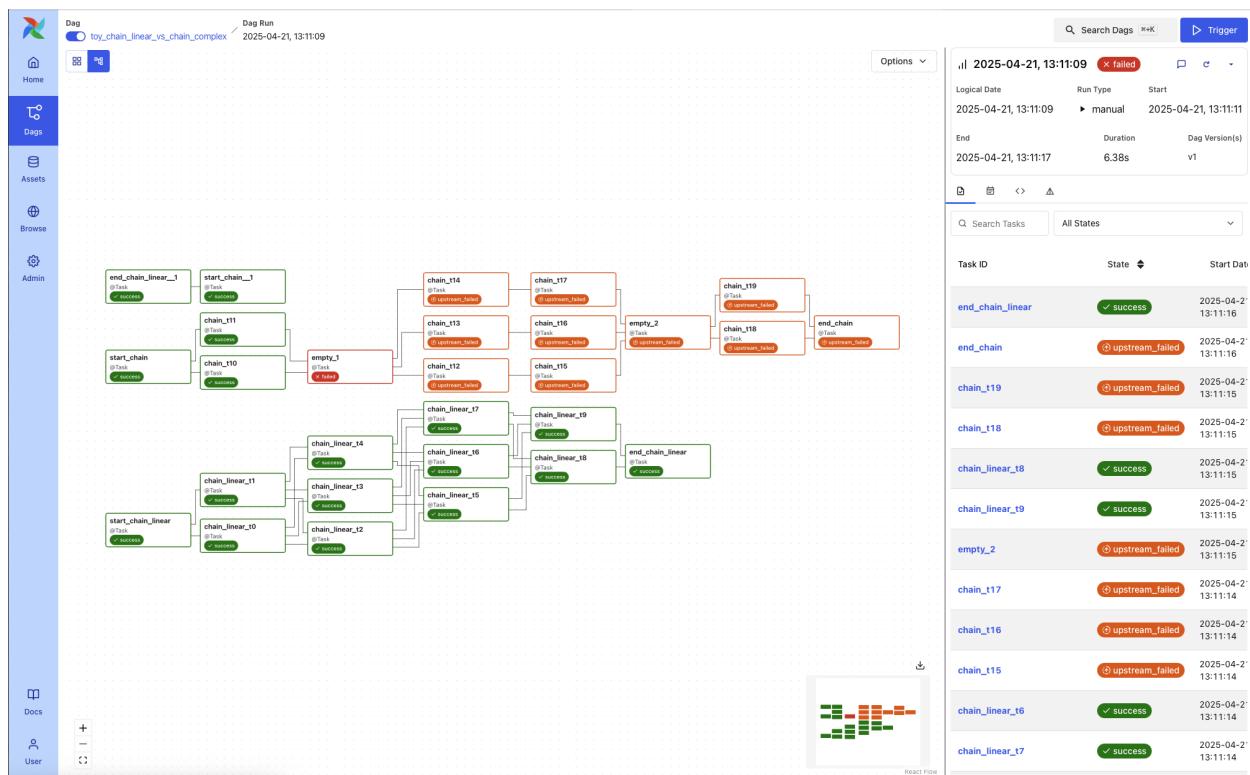
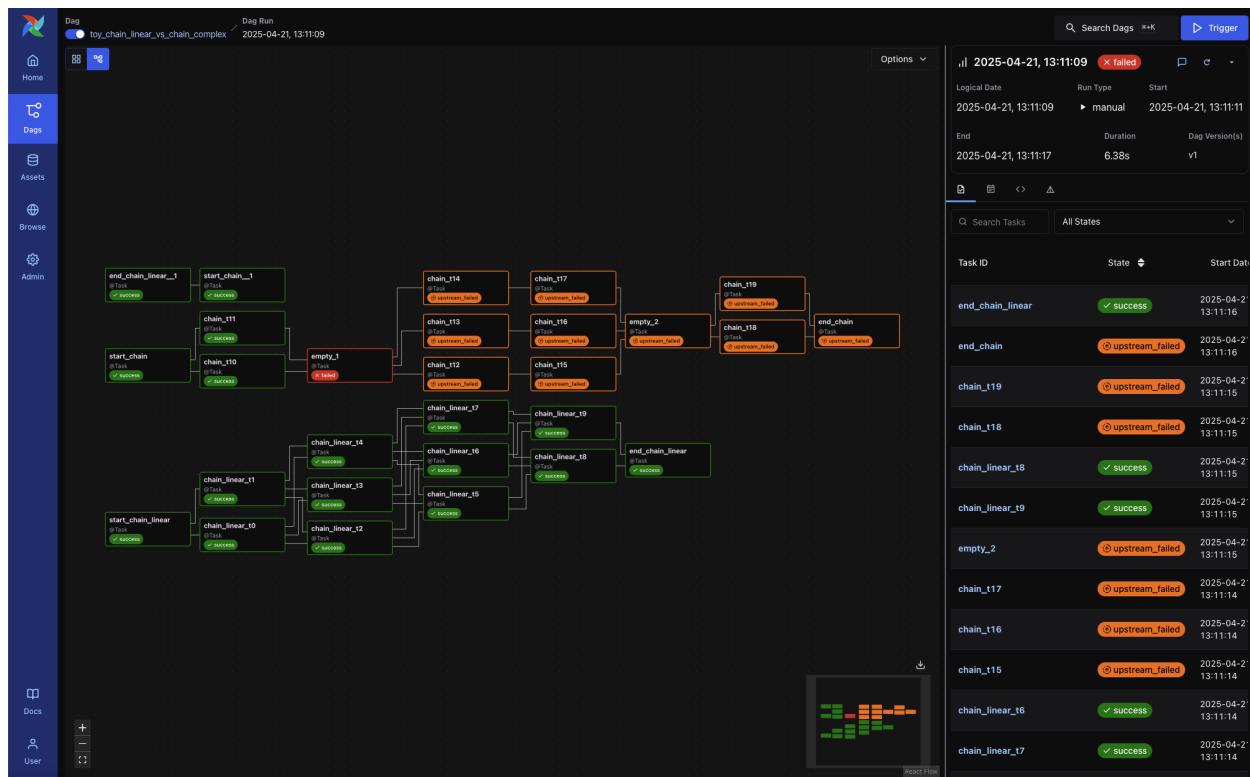




Graph View

Shows the DAG's task dependency structure overlaid with the status of each task in this specific run. This is useful for visual debugging of task failure paths or identifying downstream blockers.

Each node includes a visual indicator of task duration.



3.6.7 Task Instance View

When you click on a specific task from the DAG Run view, you're brought to the **Task Instance View**, which shows detailed logs and metadata for that individual task execution.

The screenshot shows the Task Instance View for the 'empty_1' task in the 'toy_chain_linear_vs_chain_complex' DAG run. The top navigation bar includes 'Dag', 'Dag Run' (2025-04-21, 13:11:09), and 'Task' (empty_1). The left sidebar has 'Home', 'Dags' (selected), 'Assets', 'Browse', and 'Admin'. The main area displays a timeline of task execution with nodes labeled 'empty_1' and 'chain_linear_1' through 'chain_linear_10'. A detailed log pane at the bottom shows the following log entries:

```

[2025-04-21, 13:11:13] INFO - Log message source details: source="/opt/airflow/logs/dag_id=toy_chain_linear_vs_chain_complex/run_id=manual_2025-04-21T17:11:10.873663+00:00/task_id=empty_1"
2 [2025-04-21, 13:11:13] INFO - DAG bundles loaded: dags-folder, example_dags, source="airflow.dag_processing.bundles_manager.DagBundlesManager"
3 [2025-04-21, 13:11:13] INFO - Filling up the DagBag from /files/dags/complex.py, source="airflow.models.dagbag.DagBag"
4 [2025-04-21, 13:11:13] ERROR - Task failed with exception: source="task"
   * Exception: Random failure
     File "/opt/airflow/task-sdks/src/airflow/sdks/execution_time/task_runner.py", line 825 in run
       File "/opt/airflow/task-sdks/src/airflow/sdks/execution_time/task_runner.py", line 3088 in _execute_task
         File "/opt/airflow/task-sdks/src/airflow/sdks/bases/operator.py", line 488 in wrapper
           File "/opt/airflow/task-sdks/src/airflow/sdks/bases/decorator.py", line 251 in execute
             File "/opt/airflow/task-sdks/src/airflow/sdks/bases/operator.py", line 488 in wrapper
               File "/opt/airflow/providers/standard/src/airflow/providers/standard/operators/python.py", line 212 in execute
                 File "/opt/airflow/providers/standard/src/airflow/providers/standard/operators/python.py", line 235 in execute_callable
                   File "/opt/airflow/task-sdks/src/airflow/sdks/execution_time/callback_runner.py", line 81 in run
                     File "/files/dags/complex.py", line 39 in empty_1
5 [2025-04-21, 13:11:13] INFO - Task instance is in running state: chan="stdout"; source="task"
6 [2025-04-21, 13:11:13] INFO - Previous state of the Task instance: queued; chan="stdout"; source="task"
7 [2025-04-21, 13:11:13] INFO - Current Task name:empty_1; chan="stdout"; source="task"
8 [2025-04-21, 13:11:13] INFO - Dag name:toy_chain_linear_vs_chain_complex; chan="stdout"; source="task"
9 [2025-04-21, 13:11:13] INFO - Task instance in failure state: chan="stdout"; source="task"
10 [2025-04-21, 13:11:13] INFO - Task start: chan="stdout"; source="task"
11 [2025-04-21, 13:11:13] INFO - Task<PythonDecoratedOperator>: empty_1>; chan="stdout"; source="task"
12 [2025-04-21, 13:11:13] INFO - Failure caused by Random failure: chan="stdout"; source="task"

```

3.6.8 Task Instance Tabs

Each task instance has a tabbed view providing access to logs, rendered templates, XComs, and execution metadata.

Logs

The default tab shows the task logs, which include system output, error messages, and traceback information. This is the first place to look when a task fails.

The screenshot shows the Task Instance View for the 'empty_1' task in the 'toy_chain_linear_vs_chain_complex' DAG run, with the 'Logs' tab selected. The log pane displays the same log entries as the previous screenshot.

Rendered Templates

Displays the rendered version of templated fields in your task. Useful for debugging context variables or verifying dynamic content.

XCom

Shows any values pushed via `XCom.push()` or returned from Python functions when using TaskFlow.

The screenshot shows the Airflow UI for a task instance named "bash_push" in a DAG named "example_xcom". The task was run on April 21, 2025, at 13:03:48. The XCom tab is selected, displaying two entries:

| Key | Dag | Run Id | Task ID | Map Index | Value |
|-----------------------|--------------|--------------------------------------|-----------|-----------|-----------------------|
| manually_pushed_value | example_xcom | scheduled__2021-01-01T00:00:00+00:00 | bash_push | -1 | manually_pushed_value |
| return_value | example_xcom | scheduled__2021-01-01T00:00:00+00:00 | bash_push | -1 | value_by_return |

The screenshot shows the Airflow UI for a task instance named "bash_push" in a DAG named "example_xcom". The task was run on April 21, 2025, at 13:03:48. The XCom tab is selected, displaying two entries:

| Key | Dag | Run Id | Task ID | Map Index | Value |
|-----------------------|--------------|--------------------------------------|-----------|-----------|-----------------------|
| manually_pushed_value | example_xcom | scheduled__2021-01-01T00:00:00+00:00 | bash_push | -1 | manually_pushed_value |
| return_value | example_xcom | scheduled__2021-01-01T00:00:00+00:00 | bash_push | -1 | value_by_return |

Events

If present, displays relevant events related to this specific task instance execution.

Code

Shows the DAG source code parsed at the time of execution. This helps verify what version of the DAG the task ran with.

Details

Displays runtime metadata about the task instance, including:

- Task ID and State
- DAG Run ID, DAG Version, and Bundle Name
- Operator used and runtime duration
- Pool and slot usage
- Executor and configuration

empty_1 2025-04-21, 13:11:13 x failed

Add a note | Clear Task Instance | Mark Task Instance as... ▾

| Operator | Start | End | Duration | DAG Version |
|--------------------------|----------------------|----------------------|----------|-------------|
| _PythonDecoratedOperator | 2025-04-21, 13:11:13 | 2025-04-21, 13:11:13 | 0.10s | v1 |

Logs Rendered Templates XCom Events Code Details

Task Instance Info

| | | |
|------------------|---|----------------------|
| State | x failed | |
| Task ID | empty_1 Copy | |
| Run ID | manual__2025-04-21T17:11:10.873663+00:00 Copy | |
| Map Index | -1 | |
| Operator | _PythonDecoratedOperator | |
| Duration | 0.10s | |
| Started | 2025-04-21, 13:11:13 | |
| Ended | 2025-04-21, 13:11:13 | |
| Version ID | 0196594d-4412-7142-8109-3fa5abb24078 | |
| Dag Version | Bundle Name | dags-folder |
| | Created At | 2025-04-21, 13:02:38 |
| Process ID (PID) | 1869 Copy | |
| Hostname | 615cd14891ab Copy | |
| Pool | default_pool | |
| Pool Slots | 1 | |
| Executor | | |
| Executor Config | {} | |
| Unix Name | root | |
| Max Tries | 0 | |

The screenshot shows the Apache Airflow task instance details page for a task named 'empty_1'. The task failed on April 21, 2025, at 13:11:13. The operator was '_PythonDecoratedOperator'. The task duration was 0.10s and it was part of DAG version v1.

Task Instance Info

| | | | | | | | |
|------------------|---|------------|--------------------------------------|-------------|-------------|------------|----------------------|
| State | failed | | | | | | |
| Task ID | empty_1 | | | | | | |
| Run ID | manual__2025-04-21T17:11:10.873663+00:00 | | | | | | |
| Map Index | -1 | | | | | | |
| Operator | _PythonDecoratedOperator | | | | | | |
| Duration | 0.10s | | | | | | |
| Started | 2025-04-21, 13:11:13 | | | | | | |
| Ended | 2025-04-21, 13:11:13 | | | | | | |
| Dag Version | <table border="1"> <tbody> <tr> <td>Version ID</td> <td>0196594d-4412-7142-8109-3fa5abb24078</td> </tr> <tr> <td>Bundle Name</td> <td>dags-folder</td> </tr> <tr> <td>Created At</td> <td>2025-04-21, 13:02:38</td> </tr> </tbody> </table> | Version ID | 0196594d-4412-7142-8109-3fa5abb24078 | Bundle Name | dags-folder | Created At | 2025-04-21, 13:02:38 |
| Version ID | 0196594d-4412-7142-8109-3fa5abb24078 | | | | | | |
| Bundle Name | dags-folder | | | | | | |
| Created At | 2025-04-21, 13:02:38 | | | | | | |
| Process ID (PID) | 1869 | | | | | | |
| Hostname | 615cd14891ab | | | | | | |
| Pool | default_pool | | | | | | |
| Pool Slots | 1 | | | | | | |
| Executor | | | | | | | |
| Executor Config | {} | | | | | | |
| Unix Name | root | | | | | | |
| Max Tries | 0 | | | | | | |

3.6.9 Asset Views

The **Assets** section provides a dedicated interface to monitor and debug asset-centric workflows. Assets represent logical data units—such as files, tables, or models—that tasks can produce or consume. Airflow tracks these dependencies and provides visualizations to better understand their orchestration.

Asset List

The Asset List shows all known assets, grouped by name. For each asset, you can see:

- The group the asset belongs to (if any)
- The DAGs that consume the asset
- The tasks that produce the asset

Hovering over a count of DAGs or tasks shows a tooltip with the full list of producers or consumers.

The screenshot shows the Apache Airflow UI's Assets page. A context menu is open over the asset named "test-asset". The menu items listed are:

- consume_1_or_both_2_and_3_with_asset_expressions
- asset_consumes_1_never_scheduled
- consume_1_or_2_with_asset_expressions
- asset_consumes_1
- asset_consumes_1_and_2
- consume_1_and_2_with_asset_expressions
- conditional_asset_and_time_based_timetable

The screenshot shows the Apache Airflow UI's Assets page. A context menu is open over the asset named "test-asset". The menu items listed are:

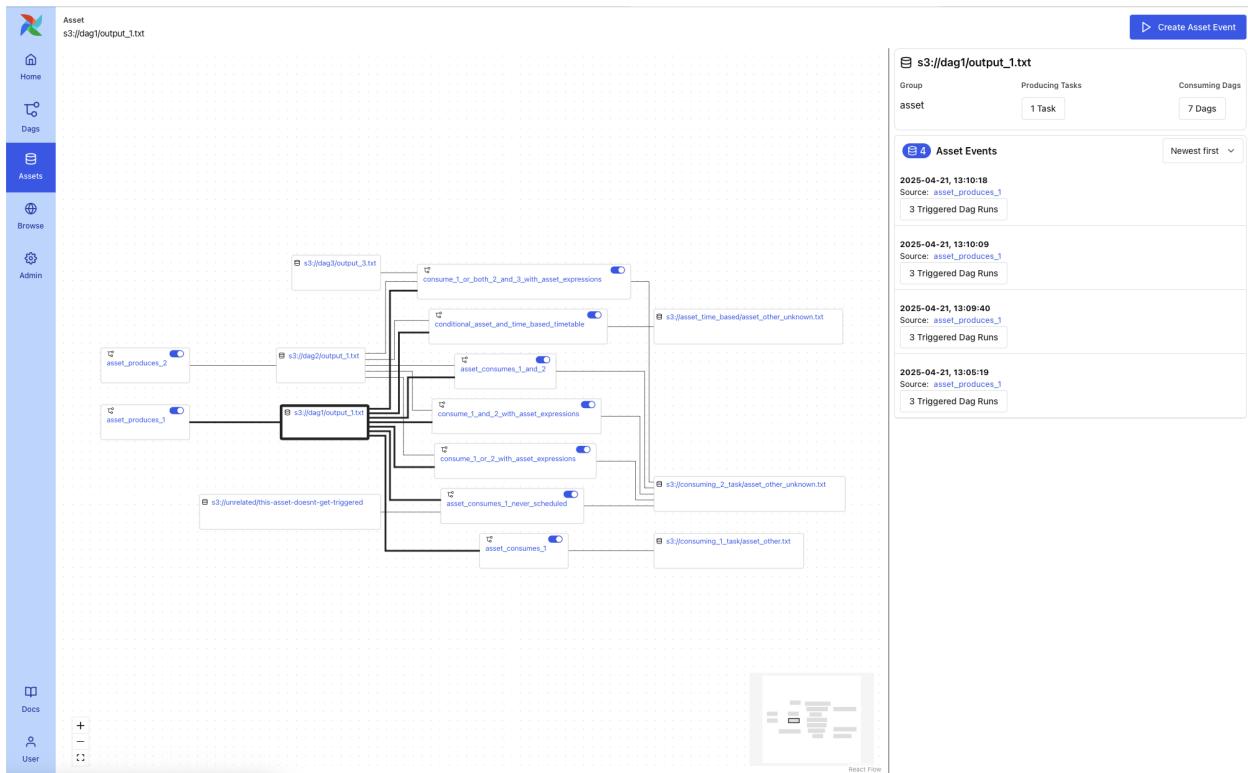
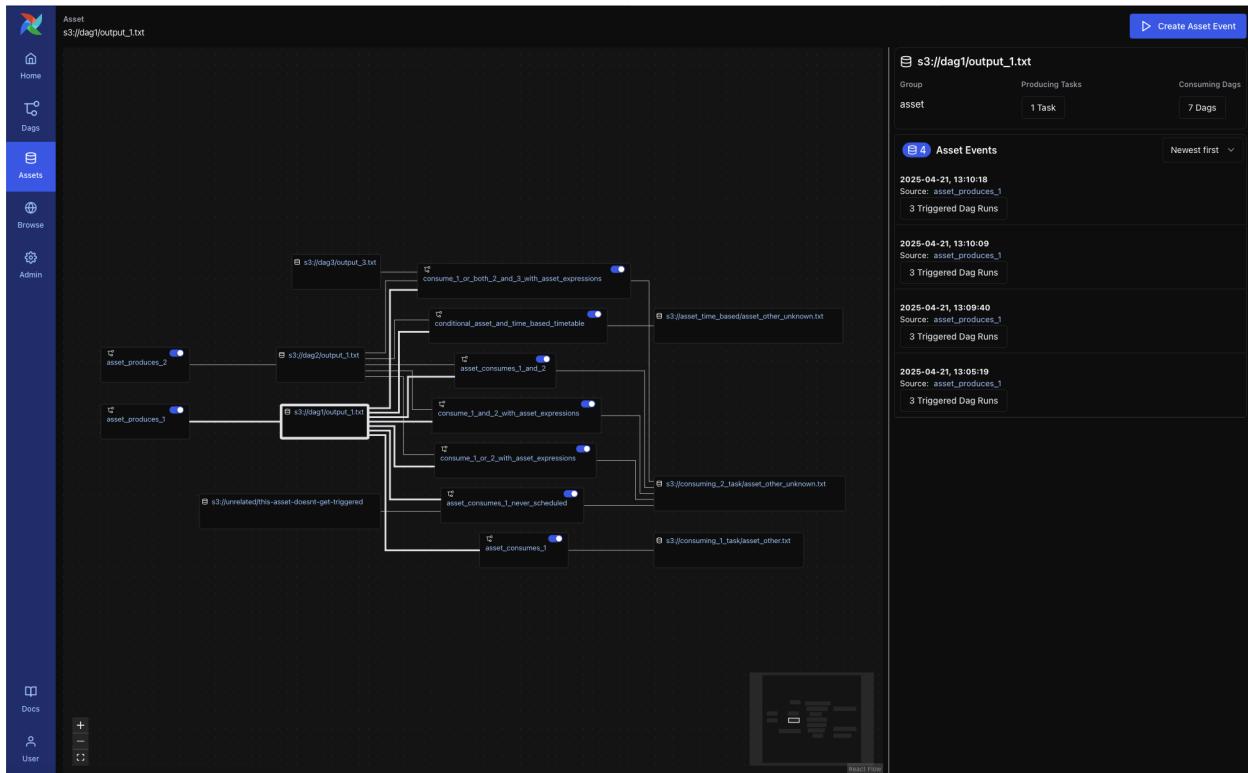
- consume_1_or_both_2_and_3_with_asset_expressions
- asset_consumes_1_never_scheduled
- consume_1_or_2_with_asset_expressions
- asset_consumes_1
- asset_consumes_1_and_2
- consume_1_and_2_with_asset_expressions
- conditional_asset_and_time_based_timetable

Clicking on the link takes you to the Asset Graph View.

Asset Graph View

The Asset Graph View shows the asset in context, including upstream producers and downstream consumers. You can use this view to:

- Understand asset lineage and the DAGs involved
- Trigger asset events manually
- View recent asset events and the DAG runs they triggered



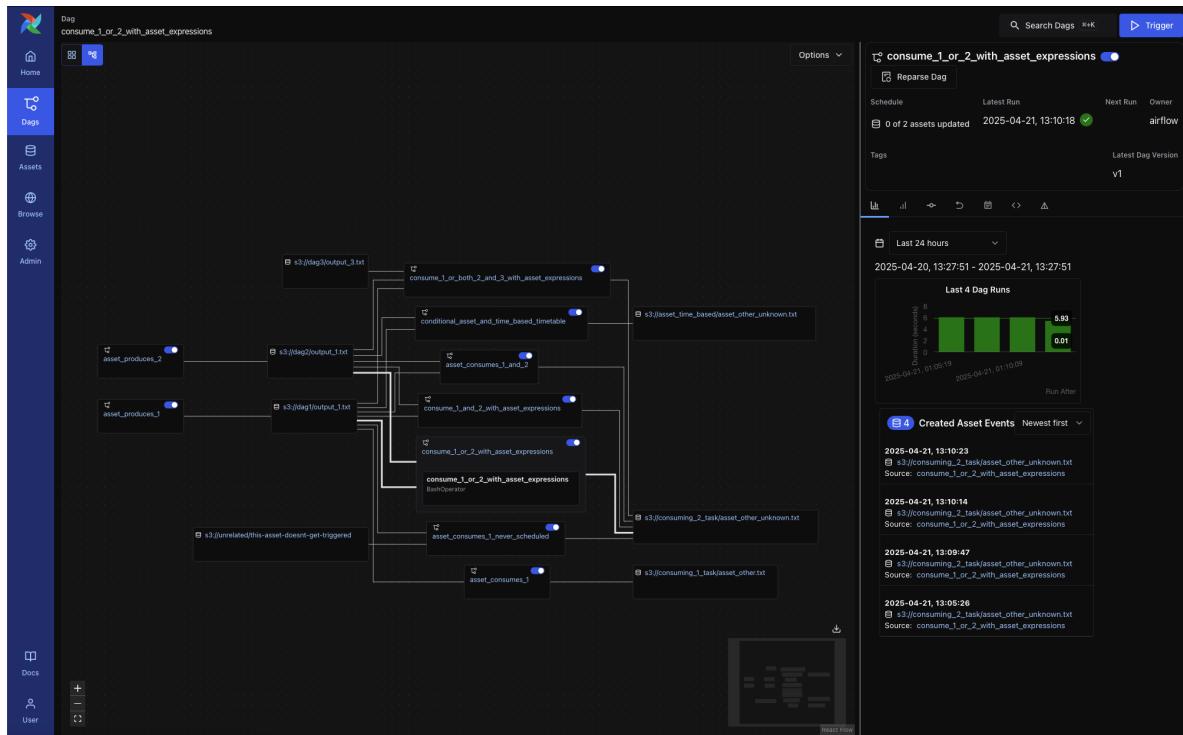
Graph Overlays in DAG View

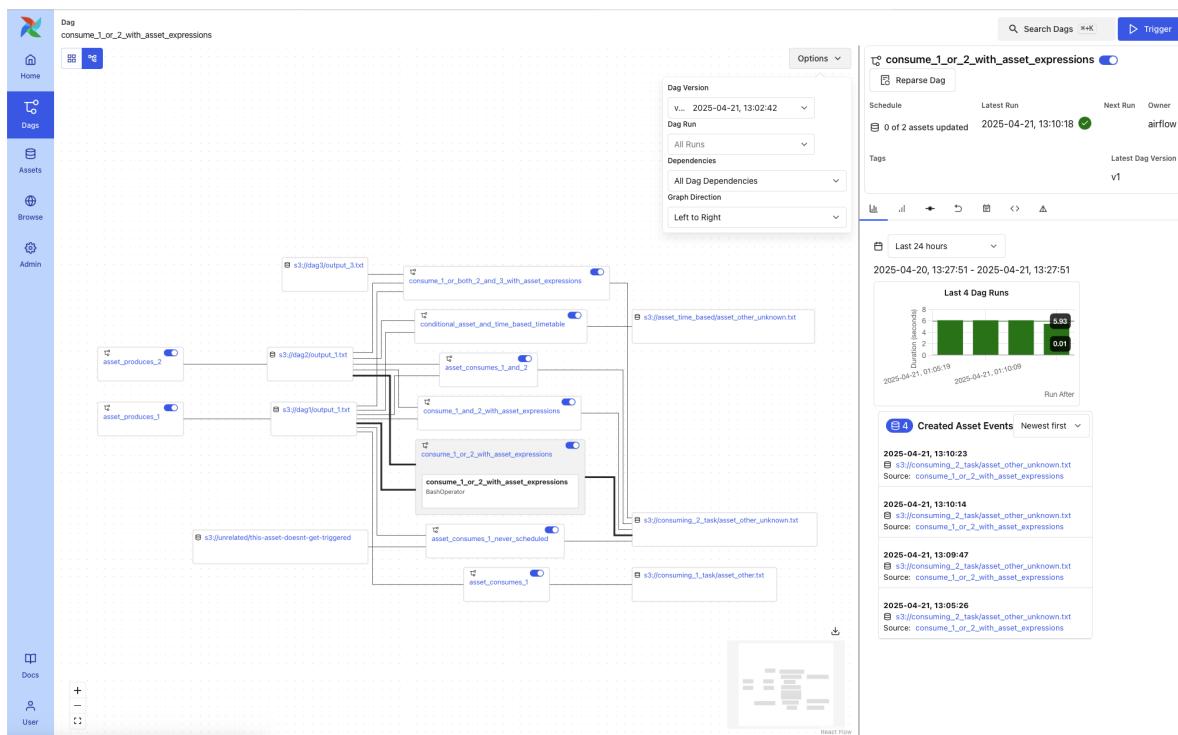
When a DAG contains asset-producing or asset-consuming tasks, you can enable asset overlays on the DAG Graph view. Toggle the switches next to each asset to:

- See how assets flow between DAGs
- Inspect asset-triggered dependencies

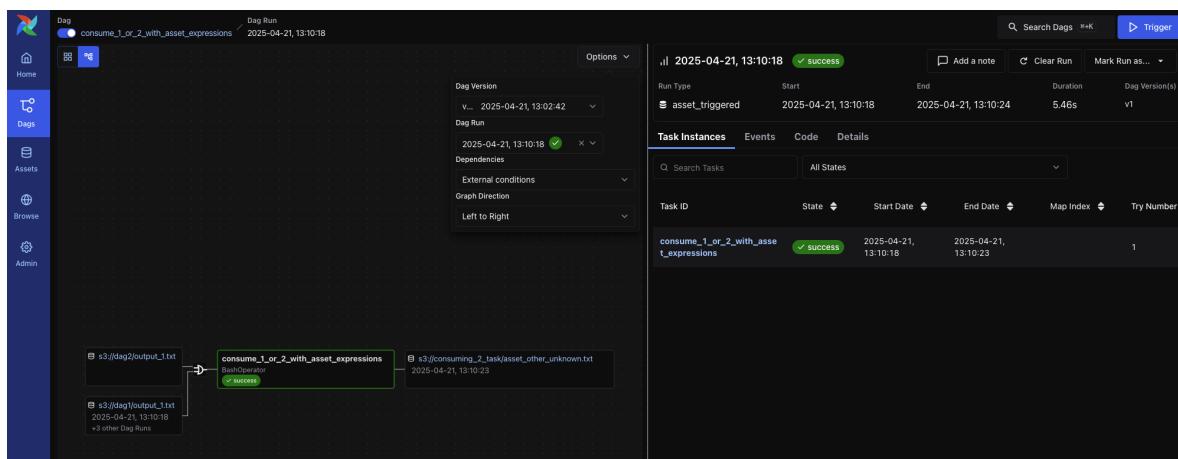
Two graph modes are available:

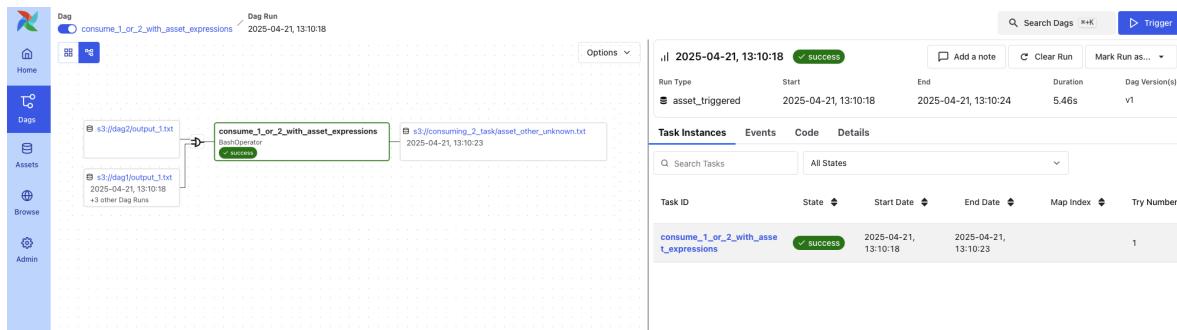
- **All DAG Dependencies:** Shows all DAG-to-DAG and task-level connections





- **External Conditions:** Shows only DAGs triggered via asset events





3.6.10 Admin Views

The **Admin** tab provides system-level tools for configuring and extending Airflow. These views are primarily intended for administrators and platform operators responsible for deployment, integration, and performance tuning.

Key pages include:

- **Variables** – Store key-value pairs accessible from DAGs. Variables can be used to manage environment-specific parameters or secrets.
- **Connections** – Define connection URIs to external systems such as databases, cloud services, or APIs. These are consumed by Airflow operators and hooks.
- **Pools** – Control resource allocation by limiting the number of concurrently running tasks assigned to a named pool. Useful for managing contention or quota-constrained systems.
- **Providers** – View installed provider packages (e.g., `apache-airflow-providers-google`), including available hooks, sensors, and operators. This is helpful for verifying provider versions or troubleshooting import errors.
- **Plugins** – Inspect registered Airflow plugins that extend the platform via custom operators, macros, or UI elements.
- **Config** – View the full effective Airflow configuration as parsed from `airflow.cfg`, environment variables, or overridden defaults. This can help debug issues related to scheduler behavior, secrets backends, and more.

Note

The Admin tab is only visible to users with appropriate RBAC permissions.

| Key | Value | Description | Is Encrypted |
|--------------------|---------------|-------------|--------------|
| snowflake_password | *** | | true |
| postgres_env | prod | | true |
| pipedrive_env | pipedrive | | true |
| environment | prod | | true |
| airtable_base_key | appzdasdasdas | | true |
| airtable_api_key | *** | | true |

| Connection Id | Connection Type | Description | Host | Port | |
|-----------------------------|---------------------|-------------------------------------|------|------|--|
| airflow_db | mysql | mysql | | | |
| athena_default | athena | | | | |
| aws_default | aws | | | | |
| azure_batch_default | azure_batch | | | | |
| azure_cosmos_default | azure_cosmos | | | | |
| azure_data_explorer_default | azure_data_explorer | https://<CLUSTER>.kusto.windows.net | | | |
| azure_data_lake_default | azure_data_lake | | | | |
| azure_default | azure | | | | |
| cassandra_default | cassandra | cassandra | | 9042 | |
| databricks_default | databricks | localhost | | | |
| dingding_default | http | | | | |
| drill_default | drill | localhost | | 8047 | |
| druid_broker_default | druid | druid-broker | | 8082 | |
| druid_ingest_default | druid | druid-overlord | | 8081 | |
| elasticsearch_default | elasticsearch | localhost | | 9200 | |
| emr_default | emr | | | | |
| facebook_default | facebook_social | | | | |

The screenshot shows the Apache Airflow web interface. On the left is a sidebar with icons for Home, Dags, Assets, Browse, Admin, Docs, and User. The main area has a search bar at the top. Below it is a table of connections. A modal window titled 'Add Connection' is open in the center. It has a 'Connection ID' input field containing 'airflow_db', a 'Connection Type' dropdown set to 'postgres', and a note below it: 'Connection type missing? Make sure you have installed the corresponding Airflow Providers Package.' The 'Standard Fields' section contains fields for Description, Host, Login, Password, Port, and Database. At the bottom of the modal is a 'Save' button. The background table lists various connections like 'airflow_db', 'athena_default', etc., with columns for Connection ID, Type, Host, Port, and edit/delete icons.

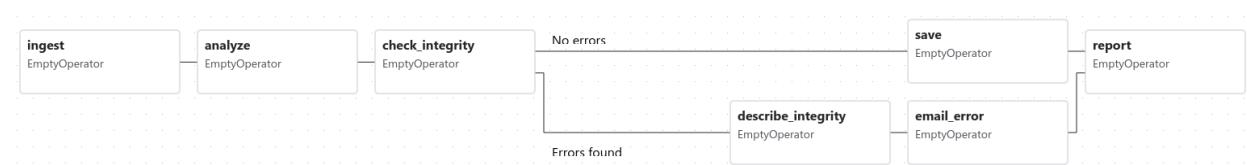
3.7 Core Concepts

Here you can find detailed documentation about each one of the core concepts of Apache Airflow® and how to use them, as well as a high-level *architectural overview*.

Architecture

3.7.1 Architecture Overview

Airflow is a platform that lets you build and run *workflows*. A workflow is represented as a *DAG* (a Directed Acyclic Graph), and contains individual pieces of work called *Tasks*, arranged with dependencies and data flows taken into account.



A DAG specifies the dependencies between tasks, which defines the order in which to execute the tasks. Tasks describe what to do, be it fetching data, running analysis, triggering other systems, or more.

Airflow itself is agnostic to what you're running - it will happily orchestrate and run anything, either with high-level support from one of our providers, or directly as a command using the shell or Python *Operators*.

Airflow components

Airflow's architecture consists of multiple components. The following sections describe each component's function and whether they're required for a bare-minimum Airflow installation, or an optional component to achieve better Airflow extensibility, performance, and scalability.

Required components

A minimal Airflow installation consists of the following components:

- A *scheduler*, which handles both triggering scheduled workflows, and submitting *Tasks* to the executor to run. The *executor*, is a configuration property of the *scheduler*, not a separate component and runs within the scheduler process. There are several executors available out of the box, and you can also write your own.
- A *dag processor*, which parses DAG files and serializes them into the *metadata database*. More about processing DAG files can be found in *DAG File Processing*
- A *webserver*, which presents a handy user interface to inspect, trigger and debug the behaviour of dags and tasks.
- A folder of *DAG files*, which is read by the *scheduler* to figure out what tasks to run and when to run them.
- A *metadata database*, which Airflow components use to store state of workflows and tasks. Setting up a metadata database is described in *Set up a Database Backend* and is required for Airflow to work.

Optional components

Some Airflow components are optional and can enable better extensibility, scalability, and performance in your Airflow:

- Optional *worker*, which executes the tasks given to it by the scheduler. In the basic installation worker might be part of the scheduler not a separate component. It can be run as a long running process in the CeleryExecutor, or as a POD in the KubernetesExecutor.
- Optional *triggerer*, which executes deferred tasks in an asyncio event loop. In basic installation where deferred tasks are not used, a triggerer is not necessary. More about deferring tasks can be found in *Deferrable Operators & Triggers*.
- Optional folder of *plugins*. Plugins are a way to extend Airflow's functionality (similar to installed packages). Plugins are read by the *scheduler*, *dag processor*, *triggerer* and *webserver*. More about plugins can be found in *Plugins*.

Deploying Airflow components

All the components are Python applications that can be deployed using various deployment mechanisms.

They can have extra *installed packages* installed in their Python environment. This is useful for example to install custom operators or sensors or extend Airflow functionality with custom plugins.

While Airflow can be run in a single machine and with simple installation where only *scheduler* and *webserver* are deployed, Airflow is designed to be scalable and secure, and is able to run in a distributed environment - where various components can run on different machines, with different security perimeters and can be scaled by running multiple instances of the components above.

The separation of components also allow for increased security, by isolating the components from each other and by allowing to perform different tasks. For example separating *dag processor* from *scheduler* allows to make sure that the *scheduler* does not have access to the *DAG files* and cannot execute code provided by *DAG author*.

Also while single person can run and manage Airflow installation, Airflow Deployment in more complex setup can involve various roles of users that can interact with different parts of the system, which is an important aspect of secure Airflow deployment. The roles are described in detail in the [Airflow Security Model](#) and generally speaking include:

- Deployment Manager - a person that installs and configures Airflow and manages the deployment
- DAG author - a person that writes dags and submits them to Airflow
- Operations User - a person that triggers dags and tasks and monitors their execution

Architecture Diagrams

The diagrams below show different ways to deploy Airflow - gradually from the simple “one machine” and single person deployment, to a more complex deployment with separate components, separate user roles and finally with more isolated security perimeters.

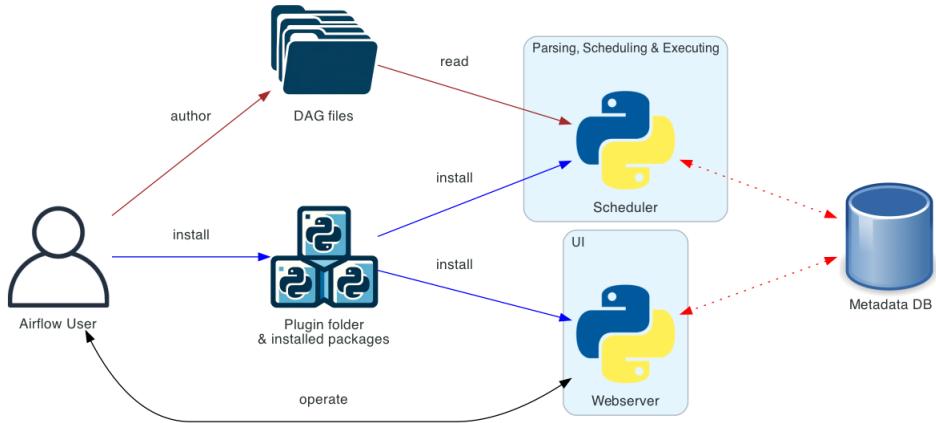
The meaning of the different connection types in the diagrams below is as follows:

- **brown solid lines** represent *DAG files* submission and synchronization
- **blue solid lines** represent deploying and accessing *installed packages* and *plugins*
- **black dashed lines** represent control flow of workers by the *scheduler* (via executor)
- **black solid lines** represent accessing the UI to manage execution of the workflows
- **red dashed lines** represent accessing the *metadata database* by all components

Basic Airflow deployment

This is the simplest deployment of Airflow, usually operated and managed on a single machine. Such a deployment usually uses the LocalExecutor, where the *scheduler* and the *workers* are in the same Python process and the *DAG files* are read directly from the local filesystem by the *scheduler*. The *webserver* runs on the same machine as the *scheduler*. There is no *triggerer* component, which means that task deferral is not possible.

Such an installation typically does not separate user roles - deployment, configuration, operation, authoring and maintenance are all done by the same person and there are no security perimeters between the components.



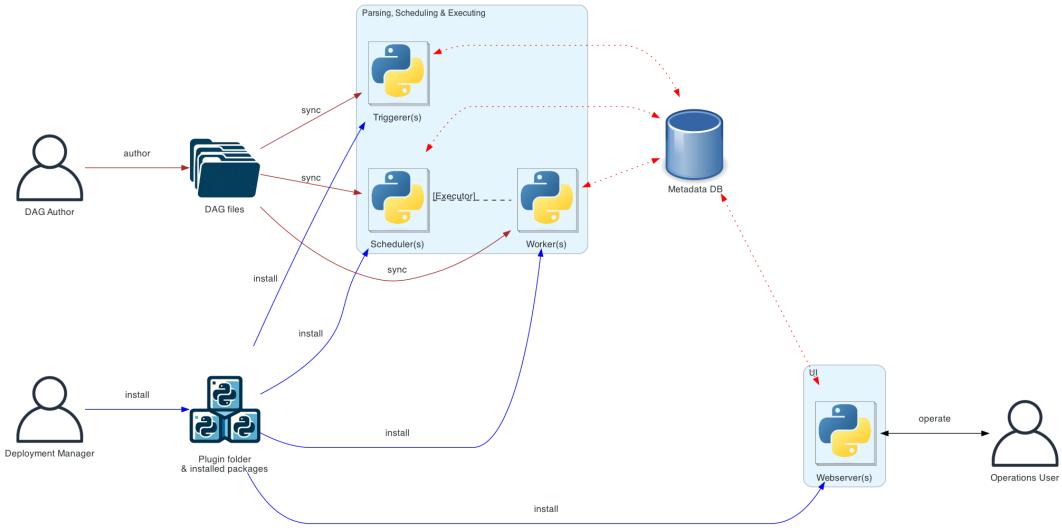
If you want to run Airflow on a single machine in a simple single-machine setup, you can skip the more complex diagrams below and go straight to the *Workloads* section.

Distributed Airflow architecture

This is the architecture of Airflow where components of Airflow are distributed among multiple machines and where various roles of users are introduced - **Deployment Manager**, **DAG author**, **Operations User**. You can read more about those various roles in the [Airflow Security Model](#).

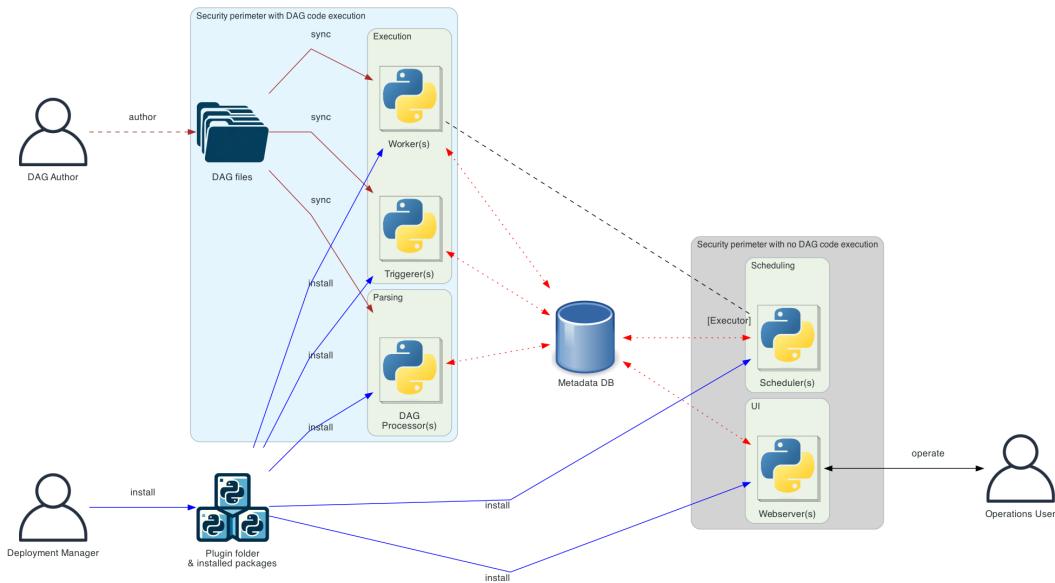
In the case of a distributed deployment, it is important to consider the security aspects of the components. The **webserver** does not have access to the **DAG files** directly. The code in the **Code tab** of the **UI** is read from the **metadata database**. The **webserver** cannot execute any code submitted by the **DAG author**. It can only execute code that is installed as an *installed package* or *plugin* by the **Deployment Manager**. The **Operations User** only has access to the **UI** and can only trigger dags and tasks, but cannot author dags.

The **DAG files** need to be synchronized between all the components that use them - *scheduler*, *triggerer* and *workers*. The **DAG files** can be synchronized by various mechanisms - typical ways how dags can be synchronized are described in `helm-chart:manage-dag-files` of our Helm Chart documentation. Helm chart is one of the ways how to deploy Airflow in K8S cluster.



Separate DAG processing architecture

In a more complex installation where security and isolation are important, you'll also see the standalone *dag processor* component that allows to separate *scheduler* from accessing *DAG files*. This is suitable if the deployment focus is on isolation between parsed tasks. While Airflow does not yet support full multi-tenant features, it can be used to make sure that **DAG author** provided code is never executed in the context of the scheduler.



Note

When DAG file is changed there can be cases where the scheduler and the worker will see different versions of the DAG until both components catch up. You can avoid the issue by making sure dag is deactivated during deployment and reactivate once finished. If needed, the cadence of sync and scan of DAG folder can be configured. Please make sure you really know what you are doing if you change the configurations.

Workloads

A DAG runs through a series of *Tasks*, and there are three common types of task you will see:

- *Operators*, predefined tasks that you can string together quickly to build most parts of your dags.
- *Sensors*, a special subclass of Operators which are entirely about waiting for an external event to happen.
- A *TaskFlow*-decorated `@task`, which is a custom Python function packaged up as a Task.

Internally, these are all actually subclasses of Airflow's `BaseOperator`, and the concepts of Task and Operator are somewhat interchangeable, but it's useful to think of them as separate concepts - essentially, Operators and Sensors are *templates*, and when you call one in a DAG file, you're making a Task.

Control Flow

Dags are designed to be run many times, and multiple runs of them can happen in parallel. Dags are parameterized, always including an interval they are “running for” (the *data interval*), but with other optional parameters as well.

Tasks have dependencies declared on each other. You'll see this in a DAG either using the `>>` and `<<` operators:

```
first_task >> [second_task, third_task]
fourth_task << third_task
```

Or, with the `set_upstream` and `set_downstream` methods:

```
first_task.set_downstream([second_task, third_task])
fourth_task.set_upstream(third_task)
```

These dependencies are what make up the “edges” of the graph, and how Airflow works out which order to run your tasks in. By default, a task will wait for all of its upstream tasks to succeed before it runs, but this can be customized using features like *Branching*, *LatestOnly*, and *Trigger Rules*.

To pass data between tasks you have three options:

- *XComs* (“Cross-communications”), a system where you can have tasks push and pull small bits of metadata.
- Uploading and downloading large files from a storage service (either one you run, or part of a public cloud)
- TaskFlow API automatically passes data between tasks via implicit *XComs*

Airflow sends out Tasks to run on Workers as space becomes available, so there’s no guarantee all the tasks in your DAG will run on the same worker or the same machine.

As you build out your dags, they are likely to get very complex, so Airflow provides several mechanisms for making this more sustainable, example *TaskGroups* let you visually group tasks in the UI.

There are also features for letting you easily pre-configure access to a central resource, like a datastore, in the form of *Connections & Hooks*, and for limiting concurrency, via *Pools*.

User interface

Airflow comes with a user interface that lets you see what dags and their tasks are doing, trigger runs of dags, view logs, and do some limited debugging and resolution of problems with your dags.

The screenshot shows the Apache Airflow web interface with the 'Dags' tab selected. On the left, there's a sidebar with icons for Home, Dags (which is selected), Assets, Browse, Admin, Docs, and User. The main content area is titled 'Dags' and shows a search bar with 'Search Dags' and an 'Advanced Search' button. Below the search bar are filters for 'All', 'Failed', 'Running', 'Success', and 'All' dropdowns, along with a 'Filter by tag' dropdown. A 'Sort by Latest Run Start Date...' dropdown is also present. The main list shows 81 Dags. Each dag is represented by a card with the dag ID, owner, schedule type, latest run date, next run date, and a green bar chart indicating task status. The dags listed are 'example_trigger_target_dag', 'example_trigger_controller_dag', 'example_xcom_args_with_operators', and 'example_xcom'.

It's generally the best way to see the status of your Airflow installation as a whole, as well as diving into individual dags to see their layout, the status of each task, and the logs from each task.

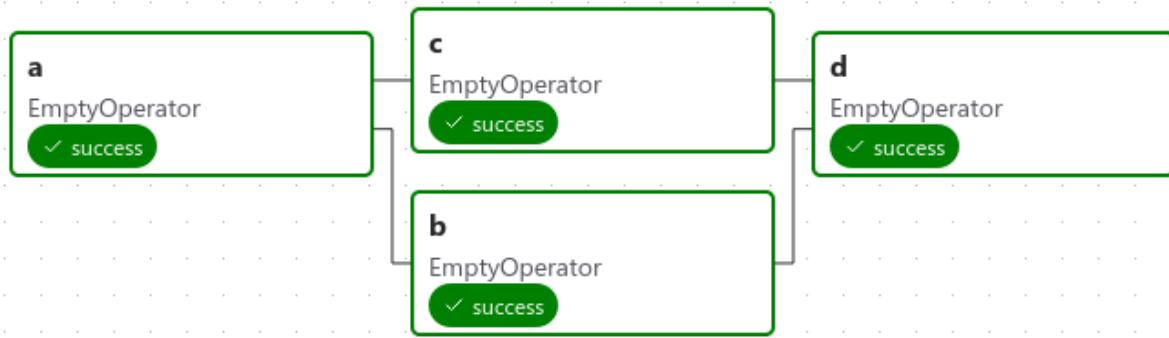
Workloads

3.7.2 Dags

A DAG is a model that encapsulates everything needed to execute a workflow. Some DAG attributes include the following:

- **Schedule:** When the workflow should run.
- **Tasks:** *tasks* are discrete units of work that are run on workers.
- **Task Dependencies:** The order and conditions under which *tasks* execute.
- **Callbacks:** Actions to take when the entire workflow completes.
- **Additional Parameters:** And many other operational details.

Here's a basic example DAG:



It defines four Tasks - A, B, C, and D - and dictates the order in which they have to run, and which tasks depend on what others. It will also say how often to run the DAG - maybe “every 5 minutes starting tomorrow”, or “every day since January 1st, 2020”.

The DAG itself doesn’t care about *what* is happening inside the tasks; it is merely concerned with *how* to execute them - the order to run them in, how many times to retry them, if they have timeouts, and so on.

i Note

The term “DAG” comes from the mathematical concept “directed acyclic graph”, but the meaning in Airflow has evolved well beyond just the literal data structure associated with the mathematical DAG concept.

Declaring a DAG

There are three ways to declare a DAG - either you can use `with` statement (context manager), which will add anything inside it to the DAG implicitly:

```

import datetime

from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

with DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
):
    EmptyOperator(task_id="task")

```

Or, you can use a standard constructor, passing the DAG into any operators you use:

```

import datetime

from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

my_dag = DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
)

```

(continues on next page)

(continued from previous page)

```
)  
EmptyOperator(task_id="task", dag=my_dag)
```

Or, you can use the `@dag` decorator to *turn a function into a DAG generator*:

```
import datetime  
  
from airflow.sdk import dag  
from airflow.providers.standard.operators.empty import EmptyOperator  
  
@dag(start_date=datetime.datetime(2021, 1, 1), schedule="@daily")  
def generate_dag():  
    EmptyOperator(task_id="task")  
  
generate_dag()
```

Dags are nothing without *Tasks* to run, and those will usually come in the form of either *Operators*, *Sensors* or *TaskFlow*.

Task Dependencies

A Task/Operator does not usually live alone; it has dependencies on other tasks (those *upstream* of it), and other tasks depend on it (those *downstream* of it). Declaring these dependencies between tasks is what makes up the DAG structure (the *edges* of the *directed acyclic graph*).

There are two main ways to declare individual task dependencies. The recommended one is to use the `>>` and `<<` operators:

```
first_task >> [second_task, third_task]  
third_task << fourth_task
```

Or, you can also use the more explicit `set_upstream` and `set_downstream` methods:

```
first_task.set_downstream([second_task, third_task])  
third_task.set_upstream(fourth_task)
```

There are also shortcuts to declaring more complex dependencies. If you want to make a list of tasks depend on another list of tasks, you can't use either of the approaches above, so you need to use `cross_downstream`:

```
from airflow.sdk import cross_downstream  
  
# Replaces  
# [op1, op2] >> op3  
# [op1, op2] >> op4  
cross_downstream([op1, op2], [op3, op4])
```

And if you want to chain together dependencies, you can use `chain`:

```
from airflow.sdk import chain  
  
# Replaces op1 >> op2 >> op3 >> op4  
chain(op1, op2, op3, op4)
```

(continues on next page)

(continued from previous page)

```
# You can also do it dynamically
chain(*[EmptyOperator(task_id=f"op{i}") for i in range(1, 6)])
```

Chain can also do *pairwise* dependencies for lists the same size (this is different from the *cross dependencies* created by `cross_downstream!`):

```
from airflow.sdk import chain

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

Loading dags

Airflow loads dags from Python source files in dag bundles. It will take each file, execute it, and then load any DAG objects from that file.

This means you can define multiple dags per Python file, or even spread one very complex DAG across multiple Python files using imports.

Note, though, that when Airflow comes to load dags from a Python file, it will only pull any objects at the *top level* that are a DAG instance. For example, take this DAG file:

```
dag_1 = DAG('this_dag_will_be_discovered')

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

my_function()
```

While both DAG constructors get called when the file is accessed, only `dag_1` is at the top level (in the `globals()`), and so only it is added to Airflow. `dag_2` is not loaded.

Note

When searching for dags inside the dag bundle, Airflow only considers Python files that contain the strings `airflow` and `dag` (case-insensitively) as an optimization.

To consider all Python files instead, disable the `DAG_DISCOVERY_SAFE_MODE` configuration flag.

You can also provide an `.airflowignore` file inside your dag bundle, or any of its subfolders, which describes patterns of files for the loader to ignore. It covers the directory it's in plus all subfolders underneath it. See `.airflowignore` below for details of the file syntax.

In the case where the `.airflowignore` does not meet your needs and you want a more flexible way to control if a python file needs to be parsed by airflow, you can plug your callable by setting `might_contain_dag_callable` in the config file. Note, this callable will replace the default Airflow heuristic, i.e. checking if the strings `airflow` and `dag` (case-insensitively) are present in the python file.

```
def might_contain_dag(file_path: str, zip_file: zipfile.ZipFile | None = None) -> bool:
    # Your logic to check if there are dags defined in the file_path
    # Return True if the file_path needs to be parsed, otherwise False
```

Running dags

Dags will run in one of two ways:

- When they are *triggered* either manually or via the API
- On a defined *schedule*, which is defined as part of the DAG

Dags do not *require* a schedule, but it's very common to define one. You define it via the `schedule` argument, like this:

```
with DAG("my_daily_dag", schedule="@daily"):  
    ...
```

There are various valid values for the `schedule` argument:

```
with DAG("my_daily_dag", schedule="0 0 * * *"):  
    ...  
  
with DAG("my_one_time_dag", schedule="@once"):  
    ...  
  
with DAG("my_continuous_dag", schedule="@continuous"):  
    ...
```

💡 Tip

For more information different types of scheduling, see *Authoring and Scheduling*.

Every time you run a DAG, you are creating a new instance of that DAG which Airflow calls a *DAG Run*. DAG Runs can run in parallel for the same DAG, and each has a defined data interval, which identifies the period of data the tasks should operate on.

As an example of why this is useful, consider writing a DAG that processes a daily set of experimental data. It's been rewritten, and you want to run it on the previous 3 months of data—no problem, since Airflow can *backfill* the DAG and run copies of it for every day in those previous 3 months, all at once.

Those DAG Runs will all have been started on the same actual day, but each DAG run will have one data interval covering a single day in that 3 month period, and that data interval is all the tasks, operators and sensors inside the DAG look at when they run.

In much the same way a DAG instantiates into a DAG Run every time it's run, Tasks specified inside a DAG are also instantiated into *Task Instances* along with it.

A DAG run will have a start date when it starts, and end date when it ends. This period describes the time when the DAG actually ‘ran.’ Aside from the DAG run’s start and end date, there is another date called *logical date* (formally known as execution date), which describes the intended time a DAG run is scheduled or triggered. The reason why this is called *logical* is because of the abstract nature of it having multiple meanings, depending on the context of the DAG run itself.

For example, if a DAG run is manually triggered by the user, its logical date would be the date and time of which the DAG run was triggered, and the value should be equal to DAG run’s start date. However, when the DAG is being automatically scheduled, with certain schedule interval put in place, the logical date is going to indicate the time at which it marks the start of the data interval, where the DAG run’s start date would then be the logical date + scheduled interval.

💡 Tip

For more information on `logical_date`, see [Data Interval](#) and [What does execution_date mean?](#).

DAG Assignment

Note that every single Operator/Task must be assigned to a DAG in order to run. Airflow has several ways of calculating the DAG without you passing it explicitly:

- If you declare your Operator inside a `with DAG` block
- If you declare your Operator inside a `@dag` decorator
- If you put your Operator upstream or downstream of an Operator that has a DAG

Otherwise, you must pass it into each Operator with `dag=`.

Default Arguments

Often, many Operators inside a DAG need the same set of default arguments (such as their `retries`). Rather than having to specify this individually for every Operator, you can instead pass `default_args` to the DAG when you create it, and it will auto-apply them to any operator tied to it:

```
import pendulum

with DAG(
    dag_id="my_dag",
    start_date=pendulum.datetime(2016, 1, 1),
    schedule="@daily",
    default_args={"retries": 2},
):
    op = BashOperator(task_id="hello_world", bash_command="Hello World!")
    print(op.retries) # 2
```

The DAG decorator

Added in version 2.0.

As well as the more traditional ways of declaring a single DAG using a context manager or the `DAG()` constructor, you can also decorate a function with `@dag` to turn it into a DAG generator function:

`airflow/example_dags/example_dag_decorator.py`

```
@dag(
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
)
def example_dag_decorator(url: str = "http://httpbin.org/get"):
    """
    DAG to get IP address and echo it via BashOperator.

    :param url: URL to get IP address from. Defaults to "http://httpbin.org/get".
    """
    get_ip = GetRequestOperator(task_id="get_ip", url=url)
```

(continues on next page)

(continued from previous page)

```

@task(multiple_outputs=True)
def prepare_command(raw_json: dict[str, Any]) -> dict[str, str]:
    external_ip = raw_json["origin"]
    return {
        "command": f"echo 'Seems like today your server executing Airflow is connected from IP {external_ip}'",
    }

    command_info = prepare_command(get_ip.output)

BashOperator(task_id="echo_ip_info", bash_command=command_info["command"])

example_dag = example_dag_decorator()

```

As well as being a new way of making dags cleanly, the decorator also sets up any parameters you have in your function as DAG parameters, letting you *set those parameters when triggering the DAG*. You can then access the parameters from Python code, or from `context.params` inside a *Jinja template*.

Note

Airflow will only load dags that *appear in the top level* of a DAG file. This means you cannot just declare a function with `@dag` - you must also call it at least once in your DAG file and assign it to a top-level object, as you can see in the example above.

Control Flow

By default, a DAG will only run a Task when all the Tasks it depends on are successful. There are several ways of modifying this, however:

- *Branching* - select which Task to move onto based on a condition
- *Trigger Rules* - set the conditions under which a DAG will run a task
- *Setup and Teardown* - define setup and teardown relationships
- *Latest Only* - a special form of branching that only runs on dags running against the present
- *Depends On Past* - tasks can depend on themselves *from a previous run*

Branching

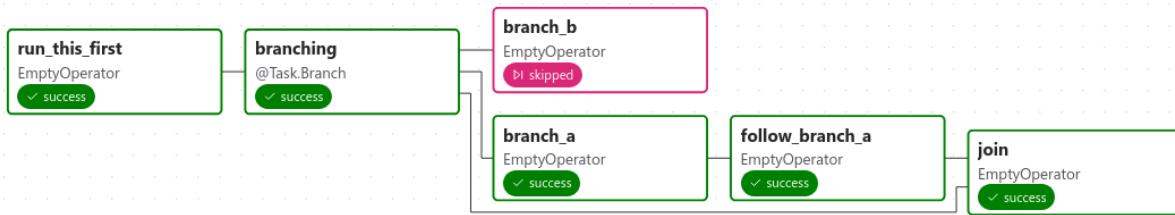
You can make use of branching in order to tell the DAG *not* to run all dependent tasks, but instead to pick and choose one or more paths to go down. This is where the `@task.branch` decorator come in.

The `@task.branch` decorator is much like `@task`, except that it expects the decorated function to return an ID to a task (or a list of IDs). The specified task is followed, while all other paths are skipped. It can also return `None` to skip all downstream tasks.

The `task_id` returned by the Python function has to reference a task directly downstream from the `@task.branch` decorated task.

Note

When a Task is downstream of both the branching operator *and* downstream of one or more of the selected tasks, it will not be skipped:



The paths of the branching task are `branch_a`, `join` and `branch_b`. Since `join` is a downstream task of `branch_a`, it will still be run, even though it was not returned as part of the branch decision.

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

```

@task.branch(task_id="branch_task")
def branch_func(ti=None):
    xcom_value = int(ti.xcom_pull(task_ids="start_task"))
    if xcom_value >= 5:
        return "continue_task"
    elif xcom_value >= 3:
        return "stop_task"
    else:
        return None

start_op = BashOperator(
    task_id="start_task",
    bash_command="echo 5",
    do_xcom_push=True,
    dag=dag,
)
branch_op = branch_func()

continue_op = EmptyOperator(task_id="continue_task", dag=dag)
stop_op = EmptyOperator(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 `@task.branch` decorator but expects you to provide an implementation of the method `choose_branch`.

Note

The `@task.branch` decorator is recommended over directly instantiating `BranchPythonOperator` in a DAG. The latter should generally only be subclassed to implement a custom operator.

As with the callable for `@task.branch`, this method can return the ID of a downstream task, or a list of task IDs, which will be run, and all others will be skipped. It can also return `None` to skip all downstream task:

```
class MyBranchOperator(BaseBranchOperator):
    def choose_branch(self, context):
        """
        Run an extra branch on the first day of the month
        """
        if context['data_interval_start'].day == 1:
            return ['daily_task_id', 'monthly_task_id']
        elif context['data_interval_start'].day == 2:
            return 'daily_task_id'
        else:
            return None
```

Similar like `@task.branch` decorator for regular Python code there are also branch decorators which use a virtual environment called `@task.branch_virtualenv` or external python called `@task.branch_external_python`.

Latest Only

Airflow's DAG Runs are often run for a date that is not the same as the current date - for example, running one copy of a DAG for every day in the last month to backfill some data.

There are situations, though, where you *don't* want to let some (or all) parts of a DAG run for a previous date; in this case, you can use the `LatestOnlyOperator`.

This special Operator skips all tasks downstream of itself if you are not on the “latest” DAG run (if the wall-clock time right now is between its `execution_time` and the next scheduled `execution_time`, and it was not an externally-triggered run).

Here's an example:

`airflow/example_dags/example_latest_only_with_trigger.py`

```
import datetime
import pendulum

from airflow.providers.standard.operators.empty import EmptyOperator
from airflow.providers.standard.operators.latest_only import LatestOnlyOperator
from airflow.sdk import DAG
from airflow.utils.trigger_rule import TriggerRule

with DAG(
    dag_id="latest_only_with_trigger",
    schedule=datetime.timedelta(hours=4),
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example3"],
) as dag:
    latest_only = LatestOnlyOperator(task_id="latest_only")
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task3 = EmptyOperator(task_id="task3")
    task4 = EmptyOperator(task_id="task4", trigger_rule=TriggerRule.ALL_DONE)
```

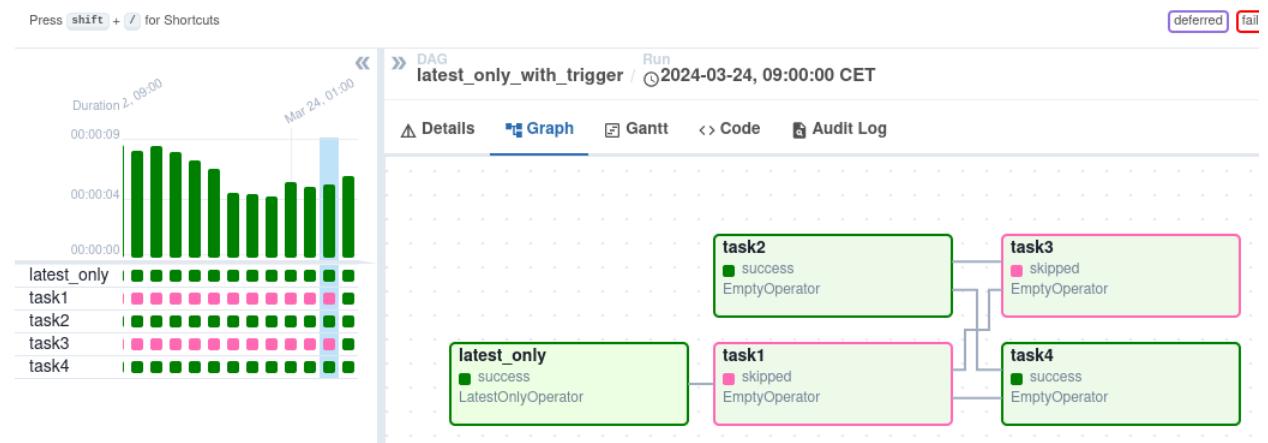
(continues on next page)

(continued from previous page)

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

In the case of this DAG:

- `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`.



Depends On Past

You can also say a task can only run if the *previous* run of the task in the previous DAG Run succeeded. To use this, you just need to set the `depends_on_past` argument on your Task to True.

Note that if you are running the DAG at the very start of its life—specifically, its first ever *automated* run—then the Task will still run, as there is no previous run to depend on.

Trigger Rules

By default, Airflow will wait for all upstream (direct parents) tasks for a task to be *successful* before it runs that task.

However, this is just the default behaviour, and you can control it using the `trigger_rule` argument to a Task. The options for `trigger_rule` are:

- `all_success` (default): All upstream tasks have succeeded
- `all_failed`: All upstream tasks are in a `failed` or `upstream_failed` state
- `all_done`: All upstream tasks are done with their execution
- `all_skipped`: All upstream tasks are in a `skipped` state
- `one_failed`: At least one upstream task has failed (does not wait for all upstream tasks to be done)
- `one_success`: At least one upstream task has succeeded (does not wait for all upstream tasks to be done)
- `one_done`: At least one upstream task succeeded or failed

- `none_failed`: All upstream tasks have not failed or `upstream_failed` - that is, all upstream tasks have succeeded or been skipped
- `none_failed_min_one_success`: All upstream tasks have not failed or `upstream_failed`, and at least one upstream task has succeeded.
- `none_skipped`: No upstream task is in a `skipped` state - that is, all upstream tasks are in a `success`, `failed`, or `upstream_failed` state
- `always`: No dependencies at all, run this task at any time

You can also combine this with the *Depends On Past* functionality if you wish.

Note

It's important to be aware of the interaction between trigger rules and skipped tasks, especially tasks that are skipped as part of a branching operation. *You almost never want to use `all_success` or `all_failed` downstream of a branching operation.*

Skipped tasks will cascade through trigger rules `all_success` and `all_failed`, and cause them to skip as well. Consider the following DAG:

```
# dags/branch_without_trigger.py
import pendulum

from airflow.sdk import task
from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

dag = DAG(
    dag_id="branch_without_trigger",
    schedule="@once",
    start_date=pendulum.datetime(2019, 2, 28, tz="UTC"),
)

run_this_first = EmptyOperator(task_id="run_this_first", dag=dag)

@task.branch(task_id="branching")
def do_branching():
    return "branch_a"

branching = do_branching()

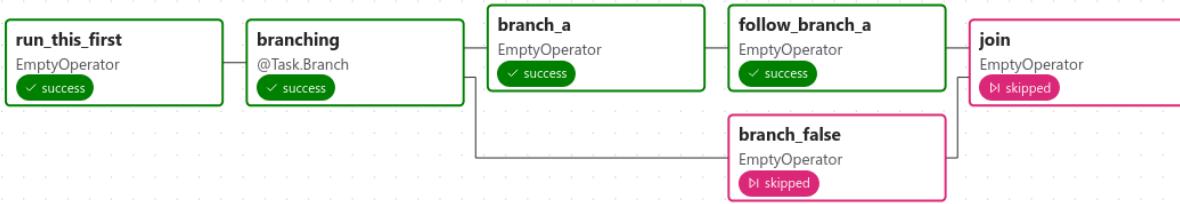
branch_a = EmptyOperator(task_id="branch_a", dag=dag)
follow_branch_a = EmptyOperator(task_id="follow_branch_a", dag=dag)

branch_false = EmptyOperator(task_id="branch_false", dag=dag)

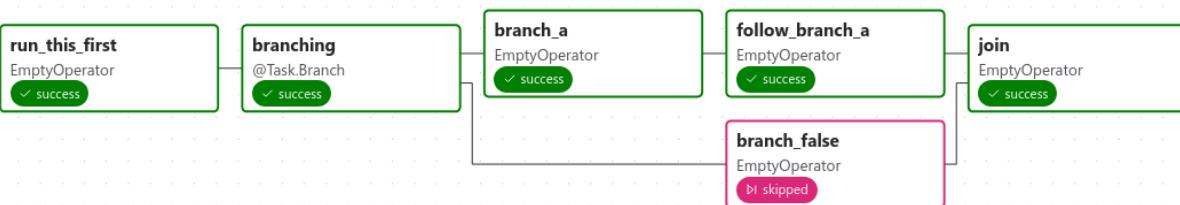
join = EmptyOperator(task_id="join", dag=dag)

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

`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 the skip caused by the branching operation cascades down to skip a task marked as `all_success`.



By setting `trigger_rule` to `none_failed_min_one_success` in the `join` task, we can instead get the intended behaviour:



Setup and teardown

In data workflows it's common to create a resource (such as a compute resource), use it to do some work, and then tear it down. Airflow provides setup and teardown tasks to support this need.

Please see main article *Setup and Teardown* for details on how to use this feature.

Dynamic dags

Since a DAG is defined by Python code, there is no need for it to be purely declarative; you are free to use loops, functions, and more to define your DAG.

For example, here is a DAG that uses a `for` loop to define some tasks:

```

with DAG("loop_example", ...):
    first = EmptyOperator(task_id="first")
    last = EmptyOperator(task_id="last")

    options = ["branch_a", "branch_b", "branch_c", "branch_d"]
    for option in options:
        t = EmptyOperator(task_id=option)
        first >> t >> last
  
```

In general, we advise you to try and keep the *topology* (the layout) of your DAG tasks relatively stable; dynamic dags are usually better used for dynamically loading configuration options or changing operator options.

DAG Visualization

If you want to see a visual representation of a DAG, you have two options:

- You can load up the Airflow UI, navigate to your DAG, and select “Graph”
- You can run `airflow dags show`, which renders it out as an image file

We generally recommend you use the Graph view, as it will also show you the state of all the *Task Instances* within any DAG Run you select.

Of course, as you develop out your dags they are going to get increasingly complex, so we provide a few ways to modify these DAG views to make them easier to understand.

TaskGroups

A 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.

Tasks in TaskGroups live on the same original DAG, and honor all the DAG settings and 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`:

```
from airflow.sdk import task_group

@task_group()
def group1():
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")

    task3 = EmptyOperator(task_id="task3")

    group1() >> task3
```

TaskGroup also supports `default_args` like DAG, it will overwrite the `default_args` in DAG level:

```
import datetime

from airflow.sdk import DAG
from airflow.sdk import task_group
from airflow.providers.standard.operators.bash import BashOperator
from airflow.providers.standard.operators.empty import EmptyOperator

with DAG(
    dag_id="dag1",
    start_date=datetime.datetime(2016, 1, 1),
    schedule="@daily",
    default_args={"retries": 1},
):

    @task_group(default_args={"retries": 3})
    def group1():
        """This docstring will become the tooltip for the TaskGroup."""
        task1 = EmptyOperator(task_id="task1")
        task2 = BashOperator(task_id="task2", bash_command="echo Hello World!", retries=2)
        print(task1.retries) # 3
        print(task2.retries) # 2
```

If you want to see a more advanced use of TaskGroup, you can look at the `example_task_group_decorator.py`

example DAG that comes with Airflow.

Note

By default, child tasks/TaskGroups have their IDs prefixed with the group_id of their parent TaskGroup. This helps to ensure uniqueness of group_id and task_id throughout the DAG.

To disable the prefixing, pass `prefix_group_id=False` when creating the TaskGroup, but note that you will now be responsible for ensuring every single task and group has a unique ID of its own.

Note

When using the `@task_group` decorator, the decorated-function's docstring will be used as the TaskGroups tooltip in the UI except when a `tooltip` value is explicitly supplied.

Edge Labels

As well as grouping tasks into groups, you can also label the *dependency edges* between different tasks in the Graph view - this can be especially useful for branching areas of your DAG, so you can label the conditions under which certain branches might run.

To add labels, you can use them directly inline with the `>>` and `<<` operators:

```
from airflow.sdk import Label

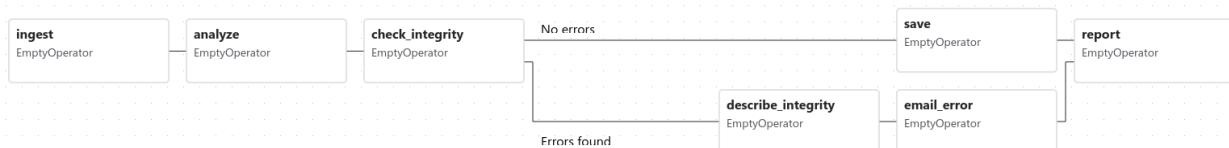
my_task >> Label("When empty") >> other_task
```

Or, you can pass a Label object to `set_upstream`/`set_downstream`:

```
from airflow.sdk import Label

my_task.set_downstream(other_task, Label("When empty"))
```

Here's an example DAG which illustrates labeling different branches:



`airflow/example_dags/example_branch_labels.py`

```
with DAG(
    "example_branch_labels",
    schedule="@daily",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
) as dag:
    ingest = EmptyOperator(task_id="ingest")
    analyse = EmptyOperator(task_id="analyze")
    check = EmptyOperator(task_id="check_integrity")
```

(continues on next page)

(continued from previous page)

```

describe = EmptyOperator(task_id="describe_integrity")
error = EmptyOperator(task_id="email_error")
save = EmptyOperator(task_id="save")
report = EmptyOperator(task_id="report")

ingest >> analyse >> check
check >> Label("No errors") >> save >> report
check >> Label("Errors found") >> describe >> error >> report

```

DAG & Task Documentation

It's possible to add documentation or notes to your dags & task objects that are visible in the web interface ("Graph" & "Tree" for dags, "Task Instance 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 |
| doc_yaml | yaml |
| doc_md | markdown |
| doc_rst | reStructuredText |

Please note that for dags, doc_md is the only attribute interpreted. For dags it can contain a string or the reference to a markdown file. Markdown files are recognized by str ending in .md. If a relative path is supplied it will be loaded from the path relative to which the Airflow Scheduler or DAG parser was started. If the markdown file does not exist, the passed filename will be used as text, no exception will be displayed. Note that the markdown file is loaded during DAG parsing, changes to the markdown content take one DAG parsing cycle to have changes be displayed.

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

```

"""
## My great DAG
"""

import pendulum

dag = DAG(
    "my_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule="@daily",
    catchup=False,
)
dag.doc_md = __doc__

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

```

Packaging dags

While simpler dags are usually only in a single Python file, it is not uncommon that more complex dags might be spread across multiple files and have dependencies that should be shipped with them (“vendored”).

You can either do this all inside of the dag bundle, with a standard filesystem layout, or you can package the DAG and all of its Python files up as a single zip file. For instance, you could ship two dags along with a dependency they need as a zip file with the following contents:

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

Note that packaged dags come with some caveats:

- They cannot be used if you have pickling enabled for serialization
- They cannot contain compiled libraries (e.g. `libz.so`), only pure Python
- They will be inserted into Python’s `sys.path` and importable by any other code in the Airflow process, so ensure the package names don’t clash with other packages already installed on your system.

In general, if you have a complex set of compiled dependencies and modules, you are likely better off using the Python `virtualenv` system and installing the necessary packages on your target systems with `pip`.

.airflowignore

An `.airflowignore` file specifies the directories or files in the dag bundle or `PLUGINS_FOLDER` that Airflow should intentionally ignore. Airflow supports two syntax flavors for patterns in the file, as specified by the `DAG_IGNORE_FILE_SYNTAX` configuration parameter (*added in Airflow 2.3*): `regexp` and `glob`.

Note

The default `DAG_IGNORE_FILE_SYNTAX` is `glob` in Airflow 3 or later (in previous versions it was `regexp`).

With the `glob` syntax (the default), the patterns work just like those in a `.gitignore` file:

- The `*` character will match any number of characters, except `/`
- The `?` character will match any single character, except `/`
- The range notation, e.g. `[a-zA-Z]`, can be used to match one of the characters in a range
- A pattern can be negated by prefixing with `!`. Patterns are evaluated in order so a negation can override a previously defined pattern in the same file or patterns defined in a parent directory.
- A double asterisk (`**`) can be used to match across directories. For example, `**/__pycache__/` will ignore `__pycache__` directories in each sub-directory to infinite depth.
- If there is a `/` at the beginning or middle (or both) of the pattern, then the pattern is relative to the directory level of the particular `.airflowignore` file itself. Otherwise the pattern may also match at any level below the `.airflowignore` level.

For the `regexp` pattern syntax, 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). Use the `#` character to indicate a comment; all characters on lines starting with `#` will be ignored.

The `.airflowignore` file should be put in your dag bundle. For example, you can prepare a `.airflowignore` file with the glob syntax

```
**/*project_a*
tenant_[0-9]*
```

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 bundle 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 your dag bundle and it would only be applicable for that subfolder.

DAG Dependencies

Added in Airflow 2.1

While dependencies between tasks in a DAG are explicitly defined through upstream and downstream relationships, dependencies between dags are a bit more complex. In general, there are two ways in which one DAG can depend on another:

- triggering - `TriggerDagRunOperator`
- waiting - `ExternalTaskSensor`

Additional difficulty is that one DAG could wait for or trigger several runs of the other DAG with different data intervals. The **Dag Dependencies** view Menu → Browse → DAG Dependencies helps visualize dependencies between dags. The dependencies are calculated by the scheduler during DAG serialization and the webserver uses them to build the dependency graph.

The dependency detector is configurable, so you can implement your own logic different than the defaults in `DependencyDetector`

DAG pausing, deactivation and deletion

The dags have several states when it comes to being “not running”. Dags can be paused, deactivated and finally all metadata for the DAG can be deleted.

Dag can be paused via UI when it is present in the `DAGS_FOLDER`, and scheduler stored it in the database, but the user chose to disable it via the UI. The “pause” and “unpause” actions are available via UI and API. Paused DAG is not scheduled by the Scheduler, but you can trigger them via UI for manual runs. In the UI, you can see paused dags (in Paused tab). The dags that are un-paused can be found in the Active tab. When a DAG is paused, any running tasks are allowed to complete and all downstream tasks are put in to a state of “Scheduled”. When the DAG is unpause, any “scheduled” tasks will begin running according to the DAG logic. Dags with no “scheduled” tasks will begin running according to their schedule.

Dags can be deactivated (do not confuse it with Active tag in the UI) by removing them from the `DAGS_FOLDER`. When scheduler parses the `DAGS_FOLDER` and misses the DAG that it had seen before and stored in the database it will set it as deactivated. The metadata and history of the DAG is kept for deactivated dags and when the dag is re-added to the `DAGS_FOLDER` it will be again activated and history will be visible. You cannot activate/deactivate DAG via UI or API, this can only be done by removing files from the `DAGS_FOLDER`. Once again - no data for historical runs of the DAG are lost when it is deactivated by the scheduler. Note that the Active tab in Airflow UI refers to dags that are not both Activated and Not paused so this might initially be a little confusing.

You can't see the deactivated dags in the UI - you can sometimes see the historical runs, but when you try to see the information about those you will see the error that the DAG is missing.

You can also delete the DAG metadata from the metadata database using UI or API, but it does not always result in disappearing of the DAG from the UI - which might be also initially a bit confusing. If the DAG is still in `DAGS_FOLDER`

when you delete the metadata, the DAG will re-appear as Scheduler will parse the folder, only historical runs information for the DAG will be removed.

This all means that if you want to actually delete a DAG and its all historical metadata, you need to do it in three steps:

- pause the DAG
- delete the historical metadata from the database, via UI or API
- delete the DAG file from the DAGS_FOLDER and wait until it becomes inactive

DAG Auto-pausing (Experimental)

Dags can be configured to be auto-paused as well. There is a Airflow configuration which allows for automatically disabling of a dag if it fails for N number of times consecutively.

- `max_consecutive_failed_dag_runs_per_dag`

we can also provide and override these configuration from DAG argument:

- `max_consecutive_failed_dag_runs`: Overrides `max_consecutive_failed_dag_runs_per_dag`.

3.7.3 DAG Runs

A DAG Run is an object representing an instantiation of the DAG in time. Any time the DAG is executed, a DAG Run is created and all tasks inside it are executed. The status of the DAG Run depends on the tasks states. Each DAG Run is run separately from one another, meaning that you can have many runs of a DAG at the same time.

DAG Run Status

A DAG Run status is determined when the execution of the DAG is finished. The execution of the DAG depends on its containing tasks and their dependencies. The status is assigned to the DAG Run when all of the tasks are in the one of the terminal states (i.e. if there is no possible transition to another state) like `success`, `failed` or `skipped`. The DAG Run is having the status assigned based on the so-called “leaf nodes” or simply “leaves”. Leaf nodes are the tasks with no children.

There are two possible terminal states for the DAG Run:

- `success` if all of the leaf nodes states are either `success` or `skipped`,
- `failed` if any of the leaf nodes state is either `failed` or `upstream_failed`.

Note

Be careful if some of your tasks have defined some specific *trigger rule*. These can lead to some unexpected behavior, e.g. if you have a leaf task with trigger rule “`all_done`”, it will be executed regardless of the states of the rest of the tasks and if it will succeed, then the whole DAG Run will also be marked as `success`, even if something failed in the middle.

Added in Airflow 2.7

Dags that have a currently running DAG run can be shown on the UI dashboard in the “Running” tab. Similarly, dags whose latest DAG run is marked as failed can be found on the “Failed” tab.

Data Interval

Each DAG run in Airflow has an assigned “data interval” that represents the time range it operates in. For a DAG scheduled with `@daily`, for example, each of its data interval would start each day at midnight (00:00) and end at midnight (24:00).

A DAG run is usually scheduled *after* its associated data interval has ended, to ensure the run is able to collect all the data within the time period. In other words, a run covering the data period of 2020-01-01 generally does not start to run until 2020-01-01 has ended, i.e. after 2020-01-02 00:00:00.

All dates in Airflow are tied to the data interval concept in some way. The “logical date” (also called `execution_date` in Airflow versions prior to 2.2) of a DAG run, for example, denotes the start of the data interval, not when the DAG is actually executed.

Similarly, since the `start_date` argument for the DAG and its tasks points to the same logical date, it marks the start of *the DAG’s first data interval*, not when tasks in the DAG will start running. In other words, a DAG run will only be scheduled one interval after `start_date`.

Tip

If a cron expression or timedelta object is not enough to express your DAG’s schedule, logical date, or data interval, see *Timetables*. For more information on logical date, see *Running dags* and *What does execution_date mean?*

Re-run DAG

There can be cases where you will want to execute your DAG again. One such case is when the scheduled DAG run fails.

Catchup

An Airflow DAG defined with a `start_date`, possibly an `end_date`, and a non-asset schedule, defines a series of intervals which the scheduler turns into individual DAG runs and executes. By default, DAG runs that have not been run since the last data interval are not created by the scheduler upon activation of a DAG (Airflow config `scheduler.catchup_by_default=False`). The scheduler creates a DAG run only for the latest interval.

If you set `catchup=True` in the DAG, the scheduler will kick off a DAG Run for any data interval that has not been run since the last data interval (or has been cleared). This concept is called Catchup.

If your DAG is not written to handle its catchup (i.e., not limited to the interval, but instead to Now for instance.), then you will want to turn catchup off, which is the default setting or can be done explicitly by setting `catchup=False` in the DAG definition, if the default config has been changed for your Airflow environment.

```
"""
Code that goes along with the Airflow tutorial located at:
https://github.com/apache/airflow/blob/main/airflow/example_dags/tutorial.py
"""

from airflow.sdk import DAG
from airflow.providers.standard.operators.bash import BashOperator

import datetime
import pendulum

dag = DAG(
    "tutorial",
```

(continues on next page)

(continued from previous page)

```

default_args={
    "depends_on_past": True,
    "retries": 1,
    "retry_delay": datetime.timedelta(minutes=3),
},
start_date=pendulum.datetime(2015, 12, 1, tz="UTC"),
description="A simple tutorial DAG",
schedule="@daily",
)

```

In the example above, if the DAG is picked up by the scheduler daemon on 2016-01-02 at 6 AM, (or from the command line), a single DAG Run will be created with a data between 2016-01-01 and 2016-01-02, and the next one will be created just after midnight on the morning of 2016-01-03 with a data interval between 2016-01-02 and 2016-01-03.

Be aware that using a `datetime.timedelta` object as schedule can lead to a different behavior. In such a case, the single DAG Run created will cover data between 2016-01-01 06:00 and 2016-01-02 06:00 (one schedule interval ending now). For a more detailed description of the differences between a cron and a delta based schedule, take a look at the [timetables comparison](#)

If the `dag.catchup` value had been `True` instead, the scheduler would have created a DAG Run for each completed interval between 2015-12-01 and 2016-01-02 (but not yet one for 2016-01-02, as that interval hasn't completed) and the scheduler will execute them sequentially.

Catchup is also triggered when you turn off a DAG for a specified period and then re-enable it.

This behavior is great for atomic assets that can easily be split into periods. Leaving catchup off is great if your DAG performs catchup internally.

Backfill

You may want to run the DAG for a specified historical period. For example, a DAG is created with `start_date 2024-11-21`, but another user requires the output data from a month prior, i.e. `2024-10-21`. This process is known as Backfill.

This can be done through API or CLI. For CLI usage, run the command below:

```

airflow backfill create --dag-id DAG_ID \
--start-date START_DATE \
--end-date END_DATE \

```

The `backfill` command will re-run all the instances of the `dag_id` for all the intervals within the start date and end date.

Re-run Tasks

Some of the tasks can fail during the scheduled run. Once you have fixed the errors after going through the logs, you can re-run the tasks by clearing them for the scheduled date. Clearing a task instance creates a record of the task instance. The `try_number` of the current task instance is incremented, the `max_tries` set to `0` and the state set to `None`, which causes the task to re-run.

Click on the failed task in the Tree or Graph views and then click on **Clear**. The executor will re-run it.

There are multiple options you can select to re-run -

- **Past** - All the instances of the task in the runs before the DAG's most recent data interval
- **Future** - All the instances of the task in the runs after the DAG's most recent data interval
- **Upstream** - The upstream tasks in the current DAG

- **Downstream** - The downstream tasks in the current DAG
- **Recursive** - All the tasks in the child dags and parent dags
- **Failed** - Only the failed tasks in the DAG's most recent run

You can also clear the task through CLI using the command:

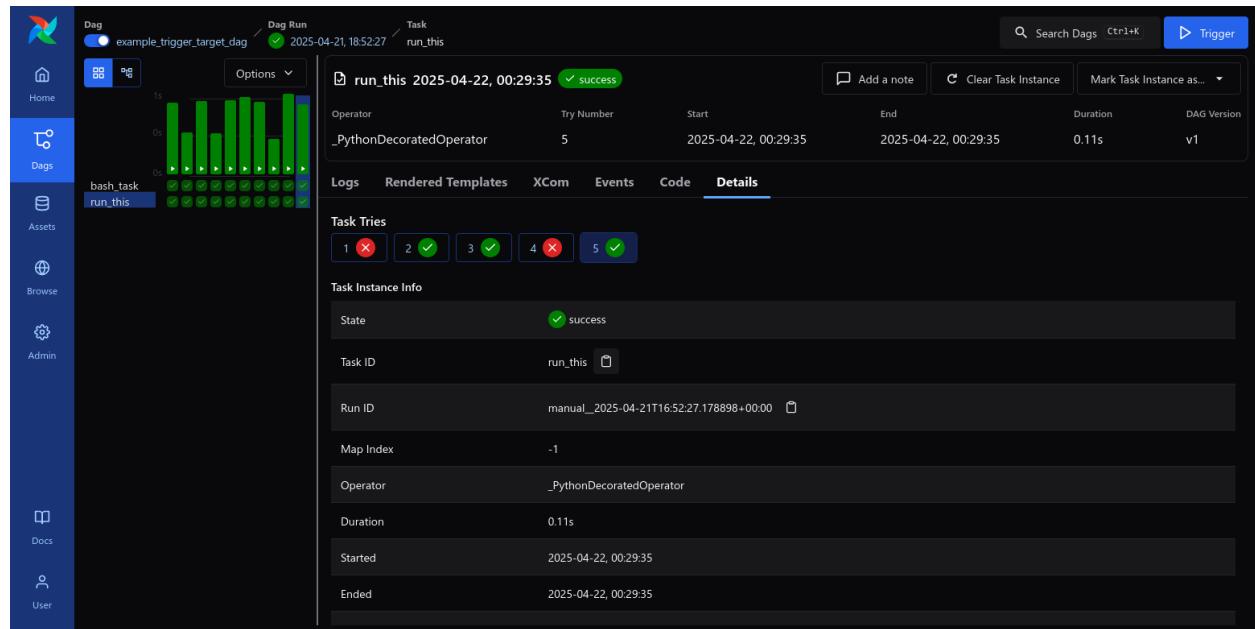
```
airflow tasks clear dag_id \
--task-regex task_regex \
--start-date START_DATE \
--end-date END_DATE
```

For the specified `dag_id` and time interval, the command clears all instances of the tasks matching the regex. For more options, you can check the help of the [clear command](#):

```
airflow tasks clear --help
```

Task Instance History

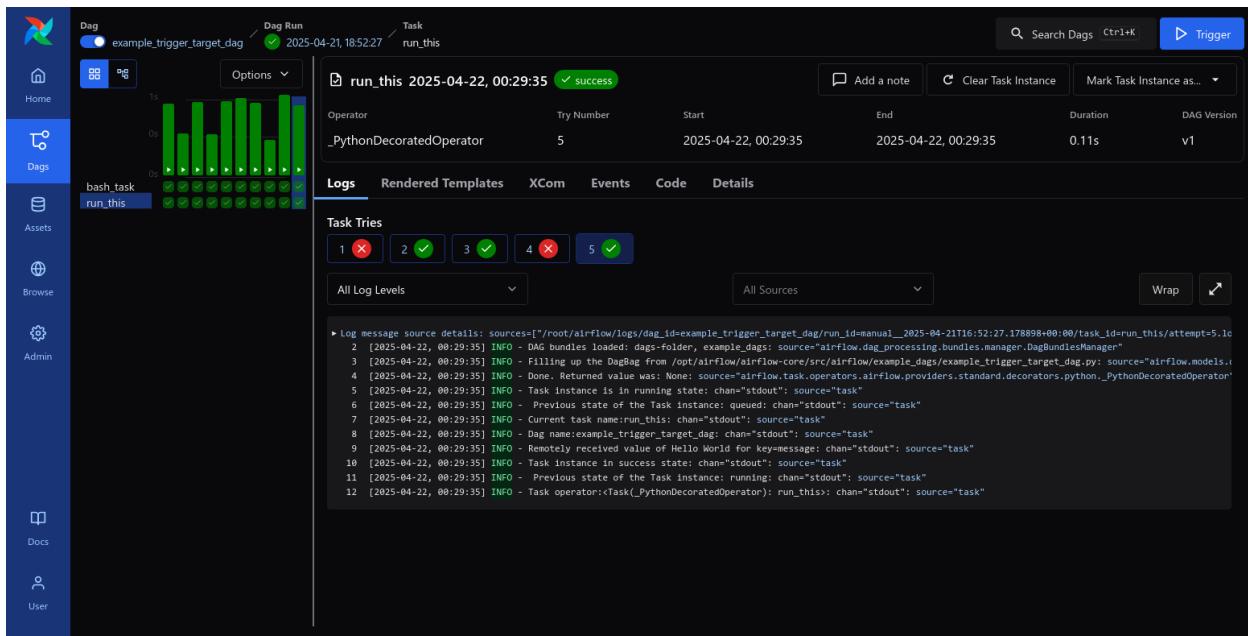
When a task instance retries or is cleared, the task instance history is preserved. You can see this history by clicking on the task instance in the Grid view.



Note

The try selector shown above is only available for tasks that have been retried or cleared.

The history shows the value of the task instance attributes at the end of the particular run. On the log page, you can also see the logs for each of the task instance tries. This can be useful for debugging.



Note

Related task instance objects like the XComs, rendered template fields, etc., are not preserved in the history. Only the task instance attributes, including the logs, are preserved.

External Triggers

Note that DAG Runs can also be created manually through the CLI. Just run the command -

```
airflow dags trigger --logical-date logical_date run_id
```

The DAG Runs created externally to the scheduler get associated with the trigger's timestamp and are displayed in the UI alongside scheduled DAG runs. The logical date passed inside the DAG can be specified using the `-e` argument. The default is the current date in the UTC timezone.

In addition, you can also manually trigger a DAG Run using the web UI (tab **Dags** -> column **Links** -> button **Trigger Dag**)

Passing Parameters when triggering dags

When triggering a DAG from the CLI, the REST API or the UI, it is possible to pass configuration for a DAG Run as a JSON blob.

Example of a parameterized DAG:

```

import pendulum

from airflow.sdk import DAG
from airflow.providers.standard.operators.bash import BashOperator

dag = DAG(
    "example_parameterized_dag",
    schedule=None,
)

```

(continues on next page)

(continued from previous page)

```
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
)

parameterized_task = BashOperator(
    task_id="parameterized_task",
    bash_command="echo value: {{ dag_run.conf['conf1'] }}",
    dag=dag,
)
```

Note: The parameters from `dag_run.conf` can only be used in a template field of an operator.

Using CLI

```
airflow dags trigger --conf '{"conf1": "value1"}' example_parameterized_dag
```

To Keep in Mind

- Marking task instances as failed can be done through the UI. This can be used to stop running task instances.
- Marking task instances as successful can be done through the UI. This is mostly to fix false negatives, or for instance, when the fix has been applied outside of Airflow.

3.7.4 Tasks

A Task is the basic unit of execution in Airflow. Tasks are arranged into *Dags*, and then have upstream and downstream dependencies set between them in order to express the order they should run in.

There are three basic kinds of Task:

- *Operators*, predefined task templates that you can string together quickly to build most parts of your dags.
- *Sensors*, a special subclass of Operators which are entirely about waiting for an external event to happen.
- A *TaskFlow*-decorated `@task`, which is a custom Python function packaged up as a Task.

Internally, these are all actually subclasses of Airflow's `BaseOperator`, and the concepts of Task and Operator are somewhat interchangeable, but it's useful to think of them as separate concepts - essentially, Operators and Sensors are *templates*, and when you call one in a DAG file, you're making a Task.

Relationships

The key part of using Tasks is defining how they relate to each other - their *dependencies*, or as we say in Airflow, their *upstream* and *downstream* tasks. You declare your Tasks first, and then you declare their dependencies second.

Note

We call the *upstream* task the one that is directly preceding the other task. We used to call it a parent task before. Be aware that this concept does not describe the tasks that are higher in the tasks hierarchy (i.e. they are not a direct parents of the task). Same definition applies to *downstream* task, which needs to be a direct child of the other task.

There are two ways of declaring dependencies - using the `>>` and `<<` (bitshift) operators:

```
first_task >> second_task >> [third_task, fourth_task]
```

Or the more explicit `set_upstream` and `set_downstream` methods:

```
first_task.set_downstream(second_task)
third_task.set_upstream(second_task)
```

These both do exactly the same thing, but in general we recommend you use the bitshift operators, as they are easier to read in most cases.

By default, a Task will run when all of its upstream (parent) tasks have succeeded, but there are many ways of modifying this behaviour to add branching, to only wait for some upstream tasks, or to change behaviour based on where the current run is in history. For more, see *Control Flow*.

Tasks don't pass information to each other by default, and run entirely independently. If you want to pass information from one Task to another, you should use *XComs*.

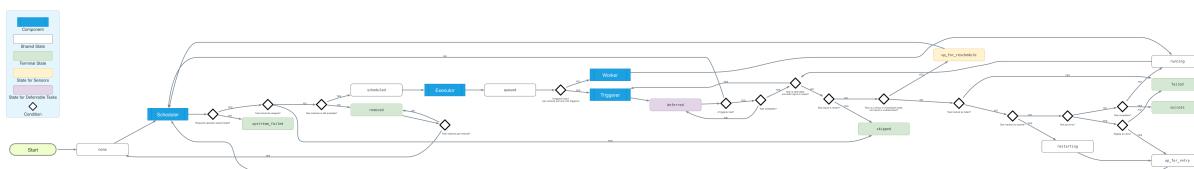
Task Instances

Much in the same way that a DAG is instantiated into a *DAG Run* each time it runs, the tasks under a DAG are instantiated into *Task Instances*.

An instance of a Task is a specific run of that task for a given DAG (and thus for a given data interval). They are also the representation of a Task that has *state*, representing what stage of the lifecycle it is in.

The possible states for a Task Instance are:

- **none**: The Task has not yet been queued for execution (its dependencies are not yet met)
- **scheduled**: The scheduler has determined the Task's dependencies are met and it should run
- **queued**: The task has been assigned to an Executor and is awaiting a worker
- **running**: The task is running on a worker (or on a local/synchronous executor)
- **success**: The task finished running without errors
- **restarting**: The task was externally requested to restart when it was running
- **failed**: The task had an error during execution and failed to run
- **skipped**: The task was skipped due to branching, `LatestOnly`, or similar.
- **upstream_failed**: An upstream task failed and the *Trigger Rule* says we needed it
- **up_for_retry**: The task failed, but has retry attempts left and will be rescheduled.
- **up_for_reschedule**: The task is a *Sensor* that is in `reschedule` mode
- **deferred**: The task has been *deferred to a trigger*
- **removed**: The task has vanished from the DAG since the run started



Ideally, a task should flow from `none`, to `scheduled`, to `queued`, to `running`, and finally to `success`.

When any custom Task (Operator) is running, it will get a copy of the task instance passed to it; as well as being able to inspect task metadata, it also contains methods for things like `XComs`.

Relationship Terminology

For any given Task Instance, there are two types of relationships it has with other instances.

Firstly, it can have *upstream* and *downstream* tasks:

```
task1 >> task2 >> task3
```

When a DAG runs, it will create instances for each of these tasks that are upstream/downstream of each other, but which all have the same data interval.

There may also be instances of the *same task*, but for different data intervals - from other runs of the same DAG. We call these *previous* and *next* - it is a different relationship to *upstream* and *downstream*!

Note

Some older Airflow documentation may still use “previous” to mean “upstream”. If you find an occurrence of this, please help us fix it!

Timeouts

If you want a task to have a maximum runtime, set its `execution_timeout` attribute to a `datetime.timedelta` value that is the maximum permissible runtime. This applies to all Airflow tasks, including sensors. `execution_timeout` controls the maximum time allowed for every execution. If `execution_timeout` is breached, the task times out and `AirflowTaskTimeout` is raised.

In addition, sensors have a `timeout` parameter. This only matters for sensors in `reschedule` mode. `timeout` controls the maximum time allowed for the sensor to succeed. If `timeout` is breached, `AirflowSensorTimeout` will be raised and the sensor fails immediately without retrying.

The following `SFTPSensor` example illustrates this. The sensor is in `reschedule` mode, meaning it is periodically executed and rescheduled until it succeeds.

- Each time the sensor pokes the SFTP server, it is allowed to take maximum 60 seconds as defined by `execution_timeout`.
- If it takes the sensor more than 60 seconds to poke the SFTP server, `AirflowTaskTimeout` will be raised. The sensor is allowed to retry when this happens. It can retry up to 2 times as defined by `retries`.
- From the start of the first execution, till it eventually succeeds (i.e. after the file ‘root/test’ appears), the sensor is allowed maximum 3600 seconds as defined by `timeout`. In other words, if the file does not appear on the SFTP server within 3600 seconds, the sensor will raise `AirflowSensorTimeout`. It will not retry when this error is raised.
- If the sensor fails due to other reasons such as network outages during the 3600 seconds interval, it can retry up to 2 times as defined by `retries`. Retrying does not reset the `timeout`. It will still have up to 3600 seconds in total for it to succeed.

```
sensor = SFTPSensor(  
    task_id="sensor",  
    path="/root/test",  
    execution_timeout=timedelta(seconds=60),  
    timeout=3600,  
    retries=2,
```

(continues on next page)

(continued from previous page)

```
    mode="reschedule",
)
```

SLAs

The SLA feature from Airflow 2 has been removed in 3.0 and will be replaced with a new implementation in Airflow 3.1.

Special Exceptions

If you want to control your task's state from within custom Task/Operator code, Airflow provides two special exceptions you can raise:

- `AirflowSkipException` will mark the current task as skipped
- `AirflowFailException` will mark the current task as failed *ignoring any remaining retry attempts*

These can be useful if your code has extra knowledge about its environment and wants to fail/skip faster - e.g., skipping when it knows there's no data available, or fast-failing when it detects its API key is invalid (as that will not be fixed by a retry).

Task Instance Heartbeat Timeout

No system runs perfectly, and task instances are expected to die once in a while.

`TaskInstances` may get stuck in a `running` state despite their associated jobs being inactive (for example if the `TaskInstance`'s worker ran out of memory). Such tasks were formerly known as zombie tasks. Airflow will find these periodically, clean them up, and mark the `TaskInstance` as failed or retry it if it has available retries. The `TaskInstance`'s heartbeat can timeout for many reasons, including:

- The Airflow worker ran out of memory and was OOMKilled.
- The Airflow worker failed its liveness probe, so the system (for example, Kubernetes) restarted the worker.
- The system (for example, Kubernetes) scaled down and moved an Airflow worker from one node to another.

Reproducing task instance heartbeat timeouts locally

If you'd like to reproduce task instance heartbeat timeouts for development/testing processes, follow the steps below:

1. Set the below environment variables for your local Airflow setup (alternatively you could tweak the corresponding config values in `airflow.cfg`)

```
export AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_SEC=600
export AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_TIMEOUT=2
export AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_TIMEOUT_DETECTION_INTERVAL=5
```

2. Have a DAG with a task that takes about 10 minutes to complete(i.e. a long-running task). For example, you could use the below DAG:

```
from airflow.sdk import dag
from airflow.providers.standard.operators.bash import BashOperator
from datetime import datetime

@dag(start_date=datetime(2021, 1, 1), schedule="@once", catchup=False)
def sleep_dag():
    pass
```

(continues on next page)

(continued from previous page)

```
t1 = BashOperator(  
    task_id="sleep_10_minutes",  
    bash_command="sleep 600",  
)  
  
sleep_dag()
```

Run the above DAG and wait for a while. The TaskInstance will be marked failed after <task_instance_heartbeat_timeout> seconds.

Executor Configuration

Some *Executors* allow optional per-task configuration - such as the `KubernetesExecutor`, which lets you set an image to run the task on.

This is achieved via the `executor_config` argument to a Task or Operator. Here's an example of setting the Docker image for a task that will run on the `KubernetesExecutor`:

```
MyOperator(...,  
    executor_config={  
        "KubernetesExecutor":  
            {"image": "myCustomDockerImage"}  
    }  
)
```

The settings you can pass into `executor_config` vary by executor, so read the *individual executor documentation* in order to see what you can set.

3.7.5 Operators

An Operator is conceptually a template for a predefined *Task*, that you can just define declaratively inside your DAG:

```
with DAG("my-dag") as dag:  
    ping = HttpOperator(endpoint="http://example.com/update/")  
    email = EmailOperator(to="admin@example.com", subject="Update complete")  
  
    ping >> email
```

Airflow has a very extensive set of operators available, with some built-in to the core or pre-installed providers. Some popular operators from core include:

- `BashOperator` - executes a bash command
- `PythonOperator` - calls an arbitrary Python function
- Use the `@task` decorator to execute an arbitrary Python function. It doesn't support rendering jinja templates passed as arguments.

Note

The `@task` decorator is recommended over the classic `PythonOperator` to execute Python callables with no template rendering in its arguments.

For a list of all core operators, see: [Core Operators and Hooks Reference](#).

If the operator you need isn't installed with Airflow by default, you can probably find it as part of our huge set of community providers. Some popular operators from here include:

- EmailOperator
- HttpOperator
- SQLExecuteQueryOperator
- DockerOperator
- HiveOperator
- S3FileTransformOperator
- PrestoToMySqlOperator
- SlackAPIOperator

But there are many, many more - you can see the full list of all community-managed operators, hooks, sensors and transfers in our providers packages documentation.

Note

Inside Airflow's code, we often mix the concepts of *Tasks* and *Operators*, and they are mostly interchangeable. However, when we talk about a *Task*, we mean the generic “unit of execution” of a DAG; when we talk about an *Operator*, we mean a reusable, pre-made Task template whose logic is all done for you and that just needs some arguments.

Jinja Templating

Airflow leverages the power of [Jinja Templating](#) and this can be a powerful tool to use in combination with *macros*.

For example, say you want to pass the start of the data interval as an environment variable to a Bash script using the `BashOperator`:

```
# The start of the data interval as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id="test_env",
    bash_command="/tmp/test.sh",
    dag=dag,
    env={"DATA_INTERVAL_START": date},
)
```

Here, `{{ ds }}` is a templated variable, and because the `env` parameter of the `BashOperator` is templated with Jinja, the data interval's start date will be available as an environment variable named `DATA_INTERVAL_START` in your Bash script.

You can also pass in a callable instead when Python is more readable than a Jinja template. The callable must accept two named arguments `context` and `jinja_env`:

```
def build_complex_command(context, jinja_env):
    with open("file.csv") as f:
        return do_complex_things(f)
```

(continues on next page)

(continued from previous page)

```
t = BashOperator(
    task_id="complex_templated_echo",
    bash_command=build_complex_command,
    dag=dag,
)
```

Since each template field is only rendered once, the callable's return value will not go through rendering again. Therefore, the callable must manually render any templates. This can be done by calling `render_template()` on the current task like this:

```
def build_complex_command(context, jinja_env):
    with open("file.csv") as f:
        data = do_complex_things(f)
    return context["task"].render_template(data, context, jinja_env)
```

You can use 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 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: Sequence[str] = ("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

The `template_fields` property is a class variable and guaranteed to be of a `Sequence[str]` type (i.e. a list or tuple of strings).

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

```
class MyDataTransformer:
    template_fields: Sequence[str] = ("reader",)

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

    # [additional code here...]
```

(continues on next page)

(continued from previous page)

```

class MyDataReader:
    template_fields: Sequence[str] = ("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 the [Jinja documentation](#) to find all available options.

Some operators will also consider strings ending in specific suffixes (defined in `template_ext`) to be references to files when rendering fields. This can be useful for loading scripts or queries directly from files rather than including them into DAG code.

For example, consider a `BashOperator` which runs a multi-line bash script, this will load the file at `script.sh` and use its contents as the value for `bash_command`:

```

run_script = BashOperator(
    task_id="run_script",
    bash_command="script.sh",
)

```

By default, paths provided in this way should be provided relative to the DAG's folder (as this is the default Jinja template search path), but additional paths can be added by setting the `template_searchpath` arg on the DAG.

In some cases, you may want to exclude a string from templating and use it directly. Consider the following task:

```

print_script = BashOperator(
    task_id="print_script",
    bash_command="cat script.sh",
)

```

This will fail with `TemplateNotFound: cat script.sh` since Airflow would treat the string as a path to a file, not a command. We can prevent Airflow from treating this value as a reference to a file by wrapping it in `literal()`. This

approach disables the rendering of both macros and files and can be applied to selected nested fields while retaining the default templating rules for the remainder of the content.

```
from airflow.sdk import literal

fixed_print_script = BashOperator(
    task_id="fixed_print_script",
    bash_command=literal("cat script.sh"),
)
```

Added in version 2.8: `literal()` was added.

Alternatively, if you want to prevent Airflow from treating a value as a reference to a file, you can override `template_ext`:

```
fixed_print_script = BashOperator(
    task_id="fixed_print_script",
    bash_command="cat script.sh",
)
fixed_print_script.template_ext = ()
```

Rendering Fields as Native Python Objects

By default, all Jinja templates in `template_fields` are rendered as strings. This however is not always desired. For example, let's say an `extract` task pushes a dictionary `{"1001": 301.27, "1002": 433.21, "1003": 502.22}` to `XCom`:

```
@task(task_id="extract")
def extract():
    data_string = '{"1001": 301.27, "1002": 433.21, "1003": 502.22}'
    return json.loads(data_string)
```

If a task depends on `extract`, `order_data` argument is passed a string `{"1001": 301.27, "1002": 433.21, "1003": 502.22}"`:

```
def transform(order_data):
    total_order_value = sum(order_data.values()) # Fails because order_data is a str :(
    return {"total_order_value": total_order_value}

transform = PythonOperator(
    task_id="transform",
    op_kwargs={"order_data": "{{ ti.xcom_pull('extract') }}"},
    python_callable=transform,
)

extract() >> transform
```

There are two solutions if we want to get the actual dict instead. The first is to use a callable:

```
def render_transform_op_kwargs(context, jinja_env):
    order_data = context["ti"].xcom_pull("extract")
    return {"order_data": order_data}
```

(continues on next page)

(continued from previous page)

```
transform = PythonOperator(
    task_id="transform",
    op_kwargs=render_transform_op_kwargs,
    python_callable=transform,
)
```

Alternatively, Jinja can also be instructed to render a native Python object. This is done by passing `render_template_as_native_obj=True` to the DAG. This makes Airflow use `NativeEnvironment` instead of the default `SandboxedEnvironment`:

```
with DAG(
    dag_id="example_template_as_python_object",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    render_template_as_native_obj=True,
):
    transform = PythonOperator(
        task_id="transform",
        op_kwargs={"order_data": "{{ ti.xcom_pull('extract') }}"},
        python_callable=transform,
)
```

Reserved params keyword

In Apache Airflow 2.2.0 `params` variable is used during DAG serialization. Please do not use that name in third party operators. If you upgrade your environment and get the following error:

```
AttributeError: 'str' object has no attribute '__module__'
```

change name from `params` in your operators.

Templating Conflicts with f-strings

When constructing strings for templated fields (like `bash_command` in `BashOperator`) using Python f-strings, be mindful of the interaction between f-string interpolation and Jinja templating syntax. Both use curly braces ({{}}).

Python f-strings interpret double curly braces ({{ and }}) as escape sequences for literal single braces ({ and }). However, Jinja uses double curly braces ({{ variable }}) to denote variables for templating.

If you need to include a Jinja template expression (e.g., {{ ds }}) literally within a string defined using an f-string, so that Airflow's Jinja engine can process it later, you must escape the braces for the f-string by doubling them *again*. This means using **four** curly braces:

```
t1 = BashOperator(
    task_id="fstring_templating_correct",
    bash_command=f"echo Data interval start: {{{{ ds }}}}",
    dag=dag,
)

python_var = "echo Data interval start:"
```

(continues on next page)

(continued from previous page)

```
t2 = BashOperator(  
    task_id="fstring_template_simple",  
    bash_command=f"{{python_var}}{{ ds }}}",  
    dag=dag,  
)
```

This ensures the f-string processing results in a string containing the literal double braces required by Jinja, which Airflow can then template correctly before execution. Failure to do this is a common issue for beginners and can lead to errors during DAG parsing or unexpected behavior at runtime when the templating does not occur as expected.

3.7.6 Sensors

Sensors are a special type of *Operator* that are designed to do exactly one thing - wait for something to occur. It can be time-based, or waiting for a file, or an external event, but all they do is wait until something happens, and then *succeed* so their downstream tasks can run.

Because they are primarily idle, Sensors have two different modes of running so you can be a bit more efficient about using them:

- **poke** (default): The Sensor takes up a worker slot for its entire runtime
- **reschedule**: The Sensor takes up a worker slot only when it is checking, and sleeps for a set duration between checks

The `poke` and `reschedule` modes can be configured directly when you instantiate the sensor; generally, the trade-off between them is latency. Something that is checking every second should be in `poke` mode, while something that is checking every minute should be in `reschedule` mode.

Much like Operators, Airflow has a large set of pre-built Sensors you can use, both in core Airflow as well as via our *providers* system.

See also

Deferrable Operators & Triggers

3.7.7 TaskFlow

Added in version 2.0.

If you write most of your dags using plain Python code rather than Operators, then the TaskFlow API will make it much easier to author clean dags without extra boilerplate, all using the `@task` decorator.

TaskFlow takes care of moving inputs and outputs between your Tasks using XComs for you, as well as automatically calculating dependencies - when you call a TaskFlow function in your DAG file, rather than executing it, you will get an object representing the XCom for the result (an `XComArg`), that you can then use as inputs to downstream tasks or operators. For example:

```
from airflow.sdk import task  
from airflow.providers.smtp.operators.smtp import EmailOperator  
  
@task
```

(continues on next page)

(continued from previous page)

```

def get_ip():
    return my_ip_service.get_main_ip()

@task(multiple_outputs=True)
def compose_email(external_ip):
    return {
        'subject': f'Server connected from {external_ip}',
        'body': f'Your server executing Airflow is connected from the external IP\n→{external_ip}<br>'
    }

email_info = compose_email(get_ip())

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

```

Here, there are three tasks - `get_ip`, `compose_email`, and `send_email_notification`.

The first two are declared using TaskFlow, and automatically pass the return value of `get_ip` into `compose_email`, not only linking the XCom across, but automatically declaring that `compose_email` is *downstream* of `get_ip`.

`send_email_notification` is a more traditional Operator, but even it can use the return value of `compose_email` to set its parameters, and again, automatically work out that it must be *downstream* of `compose_email`.

You can also use a plain value or variable to call a TaskFlow function - for example, this will work as you expect (but, of course, won't run the code inside the task until the DAG is executed - the `name` value is persisted as a task parameter until that time):

```

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

hello_name('Airflow users')

```

If you want to learn more about using TaskFlow, you should consult [the TaskFlow tutorial](#).

Context

You can access Airflow *context variables* by adding them as keyword arguments as shown in the following example:

```

from airflow.models.taskinstance import TaskInstance
from airflow.models.dagrun import DagRun


@task
def print_ti_info(task_instance: TaskInstance, dag_run: DagRun):
    print(f"Run ID: {task_instance.run_id}") # Run ID: scheduled_2023-08-
→09T00:00:00+00:00
    print(f"Duration: {task_instance.duration}") # Duration: 0.972019
    print(f"DAG Run queued at: {dag_run.queued_at}") # 2023-08-10_
→00:00:01+02:20

```

Alternatively, you may add `**kwargs` to the signature of your task and all Airflow context variables will be accessible in the `kwargs` dict:

```
from airflow.models.taskinstance import TaskInstance
from airflow.models.dagrun import DagRun

@task
def print_ti_info(**kwargs):
    ti: TaskInstance = kwargs["task_instance"]
    print(f"Run ID: {ti.run_id}") # Run ID: scheduled_2023-08-
    ↪09T00:00:00+00:00
    print(f"Duration: {ti.duration}") # Duration: 0.972019

    dr: DagRun = kwargs["dag_run"]
    print(f"DAG Run queued at: {dr.queued_at}") # 2023-08-10 00:00:01+02:20
```

For a full list of context variables, see *context variables*.

Logging

To use logging from your task functions, simply import and use Python's logging system:

```
logger = logging.getLogger("airflow.task")
```

Every logging line created this way will be recorded in the task log.

Passing Arbitrary Objects As Arguments

Added in version 2.5.0.

As mentioned TaskFlow uses XCom to pass variables to each task. This requires that variables that are used as arguments need to be able to be serialized. Airflow out of the box supports all built-in types (like int or str) and it supports objects that are decorated with `@dataclass` or `@attr.define`. The following example shows the use of a `Asset`, which is `@attr.define` decorated, together with TaskFlow.

Note

An additional benefit of using `Asset` is that it automatically registers as an `inlet` in case it is used as an input argument. It also auto registers as an `outlet` if the return value of your task is a `Asset` or a `list[Asset]`.

```
import json
import pendulum
import requests

from airflow import Asset
from airflow.sdk import dag, task

SRC = Asset(
    "https://www.ncei.noaa.gov/access/monitoring/climate-at-a-glance/global/time-series/
    ↪globe/land_ocean/ytd/12/1880-2022.json"
)
now = pendulum.now()
```

(continues on next page)

(continued from previous page)

```

@dag(start_date=now, schedule="@daily", catchup=False)
def etl():
    @task()
    def retrieve(src: Asset) -> dict:
        resp = requests.get(url=src.uri)
        data = resp.json()
        return data["data"]

    @task()
    def to_fahrenheit(temp: dict[int, dict[str, float]]) -> dict[int, float]:
        ret: dict[int, float] = {}
        for year, info in temp.items():
            ret[year] = float(info["anomaly"]) * 1.8 + 32

        return ret

    @task()
    def load(fahrenheit: dict[int, float]) -> Asset:
        filename = "/tmp/fahrenheit.json"
        s = json.dumps(fahrenheit)
        f = open(filename, "w")
        f.write(s)
        f.close()

        return Asset(f"file:///{{filename}}")

    data = retrieve(SRC)
    fahrenheit = to_fahrenheit(data)
    load(fahrenheit)

etl()

```

Custom Objects

It could be that you would like to pass custom objects. Typically you would decorate your classes with `@dataclass` or `@attr.define` and Airflow will figure out what it needs to do. Sometime you might want to control serialization yourself. To do so add the `serialize()` method to your class and the staticmethod `deserialize(data: dict, version: int)` to your class. Like so:

```

from typing import ClassVar

class MyCustom:
    __version__: ClassVar[int] = 1

    def __init__(self, x):
        self.x = x

    def serialize(self) -> dict:

```

(continues on next page)

(continued from previous page)

```
return dict({"x": self.x})  
  
@staticmethod  
def deserialize(data: dict, version: int):  
    if version > 1:  
        raise TypeError(f"version > {MyCustom.version}")  
    return MyCustom(data["x"])
```

Object Versioning

It is good practice to version the objects that will be used in serialization. To do this add `__version__`: `ClassVar[int] = <x>` to your class. Airflow assumes that your classes are backwards compatible, so that a version 2 is able to deserialize a version 1. In case you need custom logic for deserialization ensure that `deserialize(data: dict, version: int)` is specified.

Note

Typing of `__version__` is required and needs to be `ClassVar[int]`

Sensors and the TaskFlow API

Added in version 2.5.0.

For an example of writing a Sensor using the TaskFlow API, see [Using the TaskFlow API with Sensor operators](#).

History

The TaskFlow API is new as of Airflow 2.0, and you are likely to encounter dags written for previous versions of Airflow that instead use `PythonOperator` to achieve similar goals, albeit with a lot more code.

More context around the addition and design of the TaskFlow API can be found as part of its Airflow Improvement Proposal [AIP-31: “TaskFlow API” for clearer/simpler DAG definition](#)

3.7.8 Executor

Executors are the mechanism by which *task instances* get run. They have a common API and are “pluggable”, meaning you can swap executors based on your installation needs.

Executors are set by the `executor` option in the `[core]` section of the *configuration file*.

Built-in executors are referred to by name, for example:

```
[core]  
executor = KubernetesExecutor
```

Custom or third-party executors can be configured by providing the module path of the executor python class, for example:

```
[core]  
executor = my.custom.executor.module.ExecutorClass
```

Note

For more information on Airflow's configuration, see [Setting Configuration Options](#).

If you want to check which executor is currently set, you can use the `airflow config get-value core executor` command:

```
$ airflow config get-value core executor
LocalExecutor
```

Executor Types

There is only one type of executor that runs tasks *locally* (inside the `scheduler` process) in the repo tree, but custom ones can be written to achieve similar results, and there are those that run their tasks *remotely* (usually via a pool of *workers*). Airflow comes configured with the `LocalExecutor` by default, which is a local executor, and the simplest option for execution. However, as the `LocalExecutor` runs processes in the scheduler process that can have an impact on the performance of the scheduler. You can use the `LocalExecutor` for small, single-machine production installations, or one of the remote executors for a multi-machine/cloud installation.

Local Executors

Airflow tasks are run locally within the scheduler process.

Pros: Very easy to use, fast, very low latency, and few requirements for setup.

Cons: Limited in capabilities and shares resources with the Airflow scheduler.

Examples:

Local Executor

`LocalExecutor` runs tasks by spawning processes in a controlled fashion in different modes.

Given that `BaseExecutor` has the option to receive a `parallelism` parameter to limit the number of process spawned, when this parameter is `0` the number of processes that `LocalExecutor` can spawn is unlimited.

The following strategies are implemented:

- **Unlimited Parallelism** (`self.parallelism == 0`): In this strategy, `LocalExecutor` will spawn a process every time `execute_async` is called, that is, every task submitted to the `LocalExecutor` will be executed in its own process. Once the task is executed and the result stored in the `result_queue`, the process terminates. There is no need for a `task_queue` in this approach, since as soon as a task is received a new process will be allocated to the task. Processes used in this strategy are of class `LocalWorker`.
- **Limited Parallelism** (`self.parallelism > 0`): In this strategy, the `LocalExecutor` spawns the number of processes equal to the value of `self.parallelism` at start time, using a `task_queue` to coordinate the ingestion of tasks and the work distribution among the workers, which will take a task as soon as they are ready. During the lifecycle of the `LocalExecutor`, the worker processes are running waiting for tasks, once the `LocalExecutor` receives the call to shutdown the executor a poison token is sent to the workers to terminate them. Processes used in this strategy are of class `QueuedLocalWorker`.

Note

When multiple Schedulers are configured with `executor = LocalExecutor` in the [core] section of your `airflow.cfg`, each Scheduler will run a LocalExecutor. This means tasks would be processed in a distributed fashion across the machines running the Schedulers.

One consideration should be taken into account:

- **Restarting a Scheduler:** If a Scheduler is restarted, it may take some time for other Schedulers to recognize the orphaned tasks and restart or fail them.

Remote Executors

Remote executors can further be divided into two categories:

Queued/Batch Executors

Airflow tasks are sent to a central queue where remote workers pull tasks to execute. Often workers are persistent and run multiple tasks at once.

Pros: More robust since you're decoupling workers from the scheduler process. Workers can be large hosts that can churn through many tasks (often in parallel) which is cost effective. Latency can be relatively low since workers can be provisioned to be running at all times to take tasks immediately from the queue.

Cons: Shared workers have the noisy neighbor problem with tasks competing for resources on the shared hosts or competing for how the environment/system is configured. They can also be expensive if your workload is not constant, you may have workers idle, overly scaled in resources, or you have to manage scaling them up and down.

Examples:

- CeleryExecutor
- BatchExecutor
- EdgeExecutor (Experimental Pre-Release)

Containerized Executors

Airflow tasks are executed ad hoc inside containers/pods. Each task is isolated in its own containerized environment that is deployed when the Airflow task is queued.

Pros: Each Airflow task is isolated to one container so no noisy neighbor problem. The execution environment can be customized for specific tasks (system libs, binaries, dependencies, amount of resources, etc). Cost effective as the workers are only alive for the duration of the task.

Cons: There is latency on startup since the container or pod needs to deploy before the task can begin. Can be expensive if you're running many short/small tasks. No workers to manage however you must manage something like a Kubernetes cluster.

Examples:

- KubernetesExecutor
- EcsExecutor

Note

New Airflow users may assume they need to run a separate executor process using one of the Local or Remote Executors. This is not correct. The executor logic runs *inside* the scheduler process, and will run the tasks locally or not depending on the executor selected.

Using Multiple Executors Concurrently

Starting with version 2.10.0, Airflow can now operate with a multi-executor configuration. Each executor has its own set of pros and cons, often they are trade-offs between latency, isolation and compute efficiency among other properties (see [here](#) for comparisons of executors). Running multiple executors allows you to make better use of the strengths of all the available executors and avoid their weaknesses. In other words, you can use a specific executor for a specific set of tasks where its particular merits and benefits make the most sense for that use case.

Configuration

Configuring multiple executors uses the same configuration option (as described [here](#)) as single executor use cases, leveraging a comma separated list notation to specify multiple executors.

Note

The first executor in the list (either on its own or along with other executors) will behave the same as it did in pre-2.10.0 releases. In other words, this will be the default executor for the environment. Any Airflow Task or DAG that does not specify a specific executor will use this environment level executor. All other executors in the list will be initialized and ready to run tasks if specified on an Airflow Task or DAG. If you do not specify an executor in this configuration list, it cannot be used to run tasks.

Some examples of valid multiple executor configuration:

```
[core]
executor = LocalExecutor
```

```
[core]
executor = LocalExecutor,CeleryExecutor
```

```
[core]
executor = KubernetesExecutor,my.custom.module.ExecutorClass
```

Note

Using two instances of the `_same_` executor class is not currently supported.

To make it easier to specify executors on tasks and dags, executor configuration now supports aliases. You may then use this alias to refer to the executor in your dags (see below).

```
[core]
executor = LocalExecutor,ShortName:my.custom.module.ExecutorClass
```

Note

If a DAG specifies a task to use an executor that is not configured, the DAG will fail to parse and a warning dialog will be shown in the Airflow UI. Please ensure that all executors you wish to use are specified in Airflow configuration on *any* host/container that is running an Airflow component (scheduler, workers, etc).

Writing dags and tasks

To specify an executor for a task, make use of the executor parameter on Airflow Operators:

```
BashOperator(  
    task_id="hello_world",  
    executor="LocalExecutor",  
    bash_command="echo 'hello world!'",  
)
```

```
@task(executor="LocalExecutor")  
def hello_world():  
    print("hello world!")
```

To specify an executor for an entire DAG, make use of the existing Airflow mechanism of default arguments. All tasks in the DAG will then use the specified executor (unless explicitly overridden by a specific task):

```
def hello_world():  
    print("hello world!")  
  
def hello_world_again():  
    print("hello world again!")  
  
with DAG(  
    dag_id="hello_worlds",  
    default_args={"executor": "LocalExecutor"}, # Applies to all tasks in the DAG  
) as dag:  
    # All tasks will use the executor from default args automatically  
    hw = hello_world()  
    hw_again = hello_world_again()
```

Note

Tasks store the executor they were configured to run on in the Airflow database. Changes are reflected after each parsing of a DAG.

Monitoring

When using a single executor, Airflow metrics will behave as they were <2.9. But if multiple executors are configured then the executor metrics (`executor.open_slots`, `executor.queued_slots`, and `executor.running_tasks`) will be published for each executor configured, with the executor name appended to the metric name (e.g. `executor.open_slots.<executor class name>`).

Logging works the same as the single executor use case.

Statically-coded Hybrid Executors

There are currently two “statically coded” executors, these executors are hybrids of two different executors: the LocalKubernetesExecutor and the CeleryKubernetesExecutor. Their implementation is not native or intrinsic to core Airflow. These hybrid executors instead make use of the `queue` field on Task Instances to indicate and persist which

sub-executor to run on. This is a misuse of the `queue` field and makes it impossible to use it for its intended purpose when using these hybrid executors.

Executors such as these also require hand crafting new “concrete” classes to create each permutation of possible combinations of executors. This is untenable as more executors are created and leads to more maintenance overhead. Bespoke coding effort should not be required to use any combination of executors.

Therefore using these types of executors is no longer recommended.

Writing Your Own Executor

All Airflow executors implement a common interface so that they are pluggable and any executor has access to all abilities and integrations within Airflow. Primarily, the Airflow scheduler uses this interface to interact with the executor, but other components such as logging and CLI do as well. The public interface is the `BaseExecutor`. You can look through the code for the most detailed and up to date interface, but some important highlights are outlined below.

Note

For more information about Airflow’s public interface see *Public Interface of Airflow*.

Some reasons you may want to write a custom executor include:

- An executor does not exist which fits your specific use case, such as a specific tool or service for compute.
- You’d like to use an executor that leverages a compute service from your preferred cloud provider.
- You have a private tool/service for task execution that is only available to you or your organization.

Workloads

A workload in context of an Executor is the fundamental unit of execution for an executor. It represents a discrete operation or job that the executor runs on a worker. For example, it can run user code encapsulated in an Airflow task on a worker.

Example:

```
ExecuteTask(
    token="mock",
    ti=TaskInstance(
        id=UUID("4d828a62-a417-4936-a7a6-2b3fabacecab"),
        task_id="mock",
        dag_id="mock",
        run_id="mock",
        try_number=1,
        map_index=-1,
        pool_slots=1,
        queue="default",
        priority_weight=1,
        executor_config=None,
        parent_context_carrier=None,
        context_carrier=None,
        queued_dttm=None,
    ),
    dag_rel_path=PurePosixPath("mock.py"),
    bundle_info=BundleInfo(name="n/a", version="no matter"),
    log_path="mock.log",
```

(continues on next page)

(continued from previous page)

```
    type="ExecuteTask",  
)
```

Important BaseExecutor Methods

These methods don't require overriding to implement your own executor, but are useful to be aware of:

- **heartbeat**: The Airflow scheduler Job loop will periodically call heartbeat on the executor. This is one of the main points of interaction between the Airflow scheduler and the executor. This method updates some metrics, triggers newly queued tasks to execute and updates state of running/completed tasks.
- **queue_workload**: The Airflow Executor will call this method of the BaseExecutor to provide tasks to be run by the executor. The BaseExecutor simply adds the *workloads* (check section above to understand) to an internal list of queued workloads to run within the executor. All executors present in the repository use this method.
- **get_event_buffer**: The Airflow scheduler calls this method to retrieve the current state of the TaskInstances the executor is executing.
- **has_task**: The scheduler uses this BaseExecutor method to determine if an executor already has a specific task instance queued or running.
- **send_callback**: Sends any callbacks to the sink configured on the executor.

Mandatory Methods to Implement

The following methods must be overridden at minimum to have your executor supported by Airflow:

- **sync**: Sync will get called periodically during executor heartbeats. Implement this method to update the state of the tasks which the executor knows about. Optionally, attempting to execute queued tasks that have been received from the scheduler.
- **execute_async**: Executes a *workload* asynchronously. This method is called (after a few layers) during executor heartbeat which is run periodically by the scheduler. In practice, this method often just enqueues tasks into an internal or external queue of tasks to be run (e.g. KubernetesExecutor). But can also execute the tasks directly as well (e.g. LocalExecutor). This will depend on the executor.

Optional Interface Methods to Implement

The following methods aren't required to override to have a functional Airflow executor. However, some powerful capabilities and stability can come from implementing them:

- **start**: The Airflow scheduler job will call this method after it initializes the executor object. Any additional setup required by the executor can be completed here.
- **end**: The Airflow scheduler job will call this method as it is tearing down. Any synchronous cleanup required to finish running jobs should be done here.
- **terminate**: More forcefully stop the executor, even killing/stopping in-flight tasks instead of synchronously waiting for completion.
- **try_adopt_task_instances**: Tasks that have been abandoned (e.g. from a scheduler job that died) are provided to the executor to adopt or otherwise handle them via this method. Any tasks that cannot be adopted (by default the BaseExecutor assumes all cannot be adopted) should be returned.
- **get_cli_commands**: Executors may vend CLI commands to users by implementing this method, see the *CLI* section below for more details.
- **get_task_log**: Executors may vend log messages to Airflow task logs by implementing this method, see the *Logging* section below for more details.

Compatibility Attributes

The `BaseExecutor` class interface contains a set of attributes that Airflow core code uses to check the features that your executor is compatible with. When writing your own Airflow executor be sure to set these correctly for your use case. Each attribute is simply a boolean to enable/disable a feature or indicate that a feature is supported/unsupported by the executor:

- `supports_pickling`: Whether or not the executor supports reading pickled dags from the Database before execution (rather than reading the DAG definition from the file system).
- `supports_sentry`: Whether or not the executor supports [Sentry](#).
- `is_local`: Whether or not the executor is remote or local. See the *Executor Types* section above.
- `is_single_threaded`: Whether or not the executor is single threaded. This is particularly relevant to what database backends are supported. Single threaded executors can run with any backend, including SQLite.
- `is_production`: Whether or not the executor should be used for production purposes. A UI message is displayed to users when they are using a non-production ready executor.
- `serve_logs`: Whether or not the executor supports serving logs, see *Logging for Tasks*.

CLI

Executors may vend CLI commands which will be included in the `airflow` command line tool by implementing the `get_cli_commands` method. Executors such as `CeleryExecutor` and `KubernetesExecutor` for example, make use of this mechanism. The commands can be used to setup required workers, initialize environment or set other configuration. Commands are only vended for the currently configured executor. A pseudo-code example of implementing CLI command vending from an executor can be seen below:

```
@staticmethod
def get_cli_commands() -> list[GroupCommand]:
    sub_commands = [
        ActionCommand(
            name="command_name",
            help="Description of what this specific command does",
            func=lazy_load_command("path.to.python.function.for.command"),
            args=(),
        ),
    ]

    return [
        GroupCommand(
            name="my_cool_executor",
            help="Description of what this group of commands do",
            subcommands=sub_commands,
        ),
    ]
```

Note

Currently there are no strict rules in place for the Airflow command namespace. It is up to developers to use names for their CLI commands that are sufficiently unique so as to not cause conflicts with other Airflow executors or components.

Note

When creating a new executor, or updating any existing executors, be sure to not import or execute any expensive operations/code at the module level. Executor classes are imported in several places and if they are slow to import this will negatively impact the performance of your Airflow environment, especially for CLI commands.

Logging

Executors may vend log messages which will be included in the Airflow task logs by implementing the `get_task_logs` method. This can be helpful if the execution environment has extra context in the case of task failures, which may be due to the execution environment itself rather than the Airflow task code. It can also be helpful to include setup/teardown logging from the execution environment. The `KubernetesExecutor` leverages this this capability to include logs from the pod which ran a specific Airflow task and display them in the logs for that Airflow task. A pseudo-code example of implementing task log vending from an executor can be seen below:

```
def get_task_log(self, ti: TaskInstance, try_number: int) -> tuple[list[str], list[str]]:
    messages = []
    log = []
    try:
        res = helper_function_to_fetch_logs_from_execution_env(ti, try_number)
        for line in res:
            log.append(remove_escape_codes(line.decode()))
        if log:
            messages.append("Found logs from execution environment!")
    except Exception as e: # No exception should cause task logs to fail
        messages.append(f"Failed to find logs from execution environment: {e}")
    return messages, ["\n".join(log)]
```

Next Steps

Once you have created a new executor class implementing the `BaseExecutor` interface, you can configure Airflow to use it by setting the `core.executor` configuration value to the module path of your executor:

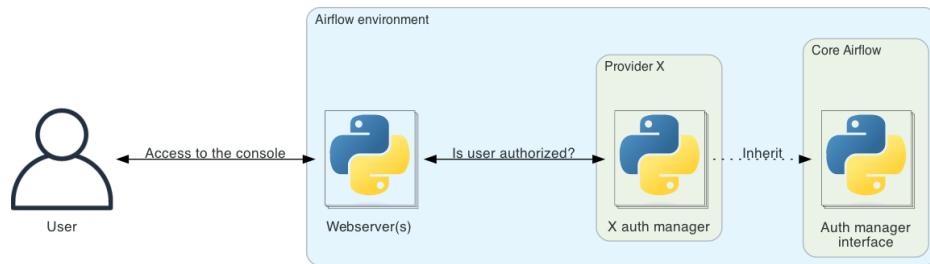
```
[core]
executor = my_company.executors.MyCustomExecutor
```

Note

For more information on Airflow's configuration, see [Setting Configuration Options](#) and for more information on managing Python modules in Airflow see [Modules Management](#).

3.7.9 Auth manager

Auth (for authentication/authorization) manager is the component in Airflow to handle user authentication and user authorization. They have a common API and are “pluggable”, meaning you can swap auth managers based on your installation needs.



Airflow can only have one auth manager configured at a time; this is set by the `auth_manager` option in the `[core]` section of *the configuration file*.

i Note

For more information on Airflow's configuration, see *Setting Configuration Options*.

If you want to check which auth manager is currently set, you can use the `airflow config get-value core auth_manager` command:

```
$ airflow config get-value core auth_manager
airflow.providers.fab.auth_manager.fab_auth_manager.FabAuthManager
```

Simple auth manager

i Note

The Simple auth manager is intended for development and testing. If you're using it in production, ensure that access is controlled through other means.

The simple auth manager is the auth manager that comes by default in Airflow 3. As its name suggests, the logic and implementation of the simple auth manager is **simple**.

Generate JWT token with simple auth manager

i Note

This guide only applies if your environment is configured with simple auth manager.

In order to use the *Airflow public API*, you need a JWT token for authentication. You can then include this token in your Airflow public API requests. To generate a JWT token, use the `Create Token API` in *Simple auth manager token API*.

Example

```
ENDPOINT_URL="http://localhost:8080/"
curl -X 'POST' \
"${ENDPOINT_URL}/auth/token" \
-H 'Content-Type: application/json' \
-d '{
  "username": "<username>",
  "password": "<password>"
}'
```

This process will return a token that you can use in the Airflow public API requests.

If `[core] simple_auth_manager_all_admins` is set to True, you can also generate a token with no credentials.

```
ENDPOINT_URL="http://localhost:8080/"
curl -X 'GET' "${ENDPOINT_URL}/auth/token"
```

Simple auth manager token API

It's a stub file. It will be converted automatically during the build process to the valid documentation by the Sphinx plugin. See: `/docs/conf.py`

Manage users

Users are managed through the Airflow configuration. Example:

```
[core]
simple_auth_manager_users = "bob:admin,peter:viewer"
```

The list of users are separated with a comma and each user is a couple username/role separated by a colon. Each user needs two pieces of information:

- **username**. The user's username
- **role**. The role associated to the user. For more information about these roles, *see next section*.

In the example above, two users are defined:

- **bob** whose role is **admin**
- **peter** whose role is **viewer**

The password is auto-generated for each user and printed out in the webserver logs. When generated, these passwords are saved in a file configured in `core.simple_auth_manager_passwords_file`. By default, this file is `$AIRFLOW_HOME/simple_auth_manager_passwords.json.generated`, you can read and update them directly in the file as well if desired.

Note

With Breeze, two users are predefined: `admin` and `viewer` (password is the same as the username). `admin` has all permissions. `viewer` has read-only permissions.

Manage roles and permissions

There is no option to manage roles and permissions in simple auth manager. They are defined as part of the simple auth manager implementation and cannot be modified. Here is the list of roles defined in simple auth manager. These roles can be associated to users.

- **viewer**. Read-only permissions on dags, assets and pools
- **user**. `viewer` permissions plus all permissions (edit, create, delete) on dags
- **op**. `user` permissions plus all permissions on pools, assets, config, connections and variables
- **admin**. All permissions

Optional features

Disable authentication and allow everyone as admin

This option allow you to disable authentication and allow everyone as admin. As a consequence, whoever access the Airflow UI is automatically logged in as an admin with all permissions.

you can enable this feature through the config. Example:

```
[core]
simple_auth_manager_all_admins = "True"
```

Available auth managers to use

Here is the list of auth managers available today that you can use in your Airflow environment.

Provided by Airflow:

- *Simple auth manager*

Provided by providers:

- apache-airflow-providers-fab:auth-manager/index
- apache-airflow-providers-amazon:auth-manager/index

Why pluggable auth managers?

Airflow is used by a lot of different users with a lot of different configurations. Some Airflow environment might be used by only one user and some might be used by thousand of users. An Airflow environment with only one (or very few) users does not need the same user management as an environment used by thousand of them.

This is why the whole user management (user authentication and user authorization) is packaged in one component called auth manager. So that it is easy to plug-and-play an auth manager that suits your specific needs.

By default, Airflow comes with the apache-airflow-providers-fab:auth-manager/index.

Note

Switching to a different auth manager is a heavy operation and should be considered as such. It will impact users of the environment. The sign-in and sign-off experience will very likely change and disturb them if they are not

advised. Plus, all current users and permissions will have to be copied over from the previous auth manager to the next.

Writing your own auth manager

All Airflow auth managers implement a common interface so that they are pluggable and any auth manager has access to all abilities and integrations within Airflow. This interface is used across Airflow to perform all user authentication and user authorization related operation.

The public interface is `BaseAuthManager`. You can look through the code for the most detailed and up to date interface, but some important highlights are outlined below.

Note

For more information about Airflow's public interface see *Public Interface of Airflow*.

Some reasons you may want to write a custom auth manager include:

- An auth manager does not exist which fits your specific use case, such as a specific tool or service for user management.
- You'd like to use an auth manager that leverages an identity provider from your preferred cloud provider.
- You have a private user management tool that is only available to you or your organization.

Authentication related `BaseAuthManager` methods

- `get_user`: Return the signed-in user.
- `get_url_login`: Return the URL the user is redirected to for signing in.

Authorization related `BaseAuthManager` methods

Most of authorization methods in `BaseAuthManager` look the same. Let's go over the different parameters used by most of these methods.

- `method`: Use HTTP method naming to determine the type of action being done on a specific resource.
 - `GET`: Can the user read the resource?
 - `POST`: Can the user create a resource?
 - `PUT`: Can the user modify the resource?
 - `DELETE`: Can the user delete the resource?
 - `MENU`: Can the user see the resource in the menu?
- `details`: Optional details about the resource being accessed.
- `user`: The user trying to access the resource.

These authorization methods are:

- `is_authorized_configuration`: Return whether the user is authorized to access Airflow configuration. Some details about the configuration can be provided (e.g. the config section).
- `is_authorized_connection`: Return whether the user is authorized to access Airflow connections. Some details about the connection can be provided (e.g. the connection ID).

- `is_authorized_dag`: Return whether the user is authorized to access a DAG. Some details about the DAG can be provided (e.g. the DAG ID). Also, `is_authorized_dag` is called for any entity related to dags (e.g. task instances, dag runs, ...). This information is passed in `access_entity`. Example: `auth_manager.is_authorized_dag(method="GET", access_entity=DagAccessEntity. Run, details=DagDetails(id="dag-1"))` asks whether the user has permission to read the Dag runs of the dag “dag-1”.
- `is_authorized_dataset`: Return whether the user is authorized to access Airflow datasets. Some details about the dataset can be provided (e.g. the dataset uri).
- `is_authorized_pool`: Return whether the user is authorized to access Airflow pools. Some details about the pool can be provided (e.g. the pool name).
- `is_authorized_variable`: Return whether the user is authorized to access Airflow variables. Some details about the variable can be provided (e.g. the variable key).
- `is_authorized_view`: Return whether the user is authorized to access a specific view in Airflow. The view is specified through `access_view` (e.g. `AccessView.CLUSTER_ACTIVITY`).
- `is_authorized_custom_view`: Return whether the user is authorized to access a specific view not defined in Airflow. This view can be provided by the auth manager itself or a plugin defined by the user.

JWT token management by auth managers

The auth manager is responsible for creating the JWT token needed to interact with Airflow public API. To achieve this, the auth manager **must** provide an endpoint to create this JWT token. This endpoint must be available at `POST /auth/token`

The auth manager is also responsible of passing the JWT token to Airflow UI. The protocol to exchange the JWT token between the auth manager and Airflow UI is using cookies. The auth manager needs to save the JWT token in a cookie named `_token` before redirecting to the Airflow UI. The Airflow UI will then read the cookie, save it and delete the cookie.

```
from airflow.api_fastapi.auth.managers.base_auth_manager import COOKIE_NAME_JWT_TOKEN

response = RedirectResponse(url="/")

secure = bool(conf.get("api", "ssl_cert", fallback=""))
response.set_cookie(COOKIE_NAME_JWT_TOKEN, token, secure=secure)
return response
```

Note

Do not set the cookie parameter `httponly` to True. Airflow UI needs to access the JWT token from the cookie.

Optional methods recommended to override for optimization

The following methods aren’t required to override to have a functional Airflow auth manager. However, it is recommended to override these to make your auth manager faster (and potentially less costly):

- `batch_is_authorized_dag`: Batch version of `is_authorized_dag`. If not overridden, it will call `is_authorized_dag` for every single item.
- `batch_is_authorized_connection`: Batch version of `is_authorized_connection`. If not overridden, it will call `is_authorized_connection` for every single item.

- `batch_is_authorized_pool`: Batch version of `is_authorized_pool`. If not overridden, it will call `is_authorized_pool` for every single item.
- `batch_is_authorized_variable`: Batch version of `is_authorized_variable`. If not overridden, it will call `is_authorized_variable` for every single item.
- `get_authorized_dag_ids`: Return the list of DAG IDs the user has access to. If not overridden, it will call `is_authorized_dag` for every single DAG available in the environment.

CLI

Auth managers may vend CLI commands which will be included in the `airflow` command line tool by implementing the `get_cli_commands` method. The commands can be used to setup required resources. Commands are only vended for the currently configured auth manager. A pseudo-code example of implementing CLI command vending from an auth manager can be seen below:

```
@staticmethod
def get_cli_commands() -> list[CLICommand]:
    sub_commands = [
        ActionCommand(
            name="command_name",
            help="Description of what this specific command does",
            func=lazy_load_command("path.to.python.function.for.command"),
            args=(),
        ),
    ]
    return [
        GroupCommand(
            name="my_cool_auth_manager",
            help="Description of what this group of commands do",
            subcommands=sub_commands,
        ),
    ]
```

Note

Currently there are no strict rules in place for the Airflow command namespace. It is up to developers to use names for their CLI commands that are sufficiently unique so as to not cause conflicts with other Airflow components.

Note

When creating a new auth manager, or updating any existing auth manager, be sure to not import or execute any expensive operations/code at the module level. Auth manager classes are imported in several places and if they are slow to import this will negatively impact the performance of your Airflow environment, especially for CLI commands.

Extending API server application

Auth managers have the option to extend the Airflow API server. Doing so, allow, for instance, to vend additional public API endpoints. To extend the API server application, you need to implement the `get_fastapi_app` method.

Such additional endpoints can be used to manage resources such as users, groups, roles (if any) handled by your auth manager. Endpoints defined by `get_fastapi_app` are mounted in `/auth`.

Next Steps

Once you have created a new auth manager class implementing the `BaseAuthManager` interface, you can configure Airflow to use it by setting the `core.auth_manager` configuration value to the module path of your auth manager:

[core]

```
auth_manager = my_company.auth_managers.MyCustomAuthManager
```

Note

For more information on Airflow’s configuration, see *Setting Configuration Options* and for more information on managing Python modules in Airflow see *Modules Management*.

3.7.10 Object Storage

Added in version 2.8.0.

All major cloud providers offer persistent data storage in object stores. These are not classic “POSIX” file systems. In order to store hundreds of petabytes of data without any single points of failure, object stores replace the classic file system directory tree with a simpler model of object-name => data. To enable remote access, operations on objects are usually offered as (slow) HTTP REST operations.

Airflow provides a generic abstraction on top of object stores, like s3, gcs, and azure blob storage. This abstraction allows you to use a variety of object storage systems in your dags without having to change your code to deal with every different object storage system. In addition, it allows you to use most of the standard Python modules, like `shutil`, that can work with file-like objects.

Support for a particular object storage system depends on the providers you have installed. For example, if you have installed the `apache-airflow-providers-google` provider, you will be able to use the `gcs` scheme for object storage. Out of the box, Airflow provides support for the `file` scheme.

Note

Support for s3 requires you to install `apache-airflow-providers-amazon[s3fs]`. This is because it depends on `aiobotocore`, which is not installed by default as it can create dependency challenges with `botocore`.

Cloud Object Stores are not real file systems

Object stores are not real file systems although they can appear so. They do not support all the operations that a real file system does. Key differences are:

- No guaranteed atomic rename operation. This means that if you move a file from one location to another, it will be copied and then deleted. If the copy fails, you will lose the file.
- Directories are emulated and might make working with them slow. For example, listing a directory might require listing all the objects in the bucket and filtering them by prefix.
- Seeking within a file may require significant call overhead hurting performance or might not be supported at all.

Airflow relies on fsspec to provide a consistent experience across different object storage systems. It implements local file caching to speed up access. However, you should be aware of the limitations of object storage when designing your dags.

Basic Use

To use object storage, you need to instantiate a Path (see below) object with the URI of the object you want to interact with. For example, to point to a bucket in s3, you would do the following:

```
from airflow.sdk import ObjectStoragePath  
  
base = ObjectStoragePath("s3://aws_default@my-bucket/")
```

The username part of the URI represents the Airflow connection id and is optional. It can alternatively be passed in as a separate keyword argument:

```
# Equivalent to the previous example.  
base = ObjectStoragePath("s3://my-bucket/", conn_id="aws_default")
```

Listing file-objects:

```
@task  
def list_files() -> list[ObjectStoragePath]:  
    files = [f for f in base.iterdir() if f.is_file()]  
    return files
```

Navigating inside a directory tree:

```
base = ObjectStoragePath("s3://my-bucket/")  
subdir = base / "subdir"  
  
# prints ObjectStoragePath("s3://my-bucket/subdir")  
print(subdir)
```

Opening a file:

```
@task  
def read_file(path: ObjectStoragePath) -> str:  
    with path.open() as f:  
        return f.read()
```

Leveraging XCOM, you can pass paths between tasks:

```
@task  
def create(path: ObjectStoragePath) -> ObjectStoragePath:  
    return path / "new_file.txt"  
  
@task  
def write_file(path: ObjectStoragePath, content: str):  
    with path.open("wb") as f:  
        f.write(content)  
  
new_file = create(base)
```

(continues on next page)

(continued from previous page)

```
write = write_file(new_file, b"data")

read >> write
```

Configuration

In its basic use, the object storage abstraction does not require much configuration and relies upon the standard Airflow connection mechanism. This means that you can use the `conn_id` argument to specify the connection to use. Any settings by the connection are pushed down to the underlying implementation. For example, if you are using s3, you can specify the `aws_access_key_id` and `aws_secret_access_key` but also add extra arguments like `endpoint_url` to specify a custom endpoint.

Alternative backends

It is possible to configure an alternative backend for a scheme or protocol. This is done by attaching a `backend` to the scheme. For example, to enable the databricks backend for the `dbfs` scheme, you would do the following:

```
from airflow.sdk import ObjectStoragePath
from airflow.sdk.io import attach

from fsspec.implementations.dbfs import DBFSFileSystem

attach(protocol="dbfs", fs=DBFSFileSystem(instance="myinstance", token="mytoken"))
base = ObjectStoragePath("dbfs://my-location/")
```

Note

To reuse the registration across tasks, make sure to attach the backend at the top-level of your DAG. Otherwise, the backend will not be available across multiple tasks.

Path API

The object storage abstraction is implemented as a [Path API](#), and builds upon [Universal Pathlib](#). This means that you can mostly use the same API to interact with object storage as you would with a local filesystem. In this section we only list the differences between the two APIs. Extended operations beyond the standard Path API, like copying and moving, are listed in the next section. For details about each operation, like what arguments they take, see the documentation of the `ObjectStoragePath` class.

mkdir

Create a directory entry at the specified path or within a bucket/container. For systems that don't have true directories, it may create a directory entry for this instance only and not affect the real filesystem.

If `parents` is `True`, any missing parents of this path are created as needed.

touch

Create a file at this given path, or update the timestamp. If `truncate` is `True`, the file is truncated, which is the default. If the file already exists, the function succeeds if `exists_ok` is `true` (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

stat

Returns a `stat_result` like object that supports the following attributes: `st_size`, `st_mtime`, `st_mode`, but also acts like a dictionary that can provide additional metadata about the object. For example, for s3 it will, return the additional keys like: `['ETag' , 'ContentType']`. If your code needs to be portable across different object stores do not rely on the extended metadata.

Extensions

The following operations are not part of the standard Path API, but are supported by the object storage abstraction.

bucket

Returns the bucket name.

checksum

Returns the checksum of the file.

container

Alias of bucket

fs

Convenience attribute to access an instantiated filesystem

key

Returns the object key.

namespace

Returns the namespace of the object. Typically this is the protocol, like `s3://` with the bucket name.

path

the `fsspec` compatible path for use with filesystem instances

protocol

the `filesystem_spec` protocol.

read_block

Read a block of bytes from the file at this given path.

Starting at offset of the file, read length bytes. If delimiter is set then we ensure that the read starts and stops at delimiter boundaries that follow the locations offset and offset + length. If offset is zero then we start at zero. The bytestring returned WILL include the end delimiter string.

If offset+length is beyond the eof, reads to eof.

sign

Create a signed URL representing the given path. Some implementations allow temporary URLs to be generated, as a way of delegating credentials.

size

Returns the size in bytes of the file at the given path.

storage_options

The storage options for instantiating the underlying filesystem.

ukey

Hash of file properties, to tell if it has changed.

Copying and Moving

This documents the expected behavior of the `copy` and `move` operations, particularly for cross object store (e.g. file -> s3) behavior. Each method copies or moves files or directories from a `source` to a `target` location. The intended behavior is the same as specified by `fsspec`. For cross object store directory copying, Airflow needs to walk the directory tree and copy each file individually. This is done by streaming each file from the source to the target.

External Integrations

Many other projects, like DuckDB, Apache Iceberg etc, can make use of the object storage abstraction. Often this is done by passing the underlying `fsspec` implementation. For this purpose `ObjectStoragePath` exposes the `fs` property. For example, the following works with duckdb so that the connection details from Airflow are used to connect to s3 and a parquet file, indicated by a `ObjectStoragePath`, is read:

```
import duckdb
from airflow.sdk import ObjectStoragePath

path = ObjectStoragePath("s3://my-bucket/my-table.parquet", conn_id="aws_default")
conn = duckdb.connect(database=":memory:")
conn.register_filesystem(path.fs)
conn.execute(f"CREATE OR REPLACE TABLE my_table AS SELECT * FROM read_parquet('{path}');
    ↪")
```

3.7.11 Backfill

Backfill is when you create runs for past dates of a dag. Airflow provides a mechanism to do this through the CLI and REST API. You provide a dag, a start date, and an end date, and Airflow will create runs in the range according to the dag's schedule.

Backfill does not make sense for dags that don't have a time-based schedule.

Control over data reprocessing

There are three options for reprocessing behavior:

- **none** - if there's already a run for this logical date, do not create another, no matter the state
- **failed** - if a run exists, if the state is failed, create a new run for this date

- **completed** - if a run exists, if the state is completed or failed, create a new run for this date

If the latest run is still running or is queued, we do not create another run, no matter the chosen reprocessing behavior.

Concurrency control

You can set `max_active_runs` on a backfill and it will control how many dag runs in the backfill can run concurrently. Backfill `max_active_runs` is applied independently the DAG `max_active_runs` setting.

Run ordering

You can run your backfill in reverse, i.e. latest runs first. The CLI option is `--run-backwards`.

Dry run

Backfill dry run is a CLI option that will print out the dates that the backfill will consider creating runs for. Whether or not they will be created depends on your chosen reprocessing behavior and the states of any existing runs in the range at the time you actually run the backfill.

Example:

```
airflow backfill create --dag-id tutorial \
--start-date 2015-06-01 \
--end-date 2015-06-07 \
--reprocessing-behavior failed \
--max-active-runs 3 \
--run-backwards \
--dag-run-conf '{"my": "param"}'
```

3.7.12 Message Queues

The Message Queues are a way to expose capability of external event-driven scheduling of Dags.

Apache Airflow is primarily designed for time-based and dependency-based scheduling of workflows. However, modern data architectures often require near real-time processing and the ability to react to events from various sources, such as message queues.

Airflow has native event-driven capability, allowing users to create workflows that can be triggered by external events, thus enabling more responsive data pipelines.

Airflow supports poll-based event-driven scheduling, where the Triggerer can poll external message queues using built-in `airflow.triggers.base.BaseTrigger` classes. This allows users to create workflows that can be triggered by external events, such as messages arriving in a queue or changes in a database efficiently.

Airflow constantly monitors the state of an external resource and updates the asset whenever the external resource reaches a given state (if it does reach it). To achieve this, we leverage Airflow Triggers. Triggers are small, asynchronous pieces of Python code whose job is to poll an external resource state.

The list of supported message queues is available in apache-airflow-providers:core-extensions/message-queues.

Communication

3.7.13 XComs

XComs (short for “cross-communications”) are a mechanism that let *Tasks* talk to each other, as by default Tasks are entirely isolated and may be running on entirely different machines.

An XCom is identified by a key (essentially its name), as well as the `task_id` and `dag_id` it came from. They can have any serializable value (including objects that are decorated with `@dataclass` or `@attr.define`, see *TaskFlow arguments*:), but they are only designed for small amounts of data; do not use them to pass around large values, like dataframes.

XComs are explicitly “pushed” and “pulled” to/from their storage using the `xcom_push` and `xcom_pull` methods on Task Instances.

To push a value within a task called “**task-1**” that will be used by another task:

```
# pushes data in any_serializable_value into xcom with key "identifier as string"
task_instance.xcom_push(key="identifier as a string", value=any_serializable_value)
```

To pull the value that was pushed in the code above in a different task:

```
# pulls the xcom variable with key "identifier as string" that was pushed from within
# task-1
task_instance.xcom_pull(key="identifier as string", task_ids="task-1")
```

Many operators will auto-push their results into an XCom key called `return_value` if the `do_xcom_push` argument is set to `True` (as it is by default), and `@task` functions do this as well. `xcom_pull` defaults to using `return_value` as key if no key is passed to it, meaning it's possible to write code like this:

```
# Pulls the return_value XCOM from "pushing_task"
value = task_instance.xcom_pull(task_ids='pushing_task')
```

You can also use XComs in *templates*:

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

XComs are a relative of *Variables*, with the main difference being that XComs are per-task-instance and designed for communication within a DAG run, while Variables are global and designed for overall configuration and value sharing.

If you want to push multiple XComs at once you can set `do_xcom_push` and `multiple_outputs` arguments to `True`, and then return a dictionary of values.

An example of pushing multiple XComs and pulling them individually:

```
# A task returning a dictionary
@task(do_xcom_push=True, multiple_outputs=True)
def push_multiple(**context):
    return {"key1": "value1", "key2": "value2"}


@task
def xcom_pull_with_multiple_outputs(**context):
    # Pulling a specific key from the multiple outputs
    key1 = context["ti"].xcom_pull(task_ids="push_multiple", key="key1") # to pull key1
    key2 = context["ti"].xcom_pull(task_ids="push_multiple", key="key2") # to pull key2

    # Pulling entire xcom data from push_multiple task
    data = context["ti"].xcom_pull(task_ids="push_multiple", key="return_value")
```

 Note

If the first task run is not succeeded then on every retry task XComs will be cleared to make the task run idempotent.

Object Storage XCom Backend

The default XCom backend, `BaseXCom`, stores XComs in the Airflow database, which works well for small values but can cause issues with large values or a high volume of XComs. To overcome this limitation, object storage is recommended for efficiently handling larger data. For a detailed overview, refer to the documentation.

Custom XCom Backends

The XCom system has interchangeable backends, and you can set which backend is being used via the `xcom_backend` configuration option.

If you want to implement your own backend, you should subclass `BaseXCom`, and override the `serialize_value` and `deserialize_value` methods.

You can override the `purge` method in the `BaseXCom` class to have control over purging the xcom data from the custom backend. This will be called as part of `delete`.

Verifying Custom XCom Backend usage in Containers

Depending on where Airflow is deployed i.e., local, Docker, K8s, etc. it can be useful to be assured that a custom XCom backend is actually being initialized. For example, the complexity of the container environment can make it more difficult to determine if your backend is being loaded correctly during container deployment. Luckily the following guidance can be used to assist you in building confidence in your custom XCom implementation.

If you can exec into a terminal in an Airflow container, you can then print out the actual XCom class that is being used:

```
from airflow.models.xcom import XCom

print(XCom.__name__)
```

3.7.14 Variables

Variables are Airflow's runtime configuration concept - a general key/value store that is global and can be queried from your tasks, and easily set via Airflow's user interface, or bulk-uploaded as a JSON file.

To use them, just import and call `get` on the Variable model:

```
from airflow.sdk import Variable

# Normal call style
foo = Variable.get("foo")

# Auto-deserializes a JSON value
bar = Variable.get("bar", deserialize_json=True)

# Returns the value of default (None) if the variable is not set
baz = Variable.get("baz", default=None)
```

You can also use them from *templates*:

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

# Auto-deserialize JSON value
echo {{ var.json.<variable_name> }}
```

Variables are **global**, and should only be used for overall configuration that covers the entire installation; to pass data from one Task/Operator to another, you should use *XComs* instead.

We also recommend that you try to keep most of your settings and configuration in your DAG files, so it can be versioned using source control; Variables are really only for values that are truly runtime-dependent.

For more information on setting and managing variables, see *Managing Variables*.

3.7.15 Params

Params enable you to provide runtime configuration to tasks. You can configure default Params in your DAG code and supply additional Params, or overwrite Param values, at runtime when you trigger a DAG. Param values are validated with JSON Schema. For scheduled DAG runs, default Param values are used.

Also defined Params are used to render a nice UI when triggering manually. When you trigger a DAG manually, you can modify its Params before the dagrun starts. If the user-supplied values don't pass validation, Airflow shows a warning instead of creating the dagrun.

DAG-level Params

To add Params to a *DAG*, initialize it with the `params` kwarg. Use a dictionary that maps Param names to either a Param or an object indicating the parameter's default value.

```
from airflow.sdk import DAG
from airflow.sdk import task
from airflow.sdk import Param

with DAG(
    "the_dag",
    params={
        "x": Param(5, type="integer", minimum=3),
        "my_int_param": 6
    },
) as dag:

    @task.python
    def example_task(params: dict):
        # This will print the default value, 6:
        dag.log.info(dag.params['my_int_param'])

        # This will print the manually-provided value, 42:
        dag.log.info(params['my_int_param'])

        # This will print the default value, 5, since it wasn't provided manually:
        dag.log.info(params['x'])

    example_task()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    dag.test(
        run_conf={"my_int_param": 42}
    )
```

Note

DAG-level parameters are the default values passed on to tasks. These should not be confused with values manually provided through the UI form or CLI, which exist solely within the context of a `DagRun` and a `TaskInstance`. This distinction is crucial for TaskFlow dags, which may include logic within the `with DAG(...)` as `dag:` block. In such cases, users might try to access the manually-provided parameter values using the `dag` object, but this will only ever contain the default values. To ensure that the manually-provided values are accessed, use a template variable such as `params` or `ti` within your task.

Task-level Params

You can also add Params to individual tasks.

```
def print_my_int_param(params):
    print(params.my_int_param)

PythonOperator(
    task_id="print_my_int_param",
    params={"my_int_param": 10},
    python_callable=print_my_int_param,
)
```

Task-level params take precedence over DAG-level params, and user-supplied params (when triggering the DAG) take precedence over task-level params.

Referencing Params in a Task

Params can be referenced in *templated strings* under `params`. For example:

```
PythonOperator(
    task_id="from_template",
    op_args=[
        "{{ params.my_int_param + 10 }}",
    ],
    python_callable=(
        lambda my_int_param: print(my_int_param)
    ),
)
```

Even though Params can use a variety of types, the default behavior of templates is to provide your task with a string. You can change this by setting `render_template_as_native_obj=True` while initializing the `DAG`.

```
with DAG(
    "the_dag",
    params={"my_int_param": Param(5, type="integer", minimum=3)},
```

(continues on next page)

(continued from previous page)

```
    render_template_as_native_obj=True
):
```

This way, the Param's type is respected when it's provided to your task:

```
# prints <class 'str'> by default
# prints <class 'int'> if render_template_as_native_obj=True
PythonOperator(
    task_id="template_type",
    op_args=[
        "{{ params.my_int_param }}",
    ],
    python_callable=
        lambda my_int_param: print(type(my_int_param))
),
)
```

Another way to access your param is via a task's context kwarg.

```
def print_my_int_param(**context):
    print(context["params"]["my_int_param"])

PythonOperator(
    task_id="print_my_int_param",
    python_callable=print_my_int_param,
    params={"my_int_param": 12345},
)
```

JSON Schema Validation

Param makes use of JSON Schema, so you can use the full JSON Schema specifications mentioned at <https://json-schema.org/draft/2020-12/json-schema-validation.html> to define Param objects.

```
with DAG(
    "my_dag",
    params={
        # an int with a default value
        "my_int_param": Param(10, type="integer", minimum=0, maximum=20),

        # a required param which can be of multiple types
        # a param must have a default value
        "multi_type_param": Param(5, type=["null", "number", "string"]),

        # an enum param, must be one of three values
        "enum_param": Param("foo", enum=["foo", "bar", 42]),

        # a param which uses json-schema formatting
        "email": Param(
            default="example@example.com",
            type="string",
            format="idn-email",
            minLength=5,
```

(continues on next page)

(continued from previous page)

```
        maxLength=255,  
    ),  
},  
):
```

Note

If `schedule` is defined for a DAG, params with defaults must be valid. This is validated during DAG parsing. If `schedule=None` then params are not validated during DAG parsing but before triggering a DAG. This is useful in cases where the DAG author does not want to provide defaults but wants to force users provide valid parameters at time of trigger.

Note

As of now, for security reasons, one can not use `Param` objects derived out of custom classes. We are planning to have a registration system for custom `Param` classes, just like we've for Operator ExtraLinks.

Use Params to Provide a Trigger UI Form

Added in version 2.6.0.

DAG level params are used to render a user friendly trigger form. This form is provided when a user clicks on the “Trigger DAG” button.

The Trigger UI Form is rendered based on the pre-defined DAG Params. If the DAG has no params defined, the trigger form is skipped. The form elements can be defined with the `Param` class and attributes define how a form field is displayed.

The following features are supported in the Trigger UI Form:

- Direct scalar values (boolean, int, string, lists, dicts) from top-level DAG params are auto-boxed into `Param` objects. From the native Python data type the `type` attribute is auto detected. So these simple types render to a corresponding field type. The name of the parameter is used as label and no further validation is made, all values are treated as optional.
- If you use the `Param` class as definition of the parameter value, the following attributes can be added:
 - The `Param` attribute `title` is used to render the form field label of the entry box. If no `title` is defined the parameter name/key is used instead.
 - The `Param` attribute `description` is rendered below an entry field as help text in gray color. If you want to provide special formatting or links you need to use the `Param` attribute `description_md`. See tutorial `DAG Params UI example DAG` for an example.
 - The `Param` attribute `type` influences how a field is rendered. The following types are supported:

| Param type | Form element type | Additional supported attributes | Example |
|---------------------|---|---|--|
| <code>string</code> | Generates a single-line text box or a text area to edit text. | <ul style="list-style-type: none"> * <code>minLength</code>: Minimum text length * <code>maxLength</code>: Maximum text length * * <code>format="date"</code>: Generate a date-picker with calendar pop-up * * <code>format="date-time"</code>: Generate a date and time-picker with calendar pop-up * * <code>format="time"</code>: Generate a time-picker * * <code>format="multiline"</code>: Generate a multi-line textarea * <code>enum=["a", "b", "c"]</code>: Generates a drop-down select list for scalar values. As of JSON validation, a value must be selected or the field must be marked as optional explicit. See also details inside the JSON Schema Description for Enum. * | <pre>Param("default", type="string", maxLength=10) Param(f"{{datetime. date.today()}}", type="string", format="date") * format="date": Generate a date-picker with calendar pop-up * format="date-time": Generate a date and time-picker with calendar pop-up * format="time": Generate a time- picker * format="multiline": Generate a multi-line textarea * enum=["a", "b", "c"]: Generates a drop-down select list for scalar values. As of JSON validation, a value must be selected or the field must be marked as optional explicit. See also details inside the JSON Schema Description for Enum. *</pre> |

- If a form field is left empty, it is passed as `None` value to the params dict.
- Form fields are rendered in the order of definition of `params` in the DAG.
- If you want to add sections to the Form, add the attribute `section` to each field. The text will be used as section label. Fields w/o `section` will be rendered in the default area. Additional sections will be collapsed per default.
- If you want to have params not being displayed, use the `const` attribute. These Params will be submitted but hidden in the Form. The `const` value must match the default value to pass [JSON Schema validation](#).
- On the bottom of the form the generated JSON configuration can be expanded. If you want to change values manually, the JSON configuration can be adjusted. Changes are overridden when form fields change.
- To pre-populate values in the form when publishing a link to the trigger form you can call the trigger URL `/dags/<dag_name>/trigger` and add query parameter to the URL in the form `name=value`, for example `/dags/example_params_ui_tutorial/trigger?required_field=some%20text`. To pre-define the run id of the DAG run, use the URL parameter `run_id`.
- Fields can be required or optional. Typed fields are required by default to ensure they pass JSON schema validation. To make typed fields optional, you must allow the “null” type.
- Fields without a “section” will be rendered in the default area. Additional sections will be collapsed by default.

Note

If the field is required the default value must be valid according to the schema as well. If the DAG is defined with `schedule=None` the parameter value validation is made at time of trigger.

For examples, please take a look at the two example dags provided: *Params trigger example DAG* and *Params UI example DAG*.

`airflow/example_dags/example_params_trigger_ui.py`

```
with DAG(
    dag_id=Path(__file__).stem,
    dag_display_name="Params Trigger UI",
    description=__doc__.partition(".")[0],
    doc_md=__doc__,
    schedule=None,
    start_date=datetime.datetime(2022, 3, 4),
    catchup=False,
    tags=["example", "params"],
    params={
        "names": Param(
            ["Linda", "Martha", "Thomas"],
            type="array",
            description="Define the list of names for which greetings should be"
            "generated in the logs."
            " Please have one name per line.",
            title="Names to greet",
        ),
        "english": Param(True, type="boolean", title="English"),
        "german": Param(True, type="boolean", title="German (Formal)"),
        "french": Param(True, type="boolean", title="French"),
    },
) as dag:
```

(continues on next page)

(continued from previous page)

```

@task(task_id="get_names", task_display_name="Get names")
def get_names(**kwargs) -> list[str]:
    params = kwargs["params"]
    if "names" not in params:
        print("Uuups, no names given, was no UI used to trigger?")
        return []
    return params["names"]

@task.branch(task_id="select_languages", task_display_name="Select languages")
def select_languages(**kwargs) -> list[str]:
    params = kwargs["params"]
    selected_languages = []
    for lang in ["english", "german", "french"]:
        if params[lang]:
            selected_languages.append(f"generate_{lang}_greeting")
    return selected_languages

@task(task_id="generate_english_greeting", task_display_name="Generate English ↵ greeting")
def generate_english_greeting(name: str) -> str:
    return f"Hello {name}!"

@task(task_id="generate_german_greeting", task_display_name="Erzeuge Deutsche ↵ Begrüßung")
def generate_german_greeting(name: str) -> str:
    return f"Sehr geehrter Herr/Frau {name}."

@task(task_id="generate_french_greeting", task_display_name="Produire un message d ↵ accueil en français")
def generate_french_greeting(name: str) -> str:
    return f"Bonjour {name}!"

@task(task_id="print_greetings", task_display_name="Print greetings", trigger_rule=TriggerRule.ALL_DONE)
def print_greetings(greetings1, greetings2, greetings3) -> None:
    for g in greetings1 or []:
        print(g)
    for g in greetings2 or []:
        print(g)
    for g in greetings3 or []:
        print(g)
    if not (greetings1 or greetings2 or greetings3):
        print("sad, nobody to greet :-(")

lang_select = select_languages()
names = get_names()
english_greetings = generate_english_greeting.expand(name=names)
german_greetings = generate_german_greeting.expand(name=names)
french_greetings = generate_french_greeting.expand(name=names)
lang_select >> [english_greetings, german_greetings, french_greetings]
results_print = print_greetings(english_greetings, german_greetings, french_

```

(continues on next page)

(continued from previous page)

↳ greetings)

airflow/example_dags/example_params_ui_tutorial.py

```

params={
    # Let's start simple: Standard dict values are detected from type and offered as
    # entry form fields.
    # Detected types are numbers, text, boolean, lists and dicts.
    # Note that such auto-detected parameters are treated as optional (not required
    # to contain a value)
    "number_param": 3,
    "text_param": "Hello World!",
    "bool_param": False,
    "list_param": ["one", "two", "three", "actually one value is made per line"],
    "dict_param": {"key": "value"},
    # You can arrange the entry fields in sections so that you can have a better
    # overview for the user
    # Therefore you can add the "section" attribute.
    # But of course you might want to have it nicer! Let's add some description to
    # parameters.
    # Note if you can add any Markdown formatting to the description, you need to
    # use the description_md
    # attribute.
    "most_loved_number": Param(
        42,
        type="integer",
        title="Your favorite number",
        description_md="Everybody should have a **favorite** number. Not only _math_
        teachers_. "
        "If you can not think of any at the moment please think of the 42 which is
        very famous because "
        "of the book [The Hitchhiker's Guide to the Galaxy]"
        "(https://en.wikipedia.org/wiki/Phrases\_from\_The\_Hitchhiker%27s\_Guide\_to\_the\_Galaxy#")
        "The_Answer_to_the_Ultimate_Question_of_Life,_the_Universe,_and_Everything_
        is_42).",
        minimum=0,
        maximum=128,
        section="Typed parameters with Param object",
    ),
    # If you want to have a selection list box then you can use the enum feature of
    # JSON schema
    "pick_one": Param(
        "value 42",
        type="string",
        title="Select one Value",
        description="You can use JSON schema enum's to generate drop down selection
        boxes.",
        enum=[f"value {i}" for i in range(16, 64)],
        section="Typed parameters with Param object",
    ),
}

```

airflow/example_dags/example_params_ui_tutorial.py

```

"required_field": Param(
    # In this example we have no default value
    # Form will enforce a value supplied by users to be able to trigger
    type="string",
    title="Required text field",
    minLength=10,
    maxLength=30,
    description="This field is required. You can not submit without having text ↵
in here.",
    section="Typed parameters with Param object",
),
"optional_field": Param(
    "optional text, you can trigger also w/o text",
    type=['null', "string"],
    title="Optional text field",
    description_md="This field is optional. As field content is JSON schema ↵
validated you must "
    "allow the `null` type.",
    section="Typed parameters with Param object",
),

```

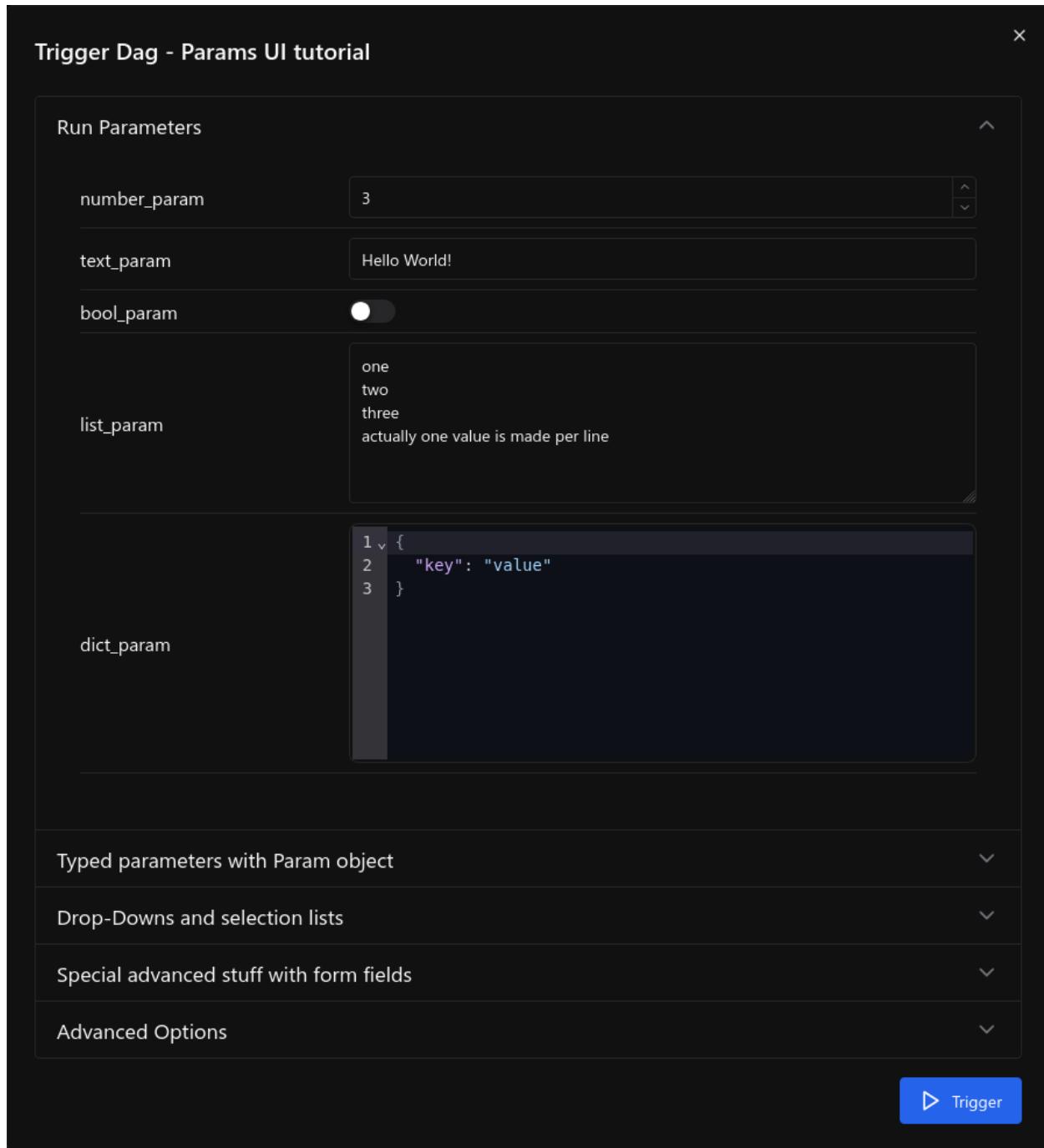
airflow/example_dags/example_params_ui_tutorial.py

```

@task(task_display_name="Show used parameters")
def show_params(**kwargs) -> None:
    params = kwargs["params"]
    print(f"This DAG was triggered with the following parameters:\n\n{json.
.dumps(params, indent=4)}\n")
show_params()

```

The Params UI Tutorial is rendered in 4 sections with the most common examples. The first section shows the basic usage without Param class.



The second section shows how to use the `Param` class to define more attributes.

Trigger Dag - Params UI tutorial

Run Parameters

Typed parameters with Param object

Your favorite number *

42

Everybody should have a **favorite** number. Not only math teachers. If you can not think of any at the moment please think of the 42 which is very famous because of the book *The Hitchhiker's Guide to the Galaxy*.

Select one Value *

value 42

You can use JSON schema enum's to generate drop down selection boxes.

Please confirm *

An On/Off selection with a proper description.

Date-Time Picker *

04/21/2025, 12:17 PM

Please select a date and time, use the button on the left for a pop-up calendar.

Date Picker *

04/21/2025

Please select a date, use the button on the left for a pop-up calendar. See that here are no times!

Time Picker

12:13:14 PM

Please select a time, use the button on the left for a pop-up tool.

Required text field *

This field is required. You can not submit without having text in here.

Optional text field

optional text, you can trigger also w/o text

This field is optional. As field content is JSON schema validated you must allow the null type.

Drop-Downs and selection lists

Special advanced stuff with form fields

Advanced Options

 Trigger

The third section shows how to model selection lists and drop-downs.

Trigger Dag - Params UI tutorial

Run Parameters

Typed parameters with Param object

Drop-Downs and selection lists

Select one Number *

Three

With drop down selections you can also have nice display labels for the values.

Field with proposals *

Alpha

You can use JSON schema examples's to generate drop down selection boxes but allow also to enter custom values. Try typing an 'a' and see options.

Multi Select *

two X three X

Select from the list of options.

Multi Select with Labels *

Two bananas X Three apples X

Select from the list of options. See that options can have nicer text and still technical valuesare propagated as values during trigger to the DAG.

Special advanced stuff with form fields

Advanced Options

Trigger

The screenshot shows a dark-themed UI for triggering a DAG. At the top, it says "Trigger Dag - Params UI tutorial". Below that are several sections: "Run Parameters", "Typed parameters with Param object", and "Drop-Downs and selection lists". Under "Drop-Downs and selection lists", there are four examples: 1) "Select one Number *" with a dropdown showing "Three" and a note about display labels. 2) "Field with proposals *" with a dropdown showing "Alpha" and a note about generating dropdowns via JSON schema examples. 3) "Multi Select *" with a multi-select field containing "two" and "three" with an "X" button. 4) "Multi Select with Labels *" with a multi-select field containing "Two bananas" and "Three apples" with an "X" button. Below these sections are "Special advanced stuff with form fields" and "Advanced Options". At the bottom right is a large blue "Trigger" button with a white triangle icon.

Finally the fourth section shows advanced form elements.

Trigger Dag - Params UI tutorial

Run Parameters

Typed parameters with Param object

Drop-Downs and selection lists

Special advanced stuff with form fields

Array of numbers *

```
1 v [  
2   1,  
3   2,  
4   3  
5 ]
```

Only integers are accepted in this array

Multiline text

```
A multiline text Param  
that will keep the newline  
characters in its value.
```

This field allows for multiline text input. The returned value will be a single string with newline (\n) characters kept intact.

JSON entry field

```
1 v {  
2   "key": "value"  
3 }
```

JSON array field *

```
1 v [  
2 v   {  
3     "name": "account_name",  
4     "country": "country_name"  
5   }  
6 ]
```

Array with complex objects and validation rules. See JSON Schema validation options in specs.

Advanced Options

Trigger

Changed in version 3.0.0: By default custom HTML is not allowed to prevent injection of scripts or other malicious HTML code. The previous field named `description_html` is now super-seeded with the attribute `description_md`. `description_html` is not supported anymore. Custom form elements using the attribute `custom_html_form` was deprecated in version 2.8.0 and support was removed in 3.0.0.

Disabling Runtime Param Modification

The ability to update params while triggering a DAG depends on the flag `core.dag_run_conf_overrides_params`. Setting this config to `False` will effectively turn your default params into constants.

Debugging

3.7.16 Debugging Airflow dags

Testing dags with `dag.test()`

To debug dags in an IDE, you can set up the `dag.test` command in your dag file and run through your DAG in a single serialized python process.

This approach can be used with any supported database (including a local SQLite database) and will *fail fast* as all tasks run in a single process.

To set up `dag.test`, add these two lines to the bottom of your dag file:

```
if __name__ == "__main__":
    dag.test()
```

and that's it! You can add optional arguments to fine tune the testing but otherwise you can run or debug dags as needed. Here are some examples of arguments:

- `execution_date` if you want to test argument-specific DAG runs
- `use_executor` if you want to test the DAG using an executor. By default `dag.test` runs the DAG without an executor, it just runs all the tasks locally. By providing this argument, the DAG is executed using the executor configured in the Airflow environment.

Conditionally skipping tasks

If you don't wish to execute some subset of tasks in your local environment (e.g. dependency check sensors or cleanup steps), you can automatically mark them successful supplying a pattern matching their `task_id` in the `mark_success_pattern` argument.

In the following example, testing the dag won't wait for either of the upstream dags to complete. Instead, testing data is manually ingested. The cleanup step is also skipped, making the intermediate csv is available for inspection.

```
with DAG("example_dag", default_args=default_args) as dag:
    sensor = ExternalTaskSensor(task_id="wait_for_ingestion_dag", external_dag_id=
    "ingest_raw_data")
    sensor2 = ExternalTaskSensor(task_id="wait_for_dim_dag", external_dag_id="ingest_dim
    ")
    collect_stats = PythonOperator(task_id="extract_stats_csv", python_callable=extract_
    stats_csv)
    # ... run other tasks
    cleanup = PythonOperator(task_id="cleanup", python_callable=Path.unlink, op_
    args=[collect_stats.output])
```

(continues on next page)

(continued from previous page)

```
[sensor, sensor2] >> collect_stats >> cleanup

if __name__ == "__main__":
    ingest_testing_data()
    run = dag.test(mark_success_pattern="wait_for_.*|cleanup")
    print(f"Intermediate csv: {run.get_task_instance('collect_stats').xcom_pull(task_id=
    'collect_stats')}")
```

Debugging Airflow dags on the command line

With the same two line addition as mentioned in the above section, you can now easily debug a DAG using pdb as well. Run `python -m pdb <path to dag file>.py` for an interactive debugging experience on the command line.

```
root@ef2c84ad4856:/opt/airflow# python -m pdb airflow/example_dags/example_bash_operator.
-> py
> /opt/airflow/airflow/example_dags/example_bash_operator.py(18)<module>()
-> """Example DAG demonstrating the usage of the BashOperator."""
(Pdb) b 45
Breakpoint 1 at /opt/airflow/airflow/example_dags/example_bash_operator.py:45
(Pdb) c
> /opt/airflow/airflow/example_dags/example_bash_operator.py(45)<module>()
-> bash_command='echo 1',
(Pdb) run_this_last
<Task(EmptyOperator): run_this_last>
```

IDE setup steps:

1. Add `main` block at the end of your DAG file to make it runnable.

```
if __name__ == "__main__":
    dag.test()
```

2. Run / debug the DAG file.

3.8 Authoring and Scheduling

Here you can find detailed documentation about advanced authoring and scheduling Airflow dags. It's recommended that you first review the pages in *core concepts*

Authoring

3.8.1 Deferrable Operators & Triggers

Standard *Operators* and *Sensors* take up a full *worker slot* for the entire time they are running, even if they are idle. For example, if you only have 100 worker slots available to run tasks, and you have 100 dags waiting on a sensor that's currently running but idle, then you *cannot run anything else* - even though your entire Airflow cluster is essentially idle. *reschedule* mode for sensors solves some of this, by allowing sensors to only run at fixed intervals, but it is inflexible and only allows using time as the reason to resume, not other criteria.

This is where *Deferrable Operators* can be used. When it has nothing to do but wait, an operator can suspend itself and free up the worker for other processes by *deferring*. When an operator defers, execution moves to the triggerer,

where the trigger specified by the operator will run. The trigger can do the polling or waiting required by the operator. Then, when the trigger finishes polling or waiting, it sends a signal for the operator to resume its execution. During the deferred phase of execution, since work has been offloaded to the triggerer, the task no longer occupies a worker slot, and you have more free workload capacity. By default, tasks in a deferred state don't occupy pool slots. If you would like them to, you can change this by editing the pool in question.

Triggers are small, asynchronous pieces of Python code designed to run in a single Python process. Because they are asynchronous, they can all co-exist efficiently in the *triggerer* Airflow component.

An overview of how this process works:

- A task instance (running operator) reaches a point where it has to wait for other operations or conditions, and defers itself with a trigger tied to an event to resume it. This frees up the worker to run something else.
- The new trigger instance is registered by Airflow, and picked up by a triggerer process.
- The trigger runs until it fires, at which point its source task is re-scheduled by the scheduler.
- The scheduler queues the task to resume on a worker node.

You can either use pre-written deferrable operators as a DAG author or write your own. Writing them, however, requires that they meet certain design criteria.

Using Deferrable Operators

If you want to use pre-written deferrable operators that come with Airflow, such as `TimeSensorAsync`, then you only need to complete two steps:

- Ensure your Airflow installation runs at least one `triggerer` process, as well as the normal `scheduler`
- Use deferrable operators/sensors in your dags

Airflow automatically handles and implements the deferral processes for you.

If you're upgrading existing dags to use deferrable operators, Airflow contains API-compatible sensor variants, like `TimeSensorAsync` for `TimeSensor`. Add these variants into your DAG to use deferrable operators with no other changes required.

Note that you can't use the deferral ability from inside custom `PythonOperator` or `TaskFlow` Python functions. Deferral is only available to traditional, class-based operators.

Writing Deferrable Operators

When writing a deferrable operators these are the main points to consider:

- Your operator must defer itself with a trigger. You can use a trigger included in core Airflow, or you can write a custom one.
- Your operator will be stopped and removed from its worker while deferred, and no state persists automatically. You can persist state by instructing Airflow to resume the operator at a certain method or by passing certain kwargs.
- You can defer multiple times, and you can defer before or after your operator does significant work. Or, you can defer if certain conditions are met. For example, if a system does not have an immediate answer. Deferral is entirely under your control.
- Any operator can defer; no special marking on its class is needed, and it's not limited to sensors.
- If you want to add an operator or sensor that supports both deferrable and non-deferrable modes, it's suggested to add `deferrable: bool = conf.getboolean("operators", "default_deferrable", fallback=False)` to the `__init__` method of the operator and use it to decide whether to run the operator in deferrable mode. You can configure the default value of `deferrable` for all the operators and sensors that

support switching between deferrable and non-deferrable mode through `default_deferrable` in the operator section. Here's an example of a sensor that supports both modes.

```
import time
from datetime import timedelta
from typing import Any

from airflow.configuration import conf
from airflow.sdk import BaseSensorOperator
from airflow.providers.standard.triggers.temporal import TimeDeltaTrigger
from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    def __init__(self, deferrable: bool = conf.getboolean("operators", "default_deferrable", fallback=False), **kwargs):
        super().__init__(**kwargs)
        self.deferrable = deferrable

    def execute(self, context: Context) -> None:
        if self.deferrable:
            self.defer(
                trigger=TimeDeltaTrigger(timedelta(hours=1)),
                method_name="execute_complete",
            )
        else:
            time.sleep(3600)

    def execute_complete(
        self,
        context: Context,
        event: dict[str, Any] | None = None,
    ) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

Writing Triggers

A `Trigger` is written as a class that inherits from `BaseTrigger`, and implements three methods:

- `__init__`: A method to receive arguments from operators instantiating it. Since 2.10.0, we're able to start task execution directly from a pre-defined trigger. To utilize this feature, all the arguments in `__init__` must be serializable.
- `run`: An asynchronous method that runs its logic and yields one or more `TriggerEvent` instances as an asynchronous generator.
- `serialize`: Returns the information needed to re-construct this trigger, as a tuple of the classpath, and keyword arguments to pass to `__init__`.

This example shows the structure of a basic trigger, a very simplified version of Airflow's `DateTimeTrigger`:

```
import asyncio

from airflow.triggers.base import BaseTrigger, TriggerEvent
from airflow.utils import timezone

class DateTimeTrigger(BaseTrigger):
    def __init__(self, moment):
        super().__init__()
        self.moment = moment

    def serialize(self):
        return ("airflow.providers.standard.triggers.temporal.DateTimeTrigger", {"moment": self.moment})

    @asyncio.coroutine
    def run(self):
        while self.moment > timezone.utcnow():
            yield TriggerEvent(self.moment)
```

The code example shows several things:

- `__init__` and `serialize` are written as a pair. The trigger is instantiated once when it is submitted by the operator as part of its deferral request, then serialized and re-instantiated on any triggerer process that runs the trigger.
- The `run` method is declared as an `async def`, as it *must* be asynchronous, and uses `asyncio.sleep` rather than the regular `time.sleep` (because that would block the process).
- When it emits its event it packs `self.moment` in there, so if this trigger is being run redundantly on multiple hosts, the event can be de-duplicated.

Triggers can be as complex or as simple as you want, provided they meet the design constraints. They can run in a highly-available fashion, and are auto-distributed among hosts running the triggerer. We encourage you to avoid any kind of persistent state in a trigger. Triggers should get everything they need from their `__init__`, so they can be serialized and moved around freely.

If you are new to writing asynchronous Python, be very careful when writing your `run()` method. Python's `async` model means that code can block the entire process if it does not correctly `await` whenever it does a blocking operation. Airflow attempts to detect process blocking code and warn you in the triggerer logs when it happens. You can enable extra checks by Python by setting the variable `PYTHONASYNCIODEBUG=1` when you are writing your trigger to make sure you're writing non-blocking code. Be especially careful when doing filesystem calls, because if the underlying filesystem is network-backed, it can be blocking.

There's some design constraints to be aware of when writing your own trigger:

- The `run` method *must be asynchronous* (using Python's `asyncio`), and correctly `await` whenever it does a blocking operation.
- `run` must `yield` its `TriggerEvents`, not return them. If it returns before yielding at least one event, Airflow will consider this an error and fail any Task Instances waiting on it. If it throws an exception, Airflow will also fail any dependent task instances.
- You should assume that a trigger instance can run *more than once*. This can happen if a network partition occurs and Airflow re-launches a trigger on a separated machine. So, you must be mindful about side effects. For example you might not want to use a trigger to insert database rows.
- If your trigger is designed to emit more than one event (not currently supported), then each emitted event *must*

contain a payload that can be used to deduplicate events if the trigger is running in multiple places. If you only fire one event and don't need to pass information back to the operator, you can just set the payload to `None`.

- A trigger can suddenly be removed from one triggerer service and started on a new one. For example, if subnets are changed and a network partition results or if there is a deployment. If desired, you can implement the `cleanup` method, which is always called after `run`, whether the trigger exits cleanly or otherwise.
- In order for any changes to a trigger to be reflected, the *triggerer* needs to be restarted whenever the trigger is modified.
- Your trigger must not come from a dag bundle - anywhere else on `sys.path` is fine. The triggerer does not initialize any bundles when running a trigger.

Note

Currently triggers are only used until their first event, because they are only used for resuming deferred tasks, and tasks resume after the first event fires. However, Airflow plans to allow dags to be launched from triggers in future, which is where multi-event triggers will be more useful.

Sensitive information in triggers

Since Airflow 2.9.0, triggers kwargs are serialized and encrypted before being stored in the database. This means that any sensitive information you pass to a trigger will be stored in the database in an encrypted form, and decrypted when it is read from the database.

Triggering Deferral

If you want to trigger deferral, at any place in your operator, you can call `self.defer(trigger, method_name, kwargs, timeout)`. This raises a special exception for Airflow. The arguments are:

- `trigger`: An instance of a trigger that you want to defer to. It will be serialized into the database.
- `method_name`: The method name on your operator that you want Airflow to call when it resumes.
- `kwargs`: (Optional) Additional keyword arguments to pass to the method when it is called. Defaults to `{}`.
- `timeout`: (Optional) A `timedelta` that specifies a timeout after which this deferral will fail, and fail the task instance. Defaults to `None`, which means no timeout.

Here's a basic example of how a sensor might trigger deferral:

```
from __future__ import annotations

from datetime import timedelta
from typing import TYPE_CHECKING, Any

from airflow.sdk import BaseSensorOperator
from airflow.providers.standard.triggers.temporal import TimeDeltaTrigger

if TYPE_CHECKING:
    from airflow.utils.context import Context

class WaitOneHourSensor(BaseSensorOperator):
    def execute(self, context: Context) -> None:
        self.defer(trigger=TimeDeltaTrigger(timedelta(hours=1)), method_name="execute_")
```

(continues on next page)

(continued from previous page)

```

→complete")  

def execute_complete(self, context: Context, event: dict[str, Any] | None = None) ->_
→None:  

    # We have no more work to do here. Mark as complete.  

    return

```

When you opt to defer, your operator will stop executing at that point and be removed from its current worker. No state will persist, such as local variables or attributes set on `self`. When your operator resumes, it resumes as a new instance of it. The only way you can pass state from the old instance of the operator to the new one is with `method_name` and `kwargs`.

When your operator resumes, Airflow adds a `context` object and an `event` object to the `kwargs` passed to the `method_name` method. This `event` object contains the payload from the trigger event that resumed your operator. Depending on the trigger, this can be useful to your operator, like it's a status code or URL to fetch results. Or, it might be unimportant information, like a datetime. Your `method_name` method, however, *must* accept `context` and `event` as a keyword argument.

If your operator returns from either its first `execute()` method when it's new, or a subsequent method specified by `method_name`, it will be considered complete and finish executing.

Let's take a deeper look into the `WaitOneHourSensor` example above. This sensor is just a thin wrapper around the trigger. It defers to the trigger, and specifies a different method to come back to when the trigger fires. When it returns immediately, it marks the sensor as successful.

The `self.defer` call raises the `TaskDeferred` exception, so it can work anywhere inside your operator's code, even when nested many calls deep inside `execute()`. You can also raise `TaskDeferred` manually, which uses the same arguments as `self.defer`.

`execution_timeout` on operators is determined from the *total runtime*, not individual executions between deferrals. This means that if `execution_timeout` is set, an operator can fail while it's deferred or while it's running after a deferral, even if it's only been resumed for a few seconds.

Deferring multiple times

Imagine a scenario where you would like your operator to iterate over a list of items that could vary in length, and defer processing of each item.

For example, submitting multiple queries to a database, or processing multiple files.

You can set `method_name` to `execute` if you want your operator to have one entrypoint, but it must also accept `event` as an optional keyword argument.

Below is an outline of how you can achieve this.

```

import asyncio  

from airflow.sdk import BaseOperator
from airflow.triggers.base import BaseTrigger, TriggerEvent  

class MyItemTrigger(BaseTrigger):
    def __init__(self, item):
        super().__init__()
        self.item = item

```

(continues on next page)

(continued from previous page)

```

def serialize(self):
    return (self.__class__.__module__ + "." + self.__class__.__name__, {"item": self.
    ↪item})

async def run(self):
    result = None
    try:
        # Somehow process the item to calculate the result
        ...
        yield TriggerEvent({"result": result})
    except Exception as e:
        yield TriggerEvent({"error": str(e)})


class MyItemsOperator(BaseOperator):
    def __init__(self, items, **kwargs):
        super().__init__(**kwargs)
        self.items = items

    def execute(self, context, current_item_index=0, event=None):
        last_result = None
        if event is not None:
            # execute method was deferred
            if "error" in event:
                raise Exception(event["error"])
            last_result = event["result"]
            current_item_index += 1

        try:
            current_item = self.items[current_item_index]
        except IndexError:
            return last_result

        self.defer(
            trigger=MyItemTrigger(item),
            method_name="execute", # The trigger will call this same method again
            kwargs={"current_item_index": current_item_index},
        )
    )

```

Triggering Deferral from Task Start

Added in version 2.10.0.

If you want to defer your task directly to the triggerer without going into the worker, you can set class level attribute `start_from_trigger` to True and add a class level attribute `start_trigger_args` with an `StartTriggerArgs` object with the following 4 attributes to your deferrable operator:

- `trigger_cls`: An importable path to your trigger class.
- `trigger_kwargs`: Keyword arguments to pass to the `trigger_cls` when it's initialized. **Note that all the arguments need to be serializable by Airflow. It's the main limitation of this feature.**
- `next_method`: The method name on your operator that you want Airflow to call when it resumes.
- `next_kwargs`: Additional keyword arguments to pass to the `next_method` when it is called.

- `timeout`: (Optional) A timedelta that specifies a timeout after which this deferral will fail, and fail the task instance. Defaults to `None`, which means no timeout.

In the sensor part, we'll need to provide the path to `TimeDeltaTrigger` as `trigger_cls`.

```
from __future__ import annotations

from datetime import timedelta
from typing import TYPE_CHECKING, Any

from airflow.sdk import BaseSensorOperator
from airflow.triggers.base import StartTriggerArgs

if TYPE_CHECKING:
    from airflow.utils.context import Context


class WaitOneHourSensor(BaseSensorOperator):
    start_trigger_args = StartTriggerArgs(
        trigger_cls="airflow.providers.standard.triggers.temporal.TimeDeltaTrigger",
        trigger_kwargs={"moment": timedelta(hours=1)},
        next_method="execute_complete",
        next_kwargs=None,
        timeout=None,
    )
    start_from_trigger = True

    def execute_complete(self, context: Context, event: dict[str, Any] | None = None) -> None:
        # We have no more work to do here. Mark as complete.
        return
```

`start_from_trigger` and `trigger_kwargs` can also be modified at the instance level for more flexible configuration.

```
from __future__ import annotations

from datetime import timedelta
from typing import TYPE_CHECKING, Any

from airflow.sdk import BaseSensorOperator
from airflow.triggers.base import StartTriggerArgs

if TYPE_CHECKING:
    from airflow.utils.context import Context


class WaitHoursSensor(BaseSensorOperator):
    start_trigger_args = StartTriggerArgs(
        trigger_cls="airflow.providers.standard.triggers.temporal.TimeDeltaTrigger",
        trigger_kwargs={"moment": timedelta(hours=1)},
        next_method="execute_complete",
        next_kwargs=None,
        timeout=None,
    )
```

(continues on next page)

(continued from previous page)

```

start_from_trigger = True

def __init__(self, *args: list[Any], **kwargs: dict[str, Any]) -> None:
    super().__init__(*args, **kwargs)
    self.start_trigger_args.trigger_kwargs = {"hours": 2}
    self.start_from_trigger = True

    def execute_complete(self, context: Context, event: dict[str, Any] | None = None) ->_
    ~None:
        # We have no more work to do here. Mark as complete.
        return

```

The initialization stage of mapped tasks occurs after the scheduler submits them to the executor. Thus, this feature offers limited dynamic task mapping support and its usage differs from standard practices. To enable dynamic task mapping support, you need to define `start_from_trigger` and `trigger_kwargs` in the `__init__` method. **Note that you don't need to define both of them to use this feature, but you need to use the exact same parameter name.** For example, if you define an argument as `t_kwargs` and assign this value to `self.start_trigger_args.trigger_kwargs`, it will not have any effect. The entire `__init__` method will be skipped when mapping a task whose `start_from_trigger` is set to `True`. The scheduler will use the provided `start_from_trigger` and `trigger_kwargs` from `partial` and `expand` (with a fallback to the ones from class attributes if not provided) to determine whether and how to submit tasks to the executor or the triggerer. Note that XCom values won't be resolved at this stage.

After the trigger has finished executing, the task may be sent back to the worker to execute the `next_method`, or the task instance may end directly. (Refer to *Exiting deferred task from Triggers*) If the task is sent back to the worker, the arguments in the `__init__` method will still take effect before the `next_method` is executed, but they will not affect the execution of the trigger.

```

from __future__ import annotations

from datetime import timedelta
from typing import TYPE_CHECKING, Any

from airflow.sdk import BaseSensorOperator
from airflow.triggers.base import StartTriggerArgs

if TYPE_CHECKING:
    from airflow.utils.context import Context

class WaitHoursSensor(BaseSensorOperator):
    start_trigger_args = StartTriggerArgs(
        trigger_cls="airflow.providers.standard.triggers.temporal.TimeDeltaTrigger",
        trigger_kwargs={"moment": timedelta(hours=1)},
        next_method="execute_complete",
        next_kwargs=None,
        timeout=None,
    )
    start_from_trigger = True

    def __init__(
        self,
        *args: list[Any],

```

(continues on next page)

(continued from previous page)

```

trigger_kw_args: dict[str, Any] | None,
start_from_trigger: bool,
**kwargs: dict[str, Any],
) -> None:
    # This whole method will be skipped during dynamic task mapping.

    super().__init__(*args, **kwargs)
    self.start_trigger_args.trigger_kw_args = trigger_kw_args
    self.start_from_trigger = start_from_trigger

    def execute_complete(self, context: Context, event: dict[str, Any] | None = None) -> None:
        # We have no more work to do here. Mark as complete.
        return
)

```

This will be expanded into 2 tasks, with their “hours” arguments set to 1 and 2 respectively.

```

WaitHoursSensor.partial(task_id="wait_for_n_hours", start_from_trigger=True).expand(
    trigger_kw_args=[{"hours": 1}, {"hours": 2}]
)

```

Exiting deferred task from Triggers

Added in version 2.10.0.

If you want to exit your task directly from the triggerer without going into the worker, you can specify the instance level attribute `end_from_trigger` with the attributes of your deferrable operator, as discussed above. This can save some resources needed to start a new worker.

Triggers can have two options: they can either send execution back to the worker or end the task instance directly. If the trigger ends the task instance itself, the `method_name` does not matter and can be `None`. Otherwise, provide `method_name` that should be used when resuming execution in the task.

```

class WaitFiveHourSensorAsync(BaseSensorOperator):
    # this sensor always exits from trigger.
    def __init__(self, **kwargs) -> None:
        super().__init__(**kwargs)
        self.end_from_trigger = True

    def execute(self, context: Context) -> NoReturn:
        self.defer(
            method_name=None,
            trigger=WaitFiveHourTrigger(duration=timedelta(hours=5), end_from_
        trigger=self.end_from_trigger),
        )
)

```

`TaskSuccessEvent` and `TaskFailureEvent` are the two events that can be used to end the task instance directly. This marks the task with the state `task_instance_state` and optionally pushes xcom if applicable. Here’s an example of how to use these events:

```

class WaitFiveHourTrigger(BaseTrigger):
    def __init__(self, duration: timedelta, *, end_from_trigger: bool = False):
        super().__init__()
)

```

(continues on next page)

(continued from previous page)

```

self.duration = duration
self.end_from_trigger = end_from_trigger

def serialize(self) -> tuple[str, dict[str, Any]]:
    return (
        "your_module.WaitFiveHourTrigger",
        {"duration": self.duration, "end_from_trigger": self.end_from_trigger},
    )

async def run(self) -> AsyncIterator[TriggerEvent]:
    await asyncio.sleep(self.duration.total_seconds())
    if self.end_from_trigger:
        yield TaskSuccessEvent()
    else:
        yield TriggerEvent({"duration": self.duration})

```

In the above example, the trigger will end the task instance directly if `end_from_trigger` is set to True by yielding `TaskSuccessEvent`. Otherwise, it will resume the task instance with the method specified in the operator.

Note

Exiting from the trigger works only when listeners are not integrated for the deferrable operator. Currently, when deferrable operator has the `end_from_trigger` attribute set to True and listeners are integrated it raises an exception during parsing to indicate this limitation. While writing the custom trigger, ensure that the trigger is not set to end the task instance directly if the listeners are added from plugins. If the `end_from_trigger` attribute is changed to different attribute by author of trigger, the DAG parsing would not raise any exception and the listeners dependent on this task would not work. This limitation will be addressed in future releases.

High Availability

Triggers are designed to work in a high availability (HA) architecture. If you want to run a high availability setup, run multiple copies of `triggerer` on multiple hosts. Much like `scheduler`, they automatically co-exist with correct locking and HA.

Depending on how much work the triggers are doing, you can fit hundreds to tens of thousands of triggers on a single `triggerer` host. By default, every `triggerer` has a capacity of 1000 triggers that it can try to run at once. You can change the number of triggers that can run simultaneously with the `--capacity` argument. If you have more triggers trying to run than you have capacity across all of your `triggerer` processes, some triggers will be delayed from running until others have completed.

Airflow tries to only run triggers in one place at once, and maintains a heartbeat to all `triggerers` that are currently running. If a `triggerer` dies, or becomes partitioned from the network where Airflow's database is running, Airflow automatically re-schedules triggers that were on that host to run elsewhere. Airflow waits $(2.1 * \text{triggerer.job_heartbeat_sec})$ seconds for the machine to re-appear before rescheduling the triggers.

This means it's possible, but unlikely, for triggers to run in multiple places at once. This behavior is designed into the trigger contract, however, and is expected behavior. Airflow de-duplicates events fired when a trigger is running in multiple places simultaneously, so this process is transparent to your operators.

Note that every extra `triggerer` you run results in an extra persistent connection to your database.

Difference between Mode='reschedule' and Deferrable=True in Sensors

In Airflow, sensors wait for specific conditions to be met before proceeding with downstream tasks. Sensors have two options for managing idle periods: `mode='reschedule'` and `deferrable=True`. Because `mode='reschedule'` is a parameter specific to the `BaseSensorOperator` in Airflow, it allows the sensor to reschedule itself if the condition is not met. `'deferrable=True'` is a convention used by some operators to indicate that the task can be retried (or deferred) later, but it is not a built-in parameter or mode in Airflow. The actual behavior of retrying the task varies depending on the specific operator implementation.

| <code>mode='reschedule'</code> | <code>deferrable=True</code> |
|--|---|
| Continuously reschedules itself until condition is met | Pauses execution when idle, resumes when condition changes |
| Resource use is higher (repeated execution) | Resource use is lower (pauses when idle, frees up worker slots) |
| Conditions expected to change over time (e.g. file creation) | Waiting for external events or resources (e.g. API response) |
| Built-in functionality for rescheduling | Requires custom logic to defer task and handle external changes |

3.8.2 Serialization

To support data exchange, like arguments, between tasks, Airflow needs to serialize the data to be exchanged and deserialize it again when required in a downstream task. Serialization also happens so that the webserver and the scheduler (as opposed to the DAG processor) do not need to read the DAG file. This is done for security purposes and efficiency.

Serialization is a surprisingly hard job. Python out of the box only has support for serialization of primitives, like `str` and `int` and it loops over iterables. When things become more complex, custom serialization is required.

Airflow out of the box supports three ways of custom serialization. Primitives are returned as is, without additional encoding, e.g. a `str` remains a `str`. When it is not a primitive (or iterable thereof) Airflow looks for a registered serializer and deserializer in the namespace of `airflow.serialization.serializers`. If not found it will look in the class for a `serialize()` method or in case of deserialization a `deserialize(data, version: int)` method. Finally, if the class is either decorated with `@dataclass` or `@attr.define` it will use the public methods for those decorators.

If you are looking to extend Airflow with a new serializer, it is good to know when to choose what way of serialization. Objects that are under the control of Airflow, i.e. residing under the namespace of `airflow.*` like `airflow.model.dag.DAG` or under control of the developer e.g. `my.company.Foo` should first be examined to see whether they can be decorated with `@attr.define` or `@dataclass`. If that is not possible then the `serialize` and `deserialize` methods should be implemented. The `serialize` method should return a primitive or a dict. It does not need to serialize the values in the dict, that will be taken care of, but the keys should be of a primitive form.

Objects that are not under control of Airflow, e.g. `numpy.int16` will need a registered serializer and deserializer. Versioning is required. Primitives, excluding `bytes`, can be returned as dicts. Again `dict` values do not need to be serialized, but its keys need to be of primitive form. In case you are implementing a registered serializer, take special care not to have circular imports. Typically, this can be avoided by using `str` for populating the list of serializers. Like so: `serializers = ["my.company.Foo"]` instead of `serializers = [Foo]`.

Note

Serialization and deserialization is dependent on speed. Use built-in functions like `dict` as much as you can and stay away from using classes and other complex structures.

Airflow Object

```
from typing import Any, ClassVar

class Foo:
    __version__: ClassVar[int] = 1

    def __init__(self, a, v) -> None:
        self.a = a
        self.b = {"x": v}

    def serialize(self) -> dict[str, Any]:
        return {
            "a": self.a,
            "b": self.b,
        }

    @staticmethod
    def deserialize(data: dict[str, Any], version: int):
        f = Foo(a=data["a"])
        f.b = data["b"]
        return f
```

Registered

```
from __future__ import annotations

from decimal import Decimal
from typing import TYPE_CHECKING

from airflow.utils.module_loading import qualname

if TYPE_CHECKING:
    from airflow.serialization.serde import U

serializers = [
    Decimal
] # this can be a type or a fully qualified str. Str can be used to prevent circular imports
deserializers = serializers # in some cases you might not have a deserializer (e.g. k8s-pod)

__version__ = 1 # required

# the serializer expects output, classname, version, is_serialized?
def serialize(o: object) -> tuple[U, str, int, bool]:
    if isinstance(o, Decimal):
        name = qualname(o)
        _, _, exponent = o.as_tuple()
        if exponent >= 0: # No digits after the decimal point.
```

(continues on next page)

(continued from previous page)

```

return int(o), name, __version__, True
# Technically lossy due to floating point errors, but the best we
# can do without implementing a custom encode function.
return float(o), name, __version__, True

return "", "", 0, False

# the deserializer sanitizes the data for you, so you do not need to deserialize values yourself
def deserialize(classname: str, version: int, data: object) -> Decimal:
    # always check version compatibility
    if version > __version__:
        raise TypeError(f"serialized {version} of {classname} > {__version__}")

    if classname != qualname(Decimal):
        raise TypeError(f"{classname} != {qualname(Decimal)}")

    return Decimal(str(data))

```

3.8.3 Connections & Hooks

Airflow is often used to pull and push data into other systems, and so it has a first-class *Connection* concept for storing credentials that are used to talk to external systems.

A Connection is essentially set of parameters - such as username, password and hostname - along with the type of system that it connects to, and a unique name, called the `conn_id`.

They can be managed via the UI or via the CLI; see *Managing Connections* for more information on creating, editing and managing connections. There are customizable connection storage and backend options.

You can use Connections directly from your own code, you can use them via Hooks or use them from *templates*:

```
echo {{ conn.<conn_id>.host }}
```

Hooks

A Hook is a high-level interface to an external platform that lets you quickly and easily talk to them without having to write low-level code that hits their API or uses special libraries. They're also often the building blocks that Operators are built out of.

They integrate with Connections to gather credentials, and many have a default `conn_id`; for example, the `PostgresHook` automatically looks for the Connection with a `conn_id` of `postgres_default` if you don't pass one in.

You can view a *full list of Airflow hooks* in our API documentation.

Custom connections

Airflow allows to define custom connection types. This is what is described in detail in `apache-airflow-providers:index` - providers give you the capability of defining your own connections. The connection customization can be done by any provider, but also many of the providers managed by the community define custom connection types. The full list of all connections delivered by Apache Airflow community managed providers can be found in `apache-airflow-providers:core-extensions/connections`.

3.8.4 Dynamic Task Mapping

Dynamic Task Mapping allows a way for a workflow to create a number of tasks at runtime based upon current data, rather than the DAG author having to know in advance how many tasks would be needed.

This is similar to defining your tasks in a for loop, but instead of having the DAG file fetch the data and do that itself, the scheduler can do this based on the output of a previous task. Right before a mapped task is executed the scheduler will create n copies of the task, one for each input.

It is also possible to have a task operate on the collected output of a mapped task, commonly known as map and reduce.

Simple mapping

In its simplest form you can map over a list defined directly in your DAG file using the `expand()` function instead of calling your task directly.

If you want to see a simple usage of Dynamic Task Mapping, you can look below:

`airflow/example_dags/example_dynamic_task_mapping.py`

```

#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.
"""Example DAG demonstrating the usage of dynamic task mapping."""

from __future__ import annotations

from datetime import datetime

from airflow.sdk import DAG, task

with DAG(dag_id="example_dynamic_task_mapping", schedule=None, start_date=datetime(2022, 3, 4)) as dag:

    @task
    def add_one(x: int):
        return x + 1

    @task
    def sum_it(values):

```

(continues on next page)

(continued from previous page)

```
total = sum(values)
print(f"Total was {total}")

added_values = add_one.expand(x=[1, 2, 3])
sum_it(added_values)

with DAG(
    dag_id="example_task_mapping_second_order", schedule=None, catchup=False, start_
    date=datetime(2022, 3, 4)
) as dag2:

    @task
    def get_nums():
        return [1, 2, 3]

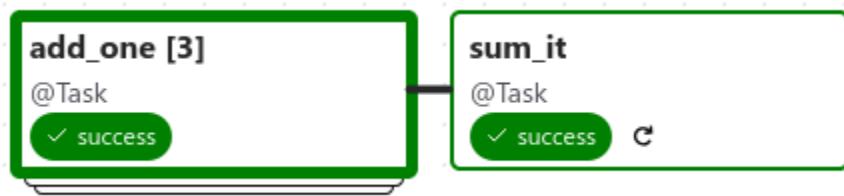
    @task
    def times_2(num):
        return num * 2

    @task
    def add_10(num):
        return num + 10

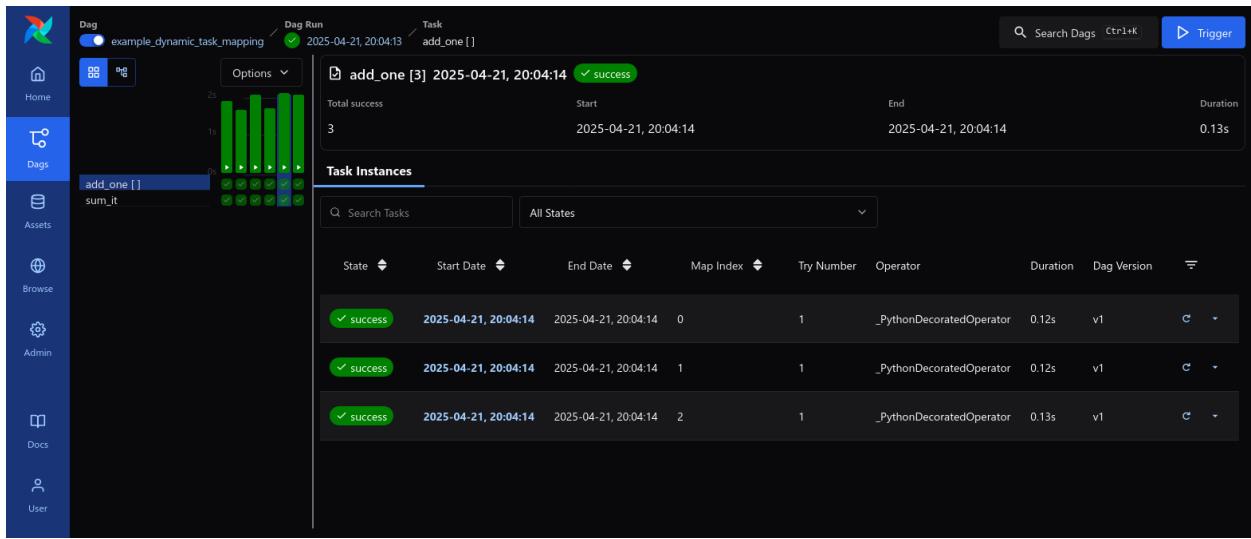
    _get_nums = get_nums()
    _times_2 = times_2.expand(num=_get_nums)
    add_10.expand(num=_times_2)
```

This will show Total was 9 in the task logs when executed.

This is the resulting DAG structure:



The grid view also provides visibility into your mapped tasks in the details panel:



Note

Only keyword arguments are allowed to be passed to `expand()`.

Note

Values passed from the mapped task is a lazy proxy

In the above example, `values` received by `sum_it` is an aggregation of all values returned by each mapped instance of `add_one`. However, since it is impossible to know how many instances of `add_one` we will have in advance, `values` is not a normal list, but a “lazy sequence” that retrieves each individual value only when asked. Therefore, if you run `print(values)` directly, you would get something like this:

```
LazySelectSequence([15 items])
```

You can use normal sequence syntax on this object (e.g. `values[0]`), or iterate through it normally with a `for` loop. `list(values)` will give you a “real” list, but since this would eagerly load values from *all* of the referenced upstream mapped tasks, you must be aware of the potential performance implications if the mapped number is large.

Note that the same also applies to when you push this proxy object into XCom. Airflow tries to be smart and coerce the value automatically, but will emit a warning for this so you are aware of this. For example:

```
@task
def forward_values(values):
    return values # This is a lazy proxy!
```

will emit a warning like this:

```
Coercing mapped lazy proxy return value from task forward_values to list, which may
→degrade
performance. Review resource requirements for this operation, and call list()
→explicitly to suppress this message. See Dynamic Task Mapping documentation for
→more information about lazy proxy objects.
```

The message can be suppressed by modifying the task like this:

```
@task
def forward_values(values):
    return list(values)
```

Note

A reduce task is not required.

Although we show a “reduce” task here (`sum_it`) you don’t have to have one, the mapped tasks will still be executed even if they have no downstream tasks.

Task-generated Mapping

The above examples we’ve shown could all be achieved with a `for` loop in the DAG file, but the real power of dynamic task mapping comes from being able to have a task generate the list to iterate over.

```
@task
def make_list():
    # This can also be from an API call, checking a database, -- almost anything you
    # like, as long as the
    # resulting list/dictionary can be stored in the current XCom backend.
    return [1, 2, {"a": "b"}, "str"]

@task
def consumer(arg):
    print(arg)

with DAG(dag_id="dynamic-map", start_date=datetime(2022, 4, 2)) as dag:
    consumer.expand(arg=make_list())
```

The `make_list` task runs as a normal task and must return a list or dict (see *What data types can be expanded?*), and then the `consumer` task will be called four times, once with each value in the return of `make_list`.

Warning

Task-generated mapping cannot be utilized with `TriggerRule.ALWAYS`

Assigning `trigger_rule=TriggerRule.ALWAYS` in task-generated mapping is not allowed, as expanded parameters are undefined with the task’s immediate execution. This is enforced at the time of the DAG parsing, for both tasks and mapped tasks groups, and will raise an error if you try to use it. In the recent example, setting `trigger_rule=TriggerRule.ALWAYS` in the `consumer` task will raise an error since `make_list` is a task-generated mapping.

Repeated mapping

The result of one mapped task can also be used as input to the next mapped task.

```
with DAG(dag_id="repeated_mapping", start_date=datetime(2022, 3, 4)) as dag:
    @task
    def add_one(x: int):
```

(continues on next page)

(continued from previous page)

```
return x + 1

first = add_one.expand(x=[1, 2, 3])
second = add_one.expand(x=first)
```

This would have a result of [3, 4, 5].

Adding parameters that do not expand

As well as passing arguments that get expanded at run-time, it is possible to pass arguments that don't change—in order to clearly differentiate between the two kinds we use different functions, `expand()` for mapped arguments, and `partial()` for unmapped ones.

```
@task
def add(x: int, y: int):
    return x + y

added_values = add.partial(y=10).expand(x=[1, 2, 3])
# This results in add function being expanded to
# add(x=1, y=10)
# add(x=2, y=10)
# add(x=3, y=10)
```

This would result in values of 11, 12, and 13.

This is also useful for passing things such as connection IDs, database table names, or bucket names to tasks.

Mapping over multiple parameters

As well as a single parameter it is possible to pass multiple parameters to expand. This will have the effect of creating a “cross product”, calling the mapped task with each combination of parameters.

```
@task
def add(x: int, y: int):
    return x + y

added_values = add.expand(x=[2, 4, 8], y=[5, 10])
# This results in the add function being called with
# add(x=2, y=5)
# add(x=2, y=10)
# add(x=4, y=5)
# add(x=4, y=10)
# add(x=8, y=5)
# add(x=8, y=10)
```

This would result in the add task being called 6 times. Please note, however, that the order of expansion is not guaranteed.

Named mapping

By default, mapped tasks are assigned an integer index. It is possible to override the integer index for each mapped task in the Airflow UI with a name based on the task's input. This is done by providing a Jinja template for the task with `map_index_template`. This will typically look like `map_index_template="{{ task.<property> }}"` when the expansion looks like `.expand(<property>=...)`. This template is rendered after each expanded task is executed using the task context. This means you can reference attributes on the task like this:

```
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator

# The two expanded task instances will be named "2024-01-01" and "2024-01-02".
SQLExecuteQueryOperator.partial(
    ...,
    sql="SELECT * FROM data WHERE date = %(date)s",
    map_index_template=""{{ task.parameters['date'] }}""",
).expand(
    parameters=[{"date": "2024-01-01"}, {"date": "2024-01-02"}],
)
```

In the above example, the expanded task instances will be named “2024-01-01” and “2024-01-02”. The names show up in the Airflow UI instead of “0” and “1”, respectively.

Since the template is rendered after the main execution block, it is possible to also dynamically inject into the rendering context. This is useful when the logic to render a desirable name is difficult to express in the Jinja template syntax, particularly in a taskflow function. For example:

```
from airflow.sdk import get_current_context

@task(map_index_template="{{ my_variable }}")
def my_task(my_value: str):
    context = get_current_context()
    context["my_variable"] = my_value * 3
    ... # Normal execution...

# The task instances will be named "aaa" and "bbb".
my_task.expand(my_value=["a", "b"])
```

Mapping with non-TaskFlow operators

It is possible to use `partial` and `expand` with classic style operators as well. Some arguments are not mappable and must be passed to `partial()`, such as `task_id`, `queue`, `pool`, and most other arguments to `BaseOperator`.

`airflow/example_dags/example_dynamic_task_mapping_with_no_taskflow_operators.py`

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
```

(continues on next page)

(continued from previous page)

```

#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.
"""Example DAG demonstrating the usage of dynamic task mapping with non-TaskFlow_
operators."""

from __future__ import annotations

from datetime import datetime

from airflow.models.baseoperator import BaseOperator
from airflow.sdk import DAG

class AddOneOperator(BaseOperator):
    """A custom operator that adds one to the input."""

    def __init__(self, value, **kwargs):
        super().__init__(**kwargs)
        self.value = value

    def execute(self, context):
        return self.value + 1

class SumItOperator(BaseOperator):
    """A custom operator that sums the input."""

    template_fields = ("values",)

    def __init__(self, values, **kwargs):
        super().__init__(**kwargs)
        self.values = values

    def execute(self, context):
        total = sum(self.values)
        print(f"Total was {total}")
        return total

with DAG(
    dag_id="example_dynamic_task_mapping_with_no_taskflow_operators",
    schedule=None,
    start_date=datetime(2022, 3, 4),
    catchup=False,
):

```

(continues on next page)

(continued from previous page)

```
# map the task to a list of values
add_one_task = AddOneOperator.partial(task_id="add_one").expand(value=[1, 2, 3])

# aggregate (reduce) the mapped tasks results
sum_it_task = SumItOperator(task_id="sum_it", values=add_one_task.output)
```

Note

Only keyword arguments are allowed to be passed to `partial()`.

Mapping over result of classic operators

If you want to map over the result of a classic operator, you should explicitly reference the `output`, instead of the operator itself.

```
# Create a list of data inputs.
extract = ExtractOperator(task_id="extract")

# Expand the operator to transform each input.
transform = TransformOperator.partial(task_id="transform").expand(input=extract.output)

# Collect the transformed inputs, expand the operator to load each one of them to the
# target.
load = LoadOperator.partial(task_id="load").expand(input=transform.output)
```

Mixing TaskFlow and classic operators

In this example, you have a regular data delivery to an S3 bucket and want to apply the same processing to every file that arrives, no matter how many arrive each time.

```
from datetime import datetime

from airflow.sdk import DAG
from airflow.sdk import task
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.providers.amazon.aws.operators.s3 import S3ListOperator


with DAG(dag_id="mapped_s3", start_date=datetime(2020, 4, 7)) as dag:
    list_filenames = S3ListOperator(
        task_id="get_input",
        bucket="example-bucket",
        prefix='incoming/provider_a/{{ data_interval_start.strftime("%Y-%m-%d") }}',
    )

    @task
    def count_lines(aws_conn_id, bucket, filename):
        hook = S3Hook(aws_conn_id=aws_conn_id)

        return len(hook.read_key(filename, bucket).splitlines())
```

(continues on next page)

(continued from previous page)

```

@task
def total(lines):
    return sum(lines)

counts = count_lines.partial(aws_conn_id="aws_default", bucket=list_filenames.
    ↪bucket).expand(
    filename=list_filenames.output
)

total(lines=counts)

```

Assigning multiple parameters to a non-TaskFlow operator

Sometimes an upstream needs to specify multiple arguments to a downstream operator. To do this, you can use the `expand_kwargs` function, which takes a sequence of mappings to map against.

```
BashOperator.partial(task_id="bash").expand_kwargs(
[
    {"bash_command": "echo $ENV1", "env": {"ENV1": "1"}},
    {"bash_command": "printf $ENV2", "env": {"ENV2": "2"}},
],
)
```

This produces two task instances at run-time printing 1 and 2 respectively.

Also it's possible to mix `expand_kwargs` with most of the operators arguments like the `op_kwargs` of the `PythonOperator`

```

def print_args(x, y):
    print(x)
    print(y)
    return x + y

PythonOperator.partial(task_id="task-1", python_callable=print_args).expand_kwargs(
[
    {"op_kwargs": {"x": 1, "y": 2}, "show_return_value_in_logs": True},
    {"op_kwargs": {"x": 3, "y": 4}, "show_return_value_in_logs": False},
]
)

```

Similar to `expand`, you can also map against a XCom that returns a list of dicts, or a list of XComs each returning a dict. Reusing the S3 example above, you can use a mapped task to perform “branching” and copy files to different buckets:

```
list_filenames = S3ListOperator(...). # Same as the above example.
```

```

@task
def create_copy_kwargs(filename):
    if filename.rsplit(".", 1)[-1] not in ("json", "yml"):
        dest_bucket_name = "my_text_bucket"
    else:

```

(continues on next page)

(continued from previous page)

```

    dest_bucket_name = "my_other_bucket"
    return {
        "source_bucket_key": filename,
        "dest_bucket_key": filename,
        "dest_bucket_name": dest_bucket_name,
    }

copy_kwargs = create_copy_kwargs.expand(filename=list_filenames.output)

# Copy files to another bucket, based on the file's extension.
copy_filenames = S3CopyObjectOperator.partial(
    task_id="copy_files", source_bucket_name=list_filenames.bucket
).expand_kwargs(copy_kwargs)

```

Mapping over a task group

Similar to a TaskFlow task, you can also call either `expand` or `expand_kwargs` on a `@task_group`-decorated function to create a mapped task group:

Note

Implementations of individual tasks in this section are omitted for brevity.

```

@task_group
def file_transforms(filename):
    return convert_to_yaml(filename)

file_transforms.expand(filename=["data1.json", "data2.json"])

```

In the above example, task `convert_to_yaml` is expanded into two task instances at runtime. The first expanded would receive "data1.json" as input, and the second "data2.json".

Value references in a task group function

One important distinction between a task function (`@task`) and a task *group* function (`@task_group`) is, since a task group does not have an associated worker, code in a task group function cannot resolve arguments passed into it; the real value and is only resolved when the reference is passed into a task.

For example, this code will *not* work:

```

@task
def my_task(value):
    print(value)

@task_group
def my_task_group(value):
    if not value: # DOES NOT work as you'd expect!
        task_a = EmptyOperator(...)

```

(continues on next page)

(continued from previous page)

```

else:
    task_a = PythonOperator(...)
    task_a << my_task(value)

my_task_group.expand(value=[0, 1, 2])

```

When code in `my_task_group` is executed, `value` would still only be a reference, not the real value, so the `if not` value branch will not work as you likely want. However, if you pass that reference into a task, it will become resolved when the task is executed, and the three `my_task` instances will therefore receive 1, 2, and 3, respectively.

It is, therefore, important to remember that, if you intend to perform any logic to a value passed into a task group function, you must always use a task to run the logic, such as `@task.branch` (or `BranchPythonOperator`) for conditions, and task mapping methods for loops.

Note

Task-mapping in a mapped task group is not permitted

It is not currently permitted to do task mapping nested inside a mapped task group. While the technical aspect of this feature is not particularly difficult, we have decided to intentionally omit this feature since it adds considerable UI complexities, and may not be necessary for general use cases. This restriction may be revisited in the future depending on user feedback.

Depth-first execution

If a mapped task group contains multiple tasks, all tasks in the group are expanded “together” against the same inputs. For example:

```

@task_group
def file_transforms(filename):
    converted = convert_to_yaml(filename)
    return replace_defaults(converted)

file_transforms.expand(filename=["data1.json", "data2.json"])

```

Since the group `file_transforms` is expanded into two, tasks `convert_to_yaml` and `replace_defaults` will each become two instances at runtime.

A similar effect can be achieved by expanding the two tasks separately like so:

```

converted = convert_to_yaml.expand(filename=["data1.json", "data2.json"])
replace_defaults.expand(filename=converted)

```

The difference, however, is that a task group allows each task inside to only depend on its “relevant inputs”. For the above example, the `replace_defaults` would only depend on `convert_to_yaml` of the same expanded group, not instances of the same task, but in a different group. This strategy, called *depth-first execution* (in contrast to the simple group-less *breadth-first execution*), allows for more logical task separation, fine-grained dependency rules, and accurate resource allocation—using the above example, the first `replace_defaults` would be able to run before `convert_to_yaml("data2.json")` is done, and does not need to care about whether it succeeds or not.

Depending on a mapped task group's output

Similar to a mapped task group, depending on a mapped task group's output would also automatically aggregate the group's results:

```
@task_group
def add_to(value):
    value = add_one(value)
    return double(value)

results = add_to.expand(value=[1, 2, 3])
consumer(results) # Will receive [4, 6, 8].
```

It is also possible to perform any operations as results from a normal mapped task.

Branching on a mapped task group's output

While it's not possible to implement branching logic (for example using `@task.branch`) on the results of a mapped task, it is possible to branch based on the *input* of a task group. The following example demonstrates executing one of three tasks based on the input to a mapped task group.

```
inputs = ["a", "b", "c"]

@task_group(group_id="my_task_group")
def my_task_group(input):
    @task.branch
    def branch(element):
        if "a" in element:
            return "my_task_group.a"
        elif "b" in element:
            return "my_task_group.b"
        else:
            return "my_task_group.c"

    @task
    def a():
        print("a")

    @task
    def b():
        print("b")

    @task
    def c():
        print("c")

    branch(input) >> [a(), b(), c()]

my_task_group.expand(input=inputs)
```

Filtering items from a mapped task

A mapped task can remove any elements from being passed on to its downstream tasks by returning `None`. For example, if we want to *only* copy files from an S3 bucket to another with certain extensions, we could implement `create_copy_kwargs` like this instead:

```
@task
def create_copy_kwargs(filename):
    # Skip files not ending with these suffixes.
    if filename.rsplit(".", 1)[-1] not in ("json", "yml"):
        return None
    return {
        "source_bucket_key": filename,
        "dest_bucket_key": filename,
        "dest_bucket_name": "my_other_bucket",
    }

# copy_kwargs and copy_files are implemented the same.
```

This makes `copy_files` only expand against `.json` and `.yml` files, while ignoring the rest.

Transforming expanding data

Since it is common to want to transform the output data format for task mapping, especially from a non-TaskFlow operator, where the output format is pre-determined and cannot be easily converted (such as `create_copy_kwargs` in the above example), a special `map()` function can be used to easily perform this kind of transformation. The above example can therefore be modified like this:

```
from airflow.exceptions import AirflowSkipException

list_filenames = S3ListOperator(...) # Unchanged.

def create_copy_kwargs(filename):
    if filename.rsplit(".", 1)[-1] not in ("json", "yml"):
        raise AirflowSkipException(f"skipping {filename}; unexpected suffix")
    return {
        "source_bucket_key": filename,
        "dest_bucket_key": filename,
        "dest_bucket_name": "my_other_bucket",
    }

copy_kwargs = list_filenames.output.map(create_copy_kwargs)

# Unchanged.
copy_filenames = S3CopyObjectOperator.partial(...).expand_kwargs(copy_kwargs)
```

There are a couple of things to note:

1. The callable argument of `map()` (`create_copy_kwargs` in the example) **must not** be a task, but a plain Python function. The transformation is as a part of the “pre-processing” of the downstream task (i.e. `copy_files`), not a standalone task in the DAG.
2. The callable always take exactly one positional argument. This function is called for each item in the iterable

used for task-mapping, similar to how Python’s built-in `map()` works.

3. Since the callable is executed as a part of the downstream task, you can use any existing techniques to write the task function. To mark a component as skipped, for example, you should raise `AirflowSkipException`. Note that returning `None` **does not** work here.

Combining upstream data (aka “zipping”)

It is also common to want to combine multiple input sources into one task mapping iterable. This is generally known as “zipping” (like Python’s built-in `zip()` function), and is also performed as pre-processing of the downstream task.

This is especially useful for conditional logic in task mapping. For example, if you want to download files from S3, but rename those files, something like this would be possible:

```
list_filenames_a = S3ListOperator(  
    task_id="list_files_in_a",  
    bucket="bucket",  
    prefix="incoming/provider_a/{{ data_interval_start|ds }}",  
)  
list_filenames_b = ["rename_1", "rename_2", "rename_3", ...]  
  
filenames_a_b = list_filenames_a.output.zip(list_filenames_b)  
  
@task  
def download_filea_from_a_rename(filenames_a_b):  
    fn_a, fn_b = filenames_a_b  
    S3Hook().download_file(fn_a, local_path=fn_b)  
  
download_filea_from_a_rename.expand(filenames_a_b=filenames_a_b)
```

Similar to the built-in `zip()`, you can zip an arbitrary number of iterables together to get an iterable of tuples of the positional arguments’ count. By default, the zipped iterable’s length is the same as the shortest of the zipped iterables, with superfluous items dropped. An optional keyword argument `default` can be passed to switch the behavior to match Python’s `itertools.zip_longest()`—the zipped iterable will have the same length as the *longest* of the zipped iterables, with missing items filled with the value provided by `default`.

Concatenating multiple upstreams

Another common pattern to combine input sources is to run the same task against multiple iterables. It is of course totally valid to simply run the same code separately for each iterable, for example:

```
list_filenames_a = S3ListOperator(  
    task_id="list_files_in_a",  
    bucket="bucket",  
    prefix="incoming/provider_a/{{ data_interval_start|ds }}",  
)  
list_filenames_b = S3ListOperator(  
    task_id="list_files_in_b",  
    bucket="bucket",  
    prefix="incoming/provider_b/{{ data_interval_start|ds }}",  
)
```

(continues on next page)

(continued from previous page)

```
@task
def download_file(filename):
    S3Hook().download_file(filename)
    # process file...

download_file.override(task_id="download_file_a").expand(filename=list_filenames_a,
    ↵output)
download_file.override(task_id="download_file_b").expand(filename=list_filenames_b,
    ↵output)
```

The DAG, however, would be both more scalable and easier to inspect if the tasks can be combined into one. This can done with `concat`:

```
# Tasks list_filenames_a and list_filenames_b, and download_file stay unchanged.

list_filenames_concat = list_filenames_a.concat(list_filenames_b)
download_file.expand(filename=list_filenames_concat)
```

This creates one single task to expand against both lists instead. You can `concat` an arbitrary number of iterables together (e.g. `foo.concat(bar, rex)`); alternatively, since the return value is also an XCom reference, the `concat` calls can be chained (e.g. `foo.concat(bar).concat(rex)`) to achieve the same result: one single iterable that concatenates all of them in order, similar to Python's `itertools.chain()`.

What data types can be expanded?

Currently it is only possible to map against a dict, a list, or one of those types stored in XCom as the result of a task.

If an upstream task returns an unmappable type, the mapped task will fail at run-time with an `UnmappableXComTypePushed` exception. For instance, you can't have the upstream task return a plain string – it must be a list or a dict.

How do templated fields and mapped arguments interact?

All arguments to an operator can be mapped, even those that do not accept templated parameters.

If a field is marked as being templated and is mapped, it **will not be templated**.

For example, this will print `{{ ds }}` and not a date stamp:

```
@task
def make_list():
    return ["{{ ds }}"]

@task
def printer(val):
    print(val)

printer.expand(val=make_list())
```

If you want to interpolate values either call `task.render_template` yourself, or use interpolation:

```
@task
def make_list(ds=None):
    return [ds]

@task
def make_list(**context):
    return [context["task"].render_template("{{ ds }}", context)]
```

Placing limits on mapped tasks

There are two limits that you can place on a task:

1. the number of mapped task instances can be created as the result of expansion.
2. The number of the mapped task can run at once.

- **Limiting number of mapped task**

The [core] `max_map_length` config option is the maximum number of tasks that `expand` can create – the default value is 1024.

If a source task (`make_list` in our earlier example) returns a list longer than this it will result in *that* task failing.

- **Limiting parallel copies of a mapped task**

If you wish to not have a large mapped task consume all available runner slots you can use the `max_active_tis_per_dag` setting on the task to restrict how many can be running at the same time.

Note, however, that this applies to all copies of that task against all active DagRuns, not just to this one specific DagRun.

```
@task(max_active_tis_per_dag=16)
def add_one(x: int):
    return x + 1

BashOperator.partial(task_id="my_task", max_active_tis_per_dag=16).expand(bash_
    command=commands)
```

Automatically skipping zero-length maps

If the input is empty (zero length), no new tasks will be created and the mapped task will be marked as SKIPPED.

3.8.5 Asset Definitions

Added in version 2.4.

Changed in version 3.0: The concept was previously called “Dataset”.

What is an “Asset”?

An Airflow asset is a logical grouping of data. Upstream producer tasks can update assets, and asset updates contribute to scheduling downstream consumer dags.

Uniform Resource Identifier (URI) define assets:

```
from airflow.sdk import Asset

example_asset = Asset("s3://asset-bucket/example.csv")
```

Airflow makes no assumptions about the content or location of the data represented by the URI, and treats the URI like a string. This means that Airflow treats any regular expressions, like `input_\d+.csv`, or file glob patterns, such as `input_2022*.csv`, as an attempt to create multiple assets from one declaration, and they will not work.

You must create assets with a valid URI. Airflow core and providers define various URI schemes that you can use, such as `file` (core), `postgres` (by the Postgres provider), and `s3` (by the Amazon provider). Third-party providers and plugins might also provide their own schemes. These pre-defined schemes have individual semantics that are expected to be followed. You can use the optional `name` argument to provide a more human-readable identifier to the asset.

```
from airflow.sdk import Asset

example_asset = Asset(uri="s3://asset-bucket/example.csv", name="bucket-1")
```

What is valid URI?

Technically, the URI must conform to the valid character set in RFC 3986, which is basically ASCII alphanumeric characters, plus %, -, _, ., and ~. To identify a resource that cannot be represented by URI-safe characters, encode the resource name with `percent-encoding`.

The URI is also case sensitive, so `s3://example/asset` and `s3://Example/asset` are considered different. Note that the `host` part of the URI is also case sensitive, which differs from RFC 3986.

For pre-defined schemes (e.g., `file`, `postgres`, and `s3`), you must provide a meaningful URI. If you can't provide one, use another scheme altogether that doesn't have the semantic restrictions. Airflow will never require a semantic for user-defined URI schemes (with a prefix `x-`), so that can be a good alternative. If you have a URI that can only be obtained later (e.g., during task execution), consider using `AssetAlias` instead and update the URI later.

```
# invalid asset:
must_contain_bucket_name = Asset("s3://")
```

Do not use the `airflow` scheme, which is reserved for Airflow's internals.

Airflow always prefers using lower cases in schemes, and case sensitivity is needed in the host part of the URI to correctly distinguish between resources.

```
# invalid assets:
reserved = Asset("airflow://example_asset")
not_ascii = Asset("èxample_datašet")
```

If you want to define assets with a scheme that doesn't include additional semantic constraints, use a scheme with the prefix `x-`. Airflow skips any semantic validation on URIs with these schemes.

```
# valid asset, treated as a plain string
my_ds = Asset("x-my-thing://foobarbaz")
```

The identifier does not have to be absolute; it can be a scheme-less, relative URI, or even just a simple path or string:

```
# valid assets:
schemeless = Asset("//example/asset")
csv_file = Asset("example_asset")
```

Non-absolute identifiers are considered plain strings that do not carry any semantic meanings to Airflow.

Extra information on asset

If needed, you can include an extra dictionary in an asset:

```
example_asset = Asset(  
    "s3://asset/example.csv",  
    extra={"team": "trainees"},  
)
```

This can be used to supply custom description to the asset, such as who has ownership to the target file, or what the file is for. The extra information does not affect an asset's identity.

Note

Security Note: Asset URI and extra fields are not encrypted, they are stored in cleartext in Airflow's metadata database. Do NOT store any sensitive values, especially credentials, in either asset URIs or extra key values!

Creating a task to emit asset events

Once an asset is defined, tasks can be created to emit events against it by specifying outlets:

```
from airflow.sdk import DAG, Asset  
from airflow.providers.standard.operators.python import PythonOperator  
  
example_asset = Asset(name="example_asset", uri="s3://asset-bucket/example.csv")  
  
def _write_example_asset():  
    """Write data to example_asset..."""  
  
with DAG(dag_id="example_asset", schedule="@daily"):  
    PythonOperator(task_id="example_asset", outlets=[example_asset], python_callable=_  
    write_example_asset)
```

This is quite a lot of boilerplate. Airflow provides a shorthand for this simple but most common case of *creating a DAG with one single task that emits events of one asset*. The code block below is exactly equivalent to the one above:

```
from airflow.sdk import asset  
  
@asset(uri="s3://asset-bucket/example.csv", schedule="@daily")  
def example_asset():  
    """Write data to example_asset..."""
```

Declaring an `@asset` automatically creates:

- An `Asset` with `name` set to the function name.
- A `DAG` with `dag_id` set to the function name.
- A task inside the `DAG` with `task_id` set to the function name, and `outlet` to the created `Asset`.

Attaching extra information to an emitting asset event

Added in version 2.10.0.

A task with an asset outlet can optionally attach extra information before it emits an asset event. This is different from *Extra information on asset*. Extra information on an asset statically describes the entity pointed to by the asset URI; extra information on the *asset event* instead should be used to annotate the triggering data change, such as how many rows in the database are changed by the update, or the date range covered by it.

The easiest way to attach extra information to the asset event is by `yield`-ing a `Metadata` object from a task:

```
from airflow.sdk import Metadata, asset

@asset(uri="s3://asset/example.csv", schedule=None)
def example_s3(self): # 'self' here refers to the current asset.
    df = ... # Get a Pandas DataFrame to write.
    # Write df to asset...
    yield Metadata(self, {"row_count": len(df)})
```

Airflow automatically collects all yielded metadata, and populates asset events with extra information for corresponding metadata objects.

This can also be done in classic operators. The best way is to subclass the operator and override `execute`. Alternatively, extras can also be added in a task's `pre_execute` or `post_execute` hook. If you choose to use hooks, however, remember that they are not rerun when a task is retried, and may cause the extra information to not match actual data in certain scenarios.

Another way to achieve the same is by accessing `outlet_events` in a task's execution context directly:

```
@asset(schedule=None)
def write_to_s3(self, context):
    context["outlet_events"][_self].extra = {"row_count": len(df)}
```

There's minimal magic here—Airflow simply writes the yielded values to the exact same accessor. This also works in classic operators, including `execute`, `pre_execute`, and `post_execute`.

Fetching information from previously emitted asset events

Added in version 2.10.0.

Events of an asset defined in a task's `outlets`, as described in the previous section, can be read by a task that declares the same asset in its `inlets`. A asset event entry contains `extra` (see previous section for details), `timestamp` indicating when the event was emitted from a task, and `source_task_instance` linking the event back to its source.

Inlet asset events can be read with the `inlet_events` accessor in the execution context. Continuing from the `write_to_s3` asset in the previous section:

```
@asset(schedule=None)
def post_process_s3_file(context, write_to_s3): # Declaring an inlet to write_to_s3.
    events = context["inlet_events"][write_to_s3]
    last_row_count = events[-1].extra["row_count"]
```

Each value in the `inlet_events` mapping is a sequence-like object that orders past events of a given asset by `timestamp`, earliest to latest. It supports most of Python's list interface, so you can use `[-1]` to access the last event, `[-2:]` for the last two, etc. The accessor is lazy and only hits the database when you access items inside it.

Dependency between @asset, @task, and classic operators

Since an `@asset` is simply a wrapper around a dag with a task and an asset, it is quite easy to read and `@asset` in a `@task` or a classic operator. For example, the above `post_process_s3_file` can also be written as a task (inside a dag, omitted here for brevity):

```
@task(inlets=[write_to_s3])
def post_process_s3_file(*, inlet_events):
    events = inlet_events[example_s3_asset]
    last_row_count = events[-1].extra["row_count"]

post_process_s3_file()
```

The other way around also applies:

```
example_asset = Asset("example_asset")

@task(outlets=[example_asset])
def emit_example_asset():
    """Write to example_asset..."""

@asset(schedule=None)
def process_example_asset(example_asset):
    """Process inlet example_asset..."""
```

Output to multiple assets in one task

It is possible for a task to emit events for multiple assets. This is generally discouraged, but needed in certain situations, such as when you need to split a data source into several. This is straightforward with tasks since `outlets` is plural by design:

```
from airflow.sdk import DAG, Asset, task

input_asset = Asset("input_asset")
out_asset_1 = Asset("out_asset_1")
out_asset_2 = Asset("out_asset_2")

with DAG(dag_id="process_input", schedule=None):
    @task(inlets=[input_asset], outlets=[out_asset_1, out_asset_2])
    def process_input():
        """Split input into two."""
```

The shorthand for this is `@asset.multi`:

```
from airflow.sdk import Asset, asset

input_asset = Asset("input_asset")
out_asset_1 = Asset("out_asset_1")
out_asset_2 = Asset("out_asset_2")
```

(continues on next page)

(continued from previous page)

```
@asset.multi(schedule=None, outlets=[out_asset_1, out_asset_2])
def process_input(input_asset):
    """Split input into two."""

```

Dynamic data events emitting and asset creation through AssetAlias

An asset alias can be used to emit asset events of assets with association to the aliases. Downstreams can depend on resolved asset. This feature allows you to define complex dependencies for DAG executions based on asset updates.

How to use AssetAlias

AssetAlias has one single argument `name` that uniquely identifies the asset. The task must first declare the alias as an outlet, and use `outlet_events` or `yield Metadata` to add events to it.

The following example creates an asset event against the S3 URI `f"s3://bucket/my-task"` with optional extra information `extra`. If the asset does not exist, Airflow will dynamically create it and log a warning message.

Emit an asset event during task execution through `outlet_events`

```
from airflow.sdk import AssetAlias

@task(outlets=[AssetAlias("my-task-outputs")])
def my_task_with_outlet_events(*, outlet_events):
    outlet_events[AssetAlias("my-task-outputs")].add(Asset("s3://bucket/my-task"), extra=
→{"k": "v"})
```

Emit an asset event during task execution through yielding Metadata

```
from airflow.sdk import Metadata

@task(outlets=[AssetAlias("my-task-outputs")])
def my_task_with_metadata():
    s3_asset = Asset(uri="s3://bucket/my-task", name="example_s3")
    yield Metadata(s3_asset, extra={"k": "v"}, alias="my-task-outputs")
```

Only one asset event is emitted for an added asset, even if it is added to the alias multiple times, or added to multiple aliases. However, if different `extra` values are passed, it can emit multiple asset events. In the following example, two asset events will be emitted.

```
from airflow.sdk import AssetAlias

@task(
    outlets=[
        AssetAlias("my-task-outputs-1"),
        AssetAlias("my-task-outputs-2"),
        AssetAlias("my-task-outputs-3"),
    ]
)
def my_task_with_outlet_events(*, outlet_events):
```

(continues on next page)

(continued from previous page)

```

outlet_events[AssetAlias("my-task-outputs-1")].add(Asset("s3://bucket/my-task"),_
↪extra={"k": "v"})
    # This line won't emit an additional asset event as the asset and extra are the same_
↪as the previous line.
outlet_events[AssetAlias("my-task-outputs-2")].add(Asset("s3://bucket/my-task"),_
↪extra={"k": "v"})
    # This line will emit an additional asset event as the extra is different.
outlet_events[AssetAlias("my-task-outputs-3")].add(Asset("s3://bucket/my-task"),_
↪extra={"k2": "v2"})

```

Fetching information from previously emitted asset events through resolved asset aliases

As mentioned in *Fetching information from previously emitted asset events*, inlet asset events can be read with the `inlet_events` accessor in the execution context, and you can also use asset aliases to access the asset events triggered by them.

```

with DAG(dag_id="asset-alias-producer"):

    @task(outlets=[AssetAlias("example-alias")])
    def produce_asset_events(*, outlet_events):
        outlet_events[AssetAlias("example-alias")].add(Asset("s3://bucket/my-task"),_
↪extra={"row_count": 1})

with DAG(dag_id="asset-alias-consumer", schedule=None):

    @task(inlets=[AssetAlias("example-alias")])
    def consume_asset_alias_events(*, inlet_events):
        events = inlet_events[AssetAlias("example-alias")]
        last_row_count = events[-1].extra["row_count"]

```

Scheduling

3.8.6 Cron & Time Intervals

You may set your DAG to run on a simple schedule by setting its `schedule` argument to either a cron expression, a `datetime.timedelta` object, or one of the *Cron Presets*.

```

from airflow.sdk import DAG

import datetime

dag = DAG("regular_interval_cron_example", schedule="@0 0 * * *", ...)
dag = DAG("regular_interval_cron_preset_example", schedule="@daily", ...)
dag = DAG("regular_interval_timedelta_example", schedule=datetime.timedelta(days=1), ...)

```

Cron Presets

For more elaborate scheduling requirements, you can implement a *custom timetable*. Note that Airflow parses cron expressions with the croniter library which supports an extended syntax for cron strings. See their documentation [in github](#). For example, you can create a DAG schedule to run at 12AM on the first Monday of the month with their extended cron syntax: `0 0 * * MON#1`.

Tip

You can use an online editor for CRON expressions such as [Crontab guru](#)

| preset | meaning | cron |
|-------------|--|-------------|
| None | Don't schedule, use for exclusively "externally triggered" dags | |
| @once | Schedule once and only once | |
| @continuous | Run as soon as the previous run finishes | |
| @hourly | Run once an hour at the end of the hour | 0 * * * * |
| @daily | Run once a day at midnight (24:00) | 0 0 * * * |
| @weekly | Run once a week at midnight (24:00) on Sunday | 0 0 * * 0 |
| @monthly | Run once a month at midnight (24:00) of the first day of the month | 0 0 1 * * |
| @quarterly | Run once a quarter at midnight (24:00) on the first day | 0 0 1 */3 * |
| @yearly | Run once a year at midnight (24:00) of January 1 | 0 0 1 1 * |

Your DAG will be instantiated for each schedule along with a corresponding DAG Run entry in the database backend.

3.8.7 Time Zones

Support for time zones is enabled by default. Airflow stores datetime information in UTC internally and in the database. It allows you to run your dags with time zone dependent schedules. At the moment, Airflow does not convert them to the end user's time zone in the user interface. It will always be displayed in UTC there. Also, templates used in Operators are not converted. Time zone information is exposed and it is up to the writer of DAG to decide what do with it.

This is handy if your users live in more than one time zone and you want to display datetime information according to each user's wall clock.

Even if you are running Airflow in only one time zone, it is still good practice to store data in UTC in your database (also before Airflow became time zone aware this was also the recommended or even required setup). The main reason is that many countries use Daylight Saving Time (DST), where clocks are moved forward in spring and backward in autumn. If you're working in local time, you're likely to encounter errors twice a year, when the transitions happen. (The pendulum and pytz documentation discuss these issues in greater detail.) This probably doesn't matter for a simple DAG, but it's a problem if you are in, for example, financial services where you have end of day deadlines to meet.

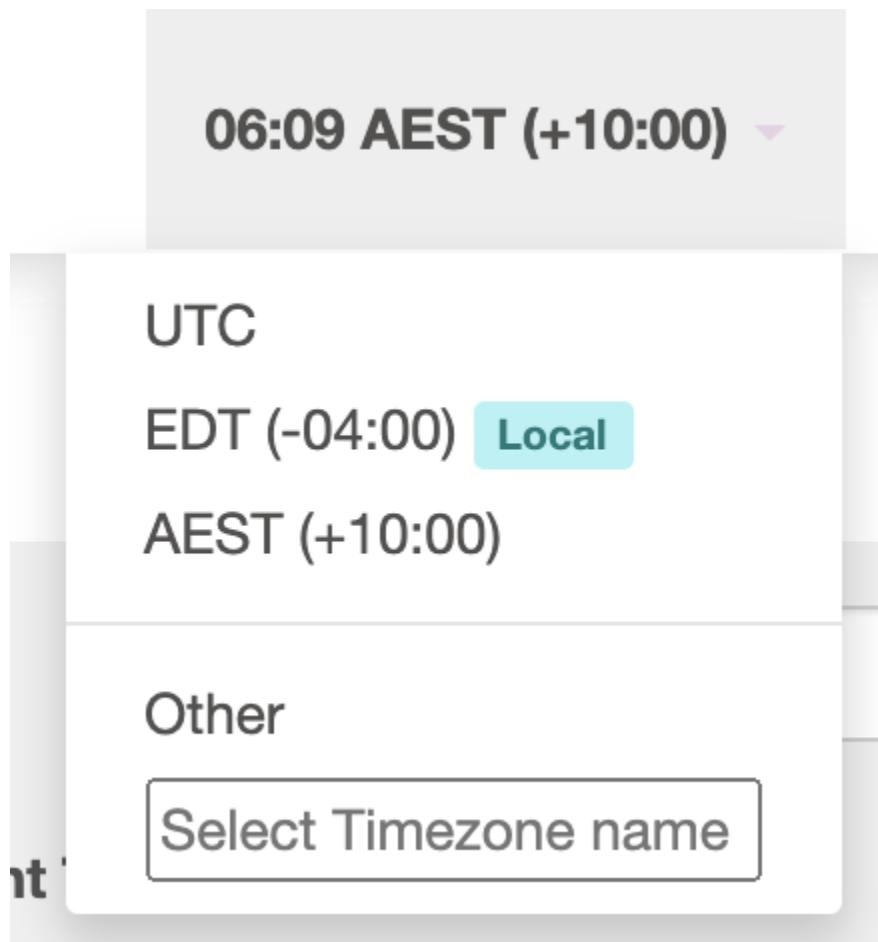
The time zone is set in `airflow.cfg`. By default it is set to UTC, but you change it to use the system's settings or an arbitrary IANA time zone, e.g. `Europe/Amsterdam`. It is dependent on `pendulum`, which is more accurate than `pytz`. Pendulum is installed when you install Airflow.

Note

Pendulum relies by default on its own timezone database, which is not updated as frequently as the IANA database. You can make Pendulum rely on the system's database by setting the `PYTZDATA_TZDATADIR` environment variable to your system's database, e.g. `/usr/share/zoneinfo`.

Web UI

By default the Web UI will show times in UTC. It is possible to change the timezone shown by using the menu in the top right (click on the clock to activate it):



“Local” is detected from the browser’s timezone. The “Server” value comes from the `default_timezone` setting in the `[core]` section.

The users’ selected timezone is stored in LocalStorage so is a per-browser setting.

Note

The UI will use the system timezone by default. Users can change their timezone preferences through the UI’s timezone selector.

Concepts

Naive and aware datetime objects

Python’s `datetime.datetime` objects have a `tzinfo` attribute that can be used to store time zone information, represented as an instance of a subclass of `datetime.tzinfo`. When this attribute is set and describes an offset, a `datetime` object is aware. Otherwise, it’s naive.

You can use `timezone.is_localized()` and `timezone.is_naive()` to determine whether datetimes are aware or naive.

Because Airflow uses time zone aware datetime objects. If your code creates datetime objects they need to be aware too.

```
from airflow.utils import timezone

now = timezone.utcnow()
a_date = timezone.datetime(2017, 1, 1)
```

Interpretation of naive datetime objects

Although Airflow operates fully time zone aware, it still accepts naive date time objects for `start_dates` and `end_dates` in your DAG definitions. This is mostly in order to preserve backwards compatibility. In case a naive `start_date` or `end_date` is encountered the default time zone is applied. It is applied in such a way that it is assumed that the naive date time is already in the default time zone. In other words if you have a default time zone setting of Europe/Amsterdam and create a naive datetime `start_date` of `datetime(2017, 1, 1)` it is assumed to be a `start_date` of Jan 1, 2017 Amsterdam time.

```
dag = DAG(
    "my_dag",
    start_date=pendulum.datetime(2017, 1, 1, tz="UTC"),
    default_args={"retries": 3},
)
op = BashOperator(task_id="hello_world", bash_command="Hello World!", dag=dag)
print(op.retries) # 3
```

Unfortunately, during DST transitions, some datetimes don't exist or are ambiguous. In such situations, pendulum raises an exception. That's why you should always create aware datetime objects when time zone support is enabled.

In practice, this is rarely an issue. Airflow gives you time zone aware datetime objects in the models and dags, and most often, new datetime objects are created from existing ones through timedelta arithmetic. The only datetime that's often created in application code is the current time, and `timezone.utcnow()` automatically does the right thing.

Default time zone

The default time zone is the time zone defined by the `default_timezone` setting under `[core]`. If you just installed Airflow it will be set to `utc`, which is recommended. You can also set it to `system` or an IANA time zone (e.g. Europe/Amsterdam). Dags are also evaluated on Airflow workers, it is therefore important to make sure this setting is equal on all Airflow nodes.

```
[core]
default_timezone = utc
```

Note

For more information on setting the configuration, see *Setting Configuration Options*

Time zone aware dags

Creating a time zone aware DAG is quite simple. Just make sure to supply a time zone aware `start_date` using `pendulum`. Don't try to use standard library `timezone` as they are known to have limitations and we deliberately disallow using them in dags.

```
import pendulum

dag = DAG("my_tz_dag", start_date=pendulum.datetime(2016, 1, 1, tz="Europe/Amsterdam"))
op = EmptyOperator(task_id="empty", dag=dag)
print(dag.timezone) # <Timezone [Europe/Amsterdam]>
```

Please note that while it is possible to set a `start_date` and `end_date` for Tasks, the DAG timezone or global timezone (in that order) will always be used to calculate data intervals. Upon first encounter, the start date or end date will be converted to UTC using the timezone associated with `start_date` or `end_date`, then for calculations this timezone information will be disregarded.

Note

When authoring a Timezone aware DAG you must make sure that the underlying timezone library (for example: `pendulum`) is updated with recent changes to regulations (daylight saving changes etc...). When a change in time is expected you should verify with the underlying timezone library that the switch will happen as expected. There might be a need to update the library version. As a general recommendation if you can author dags in UTC that is preferred.

Templates

Airflow returns time zone aware datetimes in templates, but does not convert them to local time so they remain in UTC. It is left up to the DAG to handle this.

```
import pendulum

local_tz = pendulum.timezone("Europe/Amsterdam")
local_tz.convert(logical_date)
```

Cron schedules

Time zone aware dags that use cron schedules respect daylight savings time. For example, a DAG with a start date in the US/Eastern time zone with a schedule of `0 0 * * *` will run daily at 04:00 UTC during daylight savings time and at 05:00 otherwise.

Time deltas

Time zone aware dags that use `timedelta` or `relativedelta` schedules respect daylight savings time for the start date but do not adjust for daylight savings time when scheduling subsequent runs. For example, a DAG with a start date of `pendulum.datetime(2020, 1, 1, tz="UTC")` and a schedule interval of `timedelta(days=1)` will run daily at 05:00 UTC regardless of daylight savings time.

3.8.8 Asset-Aware Scheduling

Added in version 2.4.

Quickstart

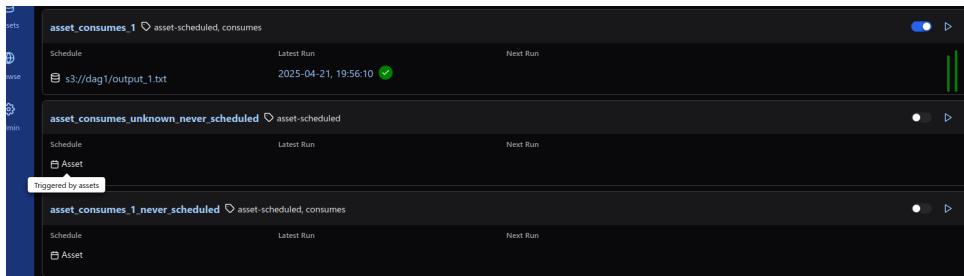
In addition to scheduling dags based on time, you can also schedule dags to run based on when a task updates an asset.

```
from airflow.sdk import DAG, Asset

with DAG(...):
    MyOperator(
        # this task updates example.csv
        outlets=[Asset("s3://asset-bucket/example.csv")],
        ...
    )

with DAG(
    # this DAG should be run when example.csv is updated (by dag1)
    schedule=[Asset("s3://asset-bucket/example.csv")],
    ...
):
    ...

```



See also

[Asset Definitions](#) for how to declare assets.

Schedule dags with assets

You can use assets to specify data dependencies in your dags. The following example shows how after the producer task in the producer DAG successfully completes, Airflow schedules the consumer DAG. Airflow marks an asset as updated only if the task completes successfully. If the task fails or if it is skipped, no update occurs, and Airflow doesn't schedule the consumer DAG.

```
example_asset = Asset("s3://asset/example.csv")

with DAG(dag_id="producer", ...):
    BashOperator(task_id="producer", outlets=[example_asset], ...)

with DAG(dag_id="consumer", schedule=[example_asset], ...):
    ...

```

You can find a listing of the relationships between assets and dags in the *Asset Views*.

Multiple assets

Because the `schedule` parameter is a list, dags can require multiple assets. Airflow schedules a DAG after **all** assets the DAG consumes have been updated at least once since the last time the DAG ran:

```
with DAG(
    dag_id="multiple_assets_example",
    schedule=[
        example_asset_1,
        example_asset_2,
        example_asset_3,
    ],
    ...
):  
    ...
```

If one asset is updated multiple times before all consumed assets update, the downstream DAG still only runs once, as shown in this illustration:

Fetching information from a triggering asset event

A triggered DAG can fetch information from the asset that triggered it using the `triggering_asset_events` template or parameter. See more at *Templates reference*.

Example:

```
example_snowflake_asset = Asset("snowflake://my_db/my_schema/my_table")

with DAG(dag_id="load_snowflake_data", schedule="@hourly", ...):
    SQLExecuteQueryOperator(
        task_id="load", conn_id="snowflake_default", outlets=[example_snowflake_asset], .
    )
    ...

with DAG(dag_id="query_snowflake_data", schedule=[example_snowflake_asset], ...):
    SQLExecuteQueryOperator(
        task_id="query",
        conn_id="snowflake_default",
        sql="""
            SELECT *
            FROM my_db.my_schema.my_table
            WHERE "updated_at" >= '{{ (triggering_asset_events.values() | first | first).source_dag_run.data_interval_start }}'
            AND "updated_at" < '{{ (triggering_asset_events.values() | first | first).source_dag_run.data_interval_end }}';
        """,
    )

@task
def print_triggering_asset_events(triggering_asset_events=None):
    for asset, asset_list in triggering_asset_events.items():
        print(asset, asset_list)
        print(asset_list[0].source_dag_run.dag_id)
```

(continues on next page)

(continued from previous page)

```
print_triggering_asset_events()
```

Note that this example is using `(.values() | first | first)` to fetch the first of one asset given to the DAG, and the first of one AssetEvent for that asset. An implementation can be quite complex if you have multiple assets, potentially with multiple AssetEvents.

Manipulating queued asset events through REST API

Added in version 2.9.

In this example, the DAG `waiting_for_asset_1_and_2` will be triggered when tasks update both assets “asset-1” and “asset-2”. Once “asset-1” is updated, Airflow creates a record. This ensures that Airflow knows to trigger the DAG when “asset-2” is updated. We call such records queued asset events.

```
with DAG(
    dag_id="waiting_for_asset_1_and_2",
    schedule=[Asset("asset-1"), Asset("asset-2")],
    ...
):
```

queuedEvent API endpoints are introduced to manipulate such records.

- Get a queued asset event for a DAG: `/assets/queuedEvent/{uri}`
- Get queued asset events for a DAG: `/dags/{dag_id}/assets/queuedEvent`
- Delete a queued asset event for a DAG: `/assets/queuedEvent/{uri}`
- Delete queued asset events for a DAG: `/dags/{dag_id}/assets/queuedEvent`
- Get queued asset events for an asset: `/dags/{dag_id}/assets/queuedEvent/{uri}`
- Delete queued asset events for an asset: `DELETE /dags/{dag_id}/assets/queuedEvent/{uri}`

For how to use REST API and the parameters needed for these endpoints, please refer to *Airflow API*.

Advanced asset scheduling with conditional expressions

Apache Airflow includes advanced scheduling capabilities that use conditional expressions with assets. This feature allows you to define complex dependencies for DAG executions based on asset updates, using logical operators for more control on workflow triggers.

Logical operators for assets

Airflow supports two logical operators for combining asset conditions:

- **AND** (`^&^`): Specifies that the DAG should be triggered only after all of the specified assets have been updated.
- **OR** (`^|^\code>): Specifies that the DAG should be triggered when any of the specified assets is updated.`

These operators enable you to configure your Airflow workflows to use more complex asset update conditions, making them more dynamic and flexible.

Example Use

Scheduling based on multiple asset updates

To schedule a DAG to run only when two specific assets have both been updated, use the AND operator (&):

```
dag1_asset = Asset("s3://dag1/output_1.txt")
dag2_asset = Asset("s3://dag2/output_1.txt")

with DAG(
    # Consume asset 1 and 2 with asset expressions
    schedule=(dag1_asset & dag2_asset),
    ...,
):
    ...
```

Scheduling based on any asset update

To trigger a DAG execution when either one of two assets is updated, apply the OR operator (|):

```
with DAG(
    # Consume asset 1 or 2 with asset expressions
    schedule=(dag1_asset | dag2_asset),
    ...,
):
    ...
```

Complex Conditional Logic

For scenarios requiring more intricate conditions, such as triggering a DAG when one asset is updated or when both of two other assets are updated, combine the OR and AND operators:

```
dag3_asset = Asset("s3://dag3/output_3.txt")

with DAG(
    # Consume asset 1 or both 2 and 3 with asset expressions
    schedule=(dag1_asset | (dag2_asset & dag3_asset)),
    ...,
):
    ...
```

Scheduling based on asset aliases

Since asset events added to an alias are just simple asset events, a downstream DAG depending on the actual asset can read asset events of it normally, without considering the associated aliases. A downstream DAG can also depend on an asset alias. The authoring syntax is referencing the `AssetAlias` by name, and the associated asset events are picked up for scheduling. Note that a DAG can be triggered by a task with `outlets=AssetAlias("xxx")` if and only if the alias is resolved into `Asset("s3://bucket/my-task")`. The DAG runs whenever a task with outlet `AssetAlias("out")` gets associated with at least one asset at runtime, regardless of the asset's identity. The downstream DAG is not triggered if no assets are associated to the alias for a particular given task run. This also means we can do conditional asset-triggering.

The asset alias is resolved to the assets during DAG parsing. Thus, if the “`min_file_process_interval`” configuration is set to a high value, there is a possibility that the asset alias may not be resolved. To resolve this issue, you can trigger DAG parsing.

```

with DAG(dag_id="asset-producer"):

    @task(outlets=[Asset("example-alias")])
    def produce_asset_events():
        pass


with DAG(dag_id="asset-alias-producer"):

    @task(outlets=[AssetAlias("example-alias")])
    def produce_asset_events(*, outlet_events):
        outlet_events[AssetAlias("example-alias")].add(Asset("s3://bucket/my-task"))

with DAG(dag_id="asset-consumer", schedule=Asset("s3://bucket/my-task")):
    ...

with DAG(dag_id="asset-alias-consumer", schedule=AssetAlias("example-alias")):
    ...

```

In the example provided, once the DAG asset-alias-producer is executed, the asset alias AssetAlias("example-alias") will be resolved to Asset("s3://bucket/my-task"). However, the DAG asset-alias-consumer will have to wait for the next DAG re-parsing to update its schedule. To address this, Airflow will re-parse the dags relying on the asset alias AssetAlias("example-alias") when it's resolved into assets that these dags did not previously depend on. As a result, both the “asset-consumer” and “asset-alias-consumer” dags will be triggered after the execution of DAG asset-alias-producer.

Combining asset and time-based schedules

AssetTimetable Integration

You can schedule dags based on both asset events and time-based schedules using `AssetOrTimeSchedule`. This allows you to create workflows when a DAG needs both to be triggered by data updates and run periodically according to a fixed timetable.

For more detailed information on `AssetOrTimeSchedule`, refer to the corresponding section in `AssetOrTimeSchedule`.

3.8.9 Timetables

For a DAG with a time-based schedule (as opposed to event-driven), the DAG’s internal “timetable” drives scheduling. The timetable also determines the data interval and the logical date of each run created for the DAG.

Dags scheduled with a cron expression or `timedelta` object are internally converted to always use a timetable.

If a cron expression or `timedelta` is sufficient for your use case, you don’t need to worry about writing a custom timetable because Airflow has default timetables that handle those cases. But for more complicated scheduling requirements, you can create your own timetable class and pass that to the DAG’s `schedule` argument.

Some examples of when custom timetable implementations are useful:

- Task runs that occur at different times each day. For example, an astronomer might find it useful to run a task at dawn to process data collected from the previous night-time period.
- Schedules that don’t follow the Gregorian calendar. For example, create a run for each month in the [Traditional Chinese Calendar](#). This is conceptually similar to the sunrise case, but for a different time scale.

- Rolling windows, or overlapping data intervals. For example, you might want to have a run each day, but make each run cover the period of the previous seven days. It is possible to hack this with a cron expression, but a custom data interval provides a more natural representation.
- Data intervals with “holes” between intervals instead of a continuous interval, as both the cron expression and `timedelta` schedules represent continuous intervals. See *Data Interval*.

Airflow allows you to write custom timetables in plugins and used by dags. You can find an example demonstrating a custom timetable in the *Customizing DAG Scheduling with Timetables* how-to guide.

Note

As a general rule, always access Variables, Connections, or anything else that needs access to the database as late as possible in your code. See *Timetables* for more best practices to follow.

Built-in Timetables

Airflow comes with several common timetables built-in to cover the most common use cases. Additional timetables may be available in plugins.

DeltaTriggerTimetable

A timetable that accepts a `datetime.timedelta` or `dateutil.relativedelta.relativedelta`, and runs the DAG once a delta passes.

See also

Differences between “trigger” and “data interval” timetables

```
from datetime import timedelta

from airflow.timetables.trigger import DeltaTriggerTimetable

@dag(schedule=DeltaTriggerTimetable(timedelta(days=7)), ...) # Once every week.
def example_dag():
    pass
```

You can also provide a static data interval to the timetable. The optional `interval` argument also should be a `datetime.timedelta` or `dateutil.relativedelta.relativedelta`. When using these arguments, a triggered DAG run’s data interval spans the specified duration, and *ends* with the trigger time.

```
from datetime import UTC, datetime, timedelta

from dateutil.relativedelta import relativedelta, FR

from airflow.timetables.trigger import DeltaTriggerTimetable

@dag(
    # Runs every Friday at 18:00 to cover the work week.
    schedule=DeltaTriggerTimetable(
```

(continues on next page)

(continued from previous page)

```

relativeedelta(weekday=FR(), hour=18),
interval=timedelta(days=4, hours=9),
),
start_date=datetime(2025, 1, 3, 18, tzinfo=UTC),
...,
)
def example_dag():
    pass

```

CronTriggerTimetable

A timetable that accepts a cron expression, and triggers DAG runs according to it.

See also

Differences between “trigger” and “data interval” timetables

```

from airflow.timetables.trigger import CronTriggerTimetable

@dag(schedule=CronTriggerTimetable("0 1 * * 3", timezone="UTC"), ...) # At 01:00 on
˓→Wednesday
def example_dag():
    pass

```

You can also provide a static data interval to the timetable. The optional `interval` argument must be a `datetime.timedelta` or `dateutil.relativedelta.relativedelta`. When using these arguments, a triggered DAG run’s data interval spans the specified duration, and *ends* with the trigger time.

```

from datetime import timedelta

from airflow.timetables.trigger import CronTriggerTimetable

@dag(
    # Runs every Friday at 18:00 to cover the work week (9:00 Monday to 18:00 Friday).
    schedule=CronTriggerTimetable(
        "0 18 * * 5",
        timezone="UTC",
        interval=timedelta(days=4, hours=9),
    ),
    ...,
)
def example_dag():
    pass

```

MultipleCronTriggerTimetable

This is similar to *CronTriggerTimetable* except it takes multiple cron expressions. A DAG run is scheduled whenever any of the expressions matches the time. It is particularly useful when the desired schedule cannot be expressed by one single cron expression.

```
from airflow.timetables.trigger import MultipleCronTriggerTimetable

# At 1:10 and 2:40 each day.
@dag(schedule=MultipleCronTriggerTimetable("10 1 * * *", "40 2 * * *", timezone="UTC"), .
     .)
def example_dag():
    pass
```

The same optional `interval` argument as *CronTriggerTimetable* is also available.

```
from datetime import timedelta

from airflow.timetables.trigger import MultipleCronTriggerTimetable

@dag(
    schedule=MultipleCronTriggerTimetable(
        "10 1 * * *",
        "40 2 * * *",
        timezone="UTC",
        interval=timedelta(hours=1),
    ),
    . . .,
)
def example_dag():
    pass
```

DeltaDataIntervalTimetable

A timetable that schedules data intervals with a time delta. You can select it by providing a `datetime.timedelta` or `dateutil.relativedelta.relativedelta` to the `schedule` parameter of a DAG.

This timetable focuses on the data interval value and does not necessarily align execution dates with arbitrary bounds, such as the start of day or of hour.

See also

Differences between the cron and delta data interval timetables

```
@dag(schedule=datetime.timedelta(minutes=30))
def example_dag():
    pass
```

CronDataIntervalTimetable

A timetable that accepts a cron expression, creates data intervals according to the interval between each cron trigger points, and triggers a DAG run at the end of each data interval.

See also

Differences between “trigger” and “data interval” timetables

See also

Differences between the cron and delta data interval timetables

Select this timetable by providing a valid cron expression as a string to the `schedule` parameter of a DAG, as described in the *Dags* documentation.

```
@dag(schedule="0 1 * * 3") # At 01:00 on Wednesday.
def example_dag():
    pass
```

EventsTimetable

Pass a list of datetimes for the DAG to run after. This can be useful for timing based on sporting events, planned communication campaigns, and other schedules that are arbitrary and irregular, but predictable.

The list of events must be finite and of reasonable size as it must be loaded every time the DAG is parsed. Optionally, use the `restrict_to_events` flag to force manual runs of the DAG that use the time of the most recent, or very first, event for the data interval. Otherwise, manual runs begin with a `data_interval_start` and `data_interval_end` equal to the time at which the manual run started. You can also name the set of events using the `description` parameter, which will be displayed in the Airflow UI.

```
from airflow.timetables.events import EventsTimetable

@dag(
    schedule=EventsTimetable(
        event_dates=[
            pendulum.datetime(2022, 4, 8, 27, tz="America/Chicago"),
            pendulum.datetime(2022, 4, 17, 8, 27, tz="America/Chicago"),
            pendulum.datetime(2022, 4, 22, 20, 50, tz="America/Chicago"),
        ],
        description="My Team's Baseball Games",
        restrict_to_events=False,
    ),
    ...
)
def example_dag():
    pass
```

Asset event based scheduling with time based scheduling

Combining conditional asset expressions with time-based schedules enhances scheduling flexibility.

The `AssetOrTimeSchedule` is a specialized timetable that allows for the scheduling of dags based on both time-based schedules and asset events. It also facilitates the creation of both scheduled runs, as per traditional timetables, and asset-triggered runs, which operate independently.

This feature is particularly useful in scenarios where a DAG needs to run on asset updates and also at periodic intervals. It ensures that the workflow remains responsive to data changes and consistently runs regular checks or updates.

Here's an example of a DAG using `AssetOrTimeSchedule`:

```
from airflow.timetables.assets import AssetOrTimeSchedule
from airflow.timetables.trigger import CronTriggerTimetable

@dag(
    schedule=AssetOrTimeSchedule(
        timetable=CronTriggerTimetable("0 1 * * 3", timezone="UTC"), assets=(dag1_asset &
    ↪ dag2_asset)
    )
    # Additional arguments here, replace this comment with actual arguments
)
def example_dag():
    # DAG tasks go here
    pass
```

Timetables comparisons

Differences between “trigger” and “data interval” timetables

Airflow has two sets of timetables for cron and delta schedules:

- `CronTriggerTimetable` and `CronDataIntervalTimetable` both accept a cron expression.
- `DeltaTriggerTimetable` and `DeltaDataIntervalTimetable` both accept a `timedelta` or `relativedelta`.
- A trigger timetable (`CronTriggerTimetable` or `DeltaTriggerTimetable`) does not address the concept of *data interval*, while a “data interval” one (`CronDataIntervalTimetable` or `DeltaDataIntervalTimetable`) does.
- The timestamp in the `run_id`, the `logical_date` of the two timetable kinds are defined differently based on how they handle the data interval, as described in *The time when a DAG run is triggered*.

Whether taking care of Data Interval

A trigger timetable *does not* include *data interval*. This means that the value of `data_interval_start` and `data_interval_end` (and the legacy `execution_date`) are the same; the time when a DAG run is triggered.

For a data interval timetable, the value of `data_interval_start` and `data_interval_end` (and legacy `execution_date`) are different. `data_interval_start` is the time when a DAG run is triggered and `data_interval_end` is the end of the interval.

Catchup behavior

By default, `catchup` is set to `False`. This prevents running unnecessary dags in the following scenarios: - If you create a new DAG with a start date in the past, and don't want to run dags for the past. If `catchup` is `True`, Airflow runs

all dags that would have run in that time interval. - If you pause an existing DAG, and then restart it at a later date, `catchup` being `False` means that Airflow does not run the dags that would have run during the paused period.

In these scenarios, the `logical_date` in the `run_id` are based on how the timetable handles the data interval.

You can change the default `catchup` behavior using the Airflow config `[scheduler] catchup_by_default`.

See [Catchup](#) for more information about how DAG runs are triggered when using `catchup`.

The time when a DAG run is triggered

Both trigger and data interval timetables trigger DAG runs at the same time. However, the timestamp for the `run_id` is different for each. This is because `run_id` is based on `logical_date`.

For example, suppose there is a cron expression `@daily` or `0 0 * * *`, which is scheduled to run at 12AM every day. If you enable dags using the two timetables at 3PM on January 31st, - `CronTriggerTimetable` creates a new DAG run at 12AM on February 1st. The `run_id` timestamp is midnight, on February 1st. - `CronDataIntervalTimetable` immediately creates a new DAG run, because a DAG run for the daily time interval beginning at 12AM on January 31st did not occur yet. The `run_id` timestamp is midnight, on January 31st, since that is the beginning of the data interval.

The following is another example showing the difference in the case of skipping DAG runs:

Suppose there are two running dags with a cron expression `@daily` or `0 0 * * *` that use the two different timetables. If you pause the dags at 3PM on January 31st and re-enable them at 3PM on February 2nd, - `CronTriggerTimetable` skips the DAG runs that were supposed to trigger on February 1st and 2nd. The next DAG run will be triggered at 12AM on February 3rd. - `CronDataIntervalTimetable` skips the DAG runs that were supposed to trigger on February 1st only. A DAG run for February 2nd is immediately triggered after you re-enable the DAG.

In these examples, you see how a trigger timetable creates DAG runs more intuitively and similar to what people expect a workflow to behave, while a data interval timetable is designed heavily around the data interval it processes, and does not reflect a workflow's own properties.

Differences between the cron and delta data interval timetables

Choosing between `DeltaDataIntervalTimetable` and `CronDataIntervalTimetable` depends on your use case. If you enable a DAG at 01:05 on February 1st, the following table summarizes the DAG runs created and the data interval that they cover, depending on 3 arguments: `schedule`, `start_date` and `catchup`.

| schedule | start_date | catchup | Intervals covered | Remarks |
|---|-----------------------------------|--------------------|--|--|
| <code>*/30 * * * *</code> | <code>year-02-01</code> | <code>True</code> | <ul style="list-style-type: none"> • 00:00 - 00:30 • 00:30 - 01:00 | Same behavior than using the timedelta object. |
| <code>*/30 * * * *</code> | <code>year-02-01</code> | <code>False</code> | <ul style="list-style-type: none"> • 00:30 - 01:00 | |
| <code>*/30 * * * *</code> | <code>year-02-01 00:10</code> | <code>True</code> | <ul style="list-style-type: none"> • 00:30 - 01:00 | Interval 00:00 - 00:30 is not after the start date, and so is skipped. |
| <code>*/30 * * * *</code> | <code>year-02-01 00:10</code> | <code>False</code> | <ul style="list-style-type: none"> • 00:30 - 01:00 | Whatever the start date, the data intervals are aligned with hour/day/etc. boundaries. |
| <code>datetime.timedelta(minutes</code> | <code>year-02-01</code> | <code>True</code> | <ul style="list-style-type: none"> • 00:00 - 00:30 • 00:30 - 01:00 | Same behavior than using the cron expression. |
| <code>datetime.timedelta(minutes</code> | <code>year-02-01</code> | <code>False</code> | <ul style="list-style-type: none"> • 00:35 - 01:05 | Interval is not aligned with start date but with the current time. |
| <code>datetime.timedelta(minutes 00:10</code> | <code>year-02-01</code> | <code>True</code> | <ul style="list-style-type: none"> • 00:10 - 00:40 | Interval is aligned with start date. Next one will be triggered in 5 minutes covering 00:40 - 01:10. |
| <code>datetime.timedelta(minutes 00:10</code> | <code>year-02-01</code> | <code>False</code> | <ul style="list-style-type: none"> • 00:35 - 01:05 | Interval is aligned with current time. Next run will be triggered in 30 minutes. |

3.8.10 Event-driven scheduling

Added in version 3.0.

Apache Airflow allows for event-driven scheduling, enabling dags to be triggered based on external events rather than predefined time-based schedules. This is particularly useful in modern data architectures where workflows need to react to real-time data changes, messages, or system signals.

By using assets, as described in *Asset-Aware Scheduling*, you can configure dags to start execution when specific external events occur. Assets provide a mechanism to establish dependencies between external events and DAG execution, ensuring that workflows react dynamically to changes in the external environment.

The `AssetWatcher` class plays a crucial role in this mechanism. It monitors an external event source, such as a message queue, and triggers an asset update when a relevant event occurs. The `watchers` parameter in the `Asset` definition allows you to associate multiple `AssetWatcher` instances with an asset, enabling it to respond to various event sources.

See the `common.messaging` provider docs for more information and examples.

Supported triggers for event-driven scheduling

Not all *triggers* in Airflow can be used for event-driven scheduling. As opposed to all triggers that inherit from `BaseTrigger`, only a subset that inherit from `BaseEventTrigger` are compatible. The reason for this restriction is that some triggers are not designed for event-driven scheduling, and using them to schedule dags could lead to unintended results.

`BaseEventTrigger` ensures that triggers used for scheduling adhere to an event-driven paradigm, reacting appropriately to external event changes without causing unexpected DAG behavior.

Writing event-driven compatible triggers

To make a trigger compatible with event-driven scheduling, it must inherit from `BaseEventTrigger`. There are three main scenarios for working with triggers in this context:

1. **Creating a new event-driven trigger:** If you need a new trigger for an unsupported event source, you should create a new class inheriting from `BaseEventTrigger` and implement its logic.
2. **Adapting an existing compatible trigger:** If an existing trigger (inheriting from `BaseEvent`) is proven to be already compatible with event-driven scheduling, then you just need to change the base class from `BaseTrigger` to `BaseEventTrigger`.
3. **Adapting an existing incompatible trigger:** If an existing trigger does not appear to be compatible with event-driven scheduling, then a new trigger must be created. This new trigger must inherit `BaseEventTrigger` and ensure it properly works with event-driven scheduling. It might inherit from the existing trigger as well if both triggers share some common code.

Avoid infinite scheduling

The reason why some triggers are not compatible with event-driven scheduling is that they are waiting for an external resource to reach a given state. Examples:

- Wait for a file to exist in a storage service
- Wait for a job to be in a success state
- Wait for a row to be present in a database

Scheduling under such conditions can lead to infinite rescheduling. This is because once the condition becomes true, it is likely to remain true for an extended period.

For example, consider a DAG scheduled to run when a specific job reaches a “success” state. Once the job succeeds, it will typically remain in that state. As a result, the DAG will be triggered repeatedly every time the triggerer checks the condition.

Another example is the `S3KeyTrigger`, which checks for the presence of a specific file in an S3 bucket. Once the file is created, the trigger will continue to succeed on every check, since the condition “is file X present in bucket Y” remains true. This leads to the DAG being triggered indefinitely every time the trigger mechanism runs.

When creating custom triggers, be cautious about using conditions that remain permanently true once met. This can unintentionally result in infinite DAG executions and overwhelm your system.

Use cases for event-driven dags

- **Data ingestion pipelines:** Trigger ETL workflows when new data arrives in a storage system.
- **Machine learning workflows:** Start training models when new datasets become available.
- **IoT and real-time analytics:** React to sensor data, logs, or application events in real-time.
- **Microservices and event-driven architectures:** Orchestrate workflows based on service-to-service messages.

3.9 Administration and Deployment

This section contains information about deploying dags into production and the administration of Airflow deployments.

3.9.1 Production Deployment

It is time to deploy your DAG in production. To do this, first, you need to make sure that the Airflow is itself production-ready. Let's see what precautions you need to take.

Database backend

Airflow comes with an `SQLite` backend by default. This allows the user to run Airflow without any external database. However, such a setup is meant to be used for testing purposes only; running the default setup in production can lead to data loss in multiple scenarios. If you want to run production-grade Airflow, make sure you *configure the backend* to be an external database such as PostgreSQL or MySQL.

You can change the backend using the following config

```
[database]
sql_alchemy_conn = my_conn_string
```

Once you have changed the backend, airflow needs to create all the tables required for operation. Create an empty DB and give Airflow's user permission to `CREATE/ALTER` it. Once that is done, you can run -

```
airflow db migrate
```

`migrate` keeps track of migrations already applied, so it's safe to run as often as you need.

⚠ Warning

Prior to Airflow version 2.7.0, `airflow db upgrade` was used to apply migrations, however, it has been deprecated in favor of `airflow db migrate`.

Multi-Node Cluster

Airflow uses `LocalExecutor` by default. For a multi-node setup, you should use the `Kubernetes` executor or the `Celery` executor.

Once you have configured the executor, it is necessary to make sure that every node in the cluster contains the same configuration and dags. Airflow sends simple instructions such as “execute task X of DAG Y”, but does not send any DAG files or configuration. You can use a simple cronjob or any other mechanism to sync dags and configs across your nodes, e.g., checkout dags from git repo every 5 minutes on all nodes.

Logging

If you are using disposable nodes in your cluster, configure the log storage to be a distributed file system (DFS) such as S3 and GCS, or external services such as Stackdriver Logging, Elasticsearch or Amazon CloudWatch. This way, the logs are available even after the node goes down or gets replaced. See *Logging for Tasks* for configurations.

Note

The logs only appear in your DFS after the task has finished. You can view the logs while the task is running in UI itself.

Configuration

Airflow comes bundled with a default `airflow.cfg` configuration file. You should use environment variables for configurations that change across deployments e.g. metadata DB, password, etc. You can accomplish this using the format `AIRFLOW_{SECTION}_{KEY}`

```
AIRFLOW__DATABASE__SQLALCHEMY_CONN=my_conn_id
AIRFLOW__WEBSERVER__BASE_URL=http://host:port
```

Some configurations such as the Airflow Backend connection URI can be derived from bash commands as well:

```
sql_alchemy_conn_cmd = bash_command_to_run
```

Scheduler Uptime

Airflow users occasionally report instances of the scheduler hanging without a trace, for example in these issues:

- Scheduler gets stuck without a trace
- Scheduler stopping frequently

To mitigate these issues, make sure you have a *health check* set up that will detect when your scheduler has not heartbeat in a while.

Production Container Images

We provide a Docker Image (OCI) for Apache Airflow for use in a containerized environment. Consider using it to guarantee that software will always run the same no matter where it's deployed.

Helm Chart for Kubernetes

[Helm](#) provides a simple mechanism to deploy software to a Kubernetes cluster. We maintain an official Helm chart for Airflow that helps you define, install, and upgrade deployment. The Helm Chart uses our official Docker image and Dockerfile that is also maintained and released by the community.

Live-upgrading Airflow

Airflow is by-design a distributed system and while the *basic Airflow deployment* requires usually a complete Airflow restart to upgrade, it is possible to upgrade Airflow without any downtime when you run Airflow in a *distributed deployment*.

Such a live upgrade is possible when there are no changes in Airflow metadata database schema, so you should aim to do it when you upgrade Airflow patch-level (bugfix) versions of the same minor Airflow version or when upgrading between adjacent minor versions (feature) of Airflow after reviewing the *release notes* and *Reference for Database Migrations* and making sure there are no changes in the database schema between them.

In some cases when database migration is not significant, such live migration could also potentially be possible with upgrading Airflow database first and between MINOR versions, however, this is not recommended and you should only do it on your own risk, carefully reviewing the modifications to be applied to the database schema and assessing the risk of such upgrade - it requires deep knowledge of Airflow database *ERD Schema of the Database* and reviewing the *Reference for Database Migrations*. You should always thoroughly test such upgrade in a staging environment first.

Usually cost connected with such live upgrade preparation will be higher than the cost of a short downtime of Airflow, so we strongly discourage such live upgrades.

Make sure to test such live upgrade procedure in a staging environment before you do it in production, to avoid any surprises and side-effects.

When it comes to live-upgrading the `Webserver`, `Triggerer` components, if you run them in separate environments and have more than one instances for each of them, you can rolling-restart them one by one, without any downtime. This should usually be done as the first step in your upgrade procedure.

When you are running a deployment with separate `DAG processor`, in a *Separate DAG processing deployment* the `DAG processor` is not horizontally scaled - even if you have more of them there is usually one `DAG processor` running at a time per specific folder, so you can just stop it and start the new one - but since the `DAG processor` is not a critical component, it's ok for it to experience a short downtime.

When it comes to upgrading the schedulers and workers, you can use the live upgrade capabilities of the executor you use:

- For the `Local executor` your tasks are running as subprocesses of scheduler and you cannot upgrade the Scheduler without killing the tasks run by it. You can either pause all your dags and wait for the running tasks to complete or just stop the scheduler and kill all the tasks it runs - then you will need to clear and restart those tasks manually after the upgrade is completed (or rely on `retry` being set for stopped tasks).
- For the Celery executor, you have to first put your workers in offline mode (usually by setting a single `TERM` signal to the workers), wait until the workers finish all the running tasks, and then upgrade the code (for example by replacing the image the workers run in and restart the workers). You can monitor your workers via `flower` monitoring tool and see the number of running tasks going down to zero. Once the workers are upgraded, they will be automatically put in online mode and start picking up new tasks. You can then upgrade the Scheduler in a rolling restart mode.
- For the Kubernetes executor, you can upgrade the scheduler triggerer, webserver in a rolling restart mode, and generally you should not worry about the workers, as they are managed by the Kubernetes cluster and will be automatically adopted by Schedulers when they are upgraded and restarted.
- For the CeleryKubernetesExecutor, you follow the same procedure as for the `CeleryExecutor` - you put the workers in offline mode, wait for the running tasks to complete, upgrade the workers, and then upgrade the scheduler, triggerer and webserver in a rolling restart mode - which should also adopt tasks run via the `KubernetesExecutor` part of the executor.

Most of the rolling-restart upgrade scenarios are implemented in the `helm-chart:index`, so you can use it to upgrade your Airflow deployment without any downtime - especially in case you do patch-level upgrades of Airflow.

Kerberos-authenticated workers

Apache Airflow has a built-in mechanism for authenticating the operation with a KDC (Key Distribution Center). Airflow has a separate command `airflow kerberos` that acts as token refresher. It uses the pre-configured Kerberos Keytab to authenticate in the KDC to obtain a valid token, and then refreshing valid token at regular intervals within the current token expiry window.

Each request for refresh uses a configured principal, and only keytab valid for the principal specified is capable of retrieving the authentication token.

The best practice to implement proper security mechanism in this case is to make sure that worker workloads have no access to the Keytab but only have access to the periodically refreshed, temporary authentication tokens. This can be achieved in Docker environment by running the `airflow kerberos` command and the worker command in separate containers - where only the `airflow kerberos` token has access to the Keytab file (preferably configured as secret resource). Those two containers should share a volume where the temporary token should be written by the `airflow kerberos` and read by the workers.

In the Kubernetes environment, this can be realized by the concept of sidecar, where both Kerberos token refresher and worker are part of the same Pod. Only the Kerberos sidecar has access to Keytab secret and both containers in the same Pod share the volume, where temporary token is written by the sidecar container and read by the worker container.

This concept is implemented in the Helm Chart for Apache Airflow.

Secured Server and Service Access on Google Cloud

This section describes techniques and solutions for securely accessing servers and services when your Airflow environment is deployed on Google Cloud, or you connect to Google services, or you are connecting to the Google API.

IAM and Service Accounts

You should not rely on internal network segmentation or firewalling as our primary security mechanisms. To protect your organization's data, every request you make should contain sender identity. In the case of Google Cloud, the identity is provided by [the IAM and Service account](#). Each Compute Engine instance has an associated service account identity. It provides cryptographic credentials that your workload can use to prove its identity when making calls to Google APIs or third-party services. Each instance has access only to short-lived credentials. If you use Google-managed service account keys, then the private key is always held in escrow and is never directly accessible.

If you are using Kubernetes Engine, you can use [Workload Identity](#) to assign an identity to individual pods.

For more information about service accounts in the Airflow, see [howto/connection:gcp](#)

Impersonate Service Accounts

If you need access to other service accounts, you can impersonate other service accounts to exchange the token with the default identity to another service account. Thus, the account keys are still managed by Google and cannot be read by your workload.

It is not recommended to generate service account keys and store them in the metadata database or the secrets backend. Even with the use of the backend secret, the service account key is available for your workload.

Access to Compute Engine Instance

If you want to establish an SSH connection to the Compute Engine instance, you must have the network address of this instance and credentials to access it. To simplify this task, you can use [ComputeEngineHook](#) instead of [SSHHook](#)

The [ComputeEngineHook](#) support authorization with Google OS Login service. It is an extremely robust way to manage Linux access properly as it stores short-lived ssh keys in the metadata service, offers PAM modules for access and sudo privilege checking and offers the [nsswitch](#) user lookup into the metadata service as well.

It also solves the discovery problem that arises as your infrastructure grows. You can use the instance name instead of the network address.

Access to Amazon Web Service

Thanks to the [Web Identity Federation](#), you can exchange the Google Cloud Platform identity to the Amazon Web Service identity, which effectively means access to Amazon Web Service platform. For more information, see: [howto/connection:aws:gcp-federation](#)

3.9.2 Logging & Monitoring

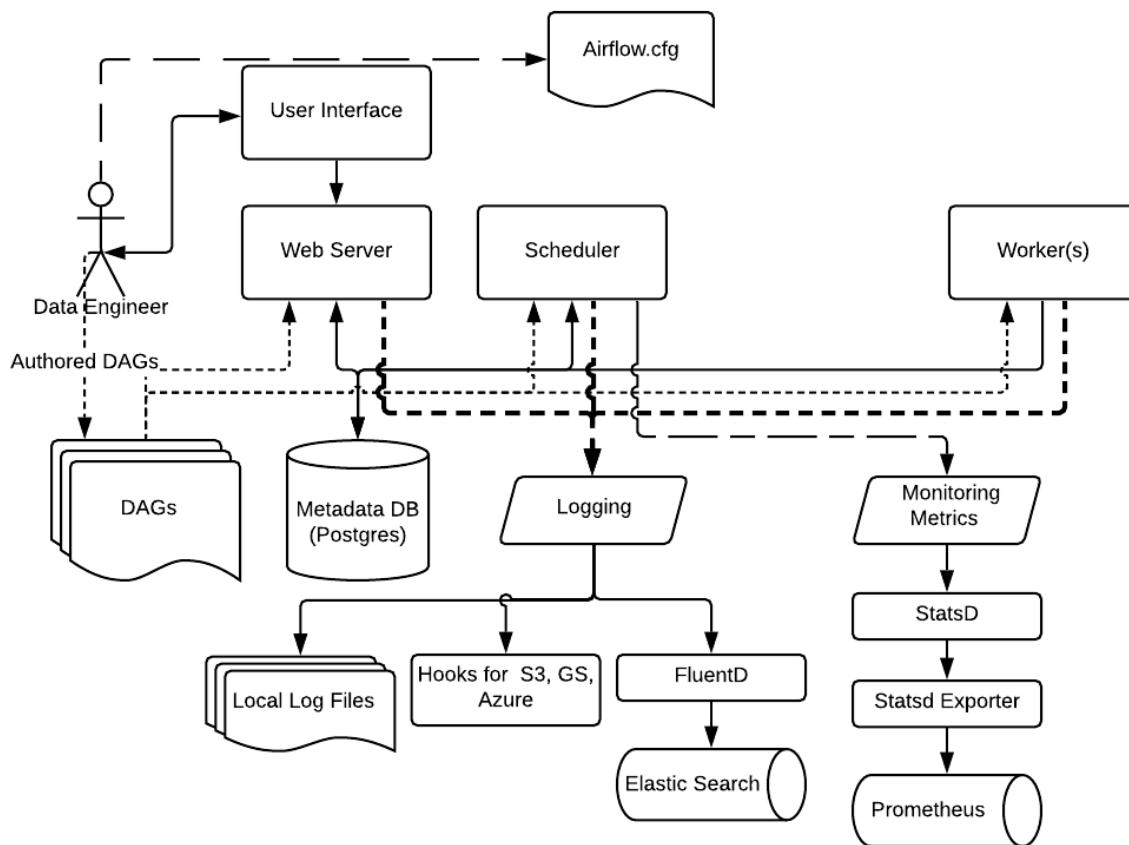
Since data pipelines are generally run without any manual supervision, observability is critical.

Airflow has support for multiple logging mechanisms, as well as a built-in mechanism to emit metrics for gathering, processing, and visualization in other downstream systems. The logging capabilities are critical for diagnosis of problems which may occur in the process of running data pipelines.

In addition to the standard logging and metrics capabilities, Airflow supports the ability to detect errors in the operation of Airflow itself, using an Airflow health check. Since Airflow is generally used for running data pipelines in production, it also supports real-time error notification via integration with Sentry.

Logging and Monitoring architecture

Airflow supports a variety of logging and monitoring mechanisms as shown below.



By default, Airflow supports logging into the local file system. These include logs from the Web server, the Scheduler, and the Workers running tasks. This is suitable for development environments and for quick debugging.

For cloud deployments, Airflow also has task handlers contributed by the Community for logging to cloud storage such as AWS, Google Cloud, and Azure.

The logging settings and options can be specified in the Airflow Configuration file, which as usual needs to be available to all the Airflow process: Web server, Scheduler, and Workers.

You can customize the logging settings for each of the Airflow components by specifying the logging settings in the Airflow Configuration file, or for advanced configuration by using *advanced features*.

For production deployments, we recommend using FluentD to capture logs and send it to destinations such as ElasticSearch or Splunk.

Note

For more information on configuring logging, see *Logging for Tasks*

Similarly, we recommend using StatsD for gathering metrics from Airflow and send them to destinations such as Prometheus.

Note

For more information on configuring metrics, see *Metrics Configuration*

Logging for Tasks

Airflow writes logs for tasks in a way that allows you to see the logs for each task separately in the Airflow UI. Core Airflow provides an interface FileTaskHandler, which writes task logs to file, and includes a mechanism to serve them from workers while tasks are running. The Apache Airflow Community also releases providers for many services (apache-airflow-providers:index) and some of them provide handlers that extend the logging capability of Apache Airflow. You can see all of these providers in apache-airflow-providers:core-extensions/logging.

When using S3, GCS, WASB, HDFS or OSS remote logging service, you can delete the local log files after they are uploaded to the remote location, by setting the config:

```
[logging]
remote_logging = True
remote_base_log_folder = schema://path/to/remote/log
delete_local_logs = True
```

Configuring logging

For the default handler, FileTaskHandler, you can specify the directory to place log files in `airflow.cfg` using `base_log_folder`. By default, logs are placed in the `AIRFLOW_HOME` directory.

Note

For more information on setting the configuration, see *Setting Configuration Options*

The default pattern is followed while naming log files for tasks:

- For normal tasks: `dag_id={dag_id}/run_id={run_id}/task_id={task_id}/attempt={try_number}.log`.
- For dynamically mapped tasks: `dag_id={dag_id}/run_id={run_id}/task_id={task_id}/map_index={map_index}/attempt={try_number}.log`.

These patterns can be adjusted by `log_filename_template`.

In addition, you can supply a remote location to store current logs and backups.

Writing to task logs from your code

Airflow uses standard the Python [logging](#) framework to write logs, and for the duration of a task, the root logger is configured to write to the task's log.

Most operators will write logs to the task log automatically. This is because they have a `log` logger that you can use to write to the task log. This logger is created and configured by `LoggingMixin` that all operators derive from. But also due to the root logger handling, any standard logger (using default settings) that propagates logging to the root will also write to the task log.

So if you want to log to the task log from custom code of yours you can do any of the following:

- Log with the `self.log` logger from `BaseOperator`
- Use standard `print` statements to print to `stdout` (not recommended, but in some cases it can be useful)
- Use the standard logger approach of creating a logger using the Python module name and using it to write to the task log

This is the usual way loggers are used directly in Python code:

```
import logging

logger = logging.getLogger(__name__)
logger.info("This is a log message")
```

Grouping of log lines

Added in version 2.9.0.

Like CI pipelines also Airflow logs can be quite large and become hard to read. Sometimes therefore it is useful to group sections of log areas and provide folding of text areas to hide non relevant content. Airflow therefore implements a compatible log message grouping like [Github](#) and [Azure DevOps](#) such that areas of text can be folded. The implemented scheme is compatible such that tools making output in CI can leverage the same experience in Airflow directly.

By adding log markers with the starting and ending positions like for example below log messages can be grouped:

```
print("Here is some standard text.")
print(":::group:::Non important details")
print("bla")
print("debug messages...")
print(":::endgroup:::")
print("Here is again some standard text.")
```

When displaying the logs in web UI, the display of logs will be condensed:

```
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - Here is some standard text.
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} Non important details
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - Here is again some standard_
 ↪text.
```

If you click on the log text label, the detailed log lies will be displayed.

```
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - Here is some standard text.
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} Non important details
```

(continues on next page)

(continued from previous page)

```
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - bla
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - debug messages...
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} Log group end
[2024-03-08, 23:30:18 CET] {logging_mixin.py:188} INFO - Here is again some standard
˓→text.
```

Interleaving of logs

Airflow's remote task logging handlers can broadly be separated into two categories: streaming handlers (such as ElasticSearch, AWS Cloudwatch, and GCP operations logging, formerly stackdriver) and blob storage handlers (e.g. S3, GCS, WASB).

For blob storage handlers, depending on the state of the task, logs could be in a lot of different places and in multiple different files. For this reason, we need to check all locations and interleave what we find. To do this we need to be able to parse the timestamp for each line. If you are using a custom formatter you may need to override the default parser by providing a callable name at Airflow setting `[logging] interleave_timestamp_parser`.

For streaming handlers, no matter the task phase or location of execution, all log messages can be sent to the logging service with the same identifier so generally speaking there isn't a need to check multiple sources and interleave.

Troubleshooting

If you want to check which task handler is currently set, you can use the `airflow info` command as in the example below.

```
$ airflow info

Apache Airflow
version           | 2.9.0.dev0
executor          | LocalExecutor
task_logging_handler | airflow.utils.log.file_task_handler.FileTaskHandler
sql_alchemy_conn   | postgresql+psycopg2://postgres:airflow@postgres/airflow
dags_folder        | /files/dags
plugins_folder     | /root/airflow/plugins
base_log_folder    | /root/airflow/logs
remote_base_log_folder |

[skipping the remaining outputs for brevity]
```

The output of `airflow info` above is truncated to only display the section that pertains to the logging configuration. You can also run `airflow config list` to check that the logging configuration options have valid values.

Advanced configuration

You can configure *advanced features* - including adding your own custom task log handlers (but also log handlers for all airflow components), and creating custom log handlers per operators, hooks and tasks.

Serving logs from workers and triggerer

Most task handlers send logs upon completion of a task. In order to view logs in real time, Airflow starts an HTTP server to serve the logs in the following cases:

- If `LocalExecutor` is used, then when `airflow scheduler` is running.
- If `CeleryExecutor` is used, then when `airflow worker` is running.

In triggerer, logs are served unless the service is started with option `--skip-serve-logs`.

The server is running on the port specified by `worker_log_server_port` option in `[logging]` section, and option `triggerer_log_server_port` for triggerer. Defaults are 8793 and 8794, respectively. Communication between the webserver and the worker is signed with the key specified by `secret_key` option in `[webserver]` section. You must ensure that the key matches so that communication can take place without problems.

We are using [Gunicorn](#) as a WSGI server. Its configuration options can be overridden with the `GUNICORN_CMD_ARGS` env variable. For details, see [Gunicorn settings](#).

Implementing a custom file task handler

Note

This is an advanced topic and most users should be able to just use an existing handler from apache-airflow-providers:core-extensions/logging.

In our providers we have a healthy variety of options with all the major cloud providers. But should you need to implement logging with a different service, and should you then decide to implement a custom FileTaskHandler, there are a few settings to be aware of, particularly in the context of trigger logging.

Triggers require a shift in the way that logging is set up. In contrast with tasks, many triggers run in the same process, and with triggers, since they run in `asyncio`, we have to be mindful of not introducing blocking calls through the logging handler. And because of the variation in handler behavior (some write to file, some upload to blob storage, some send messages over network as they arrive, some do so in thread), we need to have some way to let triggerer know how to use them.

To accomplish this we have a few attributes that may be set on the handler, either the instance or the class. Inheritance is not respected for these parameters, because subclasses of FileTaskHandler may differ from it in the relevant characteristics. These params are described below:

- `trigger_should_wrap`: Controls whether this handler should be wrapped by `TriggerHandlerWrapper`. This is necessary when each instance of handler creates a file handler that it writes all messages to.
- `trigger_should_queue`: Controls whether the triggerer should put a `QueueListener` between the event loop and the handler, to ensure blocking IO in the handler does not disrupt the event loop.
- `trigger_send_end_marker`: Controls whether an END signal should be sent to the logger when trigger completes. It is used to tell the wrapper to close and remove the individual file handler specific to the trigger that just completed.
- `trigger_supported`: If `trigger_should_wrap` and `trigger_should_queue` are not True, we generally assume that the handler does not support triggers. But if in this case the handler has `trigger_supported` set to True, then we'll still move the handler to root at triggerer start so that it will process trigger messages. Essentially, this should be true for handlers that "natively" support triggers. One such example of this is the `StackdriverTaskHandler`.

External Links

When using remote logging, you can configure Airflow to show a link to an external UI within the Airflow Web UI. Clicking the link redirects you to the external UI.

Some external systems require specific configuration in Airflow for redirection to work but others do not.

Advanced logging configuration

Not all configuration options are available from the `airflow.cfg` file. The config file describes how to configure logging for tasks, because the logs generated by tasks are not only logged in separate files by default but has to be also accessible via the webserver.

By default standard Airflow component logs are written to the `$AIRFLOW_HOME/logs` directory, but you can also customize it and configure it as you want by overriding Python logger configuration that can be configured by providing custom logging configuration object. You can also create and use logging configuration for specific operators and tasks.

Some configuration options require that the logging config class be overwritten. You can do it by copying the default configuration of Airflow and modifying it to suit your needs.

The default configuration can be seen in the `airflow_local_settings.py` template and you can see the loggers and handlers used there.

See *Configuring local settings* for details on how to configure local settings.

Except the custom loggers and handlers configurable there via the `airflow.cfg`, the logging methods in Airflow follow the usual Python logging convention, that Python objects log to loggers that follow naming convention of `<package>. <module_name>`.

You can read more about standard python logging classes (Loggers, Handlers, Formatters) in the [Python logging documentation](#).

Create a custom logging class

Configuring your logging classes can be done via the `logging_config_class` option in `airflow.cfg` file. This configuration should specify the import path to a configuration compatible with `logging.config.dictConfig()`. If your file is a standard import location, then you should set a `PYTHONPATH` environment variable.

Follow the steps below to enable custom logging config class:

1. Start by setting environment variable to known directory e.g. `~/airflow/`

```
export PYTHONPATH=~/airflow/
```

2. Create a directory to store the config file e.g. `~/airflow/config`
3. Create file called `~/airflow/config/log_config.py` with following the contents:

```
from copy import deepcopy
from airflow.config_templates.airflow_local_settings import DEFAULT_LOGGING_
    CONFIG

LOGGING_CONFIG = deepcopy(DEFAULT_LOGGING_CONFIG)
```

4. At the end of the file, add code to modify the default dictionary configuration.
5. Update `$AIRFLOW_HOME/airflow.cfg` to contain:

```
[logging]
logging_config_class = log_config.LOGGING_CONFIG
```

You can also use the `logging_config_class` together with remote logging if you plan to just extend/update the configuration with remote logging enabled. Then the deep-copied dictionary will contain the remote logging configuration generated for you and your modification will apply after remote logging configuration has been added:

```
[logging]
remote_logging = True
logging_config_class = log_config.LOGGING_CONFIG
```

1. Restart the application.

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

Note

You can override the way both standard logs of the components and “task” logs are handled.

Custom logger for Operators, Hooks and Tasks

You can create custom logging handlers and apply them to specific Operators, Hooks and tasks. By default, the Operators and Hooks loggers are child of the `airflow.task` logger: They follow respectively the naming convention `airflow.task.operators.<package>. <module_name>` and `airflow.task.hooks.<package>. <module_name>`. After creating a custom logging class, you can assign specific loggers to them.

Example of custom logging for the `SQLExecuteQueryOperator` and the `HttpHook`:

```
from copy import deepcopy
from pydantic.utils import deep_update
from airflow.config_templates.airflow_local_settings import DEFAULT_LOGGING_
˓→CONFIG

LOGGING_CONFIG = deep_update(
    deepcopy(DEFAULT_LOGGING_CONFIG),
    {
        "loggers": {
            "airflow.task.operators.airflow.providers.common.sql.operators.sql.
˓→SQLExecuteQueryOperator": {
                "handlers": ["task"],
                "level": "DEBUG",
                "propagate": True,
            },
            "airflow.task.hooks.airflow.providers.http.hooks.http.HttpHook": {
                "handlers": ["task"],
                "level": "WARNING",
                "propagate": False,
            },
        },
    },
)
```

You can also set a custom name to a Dag’s task with the `logger_name` attribute. This can be useful if multiple tasks are using the same Operator, but you want to disable logging for some of them.

Example of custom logger name:

```
# In your Dag file
SQLExecuteQueryOperator(..., logger_name="sql.big_query")
```

(continues on next page)

(continued from previous page)

```
# In your custom `log_config.py`
LOGGING_CONFIG = deep_update(
    deepcopy(DEFAULT_LOGGING_CONFIG),
    {
        "loggers": {
            "airflow.task.operators.sql.big_query": {
                "handlers": ["task"],
                "level": "WARNING",
                "propagate": True,
            },
        },
    },
)
```

If you want to limit the log size of the tasks, you can add the handlers.task.max_bytes parameter.

Example of limiting the size of tasks:

```
from copy import deepcopy
from pydantic.utils import deep_update
from airflow.config_templates.airflow_local_settings import DEFAULT_LOGGING_
CONFIG

LOGGING_CONFIG = deep_update(
    deepcopy(DEFAULT_LOGGING_CONFIG),
    {
        "handlers": {
            "task": {"max_bytes": 104857600, "backup_count": 1} # 100MB and_
            ↵keep 1 history rotate log.
        }
    },
)
```

Metrics Configuration

Airflow can be set up to send metrics to StatsD or OpenTelemetry.

Setup - StatsD

To use StatsD you must first install the required packages:

```
pip install 'apache-airflow[statsd]'
```

then add the following lines to your configuration file e.g. `airflow.cfg`

```
[metrics]
statsd_on = True
statsd_host = localhost
statsd_port = 8125
statsd_prefix = airflow
```

If you want to use a custom StatsD client instead of the default one provided by Airflow, the following key must be added to the configuration file alongside the module path of your custom StatsD client. This module must be available on your PYTHONPATH.

```
[metrics]
statsd_custom_client_path = x.y.customclient
```

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

Setup - OpenTelemetry

To use OpenTelemetry you must first install the required packages:

```
pip install 'apache-airflow[otel]'
```

Add the following lines to your configuration file e.g. `airflow.cfg`

```
[metrics]
otel_on = True
otel_host = localhost
otel_port = 8889
otel_prefix = airflow
otel_interval_milliseconds = 30000 # The interval between exports, defaults to 60000
otel_ssl_active = False
```

EnableHttps

To establish an HTTPS connection to the OpenTelemetry collector You need to configure the SSL certificate and key within the OpenTelemetry collector's `config.yml` file.

```
receivers:
  otlp:
    protocols:
      http:
        endpoint: 0.0.0.0:4318
        tls:
          cert_file: "/path/to/cert/cert.crt"
          key_file: "/path/to/key/key.pem"
```

Allow/Block Lists

If you want to avoid sending all the available metrics, you can configure an allow list or block list of prefixes to send or block only the metrics that start with the elements of the list:

```
[metrics]
metrics_allow_list = scheduler,executor,dagrun,pool,triggerer,celery
```

```
[metrics]
metrics_block_list = scheduler,executor,dagrun,pool,triggerer,celery
```

Rename Metrics

If you want to redirect metrics to a different name, you can configure the `stat_name_handler` option in `[metrics]` section. It should point to a function that validates the stat name, applies changes to the stat name if necessary, and returns the transformed stat name. The function may look as follows:

```
def my_custom_stat_name_handler(stat_name: str) -> str:
    return stat_name.lower()[:32]
```

Other Configuration Options

Note

For a detailed listing of configuration options regarding metrics, see the configuration reference documentation - [\[metrics\]](#).

Metric Descriptions

Counters

| Name | Description |
|--|---|
| <job_name>_start | Number of started <job_name> job, ex. Scheduler |
| <job_name>_end | Number of ended <job_name> job, ex. Scheduler |
| <job_name>_heartbeat_failure | Number of failed Heartbeats for a <job_name> |
| local_task_job.task_exit.<job_id>.<dag_id>.<task_id>.<return_code> | Number of LocalTaskJob terminations with code |
| local_task_job.task_exit | Number of LocalTaskJob terminations with code |
| operator_failures_<operator_name> | Operator <operator_name> failures |
| operator_failures | Operator <operator_name> failures. Metric with code |
| operator_successes_<operator_name> | Operator <operator_name> successes |
| operator_successes | Operator <operator_name> successes. Metric with code |
| ti_failures | Overall task instances failures. Metric with code |
| ti_successes | Overall task instances successes. Metric with code |
| previously_succeeded | Number of previously succeeded task instances |
| task_instances_without_heartbeats_killed | Task instances without heartbeats killed. Metric with code |
| scheduler_heartbeat | Scheduler heartbeats |
| dag_processor_heartbeat | Standalone DAG processor heartbeats |
| dag_processing.processes | Relative number of currently running DAG processes |
| dag_processing.processor_timeouts | Number of file processors that have been killed |
| dag_processing.other_callback_count | Number of non-SLA callbacks received |
| dag_processing.file_path_queue_update_count | Number of times we've scanned the filesystem |
| dag_file_processor_timeouts | (DEPRECATED) same behavior as dag_processor_timeouts |
| dag_processing.manager_stalls | Number of stalled DagFileProcessorManager |
| dag_file_refresh_error | Number of failures loading any DAG files |
| scheduler.tasks.killed_externally | Number of tasks killed externally. Metric with code |
| scheduler.orphaned_tasks.cleared | Number of Orphaned tasks cleared by the Scheduler |
| scheduler.orphaned_tasks.adopted | Number of Orphaned tasks adopted by the Scheduler |
| scheduler.critical_section_busy | Count of times a scheduler process tried to go into critical section |
| ti.start.<dag_id>.<task_id> | Number of started task in a given dag. Similar to dag_processing.processor_timeouts |
| ti.start | Number of started task in a given dag. Similar to dag_processing.processor_timeouts |
| ti.finish.<dag_id>.<task_id>.<state> | Number of completed task in a given dag. Similar to dag_processing.processor_timeouts |
| ti.finish | Number of completed task in a given dag. Similar to dag_processing.processor_timeouts |

continues on next page

Table 1 – continued from previous page

| Name | Description |
|---------------------------------------|--|
| dag.callback_exceptions | Number of exceptions raised from DAG call |
| celery.task_timeout_error | Number of AirflowTaskTimeout errors ra |
| celery.execute_command.failure | Number of non-zero exit code from Celery t |
| task_removed_from_dag.<dag_id> | Number of tasks removed for a given dag (i.e. |
| task_removed_from_dag | Number of tasks removed for a given dag (i.e. |
| task_restored_to_dag.<dag_id> | Number of tasks restored for a given dag (i.e. |
| task_restored_to_dag.<dag_id> | Number of tasks restored for a given dag (i.e. |
| task_instance_created_<operator_name> | Number of tasks instances created for a give |
| task_instance_created | Number of tasks instances created for a give |
| triggerer_heartbeat | Triggerer heartbeats |
| triggers.blocked_main_thread | Number of triggers that blocked the main th |
| triggers.failed | Number of triggers that errored before they c |
| triggers.succeeded | Number of triggers that have fired at least o |
| asset.updates | Number of updated assets |
| asset.orphaned | Number of assets marked as orphans because |
| asset.triggered_dagrungs | Number of DAG runs triggered by an asset u |

Gauges

| Name | Description |
|--|---|
| dagbag_size | Number of dags found when the scheduler ran a scan based on its co |
| dag_processing.import_errors | Number of errors from trying to parse DAG files |
| dag_processing.total_parse_time | Seconds taken to scan and import dag_processing.file_path_q |
| dag_processing.file_path_queue_size | Number of DAG files to be considered for the next scan |
| dag_processing.last_run.seconds_ago.<dag_file> | Seconds since <dag_file> was last processed |
| dag_processing.last_num_of_db_queries.<dag_file> | Number of queries to Airflow database during parsing per <dag_fi |
| scheduler.tasks.starving | Number of tasks that cannot be scheduled because of no open slot in |
| scheduler.tasks.executable | Number of tasks that are ready for execution (set to queued) with res |
| executor.open_slots.<executor_class_name> | Number of open slots on a specific executor. Only emitted when mu |
| executor.open_slots | Number of open slots on executor |
| executor.queued_tasks.<executor_class_name> | Number of queued tasks on on a specific executor. Only emitted wh |
| executor.queued_tasks | Number of queued tasks on executor |
| executor.running_tasks.<executor_class_name> | Number of running tasks on on a specific executor. Only emitted wh |
| executor.running_tasks | Number of running tasks on executor |
| pool.open_slots.<pool_name> | Number of open slots in the pool |
| pool.open_slots | Number of open slots in the pool. Metric with pool_name tagging. |
| pool.queued_slots.<pool_name> | Number of queued slots in the pool |
| pool.queued_slots | Number of queued slots in the pool. Metric with pool_name tagging |
| pool.running_slots.<pool_name> | Number of running slots in the pool |
| pool.running_slots | Number of running slots in the pool. Metric with pool_name tagging |
| pool.deferred_slots.<pool_name> | Number of deferred slots in the pool |
| pool.deferred_slots | Number of deferred slots in the pool. Metric with pool_name taggin |
| pool.scheduled_slots.<pool_name> | Number of scheduled slots in the pool |
| pool.scheduled_slots | Number of scheduled slots in the pool. Metric with pool_name tagg |
| pool.starving_tasks.<pool_name> | Number of starving tasks in the pool |
| pool.starving_tasks | Number of starving tasks in the pool. Metric with pool_name taggin |
| task.cpu_usage.<dag_id>.<task_id> | Percentage of CPU used by a task |
| task.mem_usage.<dag_id>.<task_id> | Percentage of memory used by a task |

continues on next page

Table 2 – continued from previous page

| Name | Description |
|---------------------------------------|---|
| triggers.running.<hostname> | Number of triggers currently running for a triggerer (described by hostname) |
| triggers.running | Number of triggers currently running for a triggerer (described by host) |
| triggerer.capacity_left.<hostname> | Capacity left on a triggerer to run triggers (described by hostname) |
| triggerer.capacity_left | Capacity left on a triggerer to run triggers (described by hostname). |
| ti.running.<queue>.<dag_id>.<task_id> | Number of running tasks in a given dag. As ti.start and ti.finish can be None |
| ti.running | Number of running tasks in a given dag. As ti.start and ti.finish can be None |

Timers

| Name | Description |
|--|---|
| dagrun.dependency-check.<dag_id> | Milliseconds taken to check DAG dependencies |
| dagrun.dependency-check | Milliseconds taken to check DAG dependencies. Metric with dag_id tagging. |
| dag.<dag_id>.<task_id>.duration | Milliseconds taken to run a task |
| task.duration | Milliseconds taken to run a task. Metric with dag_id and task-id tagging. |
| dag.<dag_id>.<task_id>.scheduled_duration | Milliseconds a task spends in the Scheduled state, before being Queued |
| task.scheduled_duration | Milliseconds a task spends in the Scheduled state, before being Queued. Metric with dag_id and task_id tagging. |
| dag.<dag_id>.<task_id>.queued_duration | Milliseconds a task spends in the Queued state, before being Running |
| task.queued_duration | Milliseconds a task spends in the Queued state, before being Running. Metric with dag_id and task_id tagging. |
| dag_processing.last_duration.<dag_file> | Milliseconds taken to load the given DAG file |
| dag_processing.last_duration | Milliseconds taken to load the given DAG file. Metric with file_name tagging. |
| dagrun.duration.success.<dag_id> | Milliseconds taken for a DagRun to reach success state |
| dagrun.duration.success | Milliseconds taken for a DagRun to reach success state. Metric with dag_id and run_type tagging. |
| dagrun.duration.failed.<dag_id> | Milliseconds taken for a DagRun to reach failed state |
| dagrun.duration.failed | Milliseconds taken for a DagRun to reach failed state. Metric with dag_id and run_type tagging. |
| dagrun.schedule_delay.<dag_id> | Milliseconds of delay between the scheduled DagRun start date and the actual DagRun start date |
| dagrun.schedule_delay | Milliseconds of delay between the scheduled DagRun start date and the actual DagRun start date. Metric with dag_id tagging. |
| scheduler.critical_section_duration | Milliseconds spent in the critical section of scheduler loop – only a single scheduler can enter this loop at a time |
| scheduler.critical_section_query_duration | Milliseconds spent running the critical section task instance query |
| scheduler.scheduler_loop_duration | Milliseconds spent running one scheduler loop |
| dagrun.<dag_id>.first_task_scheduling_delay | Milliseconds elapsed between first task start_date and dagrun expected start |
| dagrun.first_task_scheduling_delay | Milliseconds elapsed between first task start_date and dagrun expected start. Metric with dag_id and run_type tagging. |
| collect_db_dags | Milliseconds taken for fetching all Serialized Dags from DB |
| kubernetes_executor.clear_not_launched_queued_tasks.duration | Milliseconds taken for clearing not launched queued tasks in Kubernetes Executor |
| kubernetes_executor.adopt_task_instances.duration | Milliseconds taken to adopt the task instances in Kubernetes Executor |

Traces Configuration

Airflow can be set up to send traces in OpenTelemetry.

Setup - OpenTelemetry

To use OpenTelemetry you must first install the required packages:

```
pip install 'apache-airflow[otel]'
```

Add the following lines to your configuration file e.g. `airflow.cfg`

```
[traces]
otel_on = True
otel_host = localhost
otel_port = 8889
otel_application = airflow
otel_ssl_active = False
otel_task_log_event = True
```

Enable Https

To establish an HTTPS connection to the OpenTelemetry collector You need to configure the SSL certificate and key within the OpenTelemetry collector's `config.yml` file.

```
receivers:
  otlp:
    protocols:
      http:
        endpoint: 0.0.0.0:4318
      tls:
        cert_file: "/path/to/cert/cert.crt"
        key_file: "/path/to/key/key.pem"
```

Callbacks

A valuable component of logging and monitoring is the use of task callbacks to act upon changes in state of a given task, or across all tasks in a given DAG. For example, you may wish to alert when certain tasks have failed, or have the last task in your DAG invoke a callback when it succeeds.

Note

Callback functions are only invoked when the task state changes due to execution by a worker. As such, task changes set by the command line interface (*CLI*) or user interface (*UI*) do not execute callback functions.

Warning

Callback functions are executed after tasks are completed. Errors in callback functions will show up in scheduler logs rather than task logs. By default, scheduler logs do not show up in the UI and instead can be found in `$AIRFLOW_HOME/logs/scheduler/latest/DAG_FILE.py.log`

Callback Types

There are five types of task events that can trigger a callback:

| Name | Description |
|----------|---|
| on_succe | Invoked when the task <i>succeeds</i> |
| on_failu | Invoked when the task <i>fails</i> |
| on_retry | Invoked when the task is <i>up for retry</i> |
| on_execu | Invoked right before the task begins executing. |
| on_skipp | Invoked when the task is <i>running</i> and AirflowSkipException raised. Explicitly it is NOT called if a task is not started to be executed because of a preceding branching decision in the DAG or a trigger rule which causes execution to skip so that the task execution is never scheduled. |

Example

In the following example, failures in any task call the `task_failure_alert` function, and success in the last task calls the `dag_success_alert` function:

```
import datetime
import pendulum

from airflow.sdk import DAG
from airflow.providers.standard.operators.empty import EmptyOperator

def task_failure_alert(context):
    print(f"Task has failed, task_instance_key_str: {context['task_instance_key_str']}")

def dag_success_alert(context):
    print(f"DAG has succeeded, run_id: {context['run_id']}")

with DAG(
    dag_id="example_callback",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    dagrun_timeout=datetime.timedelta(minutes=60),
    catchup=False,
    on_success_callback=dag_success_alert,
    on_failure_callback=None,
    tags=["example"],
):
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task3 = EmptyOperator(task_id="task3", on_failure_callback=[task_failure_alert])
    task1 >> task2 >> task3
```

Note

As of Airflow 2.6.0, callbacks now supports a list of callback functions, allowing users to specify multiple functions to be executed in the desired event. Simply pass a list of callback functions to the callback args when defining your

DAG/task callbacks: e.g `on_failure_callback=[callback_func_1, callback_func_2]`

Full list of variables available in `context` in *docs* and *code*.

Checking Airflow Health Status

Airflow has two methods to check the health of components - HTTP checks and CLI checks. All available checks are accessible through the CLI, but only some are accessible through HTTP due to the role of the component being checked and the tools being used to monitor the deployment.

For example, when running on Kubernetes, use a [Liveness probes](#) (`livenessProbe` property) with *CLI checks* on the scheduler deployment to restart it when it fails. For the webserver, you can configure the readiness probe (`readinessProbe` property) using *Webserver Health Check Endpoint*.

For an example for a Docker Compose environment, see the `docker-compose.yaml` file available in the *Running Airflow in Docker*.

Webserver Health Check Endpoint

To check the health status of your Airflow instance, you can simply access the endpoint `/health`. It will return a JSON object in which a high-level glance is provided.

```
{
  "metadatabase": {
    "status": "healthy"
  },
  "scheduler": {
    "status": "healthy",
    "latest_scheduler_heartbeat": "2018-12-26 17:15:11+00:00"
  },
  "triggerer": {
    "status": "healthy",
    "latest_triggerer_heartbeat": "2018-12-26 17:16:12+00:00"
  },
  "dag_processor": {
    "status": "healthy",
    "latest_dag_processor_heartbeat": "2018-12-26 17:16:12+00:00"
  }
}
```

- The status of each component can be either “healthy” or “unhealthy”
 - The status of `metadatabase` depends on whether a valid connection can be initiated with the database
 - The status of `scheduler` depends on when the latest scheduler heartbeat was received
 - * If the last heartbeat was received more than 30 seconds (default value) earlier than the current time, the scheduler is considered unhealthy
 - * This threshold value can be specified using the option `scheduler_health_check_threshold` within the `[scheduler]` section in `airflow.cfg`
 - * If you run more than one scheduler, only the state of one scheduler will be reported, i.e. only one working scheduler is enough for the scheduler state to be considered healthy

- The status of the `triggerer` behaves exactly like that of the `scheduler` as described above. Note that the `status` and `latest_triggerer_heartbeat` fields in the health check response will be null for deployments that do not include a `triggerer` component.
- The status of the `dag_processor` behaves exactly like that of the `scheduler` as described above. Note that the `status` and `latest_dag_processor_heartbeat` fields in the health check response will be null for deployments that do not include a `dag_processor` component.

Please keep in mind that the HTTP response code of `/health` endpoint **should not** be used to determine the health status of the application. The return code is only indicative of the state of the rest call (200 for success).

Served by the web server, this health check endpoint is independent of the newer *Scheduler Health Check Server*, which optionally runs on each scheduler.

Note

For this check to work, at least one working web server is required. Suppose you use this check for scheduler monitoring, then in case of failure of the web server, you will lose the ability to monitor scheduler, which means that it can be restarted even if it is in good condition. For greater confidence, consider using *CLI Check for Scheduler* or *Scheduler Health Check Server*.

Scheduler Health Check Server

In order to check scheduler health independent of the web server, Airflow optionally starts a small HTTP server in each scheduler to serve a scheduler `\health` endpoint. It returns status code `200` when the scheduler is healthy and status code `503` when the scheduler is unhealthy. To run this server in each scheduler, set `[scheduler]enable_health_check` to `True`. By default, it is `False`. The server is running on the port specified by the `[scheduler]scheduler_health_check_server_port` option. By default, it is `8974`. We are using `http.server.BaseHTTPRequestHandler` as a small server.

CLI Check for Scheduler

Scheduler creates an entry in the table `airflow.jobs.job`. Job with information about the host and timestamp (heartbeat) at startup, and then updates it regularly. You can use this to check if the scheduler is working correctly. To do this, you can use the `airflow jobs check` command. On failure, the command will exit with a non-zero error code.

To check if the local scheduler is still working properly, run:

```
airflow jobs check --job-type SchedulerJob --local
```

To check if any scheduler is running when you are using high availability, run:

```
airflow jobs check --job-type SchedulerJob --allow-multiple --limit 100
```

CLI Check for Database

To verify that the database is working correctly, you can use the `airflow db check` command. On failure, the command will exit with a non-zero error code.

HTTP monitoring for Celery Cluster

You can optionally use Flower to monitor the health of the Celery cluster. It also provides an HTTP API that you can use to build a health check for your environment.

For details about installation, see: `apache-airflow-providers-celery:celery_executor`. For details about usage, see: [The Flower project documentation](#).

CLI Check for Celery Workers

To verify that the Celery workers are working correctly, you can use the `celery inspect ping` command. On failure, the command will exit with a non-zero error code.

Note

For this check to work, `[celery]worker_enable_remote_control` must be `True`. If the parameter is set to `False`, the command will exit with a non-zero error code.

To check if the worker running on the local host is working correctly, run:

```
celery --app airflow.providers.celery.executors.celery_executor.app inspect ping -d
→ celery@$HOSTNAME
```

To check if all workers in the cluster running is working correctly, run:

```
celery --app airflow.providers.celery.executors.celery_executor.app inspect ping
```

For more information, see: [Management Command-line Utilities \(inspect/control\)](#) and [Workers Guide](#) in the Celery documentation.

Error Tracking

Airflow can be set up to send errors to [Sentry](#).

Setup

First you must install sentry requirement:

```
pip install 'apache-airflow[sentry]'
```

After that, you need to enable the integration by setting the `sentry_on` option in the `[sentry]` section to `True`.

Add your `SENTRY_DSN` to your configuration file e.g. `airflow.cfg` in `[sentry]` section. Its template resembles the following: `{PROTOCOL}://{PUBLIC_KEY}@{HOST}/{PROJECT_ID}`

```
[sentry]
sentry_on = True
sentry_dsn = http://foo@sentry.io/123
```

Note

If this value is not provided, the SDK will try to read it from the `SENTRY_DSN` environment variable.

The `before_send` option can be used to modify or drop events before they are sent to Sentry. To set this option, provide a dotted path to a `before_send` function that the sentry SDK should be configured to use.

```
[sentry]
before_send = path.to.my.sentry.before_send
```

The `transport` option can be used to change the transport used to send events to Sentry, and possibly other Systems. To set this option, provide a dotted path to a Transport class that the sentry SDK should be configured to use.

```
[sentry]
transport = path.to.my.sentry.Transport
```

You can supply additional configuration options based on the Python platform via [sentry] section. Unsupported options: `integrations`, `in_app_include`, `in_app_exclude`, `ignore_errors`, `before_breadcrumb`.

Tags

| Name | Description |
|----------------------------------|---|
| <code>dag_id</code> | Dag name of the dag that failed |
| <code>task_id</code> | Task name of the task that failed |
| <code>data_interval_start</code> | Start of data interval when the task failed |
| <code>data_interval_end</code> | End of data interval when the task failed |
| <code>operator</code> | Operator name of the task that failed |

For backward compatibility, an additional tag `execution_date` is also available to represent the logical date. The tag should be considered deprecated in favor of `data_interval_start`.

Breadcrumbs

When a task fails with an error `breadcrumbs` will be added for the other tasks in the current DAG run.

| Name | Description |
|--|--|
| <code>completed_tasks[task_id]</code> | Task ID of task that executed before failed task |
| <code>completed_tasks[state]</code> | Final state of task that executed before failed task (only Success and Failed states are captured) |
| <code>completed_tasks[operator]</code> | Task operator of task that executed before failed task |
| <code>completed_tasks[duration]</code> | Duration in seconds of task that executed before failed task |

Impact of Sentry on Environment variables passed to Subprocess Hook

When Sentry is enabled, by default it changes the standard library to pass all environment variables to subprocesses opened by Airflow. This changes the default behaviour of `airflow.providers.standard.hooks.subprocess.SubprocessHook` - always all environment variables are passed to the subprocess executed with specific set of environment variables. In this case not only the specified environment variables are passed but also all existing environment variables are passed with `SUBPROCESS_` prefix added. This happens also for all other subprocesses.

This behaviour can be disabled by setting `default_integrations` sentry configuration parameter to `False` which disables `StdlibIntegration`. However, this also disables other default integrations, so you need to enable them manually if you want them to remain enabled (see [Sentry Default Integrations](#)).

```
[sentry]
default_integrations = False
```

3.9.3 Kubernetes

Apache Airflow aims to be a very Kubernetes-friendly project, and many users run Airflow from within a Kubernetes cluster in order to take advantage of the increased stability and autoscaling options that Kubernetes provides.

Helm Chart for Kubernetes

We maintain an official Helm chart for Airflow that helps you define, install, and upgrade deployment. The Helm Chart uses official Docker image and Dockerfile that is also maintained and released by the community.

Kubernetes Executor

The Kubernetes Executor allows you to run all the Airflow tasks on Kubernetes as separate Pods.

KubernetesPodOperator

The KubernetesPodOperator allows you to create Pods on Kubernetes.

Pod Mutation Hook

The Airflow local settings file (`airflow_local_settings.py`) can define a `pod_mutation_hook` function that has the ability to mutate pod objects before sending them to the Kubernetes client for scheduling. It receives a single argument as a reference to pod objects, and are expected to alter its attributes.

This could be used, for instance, to add sidecar or init containers to every worker pod launched by KubernetesExecutor or KubernetesPodOperator.

See *Configuring local settings* for details on how to configure local settings.

```
from kubernetes.client.models import V1Pod

def pod_mutation_hook(pod: V1Pod):
    pod.metadata.annotations["airflow.apache.org/launched-by"] = "Tests"
```

3.9.4 Lineage

Note

Lineage support is very experimental and subject to change.

Airflow provides a powerful feature for tracking data lineage not only between tasks but also from hooks used within those tasks. This functionality helps you understand how data flows throughout your Airflow pipelines.

A global instance of `HookLineageCollector` serves as the central hub for collecting lineage information. Hooks can send details about assets they interact with to this collector. The collector then uses this data to construct AIP-60 compliant Assets, a standard format for describing assets.

```
from airflow.lineage.hook import get_hook_lineage_collector

class CustomHook(BaseHook):
```

(continues on next page)

(continued from previous page)

```
def run(self):
    # run actual code
    collector = get_hook_lineage_collector()
    collector.add_input_asset(self, asset_kwargs={"scheme": "file", "path": "/tmp/in"
→"})
    collector.add_output_asset(self, asset_kwargs={"scheme": "file", "path": "/tmp/
→out"})
```

Lineage data collected by the HookLineageCollector can be accessed using an instance of HookLineageReader, which is registered in an Airflow plugin.

```
from airflow.lineage.hook_lineage import HookLineageReader
from airflow.plugins_manager import AirflowPlugin

class CustomHookLineageReader(HookLineageReader):
    def get_inputs(self):
        return self.lineage_collector.collected_assets.inputs

class HookLineageCollectionPlugin(AirflowPlugin):
    name = "HookLineageCollectionPlugin"
    hook_lineage_readers = [CustomHookLineageReader]
```

If no HookLineageReader is registered within Airflow, a default NoOpCollector is used instead. This collector does not create AIP-60 compliant assets or collect lineage information.

3.9.5 Listeners

You can write listeners to enable Airflow to notify you when events happen. Pluggy powers these listeners.

⚠ Warning

Listeners are an advanced feature of Airflow. They are not isolated from the Airflow components they run in, and can slow down or in some cases take down your Airflow instance. As such, extra care should be taken when writing listeners.

Airflow supports notifications for the following events:

Lifecycle Events

- `on_starting`
- `before_stopping`

Lifecycle events allow you to react to start and stop events for an Airflow Job, like SchedulerJob.

DagRun State Change Events

DagRun state change events occur when a *DagRun* changes state. Beginning with Airflow 3, listeners are also notified whenever a state change is triggered through the API (for `on_dag_run_success` and `on_dag_run_failed`) e.g., when a DagRun is marked as success from the Airflow UI.

- `on_dag_run_running`

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_dag_run_running(dag_run: DagRun, msg: str):
    """
    This method is called when dag run state changes to RUNNING.
    """
    print("Dag run in running state")
    queued_at = dag_run.queued_at

    version = dag_run.version_number

    print(f"Dag information Queued at: {queued_at} version: {version}")
```

- `on_dag_run_success`

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_dag_run_success(dag_run: DagRun, msg: str):
    """
    This method is called when dag run state changes to SUCCESS.
    """
    print("Dag run in success state")
    start_date = dag_run.start_date
    end_date = dag_run.end_date

    print(f"Dag run start:{start_date} end:{end_date}")
```

- `on_dag_run_failed`

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_dag_run_failed(dag_run: DagRun, msg: str):
    """
    This method is called when dag run state changes to FAILED.
    """
    print("Dag run in failure state")
    dag_id = dag_run.dag_id
    run_id = dag_run.run_id
    run_type = dag_run.run_type

    print(f"Dag information:{dag_id} Run id: {run_id} Run type: {run_type}")
```

(continues on next page)

(continued from previous page)

```
print(f"Failed with message: {msg}")
```

TaskInstance State Change Events

TaskInstance state change events occur when a RuntimeTaskInstance changes state. You can use these events to react to LocalTaskJob state changes. Starting with Airflow 3, listeners are also notified when a state change is triggered through the API (for `on_task_instance_success` and `on_task_instance_failed`) e.g., when marking a task instance as success from the Airflow UI. In such cases, the listener will receive a `TaskInstance` instance instead of a `RuntimeTaskInstance` instance.

- `on_task_instance_running`

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_task_instance_running(previous_state: TaskInstanceState, task_instance: RuntimeTaskInstance):
    """
        Called when task state changes to RUNNING.

        previous_task_state and task_instance object can be used to retrieve more
        information about current
        task_instance that is running, its dag_run, task and dag information.
    """
    print("Task instance is in running state")
    print(" Previous state of the Task instance:", previous_state)

    name: str = task_instance.task_id

    context = task_instance.get_template_context()

    task = context["task"]

    if TYPE_CHECKING:
        assert task

    dag = task.dag
    dag_name = None
    if dag:
        dag_name = dag.dag_id
    print(f"Current task name:{name}")
    print(f"Dag name:{dag_name}")
```

- `on_task_instance_success`

`airflow/example_dags/plugins/event_listener.py`

```
@hookimpl
def on_task_instance_success(
    previous_state: TaskInstanceState, task_instance: RuntimeTaskInstance | TaskInstance
)
    (continues on next page)
```

(continued from previous page)

```

):

    """
    Called when task state changes to SUCCESS.

    previous_task_state and task_instance object can be used to retrieve more
    information about current
    task_instance that has succeeded, its dag_run, task and dag information.

    A RuntimeTaskInstance is provided in most cases, except when the task's state change
    is triggered
    through the API. In that case, the TaskInstance available on the API server will be
    provided instead.
    """

    print("Task instance in success state")
    print(" Previous state of the Task instance:", previous_state)

    if isinstance(task_instance, TaskInstance):
        print("Task instance's state was changed through the API.")

        print(f"Task operator:{task_instance.operator}")
        return

    context = task_instance.get_template_context()
    operator = context["task"]

    print(f"Task operator:{operator}")

```

- `on_task_instance_failed`

`airflow/example_dags/plugins/event_listener.py`

```

@hookimpl
def on_task_instance_failed(
    previous_state: TaskInstanceState,
    task_instance: RuntimeTaskInstance | TaskInstance,
    error: None | str | BaseException,
):
    """
    Called when task state changes to FAILED.

    previous_task_state, task_instance object and error can be used to retrieve more
    information about current
    task_instance that has failed, its dag_run, task and dag information.

    A RuntimeTaskInstance is provided in most cases, except when the task's state change
    is triggered
    through the API. In that case, the TaskInstance available on the API server will be
    provided instead.
    """

    print("Task instance in failure state")

```

(continues on next page)

(continued from previous page)

```

if isinstance(task_instance, TaskInstance):
    print("Task instance's state was changed through the API.")

    print(f"Task operator:{task_instance.operator}")
    if error:
        print(f"Failure caused by {error}")
    return

context = task_instance.get_template_context()
task = context["task"]

if TYPE_CHECKING:
    assert task

print("Task start")
print(f"Task:{task}")
if error:
    print(f"Failure caused by {error}")

```

Asset Events

- on_asset_created
- on_asset_alias_created
- on_asset_changed

Asset events occur when Asset management operations are run.

Dag Import Error Events

- on_new_dag_import_error
- on_existing_dag_import_error

Dag import error events occur when dag processor finds import error in the Dag code and update the metadata database table.

This is an *experimental feature*.

Usage

To create a listener:

- import airflow.listeners.hookimpl
- implement the hookimpls for events that you'd like to generate notifications

Airflow defines the specification as `hookspec`. Your implementation must accept the same named parameters as defined in `hookspec`. If you don't use the same parameters as `hookspec`, Pluggy throws an error when you try to use your plugin. But you don't need to implement every method. Many listeners only implement one method, or a subset of methods.

To include the listener in your Airflow installation, include it as a part of an *Airflow Plugin*.

Listener API is meant to be called across all dags and all operators. You can't listen to events generated by specific dags. For that behavior, try methods like `on_success_callback` and `pre_execute`. These provide callbacks for particular DAG authors or operator creators. The logs and `print()` calls will be handled as part of the listeners.

Compatibility note

The listeners interface might change over time. We are using pluggy specifications which means that implementation of the listeners written for older versions of the interface should be forward-compatible with future versions of Airflow.

However, the opposite is not guaranteed, so if your listener is implemented against a newer version of the interface, it might not work with older versions of Airflow. It is not a problem if you target single version of Airflow, because you can adjust your implementation to the version of Airflow you use, but it is important if you are writing plugins or extensions that could be used with different versions of Airflow.

For example if a new field is added to the interface (like the `error` field in the `on_task_instance_failed` method in 2.10.0), the listener implementation will not handle the case when the field is not present in the event object and such listeners will only work for Airflow 2.10.0 and later.

In order to implement a listener that is compatible with multiple versions of Airflow including using features and fields added in newer versions of Airflow, you should check version of Airflow used and use newer version of the interface implementation, but for older versions of Airflow you should use older version of the interface.

For example if you want to implement a listener that uses the `error` field in the `on_task_instance_failed`, you should use code like this:

```
from importlib.metadata import version
from packaging.version import Version
from airflow.listeners import hookimpl

airflow_version = Version(version("apache-airflow"))
if airflow_version >= Version("2.10.0"):

    class ClassBasedListener:
        ...

        @hookimpl
        def on_task_instance_failed(self, previous_state, task_instance, error: None | str | BaseException):
            # Handle error case here
            pass

else:

    class ClassBasedListener: # type: ignore[no-redef]
        ...

        @hookimpl
        def on_task_instance_failed(self, previous_state, task_instance):
            # Handle no error case here
            pass
```

List of changes in the listener interfaces since 2.8.0 when they were introduced:

| Airflow Version | Affected method | Change |
|-----------------|---|--|
| 2.10.0 | <code>on_task_instance_fail</code> | An error field added to the interface |
| 3.0.0 | <code>on_task_instance_runn</code> | <code>session</code> argument removed from task instance listeners, <code>task_instance</code> object is now an instance of <code>RuntimeTaskInstance</code> |
| 3.0.0 | <code>on_task_instance_fail</code> <code>on_task_instance_succ</code> | <code>session</code> argument removed from task instance listeners, <code>task_instance</code> object is now an instance of <code>RuntimeTaskInstance</code> when on worker and <code>TaskInstance</code> when on API server |

3.9.6 Dag Bundles

A dag bundle is a collection of one or more dags, files along with their associated files, such as other Python scripts, configuration files, or other resources. Dag bundles can source the dags from various locations, such as local directories, Git repositories, or other external systems. Deployment administrators can also write their own dag bundle classes to support custom sources. You can also define more than one dag bundle in an Airflow deployments, allowing for better organization of your dags. By keeping the bundle at a higher level, it allows for versioning everything the dag needs to run.

This is similar, but more powerful than the *dags folder* in Airflow 2 or earlier, where dags were required to be in one place on the local disk, and getting the dags there was solely the responsibility of the deployment manager.

Since dag bundles support versioning, they also allow Airflow to run a task using a specific version of the dag bundle, allowing for a dag run to use the same code for the whole run, even if the dag is updated mid-way through the run.

Why are dag bundles important?

- **Version Control:** By supporting versioning, dag bundles allow dag runs to use the same code for the whole run, even if the dag is updated mid way through the run.
- **Scalability:** With dag bundles, Airflow can efficiently manage large numbers of DAGs by organizing them into logical units.
- **Flexibility:** Dag bundles enable seamless integration with external systems, such as Git repositories, to source dags.

Types of dag bundles

Airflow supports multiple types of dag Bundles, each catering to specific use cases:

`airflow.dag_processing.bundles.local.LocalDagBundle`

These bundles reference a local directory containing DAG files. They are ideal for development and testing environments, but do not support versioning of the bundle, meaning tasks always run using the latest code.

`airflow.providers.git.bundles.git.GitDagBundle`

These bundles integrate with Git repositories, allowing Airflow to fetch dags directly from a repository.

Configuring dag bundles

Dag bundles are configured in `dag_bundle_config_list`. You can add one or more dag bundles here.

By default, Airflow adds a local dag bundle, which is the same as the old dags folder. This is done for backwards compatibility, and you can remove it if you do not want to use it. You can also keep it and add other dag bundles, such as a git dag bundle.

For example, adding multiple dag bundles to your `airflow.cfg` file:

```
[dag_processor]
dag_bundle_config_list = [
    {
        "name": "my_git_repo",
        "classpath": "airflow.dag_processing.bundles.git.GitDagBundle",
        "kwargs": {"tracking_ref": "main", "git_conn_id": "my_git_conn"}
    },
    {
        "name": "dags-folder",
        "classpath": "airflow.dag_processing.bundles.local.LocalDagBundle",
        "kwargs": {}
    }
]
```

Note

The whitespace, particularly on the last line, is important so a multi-line value works properly. More details can be found in the the [configparser docs](#).

You can also override the `refresh_interval` per dag bundle by passing it in kwargs. This controls how often the dag processor refreshes, or looks for new files, in the dag bundles.

Writing custom dag bundles

When implementing your own dag bundle by extending the `BaseDagBundle` class, there are several methods you must implement. Below is a guide to help you implement a custom dag bundle.

Abstract Methods

The following methods are abstract and must be implemented in your custom bundle class:

`path`

This property should return a `Path` to the directory where the dag files for this bundle are stored. Airflow uses this property to locate the DAG files for processing.

`get_current_version`

This method should return the current version of the bundle as a string. Airflow will use pass this version to `__init__` later to get this version of the bundle again when it runs tasks. If versioning is not supported, it should return `None`.

`refresh`

This method should handle refreshing the bundle's contents from its source (e.g., pulling the latest changes from a remote repository). This is used by the dag processor periodically to ensure that the bundle is up-to-date.

Optional Methods

In addition to the abstract methods, you may choose to override the following methods to customize the behavior of your bundle:

`__init__`

This method can be extended to initialize the bundle with extra parameters, such as `tracking_ref` for the

`GitDagBundle`. It should also call the parent class's `__init__` method to ensure proper initialization. Expensive operations, such as network calls, should be avoided in this method to prevent delays during the bundle's instantiation, and done in the `initialize` method instead.

initialize

This method is called before the bundle is first used in the dag processor or worker. It allows you to perform expensive operations only when the bundle's content is accessed.

view_url

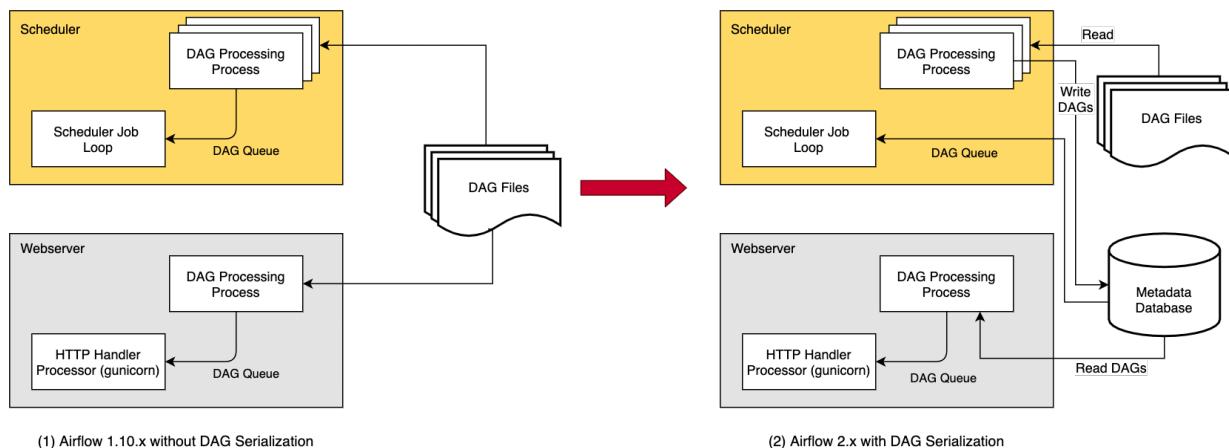
This method should return a URL as a string to view the bundle on an external system (e.g., a Git repository's web interface).

Other Considerations

- **Versioning:** If your bundle supports versioning, ensure that `initialize`, `get_current_version` and `refresh` are implemented to handle version-specific logic.
- **Concurrency:** Workers may create many bundles simultaneously, and does nothing to serialize calls to the bundle objects. Thus, the bundle class must handle locking if that is problematic for the underlying technology. For example, if you are cloning a git repo, the bundle class is responsible for locking to ensure only 1 bundle object is cloning at a time. There is a `lock` method in the base class that can be used for this purpose, if necessary.

3.9.7 DAG Serialization

In order to make Airflow Webserver stateless, Airflow $\geq 1.10.7$ supports DAG Serialization and DB Persistence. From Airflow 2.0.0, the Scheduler also uses serialized dags for consistency and makes scheduling decisions.



Without DAG Serialization & persistence in DB, the Webserver and the Scheduler both need access to the DAG files. Both the Scheduler and Webserver parse the DAG files.

With **DAG Serialization** we aim to decouple the Webserver from DAG parsing which would make the Webserver very light-weight.

As shown in the image above, when using this feature, the `DagFileProcessorProcess` in the Scheduler parses the DAG files, serializes them in JSON format and saves them in the Metadata DB as `SerializedDagModel` model.

The Webserver now instead of having to parse the DAG files again, reads the serialized dags in JSON, de-serializes them and creates the `DagBag` and uses it to show in the UI. And the Scheduler does not need the actual dags for making scheduling decisions, instead of using the dag files, we use the serialized dags that contain all the information needed to schedule the dags from Airflow 2.0.0 (this was done as part of *Scheduler HA*).

One of the key features that is implemented as a part of DAG Serialization is that instead of loading an entire DagBag when the WebServer starts we only load each DAG on demand from the Serialized Dag table. It helps to reduce the Webserver startup time and memory. This reduction is notable when you have a large number of dags.

You can enable the source code to be stored in the database to make the Webserver completely independent of the DAG files. This is not necessary if your files are embedded in the Docker image or you can otherwise provide them to the Webserver. The data is stored in the DagCode model.

The last element is rendering template fields. When serialization is enabled, templates are not rendered to requests, but a copy of the field contents is saved before the task is executed on worker. The data is stored in the RenderedTaskInstanceFields model. To limit the excessive growth of the database, only the most recent entries are kept and older entries are purged.

Note

DAG Serialization is strictly required and can not be turned off from Airflow 2.0+.

Dag Serialization Settings

Add the following settings in `airflow.cfg`:

[core]

```
# You can also update the following default configurations based on your needs
min_serialized_dag_update_interval = 30
min_serialized_dag_fetch_interval = 10
max_num_rendered_ti_fields_per_task = 30
compress_serialized_dags = False
```

- `min_serialized_dag_update_interval`: This flag sets the minimum interval (in seconds) after which the serialized dags in the DB should be updated. This helps in reducing database write rate.
- `min_serialized_dag_fetch_interval`: This option controls how often the Serialized DAG will be re-fetched from the DB when it is already loaded in the DagBag in the Webserver. Setting this higher will reduce load on the DB, but at the expense of displaying a possibly stale cached version of the DAG.
- `max_num_rendered_ti_fields_per_task`: This option controls the maximum number of Rendered Task Instance Fields (Template Fields) per task to store in the Database.
- `compress_serialized_dags`: This option controls whether to compress the Serialized DAG to the Database. It is useful when there are very large dags in your cluster. When True, this will disable the DAG dependencies view.

If you are updating Airflow from <1.10.7, please do not forget to run `airflow db migrate`.

Limitations

- When using user-defined filters and macros, the Rendered View in the Webserver might show incorrect results for TIs that have not yet executed as it might be using external modules that the Webserver won't have access to. Use `airflow tasks render` CLI command in such situation to debug or test rendering of your template_fields. Once the tasks execution starts the Rendered Template Fields will be stored in the DB in a separate table and after which the correct values would be showed in the Webserver (Rendered View tab).

Note

You need Airflow >= 1.10.10 for completely stateless Webserver. Airflow 1.10.7 to 1.10.9 needed access to DAG files in some cases. More Information: <https://airflow.apache.org/docs/1.10.9/dag-serialization.html#limitations>

Using a different JSON Library

To use a different JSON library instead of the standard `json` library like `ujson`, you need to define a `json` variable in local Airflow settings (`airflow_local_settings.py`) file as follows:

```
import ujson  
  
json = ujson
```

See *Configuring local settings* for details on how to configure local settings.

3.9.8 Modules Management

Airflow allows you to use your own Python modules in the DAG and in the Airflow configuration. The following article will describe how you can create your own module so that Airflow can load it correctly, as well as diagnose problems when modules are not loaded properly.

Often you want to use your own python code in your Airflow deployment, for example common code, libraries, you might want to generate dags using shared python code and have several DAG python files.

You can do it in one of those ways:

- add your modules to one of the folders that Airflow automatically adds to `PYTHONPATH`
- add extra folders where you keep your code to `PYTHONPATH`
- package your code into a Python package and install it together with Airflow.

The next chapter has a general description of how Python loads packages and modules, and dives deeper into the specifics of each of the three possibilities above.

How package/modules loading in Python works

The list of directories from which Python tries to load the module is given by the variable `sys.path`. Python really tries to intelligently determine the contents of this variable, depending on the operating system and how Python is installed and which Python version is used.

You can check the contents of this variable for the current Python environment by running an interactive terminal as in the example below:

```
>>> import sys  
>>> from pprint import pprint  
>>> pprint(sys.path)  
['',  
 '/home/arch/.pyenv/versions/3.9.4/lib/python37.zip',  
 '/home/arch/.pyenv/versions/3.9.4/lib/python3.9',  
 '/home/arch/.pyenv/versions/3.9.4/lib/python3.9/lib-dynload',  
 '/home/arch/venvs/airflow/lib/python3.9/site-packages']
```

`sys.path` is initialized during program startup. The first precedence is given to the current directory, i.e. `path[0]` is the directory containing the current script that was used to invoke or an empty string in case it was an interactive shell. Second precedence is given to the `PYTHONPATH` if provided, followed by installation-dependent default paths which is managed by `site` module.

`sys.path` can also be modified during a Python session by simply using `append` (for example, `sys.path.append("/path/to/custom/package")`). Python will start searching for packages in the newer paths once they're added. Airflow makes use of this feature as described in the section *Adding directories to the PYTHONPATH*.

In the variable `sys.path` there is a directory `site-packages` which contains the installed **external packages**, which means you can install packages with `pip` or `anaconda` and you can use them in Airflow. In the next section, you will learn how to create your own simple installable package and how to specify additional directories to be added to `sys.path` using the environment variable `PYTHONPATH`.

Also make sure to *Add init file to your folders*.

Typical structure of packages

This is an example structure that you might have in your `dags` folder:

```
<DIRECTORY ON PYTHONPATH>
| .airflowignore -- only needed in ``dags`` folder, see below
| -- my_company
|   | __init__.py
|   | common_package
|   |   | __init__.py
|   |   | common_module.py
|   |   | subpackage
|   |   |   | __init__.py
|   |   |   | subpackaged_util_module.py
|   |
|   | my_custom_dags
|   |   | __init__.py
|   |   | my_dag1.py
|   |   | my_dag2.py
|   |   | base_dag.py
```

In the case above, these are the ways you could import the python files:

```
from my_company.common_package.common_module import SomeClass
from my_company.common_package.subpackage.subpackaged_util_module import AnotherClass
from my_company.my_custom_dags.base_dag import BaseDag
```

You can see the `.airflowignore` file at the root of your folder. This is a file that you can put in your `dags` folder to tell Airflow which files from the folder should be ignored when the Airflow scheduler looks for dags. It should contain either regular expressions (the default) or glob expressions for the paths that should be ignored. You do not need to have that file in any other folder in `PYTHONPATH` (and also you can only keep shared code in the other folders, not the actual dags).

In the example above the dags are only in `my_custom_dags` folder, the `common_package` should not be scanned by scheduler when searching for DAGS, so we should ignore `common_package` folder. You also want to ignore the `base_dag.py` if you keep a base DAG there that `my_dag1.py` and `my_dag2.py` derives from. Your `.airflowignore` should look then like this (using the default glob syntax):

```
my_company/common_package/
my_company/my_custom_dags/base_dag.py
```

Built-in PYTHONPATH entries in Airflow

Airflow, when running dynamically adds three directories to the `sys.path`:

- The `dags` folder: It is configured with option `dags_folder` in section `[core]`.
- The `config` folder: It is configured by setting `AIRFLOW_HOME` variable (`{AIRFLOW_HOME}/config`) by default.
- The `plugins` Folder: It is configured with option `plugins_folder` in section `[core]`.

Note

The DAGS folder in Airflow 2 should not be shared with the webserver. While you can do it, unlike in Airflow 1.10, Airflow has no expectations that the DAGS folder is present in the webserver. In fact it's a bit of security risk to share the `dags` folder with the webserver, because it means that people who write DAGS can write code that the webserver will be able to execute (ideally the webserver should never run code which can be modified by users who write dags). Therefore if you need to share some code with the webserver, it is highly recommended that you share it via `config` or `plugins` folder or via installed Airflow packages (see below). Those folders are usually managed and accessible by different users (Admins/DevOps) than DAG folders (those are usually data-scientists), so they are considered as safe because they are part of configuration of the Airflow installation and controlled by the people managing the installation.

Best practices for your code naming

There are a few gotchas you should be careful about when you import your code.

Sometimes, you might see exceptions that `module 'X' has no attribute 'Y'` raised from Airflow or other library code that you use. This is usually caused by the fact that you have a module or package named 'X' in your `PYTHONPATH` at the top level and it is imported instead of the module that the original code expects.

You should always use unique names for your packages and modules and there are ways how you can make sure that uniqueness is enforced described below.

Use unique top package name

Most importantly avoid using generic names for anything that you add directly at the top level of your `PYTHONPATH`. For example if you add `airflow` folder with `__init__.py` to your `DAGS_FOLDER`, it will clash with the Airflow package and you will not be able to import anything from Airflow package. Similarly do not add `airflow.py` file directly there. Also common names used by standard library packages such as `multiprocessing` or `logging` etc. should not be used as top level - either as packages (i.e. folders with `__init__.py`) or as modules (i.e. `.py` files).

The same applies to `config` and `plugins` folders which are also at the `PYTHONPATH` and anything you add to your `PYTHONPATH` manually (see details in the following chapters).

It is recommended that you always put your `dags` / common files in a subpackage which is unique to your deployment (`my_company` in the example below). It is far too easy to use generic names for the folders that will clash with other packages already present in the system. For example if you create `airflow/operators` subfolder it will not be accessible because Airflow already has a package named `airflow.operators` and it will look there when importing from `airflow.operators`.

Don't use relative imports

Never use relative imports (starting with `.`) that were added in Python 3.

This is tempting to do something like that in `my_dag1.py`:

```
from .base_dag import BaseDag # NEVER DO THAT!!!!
```

You should import such shared DAG using full path (starting from the directory which is added to PYTHONPATH):

```
from my_company.my_custom_dags.base_dag import BaseDag # This is cool
```

The relative imports are counter-intuitive, and depending on how you start your python code, they can behave differently. In Airflow the same DAG file might be parsed in different contexts (by schedulers, by workers or during tests) and in those cases, relative imports might behave differently. Always use full python package paths when you import anything in Airflow dags, this will save you a lot of troubles. You can read more about relative import caveats in [this Stack Overflow thread](#).

Add `__init__.py` in package folders

When you create folders you should add `__init__.py` file as empty files in your folders. While in Python 3 there is a concept of implicit namespaces where you do not have to add those files to folder, Airflow expects that the files are added to all packages you added.

Inspecting your PYTHONPATH loading configuration

You can also see the exact paths using the `airflow info` command, and use them similar to directories specified with the environment variable PYTHONPATH. An example of the contents of the `sys.path` variable specified by this command may be as follows:

```
Python PATH: [/home/rootcss/venvs/airflow/bin:/usr/lib/python38.zip:/usr/lib/python3.9:/usr/lib/python3.9/lib-dynload:/home/rootcss/venvs/airflow/lib/python3.9/site-packages:/home/rootcss/airflow/dags:/home/rootcss/airflow/config:/home/rootcss/airflow/plugins]
```

Below is the sample output of the `airflow info` command:

See also

[When are plugins \(re\)loaded?](#)

Apache Airflow: 2.0.0b3

```
System info
OS           | Linux
architecture | x86_64
uname        | uname_result(system='Linux', node='85cd7ab7018e', release='4.19.76-
             linuxkit', version='#1 SMP Tue May 26 11:42:35 UTC 2020', machine='x86_64', processor='
             ')
locale       | ('en_US', 'UTF-8')
python_version | 3.9.6 (default, Nov 25 2020, 02:47:44) [GCC 8.3.0]
python_location | /usr/local/bin/python
```

Tools info

```
git          | git version 2.20.1
ssh          | OpenSSH_7.9p1 Debian-10+deb10u2, OpenSSL 1.1.1d 10 Sep 2019
kubectl     | NOT AVAILABLE
gcloud      | NOT AVAILABLE
cloud_sql_proxy | NOT AVAILABLE
mysql        | mysql Ver 8.0.22 for Linux on x86_64 (MySQL Community Server - GPL)
sqlite3     | 3.27.2 2019-02-25 16:06:06
             ↵bd49a8271d650fa89e446b42e513b595a717b9212c91dd384aab871fc1d0alt1
```

(continues on next page)

(continued from previous page)

```

psql      | psql (PostgreSQL) 11.9 (Debian 11.9-0+deb10u1)

Paths info
airflow_home   | /root/airflow
system_path    | /usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
  ↵sbin:/bin
python_path     | /usr/local/bin:/opt/airflow:/files/plugins:/usr/local/lib/python38.
  ↵zip:/usr/local/lib/python3.9:/usr/
                |   local/lib/python3.9/lib-dynload:/usr/local/lib/python3.9/site-
  ↵packages:/files/dags:/root/airflow/conf
                |   ig:/root/airflow/plugins
airflow_on_path | True

Config info
executor        | LocalExecutor
task_logging_handler | airflow.utils.log.file_task_handler.FileTaskHandler
sql_alchemy_conn | postgresql+psycopg2://postgres:airflow@postgres/airflow
dags_folder      | /files/dags
plugins_folder   | /root/airflow/plugins
base_log_folder  | /root/airflow/logs

Providers info
apache-airflow-providers-amazon          | 1.0.0b2
apache-airflow-providers-apache-cassandra | 1.0.0b2
apache-airflow-providers-apache-druid     | 1.0.0b2
apache-airflow-providers-apache-hdfs      | 1.0.0b2
apache-airflow-providers-apache-hive       | 1.0.0b2

```

Adding directories to the PYTHONPATH

You can specify additional directories to be added to `sys.path` using the environment variable `PYTHONPATH`. Start the python shell by providing the path to root of your project using the following command:

```
PYTHONPATH=/home/arch/projects/airflow_operators python
```

The `sys.path` variable will look like below:

```

>>> import sys
>>> from pprint import pprint
>>> pprint(sys.path)
[ '',
  '/home/arch/projects/airflow_operators',
  '/home/arch/.pyenv/versions/3.9.4/lib/python37.zip',
  '/home/arch/.pyenv/versions/3.9.4/lib/python3.9',
  '/home/arch/.pyenv/versions/3.9.4/lib/python3.9/lib-dynload',
  '/home/arch/venvs/airflow/lib/python3.9/site-packages']

```

As we can see that our provided directory is now added to the path, let's try to import the package now:

```

>>> import airflow_operators
Hello from airflow_operators
>>>

```

We can also use PYTHONPATH variable with the airflow commands. For example, if we run the following Airflow command:

```
PYTHONPATH=/home/arch/projects/airflow_operators airflow info
```

We'll see the Python PATH updated with our mentioned PYTHONPATH value as shown below:

```
Python PATH: [/home/arch/venv/bin:/home/arch/projects/airflow_operators:/usr/lib/
˓→python38.zip:/usr/lib/python3.9:/usr/lib/python3.9/lib-dynload:/home/arch/venv/lib/
˓→python3.9/site-packages:/home/arch/airflow/dags:/home/arch/airflow/config:/home/arch/
˓→airflow/plugins]
```

Creating a package in Python

This is most organized way of adding your custom code. Thanks to using packages, you might organize your versioning approach, control which versions of the shared code are installed and deploy the code to all your instances and containers in controlled way - all by system admins/DevOps rather than by the DAG writers. It is usually suitable when you have a separate team that manages this shared code, but if you know your python ways you can also distribute your code this way in smaller deployments. You can also install your *Plugins* and apache-airflow-providers:index as python packages, so learning how to build your package is handy.

Here is how to create your package:

1. Before starting, choose and install the build/packaging tool that you will use, ideally it should be PEP-621 compliant to be able to switch to a different tool easily. The popular choices are setuptools, poetry, hatch, flit.
2. Decide when you create your own package. create the package directory - in our case, we will call it `airflow_operators`.

```
mkdir airflow_operators
```

3. Create the file `__init__.py` inside the package and add following code:

```
print("Hello from airflow_operators")
```

When we import this package, it should print the above message.

4. Create `pyproject.toml` and fill it with build tool configuration of your choice See [The pyproject.toml specification](#)
5. Build your project using the tool of your choice. For example for hatch it can be:

```
hatch build -t wheel
```

This will create .whl file in your `dist` folder

6. Install the .whl file using pip:

```
pip install dist/airflow_operators-0.0.0-py3-none-any.whl
```

7. The package is now ready to use!

```
>>> import airflow_operators
Hello from airflow_operators
>>>
```

The package can be removed using pip command:

```
pip uninstall airflow_operators
```

For more details on how to create and publish python packages, see [Packaging Python Projects](#).

3.9.9 Scheduler

The Airflow scheduler monitors all tasks and dags, then triggers the task instances once their dependencies are complete. Behind the scenes, the scheduler spins up a subprocess, which monitors and stays in sync with all dags in the specified DAG directory. Once per minute, by default, the scheduler collects DAG parsing results and checks whether any active tasks can be triggered.

The Airflow scheduler is designed to run as a persistent service in an Airflow production environment. To kick it off, all you need to do is execute the `airflow scheduler` command. It uses the configuration specified in `airflow.cfg`.

The scheduler uses the configured *Executor* to run tasks that are ready.

To start a scheduler, simply run the command:

```
airflow scheduler
```

Your dags will start executing once the scheduler is running successfully.

Note

The first DAG Run is created based on the minimum `start_date` for the tasks in your DAG. Subsequent DAG Runs are created according to your DAG's *timetable*.

For dags with a cron or timedelta schedule, scheduler won't trigger your tasks until the period it covers has ended e.g., A job with `schedule` set as `@daily` runs after the day has ended. This technique makes sure that whatever data is required for that period is fully available before the DAG is executed. In the UI, it appears as if Airflow is running your tasks a day **late**

Note

If you run a DAG on a `schedule` of one day, the run with data interval starting on `2019-11-21` triggers after `2019-11-21T23:59`.

Let's Repeat That, the scheduler runs your job one schedule AFTER the start date, at the END of the interval.

You should refer to *DAG Runs* for details on scheduling a DAG.

Note

The scheduler is designed for high throughput. This is an informed design decision to achieve scheduling tasks as soon as possible. The scheduler checks how many free slots available in a pool and schedule at most that number of tasks instances in one iteration. This means that task priority will only come into effect when there are more scheduled tasks waiting than the queue slots. Thus there can be cases where low priority tasks will be scheduled before high priority tasks if they share the same batch. For more read about that you can reference [this GitHub discussion](#).

Running More Than One Scheduler

Airflow supports running more than one scheduler concurrently – both for performance reasons and for resiliency.

Overview

The HA (highly available) scheduler is designed to take advantage of the existing metadata database. This was primarily done for operational simplicity: every component already has to speak to this DB, and by not using direct communication or consensus algorithm between schedulers (Raft, Paxos, etc.) nor another consensus tool (Apache Zookeeper, or Consul for instance) we have kept the “operational surface area” to a minimum.

The scheduler now uses the serialized DAG representation to make its scheduling decisions and the rough outline of the scheduling loop is:

- Check for any dags needing a new DagRun, and create them
- Examine a batch of DagRuns for schedulable TaskInstances or complete DagRuns
- Select schedulable TaskInstances, and whilst respecting Pool limits and other concurrency limits, enqueue them for execution

This does, however, place some requirements on the Database.

Database Requirements

The short version is that users of PostgreSQL 12+ or MySQL 8.0+ are all ready to go – you can start running as many copies of the scheduler as you like – there is no further set up or config options needed. If you are using a different database please read on.

To maintain performance and throughput there is one part of the scheduling loop that does a number of calculations in memory (because having to round-trip to the DB for each TaskInstance would be too slow) so we need to ensure that only a single scheduler is in this critical section at once - otherwise limits would not be correctly respected. To achieve this we use database row-level locks (using `SELECT ... FOR UPDATE`).

This critical section is where TaskInstances go from scheduled state and are enqueued to the executor, whilst ensuring the various concurrency and pool limits are respected. The critical section is obtained by asking for a row-level write lock on every row of the Pool table (roughly equivalent to `SELECT * FROM slot_pool FOR UPDATE NOWAIT` but the exact query is slightly different).

The following databases are fully supported and provide an “optimal” experience:

- PostgreSQL 12+
- MySQL 8.0+

Warning

MariaDB did not implement the `SKIP LOCKED` or `NOWAIT` SQL clauses until version 10.6.0. Without these features, running multiple schedulers is not supported and deadlock errors have been reported. MariaDB 10.6.0 and following may work appropriately with multiple schedulers, but this has not been tested.

Note

Microsoft SQL Server has not been tested with HA.

Fine-tuning your Scheduler performance

What impacts scheduler's performance

The Scheduler is responsible for continuously scheduling tasks for execution. In order to fine-tune your scheduler, you need to include a number of factors:

- **The kind of deployment you have**
 - how much memory you have available
 - how much CPU you have available
 - how much networking throughput you have available
- **The logic and definition of your DAG structure:**
 - how many dags you have
 - how complex they are (i.e. how many tasks and dependencies they have)
- **The scheduler configuration**
 - How many schedulers you have
 - How many task instances scheduler processes in one loop
 - How many new DAG runs should be created/scheduled per loop
 - How often the scheduler should perform cleanup and check for orphaned tasks/adopting them

In order to perform fine-tuning, it's good to understand how Scheduler works under-the-hood. You can take a look at the Airflow Summit 2021 talk [Deep Dive into the Airflow Scheduler talk](#) to perform the fine-tuning.

How to approach Scheduler's fine-tuning

Airflow gives you a lot of “knobs” to turn to fine tune the performance but it's a separate task, depending on your particular deployment, your DAG structure, hardware availability and expectations, to decide which knobs to turn to get best effect for you. Part of the job when managing the deployment is to decide what you are going to optimize for.

Airflow gives you the flexibility to decide, but you should find out what aspect of performance is most important for you and decide which knobs you want to turn in which direction.

Generally for fine-tuning, your approach should be the same as for any performance improvement and optimizations (we will not recommend any specific tools - just use the tools that you usually use to observe and monitor your systems):

- it's extremely important to monitor your system with the right set of tools that you usually use to monitor your system. This document does not go into details of particular metrics and tools that you can use, it just describes what kind of resources you should monitor, but you should follow your best practices for monitoring to grab the right data.
- decide which aspect of performance is most important for you (what you want to improve)
- observe your system to see where your bottlenecks are: CPU, memory, I/O are the usual limiting factors
- based on your expectations and observations - decide what is your next improvement and go back to the observation of your performance, bottlenecks. Performance improvement is an iterative process.

What resources might limit Scheduler's performance

There are several areas of resource usage that you should pay attention to:

- Database connections and Database usage might become a problem as you want to increase performance and process more things in parallel. Airflow is known for being “database-connection hungry” - the more dags you have and the more you want to process in parallel, the more database connections will be opened. This is generally

not a problem for MySQL as its model of handling connections is thread-based, but this might be a problem for Postgres, where connection handling is process-based. It is a general consensus that if you have even medium size Postgres-based Airflow installation, the best solution is to use [PGBouncer](#) as a proxy to your database. The helm-chart:index supports PGBouncer out-of-the-box.

- The Airflow Scheduler scales almost linearly with several instances, so you can also add more Schedulers if your Scheduler's performance is CPU-bound.
- Make sure when you look at memory usage, pay attention to the kind of memory you are observing. Usually you should look at `working memory` (names might vary depending on your deployment) rather than `total memory used`.

What can you do, to improve Scheduler's performance

When you know what your resource usage is, the improvements that you can consider might be:

- improve utilization of your resources. This is when you have a free capacity in your system that seems under-utilized (again CPU, memory I/O, networking are the prime candidates) - you can take actions like increasing number of schedulers or decreasing intervals for more frequent actions might bring improvements in performance at the expense of higher utilization of those.
- increase hardware capacity (for example if you see that CPU is limiting you). Often the problem with scheduler performance is simply because your system is not “capable” enough and this might be the only way. For example if you see that you are using all CPU you have on machine, you might want to add another scheduler on a new machine - in most cases, when you add 2nd or 3rd scheduler, the capacity of scheduling grows linearly (unless the shared database or similar is a bottleneck).
- experiment with different values for the “scheduler tunables”. Often you might get better effects by simply exchanging one performance aspect for another. Usually performance tuning is the art of balancing different aspects.

Scheduler Configuration options

The following config settings can be used to control aspects of the Scheduler. However, you can also look at other non-performance-related scheduler configuration parameters available at [Configuration Reference](#) in the [scheduler] section.

- `max_dagruns_to_create_per_loop`

This changes the number of dags that are locked by each scheduler when creating DAG runs. One possible reason for setting this lower is if you have huge dags (in the order of 10k+ tasks per DAG) and are running multiple schedulers, you won't want one scheduler to do all the work.

- `max_dagruns_per_loop_to_schedule`

How many DagRuns should a scheduler examine (and lock) when scheduling and queuing tasks. Increasing this limit will allow more throughput for smaller dags but will likely slow down throughput for larger (>500 tasks for example) dags. Setting this too high when using multiple schedulers could also lead to one scheduler taking all the DAG runs leaving no work for the others.

- `use_row_level_locking`

Should the scheduler issue `SELECT ... FOR UPDATE` in relevant queries. If this is set to False then you should not run more than a single scheduler at once.

- `pool_metrics_interval`

How often (in seconds) should pool usage stats be sent to StatsD (if statsd_on is enabled). This is a *relatively* expensive query to compute this, so this should be set to match the same period as your StatsD roll-up period.

- *running_metrics_interval*

How often (in seconds) should running task instance stats be sent to StatsD (if `statsd_on` is enabled). This is a *relatively* expensive query to compute this, so this should be set to match the same period as your StatsD roll-up period.

- *orphaned_tasks_check_interval*

How often (in seconds) should the scheduler check for orphaned tasks or dead SchedulerJobs.

This setting controls how a dead scheduler will be noticed and the tasks it was “supervising” get picked up by another scheduler. The tasks will stay running, so there is no harm in not detecting this for a while.

When a SchedulerJob is detected as “dead” (as determined by `scheduler_health_check_threshold`) any running or queued tasks that were launched by the dead process will be “adopted” and monitored by this scheduler instead.

- *max_tis_per_query* The batch size of queries in the scheduling main loop. This should not be greater than `core.parallelism`. If this is too high then SQL query performance may be impacted by complexity of query predicate, and/or excessive locking.

Additionally, you may hit the maximum allowable query length for your db. Set this to 0 to use the value of `core.parallelism`.

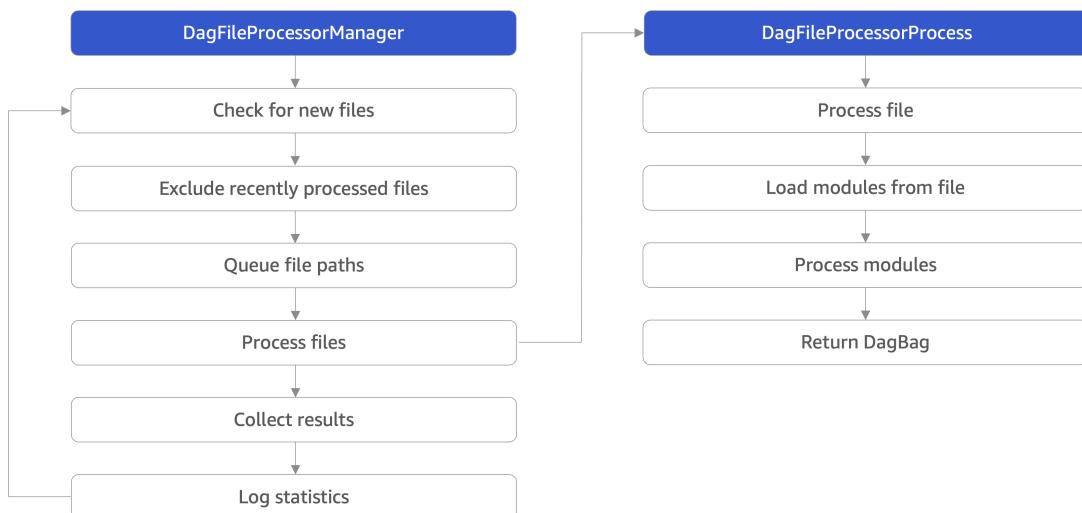
- *scheduler_idle_sleep_time* Controls how long the scheduler will sleep between loops, but if there was nothing to do in the loop. i.e. if it scheduled something then it will start the next loop iteration straight away. This parameter is badly named (historical reasons) and it will be renamed in the future with deprecation of the current name.

3.9.10 DAG File Processing

DAG File Processing refers to the process of reading the python files that define your dags and storing them such that the scheduler can schedule them.

There are two primary components involved in DAG file processing. The `DagFileProcessorManager` is a process executing an infinite loop that determines which files need to be processed, and the `DagFileProcessorProcess` is a separate process that is started to convert an individual file into one or more DAG objects.

The `DagFileProcessorManager` runs user codes. As a result, it runs as a standalone process by running the `airflow dag-processor` CLI command.



`DagFileProcessorManager` has the following steps:

1. Check for new files: If the elapsed time since the DAG was last refreshed is > `dag_dir_list_interval` (*Deprecated*) then update the file paths list
2. Exclude recently processed files: Exclude files that have been processed more recently than `min_file_process_interval` and have not been modified
3. Queue file paths: Add files discovered to the file path queue
4. Process files: Start a new `DagFileProcessorProcess` for each file, up to a maximum of `parsing_processes`
5. Collect results: Collect the result from any finished DAG processors
6. Log statistics: Print statistics and emit `dag_processing.total_parse_time`

`DagFileProcessorProcess` has the following steps:

1. Process file: The entire process must complete within `dag_file_processor_timeout`
2. The DAG files are loaded as Python module: Must complete within `dagbag_import_timeout`
3. Process modules: Find DAG objects within Python module
4. Return DagBag: Provide the `DagFileProcessorManager` a list of the discovered DAG objects

3.9.11 Fine-tuning your DAG processor performance

What impacts DAG processor's performance

The DAG processor is responsible for continuously parsing DAG files and synchronizing with the DAG in the database. In order to fine-tune your DAG processor, you need to include a number of factors:

- **The kind of deployment you have**
 - what kind of filesystem you have to share the dags (impacts performance of continuously reading dags)
 - how fast the filesystem is (in many cases of distributed cloud filesystem you can pay extra to get more throughput/faster filesystem)
 - how much memory you have for your processing
 - how much CPU you have available
 - how much networking throughput you have available
- **The logic and definition of your DAG structure:**
 - how many DAG files you have
 - how many dags you have in your files
 - how large the DAG files are (remember DAG parser needs to read and parse the file every n seconds)
 - how complex they are (i.e. how fast they can be parsed, how many tasks and dependencies they have)
 - whether parsing your DAG file involves importing a lot of libraries or heavy processing at the top level
(Hint! It should not. See *Top level Python Code*)
- **The DAG processor configuration**
 - How many DAG processors you have
 - How many parsing processes you have in your DAG processor
 - How much time DAG processor waits between re-parsing of the same DAG (it happens continuously)
 - How many callbacks you run per DAG processor loop

How to approach DAG processor's fine-tuning

Airflow gives you a lot of “knobs” to turn to fine tune the performance but it’s a separate task, depending on your particular deployment, your DAG structure, hardware availability and expectations, to decide which knobs to turn to get best effect for you. Part of the job when managing the deployment is to decide what you are going to optimize for. Some users are ok with 30 seconds delays of new DAG parsing, at the expense of lower CPU usage, whereas some other users expect the dags to be parsed almost instantly when they appear in the dags folder at the expense of higher CPU usage for example.

Airflow gives you the flexibility to decide, but you should find out what aspect of performance is most important for you and decide which knobs you want to turn in which direction.

Generally for fine-tuning, your approach should be the same as for any performance improvement and optimizations (we will not recommend any specific tools - just use the tools that you usually use to observe and monitor your systems):

- it’s extremely important to monitor your system with the right set of tools that you usually use to monitor your system. This document does not go into details of particular metrics and tools that you can use, it just describes what kind of resources you should monitor, but you should follow your best practices for monitoring to grab the right data.
- decide which aspect of performance is most important for you (what you want to improve)
- observe your system to see where your bottlenecks are: CPU, memory, I/O are the usual limiting factors
- based on your expectations and observations - decide what is your next improvement and go back to the observation of your performance, bottlenecks. Performance improvement is an iterative process.

What resources might limit DAG processors's performance

There are several areas of resource usage that you should pay attention to:

- FileSystem performance. The Airflow DAG processor relies heavily on parsing (sometimes a lot) of Python files, which are often located on a shared filesystem. The DAG processor continuously reads and re-parses those files. The same files have to be made available to workers, so often they are stored in a distributed filesystem. You can use various filesystems for that purpose (NFS, CIFS, EFS, GCS fuse, Azure File System are good examples). There are various parameters you can control for those filesystems and fine-tune their performance, but this is beyond the scope of this document. You should observe statistics and usage of your filesystem to determine if problems come from the filesystem performance. For example there are anecdotal evidences that increasing IOPS (and paying more) for the EFS performance, dramatically improves stability and speed of parsing Airflow dags when EFS is used.
- Another solution to FileSystem performance, if it becomes your bottleneck, is to turn to alternative mechanisms of distributing your dags. Embedding dags in your image and GitSync distribution have both the property that the files are available locally for the DAG processor and it does not have to use a distributed filesystem to read the files, the files are available locally for the DAG processor and it is usually as fast as it can be, especially if your machines use fast SSD disks for local storage. Those distribution mechanisms have other characteristics that might make them not the best choice for you, but if your problems with performance come from distributed filesystem performance, they might be the best approach to follow.
- Database connections and Database usage might become a problem as you want to increase performance and process more things in parallel. Airflow is known for being “database-connection hungry” - the more dags you have and the more you want to process in parallel, the more database connections will be opened. This is generally not a problem for MySQL as its model of handling connections is thread-based, but this might be a problem for Postgres, where connection handling is process-based. It is a general consensus that if you have even medium size Postgres-based Airflow installation, the best solution is to use [PGBouncer](#) as a proxy to your database. The helm-chart:index supports PGBouncer out-of-the-box.
- CPU usage is most important for FileProcessors - those are the processes that parse and execute Python DAG files. Since DAG processors typically triggers such parsing continuously, when you have a lot of dags, the processing might take a lot of CPU. You can mitigate it by increasing the `min_file_process_interval`, but this is one of the

mentioned trade-offs, result of this is that changes to such files will be picked up slower and you will see delays between submitting the files and getting them available in Airflow UI and executed by Scheduler. Optimizing the way how your dags are built, avoiding external data sources is your best approach to improve CPU usage. If you have more CPUs available, you can increase number of processing threads `parsing_processes`.

- Airflow might use quite a significant amount of memory when you try to get more performance out of it. Often more performance is achieved in Airflow by increasing the number of processes handling the load, and each process requires whole interpreter of Python loaded, a lot of classes imported, temporary in-memory storage. A lot of it is optimized by Airflow by using forking and copy-on-write memory used but in case new classes are imported after forking this can lead to extra memory pressure. You need to observe if your system is using more memory than it has - which results with using swap disk, which dramatically decreases performance. Make sure when you look at memory usage, pay attention to the kind of memory you are observing. Usually you should look at `working memory` (names might vary depending on your deployment) rather than `total memory` used.

What can you do, to improve DAG processor's performance

When you know what your resource usage is, the improvements that you can consider might be:

- improve the logic, efficiency of parsing and reduce complexity of your top-level DAG Python code. It is parsed continuously so optimizing that code might bring tremendous improvements, especially if you try to reach out to some external databases etc. while parsing dags (this should be avoided at all cost). The *Top level Python Code* explains what are the best practices for writing your top-level Python code. The *Reducing DAG complexity* document provides some areas that you might look at when you want to reduce complexity of your code.
- improve utilization of your resources. This is when you have a free capacity in your system that seems under-utilized (again CPU, memory I/O, networking are the prime candidates) - you can take actions like increasing number of parsing processes might bring improvements in performance at the expense of higher utilization of those.
- increase hardware capacity (for example if you see that CPU is limiting you or that I/O you use for DAG filesystem is at its limits). Often the problem with DAG processor performance is simply because your system is not “capable” enough and this might be the only way, unless a shared database or filesystem is a bottleneck.
- experiment with different values for the “DAG processor tunables”. Often you might get better effects by simply exchanging one performance aspect for another. For example if you want to decrease the CPU usage, you might increase file processing interval (but the result will be that new dags will appear with bigger delay). Usually performance tuning is the art of balancing different aspects.
- sometimes you change DAG processor behavior slightly (for example change parsing sort order) in order to get better fine-tuned results for your particular deployment.

DAG processor Configuration options

The following config settings can be used to control aspects of the Scheduler. However, you can also look at other non-performance-related scheduler configuration parameters available at *Configuration Reference* in the [scheduler] section.

- `file_parsing_sort_mode` The scheduler will list and sort the DAG files to decide the parsing order.
- `min_file_process_interval` Number of seconds after which a DAG file is re-parsed. The DAG file is parsed every `min_file_process_interval` number of seconds. Updates to dags are reflected after this interval. Keeping this number low will increase CPU usage.
- `parsing_processes` The scheduler can run multiple processes in parallel to parse DAG files. This defines how many processes will run.

3.9.12 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. There you can also decide whether the pool should include *deferred tasks* in its calculation of occupied slots.

Tasks can then be associated with one of the existing pools by using the `pool` parameter when creating tasks:

```
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)
```

Tasks will be scheduled as usual while the slots fill up. The number of slots occupied by a task can be configured by `pool_slots` (see section below). 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 Weights* of the task and its descendants.

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

Using multiple pool slots

Airflow tasks will each occupy a single pool slot by default, but they can be configured to occupy more with the `pool_slots` argument if required. This is particularly useful when several tasks that belong to the same pool don't carry the same "computational weight".

For instance, consider a pool with 2 slots, `Pool(pool='maintenance', slots=2)`, and the following tasks:

```
BashOperator(  
    task_id="heavy_task",  
    bash_command="bash backup_data.sh",  
    pool_slots=2,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task1",  
    bash_command="bash check_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)  
  
BashOperator(  
    task_id="light_task2",  
    bash_command="bash remove_files.sh",  
    pool_slots=1,  
    pool="maintenance",  
)
```

Since the heavy task is configured to use 2 pool slots, it depletes the pool when running. Therefore, any of the light tasks must queue and wait for the heavy task to complete before they are executed. Here, in terms of resource usage, the heavy task is equivalent to two light tasks running concurrently.

This implementation can prevent overwhelming system resources, which (in this example) could occur when a heavy and a light task are running concurrently. On the other hand, both light tasks can run concurrently since they only occupy one pool slot each, while the heavy task would have to wait for two pool slots to become available before getting executed.

3.9.13 Cluster Policies

If you want to check or mutate dags or Tasks on a cluster-wide level, then a Cluster Policy will let you do that. They have three main purposes:

- Checking that dags / tasks meet a certain standard
- Setting default arguments on dags / tasks
- Performing custom routing logic

There are three main types of cluster policy:

- `dag_policy`: Takes a `DAG` parameter called `dag`. Runs at load time of the DAG from `DagBag DagBag`.
- `task_policy`: Takes a `BaseOperator` parameter called `task`. The policy gets executed when the task is created during parsing of the task from `DagBag` at load time. This means that the whole task definition can be altered in the task policy. It does not relate to a specific task running in a `DagRun`. The `task_policy` defined is applied to all the task instances that will be executed in the future.
- `task_instance_mutation_hook`: Takes a `TaskInstance` parameter called `task_instance`. The `task_instance_mutation_hook` applies not to a task but to the instance of a task that relates to a particular `DagRun`. It is executed in a “worker”, not in the dag file processor, just before the task instance is executed. The policy is only applied to the currently executed run (i.e. instance) of that task.

The DAG and Task cluster policies can raise the `AirflowClusterPolicyViolation` exception to indicate that the dag/task they were passed is not compliant and should not be loaded.

They can also raise the `AirflowClusterPolicySkipDag` exception when skipping that DAG is needed intentionally. Unlike `AirflowClusterPolicyViolation`, this exception is not displayed on the Airflow web UI (Internally, it's not recorded on `import_error` table on meta database.)

Any extra attributes set by a cluster policy take priority over those defined in your DAG file; for example, if you set an `sla` on your Task in the DAG file, and then your cluster policy also sets an `sla`, the cluster policy's value will take precedence.

How do define a policy function

There are two ways to configure cluster policies:

1. create an `airflow_local_settings.py` file somewhere in the python search path (the `config/` folder under your `$AIRFLOW_HOME` is a good “default” location) and then add callables to the file matching one or more of the cluster policy names above (e.g. `dag_policy`).

See `Configuring local settings` for details on how to configure local settings.

2. By using a `setuptools entrypoint` in a custom module using the `Pluggy` interface.

Added in version 2.6.

This method is more advanced and for people who are already comfortable with python packaging.

First create your policy function in a module:

```
from airflow.policies import hookimpl

@hookimpl
def task_policy(task) -> None:
    # Mutate task in place
    #
    print(f"Hello from {__file__}")
```

And then add the entrypoint to your project specification. For example, using `pyproject.toml` and `setuptools`:

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "my-airflow-plugin"
version = "0.0.1"
#
dependencies = ["apache-airflow>=2.6"]
[project.entry-points.'airflow.policy']
_ = 'my_airflow_plugin.policies'
```

The entrypoint group must be `airflow.policy`, and the name is ignored. The value should be your module (or class) decorated with the `@hookimpl` marker.

Once you have done that, and you have installed your distribution into your Airflow env, the policy functions will get called by the various Airflow components. (The exact call order is undefined, so don't rely on any particular calling order if you have multiple plugins).

One important thing to note (for either means of defining policy functions) is that the argument names must exactly match as documented below.

Available Policy Functions

`airflow.policies.task_policy(task)`

Allow altering tasks after they are loaded in the DagBag.

It allows administrator to rewire some task's parameters. Alternatively you can raise `AirflowClusterPolicyViolation` exception to stop DAG from being executed.

Here are a few examples of how this can be useful:

- You could enforce a specific queue (say the spark queue) for tasks using the `SparkOperator` to make sure that these tasks get wired to the right workers
- You could enforce a task timeout policy, making sure that no tasks run for more than 48 hours

Parameters

`task (airflow.models.baseoperator.BaseOperator)` – task to be mutated

`airflow.policies.dag_policy(dag)`

Allow altering DAGs after they are loaded in the DagBag.

It allows administrator to rewire some DAG's parameters. Alternatively you can raise `AirflowClusterPolicyViolation` exception to stop DAG from being executed.

Here are a few examples of how this can be useful:

- You could enforce default user for DAGs
- Check if every DAG has configured tags

Parameters

`dag` (`airflow.models.dag.DAG`) – dag to be mutated

`airflow.policies.task_instance_mutation_hook(task_instance)`

Allow altering task instances before being queued by the Airflow scheduler.

This could be used, for instance, to modify the task instance during retries.

Parameters

`task_instance` (`airflow.models.taskinstance.TaskInstance`) – task instance to be mutated

`airflow.policies.pod_mutation_hook(pod)`

Mutate pod before scheduling.

This setting allows altering `kubernetes.client.models.V1Pod` object before they are passed to the Kubernetes client for scheduling.

This could be used, for instance, to add sidecar or init containers to every worker pod launched by KubernetesExecutor or KubernetesPodOperator.

`airflow.policies.get_airflow_context_vars(context)`

Inject airflow context vars into default airflow context vars.

This setting allows getting the airflow context vars, which are key value pairs. They are then injected to default airflow context vars, which in the end are available as environment variables when running tasks `dag_id`, `task_id`, `logical_date`, `dag_run_id`, `try_number` are reserved keys.

Parameters

`context` – The context for the task_instance of interest.

Examples

DAG policies

This policy checks if each DAG has at least one tag defined:

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

    if "only_for_beta" in dag.tags:
        raise AirflowClusterPolicySkipDag(
            f"DAG {dag.dag_id} is not loaded on the production cluster, due to `only_for_"

```

(continues on next page)

(continued from previous page)

```
→beta` tag."
    )
```

Note

To avoid import cycles, if you use DAG in type annotations in your cluster policy, be sure to import from `airflow.models` and not from `airflow`.

Note

DAG policies are applied after the DAG has been completely loaded, so overriding the `default_args` parameter has no effect. If you want to override the default operator settings, use task policies instead.

Task policies

Here's an example of enforcing a maximum timeout policy on every task:

```
class TimedOperator(BaseOperator, ABC):
    timeout: timedelta

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

You could also implement to protect against common errors, rather than as technical security controls. For example, don't run tasks without Airflow owners:

```
def task_must_have_owners(task: BaseOperator):
    if task.owner and not isinstance(task.owner, str):
        raise AirflowClusterPolicyViolation(f"""owner should be a string. Current value: {task.owner!r}""")

    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, your `airflow_local_settings.py` might follow this pattern:

```
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.fileloc}): \n"
            f"Notices:\n"
            f"{notices_list}"
        )

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

See [Configuring local settings](#) for details on how to configure local settings.

Task instance mutation

Here's an example of re-routing tasks that are on their second (or greater) retry to a different queue:

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

Note that since priority weight is determined dynamically using weight rules, you cannot alter the `priority_weight` of a task instance within the mutation hook.

3.9.14 Priority Weights

`priority_weight` defines priorities in the executor queue. The default `priority_weight` is 1, and can be bumped to any integer. Moreover, each task has a true `priority_weight` that is calculated based on its `weight_rule` which defines the weighting method used for the effective total priority weight of the task.

Below are the weighting methods. By default, Airflow's weighting method is `downstream`.

downstream The effective weight of the task is the aggregate sum of all downstream descendants. As a result, upstream tasks will have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple DAG run instances and desire to have all upstream tasks to complete for all runs before each DAG can continue processing downstream tasks.

upstream The effective weight is the aggregate sum of all upstream ancestors. This is the opposite where downstream tasks have higher weight and will be scheduled more aggressively when using positive weight values. This is useful when you have multiple DAG run instances and prefer to have each DAG complete before starting upstream tasks of other DAG runs.

absolute The effective weight is the exact `priority_weight` specified without additional weighting. You may want to do this when you know exactly what priority weight each task should have. Additionally, when set to `absolute`, there is bonus effect of significantly speeding up the task creation process as for very large dags.

The `priority_weight` parameter can be used in conjunction with *Pools*.

 **Note**

As most database engines are using 32-bit for integers, the maximum value for any calculated or defined `priority_weight` is 2,147,483,647 and the minimum value is -2,147,483,648.

Custom Weight Rule

Added in version 2.9.0.

You can implement your own custom weighting method by extending the `PriorityWeightStrategy` class and registering it in a plugin.

`airflow/example_dags/plugins/decreasing_priority_weight_strategy.py`

```
class DecreasingPriorityStrategy(PriorityWeightStrategy):
    """A priority weight strategy that decreases the priority weight with each attempt
    of the DAG task."""
    def get_weight(self, ti: TaskInstance):
        return max(3 - ti.try_number + 1, 1)

class DecreasingPriorityWeightStrategyPlugin(AirflowPlugin):
    name = "decreasing_priority_weight_strategy_plugin"
    priority_weight_strategies = [DecreasingPriorityStrategy]
```

To check if the custom priority weight strategy is already available in Airflow, you can run the bash command `airflow plugins`. Then to use it, you can create an instance of the custom class and provide it in the `weight_rule` parameter of the task or provide the path of the custom class:

`airflow/example_dags/example_custom_weight.py`

```
with DAG(
    dag_id="example_custom_weight",
    schedule="@0 * * * *",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
```

(continues on next page)

(continued from previous page)

```

dagrun_timeout=datetime.timedelta(minutes=60),
tags=["example", "example2"],
) as dag:
    start = EmptyOperator(
        task_id="start",
    )

    # provide the class instance
    task_1 = BashOperator(task_id="task_1", bash_command="echo 1", weight_
    ↵rule=DecreasingPriorityStrategy())

    # or provide the path of the class
    task_2 = BashOperator(
        task_id="task_2",
        bash_command="echo 1",
        weight_rule="airflow.example_dags.plugins.decreasing_priority_weight_strategy.
    ↵DecreasingPriorityStrategy",
    )

    task_non_custom = BashOperator(task_id="task_non_custom", bash_command="echo 1", ↵
    ↵priority_weight=2)

    start >> [task_1, task_2, task_non_custom]

```

After the DAG is running, you can check the `priority_weight` parameter on the task to verify that it is using the custom priority strategy rule.

This is an *experimental feature*.

3.9.15 Web Stack

Sometimes you want to deploy the backend and frontend behind a variable url path prefix. To do so, you can configure the url `base_url` for instance, set it to `http://localhost:28080/d12345`. All the APIs routes will now be available through that additional d12345 prefix. Without rebuilding the frontend, XHR requests and static file queries should be directed to the prefixed url and served successfully.

You will also need to update the execution API server url `execution_api_server_url` for tasks to be able to reach the API with the new prefix.

3.9.16 Plugins

Warning

As part of Airflow 3.0.0 development, support for user-defined macros is pending. This will be added soon in an upcoming release. See [#48476](#) for further support updates on this feature.

Airflow has a simple plugin manager built-in that can integrate external features to its core by simply dropping files in your `$AIRFLOW_HOME/plugins` folder.

The python modules in the `plugins` folder get imported, and **macros** and web **views** get integrated to Airflow's main collections and become available for use.

To troubleshoot issues with plugins, you can use the `airflow plugins` command. This command dumps information about loaded plugins.

What for?

Airflow offers a generic toolbox for working with data. Different organizations have different stacks and different needs. Using Airflow plugins can be a way for companies to customize their Airflow installation to reflect their ecosystem.

Plugins can be used as an easy way to write, share and activate new sets of features.

There's also a need for a set of more complex applications to interact with different flavors of data and metadata.

Examples:

- A set of tools to parse Hive logs and expose Hive metadata (CPU /IO / phases/ skew /...)
- An anomaly detection framework, allowing people to collect metrics, set thresholds and alerts
- An auditing tool, helping understand who accesses what
- A config-driven SLA monitoring tool, allowing you to set monitored tables and at what time they should land, alert people, and expose visualizations of outages

Why build on top of Airflow?

Airflow has many components that can be reused when building an application:

- A web server you can use to render your views
- A metadata database to store your models
- Access to your databases, and knowledge of how to connect to them
- An array of workers that your application can push workload to
- Airflow is deployed, you can just piggy back on its deployment logistics
- Basic charting capabilities, underlying libraries and abstractions

When are plugins (re)loaded?

Plugins are by default lazily loaded and once loaded, they are never reloaded (except the UI plugins are automatically loaded in Webserver). To load them at the start of each Airflow process, set `[core] lazy_load_plugins = False` in `airflow.cfg`.

This means that if you make any changes to plugins and you want the webserver or scheduler to use that new code you will need to restart those processes. However, it will not be reflected in new running tasks until after the scheduler boots.

By default, task execution uses forking. This avoids the slowdown associated with creating a new Python interpreter and re-parsing all of Airflow's code and startup routines. This approach offers significant benefits, especially for shorter tasks. This does mean that if you use plugins in your tasks, and want them to update you will either need to restart the worker (if using CeleryExecutor) or scheduler (LocalExecutor). The other option is you can accept the speed hit at start up set the `core.execute_tasks_new_python_interpreter` config setting to True, resulting in launching a whole new python interpreter for tasks.

(Modules only imported by DAG files on the other hand do not suffer this problem, as DAG files are not loaded/parsed in any long-running Airflow process.)

Interface

To create a plugin you will need to derive the `airflow.plugins_manager.AirflowPlugin` class and reference the objects you want to plug into Airflow. Here's what the class you need to derive looks like:

```
class AirflowPlugin:
    # The name of your plugin (str)
    name = None
    # A list of references to inject into the macros namespace
    macros = []
    # A list of dictionaries containing FastAPI app objects and some metadata. See the
    # example below.
    fastapi_apps = []
    # A list of dictionaries containing FastAPI middleware factory objects and some
    # metadata. See the example below.
    fastapi_root_middlewares = []

    # A callback to perform actions when Airflow starts and the plugin is loaded.
    # NOTE: Ensure your plugin has *args, and **kwargs in the method definition
    # to protect against extra parameters injected into the on_load(...)
    # function in future changes
    def on_load(*args, **kwargs):
        # ... perform Plugin boot actions
        pass

    # A list of global operator extra links that can redirect users to
    # external systems. These extra links will be available on the
    # task page in the form of buttons.
    #
    # Note: the global operator extra link can be overridden at each
    # operator level.
    global_operator_extra_links = []

    # A list of operator extra links to override or add operator links
    # to existing Airflow Operators.
    # These extra links will be available on the task page in form of
    # buttons.
    operator_extra_links = []

    # A list of timetable classes to register so they can be used in dags.
    timetables = []

    # A list of Listeners that plugin provides. Listeners can register to
    # listen to particular events that happen in Airflow, like
    # TaskInstance state changes. Listeners are python modules.
    listeners = []
```

You can derive it by inheritance (please refer to the example below). In the example, all options have been defined as class attributes, but you can also define them as properties if you need to perform additional initialization. Please note name inside this class must be specified.

Make sure you restart the webserver and scheduler after making changes to plugins so that they take effect.

Example

The code below defines a plugin that injects a set of illustrative object definitions in Airflow.

```
# This is the class you derive to create a plugin
from airflow.plugins_manager import AirflowPlugin

from fastapi import FastAPI
from fastapi.middleware.trustedhost import TrustedHostMiddleware

# Importing base classes that we need to derive
from airflow.hooks.base import BaseHook
from airflow.providers.amazon.aws.transfers.gcs_to_s3 import GCSToS3Operator

# Will show up under airflow.macros.test_plugin.plugin_macro
# and in templates through {{ macros.test_plugin.plugin_macro }}
def plugin_macro():
    pass

# Creating a FastAPI application to integrate in Airflow Rest API.
app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World from FastAPI plugin"}

app_with_metadata = {"app": app, "url_prefix": "/some_prefix", "name": "Name of the App"}

# Creating a FastAPI middleware that will operates on all the server api requests.
middleware_with_metadata = {
    "middleware": TrustedHostMiddleware,
    "args": [],
    "kwargs": {"allowed_hosts": ["example.com", "*.example.com"]},
    "name": "Name of the Middleware",
}

# Defining the plugin class
class AirflowTestPlugin(AirflowPlugin):
    name = "test_plugin"
    macros = [plugin_macro]
    fastapi_apps = [app_with_metadata]
    fastapi_root_middlewares = [middleware_with_metadata]
```

See also

Define an operator extra link

Exclude views from CSRF protection

We strongly suggest that you should protect all your views with CSRF. But if needed, you can exclude some views using a decorator.

```
from airflow.www.app import csrf

@csrf.exempt
def my_handler():
    # ...
    return "ok"
```

Plugins as Python packages

It is possible to load plugins via `setuptools entrypoint` mechanism. To do this link your plugin using an entrypoint in your package. If the package is installed, Airflow will automatically load the registered plugins from the entrypoint list.

Note

Neither the entrypoint name (eg, `my_plugin`) nor the name of the plugin class will contribute towards the module and class name of the plugin itself.

```
# my_package/my_plugin.py
from airflow.plugins_manager import AirflowPlugin

class MyAirflowPlugin(AirflowPlugin):
    name = "my_namespace"
```

Then inside `pyproject.toml`:

```
[project.entry-points."airflow.plugins"]
my_plugin = "my_package.my_plugin:MyAirflowPlugin"
```

Flask Appbuilder and Flask Blueprints in Airflow 3

Airflow 2 supported Flask Appbuilder views (`appbuilder_views`), Flask AppBuilder menu items (`appbuilder_menu_items`), and Flask Blueprints (`flask_blueprints`) in plugins. These has been superseded by FastAPI apps in Airflow 3. All new plugins should use FastAPI apps (`fastapi_apps`) instead.

However, a compatibility layer is provided for Flask and FAB plugins to ease the transition to Airflow 3 - simply install the FAB provider. Ideally, you should convert your plugins to FastAPI apps (`fastapi_apps`) during the upgrade process, as this compatibility layer is deprecated.

Troubleshooting

You can use the `Flask CLI` to troubleshoot problems. To run this, you need to set the variable `FLASK_APP` to `airflow.www.app:create_app`.

For example, to print all routes, run:

```
FLASK_APP=airflow.www.app:create_app flask routes
```

3.10 Integration

Airflow has a mechanism that allows you to expand its functionality and integrate with other systems.

- *API Authentication backends*
- *Email backends*
- *Executor*
- *Kerberos*
- *Logging*
- *Metrics (statsd)*
- *Operators and hooks*
- *Plugins*
- *Listeners*
- *Secrets backends*
- *Web UI Authentication backends*
- *Serialization*

It also has integration with *Sentry* service for error tracking. Other applications can also integrate using the *REST API*.

3.11 Public Interface of Airflow

The Public Interface of Apache Airflow is the collection of interfaces and behaviors in Apache Airflow whose changes are governed by semantic versioning. A user interacts with Airflow's public interface by creating and managing dags, managing tasks and dependencies, and extending Airflow capabilities by writing new executors, plugins, operators and providers. The Public Interface can be useful for building custom tools and integrations with other systems, and for automating certain aspects of the Airflow workflow.

3.11.1 Using Airflow Public Interfaces

The following are some examples of the public interface of Airflow:

- When you are writing your own operators or hooks. This is commonly done when no hook or operator exists for your use case, or when perhaps when one exists but you need to customize the behavior.
- When writing new *Plugins* that extend Airflow's functionality beyond DAG building blocks. Secrets, Timetables, Triggers, Listeners are all examples of such functionality. This is usually done by users who manage Airflow instances.
- Bundling custom Operators, Hooks, Plugins and releasing them together via providers - this is usually done by those who intend to provide a reusable set of functionality for external services or applications Airflow integrates with.
- Using the taskflow API to write tasks
- Relying on the consistent behavior of Airflow objects

One aspect of “public interface” is extending or using Airflow Python classes and functions. The classes and functions mentioned below can be relied on to maintain backwards-compatible signatures and behaviours within MAJOR version of Airflow. On the other hand, classes and methods starting with `_` (also known as protected Python methods) and `__` (also known as private Python methods) are not part of the Public Airflow Interface and might change at any time.

You can also use Airflow’s Public Interface via the [Stable REST API](#) (based on the OpenAPI specification). For specific needs you can also use the [Airflow Command Line Interface \(CLI\)](#) though its behaviour might change in details (such as output format and available flags) so if you want to rely on those in programmatic way, the Stable REST API is recommended.

3.11.2 Using the Public Interface for DAG Authors

Dags

The DAG is Airflow’s core entity that represents a recurring workflow. You can create a DAG by instantiating the `DAG` class in your DAG file. You can also instantiate them via `DagBag` class that reads dags from a file or a folder. Dags can also have parameters specified via `Param` class.

Airflow has a set of example dags that you can use to learn how to write dags

`airflow.example_dags`

Submodules

`airflow.example_dags.example_asset_alias`

Example DAG for demonstrating the behavior of the AssetAlias feature in Airflow, including conditional and asset expression-based scheduling.

Notes on usage:

Turn on all the DAGs.

Once the “`asset_s3_bucket_producer`” DAG is triggered, the “`asset_s3_bucket_consumer`” DAG should be triggered upon completion. This is because the asset alias “example-alias” is used to add an asset event to the asset “`s3://bucket/my-task`” during the “`produce_asset_events_through_asset_alias`” task. As the DAG “`asset-alias-consumer`” relies on asset alias “example-alias” which was previously unresolved, the DAG “`asset-alias-consumer`” (along with all the DAGs in the same file) will be re-parsed and thus update its schedule to the asset “`s3://bucket/my-task`” and will also be triggered.

Functions

```
produce_asset_events()  
  
produce_asset_events_through_asset_alias(*[  
...])  
consume_asset_event()  
  
consume_asset_event_from_asset_alias(*[, in-  
let_events])
```

Module Contents

```
airflow.example_dags.example_asset_alias.produce_asset_events()  
airflow.example_dags.example_asset_alias.produce_asset_events_through_asset_alias(*  
    (Keyword-  
     only  
     parame-  
     ters  
     separator  
     (PEP  
     3102)),  
    out-  
    let_events=None)  
  
airflow.example_dags.example_asset_alias.consume_asset_event()  
airflow.example_dags.example_asset_alias.consume_asset_event_from_asset_alias(*, in-  
    let_events=None)
```

airflow.example_dags.example_asset_alias_with_no_taskflow

Example DAG for demonstrating the behavior of the AssetAlias feature in Airflow, including conditional and asset expression-based scheduling.

Notes on usage:

Turn on all the DAGs.

Before running any DAG, the schedule of the “asset_alias_example_alias_consumer_with_no_taskflow” DAG will show as “unresolved AssetAlias”. This is expected because the asset alias has not been resolved into any asset yet.

Once the “asset_s3_bucket_producer_with_no_taskflow” DAG is triggered, the “asset_s3_bucket_consumer_with_no_taskflow” DAG should be triggered upon completion. This is because the asset alias “example-alias-no-taskflow” is used to add an asset event to the asset “s3://bucket/my-task-with-no-taskflow” during the “produce_asset_events_through_asset_alias_with_no_taskflow” task. Also, the schedule of the “asset_alias_example_alias_consumer_with_no_taskflow” DAG should change to “Asset” as the asset alias “example-alias-no-taskflow” is now resolved to the asset “s3://bucket/my-task-with-no-taskflow” and this DAG should also be triggered.

Functions

```
produce_asset_events()  
produce_asset_events_through_asset_alias wi  
consume_asset_event()  
consume_asset_event_from_asset_alias(*[, in-  
let_events])
```

Module Contents

```
airflow.example_dags.example_asset_alias_with_no_taskflow.produce_asset_events()  
airflow.example_dags.example_asset_alias_with_no_taskflow.produce_asset_events_through_asset_alias_with_no_taskflow()  
  
airflow.example_dags.example_asset_alias_with_no_taskflow.consume_asset_event()  
airflow.example_dags.example_asset_alias_with_no_taskflow.consume_asset_event_from_asset_alias(*, in-  
let_events=)
```

airflow.example_dags.example_asset_decorator

Functions

```
asset1_producer()  
asset2_producer(self, context, asset1_producer)  
consumes_asset_decorator()
```

Module Contents

```
airflow.example_dags.example_asset_decorator.asset1_producer()  
airflow.example_dags.example_asset_decorator.asset2_producer(self, context, asset1_producer)  
airflow.example_dags.example_asset_decorator.consumes_asset_decorator()
```

airflow.example_dags.example_asset_with_watchers

Example DAG for demonstrating the usage of event driven scheduling using assets and triggers.

Attributes

```
file_path  
trigger  
asset
```

Functions

```
test_task()
```

Module Contents

```
airflow.example_dags.example_asset_with_watchers.file_path = '/tmp/test'  
airflow.example_dags.example_asset_with_watchers.trigger  
airflow.example_dags.example_asset_with_watchers.asset  
airflow.example_dags.example_asset_with_watchers.test_task()
```

airflow.example_dags.example_assets

Example DAG for demonstrating the behavior of the Assets feature in Airflow, including conditional and asset expression-based scheduling.

Notes on usage:

Turn on all the DAGs.

`asset_produces_1` is scheduled to run daily. Once it completes, it triggers several DAGs due to its asset being updated. `asset_consumes_1` is triggered immediately, as it depends solely on the asset produced by `asset_produces_1`. `consume_1_or_2_with_asset_expressions` will also be triggered, as its condition of either `asset_produces_1` or `asset_produces_2` being updated is satisfied with `asset_produces_1`.

`asset_consumes_1_and_2` will not be triggered after `asset_produces_1` runs because it requires the asset from `asset_produces_2`, which has no schedule and must be manually triggered.

After manually triggering `asset_produces_2`, several DAGs will be affected. `asset_consumes_1_and_2` should run because both its asset dependencies are now met. `consume_1_and_2_with_asset_expressions` will be triggered, as it requires both `asset_produces_1` and `asset_produces_2` assets to be updated. `consume_1_or_2_with_asset_expressions` will be triggered again, since it's conditionally set to run when either asset is updated.

`consume_1_or_both_2_and_3_with_asset_expressions` demonstrates complex asset dependency logic. This DAG triggers if `asset_produces_1` is updated or if both `asset_produces_2` and `dag3_asset` are updated. This example highlights the capability to combine updates from multiple assets with logical expressions for advanced scheduling.

`conditional_asset_and_time_based_timetable` illustrates the integration of time-based scheduling with asset dependencies. This DAG is configured to execute either when both `asset_produces_1` and `asset_produces_2` assets have been updated or according to a specific cron schedule, showcasing Airflow's versatility in handling mixed triggers for asset and time-based scheduling.

The DAGs `asset_consumes_1_never_scheduled` and `asset_consumes_unknown_never_scheduled` will not run automatically as they depend on assets that do not get updated or are not produced by any scheduled tasks.

Attributes

```
dag1_asset  
dag2_asset  
dag3_asset
```

Module Contents

`airflow.example_dags.example_assets.dag1_asset`
`airflow.example_dags.example_assets.dag2_asset`
`airflow.example_dags.example_assets.dag3_asset`

airflow.example_dags.example_branch_labels

Example DAG demonstrating the usage of labels with different branches.

Attributes

`ingest`

Module Contents

`airflow.example_dags.example_branch_labels.ingest`

airflow.example_dags.example_branch_python_dop_operator_3

Example DAG demonstrating the usage of `@task.branch` TaskFlow API decorator with `depends_on_past=True`, where tasks may be run or skipped on alternating runs.

Attributes

`cond`

Functions

`should_run(**kwargs)`

Determine which empty_task should be run based on if the logical date minute is even or odd.

Module Contents

`airflow.example_dags.example_branch_python_dop_operator_3.should_run(**kwargs)`

Determine which empty_task should be run based on if the logical date minute is even or odd.

Parameters

`kwargs (dict)` – Context

Returns

Id of the task to run

Return type

str

`airflow.example_dags.example_branch_python_dop_operator_3.cond = 'empty_task_1'`

[airflow.example_dags.example_complex](#)

Example Airflow DAG that shows the complex DAG structure.

Attributes

```
create_entry_group
```

Module Contents

`airflow.example_dags.example_complex.create_entry_group`

[airflow.example_dags.example_custom_weight](#)

Example DAG demonstrating the usage of the CustomWeightRule.

Attributes

```
start
```

Module Contents

`airflow.example_dags.example_custom_weight.start`

[airflow.example_dags.example_display_name](#)

Attributes

```
example_dag
```

Functions

```
example_display_name()
```

Module Contents

`airflow.example_dags.example_display_name.example_display_name()`

`airflow.example_dags.example_display_name.example_dag = None`

`airflow.example_dags.example_dynamic_task_mapping`

Example DAG demonstrating the usage of dynamic task mapping.

Functions

```
add_one(x)
```

```
get_nums()
```

Module Contents

```
airflow.example_dags.example_dynamic_task_mapping.add_one(x)
```

```
airflow.example_dags.example_dynamic_task_mapping.get_nums()
```

`airflow.example_dags.example_dynamic_task_mapping_with_no_taskflow_operators`

Example DAG demonstrating the usage of dynamic task mapping with non-TaskFlow operators.

Attributes

```
add_one_task
```

Classes

| | |
|-----------------------------|---|
| <code>AddOneOperator</code> | A custom operator that adds one to the input. |
|-----------------------------|---|

Module Contents

```
class airflow.example_dags.example_dynamic_task_mapping_with_no_taskflow_operators.AddOneOperator(value,  
**kwargs)
```

Bases: `airflow.models.baseoperator.BaseOperator`

A custom operator that adds one to the input.

value

execute(context)

Derive when creating an operator.

Context is the same dictionary used as when rendering jinja templates.

Refer to `get_template_context` for more context.

```
class airflow.example_dags.example_dynamic_task_mapping_with_no_taskflow_operators.SumItOperator(values,
**kwargs):
    Bases: airflow.models.baseoperator.BaseOperator
    A custom operator that sums the input.
    template_fields = ('values',)
    values
    execute(context)
        Derive when creating an operator.
        Context is the same dictionary used as when rendering jinja templates.
        Refer to get_template_context for more context.
airflow.example_dags.example_dynamic_task_mapping_with_no_taskflow_operators.add_one_task
```

airflow.example_dags.example_inlet_event_extra

Example DAG to demonstrate reading asset events annotated with extra information.

Also see examples in `example_outlet_event_extra.py`.

Attributes

```
asset
```

Functions

```
read_asset_event(*[, inlet_events])
```

Module Contents

```
airflow.example_dags.example_inlet_event_extra.asset
```

```
airflow.example_dags.example_inlet_event_extra.read_asset_event(*, inlet_events=None)
```

airflow.example_dags.example_kubernetes_executor

This is an example dag for using a Kubernetes Executor Configuration.

Attributes

```
log
```

```
k8s
```

continues on next page [—](#)

Table 21 – continued from previous page

```
start_task_executor_config
```

Module Contents

`airflow.example_dags.example_kubernetes_executor.log`

`airflow.example_dags.example_kubernetes_executor.k8s = None`

`airflow.example_dags.example_kubernetes_executor.start_task_executor_config`

`airflow.example_dags.example_latest_only_with_trigger`

Example LatestOnlyOperator and TriggerRule interactions

Attributes

```
latest_only
```

Module Contents

`airflow.example_dags.example_latest_only_with_trigger.latest_only`

`airflow.example_dags.example_local_kubernetes_executor`

This is an example dag for using a Local Kubernetes Executor Configuration.

Attributes

```
log
```

```
worker_container_repository
```

```
worker_container_tag
```

```
k8s
```

```
start_task_executor_config
```

Module Contents

`airflow.example_dags.example_local_kubernetes_executor.log`

`airflow.example_dags.example_local_kubernetes_executor.worker_container_repository`

`airflow.example_dags.example_local_kubernetes_executor.worker_container_tag`

```
airflow.example_dags.example_local_kubernetes_executor.k8s = None  
airflow.example_dags.example_local_kubernetes_executor.start_task_executor_config
```

airflow.example_dags.example_nested_branch_dag

Example DAG demonstrating a workflow with nested branching. The join tasks are created with `none_failed_min_one_success` trigger rule such that they are skipped whenever their corresponding branching tasks are skipped.

Functions

```
branch(task_id_to_return)
```

Module Contents

```
airflow.example_dags.example_nested_branch_dag.branch(task_id_to_return)
```

airflow.example_dags.example_outlet_event_extra

Example DAG to demonstrate annotating an asset event with extra information.

Also see examples in `example_inlet_event_extra.py`.

Attributes

```
asset
```

Functions

```
asset_with_extra_by_yield()  
asset_with_extra_by_context(*[outlet_events])
```

Module Contents

```
airflow.example_dags.example_outlet_event_extra.asset  
airflow.example_dags.example_outlet_event_extra.asset_with_extra_by_yield()  
airflow.example_dags.example_outlet_event_extra.asset_with_extra_by_context(*outlet_events=None)
```

airflow.example_dags.example_params_trigger_ui

Example DAG demonstrating the usage DAG params to model a trigger UI with a user form.

This example DAG generates greetings to a list of provided names in selected languages in the logs.

Functions

```
get_names(**kwargs)
```

Module Contents

```
airflow.example_dags.example_params_trigger_ui.get_names(**kwargs)
```

airflow.example_dags.example_params_ui_tutorial

DAG demonstrating various options for a trigger form generated by DAG params.

The DAG attribute *params* is used to define a default dictionary of parameters which are usually passed to the DAG and which are used to render a trigger form.

Functions

```
show_params(**kwargs)
```

Module Contents

```
airflow.example_dags.example_params_ui_tutorial.show_params(**kwargs)
```

airflow.example_dags.example_passing_params_via_test_command

Example DAG demonstrating the usage of the params arguments in templated arguments.

Attributes

```
run_this
```

Functions

```
my_py_command(params[, test_mode, task])  
print_env_vars([test_mode])
```

Print out the "foo" param passed in via
Print out the "foo" param passed in via

Module Contents

```
airflow.example_dags.example_passing_params_via_test_command.my_py_command(params,  
                           test_mode=None,  
                           task=None)  
  
    Print out the “foo” param passed in via airflow tasks test example_passing_params_via_test_command run_this  
    <date> -t {'foo': "bar"}  
  
airflow.example_dags.example_passing_params_via_test_command.print_env_vars(test_mode=None)  
    Print out the “foo” param passed in via airflow tasks test example_passing_params_via_test_command  
    env_var_test_task <date> -env-vars {'foo': "bar"}  
  
airflow.example_dags.example_passing_params_via_test_command.run_this = 1
```

airflow.example_dags.example_setup_teardown

Example DAG demonstrating the usage of setup and teardown tasks.

Attributes

```
root_setup
```

Module Contents

```
airflow.example_dags.example_setup_teardown.root_setup
```

airflow.example_dags.example_setup_teardown_taskflow

Example DAG demonstrating the usage of setup and teardown tasks.

Functions

```
my_first_task()
```

Module Contents

```
airflow.example_dags.example_setup_teardown_taskflow.my_first_task()
```

airflow.example_dags.example_simplest_dag

Example DAG demonstrating the simplest use of the @dag decorator.

Functions

```
example_simplest_dag()
```

Module Contents

`airflow.example_dags.example_simplest_dag.example_simplest_dag()`

`airflow.example_dags.example_skip_dag`

Example DAG demonstrating the EmptyOperator and a custom EmptySkipOperator which skips by default.

Classes

| | |
|--------------------------------|---|
| <code>EmptySkipOperator</code> | Empty operator which always skips the task. |
|--------------------------------|---|

Functions

| | |
|---|--|
| <code>create_test_pipeline(suffix, trigger_rule)</code> | Instantiate a number of operators for the given DAG. |
|---|--|

Module Contents

`class airflow.example_dags.example_skip_dag.EmptySkipOperator(**kwargs)`

Bases: `airflow.models.baseoperator.BaseOperator`

Empty operator which always skips the task.

`ui_color = '#e8b7e4'`

`execute(context)`

Derive when creating an operator.

Context is the same dictionary used as when rendering jinja templates.

Refer to `get_template_context` for more context.

`airflow.example_dags.example_skip_dag.create_test_pipeline(suffix, trigger_rule)`

Instantiate a number of operators for the given DAG.

Parameters

- `suffix (str)` – Suffix to append to the operator task_ids
- `trigger_rule (str)` – TriggerRule for the join task
- `dag (DAG)` – The DAG to run the operators on

`airflow.example_dags.example_task_group`

Example DAG demonstrating the usage of the TaskGroup.

Attributes

| |
|--------------------|
| <code>start</code> |
|--------------------|

Module Contents

`airflow.example_dags.example_task_group.start`

`airflow.example_dags.example_task_group_decorator`

Example DAG demonstrating the usage of the `@taskgroup` decorator.

Attributes

`start_task`

Functions

| | |
|---|---------------------------------------|
| <code>task_start()</code> | Empty Task which is First Task of Dag |
| <code>task_1(value)</code> | Empty Task1 |
| <code>task_2(value)</code> | Empty Task2 |
| <code>task_3(value)</code> | Empty Task3 |
| <code>task_end()</code> | Empty Task which is Last Task of Dag |
| <code>task_group_function(value)</code> | TaskGroup for grouping related Tasks |

Module Contents

`airflow.example_dags.example_task_group_decorator.task_start()`

Empty Task which is First Task of Dag

`airflow.example_dags.example_task_group_decorator.task_1(value)`

Empty Task1

`airflow.example_dags.example_task_group_decorator.task_2(value)`

Empty Task2

`airflow.example_dags.example_task_group_decorator.task_3(value)`

Empty Task3

`airflow.example_dags.example_task_group_decorator.task_end()`

Empty Task which is Last Task of Dag

`airflow.example_dags.example_task_group_decorator.task_group_function(value)`

TaskGroup for grouping related Tasks

`airflow.example_dags.example_task_group_decorator.start_task = '[Task_start]'`

airflow.example_dags.example_time_delta_sensor_async

Example DAG demonstrating TimeDeltaSensorAsync, a drop in replacement for TimeDeltaSensor that defers and doesn't occupy a worker slot while it waits

Attributes

```
wait
```

Module Contents

`airflow.example_dags.example_time_delta_sensor_async.wait`

airflow.example_dags.example_trigger_target_dag

Example usage of the TriggerDagRunOperator. This example holds 2 DAGs: 1. 1st DAG (example_trigger_controller_dag) holds a TriggerDagRunOperator, which will trigger the 2nd DAG 2. 2nd DAG (example_trigger_target_dag) which will be triggered by the TriggerDagRunOperator in the 1st DAG

Attributes

```
run_this
```

Functions

`run_this_func([dag_run])`

Print the payload "message" passed to the DagRun conf attribute.

Module Contents

`airflow.example_dags.example_trigger_target_dag.run_this_func(dag_run=None)`

Print the payload "message" passed to the DagRun conf attribute.

Parameters

`dag_run` – The DagRun object

`airflow.example_dags.example_trigger_target_dag.run_this = None`

airflow.example_dags.example_workday_timetable

airflow.example_dags.example_xcom

Example DAG demonstrating the usage of XComs.

Attributes

`value_1`

`value_2`

`bash_push`

Functions

| | |
|--|--|
| <code>push([ti])</code> | Pushes an XCom without a specific target |
| <code>push_by_returning()</code> | Pushes an XCom without a specific target, just by returning it |
| <code>puller(pulled_value_2[, ti])</code> | Pull all previously pushed XComs and check if the pushed values match the pulled values. |
| <code>pull_value_from_bash_push([ti])</code> | |

Module Contents

`airflow.example_dags.example_xcom.value_1 = [1, 2, 3]`

`airflow.example_dags.example_xcom.value_2`

`airflow.example_dags.example_xcom.push(ti=None)`

Pushes an XCom without a specific target

`airflow.example_dags.example_xcom.push_by_returning()`

Pushes an XCom without a specific target, just by returning it

`airflow.example_dags.example_xcom.puller(pulled_value_2, ti=None)`

Pushes an XCom without a specific target, just by returning it

`airflow.example_dags.example_xcom.pull_value_from_bash_push(ti=None)`

`airflow.example_dags.example_xcom.bash_push`

`airflow.example_dags.example_xcomargs`

Example DAG demonstrating the usage of the XComArgs.

Attributes

`log`

`bash_op1`

Functions

| | |
|---------------------------------------|----------------|
| <code>generate_value()</code> | Empty function |
| <code>print_value(value[, ts])</code> | Empty function |

Module Contents

`airflow.example_dags.example_xcomargs.log`
`airflow.example_dags.example_xcomargs.generate_value()`
 Empty function
`airflow.example_dags.example_xcomargs.print_value(value, ts=None)`
 Empty function
`airflow.example_dags.example_xcomargs.bash_op1`

`airflow.example_dags.libs`

Submodules

`airflow.example_dags.libs.helper`

Functions

| |
|----------------------------|
| <code>print_stuff()</code> |
|----------------------------|

Module Contents

`airflow.example_dags.libs.helper.print_stuff()`

`airflow.example_dags.plugins`

Submodules

`airflow.example_dags.plugins.decreasing_priority_weight_strategy`

Classes

| | |
|---|--|
| <code>DecreasingPriorityStrategy</code> | A priority weight strategy that decreases the priority weight with each attempt of the DAG task. |
| <code>DecreasingPriorityWeightStrategyPlugin</code> | Class used to define AirflowPlugin. |

Module Contents

`class airflow.example_dags.plugins.decreasing_priority_weight_strategy.
 DecreasingPriorityStrategy`

Bases: `airflow.task.priority_strategy.PriorityWeightStrategy`

A priority weight strategy that decreases the priority weight with each attempt of the DAG task.

get_weight(ti)

Get the priority weight of a task.

```
class airflow.example_dags.plugins.decreasing_priority_weight_strategy.  
DecreasingPriorityWeightStrategyPlugin
```

Bases: airflow.plugins_manager.AirflowPlugin

Class used to define AirflowPlugin.

```
name = 'decreasing_priority_weight_strategy_plugin'
```

```
priority_weight_strategies
```

airflow.example_dags.plugins.event_listener**Functions**

| | |
|--|---|
| <code>on_task_instance_running(previous_state, task_instance)</code> | Called when task state changes to RUNNING. |
| <code>on_task_instance_success(previous_state, task_instance)</code> | Called when task state changes to SUCCESS. |
| <code>on_task_instance_failed(previous_state, task_instance, ...)</code> | Called when task state changes to FAILED. |
| <code>on_dag_run_success(dag_run, msg)</code> | This method is called when dag run state changes to SUCCESS. |
| <code>on_dag_run_failed(dag_run, msg)</code> | This method is called when dag run state changes to FAILED. |
| <code>on_dag_run_running(dag_run, msg)</code> | This method is called when dag run state changes to RUNNING. |

Module Contents

```
airflow.example_dags.plugins.event_listener.on_task_instance_running(previous_state,  
task_instance)
```

Called when task state changes to RUNNING.

previous_task_state and task_instance object can be used to retrieve more information about current task_instance that is running, its dag_run, task and dag information.

```
airflow.example_dags.plugins.event_listener.on_task_instance_success(previous_state,  
task_instance)
```

Called when task state changes to SUCCESS.

previous_task_state and task_instance object can be used to retrieve more information about current task_instance that has succeeded, its dag_run, task and dag information.

A RuntimeTaskInstance is provided in most cases, except when the task's state change is triggered through the API. In that case, the TaskInstance available on the API server will be provided instead.

```
airflow.example_dags.plugins.event_listener.on_task_instance_failed(previous_state,
task_instance, error)
```

Called when task state changes to FAILED.

previous_task_state, task_instance object and error can be used to retrieve more information about current task_instance that has failed, its dag_run, task and dag information.

A RuntimeTaskInstance is provided in most cases, except when the task's state change is triggered through the API. In that case, the TaskInstance available on the API server will be provided instead.

```
airflow.example_dags.plugins.event_listener.on_dag_run_success(dag_run, msg)
```

This method is called when dag run state changes to SUCCESS.

```
airflow.example_dags.plugins.event_listener.on_dag_run_failed(dag_run, msg)
```

This method is called when dag run state changes to FAILED.

```
airflow.example_dags.plugins.event_listener.on_dag_run_running(dag_run, msg)
```

This method is called when dag run state changes to RUNNING.

airflow.example_dags.plugins.listener_plugin

Classes

| | |
|---------------------------------|-------------------------------------|
| <i>MetadataCollectionPlugin</i> | Class used to define AirflowPlugin. |
|---------------------------------|-------------------------------------|

Module Contents

```
class airflow.example_dags.plugins.listener_plugin.MetadataCollectionPlugin
```

Bases: airflow.plugins_manager.AirflowPlugin

Class used to define AirflowPlugin.

```
name = 'MetadataCollectionPlugin'
```

```
listeners
```

airflow.example_dags.plugins.workday

Plugin to demonstrate timetable registration and accommodate example DAGs.

Attributes

| |
|------------|
| <i>log</i> |
|------------|

| |
|-------------------------|
| <i>holiday_calendar</i> |
|-------------------------|

Classes

| | |
|-------------------------------------|--|
| <code>AfterWorkdayTimetable</code> | Protocol that all Timetable classes are expected to implement. |
| <code>WorkdayTimetablePlugin</code> | Class used to define AirflowPlugin. |

Module Contents

`airflow.example_dags.plugins.workday.log`

`airflow.example_dags.plugins.workday.holiday_calendar`

`class airflow.example_dags.plugins.workday.AfterWorkdayTimetable`

Bases: `airflow.timetables.base.Timetable`

Protocol that all Timetable classes are expected to implement.

`get_next_workday(d, incr=1)`

`infer_manual_data_interval(run_after)`

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

`next_dagrun_info(*, last_automated_data_interval, restriction)`

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- `last_automated_data_interval` (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG's last scheduled or backfilled run (manual runs not considered).
- `restriction` (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

`airflow.timetables.base.DagRunInfo | None`

`class airflow.example_dags.plugins.workday.WorkdayTimetablePlugin`

Bases: `airflow.plugins_manager.AirflowPlugin`

Class used to define AirflowPlugin.

`name = 'workday_timetable_plugin'`

`timetables`

airflow.example_dags.tutorial

Tutorial Documentation Documentation that goes along with the Airflow tutorial located [here](<https://airflow.apache.org/tutorial.html>)

Attributes

```
t1
```

Module Contents

```
airflow.example_dags.tutorial.t1
```

airflow.example_dags.tutorial_dag

DAG Tutorial Documentation This DAG is demonstrating an Extract -> Transform -> Load pipeline

Functions

```
extract(**kwargs)
```

Module Contents

```
airflow.example_dags.tutorial_dag.extract(**kwargs)
```

airflow.example_dags.tutorial_objectstorage

Attributes

```
API
```

```
aq_fields
```

```
base
```

Functions

```
tutorial_objectstorage()
```

Object Storage Tutorial Documentation

Module Contents

```
airflow.example_dags.tutorial_objectstorage.API = 'https://opendata.fmi.fi/timeseries'
```

```
airflow.example_dags.tutorial_objectstorage.aq_fields
```

`airflow.example_dags.tutorial_objectstorage.base`

`airflow.example_dags.tutorial_objectstorage.tutorial_objectstorage()`

Object Storage Tutorial Documentation This is a tutorial DAG to showcase the usage of the Object Storage API. Documentation that goes along with the Airflow Object Storage tutorial is located [here](<https://airflow.apache.org/docs/apache-airflow/stable/tutorial/objectstorage.html>)

airflow.example_dags.tutorial_taskflow_api

Functions

`tutorial_taskflow_api()`

TaskFlow API Tutorial Documentation

Module Contents

`airflow.example_dags.tutorial_taskflow_api.tutorial_taskflow_api()`

TaskFlow API Tutorial Documentation This is a simple data pipeline example which demonstrates the use of the TaskFlow API using three simple tasks for Extract, Transform, and Load. Documentation that goes along with the Airflow TaskFlow API tutorial is located [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html)

airflow.example_dags.tutorial_taskflow_api_virtualenv

Attributes

`log`

`tutorial_dag`

Functions

`tutorial_taskflow_api_virtualenv()`

TaskFlow API example using virtualenv

Module Contents

`airflow.example_dags.tutorial_taskflow_api_virtualenv.log`

`airflow.example_dags.tutorial_taskflow_api_virtualenv.tutorial_taskflow_api_virtualenv()`

TaskFlow API example using virtualenv This is a simple data pipeline example which demonstrates the use of the TaskFlow API using three simple tasks for Extract, Transform, and Load.

`airflow.example_dags.tutorial_taskflow_api_virtualenv.tutorial_dag = None`

airflow.example_dags.tutorial_taskflow_templates

Functions

`tutorial_taskflow_templates()`

TaskFlow API Tutorial Documentation

Module Contents

`airflow.example_dags.tutorial_taskflow_templates.tutorial_taskflow_templates()`

TaskFlow API Tutorial Documentation This is a simple data pipeline example which demonstrates the use of the templates in the TaskFlow API. Documentation that goes along with the Airflow TaskFlow API tutorial is located [here](https://airflow.apache.org/docs/apache-airflow/stable/tutorial_taskflow_api.html)

You can read more about dags in *Dags*.

References for the modules used in dags are here:

`airflow.models.dag`

Attributes

`log``AssetT``TAG_MAX_LEN``DagStateChangeCallback``ScheduleInterval``ScheduleArg``dag`

Exceptions

`InconsistentDataInterval`

Exception raised when a model populates data interval fields incorrectly.

Classes

`DAG`

A dag (directed acyclic graph) is a collection of tasks with directional dependencies.

`DagTag`

A tag name per dag, to allow quick filtering in the DAG view.

`DagOwnerAttributes`

Table defining different owner attributes.

`DagModel`

Table containing DAG properties.

Functions

| | |
|---|--|
| <code>get_last_dagrun(dag_id, session[, ...])</code> | Return the last dag run for a dag, None if there was none. |
| <code>get_asset_triggered_next_run_info(dag_ids, *, session)</code> | Get next run info for a list of dag_ids. |

Module Contents

`airflow.models.dag.log`

`airflow.models.dag.AssetT`

`airflow.models.dag.TAG_MAX_LEN = 100`

`airflow.models.dag.DagStateChangeCallback`

`airflow.models.dag.ScheduleInterval`

`airflow.models.dag.ScheduleArg`

`exception airflow.models.dag.InconsistentDataInterval(instance, start_field_name, end_field_name)`

Bases: `airflow.exceptions.AirflowException`

Exception raised when a model populates data interval fields incorrectly.

The data interval fields should either both be None (for runs scheduled prior to AIP-39), or both be datetime (for runs scheduled after AIP-39 is implemented). This is raised if exactly one of the fields is None.

`__str__()`

Return str(self).

`airflow.models.dag.get_last_dagrun(dag_id, session, include_manually_triggered=False)`

Return the last dag run for a dag, None if there was none.

Last dag run can be any type of run e.g. scheduled or backfilled. Overridden DagRuns are ignored.

`airflow.models.dag.get_asset_triggered_next_run_info(dag_ids, *, session)`

Get next run info for a list of dag_ids.

Given a list of dag_ids, get string representing how close any that are asset triggered are their next run, e.g. “1 of 2 assets updated”.

`airflow.models.dag.dag`

`class airflow.models.dag.DAG(context=None)`

Bases: `airflow.sdkdefinitions.dag.DAG, airflow.utils.log.logging_mixin.LoggingMixin`

A dag (directed acyclic graph) is a collection of tasks with directional dependencies.

A dag also has a schedule, a start date and an end date (optional). For each schedule, (say daily or hourly), the DAG needs to run each individual tasks as their dependencies are met. Certain tasks have the property of

depending on their own past, meaning that they can't run until their previous schedule (and upstream tasks) are completed.

DAGs essentially act as namespaces for tasks. A task_id can only be added once to a DAG.

Note that if you plan to use time zones all the dates provided should be pendulum dates. See *Time zone aware dags*.

Added in version 2.4: The `schedule` argument to specify either time-based scheduling logic (timetable), or asset-driven triggers.

Changed in version 3.0: The default value of `schedule` has been changed to `None` (no schedule). The previous default was `timedelta(days=1)`.

Parameters

- **`dag_id`** – The id of the DAG; must consist exclusively of alphanumeric characters, dashes, dots and underscores (all ASCII)
- **`description`** – The description for the DAG to e.g. be shown on the webserver
- **`schedule`** – If provided, this defines the rules according to which DAG runs are scheduled. Possible values include a cron expression string, timedelta object, Timetable, or list of Asset objects. See also *Customizing DAG Scheduling with Timetables*.
- **`start_date`** – The timestamp from which the scheduler will attempt to backfill. If this is not provided, backfilling must be done manually with an explicit time range.
- **`end_date`** – A date beyond which your DAG won't run, leave to None for open-ended scheduling.
- **`template_searchpath`** – This list of folders (non-relative) defines where jinja will look for your templates. Order matters. Note that jinja/airflow includes the path of your DAG file by default
- **`template_undefined`** – Template undefined type.
- **`user_defined_macros`** – a dictionary of macros that will be exposed in your jinja templates. For example, passing `dict(foo='bar')` to this argument allows you to `{{ foo }}` in all jinja templates related to this DAG. Note that you can pass any type of object here.
- **`user_defined_filters`** – a dictionary of filters that will be exposed in your jinja templates. For example, passing `dict(hello=lambda name: 'Hello %s' % name)` to this argument allows you to `{{ 'world' | hello }}` in all jinja templates related to this DAG.
- **`default_args`** – A dictionary of default parameters to be used as constructor keyword parameters when initialising operators. Note that operators have the same hook, and precede those defined here, meaning that if your dict contains '`depends_on_past`': `True` here and '`depends_on_past`': `False` in the operator's call `default_args`, the actual value will be `False`.
- **`params`** – a dictionary of DAG level parameters that are made accessible in templates, namespaced under `params`. These params can be overridden at the task level.
- **`max_active_tasks`** – the number of task instances allowed to run concurrently
- **`max_active_runs`** – maximum number of active DAG runs, beyond this number of DAG runs in a running state, the scheduler won't create new active DAG runs
- **`max_consecutive_failed_dag_runs`** – (experimental) maximum number of consecutive failed DAG runs, beyond this the scheduler will disable the DAG
- **`dagrun_timeout`** – Specify the duration a DagRun should be allowed to run before it times out or fails. Task instances that are running when a DagRun is timed out will be marked as skipped.

- **sla_miss_callback** – DEPRECATED - The SLA feature is removed in Airflow 3.0, to be replaced with a new implementation in 3.1
- **catchup** – Perform scheduler catchup (or only run latest)? Defaults to False
- **on_failure_callback** – A function or list of functions to be called when a DagRun of this dag fails. A context dictionary is passed as a single parameter to this function.
- **on_success_callback** – Much like the `on_failure_callback` except that it is executed when the dag succeeds.
- **access_control** – Specify optional DAG-level actions, e.g., “{‘role1’: {‘can_read’}, ‘role2’: {‘can_read’, ‘can_edit’, ‘can_delete’}}” or it can specify the resource name if there is a DAGs Run resource, e.g., “{‘role1’: {‘DAG Runs’: {‘can_create’}}}, ‘role2’: {‘DAGs’: {‘can_read’, ‘can_edit’, ‘can_delete’}}”
- **is_paused_upon_creation** – Specifies if the dag is paused when created for the first time. If the dag exists already, this flag will be ignored. If this optional parameter is not specified, the global config setting will be used.
- **jinja_environment_kwargs** – additional configuration options to be passed to Jinja Environment for template rendering

Example: to avoid Jinja from removing a trailing newline from template strings

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

See: [Jinja Environment documentation](#)

- **render_template_as_native_obj** – If True, uses a `Jinja NativeEnvironment` to render templates as native Python types. If False, a `Jinja Environment` is used to render templates as string values.
- **tags** – List of tags to help filtering DAGs in the UI.
- **owner_links** – Dict of owners and their links, that will be clickable on the DAGs view UI. Can be used as an HTTP link (for example the link to your Slack channel), or a mailto link. e.g: {“dag_owner”: “<https://airflow.apache.org/>”}
- **auto_register** – Automatically register this DAG when it is used in a `with` block
- **fail_fast** – Fails currently running tasks when task in DAG fails. **Warning:** A fail fast dag can only have tasks with the default trigger rule (“all_success”). An exception will be thrown if any task in a fail fast dag has a non default trigger rule.
- **dag_display_name** – The display name of the DAG which appears on the UI.

```
partial: bool = False  
  
last_loaded: datetime.datetime | None  
  
max_consecutive_failed_dag_runs: int  
  
property safe_dag_id
```

validate()

Validate the DAG has a coherent setup.

This is called by the DAG bag before bagging the DAG.

validate_executor_field()**next_dagrun_info(*last_automated_dagrun*, *, *restricted*=True)**

Get information about the next DagRun of this dag after `date_last_automated_dagrun`.

This calculates what time interval the next DagRun should operate on (its logical date) and when it can be scheduled, according to the dag's timetable, `start_date`, `end_date`, etc. This doesn't check max active run or any other "max_active_tasks" type limits, but only performs calculations based on the various date and interval fields of this dag and its tasks.

Parameters

- **`last_automated_dagrun`** (`None` / `airflow.timetables.base.DataInterval`) – The `max(logical_date)` of existing "automated" DagRuns for this dag (scheduled or backfill, but not manual).
- **`restricted` (bool)** – If set to `False` (default is `True`), ignore `start_date`, `end_date`, and `catchup` specified on the DAG or tasks.

Returns

`DagRunInfo` of the next dagrun, or `None` if a dagrun is not going to be scheduled.

Return type

`airflow.timetables.base.DagRunInfo` | `None`

iter_dagrun_infos_between(*earliest*, *latest*, *, *align*=True)

Yield `DagRunInfo` using this DAG's timetable between given interval.

`DagRunInfo` instances yielded if their `logical_date` is not earlier than `earliest`, nor later than `latest`. The instances are ordered by their `logical_date` from earliest to latest.

If `align` is `False`, the first run will happen immediately on `earliest`, even if it does not fall on the logical timetable schedule. The default is `True`.

Example: A DAG is scheduled to run every midnight (`0 0 * * *`). If `earliest` is `2021-06-03 23:00:00`, the first `DagRunInfo` would be `2021-06-03 23:00:00` if `align=False`, and `2021-06-04 00:00:00` if `align=True`.

get_last_dagrun(*session*=`NEW_SESSION`, *include_manually_triggered*=False)**has_dag_runs(*session*=`NEW_SESSION`, *include_manually_triggered*=True)****property `dag_id`: str****property `timetable_summary`: str****get_concurrency_reached(*session*=`NEW_SESSION`)**

Return a boolean indicating whether the `max_active_tasks` limit for this DAG has been reached.

get_is_active(*session=NEW_SESSION*)

Return a boolean indicating whether this DAG is active.

get_is_stale(*session=NEW_SESSION*)

Return a boolean indicating whether this DAG is stale.

get_is_paused(*session=NEW_SESSION*)

Return a boolean indicating whether this DAG is paused.

get_bundle_name(*session=NEW_SESSION*)

Return the bundle name this DAG is in.

get_bundle_version(*session=NEW_SESSION*)

Return the bundle version that was seen when this dag was processed.

classmethod get_serialized_fields()

Stringified DAGs and operators contain exactly these fields.

get_active_runs()

Return a list of dag run logical dates currently running.

Returns

List of logical dates

static fetch_dagrun(*dag_id, run_id, session=NEW_SESSION*)

Return the dag run for a given run_id if it exists, otherwise none.

Parameters

- **dag_id** (*str*) – The dag_id of the DAG to find.
- **run_id** (*str*) – The run_id of the DagRun to find.
- **session** (*sqlalchemy.orm.Session*)

Returns

The DagRun if found, otherwise None.

Return type

airflow.models.dagrun.DagRun

get_dagrun(*run_id, session=NEW_SESSION*)

get_dagruns_between(*start_date, end_date, session=NEW_SESSION*)

Return the list of dag runs between start_date (inclusive) and end_date (inclusive).

Parameters

- **start_date** – The starting logical date of the DagRun to find.
- **end_date** – The ending logical date of the DagRun to find.
- **session**

Returns

The list of DagRuns found.

get_latest_logical_date(session=NEW_SESSION)

Return the latest date for which at least one dag run exists.

get_task_instances_before(base_date, num, *, session=NEW_SESSION)

Get num task instances before (including) base_date.

The returned list may contain exactly num task instances corresponding to any DagRunType. It can have less if there are less than num scheduled DAG runs before base_date.

get_task_instances(start_date=None, end_date=None, state=None, session=NEW_SESSION)

set_task_instance_state(*, task_id, map_indexes=None, run_id=None, state, upstream=False, downstream=False, future=False, past=False, commit=True, session=NEW_SESSION)

Set the state of a TaskInstance and clear downstream tasks in failed or upstream_failed state.

Parameters

- **task_id (str)** – Task ID of the TaskInstance
- **map_indexes (collections.abc.Collection[int] / None)** – Only set TaskInstance if its map_index matches. If None (default), all mapped TaskInstances of the task are set.
- **run_id (str / None)** – The run_id of the TaskInstance
- **state (airflow.utils.state.TaskInstanceState)** – State to set the TaskInstance to
- **upstream (bool)** – Include all upstream tasks of the given task_id
- **downstream (bool)** – Include all downstream tasks of the given task_id
- **future (bool)** – Include all future TaskInstances of the given task_id
- **commit (bool)** – Commit changes
- **past (bool)** – Include all past TaskInstances of the given task_id

set_task_group_state(*, group_id, run_id=None, state, upstream=False, downstream=False, future=False, past=False, commit=True, session=NEW_SESSION)

Set TaskGroup to the given state and clear downstream tasks in failed or upstream_failed state.

Parameters

- **group_id (str)** – The group_id of the TaskGroup
- **run_id (str / None)** – The run_id of the TaskInstance
- **state (airflow.utils.state.TaskInstanceState)** – State to set the TaskInstance to
- **upstream (bool)** – Include all upstream tasks of the given task_id
- **downstream (bool)** – Include all downstream tasks of the given task_id
- **future (bool)** – Include all future TaskInstances of the given task_id
- **commit (bool)** – Commit changes

- **past** (*bool*) – Include all past TaskInstances of the given task_id
- **session** (*sqlalchemy.orm.session.Session*) – new session

```
clear(*, dry_run: airflow.typing_compat.Literal[True], task_ids: collections.abc.Collection[str | tuple[str, int]] | None = None, run_id: str, only_failed: bool = False, only_running: bool = False, confirm_prompt: bool = False, dag_run_state: airflow.utils.state.DagRunState = DagRunState.QUEUED, session: sqlalchemy.orm.session.Session = NEW_SESSION, dag_bag: airflow.models.dagbag.DagBag | None = None, exclude_task_ids: frozenset[str] | frozenset[tuple[str, int]] | None = frozenset(), exclude_run_ids: frozenset[str] | None = frozenset()) → list[airflow.models.taskinstance.TaskInstance]

clear(*, task_ids: collections.abc.Collection[str | tuple[str, int]] | None = None, run_id: str, only_failed: bool = False, only_running: bool = False, confirm_prompt: bool = False, dag_run_state: airflow.utils.state.DagRunState = DagRunState.QUEUED, dry_run: airflow.typing_compat.Literal[False] = False, session: sqlalchemy.orm.session.Session = NEW_SESSION, dag_bag: airflow.models.dagbag.DagBag | None = None, exclude_task_ids: frozenset[str] | frozenset[tuple[str, int]] | None = frozenset(), exclude_run_ids: frozenset[str] | None = frozenset()) → int

clear(*, dry_run: airflow.typing_compat.Literal[True], task_ids: collections.abc.Collection[str | tuple[str, int]] | None = None, start_date: datetime.datetime | None = None, end_date: datetime.datetime | None = None, only_failed: bool = False, only_running: bool = False, confirm_prompt: bool = False, dag_run_state: airflow.utils.state.DagRunState = DagRunState.QUEUED, session: sqlalchemy.orm.session.Session = NEW_SESSION, dag_bag: airflow.models.dagbag.DagBag | None = None, exclude_task_ids: frozenset[str] | frozenset[tuple[str, int]] | None = frozenset(), exclude_run_ids: frozenset[str] | None = frozenset()) → list[airflow.models.taskinstance.TaskInstance]

clear(*, task_ids: collections.abc.Collection[str | tuple[str, int]] | None = None, start_date: datetime.datetime | None = None, end_date: datetime.datetime | None = None, only_failed: bool = False, only_running: bool = False, confirm_prompt: bool = False, dag_run_state: airflow.utils.state.DagRunState = DagRunState.QUEUED, dry_run: airflow.typing_compat.Literal[False] = False, session: sqlalchemy.orm.session.Session = NEW_SESSION, dag_bag: airflow.models.dagbag.DagBag | None = None, exclude_task_ids: frozenset[str] | frozenset[tuple[str, int]] | None = frozenset(), exclude_run_ids: frozenset[str] | None = frozenset()) → int
```

Clear a set of task instances associated with the current dag for a specified date range.

Parameters

- **task_ids** – List of task ids or (task_id, map_index) tuples to clear
- **run_id** – The run_id for which the tasks should be cleared
- **start_date** – The minimum logical_date to clear
- **end_date** – The maximum logical_date to clear
- **only_failed** – Only clear failed tasks
- **only_running** – Only clear running tasks.
- **confirm_prompt** – Ask for confirmation
- **dag_run_state** – state to set DagRun to. If set to False, dagrun state will not be changed.
- **dry_run** – Find the tasks to clear but don't clear them.
- **session** – The sqlalchemy session to use
- **dag_bag** – The DagBag used to find the dags (Optional)

- **exclude_task_ids** – A set of task_id or (task_id, map_index) tuples that should not be cleared

- **exclude_run_ids** – A set of run_id or (run_id)

classmethod `clear_dags(dags, start_date=None, end_date=None, only_failed=False, only_running=False, confirm_prompt=False, dag_run_state=DagRunState.QUEUED, dry_run=False)`

cli()

Exposes a CLI specific to this DAG.

classmethod `bulk_write_to_db(bundle_name, bundle_version, dags, session=NEW_SESSION)`

Ensure the DagModel rows for the given dags are up-to-date in the dag table in the DB.

Parameters

dags (`collections.abc.Collection[airflow.serialization.serialized_objects.MaybeSerializedDAG]`) – the DAG objects to save to the DB

Returns

None

sync_to_db(session=NEW_SESSION)

Save attributes about this DAG to the DB.

Returns

None

static deactivate_unknown_dags(active_dag_ids, session=NEW_SESSION)

Given a list of known DAGs, deactivate any other DAGs that are marked as active in the ORM.

Parameters

active_dag_ids – list of DAG IDs that are active

Returns

None

static deactivate_stale_dags(expiration_date, session=NEW_SESSION)

Deactivate any DAGs that were last touched by the scheduler before the expiration date.

These DAGs were likely deleted.

Parameters

expiration_date – set inactive DAGs that were touched before this time

Returns

None

static get_num_task_instances(dag_id, run_id=None, task_ids=None, states=None, session=NEW_SESSION)

Return the number of task instances in the given DAG.

Parameters

- **session** – ORM session
- **dag_id** – ID of the DAG to get the task concurrency of
- **run_id** – ID of the DAG run to get the task concurrency of
- **task_ids** – A list of valid task IDs for the given DAG

- **states** – A list of states to filter by if supplied

Returns

The number of running tasks

Return type

int

get_task_assets(*inlets=True, outlets=True, of_type=Asset*)

classmethod from_sdk_dag(dag)

Create a new (Scheduler) DAG object from a TaskSDKDag.

class airflow.models.dag.DagTag

Bases: airflow.models.base.Base

A tag name per dag, to allow quick filtering in the DAG view.

__tablename__ = 'dag_tag'

name

dag_id

__table_args__

__repr__()

class airflow.models.dag.DagOwnerAttributes

Bases: airflow.models.base.Base

Table defining different owner attributes.

For example, a link for an owner that will be passed as a hyperlink to the “DAGs” view.

__tablename__ = 'dag_owner_attributes'

dag_id

owner

link

__repr__()

classmethod get_all(session)

class airflow.models.dag.DagModel(kwargs)**

Bases: airflow.models.base.Base

Table containing DAG properties.

__tablename__ = 'dag'

These items are stored in the database for state related information

dag_id

```
is_paused_at_creation = True
is_paused
is_stale
last_parsed_time
last_expired
fileloc
relative_fileloc
bundle_name
bundle_version
owners
description
timetable_summary
timetable_description
asset_expression
tags
dag_owner_links
max_active_tasks
max_active_runs
max_consecutive_failed_dag_runs
has_task_concurrency_limits
has_import_errors
next_dagrun
next_dagrun_data_interval_start
next_dagrun_data_interval_end
next_dagrun_create_after
__table_args__
schedule_asset_references
schedule_asset_alias_references
schedule_asset_name_references
schedule_asset_uri_references
schedule_assets
```

```
task_outlet_asset_references
NUM_DAGS_PER_DAGRUN_QUERY
dag_versions
__repr__()
property next_dagrun_data_interval: airflow.timetables.base.DataInterval | None

property timezone
static get_dagmodel(dag_id, session=NEW_SESSION)

classmethod get_current(dag_id, session=NEW_SESSION)

get_last_dagrun(session=NEW_SESSION, include_manually_triggered=False)
get_is_paused(*, session=None)
    Provide interface compatibility to 'DAG'.

get_is_active(*, session=None)
    Provide interface compatibility to 'DAG'.

static get_paused_dag_ids(dag_ids, session=NEW_SESSION)
    Given a list of dag_ids, get a set of Paused Dag Ids.

    Parameters
    • dag_ids (list[str]) – List of Dag ids
    • session (sqlalchemy.orm.session.Session) – ORM Session

    Returns
    Paused Dag_ids

    Return type
    set[str]

property safe_dag_id

set_is_paused(is_paused, session=NEW_SESSION)
    Pause/Un-pause a DAG.

    Parameters
    • is_paused (bool) – Is the DAG paused
    • session – session

dag_display_name()
```

classmethod deactivate_deleted_dags(bundle_name, rel_filelocs, session=NEW_SESSION)

Set `is_active=False` on the DAGs for which the DAG files have been removed.

Parameters

- `bundle_name (str)` – bundle for filelocs
- `rel_filelocs (list [str])` – relative filelocs for bundle
- `session (sqlalchemy.orm.session.Session)` – ORM Session

classmethod dags_needing_dagruns(session)

Return (and lock) a list of Dag objects that are due to create a new DagRun.

This will return a resultset of rows that is row-level-locked with a “SELECT … FOR UPDATE” query, you should ensure that any scheduling decisions are made in a single transaction – as soon as the transaction is committed it will be unlocked.

calculate_dagrun_date_fields(dag, last_automated_dag_run)

Calculate `next_dagrun` and `next_dagrun_create_after`.

Parameters

- `dag (DAG)` – The DAG object
- `last_automated_dag_run (None / airflow.timetables.base.DataInterval)` – DataInterval (or datetime) of most recent run of this dag, or none if not yet scheduled.

get_asset_triggered_next_run_info(*, session=NEW_SESSION)**airflow.models.dagbag****Classes**

| | |
|--|--|
| <code>FileLoadStat</code> | Information about single file. |
| <code>DagBag</code> | A dagbag is a collection of dags, parsed out of a folder tree and has high level configuration settings. |
| <code>DagPriorityParsingRequest</code> | Model to store the dag parsing requests that will be prioritized when parsing files. |

Functions**generate_md5_hash(context)****Module Contents****class airflow.models.dagbag.FileLoadStat**

Bases: `NamedTuple`

Information about single file.

Parameters

- **file** – Loaded file.
- **duration** – Time spent on process file.
- **dag_num** – Total number of DAGs loaded in this file.
- **task_num** – Total number of Tasks loaded in this file.
- **dags** – DAGs names loaded in this file.
- **warning_num** – Total number of warnings captured from processing this file.

file: str

duration: datetime.timedelta

dag_num: int

task_num: int

dags: str

warning_num: int

```
class airflow.models.dagbag.DagBag(dag_folder=None, include_examples=NOTSET, safe_mode=NOTSET,
                                    read_dags_from_db=False, load_op_links=True, collect_dags=True,
                                    known_pools=None, bundle_path=None)
```

Bases: airflow.utils.log.logging_mixin.LoggingMixin

A dagbag is a collection of dags, parsed out of a folder tree and has high level configuration settings.

Some possible setting are database to use as a backend and what executor to use to fire off tasks. This makes it easier to run distinct environments for say production and development, tests, or for different teams or security profiles. What would have been system level settings are now dagbag level so that one system can run multiple, independent settings sets.

Parameters

- **dag_folder** (str / pathlib.Path / None) – the folder to scan to find DAGs
- **include_examples** (bool / airflow.utils.types.ArgNotSet) – whether to include the examples that ship with airflow or not
- **safe_mode** (bool / airflow.utils.types.ArgNotSet) – when False, scans all python modules for dags. When True uses heuristics (files containing DAG and airflow strings) to filter python modules to scan for dags.
- **read_dags_from_db** (bool) – Read DAGs from DB if True is passed. If False DAGs are read from python files.
- **load_op_links** (bool) – Should the extra operator link be loaded via plugins when deserializing the DAG? This flag is set to False in Scheduler so that Extra Operator links are not loaded to not run User code in Scheduler.
- **collect_dags** (bool) – when True, collects dags during class initialization.
- **known_pools** (set[str] / None) – If not none, then generate warnings if a Task attempts to use an unknown pool.

bundle_path: pathlib.Path | None = None

dag_folder

```

dags: dict[str, airflow.models.dag.DAG]
file_last_changed: dict[str, datetime.datetime]
import_errors: dict[str, str]
captured_warnings: dict[str, tuple[str, Ellipsis]]
has_logged = False
read_dags_from_db = False
dags_last_fetched: dict[str, datetime.datetime]
dags_hash: dict[str, str]
known_pools = None
dagbag_import_error_tracebacks = True
dagbag_import_error_traceback_depth
load_op_links = True
size()

```

Returns

the amount of dags contained in this dagbag

Return type

int

property dag_ids: list[str]

Get DAG ids.

Returns

a list of DAG IDs in this bag

Return type

list[str]

get_dag(dag_id, session=None)

Get the DAG out of the dictionary, and refreshes it if expired.

Parameters

dag_id – DAG ID

process_file(filepath, only_if_updated=True, safe_mode=True)

Given a path to a python module or zip file, import the module and look for dag objects within.

property dag_warnings: set[airflow.models.dagwarning.DagWarning]

Get the set of DagWarnings for the bagged dags.

bag_dag(dag)

Add the DAG into the bag.

Raises

AirflowDagCycleException if a cycle is detected.

Raises

AirflowDagDuplicatedIdException if this dag already exists in the bag.

```
collect_dags(dag_folder=None, only_if_updated=True, include_examples=conf.getboolean('core',  
    'LOAD_EXAMPLES'), safe_mode=conf.getboolean('core',  
    'DAG_DISCOVERY_SAFE_MODE'))
```

Look for python modules in a given path, import them, and add them to the dagbag collection.

Note that if a `.airflowignore` file is found while processing the directory, it will behave much like a `.gitignore`, ignoring files that match any of the patterns specified in the file.

Note: The patterns in `.airflowignore` are interpreted as either un-anchored regexes or gitignore-like glob expressions, depending on the `DAG_IGNORE_FILE_SYNTAX` configuration parameter.

collect_dags_from_db()

Collect DAGs from database.

dagbag_report()

Print a report around DagBag loading stats.

sync_to_db(bundle_name, bundle_version, session=NEW_SESSION)

Save attributes about list of DAG to the DB.

`airflow.models.dagbag.generate_md5_hash(context)`

class airflow.models.dagbag.DagPriorityParsingRequest(bundle_name, relative_fileloc)

Bases: `airflow.models.base.Base`

Model to store the dag parsing requests that will be prioritized when parsing files.

```
__tablename__ = 'dag_priority_parsing_request'  
  
id  
  
bundle_name  
  
relative_fileloc  
  
__repr__()
```

airflow.models.param

Re exporting the new param module from Task SDK for backward compatibility.

Properties of a `DagRun` can also be referenced in things like *Templates*.

airflow.models.dagrun

Attributes

CreatedTasks`RUN_ID_REGEX`**Classes**

| | |
|-----------------------------------|--|
| <code>TISchedulingDecision</code> | Type of return for DagRun.task_instance_scheduling_decisions. |
| <code>DagRun</code> | Invocation instance of a DAG. |
| <code>DagRunNote</code> | For storage of arbitrary notes concerning the dagrun instance. |

Module Contents`airflow.models.dagrun.CreatedTasks``airflow.models.dagrun.RUN_ID_REGEX = '^(:manual|scheduled|asset_triggered)__(?:\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}\\+00:00)$'``class airflow.models.dagrun.TISchedulingDecision`Bases: `NamedTuple`Type of return for `DagRun.task_instance_scheduling_decisions`.`tis: list[airflow.models.taskinstance.TaskInstance]``schedulable_tis: list[airflow.models.taskinstance.TaskInstance]``changed_tis: bool``unfinished_tis: list[airflow.models.taskinstance.TaskInstance]``finished_tis: list[airflow.models.taskinstance.TaskInstance]``class airflow.models.dagrun.DagRun(dag_id=None, run_id=None, *, queued_at=NOTSET, logical_date=None, run_after=None, start_date=None, conf=None, state=None, run_type=None, creating_job_id=None, data_interval=None, triggered_by=None, backfill_id=None, bundle_version=None)`Bases: `airflow.models.base.Base, airflow.utils.log.logging_mixin.LoggingMixin`

Invocation instance of a DAG.

A DAG run can be created by the scheduler (i.e. scheduled runs), or by an external trigger (i.e. manual runs).

`active_spans``__tablename__ = 'dag_run'``id``dag_id`

```
queued_at
logical_date
start_date
end_date
run_id
creating_job_id
run_type
triggered_by
conf
data_interval_start
data_interval_end
run_after
last_scheduling_decision
log_template_id
updated_at
clear_number
backfill_id

The backfill this DagRun is currently associated with.

It's possible this could change if e.g. the dag run is cleared to be rerun, or perhaps re-backfilled.

bundle_version
scheduled_by_job_id
context_carrier
span_status
dag: airflow.models.dag.DAG | None
__table_args__
task_instances
task_instances_histories
dag_model
dag_run_note
backfill
backfill_max_active_runs
```

`max_active_runs`

`note`

`DEFAULT_DAGRUNS_TO_EXAMINE`

`__repr__()`

`validate_run_id(key, run_id)`

`property dag_versions: list[airflow.models.dag_version.DagVersion]`

Return the DAG versions associated with the TIs of this DagRun.

`property version_number: int | None`

Return the DAG version number associated with the latest TI of this DagRun.

`check_version_id_exists_in_dr(dag_version_id, session=NEW_SESSION)`

`property stats_tags: dict[str, str]`

`classmethod set_active_spans(active_spans)`

`get_state()`

`set_state(state)`

Change the state of the DagRan.

Changes to attributes are implemented in accordance with the following table (rows represent old states, columns represent new states):

Table 68: State transition matrix

| QUEUED | | RUNNING | | SUCCESS | FAILED |
|---------|--|--|--|-------------------------------|-------------------------------|
| None | queued_at = zone.utcnow() | time- if empty: start_date = timezone.utcnow() end_date = None | end_date = time-zone.utcnow() = time-zone.utcnow() | end_date = time-zone.utcnow() | end_date = time-zone.utcnow() |
| QUEUED | queued_at = zone.utcnow() | time- if empty: start_date = timezone.utcnow() end_date = None | end_date = time-zone.utcnow() = time-zone.utcnow() | end_date = time-zone.utcnow() | end_date = time-zone.utcnow() |
| RUNNING | queued_at = time-zone.utcnow() start_date = None end_date = None | | | end_date = time-zone.utcnow() | end_date = time-zone.utcnow() |
| SUCCESS | queued_at = time-zone.utcnow() start_date = None end_date = None | start_date = time-zone.utcnow() end_date = None | | | |
| FAILED | queued_at = time-zone.utcnow() start_date = None end_date = None | start_date = time-zone.utcnow() end_date = None | | | |

property state

refresh_from_db(session=NEW_SESSION)

Reload the current dagrun from the database.

Parameters

session (`sqlalchemy.orm.Session`) – database session

classmethod find(dag_id=None, run_id=None, logical_date=None, state=None, no_backfills=False, run_type=None, session=NEW_SESSION, logical_start_date=None, logical_end_date=None)

Return a set of dag runs for the given search criteria.

Parameters

- **dag_id** (`str` / `list[str]` / `None`) – the dag_id or list of dag_id to find dag runs for
- **run_id** (`collections.abc.Iterable[str]` / `None`) – defines the run id for this dag run
- **run_type** (`airflow.utils.types.DagRunType` / `None`) – type of DagRun
- **logical_date** (`datetime.datetime` / `collections.abc.Iterable[datetime.datetime]` / `None`) – the logical date
- **state** (`airflow.utils.state.DagRunState` / `None`) – the state of the dag run
- **no_backfills** (`bool`) – return no backfills (True), return all (False). Defaults to False
- **session** (`sqlalchemy.orm.Session`) – database session
- **logical_start_date** (`datetime.datetime` / `None`) – dag run that was executed from this date
- **logical_end_date** (`datetime.datetime` / `None`) – dag run that was executed until this date

classmethod find_duplicate(dag_id, run_id, *, session=NEW_SESSION)

Return an existing run for the DAG with a specific run_id.

None is returned if no such DAG run is found.

Parameters

- **dag_id** (`str`) – the dag_id to find duplicates for
- **run_id** (`str`) – defines the run id for this dag run
- **session** (`sqlalchemy.orm.Session`) – database session

static generate_run_id(*, run_type, logical_date=None, run_after)

Generate Run ID based on Run Type, run_after and logical Date.

Parameters

- **run_type** (`airflow.utils.types.DagRunType`) – type of DagRun
- **logical_date** (`datetime.datetime` / `None`) – the logical date
- **run_after** (`datetime.datetime`) – the date before which dag run won't start.

```
static fetch_task_instances(dag_id=None, run_id=None, task_ids=None, state=None,
session=NEW_SESSION)
```

Return the task instances for this dag run.

```
get_task_instances(state=None, session=NEW_SESSION)
```

Return the task instances for this dag run.

Redirect to DagRun.fetch_task_instances method. Keep this method because it is widely used across the code.

```
get_task_instance(task_id, session=NEW_SESSION, *, map_index=-1)
```

Return the task instance specified by task_id for this dag run.

Parameters

- **task_id** (*str*) – the task id
- **session** (*sqlalchemy.orm.Session*) – Sqlalchemy ORM Session

```
static fetch_task_instance(dag_id, dag_run_id, task_id, session=NEW_SESSION, map_index=-1)
```

Return the task instance specified by task_id for this dag run.

Parameters

- **dag_id** (*str*) – the DAG id
- **dag_run_id** (*str*) – the DAG run id
- **task_id** (*str*) – the task id
- **session** (*sqlalchemy.orm.Session*) – Sqlalchemy ORM Session

```
get_dag()
```

Return the Dag associated with this DagRun.

Returns

DAG

Return type

airflow.models.dag.DAG

```
static get_previous_dagrun(dag_run, state=None, session=NEW_SESSION)
```

Return the previous DagRun, if there is one.

Parameters

- **dag_run** (*DagRun*) – the dag run
- **session** (*sqlalchemy.orm.Session*) – SQLAlchemy ORM Session
- **state** (*airflow.utils.state.DagRunState* / *None*) – the dag run state

```
static get_previous_scheduled_dagrun(dag_run_id, session=NEW_SESSION)
```

Return the previous SCHEDULED DagRun, if there is one.

Parameters

- **dag_run_id** (*int*) – the DAG run ID
- **session** (*sqlalchemy.orm.Session*) – SQLAlchemy ORM Session

`set_dagrun_span_attrs(span)`

`start_dr_spans_if_needed(tis)`

`end_dr_span_if_needed()`

`update_state(session=NEW_SESSION, execute_callbacks=True)`

Determine the overall state of the DagRun based on the state of its TaskInstances.

Parameters

- `session (sqlalchemy.orm.Session)` – Sqlalchemy ORM Session
- `execute_callbacks (bool)` – Should dag callbacks (success/failure, SLA etc.) be invoked directly (default: true) or recorded as a pending request in the `returned_callback` property

Returns

Tuple containing tis that can be scheduled in the current loop & `returned_callback` that needs to be executed

Return type

`tuple[list[airflow.models.taskinstance.TaskInstance], airflow.callbacks.callback_requests.DagCallbackRequest | None]`

`task_instance_scheduling_decisions(session=NEW_SESSION)`

`notify_dagrun_state_changed(msg='')`

`handle_dag_callback(dag, success=True, reason='success')`

Only needed for `dag.test` where `execute_callbacks=True` is passed to `update_state`.

`verify_integrity(*, session=NEW_SESSION, dag_version_id=None)`

Verify the DagRun by checking for removed tasks or tasks that are not in the database yet.

It will set state to removed or add the task if required.

Parameters

- `dag_version_id (sqlalchemy_utils.UUIDType / None)` – The DAG version ID
- `session (sqlalchemy.orm.Session)` – Sqlalchemy ORM Session

`classmethod get_latest_runs(session=NEW_SESSION)`

Return the latest DagRun for each DAG.

`schedule_tis(schedulable_tis, session=NEW_SESSION, max_tis_per_query=None)`

Set the given task instances in to the scheduled state.

Each element of `schedulable_tis` should have its `task` attribute already set.

Any EmptyOperator without callbacks or outlets is instead set straight to the success state.

All the TIs should belong to this DagRun, but this code is in the hot-path, this is not checked – it is the caller's responsibility to call this function only with TIs from a single dag run.

```
get_log_template(*, session=NEW_SESSION)

class airflow.models.dagrun.DagRunNote(content, user_id=None)
    Bases: airflow.models.base.Base

    For storage of arbitrary notes concerning the dagrun instance.

    __tablename__ = 'dag_run_note'

    user_id
    dag_run_id
    content
    created_at
    updated_at
    dag_run
    __table_args__
    __repr__()


```

Operators

The base classes `BaseOperator` and `BaseSensorOperator` are public and may be extended to make new operators.

Subclasses of `BaseOperator` which are published in Apache Airflow are public in *behavior* but not in *structure*. That is to say, the Operator's parameters and behavior is governed by semver but the methods are subject to change at any time.

Task Instances

Task instances are the individual runs of a single task in a DAG (in a DAG Run). They are available in the context passed to the `execute` method of the operators via the `TaskInstance` class.

airflow.models.taskinstance

Attributes

| |
|------------------------------------|
| <code>TR</code> |
| <code>log</code> |
| <code>PAST_DEPENDS_MET</code> |
| <code>TaskInstanceStateType</code> |

Classes

| | |
|---------------------------------|--|
| <code>TaskInstance</code> | Task instances store the state of a task instance. |
| <code>SimpleTaskInstance</code> | Simplified Task Instance. |
| <code>TaskInstanceNote</code> | For storage of arbitrary notes concerning the task instance. |

Functions

| | |
|---|---|
| <code>set_current_context(context)</code> | Set the current execution context to the provided context object. |
| <code>uuid7()</code> | Generate a new UUID7 string. |

Module Contents

`airflow.models.taskinstance.TR`

`airflow.models.taskinstance.log`

`airflow.models.taskinstance.PAST_DEPENDS_MET = 'past_depends_met'`

`airflow.models.taskinstance.set_current_context(context)`

Set the current execution context to the provided context object.

This method should be called once per Task execution, before calling operator.execute.

`airflow.models.taskinstance.uuid7()`

Generate a new UUID7 string.

`class airflow.models.taskinstance.TaskInstance(task, run_id=None, state=None, map_index=-1, dag_version_id=None)`

Bases: `airflow.models.base.Base, airflow.utils.log.logging_mixin.LoggingMixin`

Task instances store the state of a task instance.

This table is the authority and single source of truth around what tasks have run and the state they are in.

The SQLAlchemy model doesn't have a SQLAlchemy foreign key to the task or dag model deliberately to have more control over transactions.

Database transactions on this table should insure double triggers and any confusion around what task instances are or aren't ready to run even while multiple schedulers may be firing task instances.

A value of -1 in map_index represents any of: a TI without mapped tasks; a TI with mapped tasks that has yet to be expanded (state=pending); a TI with mapped tasks that expanded to an empty list (state=skipped).

`__tablename__ = 'task_instance'`

`id`

`task_id`

`dag_id`
`run_id`
`map_index`
`start_date`
`end_date`
`duration`
`state`
`try_number`
`max_tries`
`hostname`
`unixname`
`pool`
`pool_slots`
`queue`
`priority_weight`
`operator`
`custom_operator_name`
`queued_dttm`
`scheduled_dttm`
`queued_by_job_id`
`last_heartbeat_at`
`pid`
`executor`
`executor_config`
`updated_at`
`context_carrier`
`span_status`
`external_executor_id`
`trigger_id`
`trigger_timeout`
`next_method`

```
next_kwargs
dag_version_id
dag_version
__table_args__
dag_model: airflow.models.dag.DagModel
trigger
triggerer_job
dag_run
rendered_task_instance_fields
run_after
logical_date
task_instance_note
note
task: airflow.sdkdefinitions._internal.abstractoperator.Operator | None = None
test_mode: bool = False
is_trigger_log_context: bool = False
run_as_user: str | None = None
__hash__()
property stats_tags: dict[str, str]
    Returns task instance tags.

init_on_load()
    Initialize the attributes that aren't stored in the DB.

property operator_name: str | None
    @property: use a more friendly display name for the operator, if set.

task_display_name()

rendered_map_index()

classmethod from_runtime_ti(runtime_ti)
```

`to_runtime_tis(context_from_server)`

`command_as_list(mark_success=False, ignore_all_deps=False, ignore_task_deps=False,
ignore_depends_on_past=False, wait_for_past_depends_before_skipping=False,
ignore_ti_state=False, local=False, raw=False, pool=None, cfg_path=None)`

Return a command that can be executed anywhere where airflow is installed.

This command is part of the message sent to executors by the orchestrator.

`static generate_command(dag_id, task_id, run_id, mark_success=False, ignore_all_deps=False,
ignore_depends_on_past=False,
wait_for_past_depends_before_skipping=False, ignore_task_deps=False,
ignore_ti_state=False, local=False, file_path=None, raw=False, pool=None,
cfg_path=None, map_index=-1)`

Generate the shell command required to execute this task instance.

Parameters

- `dag_id (str)` – DAG ID
- `task_id (str)` – Task ID
- `run_id (str)` – The run_id of this task’s DagRun
- `mark_success (bool)` – Whether to mark the task as successful
- `ignore_all_deps (bool)` – Ignore all ignorable dependencies. Overrides the other ignore_* parameters.
- `ignore_depends_on_past (bool)` – Ignore depends_on_past parameter of DAGs (e.g. for Backfills)
- `wait_for_past_depends_before_skipping (bool)` – Wait for past depends before marking the ti as skipped
- `ignore_task_deps (bool)` – Ignore task-specific dependencies such as depends_on_past and trigger rule
- `ignore_ti_state (bool)` – Ignore the task instance’s previous failure/success
- `local (bool)` – Whether to run the task locally
- `file_path (pathlib.PurePath / str / None)` – path to the file containing the DAG definition
- `raw (bool)` – raw mode (needs more details)
- `pool (str / None)` – the Airflow pool that the task should run in
- `cfg_path (str / None)` – the Path to the configuration file

Returns

shell command that can be used to run the task instance

Return type

list[str]

`property log_url: str`

Log URL for TaskInstance.

property mark_success_url: str

URL to mark TI success.

error(session=NEW_SESSION)

Force the task instance's state to FAILED in the database.

Parameters

session(sqlalchemy.orm.session.Session) – SQLAlchemy ORM Session

classmethod get_task_instance(dag_id, run_id, task_id, map_index, lock_for_update=False, session=NEW_SESSION)**refresh_from_db(session=NEW_SESSION, lock_for_update=False, keep_local_changes=False)**

Refresh the task instance from the database based on the primary key.

Parameters

- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy ORM Session
- **lock_for_update** (bool) – if True, indicates that the database should lock the TaskInstance (issuing a FOR UPDATE clause) until the session is committed.
- **keep_local_changes** (bool) – Force all attributes to the values from the database if False (the default), or if True don't overwrite locally set attributes

refresh_from_task(task, pool_override=None)

Copy common attributes from the given task.

Parameters

- **task** (airflow.sdkdefinitions._internal.abstractoperator.Operator) – The task object to copy from
- **pool_override** (str / None) – Use the pool_override instead of task's pool

clear_xcom_data(session=NEW_SESSION)**property key: airflow.models.taskinstancekey.TaskInstanceKey**

Returns a tuple that identifies the task instance uniquely.

set_state(state, session=NEW_SESSION)

Set TaskInstance state.

Parameters

- **state** (str / None) – State to set for the TI
- **session** (sqlalchemy.orm.session.Session) – SQLAlchemy ORM Session

Returns

Was the state changed

Return type

bool

property is_premature: bool

Returns whether a task is in UP_FOR_RETRY state and its retry interval has elapsed.

prepare_db_for_next_try(session)

Update the metadata with all the records needed to put this TI in queued for the next try.

are_dependents_done(session=NEW_SESSION)

Check whether the immediate dependents of this task instance have succeeded or have been skipped.

This is meant to be used by wait_for_downstream.

This is useful when you do not want to start processing the next schedule of a task until the dependents are done. For instance, if the task DROPS and recreates a table.

Parameters

- **session** (*sqlalchemy.orm.session.Session*) – SQLAlchemy ORM Session

get_previous_dagrun(state=None, session=None)

Return the DagRun that ran before this task instance's DagRun.

Parameters

- **state** (*airflow.utils.state.DagRunState* / *None*) – If passed, it only take into account instances of a specific state.
- **session** (*sqlalchemy.orm.session.Session* / *None*) – SQLAlchemy ORM Session.

get_previous_ti(state=None, session=NEW_SESSION)

Return the task instance for the task that ran before this task instance.

Parameters

- **session** (*sqlalchemy.orm.session.Session*) – SQLAlchemy ORM Session
- **state** (*airflow.utils.state.DagRunState* / *None*) – If passed, it only take into account instances of a specific state.

are_dependencies_met(dep_context=None, session=NEW_SESSION, verbose=False)

Are all conditions met for this task instance to be run given the context for the dependencies.

(e.g. a task instance being force run from the UI will ignore some dependencies).

Parameters

- **dep_context** (*airflow.ti_deps.dep_context.DepContext* / *None*) – The execution context that determines the dependencies that should be evaluated.
- **session** (*sqlalchemy.orm.session.Session*) – database session
- **verbose** (*bool*) – whether log details on failed dependencies on info or debug log level

get_failed_dep_statuses(dep_context=None, session=NEW_SESSION)

Get failed Dependencies.

__repr__()

next_retry_datetime()

Get datetime of the next retry if the task instance fails.

For exponential backoff, retry_delay is used as base and will be converted to seconds.

ready_for_retry()

Check on whether the task instance is in the right state and timeframe to be retried.

get_dagrun(session=NEW_SESSION)

Return the DagRun for this TaskInstance.

Parameters

session (*sqlalchemy.orm.session.Session*) – SQLAlchemy ORM Session

Returns

DagRun

Return type

airflow.models.dagrun.DagRun

check_and_change_state_before_execution(*verbose=True, ignore_all_deps=False, ignore_depends_on_past=False, wait_for_past_depends_before_skipping=False, ignore_task_deps=False, ignore_ti_state=False, mark_success=False, test_mode=False, pool=None, external_executor_id=None, session=NEW_SESSION)***emit_state_change_metric(*new_state*)**

Send a time metric representing how much time a given state transition took.

The previous state and metric name is deduced from the state the task was put in.

Parameters

new_state (*airflow.utils.state.TaskInstanceState*) – The state that has just been set for this task. We do not use *self.state*, because sometimes the state is updated directly in the DB and not in the local TaskInstance object. Supported states: QUEUED and RUNNING

clear_next_method_args()

Ensure we unset next_method and next_kwargs to ensure that any retries don't reuse them.

static register_asset_changes_in_db(*ti, task_outlets, outlet_events, session=NEW_SESSION*)**update_rtif(*rendered_fields, session=NEW_SESSION*)****update_heartbeat()****defer_task(*exception, session=NEW_SESSION*)**

Mark the task as deferred and sets up the trigger that is needed to resume it when TaskDeferred is raised.

```
run(verbose=True, ignore_all_deps=False, ignore_depends_on_past=False,
     wait_for_past_depends_before_skipping=False, ignore_task_deps=False, ignore_ti_state=False,
     mark_success=False, test_mode=False, pool=None, session=NEW_SESSION, raise_on_defer=False)

Run TaskInstance.
```

dry_run()

Only Renders Templates for the TI.

```
classmethod fetch_handle_failure_context(ti, error, test_mode=None, context=None,
                                         force_fail=False, *, session, fail_fast=False)
```

Fetch the context needed to handle a failure.

Parameters

- **ti** (`TaskInstance`) – `TaskInstance`
- **error** (`None` / `str` / `BaseException`) – if specified, log the specific exception if thrown
- **test_mode** (`bool` / `None`) – doesn't record success or failure in the DB if True
- **context** (`airflow.utils.context.Context` / `None`) – `Jinja2` context
- **force_fail** (`bool`) – if True, task does not retry
- **session** (`sqlalchemy.orm.session.Session`) – `SQLAlchemy` ORM Session
- **fail_fast** (`bool`) – if True, fail all downstream tasks

```
static save_to_db(ti, session=NEW_SESSION)
```

```
handle_failure(error, test_mode=None, context=None, force_fail=False, session=NEW_SESSION)
```

Handle Failure for a task instance.

Parameters

- **error** (`None` / `str` / `BaseException`) – if specified, log the specific exception if thrown
- **session** (`sqlalchemy.orm.session.Session`) – `SQLAlchemy` ORM Session
- **test_mode** (`bool` / `None`) – doesn't record success or failure in the DB if True
- **context** (`airflow.utils.context.Context` / `None`) – `Jinja2` context
- **force_fail** (`bool`) – if True, task does not retry

is_eligible_to_retry()

Is task instance is eligible for retry.

```
get_template_context(session=None, ignore_param_exceptions=True)
```

Return TI Context.

Parameters

- **session** (`sqlalchemy.orm.session.Session` / `None`) – `SQLAlchemy` ORM Session

- **ignore_param_exceptions** (*bool*) – flag to suppress value exceptions while initializing the ParamsDict

get_rendered_template_fields(*session=NEW_SESSION*)

Update task with rendered template fields for presentation in UI.

If task has already run, will fetch from DB; otherwise will render.

overwrite_params_with_dag_run_conf(*params, dag_run*)

Overwrite Task Params with DagRun.conf.

render_templates(*context=None, jinja_env=None*)

Render templates in the operator fields.

If the task was originally mapped, this may replace `self.task` with the unmapped, fully rendered BaseOperator. The original `self.task` before replacement is returned.

get_email_subject_content(*exception, task=None*)

Get the email subject content for exceptions.

Parameters

- **exception** (*BaseException*) – the exception sent in the email
- **task** (*airflow.models.baseoperator.BaseOperator / None*)

email_alert(*exception, task*)

Send alert email with exception information.

Parameters

- **exception** – the exception
- **task** (*airflow.models.baseoperator.BaseOperator*) – task related to the exception

set_duration()

Set task instance duration.

xcom_push(*key, value, session=NEW_SESSION*)

Make an XCom available for tasks to pull.

Parameters

- **key** (*str*) – Key to store the value under.
- **value** (*Any*) – Value to store. Only be JSON-serializable may be used otherwise.

get_num_running_task_instances(*session, same_dagrun=False*)

Return Number of running TIs from the DB.

static filter_for_tis(*tis*)

Return SQLAlchemy filter to query selected task instances.

get_relevant_upstream_map_indexes(upstream, ti_count, *, session)

Infer the map indexes of an upstream “relevant” to this ti.

The bulk of the logic mainly exists to solve the problem described by the following example, where ‘val’ must resolve to different values, depending on where the reference is being used:

```
@task
def this_task(v): # This is self.task.
    return v * 2


@task_group
def tg1(inp):
    val = upstream(inp) # This is the upstream task.
    this_task(val) # When inp is 1, val here should resolve to 2.
    return val


# This val is the same object returned by tg1.
val = tg1.expand(inp=[1, 2, 3])


@task_group
def tg2(inp):
    another_task(inp, val) # val here should resolve to [2, 4, 6].
    tg2.expand(inp=["a", "b"])
```

The surrounding mapped task groups of `upstream` and `self.task` are inspected to find a common “ancestor”. If such an ancestor is found, we need to return specific map indexes to pull a partial value from upstream XCom.

Parameters

- **upstream** (`airflow.sdkdefinitions._internal.abstractoperator.Operator`) – The referenced upstream task.
- **ti_count** (`int / None`) – The total count of task instance this task was expanded by the scheduler, i.e. `expanded_ti_count` in the template context.

Returns

Specific map index or map indexes to pull, or `None` if we want to “whole” return value (i.e. no mapped task groups involved).

Return type

`int | range | None`

classmethod duration_expression_update(end_date, query, bind)

Return a SQL expression for calculating the duration of this TI, based on the start and end date columns.

static validate_inlet_outlet_assets_activeness(inlets, outlets, session)**get_first_reschedule_date(context)**

Get the first reschedule date for the task instance.

`airflow.models.taskinstance.TaskInstanceStateType`

```
class airflow.models.taskinstance.SimpleTaskInstance(dag_id, task_id, run_id, queued_dttm,
                                                    start_date, end_date, try_number, map_index,
                                                    state, executor, executor_config, pool, queue,
                                                    key, run_as_user=None,
                                                    priority_weight=None,
                                                    parent_context_carrier=None,
                                                    context_carrier=None, span_status=None)
```

Simplified Task Instance.

Used to send data between processes via Queues.

```
dag_id
task_id
run_id
map_index
queued_dttm
start_date
end_date
try_number
state
executor
executor_config
run_as_user = None
pool
priority_weight = None
queue
key
parent_context_carrier = None
context_carrier = None
span_status = None
__repr__()

__eq__(other)
```

```

classmethod from_ti(ti)

class airflow.models.taskinstance.TaskInstanceNote(content, user_id=None)
Bases: airflow.models.base.Base

For storage of arbitrary notes concerning the task instance.

__tablename__ = 'task_instance_note'

ti_id
user_id
content
created_at
updated_at
task_instance
__table_args__
__repr__()

```

Task Instance Keys

Task instance keys are unique identifiers of task instances in a DAG (in a DAG Run). A key is a tuple that consists of `dag_id`, `task_id`, `run_id`, `try_number`, and `map_index`. The key of a task instance can be retrieved via `key()`.

airflow.models.taskinstancekey

Classes

| | |
|------------------------------|-------------------------------------|
| <code>TaskInstanceKey</code> | Key used to identify task instance. |
|------------------------------|-------------------------------------|

Module Contents

```

class airflow.models.taskinstancekey.TaskInstanceKey

```

Bases: NamedTuple

Key used to identify task instance.

`dag_id: str`

`task_id: str`

`run_id: str`

`try_number: int = 1`

`map_index: int = -1`

`property primary: tuple[str, str, str, int]`

Return task instance primary key part of the key.

with_try_number(try_number)

Return TaskInstanceKey with provided `try_number`.

property key: TaskInstanceKey

For API-compatibility with `TaskInstance`.

Returns self

classmethod from_dict(dictionary)

Create `TaskInstanceKey` from dictionary.

Hooks

Hooks are interfaces to external platforms and databases, implementing a common interface when possible and acting as building blocks for operators. All hooks are derived from `BaseHook`.

Airflow has a set of Hooks that are considered public. You are free to extend their functionality by extending them:

airflow.hooks

Submodules

airflow.hooks.base

Base class for all hooks.

Attributes

`log`

Classes

`BaseHook`

Abstract base class for hooks.

`DiscoverableHook`

Interface that providers *can* implement to be discovered by `ProvidersManager`.

Module Contents

airflow.hooks.base.log

class airflow.hooks.base.BaseHook(logger_name=None)

Bases: `airflow.utils.log.logging_mixin.LoggingMixin`

Abstract base class for hooks.

Hooks are meant as an interface to interact with external systems. `MySqlHook`, `HiveHook`, `PigHook` return object that can handle the connection and interaction to specific instances of these systems, and expose consistent methods to interact with them.

Parameters

logger_name (*str* / *None*) – Name of the logger used by the Hook to emit logs. If set to *None* (default), the logger name will fall back to *airflow.task.hooks.{class.__module__}.{class.__name__}* (e.g. *DbApiHook* will have *airflow.task.hooks.airflow.providers.common.sql.hooks.sql.DbApiHook* as logger).

classmethod get_connection(*conn_id*)

Get connection, given connection id.

Parameters

conn_id (*str*) – connection id

Returns

connection

Return type

airflow.models.connection.Connection

classmethod get_hook(*conn_id*, *hook_params=None*)

Return default hook for this connection id.

Parameters

- **conn_id** (*str*) – connection id
- **hook_params** (*dict* / *None*) – hook parameters

Returns

default hook for this connection

Return type

BaseHook

abstractmethod get_conn()

Return connection for the hook.

classmethod get_connection_form_widgets()**classmethod get_ui_field_behaviour()****class airflow.hooks.base.DiscoverableHook**

Bases: *Protocol*

Interface that providers *can* implement to be discovered by *ProvidersManager*.

It is not used by any of the Hooks, but simply methods and class fields described here are implemented by those Hooks. Each method is optional – only implement the ones you need.

The *conn_name_attr*, *default_conn_name*, *conn_type* should be implemented by those Hooks that want to be automatically mapped from the *connection_type* -> Hook when *get_hook* method is called with *connection_type*.

Additionally *hook_name* should be set when you want the hook to have a custom name in the UI selection Name. If not specified, *conn_name* will be used.

The “*get_ui_field_behaviour*” and “*get_connection_form_widgets*” are optional - override them if you want to customize the Connection Form screen. You can add extra widgets to parse your extra fields via the *get_connection_form_widgets* method as well as hide or relabel the fields or pre-fill them with placeholders via *get_ui_field_behaviour* method.

Note that the “get_ui_field_behaviour” and “get_connection_form_widgets” need to be set by each class in the class hierarchy in order to apply widget customizations.

For example, even if you want to use the fields from your parent class, you must explicitly have a method on *your* class:

```
@classmethod  
def get_ui_field_behaviour(cls):  
    return super().get_ui_field_behaviour()
```

You also need to add the Hook class name to list ‘hook_class_names’ in provider.yaml in case you build an internal provider or to return it in dictionary returned by provider_info entrypoint in the package you prepare.

You can see some examples in airflow/providers/jdbc/hooks/jdbc.py.

```
conn_name_attr: str  
  
default_conn_name: str  
  
conn_type: str  
  
hook_name: str  
  
static get_connection_form_widgets()
```

Return dictionary of widgets to be added for the hook to handle extra values.

If you have class hierarchy, usually the widgets needed by your class are already added by the base class, so there is no need to implement this method. It might actually result in warning in the logs if you try to add widgets that have already been added by the base class.

Note that values of Dict should be of wtforms.Field type. It’s not added here for the efficiency of imports.

```
static get_ui_field_behaviour()
```

Attributes of the UI field.

Returns dictionary describing customizations to implement in javascript handling the connection form. Should be compliant with airflow/customized_form_field_behaviours.schema.json’

If you change conn_type in a derived class, you should also implement this method and return field customizations appropriate to your Hook. This is because the child hook will have usually different conn_type and the customizations are per connection type.

See also

[ComputeSSH](#) as an example

Public Airflow utilities

When writing or extending Hooks and Operators, DAG authors and developers can use the following classes:

- The *Connection*, which provides access to external service credentials and configuration.
- The *Variable*, which provides access to Airflow configuration variables.
- The *XCom* which are used to access to inter-task communication data.

You can read more about the public Airflow utilities in *Managing Connections, Variables, XComs*
 Reference for classes used for the utilities are here:

airflow.models.connection

Attributes

| |
|----------------------------------|
| <code>log</code> |
| <code>RE_SANITIZE_CONN_ID</code> |
| <code>CONN_ID_MAX_LEN</code> |

Classes

| | |
|-------------------------|---|
| <code>Connection</code> | Placeholder to store information about different database instances connection information. |
|-------------------------|---|

Functions

| | |
|--|---|
| <code>sanitize_conn_id(conn_id[, max_length])</code> | Sanitizes the connection id and allows only specific characters to be within. |
|--|---|

Module Contents

`airflow.models.connection.log`

`airflow.models.connection.RE_SANITIZE_CONN_ID`

`airflow.models.connection.CONN_ID_MAX_LEN: int = 250`

`airflow.models.connection.sanitize_conn_id(conn_id, max_length=CONN_ID_MAX_LEN)`

Sanitizes the connection id and allows only specific characters to be within.

Namely, it allows alphanumeric characters plus the symbols #,!,-_,.,:,/ and () from 1 and up to 250 consecutive matches. If desired, the max length can be adjusted by setting `max_length`.

You can try to play with the regex here: <https://regex101.com/r/69033B/1>

The character selection is such that it prevents the injection of javascript or executable bits to avoid any awkward behaviour in the front-end.

Parameters

- `conn_id` (`str` / `None`) – The connection id to sanitize.
- `max_length` – The max length of the connection ID, by default it is 250.

Returns

the sanitized string, `None` otherwise.

Return type

`str | None`

```
class airflow.models.connection.Connection(conn_id=None, conn_type=None, description=None,
                                         host=None, login=None, password=None, schema=None,
                                         port=None, extra=None, uri=None)
```

Bases: airflow.models.base.Base, airflow.utils.log.logging_mixin.LoggingMixin

Placeholder to store information about different database instances connection information.

The idea here is that scripts use references to database instances (conn_id) instead of hard coding hostname, logins and passwords when using operators or hooks.

See also

For more information on how to use this class, see: *Managing Connections*

Parameters

- **conn_id** (*str* / *None*) – The connection ID.
- **conn_type** (*str* / *None*) – The connection type.
- **description** (*str* / *None*) – The connection description.
- **host** (*str* / *None*) – The host.
- **login** (*str* / *None*) – The login.
- **password** (*str* / *None*) – The password.
- **schema** (*str* / *None*) – The schema.
- **port** (*int* / *None*) – The port number.
- **extra** (*str* / *dict* / *None*) – Extra metadata. Non-standard data such as private/SSH keys can be saved here. JSON encoded object.
- **uri** (*str* / *None*) – URI address describing connection parameters.

```
EXTRA_KEY = '__extra__'

__tablename__ = 'connection'

id
conn_id
conn_type
description
host
schema
login
port
is_encrypted
is_extra_encrypted
```

on_db_load()**get_uri()**

Return the connection URI in Airflow format.

The Airflow URI format examples: <https://airflow.apache.org/docs/apache-airflow/stable/howto/connection.html#uri-format-example>

Note that the URI returned by this method is **not** SQLAlchemy-compatible, if you need a SQLAlchemy-compatible URI, use the `sqlalchemy_url`

get_password()

Return encrypted password.

set_password(*value*)

Encrypt password and set in object attribute.

property password

Password. The value is decrypted/encrypted when reading/setting the value.

get_extra()

Return encrypted extra-data.

set_extra(*value*)

Encrypt extra-data and save in object attribute to object.

property extra

Extra data. The value is decrypted/encrypted when reading/setting the value.

rotate_fernet_key()

Encrypts data with a new key. See: *Fernet*.

get_hook(*, hook_params=None)

Return hook based on conn_type.

__repr__()**test_connection()**

Calls out get_hook method and executes test_connection method on that.

get_extra_dejson(*nested=False*)

Deserialize extra property to JSON.

Parameters

nested (*bool*) – Determines whether nested structures are also deserialized into JSON (default False).

property extra_dejson: dict

Returns the extra property by deserializing json.

```
classmethod get_connection_from_secrets(conn_id)
    Get connection by conn_id.
```

If *MetastoreBackend* is getting used in the execution context, use Task SDK API.

Parameters

conn_id (str) – connection id

Returns

connection

Return type

Connection

```
classmethod from_json(value, conn_id=None)
```

as_json()

Convert Connection to JSON-string object.

airflow.models.variable

Attributes

log

Classes

Variable

A generic way to store and retrieve arbitrary content or settings as a simple key/value store.

Module Contents

airflow.models.variable.log

```
class airflow.models.variable.Variable(key=None, val=None, description=None)
```

Bases: `airflow.models.base.Base`, `airflow.utils.log.logging_mixin.LoggingMixin`

A generic way to store and retrieve arbitrary content or settings as a simple key/value store.

__tablename__ = 'variable'

id

key

description

is_encrypted

property val

Get Airflow Variable from Metadata DB and decode it using the Fernet Key.

on_db_load()**__repr__()****get_val()**

Get Airflow Variable from Metadata DB and decode it using the Fernet Key.

set_val(*value*)

Encode the specified value with Fernet Key and store it in Variables Table.

classmethod setdefault(*key*, *default*, *description=None*, *deserialize_json=False*)

Return the current value for a key or store the default value and return it.

Works the same as the Python builtin dict object.

Parameters

- **key** – Dict key for this Variable
- **default** – Default value to set and return if the variable isn't already in the DB
- **description** – Default value to set Description of the Variable
- **deserialize_json** – Store this as a JSON encoded value in the DB and un-encode it when retrieving a value
- **session** – Session

Returns

Mixed

classmethod get(*key*, *default_var=__NO_DEFAULT_SENTINEL*, *deserialize_json=False*)

Get a value for an Airflow Variable Key.

Parameters

- **key (str)** – Variable Key
- **default_var (Any)** – Default value of the Variable if the Variable doesn't exist
- **deserialize_json (bool)** – Deserialize the value to a Python dict

static set(*key*, *value*, *description=None*, *serialize_json=False*, *session=None*)

Set a value for an Airflow Variable with a given Key.

This operation overwrites an existing variable.

Parameters

- **key (str)** – Variable Key
- **value (Any)** – Value to set for the Variable
- **description (str / None)** – Description of the Variable
- **serialize_json (bool)** – Serialize the value to a JSON string
- **session (sqlalchemy.orm.Session / None)** – optional session, use if provided or create a new one

static update(*key*, *value*, *serialize_json=False*, *session=None*)

Update a given Airflow Variable with the Provided value.

Parameters

- **key (str)** – Variable Key

- **value** (*Any*) – Value to set for the Variable
- **serialize_json** (*bool*) – Serialize the value to a JSON string
- **session** (*sqlalchemy.orm.Session* / *None*) – optional session, use if provided or create a new one

static delete(key, session=None)

Delete an Airflow Variable for a given key.

Parameters

- **key** (*str*) – Variable Keys
- **session** (*sqlalchemy.orm.Session* / *None*) – optional session, use if provided or create a new one

rotate_fernet_key()

Rotate Fernet Key.

static check_for_write_conflict(key)

Log a warning if a variable exists outside the metastore.

If we try to write a variable to the metastore while the same key exists in an environment variable or custom secrets backend, then subsequent reads will not read the set value.

Parameters

key (*str*) – Variable Key

static get_variable_from_secrets(key)

Get Airflow Variable by iterating over all Secret Backends.

Parameters

key (*str*) – Variable Key

Returns

Variable Value

Return type

str | *None*

airflow.models.xcom

Attributes

log

Classes

XComModel

XCom model class. Contains table and some utilities.

Functions

```
__getattr__(name)
```

Module Contents

`airflow.models.xcom.log`

`class airflow.models.xcom.XComModel`

Bases: `airflow.models.base.TaskInstanceDependencies`

XCom model class. Contains table and some utilities.

`__tablename__ = 'xcom'`

`dag_run_id`

`task_id`

`map_index`

`key`

`dag_id`

`run_id`

`value`

`timestamp`

`__table_args__`

`dag_run`

`logical_date`

`classmethod clear(*, dag_id, task_id, run_id, map_index=None, session=NEW_SESSION)`

Clear all XCom data from the database for the given task instance.

Note

This **will not** purge any data from a custom XCom backend.

Parameters

- `dag_id (str)` – ID of DAG to clear the XCom for.
- `task_id (str)` – ID of task to clear the XCom for.
- `run_id (str)` – ID of DAG run to clear the XCom for.
- `map_index (int / None)` – If given, only clear XCom from this particular mapped task. The default `None` clears *all* XComs from the task.
- `session (sqlalchemy.orm.Session)` – Database session. If not given, a new session will be created for this function.

```
classmethod set(key, value, *, dag_id, task_id, run_id, map_index=-1, session=NEW_SESSION)
```

Store an XCom value.

Parameters

- **key** (*str*) – Key to store the XCom.
- **value** (*Any*) – XCom value to store.
- **dag_id** (*str*) – DAG ID.
- **task_id** (*str*) – Task ID.
- **run_id** (*str*) – DAG run ID for the task.
- **map_index** (*int*) – Optional map index to assign XCom for a mapped task. The default is -1 (set for a non-mapped task).
- **session** (*sqlalchemy.orm.Session*) – Database session. If not given, a new session will be created for this function.

```
classmethod get_many(*, run_id, key=None, task_ids=None, dag_ids=None, map_indexes=None, include_prior_dates=False, limit=None, session=NEW_SESSION)
```

Composes a query to get one or more XCom entries.

This function returns an SQLAlchemy query of full XCom objects. If you just want one stored value, use `get_one()` instead.

Parameters

- **run_id** (*str*) – DAG run ID for the task.
- **key** (*str* / *None*) – A key for the XComs. If provided, only XComs with matching keys will be returned. Pass *None* (default) to remove the filter.
- **task_ids** (*str* / *collections.abc.Iterable[str]* / *None*) – Only XComs from task with matching IDs will be pulled. Pass *None* (default) to remove the filter.
- **dag_ids** (*str* / *collections.abc.Iterable[str]* / *None*) – Only pulls XComs from specified DAGs. Pass *None* (default) to remove the filter.
- **map_indexes** (*int* / *collections.abc.Iterable[int]* / *None*) – Only XComs from matching map indexes will be pulled. Pass *None* (default) to remove the filter.
- **include_prior_dates** (*bool*) – If *False* (default), only XComs from the specified DAG run are returned. If *True*, all matching XComs are returned regardless of the run it belongs to.
- **session** (*sqlalchemy.orm.Session*) – Database session. If not given, a new session will be created for this function.
- **limit** (*int* / *None*) – Limiting returning XComs

```
static serialize_value(value, *, key=None, task_id=None, dag_id=None, run_id=None, map_index=None)
```

Serialize XCom value to JSON str.

```
static deserialize_value(result)
```

Deserialize XCom value from a database result.

If deserialization fails, the raw value is returned, which must still be a valid Python JSON-compatible type (e.g., `dict`, `list`, `str`, `int`, `float`, or `bool`).

XCom values are stored as JSON in the database, and SQLAlchemy automatically handles serialization (`json.dumps`) and deserialization (`json.loads`). However, we use a custom encoder for serialization (`serialize_value`) and deserialization to handle special cases, such as encoding tuples via the Airflow Serialization module. These must be decoded using `XComDecoder` to restore original types.

Some XCom values, such as those set via the Task Execution API, bypass `serialize_value` and are stored directly in JSON format. Since these values are already deserialized by SQLAlchemy, they are returned as-is.

Example: Handling a tuple:

```
original_value = (1, 2, 3)
serialized_value = XComModel.serialize_value(original_value)
print(serialized_value)
# '{"__classname__": "builtins.tuple", "__version__": 1, "__data__": [1, 2, 3]}
```

This serialized value is stored in the database. When deserialized, the value is restored to the original tuple.

Parameters

`result` – The XCom database row or object containing a `value` attribute.

Returns

The deserialized Python object.

Return type

Any

`airflow.models.xcom.__getattr__(name)`

Public Exceptions

When writing the custom Operators and Hooks, you can handle and raise public Exceptions that Airflow exposes:

airflow.exceptions

Exceptions used by Airflow.

Exceptions

| | |
|---|--|
| <code>AirflowException</code> | Base class for all Airflow's errors. |
| <code>AirflowBadRequest</code> | Raise when the application or server cannot handle the request. |
| <code>AirflowNotFoundException</code> | Raise when the requested object/resource is not available in the system. |
| <code>AirflowConfigException</code> | Raise when there is configuration problem. |
| <code>AirflowSensorTimeout</code> | Raise when there is a timeout on sensor polling. |
| <code>AirflowRescheduleException</code> | Raise when the task should be re-scheduled at a later time. |
| <code>InvalidStatsNameException</code> | Raise when name of the stats is invalid. |
| <code>AirflowTaskTimeout</code> | Raise when the task execution times-out. |
| <code>AirflowTaskTerminated</code> | Raise when the task execution is terminated. |
| <code>AirflowWebServerTimeout</code> | Raise when the web server times out. |
| <code>AirflowSkipException</code> | Raise when the task should be skipped. |
| <code>AirflowFailException</code> | Raise when the task should be failed without retrying. |

continues on next page

Table 83 – continued from previous page

| | |
|---|---|
| <i>AirflowInactiveAssetInInletOrOutletException</i> | Raise when the task is executed with inactive assets in its inlet or outlet. |
| <i>AirflowOptionalProviderFeatureException</i> | Raise by providers when imports are missing for optional provider features. |
| <i>XComNotFound</i> | Raise when an XCom reference is being resolved against a non-existent XCom. |
| <i>XComForMappingNotPushed</i> | Raise when a mapped downstream's dependency fails to push XCom for task mapping. |
| <i>UnmappableXComTypePushed</i> | Raise when an unmappable type is pushed as a mapped downstream's dependency. |
| <i>UnmappableXComLengthPushed</i> | Raise when the pushed value is too large to map as a downstream's dependency. |
| <i>AirflowDagCycleException</i> | Raise when there is a cycle in DAG definition. |
| <i>AirflowDuplicatedIdException</i> | Raise when a DAG's ID is already used by another DAG. |
| <i>AirflowClusterPolicyViolation</i> | Raise when there is a violation of a Cluster Policy in DAG definition. |
| <i>AirflowClusterPolicySkipDag</i> | Raise when skipping dag is needed in Cluster Policy. |
| <i>AirflowClusterPolicyError</i> | Raise for a Cluster Policy other than AirflowClusterPolicyViolation or AirflowClusterPolicySkipDag. |
| <i>AirflowTimetableInvalid</i> | Raise when a DAG has an invalid timetable. |
| <i>DagNotFound</i> | Raise when a DAG is not available in the system. |
| <i>DagCodeNotFound</i> | Raise when a DAG code is not available in the system. |
| <i>DagRunNotFound</i> | Raise when a DAG Run is not available in the system. |
| <i>DagRunAlreadyExists</i> | Raise when creating a DAG run for DAG which already has DAG run entry. |
| <i>DagFileExists</i> | Raise when a DAG ID is still in DagBag i.e., DAG file is in DAG folder. |
| <i>FailFastDagInvalidTriggerRule</i> | Raise when a dag has 'fail_fast' enabled yet has a non-default trigger rule. |
| <i>DuplicateTaskIdFound</i> | Raise when a Task with duplicate task_id is defined in the same DAG. |
| <i>TaskAlreadyInTaskGroup</i> | Raise when a Task cannot be added to a TaskGroup since it already belongs to another TaskGroup. |
| <i>SerializationError</i> | A problem occurred when trying to serialize something. |
| <i>ParamValidationError</i> | Raise when DAG params is invalid. |
| <i>TaskNotFound</i> | Raise when a Task is not available in the system. |
| <i>TaskInstanceNotFound</i> | Raise when a task instance is not available in the system. |
| <i>PoolNotFound</i> | Raise when a Pool is not available in the system. |
| <i>AirflowFileParseException</i> | Raises when connection or variable file can not be parsed. |
| <i>ConnectionNotUnique</i> | Raise when multiple values are found for the same connection ID. |
| <i>DownstreamTasksSkipped</i> | Signal by an operator to skip its downstream tasks. |
| <i>DagRunTriggerException</i> | Signal by an operator to trigger a specific Dag Run of a dag. |
| <i>TaskDeferred</i> | Signal an operator moving to deferred state. |
| <i>TaskDeferralError</i> | Raised when a task failed during deferral for some reason. |
| <i>TaskDeferralTimeout</i> | Raise when there is a timeout on the deferral. |
| <i>PodMutationHookException</i> | Raised when exception happens during Pod Mutation Hook execution. |

continues on next page

Table 83 – continued from previous page

| | |
|--|--|
| <i>PodReconciliationError</i> | Raised when an error is encountered while trying to merge pod configs. |
| <i>RemovedInAirflow4Warning</i> | Issued for usage of deprecated features that will be removed in Airflow4. |
| <i>AirflowProviderDeprecationWarning</i> | Issued for usage of deprecated features of Airflow provider. |
| <i>DeserializingResultError</i> | Raised when an error is encountered while a pickling library deserializes a pickle file. |
| <i>UnknownExecutorException</i> | Raised when an attempt is made to load an executor which is not configured. |

Classes

| | |
|------------------------|---|
| <i>FileSyntaxError</i> | Information about a single error in a file. |
|------------------------|---|

Module Contents

exception airflow.exceptions.AirflowException

Bases: *Exception*

Base class for all Airflow's errors.

Each custom exception should be derived from this class.

status_code

serialize()

exception airflow.exceptions.AirflowBadRequest

Bases: *AirflowException*

Raise when the application or server cannot handle the request.

status_code

exception airflow.exceptions.AirflowNotFoundException

Bases: *AirflowException*

Raise when the requested object/resource is not available in the system.

status_code

exception airflow.exceptions.AirflowConfigException

Bases: *AirflowException*

Raise when there is configuration problem.

exception airflow.exceptions.AirflowSensorTimeout

Bases: *AirflowException*

Raise when there is a timeout on sensor polling.

exception airflow.exceptions.AirflowRescheduleException(*reschedule_date*)

Bases: *AirflowException*

Raise when the task should be re-scheduled at a later time.

Parameters

reschedule_date – The date when the task should be rescheduled

reschedule_date

serialize()

exception airflow.exceptions.InvalidStatsNameException
Bases: *AirflowException*
Raise when name of the stats is invalid.

exception airflow.exceptions.AirflowTaskTimeout
Bases: *BaseException*
Raise when the task execution times-out.

exception airflow.exceptions.AirflowTaskTerminated
Bases: *BaseException*
Raise when the task execution is terminated.

exception airflow.exceptions.AirflowWebServerTimeout
Bases: *AirflowException*
Raise when the web server times out.

exception airflow.exceptions.AirflowSkipException
Bases: *AirflowException*
Raise when the task should be skipped.

exception airflow.exceptions.AirflowFailException
Bases: *AirflowException*
Raise when the task should be failed without retrying.

exception airflow.exceptions.AirflowInactiveAssetInInletOrOutletException(*inactive_asset_keys*)
Bases: *_AirflowExecuteWithInactiveAssetException*
Raise when the task is executed with inactive assets in its inlet or outlet.

main_message = 'Task has the following inactive assets in its inlets or outlets'

exception airflow.exceptions.AirflowOptionalProviderFeatureException
Bases: *AirflowException*
Raise by providers when imports are missing for optional provider features.

exception airflow.exceptions.XComNotFound(*dag_id*, *task_id*, *key*)
Bases: *AirflowException*
Raise when an XCom reference is being resolved against a non-existent XCom.

dag_id

task_id

key

__str__()

Return str(self).

serialize()**exception airflow.exceptions.XComForMappingNotPushed**

Bases: *AirflowException*

Raise when a mapped downstream's dependency fails to push XCom for task mapping.

__str__()

Return str(self).

exception airflow.exceptions.UnmappableXComTypePushed(value, *values)

Bases: *AirflowException*

Raise when an unmappable type is pushed as a mapped downstream's dependency.

__str__()

Return str(self).

exception airflow.exceptions.UnmappableXComLengthPushed(value, max_length)

Bases: *AirflowException*

Raise when the pushed value is too large to map as a downstream's dependency.

value**max_length****__str__()**

Return str(self).

exception airflow.exceptions.AirflowDagCycleException

Bases: *AirflowException*

Raise when there is a cycle in DAG definition.

exception airflow.exceptions.AirflowDuplicatedIdException(dag_id, incoming, existing)

Bases: *AirflowException*

Raise when a DAG's ID is already used by another DAG.

dag_id**incoming****existing**

__str__()

Return str(self).

exception airflow.exceptions.AirflowClusterPolicyViolation

Bases: *AirflowException*

Raise when there is a violation of a Cluster Policy in DAG definition.

exception airflow.exceptions.AirflowClusterPolicySkipDag

Bases: *AirflowException*

Raise when skipping dag is needed in Cluster Policy.

exception airflow.exceptions.AirflowClusterPolicyError

Bases: *AirflowException*

Raise for a Cluster Policy other than AirflowClusterPolicyViolation or AirflowClusterPolicySkipDag.

exception airflow.exceptions.AirflowTimetableInvalid

Bases: *AirflowException*

Raise when a DAG has an invalid timetable.

exception airflow.exceptions.DagNotFound

Bases: *AirflowNotFoundException*

Raise when a DAG is not available in the system.

exception airflow.exceptions.DagCodeNotFound

Bases: *AirflowNotFoundException*

Raise when a DAG code is not available in the system.

exception airflow.exceptions.DagRunNotFound

Bases: *AirflowNotFoundException*

Raise when a DAG Run is not available in the system.

exception airflow.exceptions.DagRunAlreadyExists(dag_run)

Bases: *AirflowBadRequest*

Raise when creating a DAG run for DAG which already has DAG run entry.

dag_run

serialize()

exception airflow.exceptions.DagFileExists(*args, **kwargs)

Bases: *AirflowBadRequest*

Raise when a DAG ID is still in DagBag i.e., DAG file is in DAG folder.

exception airflow.exceptions.FailFastDagInvalidTriggerRule

Bases: *AirflowException*

Raise when a dag has ‘fail_fast’ enabled yet has a non-default trigger rule.

__str__()

Return str(self).

exception airflow.exceptions.DuplicateTaskIdFound

Bases: *AirflowException*

Raise when a Task with duplicate task_id is defined in the same DAG.

exception airflow.exceptions.TaskAlreadyInTaskGroup(task_id, existing_group_id, new_group_id)

Bases: *AirflowException*

Raise when a Task cannot be added to a TaskGroup since it already belongs to another TaskGroup.

task_id**existing_group_id****new_group_id****__str__()**

Return str(self).

exception airflow.exceptions.SerializationError

Bases: *AirflowException*

A problem occurred when trying to serialize something.

exception airflow.exceptions.ParamValidationError

Bases: *AirflowException*

Raise when DAG params is invalid.

exception airflow.exceptions.TaskNotFound

Bases: *AirflowNotFoundException*

Raise when a Task is not available in the system.

exception airflow.exceptions.TaskInstanceNotFound

Bases: *AirflowNotFoundException*

Raise when a task instance is not available in the system.

exception airflow.exceptions.PoolNotFound

Bases: *AirflowNotFoundException*

Raise when a Pool is not available in the system.

class airflow.exceptions.FileSyntaxError

Bases: *NamedTuple*

Information about a single error in a file.

line_no: int | None**message: str**

`__str__()`

exception `airflow.exceptions.AirflowFileParseException(msg, file_path, parse_errors)`

Bases: *AirflowException*

Raises when connection or variable file can not be parsed.

Parameters

- **msg** (*str*) – The human-readable description of the exception
- **file_path** (*str*) – A processed file that contains errors
- **parse_errors** (*list[FileSyntaxError]*) – File syntax errors

msg

file_path

parse_errors

`__str__()`

Return str(self).

exception `airflow.exceptions.ConnectionNotUnique`

Bases: *AirflowException*

Raise when multiple values are found for the same connection ID.

exception `airflow.exceptions.DownstreamTasksSkipped(*, tasks)`

Bases: *AirflowException*

Signal by an operator to skip its downstream tasks.

Special exception raised to signal that the operator it was raised from wishes to skip downstream tasks. This is used in the ShortCircuitOperator.

Parameters

tasks (*collections.abc.Sequence[str | tuple[str, int]]*) – List of task_ids to skip or a list of tuples with task_id and map_index to skip.

tasks

exception `airflow.exceptions.DagRunTriggerException(*, trigger_dag_id, dag_run_id, conf, logical_date, reset_dag_run, skip_when_already_exists, wait_for_completion, allowed_states, failed_states, poke_interval, deferrable)`

Bases: *AirflowException*

Signal by an operator to trigger a specific Dag Run of a dag.

Special exception raised to signal that the operator it was raised from wishes to trigger a specific Dag Run of a dag. This is used in the TriggerDagRunOperator.

trigger_dag_id

dag_run_id

conf

```
logical_date
reset_dag_run
skip_when_already_exists
wait_for_completion
allowed_states
failed_states
poke_interval
deferrable
```

exception airflow.exceptions.TaskDeferred(*, trigger, method_name, kwargs=None, timeout=None)

Bases: BaseException

Signal an operator moving to deferred state.

Special exception raised to signal that the operator it was raised from wishes to defer until a trigger fires. Triggers can send execution back to task or end the task instance directly. If the trigger should end the task instance itself, `method_name` does not matter, and can be None; otherwise, provide the name of the method that should be used when resuming execution in the task.

```
trigger
method_name
kwargs = None
timeout: datetime.timedelta | None
serialize()
__repr__()
    Return repr(self).
```

exception airflow.exceptions.TaskDeferralError

Bases: AirflowException

Raised when a task failed during deferral for some reason.

exception airflow.exceptions.TaskDeferralTimeout

Bases: AirflowException

Raise when there is a timeout on the deferral.

exception airflow.exceptions.PodMutationHookException

Bases: AirflowException

Raised when exception happens during Pod Mutation Hook execution.

exception airflow.exceptions.PodReconciliationError

Bases: AirflowException

Raised when an error is encountered while trying to merge pod configs.

exception airflow.exceptions.RemovedInAirflow4Warning

Bases: `DeprecationWarning`

Issued for usage of deprecated features that will be removed in Airflow4.

deprecated_since: str | None = None

Indicates the airflow version that started raising this deprecation warning

exception airflow.exceptions.AirflowProviderDeprecationWarning

Bases: `DeprecationWarning`

Issued for usage of deprecated features of Airflow provider.

deprecated_provider_since: str | None = None

Indicates the provider version that started raising this deprecation warning

exception airflow.exceptions.DeserializingResultError

Bases: `ValueError`

Raised when an error is encountered while a pickling library deserializes a pickle file.

__str__()

Return `str(self)`.

exception airflow.exceptions.UnknownExecutorException

Bases: `ValueError`

Raised when an attempt is made to load an executor which is not configured.

Public Utility classes

airflow.utils.state

Classes

| | |
|-----------------------------------|---|
| <code>JobState</code> | All possible states that a Job can be in. |
| <code>TerminalTISState</code> | States that a Task Instance can be in that indicate it has reached a terminal state. |
| <code>IntermediateTISState</code> | States that a Task Instance can be in that indicate it is not yet in a terminal or running state. |
| <code>TaskInstanceState</code> | All possible states that a Task Instance can be in. |
| <code>DagRunState</code> | All possible states that a DagRun can be in. |
| <code>State</code> | Static class with task instance state constants and color methods to avoid hard-coding. |

Module Contents

class airflow.utils.state.JobState

Bases: `str, enum.Enum`

All possible states that a Job can be in.

`RUNNING = 'running'`

`SUCCESS = 'success'`

`RESTARTING = 'restarting'`

```
FAILED = 'failed'
```

```
__str__()
```

Return str(self).

```
class airflow.utils.state.TerminalTISState
```

Bases: str, enum.Enum

States that a Task Instance can be in that indicate it has reached a terminal state.

```
SUCCESS = 'success'
```

```
FAILED = 'failed'
```

```
SKIPPED = 'skipped'
```

```
REMOVED = 'removed'
```

```
__str__()
```

Return str(self).

```
class airflow.utils.state.IntermediateTISState
```

Bases: str, enum.Enum

States that a Task Instance can be in that indicate it is not yet in a terminal or running state.

```
SCHEDULED = 'scheduled'
```

```
QUEUED = 'queued'
```

```
RESTARTING = 'restarting'
```

```
UP_FOR_RETRY = 'up_for_retry'
```

```
UP_FOR_RESCHEDULE = 'up_for_reschedule'
```

```
UPSTREAM_FAILED = 'upstream_failed'
```

```
DEFERRED = 'deferred'
```

```
__str__()
```

Return str(self).

```
class airflow.utils.state.TaskInstanceState
```

Bases: str, enum.Enum

All possible states that a Task Instance can be in.

Note that None is also allowed, so always use this in a type hint with Optional.

```
REMOVED
```

```
SCHEDULED
```

```
QUEUED
```

```
RUNNING = 'running'
```

```
SUCCESS
```

```
RESTARTING
```

```
FAILED
```

```
UP_FOR_RETRY
```

```
UP_FOR_RESCHEDULE
```

```
UPSTREAM_FAILED
```

```
SKIPPED
```

```
DEFERRED
```

```
__str__()
```

Return str(self).

```
class airflow.utils.state.DagRunState
```

Bases: str, enum.Enum

All possible states that a DagRun can be in.

These are “shared” with TaskInstanceState in some parts of the code, so please ensure that their values always match the ones with the same name in TaskInstanceState.

```
QUEUED = 'queued'
```

```
RUNNING = 'running'
```

```
SUCCESS = 'success'
```

```
FAILED = 'failed'
```

```
__str__()
```

Return str(self).

```
class airflow.utils.state.State
```

Static class with task instance state constants and color methods to avoid hard-coding.

```
SUCCESS
```

```
RUNNING
```

```
FAILED
```

```
NONE = None
```

```
REMOVED
```

```
SCHEDULED
```

```
QUEUED
```

```

RESTARTING
UP_FOR_RETRY
UP_FOR_RSCHEDULE
UPSTREAM_FAILED
SKIPPED
DEFERRED

finished_dr_states: frozenset[DagRunState]
unfinished_dr_states: frozenset[DagRunState]
task_states: tuple[TaskInstanceState | None, Ellipsis]
dag_states: tuple[DagRunState, Ellipsis]
state_color: dict[TaskInstanceState | None, str]

@classmethod color(state)
    Return color for a state.

@classmethod color_fg(state)
    Black&white colors for a state.

finished: frozenset[TaskInstanceState]
    A list of states indicating a task has reached a terminal state (i.e. it has “finished”) and needs no further action.

    Note that the attempt could have resulted in failure or have been interrupted; or perhaps never run at all (skip, or upstream_failed) in any case, it is no longer running.

unfinished: frozenset[TaskInstanceState | None]
    A list of states indicating that a task either has not completed a run or has not even started.

failed_states: frozenset[TaskInstanceState]
    A list of states indicating that a task or dag is a failed state.

success_states: frozenset[TaskInstanceState]
    A list of states indicating that a task or dag is a success state.

adoptable_states
    A list of states indicating that a task can be adopted or reset by a scheduler job if it was queued by another scheduler job that is not running anymore.

```

3.11.3 Using Public Interface to extend Airflow capabilities

Airflow uses Plugin mechanism to extend Airflow platform capabilities. They allow to extend Airflow UI but also they are the way to expose the below customizations (Triggers, Timetables, Listeners, etc.). Providers can also implement plugin endpoints and customize Airflow UI and the customizations.

You can read more about plugins in *Plugins*. You can read how to extend Airflow UI in *Customize view of Apache from Airflow web UI*. Note that there are some simple customizations of the UI that do not require plugins - you can read more about them in *Customizing the UI*.

Here are the ways how Plugins can be used to extend Airflow:

Triggers

Airflow uses Triggers to implement `asyncio` compatible Deferrable Operators. All Triggers derive from `BaseTrigger`.

Airflow has a set of Triggers that are considered public. You are free to extend their functionality by extending them:

`airflow.triggers`

`Submodules`

`airflow.triggers.base`

`Attributes`

`log`

`DiscriminatedTriggerEvent`

Classes

| | |
|-------------------------------|---|
| <code>StartTriggerArgs</code> | Arguments required for start task execution from triggerer. |
| <code>BaseTrigger</code> | Base class for all triggers. |
| <code>BaseEventTrigger</code> | Base class for triggers used to schedule DAGs based on external events. |
| <code>TriggerEvent</code> | Something that a trigger can fire when its conditions are met. |
| <code>TaskSuccessEvent</code> | Yield this event in order to end the task successfully. |
| <code>TaskFailedEvent</code> | Yield this event in order to end the task with failure. |
| <code>TaskSkippedEvent</code> | Yield this event in order to end the task with status 'skipped'. |

Functions

`trigger_event_discriminator(v)`

Module Contents

`airflow.triggers.base.log`

`class airflow.triggers.base.StartTriggerArgs`

Arguments required for start task execution from triggerer.

`trigger_cls: str`

`next_method: str`

`trigger_kwargs: dict[str, Any] | None = None`

```
next_kwargs: dict[str, Any] | None = None
timeout: datetime.timedelta | None = None

class airflow.triggers.base.BaseTrigger(**kwargs)
Bases: abc.ABC, airflow.utils.log.logging_mixin.LoggingMixin
```

Base class for all triggers.

A trigger has two contexts it can exist in:

- Inside an Operator, when it's passed to TaskDeferred
- Actively running in a trigger worker

We use the same class for both situations, and rely on all Trigger classes to be able to return the arguments (possible to encode with Airflow-JSON) that will let them be re-instantiated elsewhere.

task_instance = None

trigger_id = None

abstractmethod serialize()

Return the information needed to reconstruct this Trigger.

Returns

Tuple of (class path, keyword arguments needed to re-instantiate).

Return type

tuple[str, dict[str, Any]]

abstractmethod run()

Async

Run the trigger in an asynchronous context.

The trigger should yield an Event whenever it wants to fire off an event, and return None if it is finished. Single-event triggers should thus yield and then immediately return.

If it yields, it is likely that it will be resumed very quickly, but it may not be (e.g. if the workload is being moved to another triggerer process, or a multi-event trigger was being used for a single-event task defer).

In either case, Trigger classes should assume they will be persisted, and then rely on cleanup() being called when they are no longer needed.

async cleanup()

Cleanup the trigger.

Called when the trigger is no longer needed, and it's being removed from the active triggerer process.

This method follows the async/await pattern to allow to run the cleanup in triggerer main event loop. Exceptions raised by the cleanup method are ignored, so if you would like to be able to debug them and be notified that cleanup method failed, you should wrap your code with try/except block and handle it appropriately (in async-compatible way).

static repr(classpath, kwargs)

`__repr__()`

```
class airflow.triggers.base.BaseEventTrigger(**kwargs)
```

Bases: `BaseTrigger`

Base class for triggers used to schedule DAGs based on external events.

`BaseEventTrigger` is a subclass of `BaseTrigger` designed to identify triggers compatible with event-driven scheduling.

`static hash(classpath, kwargs)`

Return the hash of the trigger classpath and kwargs. This is used to uniquely identify a trigger.

We do not want to have this logic in `BaseTrigger` because, when used to defer tasks, 2 triggers can have the same classpath and kwargs. This is not true for event driven scheduling.

```
class airflow.triggers.base.TriggerEvent(payload, **kwargs)
```

Bases: `pydantic.BaseModel`

Something that a trigger can fire when its conditions are met.

Events must have a uniquely identifying value that would be the same wherever the trigger is run; this is to ensure that if the same trigger is being run in two locations (for HA reasons) that we can deduplicate its events.

`payload: Any = None`

The payload for the event to send back to the task.

Must be natively JSON-serializable, or registered with the airflow serialization code.

`__repr__()`

```
class airflow.triggers.base.TaskSuccessEvent(*, xcoms=None, **kwargs)
```

Bases: `BaseTaskEndEvent`

Yield this event in order to end the task successfully.

`task_instance_state: airflow.utils.state.TaskInstanceState`

```
class airflow.triggers.base.TaskFailedEvent(*, xcoms=None, **kwargs)
```

Bases: `BaseTaskEndEvent`

Yield this event in order to end the task with failure.

`task_instance_state: airflow.utils.state.TaskInstanceState`

```
class airflow.triggers.base.TaskSkippedEvent(*, xcoms=None, **kwargs)
```

Bases: `BaseTaskEndEvent`

Yield this event in order to end the task with status ‘skipped’.

`task_instance_state: airflow.utils.state.TaskInstanceState`

```
airflow.triggers.base.trigger_event_discriminator(v)
airflow.triggers.base.DiscriminatedTriggerEvent
```

airflow.triggers.testing

Classes

| | |
|-----------------------|---|
| <i>SuccessTrigger</i> | A trigger that always succeeds immediately. |
| <i>FailureTrigger</i> | A trigger that always errors immediately. |

Module Contents

class airflow.triggers.testing.**SuccessTrigger**(**kwargs)

Bases: *airflow.triggers.base.BaseTrigger*

A trigger that always succeeds immediately.

Should only be used for testing.

serialize()

Return the information needed to reconstruct this Trigger.

Returns

Tuple of (class path, keyword arguments needed to re-instantiate).

Return type

tuple[str, dict[str, Any]]

async run()

Run the trigger in an asynchronous context.

The trigger should yield an Event whenever it wants to fire off an event, and return None if it is finished. Single-event triggers should thus yield and then immediately return.

If it yields, it is likely that it will be resumed very quickly, but it may not be (e.g. if the workload is being moved to another triggerer process, or a multi-event trigger was being used for a single-event task defer).

In either case, Trigger classes should assume they will be persisted, and then rely on cleanup() being called when they are no longer needed.

class airflow.triggers.testing.**FailureTrigger**(**kwargs)

Bases: *airflow.triggers.base.BaseTrigger*

A trigger that always errors immediately.

Should only be used for testing.

serialize()

Return the information needed to reconstruct this Trigger.

Returns

Tuple of (class path, keyword arguments needed to re-instantiate).

Return type

tuple[str, dict[str, Any]]

`async run()`

Run the trigger in an asynchronous context.

The trigger should yield an Event whenever it wants to fire off an event, and return None if it is finished. Single-event triggers should thus yield and then immediately return.

If it yields, it is likely that it will be resumed very quickly, but it may not be (e.g. if the workload is being moved to another triggerer process, or a multi-event trigger was being used for a single-event task defer).

In either case, Trigger classes should assume they will be persisted, and then rely on cleanup() being called when they are no longer needed.

You can read more about Triggers in *Deferrable Operators & Triggers*.

Timetables

Custom timetable implementations provide Airflow's scheduler additional logic to schedule DAG runs in ways not possible with built-in schedule expressions. All Timetables derive from *Timetable*.

Airflow has a set of Timetables that are considered public. You are free to extend their functionality by extending them:

`airflow.timetables`

Timetables.

Submodules

`airflow.timetables.assets`

Classes

`AssetOrTimeSchedule`

Combine time-based scheduling with event-based scheduling.

Module Contents

`class airflow.timetables.assets.AssetOrTimeSchedule(*, timetable, assets)`

Bases: `airflow.timetables.simple.AssetTriggeredTimetable`

Combine time-based scheduling with event-based scheduling.

`timetable`

`description = 'Triggered by assets or Uninferable'`

Human-readable description of the timetable.

For example, this can produce something like 'At 21:30, only on Friday' from the cron expression '30 21 * * 5'. This is used in the webserver UI.

`periodic`

Whether this timetable runs periodically.

This defaults to and should generally be *True*, but some special setups like `schedule=None` and "`@once`" set it to *False*.

can_be_scheduled

Whether this timetable can actually schedule runs in an automated manner.

This defaults to and should generally be *True* (including non periodic execution types like `@once` and data triggered tables), but `NullTimetable` sets this to *False*.

active_runs_limit

Maximum active runs that can be active at one time for a DAG.

This is called during DAG initialization, and the return value is used as the DAG's default `max_active_runs`. This should generally return `None`, but there are good reasons to limit DAG run parallelism in some cases, such as for `ContinuousTimetable`.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

validate()

Validate the timetable is correctly specified.

Override this method to provide run-time validation raised when a DAG is put into a dagbag. The default implementation does nothing.

Raises

`AirflowTimetableInvalid` on validation failure.

property summary: str

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

infer_manual_data_interval(*, run_after)

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

next_dagrun_info(*, last_automated_data_interval, restriction)

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- **last_automated_data_interval** (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG’s last scheduled or backfilled run (manual runs not considered).
- **restriction** (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

`airflow.timetables.base.DagRunInfo | None`

generate_run_id(*, run_type, **kwargs)

Generate Run ID based on Run Type, `run_after` and logical Date.

Parameters

- **run_type** (`airflow.utils.types.DagRunType`) – type of DagRun
- **data_interval** – the data interval
- **run_after** – the date before which dag run won’t start.

airflow.timetables.base**Classes**

| | |
|------------------------------|--|
| <code>DataInterval</code> | A data interval for a DagRun to operate over. |
| <code>TimeRestriction</code> | Restriction on when a DAG can be scheduled for a run. |
| <code>DagRunInfo</code> | Information to schedule a DagRun. |
| <code>Timetable</code> | Protocol that all Timetable classes are expected to implement. |

Module Contents**class airflow.timetables.base.DataInterval**

Bases: `NamedTuple`

A data interval for a DagRun to operate over.

Both `start` and `end` **MUST** be “aware”, i.e. contain timezone information.

start: `pendulum.DateTime`

end: `pendulum.DateTime`

classmethod exact(at)

Represent an “interval” containing only an exact time.

class airflow.timetables.base.TimeRestriction

Bases: `NamedTuple`

Restriction on when a DAG can be scheduled for a run.

Specifically, the run must not be earlier than `earliest`, nor later than `latest`. If `catchup` is `False`, the run must also not be earlier than the current time, i.e. “missed” schedules are not backfilled.

These values are generally set on the DAG or task’s `start_date`, `end_date`, and `catchup` arguments.

Both `earliest` and `latest`, if not `None`, are inclusive; a DAG run can happen exactly at either point of time. They are guaranteed to be aware (i.e. contain timezone information) for `TimeRestriction` instances created by Airflow.

earliest: `pendulum.DateTime | None`

latest: `pendulum.DateTime | None`

catchup: `bool`

class `airflow.timetables.base.DagRunInfo`

Bases: `NamedTuple`

Information to schedule a DagRun.

Instances of this will be returned by timetables when they are asked to schedule a DagRun creation.

run_after: `pendulum.DateTime`

The earliest time this DagRun is created and its tasks scheduled.

This **MUST** be “aware”, i.e. contain timezone information.

data_interval: `DataInterval`

The data interval this DagRun to operate over.

classmethod exact(at)

Represent a run on an exact time.

classmethod interval(start, end)

Represent a run on a continuous schedule.

In such a schedule, each data interval starts right after the previous one ends, and each run is scheduled right after the interval ends. This applies to all schedules prior to AIP-39 except `@once` and `None`.

property logical_date: `pendulum.DateTime`

Infer the logical date to represent a DagRun.

This replaces `execution_date` in Airflow 2.1 and prior. The idea is essentially the same, just a different name.

class `airflow.timetables.base.Timetable`

Bases: `Protocol`

Protocol that all Timetable classes are expected to implement.

description: `str = ''`

Human-readable description of the timetable.

For example, this can produce something like ‘At 21:30, only on Friday’ from the cron expression ‘`30 21 * * 5`’. This is used in the webserver UI.

periodic: bool = True

Whether this timetable runs periodically.

This defaults to and should generally be *True*, but some special setups like `schedule=None` and "`@once`" set it to *False*.

can_be_scheduled: bool = True

Whether this timetable can actually schedule runs in an automated manner.

This defaults to and should generally be *True* (including non periodic execution types like `@once` and data triggered tables), but `NullTimetable` sets this to *False*.

run_ordering: collections.abc.Sequence[str] = ('data_interval_end', 'logical_date')

How runs triggered from this timetable should be ordered in UI.

This should be a list of field names on the DAG run object.

active_runs_limit: int | None = None

Maximum active runs that can be active at one time for a DAG.

This is called during DAG initialization, and the return value is used as the DAG's default `max_active_runs`. This should generally return `None`, but there are good reasons to limit DAG run parallelism in some cases, such as for `ContinuousTimetable`.

asset_condition: airflow.sdkdefinitions.asset.BaseAsset

The asset condition that triggers a DAG using this timetable.

If this is not `None`, this should be an asset, or a combination of, that controls the DAG's asset triggers.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

validate()

Validate the timetable is correctly specified.

Override this method to provide run-time validation raised when a DAG is put into a dagbag. The default implementation does nothing.

Raises

`AirflowTimetableInvalid` on validation failure.

property summary: str

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

abstractmethod `infer_manual_data_interval`(*, `run_after`)

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

abstractmethod `next_dagrun_info`(*, `last_automated_data_interval`, `restriction`)

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- `last_automated_data_interval` (`DataInterval` / `None`) – The data interval of the associated DAG's last scheduled or backfilled run (manual runs not considered).
- `restriction` (`TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

`DagRunInfo` | `None`

`generate_run_id`(*, `run_type`, `run_after`, `data_interval`, **`extra`)

Generate a unique run ID.

Parameters

- `run_type` (`airflow.utils.types.DagRunType`) – The type of DAG run.
- `run_after` (`pendulum.DateTime`) – the datetime before which to Dag cannot run.
- `data_interval` (`DataInterval` / `None`) – The data interval of the DAG run.

airflow.timetables.datasets**Classes****`DatasetOrTimeSchedule`**

Deprecated alias for `AssetOrTimeSchedule`.

Module Contents**`class airflow.timetables.datasets.DatasetOrTimeSchedule`**

Deprecated alias for `AssetOrTimeSchedule`.

airflow.timetables.events**Classes**

| | |
|------------------------------|---|
| <code>EventsTimetable</code> | Timetable that schedules DAG runs at specific listed datetimes. |
|------------------------------|---|

Module Contents

```
class airflow.timetables.events.EventsTimetable(event_dates, restrict_to_events=False,  
    presorted=False, description=None)
```

Bases: `airflow.timetables.base.Timetable`

Timetable that schedules DAG runs at specific listed datetimes.

Suitable for predictable but truly irregular scheduling such as sporting events.

Parameters

- **event_dates** (`collections.abc.Iterable[pendulum.DateTime]`) – List of datetimes for the DAG to run at. Duplicates will be ignored. Must be finite and of reasonable size as it will be loaded in its entirety.
- **restrict_to_events** (`bool`) – Whether manual runs should use the most recent event or the current time
- **presorted** (`bool`) – if True, event_dates will be assumed to be in ascending order. Provides modest performance improvement for larger lists of event_dates.
- **description** (`str / None`) – A name for the timetable to display in the UI. Default None will be shown as “X Events” where X is the len of event_dates

`event_dates`

`restrict_to_events = False`

`property summary: str`

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable’s type name.

`__repr__()`

`next_dagrun_info(*, last_automated_data_interval, restriction)`

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- **last_automated_data_interval** (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG’s last scheduled or backfilled run (manual runs not considered).
- **restriction** (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type*airflow.timetables.base.DagRunInfo | None***`infer_manual_data_interval(*, run_after)`**

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

`serialize()`

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

`classmethod deserialize(data)`

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

`airflow.timetables.interval`**Attributes***Delta***Classes**`CronDataIntervalTimetable`

Timetable that schedules data intervals with a cron expression.

`DeltaDataIntervalTimetable`

Timetable that schedules data intervals with a time delta.

Module Contents**`airflow.timetables.interval.Delta`****`class airflow.timetables.interval.CronDataIntervalTimetable(cron, timezone)`**

Bases: `airflow.timetables._cron.CronMixin, _DataIntervalTimetable`

Timetable that schedules data intervals with a cron expression.

This corresponds to `schedule=<cron>`, where `<cron>` is either a five/six-segment representation, or one of `cron_presets`.

The implementation extends on croniter to add timezone awareness. This is because croniter works only with naive timestamps, and cannot consider DST when determining the next/previous time.

Don't pass `@once` in here; use `OnceTimetable` instead.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

infer_manual_data_interval(*, run_after)

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

class airflow.timetables.interval.DeltaDataIntervalTimetable(delta)

Bases: `airflow.timetables._delta.DeltaMixin, _DataIntervalTimetable`

Timetable that schedules data intervals with a time delta.

This corresponds to `schedule=<delta>`, where `<delta>` is either a `datetime.timedelta` or `dateutil.relativedelta.relativedelta` instance.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

__eq__(other)

Return if the offsets match.

This is only for testing purposes and should not be relied on otherwise.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

infer_manual_data_interval(run_after)

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

airflow.timetables.simple

Classes

| | |
|----------------------------------|---|
| <code>NullTimetable</code> | Timetable that never schedules anything. |
| <code>OnceTimetable</code> | Timetable that schedules the execution once as soon as possible. |
| <code>ContinuousTimetable</code> | Timetable that schedules continually, while still respecting start_date and end_date. |

Module Contents

`class airflow.timetables.simple.NullTimetable`

Bases: `_TrivialTimetable`

Timetable that never schedules anything.

This corresponds to `schedule=None`.

`can_be_scheduled = False`

Whether this timetable can actually schedule runs in an automated manner.

This defaults to and should generally be `True` (including non periodic execution types like `@once` and data triggered tables), but `NullTimetable` sets this to `False`.

`description: str = 'Never, external triggers only'`

Human-readable description of the timetable.

For example, this can produce something like 'At 21:30, only on Friday' from the cron expression '`'30 21 * * 5'`'. This is used in the webserver UI.

`property summary: str`

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

`next_dagrun_info(*, last_automated_data_interval, restriction)`

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- `last_automated_data_interval` (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG's last scheduled or backfilled run (manual runs not considered).
- `restriction` (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

airflow.timetables.base.DagRunInfo | None

class airflow.timetables.simple.OnceTimetable

Bases: `_TrivialTimetable`

Timetable that schedules the execution once as soon as possible.

This corresponds to `schedule="@once"`.

description: str = 'Once, as soon as possible'

Human-readable description of the timetable.

For example, this can produce something like 'At 21:30, only on Friday' from the cron expression '`30 21 * * 5`'. This is used in the webserver UI.

property summary: str

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

next_dagrun_info(*, last_automated_data_interval, restriction)

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- **last_automated_data_interval** (*airflow.timetables.base.DataInterval / None*) – The data interval of the associated DAG's last scheduled or backfilled run (manual runs not considered).
- **restriction** (*airflow.timetables.base.TimeRestriction*) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

airflow.timetables.base.DagRunInfo | None

class airflow.timetables.simple.ContinuousTimetable

Bases: `_TrivialTimetable`

Timetable that schedules continually, while still respecting `start_date` and `end_date`.

This corresponds to `schedule="@continuous"`.

description: str = 'As frequently as possible, but only one run at a time.'

Human-readable description of the timetable.

For example, this can produce something like 'At 21:30, only on Friday' from the cron expression '`30 21 * * 5`'. This is used in the webserver UI.

active_runs_limit = 1

Maximum active runs that can be active at one time for a DAG.

This is called during DAG initialization, and the return value is used as the DAG's default `max_active_runs`. This should generally return `None`, but there are good reasons to limit DAG run parallelism in some cases, such as for `ContinuousTimetable`.

property summary: str

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

next_dagrun_info(*, last_automated_data_interval, restriction)

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- **last_automated_data_interval** (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG's last scheduled or backfilled run (manual runs not considered).
- **restriction** (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

`airflow.timetables.base.DagRunInfo | None`

airflow.timetables.trigger

Classes

| | |
|---|--|
| <code>DeltaTriggerTimetable</code> | Timetable that triggers DAG runs according to a cron expression. |
| <code>CronTriggerTimetable</code> | Timetable that triggers DAG runs according to a cron expression. |
| <code>MultipleCronTriggerTimetable</code> | Timetable that triggers DAG runs according to multiple cron expressions. |

Module Contents

class airflow.timetables.trigger.DeltaTriggerTimetable(delta, *, interval=datetime.timedelta())

Bases: `airflow.timetables._delta.DeltaMixin, _TriggerTimetable`

Timetable that triggers DAG runs according to a cron expression.

This is different from `DeltaDataIntervalTimetable`, where the delta value specifies the *data interval* of a DAG run. With this timetable, the data intervals are specified independently. Also for the same reason, this timetable kicks off a DAG run immediately at the start of the period, instead of needing to wait for one data interval to pass.

Parameters

- **delta** (`datetime.timedelta / dateutil.relativedelta.relativedelta`) – How much time to wait between each run.
- **interval** (`datetime.timedelta / dateutil.relativedelta.relativedelta`) – The data interval of each run. Default is 0.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

```
class airflow.timetables.trigger.CronTriggerTimetable(cron, *, timezone,
                                                       interval=datetime.timedelta(),
                                                       run_immediately=False)
```

Bases: `airflow.timetables._cron.CronMixin, _TriggerTimetable`

Timetable that triggers DAG runs according to a cron expression.

This is different from `CronDataIntervalTimetable`, where the cron expression specifies the *data interval* of a DAG run. With this timetable, the data intervals are specified independently from the cron expression. Also for the same reason, this timetable kicks off a DAG run immediately at the start of the period (similar to POSIX cron), instead of needing to wait for one data interval to pass.

Don't pass `@once` in here; use `OnceTimetable` instead.

Parameters

- **cron** (`str`) – cron string that defines when to run
- **timezone** (`str / pendulum.tz.timezone.Timezone / pendulum.tz.timezone.FixedTimezone`) – Which timezone to use to interpret the cron string
- **interval** (`datetime.timedelta / dateutil.relativedelta.relativedelta`) – `timedelta` that defines the data interval start. Default 0.

`run_immediately` controls, if no `start_time` is given to the DAG, when the first run of the DAG should be scheduled. It has no effect if there already exist runs for this DAG.

- If `True`, always run immediately the most recent possible DAG run.
- If `False`, wait to run until the next scheduled time in the future.
- If passed a `timedelta`, will run the most recent possible DAG run if that run's `data_interval_end` is within `timedelta` of now.
- If `None`, the `timedelta` is calculated as 10% of the time between the most recent past scheduled time and the next scheduled time. E.g. if running every hour, this would run the previous time if less than 6 minutes had past since the previous run time, otherwise it would wait until the next hour.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

```
class airflow.timetables.trigger.MultipleCronTriggerTimetable(*crons, timezone,
                                                               interval=datetime.timedelta(),
                                                               run_immediately=False)
```

Bases: `airflow.timetables.base.Timetable`

Timetable that triggers DAG runs according to multiple cron expressions.

This combines multiple `CronTriggerTimetable` instances underneath, and triggers a DAG run whenever one of the timetables want to trigger a run.

Only at most one run is triggered for any given time, even if more than one timetable fires at the same time.

description = ''

Human-readable description of the timetable.

For example, this can produce something like 'At 21:30, only on Friday' from the cron expression '30 21 * * 5'. This is used in the webserver UI.

classmethod deserialize(data)

Deserialize a timetable from data.

This is called when a serialized DAG is deserialized. `data` will be whatever was returned by `serialize` during DAG serialization. The default implementation constructs the timetable without any arguments.

serialize()

Serialize the timetable for JSON encoding.

This is called during DAG serialization to store timetable information in the database. This should return a JSON-serializable dict that will be fed into `deserialize` when the DAG is deserialized. The default implementation returns an empty dict.

property summary: str

A short summary for the timetable.

This is used to display the timetable in the web UI. A cron expression timetable, for example, can use this to display the expression. The default implementation returns the timetable's type name.

`infer_manual_data_interval(*, run_after)`

When a DAG run is manually triggered, infer a data interval for it.

This is used for e.g. manually-triggered runs, where `run_after` would be when the user triggers the run. The default implementation raises `NotImplementedError`.

`next_dagrun_info(*, last_automated_data_interval, restriction)`

Provide information to schedule the next DagRun.

The default implementation raises `NotImplementedError`.

Parameters

- `last_automated_data_interval` (`airflow.timetables.base.DataInterval / None`) – The data interval of the associated DAG’s last scheduled or backfilled run (manual runs not considered).
- `restriction` (`airflow.timetables.base.TimeRestriction`) – Restriction to apply when scheduling the DAG run. See documentation of `TimeRestriction` for details.

Returns

Information on when the next DagRun can be scheduled. `None` means a DagRun will not happen. This does not mean no more runs will be scheduled even again for this DAG; the timetable can return a `DagRunInfo` object when asked at another time.

Return type

`airflow.timetables.base.DagRunInfo | None`

You can read more about Timetables in *Customizing DAG Scheduling with Timetables*.

Listeners

Listeners enable you to respond to DAG/Task lifecycle events.

This is implemented via `ListenerManager` class that provides hooks that can be implemented to respond to DAG/Task lifecycle events.

Added in version 2.5: Listener public interface has been added in version 2.5.

You can read more about Listeners in *Listeners*.

Extra Links

Extra links are dynamic links that could be added to Airflow independently from custom Operators. Normally they can be defined by the Operators, but plugins allow you to override the links on a global level.

You can read more about the Extra Links in *Define an operator extra link*.

3.11.4 Using Public Interface to integrate with external services and applications

Tasks in Airflow can orchestrate external services via Hooks and Operators. The core functionality of Airflow (such as authentication) can also be extended to leverage external services. You can read more about providers providers and core extensions they can provide in providers.

Executors

Executors are the mechanism by which task instances get run. All executors are derived from `BaseExecutor`. There are several executor implementations built-in Airflow, each with their own unique characteristics and capabilities.

The executor interface itself (the `BaseExecutor` class) is public, but the built-in executors are not (i.e. `KubernetesExecutor`, `LocalExecutor`, etc). This means that, to use `KubernetesExecutor` as an example, we may make changes to `KubernetesExecutor` in minor or patch Airflow releases which could break an executor that subclasses `KubernetesExecutor`. This is necessary to allow Airflow developers sufficient freedom to continue to improve the executors we offer. Accordingly, if you want to modify or extend a built-in executor, you should incorporate the full executor code into your project so that such changes will not break your derivative executor.

You can read more about executors and how to write your own in *Executor*.

Added in version 2.6: The executor interface has been present in Airflow for quite some time but prior to 2.6, there was executor-specific code elsewhere in the codebase. As of version 2.6 executors are fully decoupled, in the sense that Airflow core no longer needs to know about the behavior of specific executors. You could have succeeded with implementing a custom executor before Airflow 2.6, and a number of people did, but there were some hard-coded behaviours that preferred in-built executors, and custom executors could not provide full functionality that built-in executors had.

Secrets Backends

Airflow can be configured to rely on secrets backends to retrieve `Connection` and `Variable`. All secrets backends derive from `BaseSecretsBackend`.

All Secrets Backend implementations are public. You can extend their functionality:

`airflow.secrets`

Secrets framework provides means of getting connection objects from various sources.

The following sources are available:

- Environment variables
- Metastore database
- Local Filesystem Secrets Backend

Submodules

`airflow.secrets.base_secrets`

Classes

| | |
|---------------------------------|--|
| <code>BaseSecretsBackend</code> | Abstract base class to retrieve Connection object given a conn_id or Variable given a key. |
|---------------------------------|--|

Module Contents

`class airflow.secrets.base_secrets.BaseSecretsBackend`

Bases: `abc.ABC`

Abstract base class to retrieve Connection object given a conn_id or Variable given a key.

`static build_path(path_prefix, secret_id, sep='/')`

Given conn_id, build path for Secrets Backend.

Parameters

- `path_prefix (str)` – Prefix of the path to get secret
- `secret_id (str)` – Secret id

- **sep** (*str*) – separator used to concatenate connections_prefix and conn_id. Default: “/”

abstractmethod `get_conn_value(conn_id)`

Retrieve from Secrets Backend a string value representing the Connection object.

If the client your secrets backend uses already returns a python dict, you should override `get_connection` instead.

Parameters

conn_id (*str*) – connection id

deserialize_connection(conn_id, value)

Given a serialized representation of the airflow Connection, return an instance.

Looks at first character to determine how to deserialize.

Parameters

- **conn_id** (*str*) – connection id

- **value** (*str*) – the serialized representation of the Connection object

Returns

the deserialized Connection

Return type

airflow.models.connection.Connection

get_connection(conn_id)

Return connection object with a given conn_id.

Tries `get_conn_value` first and if not implemented, tries `get_conn_uri`

Parameters

conn_id (*str*) – connection id

abstractmethod `get_variable(key)`

Return value for Airflow Variable.

Parameters

key (*str*) – Variable Key

Returns

Variable Value

Return type

str | None

get_config(key)

Return value for Airflow Config Key.

Parameters

key (*str*) – Config Key

Returns

Config Value

Return type

str | None

airflow.secrets.cache

Re exporting the new cache module from Task SDK for backward compatibility.

airflow.secrets.environment_variables

Objects relating to sourcing connections from environment variables.

Attributes

`CONN_ENV_PREFIX`

`VAR_ENV_PREFIX`

Classes

`EnvironmentVariablesBackend`

Retrieves Connection object and Variable from environment variable.

Module Contents

`airflow.secrets.environment_variables.CONN_ENV_PREFIX = 'AIRFLOW_CONN_'`

`airflow.secrets.environment_variables.VAR_ENV_PREFIX = 'AIRFLOW_VAR_'`

`class airflow.secrets.environment_variables.EnvironmentVariablesBackend`

Bases: `airflow.secrets.BaseSecretsBackend`

Retrieves Connection object and Variable from environment variable.

get_conn_value(conn_id)

Retrieve from Secrets Backend a string value representing the Connection object.

If the client your secrets backend uses already returns a python dict, you should override `get_connection` instead.

Parameters

`conn_id (str)` – connection id

get_variable(key)

Get Airflow Variable from Environment Variable.

Parameters

`key (str)` – Variable Key

Returns

Variable Value

Return type

str | None

airflow.secrets.local_filesystem

Objects relating to retrieving connections and variables from local file.

Attributes

log

FILE_PARSERS

Classes

LocalFilesystemBackend

Retrieves Connection objects and Variables from local files.

Functions

get_connection_parameter_names()

Return *airflow.models.connection.Connection* constructor parameters.

load_variables(file_path)

Load variables from a text file.

load_connections_dict(file_path)

Load connection from text file.

Module Contents

`airflow.secrets.local_filesystem.log`

`airflow.secrets.local_filesystem.get_connection_parameter_names()`

Return *airflow.models.connection.Connection* constructor parameters.

`airflow.secrets.local_filesystem.FILE_PARSERS`

`airflow.secrets.local_filesystem.load_variables(file_path)`

Load variables from a text file.

JSON, YAML and .env files are supported.

Parameters

file_path (*str*) – The location of the file that will be processed.

`airflow.secrets.local_filesystem.load_connections_dict(file_path)`

Load connection from text file.

JSON, YAML and .env files are supported.

Returns

A dictionary where the key contains a connection ID and the value contains the connection.

Return type

`dict[str, Any]`

```
class airflow.secrets.local_filesystem.LocalFilesystemBackend(variables_file_path=None,
                                                               connections_file_path=None)
```

Bases: *airflow.secrets.base_secrets.BaseSecretsBackend*, *airflow.utils.log.logging_mixin.LoggingMixin*

Retrieves Connection objects and Variables from local files.

JSON, YAML and .env files are supported.

Parameters

- **variables_file_path** (*str* / *None*) – File location with variables data.
- **connections_file_path** (*str* / *None*) – File location with connection data.

variables_file = None

connections_file = None

get_connection(*conn_id*)

Return connection object with a given *conn_id*.

Tries `get_conn_value` first and if not implemented, tries `get_conn_uri`

Parameters

conn_id (*str*) – connection id

get_variable(*key*)

Return value for Airflow Variable.

Parameters

key (*str*) – Variable Key

Returns

Variable Value

Return type

str | *None*

airflow.secrets.metastore

Objects relating to sourcing connections from metastore database.

Classes

| | |
|-------------------------|---|
| MetastoreBackend | Retrieves Connection object and Variable from airflow metastore database. |
|-------------------------|---|

Module Contents

```
class airflow.secrets.metastore.MetastoreBackend
```

Bases: *airflow.secrets.BaseSecretsBackend*

Retrieves Connection object and Variable from airflow metastore database.

get_connection(*conn_id*, session=*NEW_SESSION*)

Get Airflow Connection from Metadata DB.

Parameters

- **conn_id** (*str*) – Connection ID
- **session** (*sqlalchemy.orm.Session*) – SQLAlchemy Session

Returns

Connection Object

Return type

`airflow.models.Connection | None`

get_variable(*key*, *session=NEW_SESSION*)

Get Airflow Variable from Metadata DB.

Parameters

- **key** (*str*) – Variable Key
- **session** (*sqlalchemy.orm.Session*) – SQLAlchemy Session

Returns

Variable Value

Return type

`str | None`

Attributes

`DEFAULT_SECRETS_SEARCH_PATH`

`DEFAULT_SECRETS_SEARCH_PATH_WORKERS`

Package Contents

```
airflow.secrets.DEFAULT_SECRETS_SEARCH_PATH =
['airflow.secrets.environment_variables.EnvironmentVariablesBackend', ...]

airflow.secrets.DEFAULT_SECRETS_SEARCH_PATH_WORKERS =
['airflow.secrets.environment_variables.EnvironmentVariablesBackend']
```

You can read more about Secret Backends in [Secrets Backend](#). You can also find all the available Secrets Backends implemented in community providers in `apache-airflow-providers:core-extensions/secrets-backends`.

Auth managers

Auth managers are responsible of user authentication and user authorization in Airflow. All auth managers are derived from `BaseAuthManager`.

The auth manager interface itself (the `BaseAuthManager` class) is public, but the different implementations of auth managers are not (i.e. `FabAuthManager`).

You can read more about auth managers and how to write your own in [Auth manager](#).

Connections

When creating Hooks, you can add custom Connections. You can read more about connections in `apache-airflow-providers:core-extensions/connections` for available Connections implemented in the community providers.

Extra Links

When creating Hooks, you can add custom Extra Links that are displayed when the tasks are run. You can find out more about extra links in apache-airflow-providers:core-extensions/extra-links that also shows available extra links implemented in the community providers.

Logging and Monitoring

You can extend the way how logs are written by Airflow. You can find out more about log writing in *Logging & Monitoring*.

The apache-airflow-providers:core-extensions/logging that also shows available log writers implemented in the community providers.

Decorators

DAG authors can use decorators to author dags using the *TaskFlow* concept. All Decorators derive from `TaskDecorator`.

Airflow has a set of Decorators that are considered public. You are free to extend their functionality by extending them:

`airflow.decorators`

You can read more about creating custom Decorators in *Creating Custom @task Decorators*.

Email notifications

Airflow has a built-in way of sending email notifications and it allows to extend it by adding custom email notification classes. You can read more about email notifications in *Email Configuration*.

Notifications

Airflow has a built-in extensible way of sending notifications using the various `on_*_callback`. You can read more about notifications in *Creating a notifier*.

Cluster Policies

Cluster Policies are the way to dynamically apply cluster-wide policies to the dags being parsed or tasks being executed. You can read more about Cluster Policies in *Cluster Policies*.

Lineage

Airflow can help track origins of data, what happens to it and where it moves over time. You can read more about lineage in *Lineage*.

3.11.5 What is not part of the Public Interface of Apache Airflow?

Everything not mentioned in this document should be considered as non-Public Interface.

Sometimes in other applications those components could be relied on to keep backwards compatibility, but in Airflow they are not parts of the Public Interface and might change any time:

- `Database structure` is considered to be an internal implementation detail and you should not assume the structure is going to be maintained in a backwards-compatible way.
- `Web UI` is continuously evolving and there are no backwards compatibility guarantees on HTML elements.
- Python classes except those explicitly mentioned in this document, are considered an internal implementation detail and you should not assume they will be maintained in a backwards-compatible way.

3.12 Best Practices

Creating a new DAG is a three-step process:

- writing Python code to create a DAG object,
- testing if the code meets your expectations,
- configuring environment dependencies to run your DAG

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

3.12.1 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 Custom Operator/Hook

Please follow our guide on *custom Operators*.

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 `data_interval_start` 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. Also, most connection types have unique parameter names in tasks, so you can declare a connection only once in `default_args` (for example `gcp_conn_id`) and it is automatically used by all operators that use this connection type.

Deleting a task

Be careful when deleting a task from a DAG. You would not be able to see the Task in Graph View, Grid View, etc making it difficult to check the logs of that Task from the Webserver. If that is not desired, please create a new DAG.

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.

Top level Python Code

You should avoid writing the top level code which is not necessary to create Operators and build DAG relations between them. This is because of the design decision for the scheduler of Airflow and the impact the top-level code parsing speed on both performance and scalability of Airflow.

Airflow scheduler executes the code outside the Operator's `execute` methods with the minimum interval of `min_file_process_interval` seconds. This is done in order to allow dynamic scheduling of the dags - where scheduling and dependencies might change over time and impact the next schedule of the DAG. Airflow scheduler tries to continuously make sure that what you have in dags is correctly reflected in scheduled tasks.

Specifically you should not run any database access, heavy computations and networking operations.

One of the important factors impacting DAG loading time, that might be overlooked by Python developers is that top-level imports might take surprisingly a lot of time and they can generate a lot of overhead and this can be easily avoided by converting them to local imports inside Python callables for example.

Consider the two examples below. In the first example, DAG will take an additional 1000 seconds to parse than the functionally equivalent DAG in the second example where the `expensive_api_call` is executed from the context of its task.

Not avoiding top-level DAG code:

```
import pendulum

from airflow.sdk import DAG
from airflow.sdk import task

def expensive_api_call():
    print("Hello from Airflow!")
    sleep(1000)

my_expensive_response = expensive_api_call()

with DAG(
    dag_id="example_python_operator",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
```

(continues on next page)

(continued from previous page)

```
) as dag:

    @task()
    def print_expensive_api_call():
        print(my_expensive_response)
```

Avoiding top-level DAG code:

```
import pendulum

from airflow.sdk import DAG
from airflow.sdk import task

def expensive_api_call():
    sleep(1000)
    return "Hello from Airflow!"

with DAG(
    dag_id="example_python_operator",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
) as dag:

    @task()
    def print_expensive_api_call():
        my_expensive_response = expensive_api_call()
        print(my_expensive_response)
```

In the first example, `expensive_api_call` is executed each time the DAG file is parsed, which will result in suboptimal performance in the DAG file processing. In the second example, `expensive_api_call` is only called when the task is running and thus is able to be parsed without suffering any performance hits. To test it out yourself, implement the first DAG and see “Hello from Airflow!” printed in the scheduler logs!

Note that import statements also count as top-level code. So, if you have an import statement that takes a long time or the imported module itself executes code at the top-level, that can also impact the performance of the scheduler. The following example illustrates how to handle expensive imports.

```
# It's ok to import modules that are not expensive to load at top-level of a DAG file
import random
import pendulum

# Expensive imports should be avoided as top level imports, because DAG files are parsed
# frequently, resulting in top-level code being executed.
#
# import pandas
# import torch
# import tensorflow
#
```

(continues on next page)

(continued from previous page)

```
...
@task()
def do_stuff_with_pandas_and_torch():
    import pandas
    import torch

    # do some operations using pandas and torch

@task()
def do_stuff_with_tensorflow():
    import tensorflow

    # do some operations using tensorflow
```

How to check if my code is “top-level” code

In order to understand whether your code is “top-level” or not you need to understand a lot of intricacies of how parsing Python works. In general, when Python parses the python file it executes the code it sees, except (in general) internal code of the methods that it does not execute.

It has a number of special cases that are not obvious - for example top-level code also means any code that is used to determine default values of methods.

However, there is an easy way to check whether your code is “top-level” or not. You simply need to parse your code and see if the piece of code gets executed.

Imagine this code:

```
from airflow.sdk import DAG
from airflow.providers.standard.operators.python import PythonOperator
import pendulum

def get_task_id():
    return "print_array_task"  # <- is that code going to be executed?

def get_array():
    return [1, 2, 3]  # <- is that code going to be executed?

with DAG(
    dag_id="example_python_operator",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
) as dag:
    operator = PythonOperator(
        task_id=get_task_id(),
        python_callable=get_array,
```

(continues on next page)

(continued from previous page)

```
    python_callable=get_array,
    dag=dag,
)
```

What you can do to check it is add some print statements to the code you want to check and then run `python <my_dag_file>.py`.

```
from airflow.sdk import DAG
from airflow.providers.standard.operators.python import PythonOperator
import pendulum

def get_task_id():
    print("Executing 1")
    return "print_array_task" # <- is that code going to be executed? YES

def get_array():
    print("Executing 2")
    return [1, 2, 3] # <- is that code going to be executed? NO

with DAG(
    dag_id="example_python_operator",
    schedule=None,
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    catchup=False,
    tags=["example"],
) as dag:
    operator = PythonOperator(
        task_id=get_task_id(),
        python_callable=get_array,
        dag=dag,
)
```

When you execute that code you will see:

```
root@cf85ab34571e:/opt/airflow# python /files/test_python.py
Executing 1
```

This means that the `get_array` is not executed as top-level code, but `get_task_id` is.

Code Quality and Linting

Maintaining high code quality is essential for the reliability and maintainability of your Airflow workflows. Utilizing linting tools can help identify potential issues and enforce coding standards. One such tool is `ruff`, a fast Python linter that now includes specific rules for Airflow.

`ruff` assists in detecting deprecated features and patterns that may affect your migration to Airflow 3.0. For instance, it includes rules prefixed with `AIR` to flag potential issues. The full list is detailed in [Airflow \(AIR\)](#).

Installing and Using ruff

1. **Installation:** Install ruff using pip:

```
pip install "ruff>=0.11.6"
```

2. **Running ruff:** Execute ruff to check your dags for potential issues:

```
ruff check dags/ --select AIR3 --preview
```

This command will analyze your dags located in the dags/ directory and report any issues related to the specified rules.

Example

Given a legacy DAG defined as:

```
from airflow import dag
from airflow.datasets import Dataset
from airflow.sensors.filesystem import FileSensor

@dag()
def legacy_dag():
    FileSensor(task_id="wait_for_file", filepath="/tmp/test_file")
```

Running ruff will produce:

```
dags/legacy_dag.py:7:2: AIR301 DAG should have an explicit schedule argument
dags/legacy_dag.py:12:6: AIR302 schedule_interval is removed in Airflow 3.0
dags/legacy_dag.py:17:15: AIR302 airflow.datasets.Dataset is removed in Airflow 3.0
dags/legacy_dag.py:19:5: AIR303 airflow.sensors.filesystem.FileSensor is moved into
  ↵ ``standard`` provider in Airflow 3.0
```

By integrating ruff into your development workflow, you can proactively address deprecations and maintain code quality, facilitating smoother transitions between Airflow versions.

For more information on ruff and its integration with Airflow, refer to the [official Airflow documentation](#).

Dynamic DAG Generation

Sometimes writing dags manually isn't practical. Maybe you have a lot of dags that do similar things with just a parameter changing between them. Or maybe you need a set of dags to load tables, but don't want to manually update dags every time those tables change. In these and other cases, it can be more useful to dynamically generate dags.

Avoiding excessive processing at the top level code described in the previous chapter is especially important in case of dynamic DAG configuration, which can be configured essentially in one of those ways:

- via [environment variables](#) (not to be mistaken with the *Airflow Variables*)
- via externally provided, generated Python code, containing meta-data in the DAG folder
- via externally provided, generated configuration meta-data file in the DAG folder

Some cases of dynamic DAG generation are described in the *Dynamic DAG Generation* section.

Airflow Variables

Using Airflow Variables yields network calls and database access, so their usage in top-level Python code for dags should be avoided as much as possible, as mentioned in the previous chapter, *Top level Python Code*. If Airflow Variables must be used in top-level DAG code, then their impact on DAG parsing can be mitigated by *enabling the experimental cache*, configured with a sensible *ttl*.

You can use the Airflow Variables freely inside the `execute()` methods of the operators, but you can also pass the Airflow Variables to the existing operators via 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> }}
```

In top-level code, variables using jinja templates do not produce a request until a task is running, whereas, `Variable.get()` produces a request every time the dag file is parsed by the scheduler if caching is not enabled. Using `Variable.get()` without *enabling caching* will lead to suboptimal performance in the dag file processing. In some cases this can cause the dag file to timeout before it is fully parsed.

Bad example:

```
from airflow.sdk import Variable

foo_var = Variable.get("foo") # AVOID THAT
bash_use_variable_bad_1 = BashOperator(
    task_id="bash_use_variable_bad_1",
    bash_command="echo variable foo=${foo_env}",
    env={"foo_env": foo_var}
)

bash_use_variable_bad_2 = BashOperator(
    task_id="bash_use_variable_bad_2",
    bash_command=f"echo variable foo=${Variable.get('foo')}", # AVOID THAT
)

bash_use_variable_bad_3 = BashOperator(
    task_id="bash_use_variable_bad_3",
    bash_command="echo variable foo=${foo_env}",
    env={"foo_env": Variable.get("foo")}, # AVOID THAT
)
```

Good example:

```
bash_use_variable_good = BashOperator(
    task_id="bash_use_variable_good",
    bash_command="echo variable foo=${foo_env}",
    env={"foo_env": "{{ var.value.get('foo') }}"},
```

```
@task
def my_task():
    var = Variable.get("foo") # This is ok since my_task is called only during task run,
```

(continues on next page)

(continued from previous page)

```
↪ not during DAG scan.
print(var)
```

For security purpose, you're recommended to use the [Secrets Backend](#) for any variable that contains sensitive data.

Timetables

Avoid using Airflow Variables/Connections or accessing Airflow database at the top level of your timetable code. Database access should be delayed until the execution time of the DAG. This means that you should not have variables/connections retrieval as argument to your timetable class initialization or have Variable/connection at the top level of your custom timetable module.

Bad example:

```
from airflow.sdk import Variable
from airflow.timetables.interval import CronDataIntervalTimetable

class CustomTimetable(CronDataIntervalTimetable):
    def __init__(self, *args, something=Variable.get("something"), **kwargs):
        self._something = something
        super().__init__(*args, **kwargs)
```

Good example:

```
from airflow.sdk import Variable
from airflow.timetables.interval import CronDataIntervalTimetable

class CustomTimetable(CronDataIntervalTimetable):
    def __init__(self, *args, something="something", **kwargs):
        self._something = Variable.get(something)
        super().__init__(*args, **kwargs)
```

Triggering dags after changes

Avoid triggering dags immediately after changing them or any other accompanying files that you change in the DAG folder.

You should give the system sufficient time to process the changed files. This takes several steps. First the files have to be distributed to scheduler - usually via distributed filesystem or Git-Sync, then scheduler has to parse the Python files and store them in the database. Depending on your configuration, speed of your distributed filesystem, number of files, number of dags, number of changes in the files, sizes of the files, number of schedulers, speed of CPUS, this can take from seconds to minutes, in extreme cases many minutes. You should wait for your DAG to appear in the UI to be able to trigger it.

In case you see long delays between updating it and the time it is ready to be triggered, you can look at the following configuration parameters and fine tune them according your needs (see details of each parameter by following the links):

- *scheduler_idle_sleep_time*
- *min_file_process_interval*
- *refresh_interval*
- *parsing_processes*

- *file_parsing_sort_mode*

Example of watcher pattern with trigger rules

The watcher pattern is how we call a DAG with a task that is “watching” the states of the other tasks. Its primary purpose is to fail a DAG Run when any other task fail. The need came from the Airflow system tests that are dags with different tasks (similarly like a test containing steps).

Normally, when any task fails, all other tasks are not executed and the whole DAG Run gets failed status too. But when we use trigger rules, we can disrupt the normal flow of running tasks and the whole DAG may represent different status that we expect. For example, we can have a teardown task (with trigger rule set to `TriggerRule.ALL_DONE`) that will be executed regardless of the state of the other tasks (e.g. to clean up the resources). In such situation, the DAG would always run this task and the DAG Run will get the status of this particular task, so we can potentially lose the information about failing tasks. If we want to ensure that the DAG with teardown task would fail if any task fails, we need to use the watcher pattern. The watcher task is a task that will always fail if triggered, but it needs to be triggered only if any other task fails. It needs to have a trigger rule set to `TriggerRule.ONE_FAILED` and it needs also to be a downstream task for all other tasks in the DAG. Thanks to this, if every other task will pass, the watcher will be skipped, but when something fails, the watcher task will be executed and fail making the DAG Run fail too.

Note

Be aware that trigger rules only rely on the direct upstream (parent) tasks, e.g. `TriggerRule.ONE_FAILED` will ignore any failed (or `upstream_failed`) tasks that are not a direct parent of the parameterized task.

It's easier to grab the concept with an example. Let's say that we have the following DAG:

```
from datetime import datetime

from airflow.sdk import DAG
from airflow.sdk import task
from airflow.exceptions import AirflowException
from airflow.providers.standard.operators.bash import BashOperator
from airflow.utils.trigger_rule import TriggerRule

@task(trigger_rule=TriggerRule.ONE_FAILED, retries=0)
def watcher():
    raise AirflowException("Failing task because one or more upstream tasks failed.")

with DAG(
    dag_id="watcher_example",
    schedule="@once",
    start_date=datetime(2021, 1, 1),
    catchup=False,
) as dag:
    failing_task = BashOperator(task_id="failing_task", bash_command="exit 1", retries=0)
    passing_task = BashOperator(task_id="passing_task", bash_command="echo passing_task")
    teardown = BashOperator(
        task_id="teardown",
        bash_command="echo teardown",
        trigger_rule=TriggerRule.ALL_DONE,
    )
```

(continues on next page)

(continued from previous page)

```
failing_task >> passing_task >> teardown
list(dag.tasks) >> watcher()
```

The visual representation of this DAG after execution looks like this:



We have several tasks that serve different purposes:

- `failing_task` always fails,
- `passing_task` always succeeds (if executed),
- `teardown` is always triggered (regardless the states of the other tasks) and it should always succeed,
- `watcher` is a downstream task for each other task, i.e. it will be triggered when any task fails and thus fail the whole DAG Run, since it's a leaf task.

It's important to note, that without `watcher` task, the whole DAG Run will get the `success` state, since the only failing task is not the leaf task, and the `teardown` task will finish with `success`. If we want the `watcher` to monitor the state of all tasks, we need to make it dependent on all of them separately. Thanks to this, we can fail the DAG Run if any of the tasks fail. Note that the `watcher` task has a trigger rule set to "`one_failed`". On the other hand, without the `teardown` task, the `watcher` task will not be needed, because `failing_task` will propagate its `failed` state to downstream task `passed_task` and the whole DAG Run will also get the `failed` status.

Using `AirflowClusterPolicySkipDag` exception in cluster policies to skip specific dags

Added in version 2.7.

Airflow dags can usually be deployed and updated with the specific branch of Git repository via `git-sync`. But, when you have to run multiple Airflow clusters for some operational reasons, it's very cumbersome to maintain multiple Git branches. Especially, you have some difficulties when you need to synchronize two separate branches (like `prod` and `beta`) periodically with proper branching strategy.

- cherry-pick is too cumbersome to maintain Git repository.
- hard-reset is not recommended way for GitOps

So, you can consider connecting multiple Airflow clusters with same Git branch (like `main`), and maintaining those with different environment variables and different connection configurations with same `connection_id`. You can also raise `AirflowClusterPolicySkipDag` exception on the cluster policy, to load specific dags to `DagBag` on the specific Airflow deployment only, if needed.

```
def dag_policy(dag: DAG):
    """Skipping the DAG with `only_for_beta` tag."""

    if "only_for_beta" in dag.tags:
        raise AirflowClusterPolicySkipDag(
            f"DAG {dag.dag_id} is not loaded on the production cluster, due to `only_for_beta` tag."
        )
```

The example above, shows the `dag_policy` code snippet to skip the DAG depending on the tags it has.

3.12.2 Reducing DAG complexity

While Airflow is good in handling a lot of dags with a lot of task and dependencies between them, when you have many complex dags, their complexity might impact performance of scheduling. One of the ways to keep your Airflow instance performant and well utilized, you should strive to simplify and optimize your dags whenever possible - you have to remember that DAG parsing process and creation is just executing Python code and it's up to you to make it as performant as possible. There are no magic recipes for making your DAG "less complex" - since this is a Python code, it's the DAG writer who controls the complexity of their code.

There are no "metrics" for DAG complexity, especially, there are no metrics that can tell you whether your DAG is "simple enough". However, as with any Python code, you can definitely tell that your DAG code is "simpler" or "faster" when it is optimized. If you want to optimize your dags there are the following actions you can take:

- Make your DAG load faster. This is a single improvement advice that might be implemented in various ways but this is the one that has biggest impact on scheduler's performance. Whenever you have a chance to make your DAG load faster - go for it, if your goal is to improve performance. Look at the *Top level Python Code* to get some tips of how you can do it. Also see at *DAG Loader Test* on how to asses your DAG loading time.
- Make your DAG generate simpler structure. Every task dependency adds additional processing overhead for scheduling and execution. The DAG that has simple linear structure A → B → C will experience less delays in task scheduling than DAG that has a deeply nested tree structure with exponentially growing number of depending tasks for example. If you can make your dags more linear - where at single point in execution there are as few potential candidates to run among the tasks, this will likely improve overall scheduling performance.
- Make smaller number of dags per file. While Airflow 2 is optimized for the case of having multiple dags in one file, there are some parts of the system that make it sometimes less performant, or introduce more delays than having those dags split among many files. Just the fact that one file can only be parsed by one FileProcessor, makes it less scalable for example. If you have many dags generated from one file, consider splitting them if you observe it takes a long time to reflect changes in your DAG files in the UI of Airflow.
- Write efficient Python code. A balance must be struck between fewer dags per file, as stated above, and writing less code overall. Creating the Python files that describe dags should follow best programming practices and not be treated like configurations. If your dags share similar code you should not copy them over and over again to a large number of nearly identical source files, as this will cause a number of unnecessary repeated imports of the same resources. Rather, you should aim to minimize repeated code across all of your dags so that the application can run efficiently and can be easily debugged. See *Code Quality and Linting* on how to create multiple dags with similar code.

3.12.3 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. Make sure that you load your DAG in an environment that corresponds to your scheduler environment - with the same dependencies, environment variables, common code referred from the DAG.

This is also a great way to check if your DAG loads faster after an optimization, if you want to attempt to optimize DAG loading time. Simply run the DAG and measure the time it takes, but again you have to make sure your DAG runs with the same dependencies, environment variables, common code.

There are many ways to measure the time of processing, one of them in Linux environment is to use built-in `time` command. Make sure to run it several times in succession to account for caching effects. Compare the results before and after the optimization (in the same conditions - using the same machine, environment etc.) in order to assess the impact of the optimization.

```
time python airflow/example_dags/example_python_operator.py
```

Result:

```
real    0m0.699s
user    0m0.590s
sys     0m0.108s
```

The important metrics is the “real time” - which tells you how long time it took to process the DAG. Note that when loading the file this way, you are starting a new interpreter so there is an initial loading time that is not present when Airflow parses the DAG. You can assess the time of initialization by running:

```
time python -c ''
```

Result:

```
real    0m0.073s
user    0m0.037s
sys     0m0.039s
```

In this case the initial interpreter startup time is ~ 0.07s which is about 10% of time needed to parse the `example_python_operator.py` above so the actual parsing time is about ~ 0.62 s for the example DAG.

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:

```
import pytest

from airflow.models import DagBag

@pytest.fixture()
def dagbag():
    return DagBag()

def test_dag_loaded(dagbag):
    dag = dagbag.get_dag(dag_id="hello_world")
    assert 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

```
def assert_dag_dict_equal(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():
    assert_dag_dict_equal(
        {
            "DummyInstruction_0": ["DummyInstruction_1"],
            "DummyInstruction_1": ["DummyInstruction_2"],
            "DummyInstruction_2": ["DummyInstruction_3"],
            "DummyInstruction_3": [],
        },
        dag,
    )
```

Unit test for custom operator:

```
import datetime
import pendulum
import pytest

from airflow.sdk import DAG
from airflow.utils.state import DagRunState, TaskInstanceState
from airflow.utils.types import DagRunTriggeredByType, DagRunType

DATA_INTERVAL_START = pendulum.datetime(2021, 9, 13, tz="UTC")
DATA_INTERVAL_END = DATA_INTERVAL_START + datetime.timedelta(days=1)

TEST_DAG_ID = "my_custom_operator_dag"
TEST_TASK_ID = "my_custom_operator_task"
TEST_RUN_ID = "my_custom_operator_dag_run"

@pytest.fixture()
def dag():
    with DAG(
        dag_id=TEST_DAG_ID,
        schedule="@daily",
        start_date=DATA_INTERVAL_START,
    ) as dag:
        MyCustomOperator(
            task_id=TEST_TASK_ID,
            prefix="s3://bucket/some/prefix",
        )
    return dag

def test_my_custom_operator_execute_no_trigger(dag):
```

(continues on next page)

(continued from previous page)

```

dagrun = dag.create_dagrun(
    run_id=TEST_RUN_ID,
    logical_date=DATA_INTERVAL_START,
    data_interval=(DATA_INTERVAL_START, DATA_INTERVAL_END),
    run_type=DagRunType.MANUAL,
    triggered_by=DagRunTriggeredByType.TIMETABLE,
    state=DagRunState.RUNNING,
    start_date=DATA_INTERVAL_END,
)
ti = dagrun.get_task_instance(task_id=TEST_TASK_ID)
ti.task = dag.get_task(task_id=TEST_TASK_ID)
ti.run(ignore_ti_state=True)
assert ti.state == TaskInstanceState.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/")

```

3.12.4 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_connection_from_secrets("my_conn").login
```

3.12.5 Metadata DB maintenance

Over time, the metadata database will increase its storage footprint as more DAG and task runs and event logs accumulate.

You can use the Airflow CLI to purge old data with the command `airflow db clean`.

See `db clean usage` for more details.

3.12.6 Upgrades and downgrades

Backup your database

It's always a wise idea to backup the metadata database before undertaking any operation modifying the database.

Disable the scheduler

You might consider disabling the Airflow cluster while you perform such maintenance.

One way to do so would be to set the param `[scheduler] > use_job_schedule` to `False` and wait for any running dags to complete; after this no new DAG runs will be created unless externally triggered.

A *better* way (though it's a bit more manual) is to use the `dags pause` command. You'll need to keep track of the dags that are paused before you begin this operation so that you know which ones to unpause after maintenance is complete. First run `airflow dags list` and store the list of unpause dags. Then use this same list to run both `dags pause` for each DAG prior to maintenance, and `dags unpause` after. A benefit of this is you can try un-pausing just one or two dags (perhaps dedicated *test dags*) after the upgrade to make sure things are working before turning everything back on.

Add “integration test” dags

It can be helpful to add a couple “integration test” dags that use all the common services in your ecosystem (e.g. S3, Snowflake, Vault) but with test resources or “dev” accounts. These test dags can be the ones you turn on *first* after an upgrade, because if they fail, it doesn't matter and you can revert to your backup without negative consequences. However, if they succeed, they should prove that your cluster is able to run tasks with the libraries and services that you need to use.

For example, if you use an external secrets backend, make sure you have a task that retrieves a connection. If you use KubernetesPodOperator, add a task that runs `sleep 30; echo "hello"`. If you need to write to s3, do so in a test task. And if you need to access a database, add a task that does `select 1` from the server.

Prune data before upgrading

Some database migrations can be time-consuming. If your metadata database is very large, consider pruning some of the old data with the `db clean` command prior to performing the upgrade. *Use with caution.*

3.12.7 Handling conflicting/complex Python dependencies

Airflow has many Python dependencies and sometimes the Airflow dependencies are conflicting with dependencies that your task code expects. Since - by default - Airflow environment is just a single set of Python dependencies and single Python environment, often there might also be cases that some of your tasks require different dependencies than other tasks and the dependencies basically conflict between those tasks.

If you are using pre-defined Airflow Operators to talk to external services, there is not much choice, but usually those operators will have dependencies that are not conflicting with basic Airflow dependencies. Airflow uses constraints mechanism which means that you have a “fixed” set of dependencies that the community guarantees that Airflow can be installed with (including all community providers) without triggering conflicts. However, you can upgrade the providers independently and their constraints do not limit you, so the chance of a conflicting dependency is lower (you still have to test those dependencies). Therefore, when you are using pre-defined operators, chance is that you will have little, to no problems with conflicting dependencies.

However, when you are approaching Airflow in a more “modern way”, where you use TaskFlow API and most of your operators are written using custom python code, or when you want to write your own Custom Operator, you might get to the point where the dependencies required by the custom code of yours are conflicting with those of Airflow, or even that dependencies of several of your Custom Operators introduce conflicts between themselves.

There are a number of strategies that can be employed to mitigate the problem. And while dealing with dependency conflict in custom operators is difficult, it’s actually quite a bit easier when it comes to using `airflow.providers.standard.operators.python.PythonVirtualenvOperator` or `airflow.providers.standard.operators.python.ExternalPythonOperator` - either directly using classic “operator” approach or by using tasks decorated with `@task.virtualenv` or `@task.external_python` decorators if you use TaskFlow.

Let’s start from the strategies that are easiest to implement (having some limits and overhead), and we will gradually go through those strategies that requires some changes in your Airflow deployment.

Using PythonVirtualenvOperator

This is simplest to use and most limited strategy. The `PythonVirtualenvOperator` allows you to dynamically create a virtualenv that your Python callable function will execute in. In the modern TaskFlow approach described in [Pythonic DAGs with the TaskFlow API](#), this also can be done with decorating your callable with `@task.virtualenv` decorator (recommended way of using the operator). Each `airflow.providers.standard.operators.python.PythonVirtualenvOperator` task can have its own independent Python virtualenv (dynamically created every time the task is run) and can specify fine-grained set of requirements that need to be installed for that task to execute.

The operator takes care of:

- creating the virtualenv based on your environment
- serializing your Python callable and passing it to execution by the virtualenv Python interpreter
- executing it and retrieving the result of the callable and pushing it via xcom if specified

The benefits of the operator are:

- There is no need to prepare the venv upfront. It will be dynamically created before task is run, and removed after it is finished, so there is nothing special (except having virtualenv package in your Airflow dependencies) to make use of multiple virtual environments
- You can run tasks with different sets of dependencies on the same workers - thus Memory resources are reused (though see below about the CPU overhead involved in creating the venvs).

- In bigger installations, DAG Authors do not need to ask anyone to create the venvs for you. As a DAG Author, you only have to have virtualenv dependency installed and you can specify and modify the environments as you see fit.
- No changes in deployment requirements - whether you use Local virtualenv, or Docker, or Kubernetes, the tasks will work without adding anything to your deployment.
- No need to learn more about containers, Kubernetes as a DAG Author. Only knowledge of Python requirements is required to author dags this way.

There are certain limitations and overhead introduced by this operator:

- Your python callable has to be serializable. There are a number of python objects that are not serializable using standard `pickle` library. You can mitigate some of those limitations by using `dill` library but even that library does not solve all the serialization limitations.
- All dependencies that are not available in the Airflow environment must be locally imported in the callable you use and the top-level Python code of your DAG should not import/use those libraries.
- The virtual environments are run in the same operating system, so they cannot have conflicting system-level dependencies (`apt` or `yum` installable packages). Only Python dependencies can be independently installed in those environments.
- The operator adds a CPU, networking and elapsed time overhead for running each task - Airflow has to re-create the virtualenv from scratch for each task
- The workers need to have access to PyPI or private repositories to install dependencies
- The dynamic creation of virtualenv is prone to transient failures (for example when your repo is not available or when there is a networking issue with reaching the repository)
- It's easy to fall into a "too" dynamic environment - since the dependencies you install might get upgraded and their transitive dependencies might get independent upgrades you might end up with the situation where your task will stop working because someone released a new version of a dependency or you might fall a victim of "supply chain" attack where new version of a dependency might become malicious
- The tasks are only isolated from each other via running in different environments. This makes it possible that running tasks will still interfere with each other - for example subsequent tasks executed on the same worker might be affected by previous tasks creating/modifying files etc.

You can see detailed examples of using `airflow.providers.standard.operators.python.PythonVirtualenvOperator` in [this section in the Taskflow API tutorial](#).

Using ExternalPythonOperator

Added in version 2.4.

A bit more involved but with significantly less overhead, security, stability problems is to use the `airflow.providers.standard.operators.python.ExternalPythonOperator``. In the modern TaskFlow approach described in [Pythonic DAGs with the TaskFlow API](#), this also can be done with decorating your callable with `@task.external_python` decorator (recommended way of using the operator). It requires, however, that you have a pre-existing, immutable Python environment, that is prepared upfront. Unlike in `airflow.providers.standard.operators.python.PythonVirtualenvOperator` you cannot add new dependencies to such pre-existing environment. All dependencies you need should be added upfront in your environment and available in all the workers in case your Airflow runs in a distributed environment.

This way you avoid the overhead and problems of re-creating the virtual environment but they have to be prepared and deployed together with Airflow installation. Usually people who manage Airflow installation need to be involved, and in bigger installations those are usually different people than DAG Authors (DevOps/System Admins).

Those virtual environments can be prepared in various ways - if you use LocalExecutor they just need to be installed at the machine where scheduler is run, if you are using distributed Celery virtualenv installations, there should be a

pipeline that installs those virtual environments across multiple machines, finally if you are using Docker Image (for example via Kubernetes), the virtualenv creation should be added to the pipeline of your custom image building.

The benefits of the operator are:

- No setup overhead when running the task. The virtualenv is ready when you start running a task.
- You can run tasks with different sets of dependencies on the same workers - thus all resources are reused.
- There is no need to have access by workers to PyPI or private repositories. Less chance for transient errors resulting from networking.
- The dependencies can be pre-vetted by the admins and your security team, no unexpected, new code will be added dynamically. This is good for both, security and stability.
- Limited impact on your deployment - you do not need to switch to Docker containers or Kubernetes to make a good use of the operator.
- No need to learn more about containers, Kubernetes as a DAG Author. Only knowledge of Python, requirements is required to author dags this way.

The drawbacks:

- Your environment needs to have the virtual environments prepared upfront. This usually means that you cannot change it on the fly, adding new or changing requirements require at least an Airflow re-deployment and iteration time when you work on new versions might be longer.
- Your python callable has to be serializable. There are a number of python objects that are not serializable using standard `pickle` library. You can mitigate some of those limitations by using `dill` library but even that library does not solve all the serialization limitations.
- All dependencies that are not available in Airflow environment must be locally imported in the callable you use and the top-level Python code of your DAG should not import/use those libraries.
- The virtual environments are run in the same operating system, so they cannot have conflicting system-level dependencies (`apt` or `yum` installable packages). Only Python dependencies can be independently installed in those environments
- The tasks are only isolated from each other via running in different environments. This makes it possible that running tasks will still interfere with each other - for example subsequent tasks executed on the same worker might be affected by previous tasks creating/modifying files etc.

You can think about the `PythonVirtualenvOperator` and `ExternalPythonOperator` as counterparts - that make it smoother to move from development phase to production phase. As a DAG author you'd normally iterate with dependencies and develop your DAG using `PythonVirtualenvOperator` (thus decorating your tasks with `@task.virtualenv` decorators) while after the iteration and changes you would likely want to change it for production to switch to the `ExternalPythonOperator` (and `@task.external_python`) after your DevOps/System Admin teams deploy your new dependencies in pre-existing virtualenv in production. The nice thing about this is that you can switch the decorator back at any time and continue developing it “dynamically” with `PythonVirtualenvOperator`.

You can see detailed examples of using `airflow.providers.standard.operators.python.ExternalPythonOperator` in [Taskflow External Python example](#)

Using DockerOperator or Kubernetes Pod Operator

Another strategy is to use the `airflow.providers.docker.operators.docker.DockerOperator` `airflow.providers.cncf.kubernetes.operators.pod.KubernetesPodOperator` Those require that Airflow has access to a Docker engine or Kubernetes cluster.

Similarly as in case of Python operators, the taskflow decorators are handy for you if you would like to use those operators to execute your callable Python code.

However, it is far more involved - you need to understand how Docker/Kubernetes Pods work if you want to use this approach, but the tasks are fully isolated from each other and you are not even limited to running Python code. You can write your tasks in any Programming language you want. Also your dependencies are fully independent from Airflow ones (including the system level dependencies) so if your task require a very different environment, this is the way to go.

Added in version 2.2.

As of version 2.2 of Airflow you can use `@task.docker` decorator to run your functions with `DockerOperator`.

Added in version 2.4.

As of version 2.2 of Airflow you can use `@task.kubernetes` decorator to run your functions with `KubernetesPodOperator`.

The benefits of using those operators are:

- You can run tasks with different sets of both Python and system level dependencies, or even tasks written in completely different language or even different processor architecture (x86 vs. arm).
- The environment used to run the tasks enjoys the optimizations and immutability of containers, where a similar set of dependencies can effectively reuse a number of cached layers of the image, so the environment is optimized for the case where you have multiple similar, but different environments.
- The dependencies can be pre-vetted by the admins and your security team, no unexpected, new code will be added dynamically. This is good for both, security and stability.
- Complete isolation between tasks. They cannot influence one another in other ways than using standard Airflow XCom mechanisms.

The drawbacks:

- There is an overhead to start the tasks. Usually not as big as when creating virtual environments dynamically, but still significant (especially for the `KubernetesPodOperator`).
- In case of TaskFlow decorators, the whole method to call needs to be serialized and sent over to the Docker Container or Kubernetes Pod, and there are system-level limitations on how big the method can be. Serializing, sending, and finally deserializing the method on remote end also adds an overhead.
- There is a resources overhead coming from multiple processes needed. Running tasks in case of those two operators requires at least two processes - one process (running in Docker Container or Kubernetes Pod) executing the task, and a supervising process in the Airflow worker that submits the job to Docker/Kubernetes and monitors the execution.
- Your environment needs to have the container images ready upfront. This usually means that you cannot change them on the fly. Adding system dependencies, modifying or changing Python requirements requires an image rebuilding and publishing (usually in your private registry). Iteration time when you work on new dependencies are usually longer and require the developer who is iterating to build and use their own images during iterations if they change dependencies. An appropriate deployment pipeline here is essential to be able to reliably maintain your deployment.
- Your python callable has to be serializable if you want to run it via decorators, also in this case all dependencies that are not available in Airflow environment must be locally imported in the callable you use and the top-level Python code of your DAG should not import/use those libraries.
- You need to understand more details about how Docker Containers or Kubernetes work. The abstraction provided by those two are “leaky”, so you need to understand a bit more about resources, networking, containers etc. in order to author a DAG that uses those operators.

You can see detailed examples of using `airflow.operators.providers.Docker` in [*Taskflow Docker example*](#) and `airflow.providers.cncf.kubernetes.operators.pod.KubernetesPodOperator` [*Taskflow Kubernetes example*](#)

Using multiple Docker Images and Celery Queues

There is a possibility (though it requires a deep knowledge of Airflow deployment) to run Airflow tasks using multiple, independent Docker images. This can be achieved via allocating different tasks to different Queues and configuring your Celery workers to use different images for different Queues. This, however, (at least currently) requires a lot of manual deployment configuration and intrinsic knowledge of how Airflow, Celery and Kubernetes works. Also it introduces quite some overhead for running the tasks - there are less chances for resource reuse and it's much more difficult to fine-tune such a deployment for cost of resources without impacting the performance and stability.

One of the possible ways to make it more useful is [AIP-46 Runtime isolation for Airflow tasks and DAG parsing](#). and completion of [AIP-43 DAG Processor Separation](#) Until those are implemented, there are very few benefits of using this approach and it is not recommended.

When those AIPs are implemented, however, this will open up the possibility of a more multi-tenant approach, where multiple teams will be able to have completely isolated sets of dependencies that will be used across the full lifecycle of a DAG - from parsing to execution.

3.13 FAQ

3.13.1 Scheduling / DAG file parsing

Why is task not 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 dags show foo_dag_id` and confirm that your task shows up in the graphviz format 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? For time-based dags, the task won’t be triggered until the the first schedule interval following the start date has passed.
- Is your `schedule` argument set properly? The default is one day (`datetime.timedelta(1)`). You must specify a different `schedule` directly to the DAG object you instantiate, not as a `default_param`, as task instances do not override their parent DAG’s `schedule`.
- 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 or been skipped (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 or been skipped. 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 about the *Scheduler* and make sure you fully understand how the scheduler cycle.

How to improve DAG performance?

There are some Airflow configuration to allow for a larger scheduling capacity and frequency:

- `parallelism`
- `max_active_tasks_per_dag`
- `max_active_runs_per_dag`

Dags have configurations that improves efficiency:

- `max_active_tasks`: Overrides `max_active_tasks_per_dag`.
- `max_active_runs`: Overrides `max_active_runs_per_dag`.

Operators or tasks also have configurations that improves efficiency and scheduling priority:

- `max_active_tis_per_dag`: This parameter controls the number of concurrent running task instances across `dag_runs` per task.
- `pool`: See *Pools*.
- `priority_weight`: See *Priority Weights*.
- `queue`: See `apache-airflow-providers-celery:celery_executor:queue` for CeleryExecutor deployments only.

How to reduce DAG scheduling latency / task delay?

Airflow 2.0 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*.

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

You can achieve this with *Trigger Rules*.

How to control DAG file parsing timeout for different DAG files?

(only valid for Airflow >= 2.3.0)

You can add a `get_dagbag_import_timeout` function in your `airflow_local_settings.py` which gets called right before a DAG file is parsed. You can return different timeout value based on the DAG file. When the return value is less than or equal to 0, it means no timeout during the DAG parsing.

Listing 1: `airflow_local_settings.py`

```
def get_dagbag_import_timeout(dag_file_path: str) -> Union[int, float]:  
    """  
    This setting allows to dynamically control the DAG file parsing timeout.  
  
    It is useful when there are a few DAG files requiring longer parsing times, while  
    (continues on next page)
```

(continued from previous page)

→ others do not.
You can control them separately instead of having one value for all DAG files.

If the return value is less than or equal to 0, it means no timeout during the DAG parsing.

```
"""
if "slow" in dag_file_path:
    return 90
if "no-timeout" in dag_file_path:
    return 0
return conf.getfloat("core", "DAGBAG_IMPORT_TIMEOUT")
```

See *Configuring local settings* for details on how to configure local settings.

When there are a lot (>1000) of DAG files, how to speed up parsing of new files?

Change the `file_parsing_sort_mode` to `modified_time`, raise the `min_file_process_interval` to 600 (10 minutes), 6000 (100 minutes) or a higher value.

The DAG parser will skip the `min_file_process_interval` check if a file is recently modified.

This might not work for case where the DAG is imported/created from a separate file. Example: `dag_file.py` that imports `dag_loader.py` where the actual logic of the DAG file is as shown below. In this case if `dag_loader.py` is updated but `dag_file.py` is not updated, the changes won't be reflected until `min_file_process_interval` is reached since DAG Parser will look for modified time for `dag_file.py` file.

Listing 2: `dag_file.py`

```
from dag_loader import create_dag

globals()[dag.dag_id] = create_dag(dag_id, schedule, dag_number, default_args)
```

Listing 3: `dag_loader.py`

```
from airflow.sdk import DAG
from airflow.sdk import task

import pendulum

def create_dag(dag_id, schedule, dag_number, default_args):
    dag = DAG(
        dag_id,
        schedule=schedule,
        default_args=default_args,
        pendulum.datetime(2021, 9, 13, tz="UTC"),
    )

    with dag:

        @task()
        def hello_world():
            print("Hello World")
            print(f"This is DAG: {dag_number}")
```

(continues on next page)

(continued from previous page)

```
hello_world()  
  
return dag
```

3.13.2 DAG construction

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. This can be done by declaring your `start_date` directly in the `DAG()` object. A DAG's first DagRun will be created based on the first complete `data_interval` after `start_date`. For example, for a DAG with `start_date=datetime(2024, 1, 1)` and `schedule="0 0 3 * *"`, the first DAG run will be triggered at midnight on 2024-02-03 with `data_interval_start=datetime(2024, 1, 3)` and `data_interval_end=datetime(2024, 2, 3)`. From that point on, the scheduler creates new DagRuns based on your `schedule` 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 DAG's `schedule`. 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`, 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` 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` is ignored in backfills.

Using time zones

Creating a time zone aware `datetime` (e.g. DAG's `start_date`) is quite simple. Just make sure to supply a time zone aware dates using `pendulum`. Don't try to use standard library `timezone` as they are known to have limitations and we deliberately disallow using them in dags.

What does execution_date mean?

`Execution date` or `execution_date` is a historical name for what is called a *logical date*, and also usually the start of the data interval represented by a DAG run.

Airflow was developed as a solution for ETL needs. In the ETL world, you typically summarize data. So, if you want to summarize data for `2016-02-19`, you would do it at `2016-02-20` midnight UTC, which would be right after all data for `2016-02-19` becomes available. This interval between midnights of `2016-02-19` and `2016-02-20` is called the *data interval*, and since it represents data in the date of `2016-02-19`, this date is also called the run's *logical date*, or the date that this DAG run is executed for, thus *execution date*.

For backward compatibility, a datetime value `execution_date` is still as *Template variables* with various formats in Jinja templated fields, and in Airflow's Python API. It is also included in the context dictionary given to an Operator's `execute` function.

```
class MyOperator(BaseOperator):
    def execute(self, context):
        logging.info(context["execution_date"])
```

However, you should always use `data_interval_start` or `data_interval_end` if possible, since those names are semantically more correct and less prone to misunderstandings.

Note that `ds` (the YYYY-MM-DD form of `data_interval_start`) refers to *date *string**, not *date *start** as may be confusing to some.

Tip

For more information on `logical_date`, see *Data Interval* and *Running dags*.

How to 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)
```

Even though Airflow supports multiple DAG definition per python file, dynamically generated or otherwise, it is not recommended as Airflow would like better isolation between dags from a fault and deployment perspective and multiple dags in the same file goes against that.

Are top level Python code allowed?

While it is not recommended to write any code outside of defining Airflow constructs, Airflow does support any arbitrary python code as long as it does not break the DAG file processor or prolong file processing time past the `dagbag_import_timeout` value.

A common example is the violation of the time limit when building a dynamic DAG which usually requires querying data from another service like a database. At the same time, the requested service is being swamped with DAG file processors requests for data to process the file. These unintended interactions may cause the service to deteriorate and eventually cause DAG file processing to fail.

Refer to *DAG writing best practices* for more information.

Do Macros resolves in another Ninja template?

It is not possible to render Macros or any Ninja template within another Ninja template. This is commonly attempted in `user_defined_macros`.

```
dag = DAG(  
    # ...  
    user_defined_macros={"my_custom_macro": "day={{ ds }}"}  
)  
  
bo = BashOperator(task_id="my_task", bash_command="echo {{ my_custom_macro }}", dag=dag)
```

This will echo “day={{ ds }}” instead of “day=2020-01-01” for a DAG run with a `data_interval_start` of 2020-01-01 00:00:00.

```
bo = BashOperator(task_id="my_task", bash_command="echo day={{ ds }}", dag=dag)
```

By using the `ds` macros directly in the `template_field`, the rendered value results in “day=2020-01-01”.

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 `logical_date` and the DAG’s schedule (if applicable). If you set `schedule` 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`.

3.13.3 Task execution interactions

What does `TemplateNotFound` mean?

`TemplateNotFound` errors are usually due to misalignment with user expectations when passing path to operator that trigger Ninja templating. A common occurrence is with `BashOperator`.

Another commonly missed fact is that the files are resolved relative to where the pipeline file lives. You can add other directories to the `template_searchpath` of the DAG object to allow for other non-relative location.

How to trigger tasks based on another task’s failure?

For tasks that are related through dependency, you can set the `trigger_rule` to `TriggerRule.ALL_FAILED` if the task execution depends on the failure of ALL its upstream tasks or `TriggerRule.ONE_FAILED` for just one of the upstream task.

```
import pendulum  
  
from airflow.sdk import dag, task  
from airflow.exceptions import AirflowException  
from airflow.utils.trigger_rule import TriggerRule  
  
  
@task()  
def a_func():  
    raise AirflowException  
  
  
@task()
```

(continues on next page)

(continued from previous page)

```

        trigger_rule=TriggerRule.ALL_FAILED,
)
def b_func():
    pass

@dag(schedule="@once", start_date=pendulum.datetime(2021, 1, 1, tz="UTC"))
def my_dag():
    a = a_func()
    b = b_func()

    a >> b

dag = my_dag()

```

See *Trigger Rules* for more information.

If the tasks are not related by dependency, you will need to *build a custom Operator*.

3.13.4 Airflow UI

Why did my task fail with no logs in the UI?

Logs are *typically served when a task reaches a terminal state*. Sometimes, a task's normal lifecycle is disrupted, and the task's worker is unable to write the task's logs. This typically happens for one of two reasons:

1. *Task Instance Heartbeat Timeout*.
2. Tasks failed after getting stuck in queued (Airflow 2.6.0+). Tasks that are in queued for longer than *scheduler.task_queued_timeout* will be marked as failed, and there will be no task logs in the Airflow UI.

Setting retries for each task drastically reduces the chance that either of these problems impact a workflow.

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

Set the value of `[fab] 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.

3.13.5 MySQL and MySQL variant Databases

What does “MySQL Server has gone away” mean?

You may occasionally experience `OperationalError` with the message “MySQL Server has gone away”. This is due to the connection pool keeping connections open too long and you are given an old connection that has expired. To ensure a valid connection, you can set `sqlalchemy_pool_recycle` to ensure connections are invalidated after that many seconds and new ones are created.

Does Airflow support extended ASCII or unicode characters?

If you intend to use extended ASCII or Unicode characters in Airflow, you have to provide a proper connection string to the MySQL database since they define charset explicitly.

```
sql_alchemy_conn = mysql://airflow@localhost:3306/airflow?charset=utf8
```

You will experience `UnicodeDecodeError` thrown by WTForms templating and other Airflow modules like below.

```
'ascii' codec can't decode byte 0xae in position 506: ordinal not in range(128)
```

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.

3.14 Troubleshooting

3.14.1 Obscure task failures

Task state changed externally

There are many potential causes for a task's state to be changed by a component other than the executor, which might cause some confusion when reviewing task instance or scheduler logs.

Below are some example scenarios that could cause a task's state to change by a component other than the executor:

- If a task's DAG failed to parse on the worker, the scheduler may mark the task as failed. If confirmed, consider increasing `core.dagbag_import_timeout` and `dag_processor.dag_file_processor_timeout`.
- The scheduler will mark a task as failed if the task has been queued for longer than `scheduler.task_queued_timeout`.
- If a *task instance's heartbeat times out*, it will be marked failed by the scheduler.
- A user marked the task as successful or failed in the Airflow UI.
- An external script or process used the *Airflow REST API* to change the state of a task.

TaskRunner killed

Sometimes, Airflow or some adjacent system will kill a task instance's TaskRunner, causing the task instance to fail.

Here are some examples that could cause such an event:

- A DAG run timeout, specified by `dagrun_timeout` in the DAG's definition.
- An Airflow worker running out of memory - Usually, Airflow workers that run out of memory receive a SIGKILL, and the scheduler will fail the corresponding task instance for not having a heartbeat. However, in some scenarios, Airflow kills the task before that happens.

3.15 Airflow's release process and version policy

Since Airflow 2.0.0 and providers 1.0.0 we aim to follow SemVer, meaning the release numbering works as follows:

- Versions are numbered in the form X.Y.Z.
- X is the major version number.
- Y is the minor version number, also called the *feature release* version number.
- Z is the patch number, which is incremented for bugfix and security releases. Before every new release, we'll make a release candidate available, and often alpha or beta release too. These are of the form X.Y.Z alpha/beta/rc N, which means the Nth alpha/beta/release candidate of version X.Y.Z

In git, each minor version will have its own branch, called vX-Y-stable where bugfix/security releases will be issued from. Commits and PRs should not normally go direct to these branches, but instead should target the main branch and then be cherry-picked by the release managers to these release branches. As a general rule of thumb, changes that will be included in a bugfix/security release will be associated with the corresponding github milestone in the PR but this is currently a manual process and deviations may occur. In all cases, the release managers reserve the right to postpone a PR to a later release if they deem it prudent. Additionally, no new features will be added to a patch release and minor releases are currently targeted at a roughly 2-3 month cadence.

Each Airflow release will also have a tag in git indicating its version number, signed with the release manager's key. Tags for the main Airflow release have the form X.Y.Z (no leading v) and providers are tagged with the form providers-<name>/X.Y.Z.

Although Airflow follows SemVer this is not a promise of 100% compatibility between minor or patch releases, simply because this is not possible: what is a bug to one person might be a feature another person is depending on.

Knowing the *intentions* of a maintainer can be valuable – especially *when* things break. Because that's all SemVer is: **a TL;DR of the changelog**.

—Hynek Schlawack <https://hynek.me/articles/semver-will-not-save-you/>

That is all SemVer is – it's a statement of our intent as package authors, and a clear statement of our goals.

Major release

Major releases (X.0.0, X+1.0.0 etc.) indicate a backwards-incompatible change.

These releases do not happen with any regular interval or on any predictable schedule.

Each time a new major version is released previously deprecated features will be removed.

Feature releases

Feature releases (X.Y.0, X.Y+1.0, etc.) will happen roughly every two or three months – see release process for details. These releases will contain new features, improvements to existing features, and such.

Patch releases

Patch releases (X.Y.Z, X.Y.Z+1, etc.) will happen, on an as-needed basis, as issues are reported and fixed.

These releases will be 100% compatible with the associated feature release. So the answer to “should I upgrade to the latest patch release?” will always be “yes.”

The only exception to the above with respect to 100% backwards compatibility is when a security or data loss issue can't be fixed without breaking backwards-compatibility. If this happens, the release notes will provide detailed upgrade instructions. **No new features will be added in patch releases**

3.15.1 Deprecation policy

From time-to-time existing features will be deprecated, or modules will be renamed.

When this happens, the existing code will continue to work but will issue a `DeprecationWarning` (or a subclass) when the code is executed. This code will continue to work for the rest of the current major version – if it works on 2.0.0, it will work for every 2.Y.Z release.

So, for example, if we decided to start the deprecation of a function in Airflow 2.2.4:

- Airflow 2.2 will contain a backwards-compatible replica of the function which will raise a `DeprecationWarning`
- Airflow 2.3 will continue to work and issue a warning
- Airflow 3.0 (the major version that follows 2.2) will remove the feature outright

The exception to this deprecation policy is any feature which is marked as “experimental”, which *may* suffer breaking changes or complete removal in a Feature release.

3.15.2 Experimental features

From time-to-time a new feature will be added to Airflow that will be marked as experimental.

Experimental features do not have any guarantees about deprecation, and *can* be changed in a breaking way between feature releases, or even removed entirely.

We always aim to maintain compatibility even for experimental features, but make no promises. By having this “get out” it lets us build new features more quickly and get them in the hand of users without having to worry about making a feature perfect.

3.16 Release Notes

Apache Airflow Releases

- *Airflow 3.0.0 (2025-04-22)*
- *Airflow 2.10.5 (2025-02-10)*
- *Airflow 2.10.4 (2024-12-16)*
- *Airflow 2.10.3 (2024-11-05)*
- *Airflow 2.10.2 (2024-09-18)*
- *Airflow 2.10.1 (2024-09-05)*
- *Airflow 2.10.0 (2024-08-15)*
- *Airflow 2.9.3 (2024-07-15)*
- *Airflow 2.9.2 (2024-06-10)*
- *Airflow 2.9.1 (2024-05-03)*
- *Airflow 2.9.0 (2024-04-08)*
- *Airflow 2.8.4 (2024-03-25)*

- *Airflow 2.8.3 (2024-03-11)*
- *Airflow 2.8.2 (2024-02-26)*
- *Airflow 2.8.1 (2024-01-19)*
- *Airflow 2.8.0 (2023-12-18)*
- *Airflow 2.7.3 (2023-11-06)*
- *Airflow 2.7.2 (2023-10-12)*
- *Airflow 2.7.1 (2023-09-07)*
- *Airflow 2.7.0 (2023-08-18)*
- *Airflow 2.6.3 (2023-07-10)*
- *Airflow 2.6.2 (2023-06-17)*
- *Airflow 2.6.1 (2023-05-16)*
- *Airflow 2.6.0 (2023-04-30)*
- *Airflow 2.5.3 (2023-04-01)*
- *Airflow 2.5.2 (2023-03-15)*
- *Airflow 2.5.1 (2023-01-20)*
- *Airflow 2.5.0 (2022-12-02)*
- *Airflow 2.4.3 (2022-11-14)*
- *Airflow 2.4.2 (2022-10-23)*
- *Airflow 2.4.1 (2022-09-30)*
- *Airflow 2.4.0 (2022-09-19)*
- *Airflow 2.3.4 (2022-08-23)*
- *Airflow 2.3.3 (2022-07-09)*
- *Airflow 2.3.2 (2022-06-04)*
- *Airflow 2.3.1 (2022-05-25)*
- *Airflow 2.3.0 (2022-04-30)*
- *Airflow 2.2.5, (2022-04-04)*
- *Airflow 2.2.4, (2022-02-22)*
- *Airflow 2.2.3, (2021-12-21)*
- *Airflow 2.2.2 (2021-11-15)*
- *Airflow 2.2.1 (2021-10-29)*
- *Airflow 2.2.0 (2021-10-11)*
- *Airflow 2.1.4 (2021-09-18)*
- *Airflow 2.1.3 (2021-08-23)*
- *Airflow 2.1.2 (2021-07-14)*
- *Airflow 2.1.1 (2021-07-02)*

- *Airflow 2.1.0 (2021-05-21)*
- *Airflow 2.0.2 (2021-04-19)*
- *Airflow 2.0.1 (2021-02-08)*
- *Airflow 2.0.0 (2020-12-18)*

3.16.1 Airflow 3.0.0 (2025-04-22)

We are proud to announce the General Availability of Apache Airflow 3.0—the most significant release in the project’s history. This version introduces a service-oriented architecture, a stable DAG authoring interface, expanded support for event-driven and ML workflows, and a fully modernized UI built on React. Airflow 3.0 reflects years of community investment and lays the foundation for the next era of scalable, modular orchestration.

Highlights

- **Service-Oriented Architecture:** A new Task Execution API and `airflow api-server` enable task execution in remote environments with improved isolation and flexibility (AIP-72).
- **Edge Executor:** A new executor that supports distributed, event-driven, and edge-compute workflows (AIP-69), now generally available.
- **Stable Authoring Interface:** DAG authors should now use the new `airflow.sdk` namespace to import core DAG constructs like `@dag`, `@task`, and `DAG`.
- **Scheduler-Managed Backfills:** Backfills are now scheduled and tracked like regular DAG runs, with native UI and API support (AIP-78).
- **DAG Versioning:** Airflow now tracks structural changes to DAGs over time, enabling inspection of historical DAG definitions via the UI and API (AIP-66).
- **Asset-Based Scheduling:** The dataset model has been renamed and redesigned as assets, with a new `@asset` decorator and cleaner event-driven DAG definition (AIP-74, AIP-75).
- **Support for ML and AI Workflows:** DAGs can now run with `logical_date=None`, enabling use cases such as model inference, hyperparameter tuning, and non-interval workflows (AIP-83).
- **Removal of Legacy Features:** SLAs, SubDAGs, DAG and Xcom pickling, and several internal context variables have been removed. Use the upgrade tools to detect deprecated usage.
- **Split CLI and API Changes:** The CLI has been split into `airflow` and `airflowctl` (AIP-81), and REST API now defaults to `logical_date=None` when triggering a new DAG run.
- **Modern React UI:** A complete UI overhaul built on React and FastAPI includes version-aware views, backfill management, and improved DAG and task introspection (AIP-38, AIP-84).
- **Migration Tooling:** Use `ruff` and `airflow config update` to validate DAGs and configurations. Upgrade requires Airflow 2.7 or later and Python 3.9–3.12.

Significant Changes

Airflow 3.0 introduces the most significant set of changes since the 2.0 release, including architectural shifts, new execution models, and improvements to DAG authoring and scheduling.

Task Execution API & Task SDK (AIP-72)

Airflow now supports a service-oriented architecture, enabling tasks to be executed remotely via a new Task Execution API. This API decouples task execution from the scheduler and introduces a stable contract for running tasks outside of Airflow's traditional runtime environment.

To support this, Airflow introduces the Task SDK — a lightweight runtime environment for running Airflow tasks in external systems such as containers, edge environments, or other runtimes. This lays the groundwork for language-agnostic task execution and brings improved isolation, portability, and extensibility to Airflow-based workflows.

Airflow 3.0 also introduces a new `airflow.sdk` namespace that exposes the core authoring interfaces for defining DAGs and tasks. DAG authors should now import objects like `DAG`, `@dag`, and `@task` from `airflow.sdk` rather than internal modules. This new namespace provides a stable, forward-compatible interface for DAG authoring across future versions of Airflow.

Edge Executor (AIP-69)

Airflow 3.0 introduces the **Edge Executor** as a generally available feature, enabling execution of tasks in distributed or remote compute environments. Designed for event-driven and edge-compute use cases, the Edge Executor integrates with the Task Execution API to support task orchestration beyond the traditional Airflow runtime. This advancement facilitates hybrid and cross-environment orchestration patterns, allowing task workers to operate closer to data or application layers.

Scheduler-Managed Backfills (AIP-78)

Backfills are now fully managed by the scheduler, rather than being launched as separate command-line jobs. This change unifies backfill logic with regular DAG execution and ensures that backfill runs follow the same scheduling, versioning, and observability models as other DAG runs.

Airflow 3.0 also introduces native UI and REST API support for initiating and monitoring backfills, making them more accessible and easier to integrate into automated workflows. These improvements lay the foundation for smarter, safer historical reprocessing — now available directly through the Airflow UI and API.

DAG Versioning (AIP-66)

Airflow 3.0 introduces native DAG versioning. DAG structure changes (e.g., renamed tasks, dependency shifts) are now tracked directly in the metadata database. This allows users to inspect historical DAG structures through the UI and API, and lays the foundation for safer backfills, improved observability, and runtime-determined DAG logic.

React UI Rewrite (AIP-38, AIP-84)

Airflow 3.0 ships with a completely redesigned user interface built on React and FastAPI. This modern architecture improves responsiveness, enables more consistent navigation across views, and unlocks new UI capabilities — including support for DAG versioning, asset-centric DAG definitions, and more intuitive filtering and search.

The new UI replaces the legacy Flask-based frontend and introduces a foundation for future extensibility and community contributions.

Asset-Based Scheduling & Terminology Alignment (AIP-74, AIP-75)

The concept of **Datasets** has been renamed to **Assets**, unifying terminology with common practices in the modern data ecosystem. The internal model has also been reworked to better support future features like asset partitions and validations.

The `@asset` decorator and related changes to the DAG parser enable clearer, asset-centric DAG definitions, allowing Airflow to more naturally support event-driven and data-aware scheduling patterns.

This renaming impacts modules, classes, functions, configuration keys, and internal models. Key changes include:

- `Dataset` → `Asset`
- `DatasetEvent` → `AssetEvent`
- `DatasetAlias` → `AssetAlias`
- `airflow.datasets.*` → `airflow.sdk.*`
- `airflow.timetables.simple.DatasetTriggeredTimetable` → `airflow.timetables.simple.AssetTriggeredTimetable`
- `airflow.timetables.datasets.DatasetOrTimeSchedule` → `airflow.timetables.assets.AssetOrTimeSchedule`
- `airflow.listeners.spec.dataset.on_dataset_created` → `airflow.listeners.spec.asset.on_asset_created`
- `airflow.listeners.spec.dataset.on_dataset_changed` → `airflow.listeners.spec.asset.on_asset_changed`
- `core.dataset_manager_class` → `core.asset_manager_class`
- `core.dataset_manager_kwargs` → `core.asset_manager_kwargs`

Unified Scheduling Field

Airflow 3.0 removes the legacy `schedule_interval` and `timetable` parameters. DAGs must now use the unified `schedule` field for all time- and event-based scheduling logic. This simplifies DAG definition and improves consistency across scheduling paradigms.

Updated Scheduling Defaults

Airflow 3.0 changes the default behavior for new DAGs by setting `catchup_by_default = False` in the configuration file. This means DAGs that do not explicitly set `catchup=...` will no longer backfill missed intervals by default. This change reduces confusion for new users and better reflects the growing use of on-demand and event-driven workflows.

The default DAG schedule has been changed to `None` from `@once`.

Restricted Metadata Database Access

Task code can no longer directly access the metadata database. Interactions with DAG state, task history, or DAG runs must be performed via the Airflow REST API or exposed context. This change improves architectural separation and enables remote execution.

Future Logical Dates No Longer Supported

Airflow no longer supports triggering DAG runs with a logical date in the future. This change aligns with the logical execution model and removes ambiguity in backfills and event-driven DAGs. Use `logical_date=None` to trigger runs with the current timestamp.

Context Behavior for Asset and Manually Triggered DAGs

For DAG runs triggered by an Asset event or through the REST API without specifying a `logical_date`, Airflow now sets `logical_date=None` by default. These DAG runs do not have a data interval, and attempting to access `data_interval_start`, `data_interval_end`, or `logical_date` from the task context will raise a `KeyError`.

DAG authors should use `dag_run.logical_date` and perform appropriate checks or fallbacks if supporting multiple trigger types. This change improves consistency with event-driven semantics but may require updates to existing DAGs that assume these values are always present.

Improved Callback Behavior

Airflow 3.0 refines task callback behavior to improve clarity and consistency. In particular, `on_success_callback` is no longer executed when a task is marked as `SKIPPED`, aligning it more closely with expected semantics.

Updated Default Configuration

Several default configuration values have been updated in Airflow 3.0 to better reflect modern usage patterns and simplify onboarding:

- `catchup_by_default` is now set to `False` by default. DAGs will not automatically backfill unless explicitly configured to do so.
- `create_cron_data_intervals` is now set to `False` by default. As a result, cron expressions will be interpreted using the `CronTriggerTimetable` instead of the legacy `CronDataIntervalTimetable`.
- `SimpleAuthManager` is now the default `auth_manager`. To continue using Flask AppBuilder-based authentication, install the `apache-airflow-providers-flask-appbuilder` provider and explicitly set `auth_manager = airflow.providers.fab.auth_manager.FabAuthManager`.

These changes represent the most significant evolution of the Airflow platform since the release of 2.0 — setting the stage for more scalable, event-driven, and language-agnostic orchestration in the years ahead.

Executor & Scheduler Updates

Airflow 3.0 introduces several important improvements and behavior changes in how DAGs and tasks are scheduled, prioritized, and executed.

Standalone DAG Processor Required

Airflow 3.0 now requires the standalone DAG processor to parse DAGs. This dedicated process improves scheduler performance, isolation, and observability. It also simplifies architecture by clearly separating DAG parsing from scheduling logic. This change may affect custom deployments that previously used embedded DAG parsing.

Priority Weight Capped by Pool Slots

The `priority_weight` value on a task is now capped by the number of available pool slots. This ensures that resource availability remains the primary constraint in task execution order, preventing high-priority tasks from starving others when resource contention exists.

Teardown Task Handling During DAG Termination

Teardown tasks will now be executed even when a DAG run is terminated early. This ensures that cleanup logic is respected, improving reliability for workflows that use teardown tasks to manage ephemeral infrastructure, temporary files, or downstream notifications.

Improved Scheduler Fault Tolerance

Scheduler components now use `run_with_db_retries` to handle transient database issues more gracefully. This enhances Airflow's fault tolerance in high-volume environments and reduces the likelihood of scheduler restarts due to temporary database connection problems.

Mapped Task Stats Accuracy

Airflow 3.0 fixes a bug that caused incorrect task statistics to be reported for dynamic task mapping. Stats now accurately reflect the number of mapped task instances and their statuses, improving observability and debugging for dynamic workflows.

SequentialExecutor has been removed

SequentialExecutor was primarily used for local testing but is now redundant, as LocalExecutor supports SQLite with WAL mode and provides better performance with parallel execution. Users should switch to LocalExecutor or CeleryExecutor as alternatives.

DAG Authoring Enhancements

Airflow 3.0 includes several changes that improve consistency, clarity, and long-term stability for DAG authors.

New Stable DAG Authoring Interface: `airflow.sdk`

Airflow 3.0 introduces a new, stable public API for DAG authoring under the `airflow.sdk` namespace, available via the `apache-airflow-task-sdk` package.

The goal of this change is to **decouple DAG authoring from Airflow internals** (Scheduler, API Server, etc.), providing a **forward-compatible, stable interface** for writing and maintaining DAGs across Airflow versions.

DAG authors should now import core constructs from `airflow.sdk` rather than internal modules.

Key Imports from `airflow.sdk`:

- Classes:
 - Asset
 - BaseNotifier
 - BaseOperator
 - BaseOperatorLink
 - BaseSensorOperator
 - Connection
 - Context
 - DAG
 - EdgeModifier
 - Label
 - ObjectStoragePath
 - Param
 - TaskGroup
 - Variable
- Decorators and Functions:
 - `@asset`
 - `@dag`
 - `@setup`

- `@task`
- `@task_group`
- `@teardown`
- `chain`
- `chain_linear`
- `cross_downstream`
- `get_current_context`
- `get_parsing_context`

For an exhaustive list of available classes, decorators, and functions, check `airflow.sdk.__all__`.

All DAGs should update imports to use `airflow.sdk` instead of referencing internal Airflow modules directly. Legacy import paths (e.g., `airflow.models.dag.DAG`, `airflow.decorator.task`) are **deprecated** and will be **removed** in a future Airflow version. Some additional utilities and helper functions that DAGs sometimes use from `airflow.utils.*` and others will be progressively migrated to the Task SDK in future minor releases.

These future changes aim to **complete the decoupling** of DAG authoring constructs from internal Airflow services. DAG authors should expect continued improvements to `airflow.sdk` with no backwards-incompatible changes to existing constructs.

For example, update:

```
# Old (Airflow 2.x)
from airflow.models import DAG
from airflow.decorators import task

# New (Airflow 3.x)
from airflow.sdk import DAG, task
```

Renamed Parameter: `fail_stop` → `fail_fast`

The DAG argument `fail_stop` has been renamed to `fail_fast` for improved clarity. This parameter controls whether a DAG run should immediately stop execution when a task fails. DAG authors should update any code referencing `fail_stop` to use the new name.

Context Cleanup and Parameter Removal

Several legacy context variables have been removed or may no longer be available in certain types of DAG runs, including:

- `conf`
- `execution_date`
- `dag_run.external_trigger`

In asset-triggered and manually triggered DAG runs with `logical_date=None`, data interval fields such as `data_interval_start` and `data_interval_end` may not be present in the task context. DAG authors should use explicit references such as `dag_run.logical_date` and conditionally check for the presence of interval-related fields where applicable.

Task Context Utilities Moved

Internal task context functions such as `get_parsing_context` have been moved to a more appropriate location (e.g., `airflow.models.taskcontext`). DAG authors using these utilities directly should update import paths accordingly.

Trigger Rule Restrictions

The `TriggerRule.ALWAYS` rule can no longer be used with teardown tasks or tasks that are expected to honor upstream dependency semantics. DAG authors should ensure that teardown logic is defined with the appropriate trigger rules for consistent task resolution behavior.

Asset Aliases for Reusability

A new utility function, `create_asset_aliases()`, allows DAG authors to define reusable aliases for frequently referenced Assets. This improves modularity and reuse across DAG files and is particularly helpful for teams adopting asset-centric DAGs.

Operator Links interface changed

The Operator Extra links, which can be defined either via plugins or custom operators now do not execute any user code in the Airflow UI, but instead push the “full” links to XCom backend and the link is fetched from the XCom backend when viewing task details, for example from grid view.

Example for users with custom links class:

```
@attr.s(auto_attribs=True)
class CustomBaseIndexOpLink(BaseOperatorLink):
    """Custom Operator Link for Google BigQuery Console."""

    index: int = attr.ib()

    @property
    def name(self) -> str:
        return f"BigQuery Console #{self.index + 1}"

    @property
    def xcom_key(self) -> str:
        return f"bigquery_{self.index + 1}"

    def get_link(self, operator, *, ti_key):
        search_query = XCom.get_one(
            task_id=ti_key.task_id, dag_id=ti_key.dag_id, run_id=ti_key.run_id, key=
            "search_query"
        )
        return f"https://console.cloud.google.com/bigquery?j={search_query}"
```

The link has an `xcom_key` defined, which is how it will be stored in the XCOM backend, with key as `xcom_key` and value as the entire link, this case: <https://console.cloud.google.com/bigquery?j=search>

Plugins no longer support adding executors, operators & hooks

Operator (including Sensors), Executors & Hooks can no longer be registered or imported via Airflow’s plugin mechanism. These types of classes are just treated as plain Python classes by Airflow, so there is no need to register them with Airflow. They can be imported directly from their respective provider packages.

Before:

```
from airflow.hooks.my_plugin import MyHook
```

You should instead import it as:

```
from my_plugin import MyHook
```

Support for ML & AI Use Cases (AIP-83)

Airflow 3.0 expands the types of DAGs that can be expressed by removing the constraint that each DAG run must correspond to a unique data interval. This change, introduced in AIP-83, enables support for workflows that don't operate on a fixed schedule — such as model training, hyperparameter tuning, and inference tasks.

These ML- and AI-oriented DAGs often run ad hoc, are triggered by external systems, or need to execute multiple times with different parameters over the same dataset. By allowing multiple DAG runs with `logical_date=None`, Airflow now supports these scenarios natively without requiring workarounds.

Config & Interface Changes

Airflow 3.0 introduces several configuration and interface updates that improve consistency, clarify ownership of core utilities, and remove legacy behaviors that were no longer aligned with modern usage patterns.

Default Value Handling

Airflow no longer silently updates configuration options that retain deprecated default values. Users are now required to explicitly set any config values that differ from the current defaults. This change improves transparency and prevents unintentional behavior changes during upgrades.

Refactored Config Defaults

Several configuration defaults have changed in Airflow 3.0 to better reflect modern usage patterns:

- The default value of `catchup_by_default` is now `False`. DAGs will not backfill missed intervals unless explicitly configured to do so.
- The default value of `create_cron_data_intervals` is now `False`. Cron expressions are now interpreted using the `CronTriggerTimetable` instead of the legacy `CronDataIntervalTimetable`. This change simplifies interval logic and aligns with the future direction of Airflow's scheduling system.

Refactored Internal Utilities

Several core components have been moved to more intuitive or stable locations:

- The `SecretsMasker` class has been relocated to `airflow.sdk.execution_time.secrets_masker`.
- The `ObjectStoragePath` utility previously located under `airflow.io` is now available via `airflow.sdk`.

These changes simplify imports and reflect broader efforts to stabilize utility interfaces across the Airflow codebase.

Improved `inlet_events`, `outlet_events`, and `triggering_asset_events`

Asset event mappings in the task context are improved to better support asset use cases, including new features introduced in AIP-74.

Events of an asset or asset alias are now accessed directly by a concrete object to avoid ambiguity. Using a `str` to access events is no longer supported. Use an `Asset` or `AssetAlias` object, or `Asset.ref` to refer to an entity explicitly instead, such as:

```
outlet_events[Asset.ref(name="myasset")] # Get events for asset named "myasset".  
outlet_events[AssetAlias(name="myalias")] # Get events for asset alias named "myalias".
```

Alternatively, two helpers `for_asset` and `for_asset_alias` are added as shortcuts:

```
outlet_events.for_asset(name="myasset") # Get events for asset named "myasset".  
outlet_events.for_asset_alias(name="myalias") # Get events for asset alias named  
↪ "myalias".
```

The internal representation of asset event triggers now also includes an explicit `uri` field, simplifying traceability and aligning with the broader asset-aware execution model introduced in Airflow 3.0. DAG authors interacting directly with `inlet_events` may need to update logic that assumes the previous structure.

Behaviour change in `xcom_pull`

In Airflow 2, the `xcom_pull()` method allowed pulling XComs by key without specifying `task_ids`, despite the fact that the underlying DB model defines `task_id` as part of the XCom primary key. This created ambiguity: if two tasks pushed XComs with the same key, `xcom_pull()` would pull whichever one happened to be first, leading to unpredictable behavior.

Airflow 3 resolves this inconsistency by requiring `task_ids` when pulling by key. This change aligns with the task-scoped nature of XComs as defined by the schema, ensuring predictable and consistent behavior.

DAG Authors should update their dags to use `task_ids` if their dags used `xcom_pull` without `task_ids` such as:

```
kwargs["ti"].xcom_pull(key="key")
```

Should be updated to:

```
kwargs["ti"].xcom_pull(task_ids="task1", key="key")
```

Removed Configuration Keys

As part of the deprecation cleanup, several legacy configuration options have been removed. These include:

- [scheduler] `allow_trigger_in_future`
- [scheduler] `use_job_schedule`
- [scheduler] `use_local_tz`
- [scheduler] `processor_poll_interval`
- [logging] `dag_processor_manager_log_location`
- [logging] `dag_processor_manager_log_stdout`
- [logging] `log_processor_filename_template`

All the webserver configurations have also been removed since API server now replaces webserver, so the configurations like below have no effect:

- [webserver] `allow_raw_html_descriptions`
- [webserver] `cookie_samesite`
- [webserver] `error_logfile`
- [webserver] `access_logformat`
- [webserver] `web_server_master_timeout`

- etc

Several configuration options previously located under the [webserver] section have been **moved to the new ``[api]`` section**. The following configuration keys have been moved:

- [webserver] web_server_host → [api] host
- [webserver] web_server_port → [api] port
- [webserver] workers → [api] workers
- [webserver] web_server_worker_timeout → [api] worker_timeout
- [webserver] web_server_ssl_cert → [api] ssl_cert
- [webserver] web_server_ssl_key → [api] ssl_key
- [webserver] access_logfile → [api] access_logfile

Users should review their airflow.cfg files or use the `airflow config lint` command to identify outdated or removed options.

Upgrade Tooling

Airflow 3.0 includes improved support for upgrade validation. Use the following tools to proactively catch incompatible configs or deprecated usage patterns:

- `airflow config lint`: Identifies removed or invalid config keys
- `ruff check --select AIR30 --preview`: Flags removed interfaces and common migration issues

CLI & API Changes

Airflow 3.0 introduces changes to both the CLI and REST API interfaces to better align with service-oriented deployments and event-driven workflows.

Split CLI Architecture (AIP-81)

The Airflow CLI has been split into two distinct interfaces:

- The core `airflow` CLI now handles only local functionality (e.g., `airflow tasks test`, `airflow dags list`).
- Remote functionality, including triggering DAGs or managing connections in service-mode environments, is now handled by a separate CLI called `airflowctl`, distributed via the `apache-airflow-client` package.

This change improves security and modularity for deployments that use Airflow in a distributed or API-first context.

REST API v2 replaces v1

The legacy REST API v1, previously built with Connexion and Marshmallow, has been replaced by a modern FastAPI-based REST API v2.

This new implementation improves performance, aligns more closely with web standards, and provides a consistent developer experience across the API and UI.

Key changes include stricter validation (422 errors instead of 400), the removal of the `execution_date` parameter in favor of `logical_date`, and more consistent query parameter handling.

The v2 API is now the stable, fully supported interface for programmatic access to Airflow, and also powers the new UI - achieving full feature parity between the UI and API.

For details, see the *Airflow REST API v2* documentation.

REST API: DAG Trigger Behavior Updated

The behavior of the POST `/dags/{dag_id}/dagRuns` endpoint has changed. If a `logical_date` is not explicitly provided when triggering a DAG via the REST API, it now defaults to `None`.

This aligns with event-driven DAGs and manual runs in Airflow 3.0, but may break backward compatibility with scripts or tools that previously relied on Airflow auto-generating a timestamped `logical_date`.

Removed CLI Flags and Commands

Several deprecated CLI arguments and commands that were marked for removal in earlier versions have now been cleaned up in Airflow 3.0. Run `airflow --help` to review the current set of available commands and arguments.

- Deprecated `--ignore-depends-on-past` cli option is replaced by `--depends-on-past ignore`.
- `--tree` flag for `airflow tasks list` command is removed. The format of the output with that flag can be expensive to generate and extremely large, depending on the DAG. `airflow dag show` is a better way to visualize the relationship of tasks in a DAG.
- Changing `dag_id` from flag (`-d, --dag-id`) to a positional argument in the `dags list-runs` CLI command.
- The `airflow db init` and `airflow db upgrade` commands have been removed. Use `airflow db migrate` instead to initialize or migrate the metadata database. If you would like to create default connections use `airflow connections create-default-connections`.
- `airflow api-server` has replaced `airflow webserver` cli command.

Provider Refactor & Standardization

Airflow 3.0 completes the migration of several core operators, sensors, and hooks into the new `apache-airflow-providers-standard` package. This package now includes commonly used components such as:

- `PythonOperator`
- `BashOperator`
- `EmailOperator`
- `SimpleHttpOperator`
- `ShortCircuitOperator`

These operators were previously bundled inside `airflow-core` but are now treated as provider-managed components to improve modularity, testability, and lifecycle independence.

This change enables more consistent versioning across providers and prepares Airflow for a future where all integrations — including “standard” ones — follow the same interface model.

To maintain compatibility with existing DAGs, the `apache-airflow-providers-standard` package is installable on both Airflow 2.x and 3.x. Users upgrading from Airflow 2.x are encouraged to begin updating import paths and testing provider installation in advance of the upgrade.

Legacy imports such as `airflow.operators.python.PythonOperator` are deprecated and will be removed soon. They should be replaced with:

```
from airflow.providers.standard.operators.python import PythonOperator
```

UI & Usability Improvements

Airflow 3.0 introduces a modernized user experience that complements the new React-based UI architecture (see Significant Changes). Several areas of the interface have been enhanced to improve visibility, consistency, and navigability.

New Home Page

The Airflow Home page now provides a high-level operational overview of your environment. It includes health checks for core components (Scheduler, Triggerer, DAG Processor), summary stats for DAG and task instance states, and a real-time feed of asset-triggered events. This view helps users quickly identify pipeline health, recent activity, and potential failures.

Unified DAG List View

The DAG List page has been refreshed with a cleaner layout and improved responsiveness. Users can browse DAGs by name, tags, or owners. While full-text search has not yet been integrated, filters and navigation have been refined for clarity in large deployments.

Version-Aware Graph and Grid Views

The Graph and Grid views now display task information in the context of the DAG version that was used at runtime. This improves traceability for DAGs that evolve over time and provides more accurate debugging of historical runs.

Expanded DAG Graph Visualization

The Graph view now supports visualizing the full chain of asset and task dependencies, including assets consumed or produced across DAG boundaries. This allows users to inspect upstream and downstream lineage in a unified view, making it easier to trace data flows, debug triggering behavior, and understand conditional dependencies between assets and tasks.

DAG Code View

The “Code” tab now displays the exact DAG source as parsed by the scheduler for the selected DAG version. This allows users to inspect the precise code that was executed, even for historical runs, and helps debug issues related to versioned DAG changes.

Improved Task Log Access

Task log access has been streamlined across views. Logs are now easier to access from both the Grid and Task Instance pages, with cleaner formatting and reduced visual noise.

Enhanced Asset and Backfill Views

New UI components support asset-centric DAGs and backfill workflows:

- Asset definitions are now visible from the DAG details page, allowing users to inspect upstream and downstream asset relationships.
- Backfills can be triggered and monitored directly from the UI, including support for scheduler-managed backfills introduced in Airflow 3.0.

These improvements make Airflow more accessible to operators, data engineers, and stakeholders working across both time-based and event-driven workflows.

Deprecations & Removals

A number of deprecated features, modules, and interfaces have been removed in Airflow 3.0, completing long-standing migrations and cleanups.

Users are encouraged to review the following removals to ensure compatibility:

- **SubDag support has been removed** entirely, including the `SubDagOperator`, related CLI and API interfaces. TaskGroups are now the recommended alternative for nested DAG structures.
- **SLAs have been removed**: The legacy SLA feature, including SLA callbacks and metrics, has been removed. A more flexible replacement mechanism, `DeadlineAlerts`, is planned for a future version of Airflow. Users who relied on SLA-based notifications should consider implementing custom alerting using task-level success/failure hooks or external monitoring integrations.
- **Pickling support has been removed**: All legacy features related to DAG pickling have been fully removed. This includes the `PickleDag` CLI/API, as well as implicit behaviors around `store_serialized_dags = False`. DAGs must now be serialized using the JSON-based serialization system. Ensure any custom Python objects used in DAGs are JSON-serializable.
- **Context parameter cleanup**: Several previously available context variables have been removed from the task execution context, including `conf`, `execution_date`, and `dag_run.external_trigger`. These values are either no longer applicable or have been renamed (e.g., use `dag_run.logical_date` instead of `execution_date`). DAG authors should ensure that templated fields and Python callables do not reference these deprecated keys.
- **Deprecated core imports** have been fully removed. Any use of `airflow.operators.*`, `airflow.hooks.*`, or similar legacy import paths should be updated to import from their respective providers.
- **Configuration cleanup**: Several legacy config options have been removed, including:
 - `scheduler.allow_trigger_in_future`: DAG runs can no longer be triggered with a future logical date. Use `logical_date=None` instead.
 - `scheduler.use_job_schedule` and `scheduler.use_local_tz` have also been removed. These options were deprecated and no longer had any effect.
- **Deprecated utility methods** such as those in `airflow.utils.helpers`, `airflow.utils.process_utils`, and `airflow.utils.timezone` have been removed. Equivalent functionality can now be found in the standard Python library or Airflow provider modules.
- **Removal of deprecated CLI flags and behavior**: Several CLI entrypoints and arguments that were marked for removal in earlier versions have been cleaned up.

To assist with the upgrade, tools like `ruff` (e.g., rule AIR302) and `airflow config lint` can help identify obsolete imports and configuration keys. These utilities are recommended for locating and resolving common incompatibilities during migration. Please see [Upgrade Guide](#) for more information.

Summary of Removed Features

The following table summarizes user-facing features removed in 3.0 and their recommended replacements. Not all of these are called out individually above.

| Feature | Replacement / Notes |
|--|--|
| SubDagOperator / SubDAGs | Use TaskGroups |
| SLA callbacks / metrics | Deadline Alerts (planned post-3.0) |
| DAG Pickling | Use JSON serialization; pickling is no longer supported |
| Xcom Pickling | Use custom Xcom backend; pickling is no longer supported |
| <code>execution_date</code> context var | Use <code>dag_run.logical_date</code> |
| <code>conf</code> and <code>dag_run.external_trigger</code> | Removed from context; use DAG params or <code>dag_run</code> APIs |
| Core EmailOperator | Use <code>EmailOperator</code> from the <code>smtp</code> provider |
| <code>none_failed_or_skipped</code> rule | Use <code>none_failed_min_one_success</code> |
| <code>dummy</code> trigger rule | Use <code>always</code> |
| <code>fail_stop</code> argument | Use <code>fail_fast</code> |
| <code>store_serialized_dags=False</code> | DAGs are always serialized; config has no effect |
| Deprecated core imports | Import from appropriate provider package |
| <code>SequentialExecutor</code> & <code>DebugExecutor</code> | Use <code>LocalExecutor</code> for testing |
| <code>.airflowignore</code> regex | Uses glob syntax by default |

Migration Tooling & Upgrade Process

Airflow 3 was designed with migration in mind. Many Airflow 2 DAGs will work without changes, especially if deprecation warnings were addressed in earlier releases. To support the upgrade, Airflow 3 includes validation tools such as `ruff` and `airflow config update`, as well as a simplified startup model.

For a step-by-step upgrade process, see the [Upgrade Guide](#).

Minimum Supported Versions

To upgrade to Airflow 3.0, you must be running **Airflow 2.7 or later**.

Airflow 3.0 supports the following Python versions:

- Python 3.9
- Python 3.10
- Python 3.11
- Python 3.12

Earlier versions of Airflow or Python are not supported due to architectural changes and updated dependency requirements.

DAG Compatibility Checks

Airflow now includes a Ruff-based linter with custom rules to detect DAG patterns and interfaces that are no longer compatible with Airflow 3.0. These checks are packaged under the `AIR30x` rule series. Example usage:

```
ruff check dags/ --select AIR301 --preview
ruff check dags/ --select AIR301 --fix --preview
```

These checks can automatically fix many common issues such as renamed arguments, removed imports, or legacy context variable usage.

Configuration Migration

Airflow 3.0 introduces a new utility to validate and upgrade your Airflow configuration file:

```
airflow config update  
airflow config update --fix
```

This utility detects removed or deprecated configuration options and, if desired, updates them in-place.

Additional validation is available via:

```
airflow config lint
```

This command surfaces obsolete configuration keys and helps align your environment with Airflow 3.0 requirements.

Metadata Database Upgrade

As with previous major releases, the Airflow 3.0 upgrade includes schema changes to the metadata database. Before upgrading, it is strongly recommended that you back up your database and optionally run:

```
airflow db clean
```

to remove old task instance, log, or XCom data. To apply the new schema:

```
airflow db migrate
```

Startup Behavior Changes

Airflow components are now started explicitly. For example:

```
airflow api-server      # Replaces airflow webserver  
airflow dag-processor  # Required in all environments
```

These changes reflect Airflow's new service-oriented architecture.

Resources

- [Upgrade Guide](#)
- [Airflow AIPs](#)

Airflow 3.0 represents more than a year of collaboration across hundreds of contributors and dozens of organizations. We thank everyone who helped shape this release through design discussions, code contributions, testing, documentation, and community feedback. For full details, migration guidance, and upgrade best practices, refer to the official Upgrade Guide and join the conversation on the Airflow dev and user mailing lists.

3.16.2 Airflow 2.10.5 (2025-02-10)

Significant Changes

Ensure teardown tasks are executed when DAG run is set to failed (#45530)

Previously when a DAG run was manually set to “failed” or to “success” state the terminal state was set to all tasks. But this was a gap for cases when setup- and teardown tasks were defined: If teardown was used to clean-up infrastructure or other resources, they were also skipped and thus resources could stay allocated.

As of now when setup tasks had been executed before and the DAG is manually set to “failed” or “success” then teardown tasks are executed. Teardown tasks are skipped if the setup was also skipped.

As a side effect this means if the DAG contains teardown tasks, then the manual marking of DAG as “failed” or “success” will need to keep the DAG in running state to ensure that teardown tasks will be scheduled. They would not be scheduled if the DAG is directly set to “failed” or “success”.

Bug Fixes

- Prevent using `trigger_rule=TriggerRule.ALWAYS` in a task-generated mapping within bare tasks (#44751)
- Fix ShortCircuitOperator mapped tasks (#44912)
- Fix premature evaluation of tasks with certain trigger rules (e.g. `ONE_DONE`) in a mapped task group (#44937)
- Fix `task_id` validation in `BaseOperator` (#44938) (#44938)
- Allow fetching XCom with forward slash from the API and escape it in the UI (#45134)
- Fix `FileTaskHandler` only read from default executor (#46000)
- Fix empty task instance for log (#45702) (#45703)
- Remove `skip_if` and `run_if` decorators before TaskFlow virtualenv tasks are run (#41832) (#45680)
- Fix request body for json requests in event log (#45546) (#45560)
- Ensure teardown tasks are executed when DAG run is set to failed (#45530) (#45581)
- Do not update DR on TI update after task execution (#45348)
- Fix object and array DAG params that have a `None` default (#45313) (#45315)
- Fix endless sensor rescheduling (#45224) (#45250)
- Evaluate `None` in SQLAlchemy’s extended JSON type decorator (#45119) (#45120)
- Allow dynamic tasks to be filtered by `rendered_map_index` (#45109) (#45122)
- Handle relative paths when sanitizing URLs (#41995) (#45080)
- Set Autocomplete Off on Login Form (#44929) (#44940)
- Add Webserver parameters `max_form_parts`, `max_form_memory_size` (#46243) (#45749)
- Fixed accessing thread local variable in `BaseOperators` `execute` safeguard mechanism (#44646) (#46280)
- Add `map_index` parameter to extra links API (#46337)

Miscellaneous

- Add traceback log output when SIGTERMs was sent (#44880) (#45077)
- Removed the ability for Operators to specify their own “scheduling deps” (#45713) (#45742)
- Deprecate `conf` from Task Context (#44993)

3.16.3 Airflow 2.10.4 (2024-12-16)

Significant Changes

TaskInstance priority_weight is capped in 32-bit signed integer ranges (#43611)

Some database engines are limited to 32-bit integer values. As some users reported errors in weight rolled-over to negative values, we decided to cap the value to the 32-bit integer. Even if internally in python smaller or larger values to 64 bit are supported, `priority_weight` is capped and only storing values from -2147483648 to 2147483647.

Bug Fixes

- Fix stats of dynamic mapped tasks after automatic retries of failed tasks (#44300)
- Fix wrong display of multi-line messages in the log after filtering (#44457)
- Allow “/” in metrics validator (#42934) (#44515)
- Fix gantt flickering (#44488) (#44517)
- Fix problem with inability to remove fields from Connection form (#40421) (#44442)
- Check pool_slots on partial task import instead of execution (#39724) (#42693)
- Avoid grouping task instance stats by try_number for dynamic mapped tasks (#44300) (#44319)
- Re-queue task when they are stuck in queued (#43520) (#44158)
- Suppress the warnings where we check for sensitive values (#44148) (#44167)
- Fix get_task_instance_try_details to return appropriate schema (#43830) (#44133)
- Log message source details are grouped (#43681) (#44070)
- Fix duplication of Task tries in the UI (#43891) (#43950)
- Add correct mime-type in OpenAPI spec (#43879) (#43901)
- Disable extra links button if link is null or empty (#43844) (#43851)
- Disable XCom list ordering by execution_date (#43680) (#43696)
- Fix venv numpy example which needs to be 1.26 at least to be working in Python 3.12 (#43659)
- Fix Try Selector in Mapped Tasks also on Index 0 (#43590) (#43591)
- Prevent using trigger_rule="always" in a dynamic mapped task (#43810)
- Prevent using trigger_rule=TriggerRule.ALWAYS in a task-generated mapping within bare tasks (#44751)

Doc Only Changes

- Update XCom docs around containers/helm (#44570) (#44573)

Miscellaneous

- Raise deprecation warning when accessing inlet or outlet events through str (#43922)

3.16.4 Airflow 2.10.3 (2024-11-05)

Significant Changes

No significant changes.

Bug Fixes

- Improves the handling of value masking when setting Airflow variables for enhanced security. (#43123) (#43278)
- Adds support for task_instance_mutation_hook to handle mapped operators with index 0. (#42661) (#43089)
- Fixes executor cleanup to properly handle zombie tasks when task instances are terminated. (#43065)
- Adds retry logic for HTTP 502 and 504 errors in internal API calls to handle webserver startup issues. (#42994) (#43044)

- Restores the use of separate sessions for writing and deleting RTIF data to prevent StaleDataError. (#42928) (#43012)
- Fixes PythonOperator error by replacing hyphens with underscores in DAG names. (#42993)
- Improving validation of task retries to handle None values (#42532) (#42915)
- Fixes error handling in dataset managers when resolving dataset aliases into new datasets (#42733)
- Enables clicking on task names in the DAG Graph View to correctly select the corresponding task. (#38782) (#42697)
- Prevent redirect loop on /home with tags/last run filters (#42607) (#42609) (#42628)
- Support of host.name in OTEL metrics and usage of OTEL_RESOURCE_ATTRIBUTES in metrics (#42428) (#42604)
- Reduce eyestrain in dark mode with reduced contrast and saturation (#42567) (#42583)
- Handle ENTER key correctly in trigger form and allow manual JSON (#42525) (#42535)
- Ensure DAG trigger form submits with updated parameters upon keyboard submit (#42487) (#42499)
- Do not attempt to provide not `stringified` objects to UI via xcom if pickling is active (#42388) (#42486)
- Fix the span link of task instance to point to the correct span in the scheduler_job_loop (#42430) (#42480)
- Bugfix task execution from runner in Windows (#42426) (#42478)
- Allows overriding the hardcoded OTEL_SERVICE_NAME with an environment variable (#42242) (#42441)
- Improves trigger performance by using `selectinload` instead of `joinedload` (#40487) (#42351)
- Suppress warnings when masking sensitive configs (#43335) (#43337)
- Masking configuration values irrelevant to DAG author (#43040) (#43336)
- Execute templated bash script as file in BashOperator (#43191)
- Fixes schedule_downstream_tasks to include upstream tasks for one_success trigger rule (#42582) (#43299)
- Add retry logic in the scheduler for updating trigger timeouts in case of deadlocks. (#41429) (#42651)
- Mark all tasks as skipped when failing a dag_run manually (#43572)
- Fix TrySelector for Mapped Tasks in Logs and Details Grid Panel (#43566)
- Conditionally add OTEL events when processing executor events (#43558) (#43567)
- Fix broken stat `scheduler_loop_duration` (#42886) (#43544)
- Ensure total_entries in /api/v1/dags (#43377) (#43429)
- Include limit and offset in request body schema for List task instances (batch) endpoint (#43479)
- Don't raise a warning in ExecutorSafeguard when execute is called from an extended operator (#42849) (#43577)

Miscellaneous

- Deprecate session auth backend (#42911)
- Removed unicodercsv dependency for providers with Airflow version 2.8.0 and above (#42765) (#42970)
- Remove the referrer from Webserver to Scarf (#42901) (#42942)
- Bump dompurify from 2.2.9 to 2.5.6 in /airflow/www (#42263) (#42270)
- Correct docstring format in `_get_template_context` (#42244) (#42272)

- Backport: Bump Flask-AppBuilder to 4.5.2 (#43309) (#43318)
- Check python version that was used to install pre-commit venvs (#43282) (#43310)
- Resolve warning in Dataset Alias migration (#43425)

Doc Only Changes

- Clarifying PLUGINS_FOLDER permissions by DAG authors (#43022) (#43029)
- Add templating info to TaskFlow tutorial (#42992)
- Airflow local settings no longer importable from dags folder (#42231) (#42603)
- Fix documentation for cpu and memory usage (#42147) (#42256)
- Fix instruction for docker compose (#43119) (#43321)
- Updates documentation to reflect that dag_warnings is returned instead of import_errors. (#42858) (#42888)

3.16.5 Airflow 2.10.2 (2024-09-18)

Significant Changes

No significant changes.

Bug Fixes

- Revert “Fix: DAGs are not marked as stale if the dags folder change” (#42220, #42217)
- Add missing open telemetry span and correct scheduled slots documentation (#41985)
- Fix require_confirmation_dag_change (#42063) (#42211)
- Only treat null/undefined as falsy when rendering XComEntry (#42199) (#42213)
- Add extra and `renderedTemplates` as keys to skip camelCasing (#42206) (#42208)
- Do not camelcase xcom entries (#42182) (#42187)
- Fix task_instance and dag_run links from list views (#42138) (#42143)
- Support multi-line input for Params of type string in trigger UI form (#40414) (#42139)
- Fix details tab log url detection (#42104) (#42114)
- Add new type of exception to catch timeout (#42064) (#42078)
- Rewrite how DAG to dataset / dataset alias are stored (#41987) (#42055)
- Allow dataset alias to add more than one dataset events (#42189) (#42247)

Miscellaneous

- Limit universal-pathlib below 0.2.4 as it breaks our integration (#42101)
- Auto-fix default deferrable with LibCST (#42089)
- Deprecate `--tree` flag for `tasks list` cli command (#41965)

Doc Only Changes

- Update `security_model.rst` to clear unauthenticated endpoints exceptions (#42085)
- Add note about dataclasses and attrs to XComs page (#42056)
- Improve docs on markdown docs in DAGs (#42013)
- Add warning that listeners can be dangerous (#41968)

3.16.6 Airflow 2.10.1 (2024-09-05)

Significant Changes

No significant changes.

Bug Fixes

- Handle Example dags case when checking for missing files (#41874)
- Fix logout link in “no roles” error page (#41845)
- Set end_date and duration for triggers completed with end_from_trigger as True. (#41834)
- DAGs are not marked as stale if the dags folder change (#41829)
- Fix compatibility with FAB provider versions <1.3.0 (#41809)
- Don’t Fail LocalTaskJob on heartbeat (#41810)
- Remove deprecation warning for cgitb in Plugins Manager (#41793)
- Fix log for notifier(instance) without __name__ (#41699)
- Splitting syspath preparation into stages (#41694)
- Adding url sanitization for extra links (#41680)
- Fix InletEventsAccessors type stub (#41607)
- Fix UI rendering when XCom is INT, FLOAT, BOOL or NULL (#41605)
- Fix try selector refresh (#41503)
- Incorrect try number subtraction producing invalid span id for OTEL airflow (#41535)
- Add WebEncoder for trigger page rendering to avoid render failure (#41485)
- Adding `tojson` filter to `example_inlet_event_extra` example dag (#41890)
- Add backward compatibility check for executors that don’t inherit BaseExecutor (#41927)

Miscellaneous

- Bump webpack from 5.76.0 to 5.94.0 in /airflow/www (#41879)
- Adding rel property to hyperlinks in logs (#41783)
- Field Deletion Warning when editing Connections (#41504)
- Make Scarf usage reporting in major+minor versions and counters in buckets (#41900)
- Lower down universal-pathlib minimum to 0.2.2 (#41943)
- Protect against None components of universal pathlib xcom backend (#41938)

Doc Only Changes

- Remove Debian bullseye support (#41569)
- Add an example for auth with keycloak (#41791)

3.16.7 Airflow 2.10.0 (2024-08-15)

Significant Changes

Scarf based telemetry: Airflow now collect telemetry data (#39510)

Airflow integrates Scarf to collect basic usage data during operation. Deployments can opt-out of data collection by setting the [usage_data_collection]enabled option to False, or the SCARF_ANALYTICS=false environment variable.

Datasets no longer trigger inactive DAGs (#38891)

Previously, when a DAG is paused or removed, incoming dataset events would still trigger it, and the DAG would run when it is unpause or added back in a DAG file. This has been changed; a DAG's dataset schedule can now only be satisfied by events that occur when the DAG is active. While this is a breaking change, the previous behavior is considered a bug.

The behavior of time-based scheduling is unchanged, including the timetable part of DatasetOrTimeSchedule.

try_number is no longer incremented during task execution (#39336)

Previously, the try number (try_number) was incremented at the beginning of task execution on the worker. This was problematic for many reasons. For one it meant that the try number was incremented when it was not supposed to, namely when resuming from reschedule or deferral. And it also resulted in the try number being “wrong” when the task had not yet started. The workarounds for these two issues caused a lot of confusion.

Now, instead, the try number for a task run is determined at the time the task is scheduled, and does not change in flight, and it is never decremented. So after the task runs, the observed try number remains the same as it was when the task was running; only when there is a “new try” will the try number be incremented again.

One consequence of this change is, if users were “manually” running tasks (e.g. by calling ti.run() directly, or command line airflow tasks run), try number will no longer be incremented. Airflow assumes that tasks are always run after being scheduled by the scheduler, so we do not regard this as a breaking change.

/logout endpoint in FAB Auth Manager is now CSRF protected (#40145)

The /logout endpoint’s method in FAB Auth Manager has been changed from GET to POST in all existing AuthViews (AuthDBView, AuthLDAPView, AuthOAuthView, AuthOIDView, AuthRemoteUserView), and now includes CSRF protection to enhance security and prevent unauthorized logouts.

OpenTelemetry Traces for Apache Airflow (#37948).

This new feature adds capability for Apache Airflow to emit 1) airflow system traces of scheduler, triggerer, executor, processor 2) DAG run traces for deployed DAG runs in OpenTelemetry format. Previously, only metrics were supported which emitted metrics in OpenTelemetry. This new feature will add richer data for users to use OpenTelemetry standard to emit and send their trace data to OTLP compatible endpoints.

Decorator for Task Flow (@skip_if, @run_if) to make it simple to apply whether or not to skip a Task. (#41116)

This feature adds a decorator to make it simple to skip a Task.

Using Multiple Executors Concurrently (#40701)

Previously known as hybrid executors, this new feature allows Airflow to use multiple executors concurrently. DAGs, or even individual tasks, can be configured to use a specific executor that suits its needs best. A single DAG can contain tasks all using different executors. Please see the Airflow documentation for more details. Note: This feature is still experimental. See [documentation on Executor](#) for a more detailed description.

New Features

- AIP-61 Hybrid Execution ([AIP-61](#))
- AIP-62 Getting Lineage from Hook Instrumentation ([AIP-62](#))
- AIP-64 TaskInstance Try History ([AIP-64](#))
- AIP-44 Internal API ([AIP-44](#))
- Enable ending the task directly from the triggerer without going into the worker. (#40084)
- Extend dataset dependencies (#40868)
- Feature/add token authentication to internal api (#40899)
- Add DatasetAlias to support dynamic Dataset Event Emission and Dataset Creation (#40478)
- Add example DAGs for inlet_events (#39893)
- Implement `accessors` to read dataset events defined as inlet (#39367)
- Decorator for Task Flow, to make it simple to apply whether or not to skip a Task. (#41116)
- Add start execution from triggerer support to dynamic task mapping (#39912)
- Add try_number to log table (#40739)
- Added `ds_format_locale` method in macros which allows localizing datetime formatting using Babel (#40746)
- Add DatasetAlias to support dynamic Dataset Event Emission and Dataset Creation (#40478, #40723, #40809, #41264, #40830, #40693, #41302)
- Use sentinel to mark dag as removed on re-serialization (#39825)
- Add parameter for the last number of queries to the DB in DAG file processing stats (#40323)
- Add prototype version dark mode for Airflow UI (#39355)
- Add ability to mark some tasks as successful in `dag test` (#40010)
- Allow use of callable for `template_fields` (#37028)
- Filter running/failed and active/paused dags on the home page (#39701)
- Add metrics about task CPU and memory usage (#39650)
- UI changes for DAG Re-parsing feature (#39636)
- Add Scarf based telemetry (#39510, #41318)
- Add dag re-parsing request endpoint (#39138)
- Redirect to new DAGRun after trigger from Grid view (#39569)

- Display endDate in task instance tooltip. (#39547)
- Implement accessors to read dataset events defined as inlet (#39367, #39893)
- Add color to log lines in UI for error and warnings based on keywords (#39006)
- Add Rendered k8s pod spec tab to ti details view (#39141)
- Make audit log before/after filterable (#39120)
- Consolidate grid collapse actions to a single full screen toggle (#39070)
- Implement Metadata to emit runtime extra (#38650)
- Add executor field to the DB and parameter to the operators (#38474)
- Implement context accessor for DatasetEvent extra (#38481)
- Add dataset event info to dag graph (#41012)
- Add button to toggle datasets on/off in dag graph (#41200)
- Add `run_if` & `skip_if` decorators (#41116)
- Add `dag_stats` rest api endpoint (#41017)
- Add listeners for Dag import errors (#39739)
- Allowing DateTimeSensorAsync, FileSensor and TimeSensorAsync to start execution from trigger during dynamic task mapping (#41182)

Improvements

- Allow set Dag Run resource into Dag Level permission: extends Dag's access_control feature to allow Dag Run resource permissions. (#40703)
- Improve security and error handling for the internal API (#40999)
- Datasets UI Improvements (#40871)
- Change DAG Audit log tab to Event Log (#40967)
- Make standalone dag file processor works in DB isolation mode (#40916)
- Show only the source on the consumer DAG page and only triggered DAG run in the producer DAG page (#41300)
- Update metrics names to allow multiple executors to report metrics (#40778)
- Format DAG run count (#39684)
- Update styles for `renderedjson` component (#40964)
- Improve ATTRIBUTE_REMOVED sentinel to use class and more context (#40920)
- Make XCom display as react json (#40640)
- Replace usages of task context logger with the log table (#40867)
- Rollback for all retry exceptions (#40882) (#40883)
- Support rendering ObjectStoragePath value (#40638)
- Add `try_number` and `map_index` as params for log event endpoint (#40845)
- Rotate fernet key in batches to limit memory usage (#40786)
- Add gauge metric for 'last_num_of_db_queries' parameter (#40833)
- Set parallelism log messages to warning level for better visibility (#39298)

- Add error handling for encoding the dag runs (#40222)
- Use params instead of dag_run.conf in example DAG (#40759)
- Load Example Plugins with Example DAGs (#39999)
- Stop deferring TimeDeltaSensorAsync task when the target_dttm is in the past (#40719)
- Send important executor logs to task logs (#40468)
- Open external links in new tabs (#40635)
- Attempt to add ReactJSON view to rendered templates (#40639)
- Speeding up regex match time for custom warnings (#40513)
- Refactor DAG.dataset_triggers into the timetable class (#39321)
- add next_kwargs to StartTriggerArgs (#40376)
- Improve UI error handling (#40350)
- Remove double warning in CLI when config value is deprecated (#40319)
- Implement XComArg concat() (#40172)
- Added get_extra_dejson method with nested parameter which allows you to specify if you want the nested json as string to be also deserialized (#39811)
- Add executor field to the task instance API (#40034)
- Support checking for db path absoluteness on Windows (#40069)
- Introduce StartTriggerArgs and prevent start trigger initialization in scheduler (#39585)
- Add task documentation to details tab in grid view (#39899)
- Allow executors to be specified with only the class name of the Executor (#40131)
- Remove obsolete conditional logic related to try_number (#40104)
- Allow Task Group Ids to be passed as branches in BranchMixIn (#38883)
- Javascript connection form will apply CodeMirror to all textarea's dynamically (#39812)
- Determine needs_expansion at time of serialization (#39604)
- Add indexes on dag_id column in referencing tables to speed up deletion of dag records (#39638)
- Add task failed dependencies to details page (#38449)
- Remove webserver try_number adjustment (#39623)
- Implement slicing in lazy sequence (#39483)
- Unify lazy db sequence implementations (#39426)
- Add __getattr__ to task decorator stub (#39425)
- Allow passing labels to FAB Views registered via Plugins (#39444)
- Simpler error message when trying to offline migrate with sqlite (#39441)
- Add soft_fail to TriggerDagRunOperator (#39173)
- Rename “dataset event” in context to use “outlet” (#39397)
- Resolve RemovedIn20Warning in airflow task command (#39244)
- Determine fail_stop on client side when db isolated (#39258)

- Refactor cloudpickle support in Python operators/decorators (#39270)
- Update trigger kwargs migration to specify existing_nullable (#39361)
- Allowing tasks to start execution directly from triggerer without going to worker (#38674)
- Better db migrate error messages (#39268)
- Add stacklevel into the suppress_and_warn warning (#39263)
- Support searching by dag_display_name (#39008)
- Allow sort by on all fields in MappedInstances.tsx (#38090)
- Expose count of scheduled tasks in metrics (#38899)
- Use declarative_base from sqlalchemy.orm instead of sqlalchemy.ext.declarative (#39134)
- Add example DAG to demonstrate emitting approaches (#38821)
- Give on_task_instance_failed access to the error that caused the failure (#38155)
- Simplify dataset serialization (#38694)
- Add heartbeat recovery message to jobs (#34457)
- Remove select_column option in TaskInstance.get_task_instance (#38571)
- Don't create session in get_dag if not reading dags from database (#38553)
- Add a migration script for encrypted trigger kwargs (#38358)
- Implement render_templates on TaskInstancePydantic (#38559)
- Handle optional session in _refresh_from_db (#38572)
- Make type annotation less confusing in task_command.py (#38561)
- Use fetch_dagrun directly to avoid session creation (#38557)
- Added output_processor parameter to BashProcessor (#40843)
- Improve serialization for Database Isolation Mode (#41239)
- Only orphan non-orphaned Datasets (#40806)
- Adjust gantt width based on task history dates (#41192)
- Enable scrolling on legend with high number of elements. (#41187)

Bug Fixes

- Bugfix for get_parsing_context() when ran with LocalExecutor (#40738)
- Validating provider documentation urls before displaying in views (#40933)
- Move import to make PythonOperator working on Windows (#40424)
- Fix dataset_with_extra_from_classic_operator example DAG (#40747)
- Call listener on_task_instance_failed() after ti state is changed (#41053)
- Add never_fail in BaseSensor (#40915)
- Fix tasks API endpoint when DAG doesn't have start_date (#40878)
- Fix and adjust URL generation for UI grid and older runs (#40764)
- Rotate fernet key optimization (#40758)

- Fix class instance vs. class type in validate_database_executor_compatibility() call (#40626)
- Clean up dark mode (#40466)
- Validate expected types for args for DAG, BaseOperator and TaskGroup (#40269)
- Exponential Backoff Not Functioning in BaseSensorOperator Reschedule Mode (#39823)
- local task job: add timeout, to not kill on_task_instance_success listener prematurely (#39890)
- Move Post Execution Log Grouping behind Exception Print (#40146)
- Fix triggerer race condition in HA setting (#38666)
- Pass triggered or existing DAG Run logical date to DagStateTrigger (#39960)
- Passing `external_task_group_id` to `WorkflowTrigger` (#39617)
- ECS Executor: Set tasks to RUNNING state once active (#39212)
- Only heartbeat if necessary in backfill loop (#39399)
- Fix trigger kwarg encryption migration (#39246)
- Fix decryption of trigger kwargs when downgrading. (#38743)
- Fix wrong link in TriggeredDagRuns (#41166)
- Pass MapIndex to LogLink component for external log systems (#41125)
- Add NonCachingRotatingFileHandler for worker task (#41064)
- Add argument include_xcom in method resolve an optional value (#41062)
- Sanitizing file names in example_bash_decorator DAG (#40949)
- Show dataset aliases in dependency graphs (#41128)
- Render Dataset Conditions in DAG Graph view (#41137)
- Add task duration plot across dagruns (#40755)
- Add start execution from trigger support for existing core sensors (#41021)
- add example dag for dataset_alias (#41037)
- Add dataset alias unique constraint and remove wrong dataset alias removing logic (#41097)
- Set “has_outlet_datasets” to true if “dataset alias” exists (#41091)
- Make HookLineageCollector group datasets by (#41034)
- Enhance start_trigger_args serialization (#40993)
- Refactor `BaseSensorOperator` introduce `skip_policy` parameter (#40924)
- Fix viewing logs from triggerer when task is deferred (#41272)
- Refactor how triggered dag run url is replaced (#41259)
- Added support for additional sql alchemy session args (#41048)
- Allow empty list in TriggerDagRun failed_state (#41249)
- Clean up the exception handler when run_as_user is the airflow user (#41241)
- Collapse docs when click and folded (#41214)
- Update updated_at when saving to db as session.merge does not trigger on-update (#40782)
- Fix query count statistics when parsing DAF file (#41149)

- Method Resolution Order in operators without `__init__` (#41086)
- Ensure `try_number` incremented for empty operator (#40426)

Miscellaneous

- Remove the Experimental flag from OTel Traces (#40874)
- Bump packaging version to 23.0 in order to fix issue with older otel (#40865)
- Simplify `_auth_manager_is_authorized_map` function (#40803)
- Use correct unknown executor exception in scheduler job (#40700)
- Add D1 `pydocstyle` rules to `pyproject.toml` (#40569)
- Enable enforcing `pydocstyle` rule D213 in ruff. (#40448, #40464)
- Update `Dag.test()` to run with an executor if desired (#40205)
- Update jest and babel minor versions (#40203)
- Refactor BashOperator and Bash decorator for consistency and simplicity (#39871)
- Add `AirflowInternalRuntimeError` for raise non `catchable` errors (#38778)
- ruff version bump 0.4.5 (#39849)
- Bump `pytest` to 8.0+ (#39450)
- Remove stale comment about TI index (#39470)
- Configure `back_populates` between `DagScheduleDatasetReference.dag` and `DagModel.schedule_dataset_references` (#39392)
- Remove deprecation warnings in `endpoints.py` (#39389)
- Fix SQLA deprecations in Airflow core (#39211)
- Use class-bound attribute directly in SA (#39198, #39195)
- Fix stacklevel for `TaskContextLogger` (#39142)
- Capture warnings during collect DAGs (#39109)
- Resolve B028 (no-explicit-stacklevel) in core (#39123)
- Rename model `ModelError` to `ParseModelError` for avoid shadowing with builtin exception (#39116)
- Add option to support cloudpickle in PythonVenv/External Operator (#38531)
- Suppress `SubDagOperator` examples warnings (#39057)
- Add log for running callback (#38892)
- Use `model_dump` instead of `dict` for serialize Pydantic V2 model (#38933)
- Widen cheat sheet column to avoid wrapping commands (#38888)
- Update `hatchling` to latest version (1.22.5) (#38780)
- bump uv to 0.1.29 (#38758)
- Add missing serializations found during provider tests fixing (#41252)
- Bump ws from 7.5.5 to 7.5.10 in /airflow/www (#40288)
- Improve typing for allowed/failed_states in `TriggerDagRunOperator` (#39855)

Doc Only Changes

- Add `filesystems` and `dataset-uris` to “how to create your own provider” page (#40801)
- Fix (TM) to (R) in Airflow repository (#40783)
- Set `otel_on` to True in example airflow.cfg (#40712)
- Add warning for `_AIRFLOW_PATCH_GEVENT` (#40677)
- Update multi-team diagram proposal after Airflow 3 discussions (#40671)
- Add stronger warning that MSSQL is not supported and no longer functional (#40565)
- Fix misleading mac menu structure in howto (#40440)
- Update k8s supported version in docs (#39878)
- Add compatibility note for Listeners (#39544)
- Update edge label image in documentation example with the new graph view (#38802)
- Update UI doc screenshots (#38680)
- Add section “Manipulating queued dataset events through REST API” (#41022)
- Add information about lack of security guarantees for docker compose (#41072)
- Add links to example dags in use params section (#41031)
- Change `task_id` from `send_email` to `send_email_notification` in `taskflow.rst` (#41060)
- Remove unnecessary nginx redirect rule from reverse proxy documentation (#38953)

3.16.8 Airflow 2.9.3 (2024-07-15)

Significant Changes

Time unit for `scheduled_duration` and `queued_duration` changed (#37936)

`scheduled_duration` and `queued_duration` metrics are now emitted in milliseconds instead of seconds.

By convention all statsd metrics should be emitted in milliseconds, this is later expected in e.g. prometheus statsd-exporter.

Support for OpenTelemetry Metrics is no longer “Experimental” (#40286)

Experimental support for OpenTelemetry was added in 2.7.0 since then fixes and improvements were added and now we announce the feature as stable.

Bug Fixes

- Fix calendar view scroll (#40458)
- Validating provider description for urls in provider list view (#40475)
- Fix compatibility with old MySQL 8.0 (#40314)
- Fix dag (un)pausing won’t work on environment where dag files are missing (#40345)
- Extra being passed to SQLAlchemy (#40391)
- Handle unsupported operand int + str when value of tag is int (job_id) (#40407)
- Fix TriggeredDagRunOperator triggered link (#40336)

- Add [webserver]update_fab_perms to deprecated configs (#40317)
- Swap dag run link from legacy graph to grid with graph tab (#40241)
- Change httpx to requests in file_task_handler (#39799)
- Fix import future annotations in venv jinja template (#40208)
- Ensures DAG params order regardless of backend (#40156)
- Use a join for TI notes in TI batch API endpoint (#40028)
- Improve trigger UI for string array format validation (#39993)
- Disable jinja2 rendering for doc_md (#40522)
- Skip checking sub dags list if taskinstance state is skipped (#40578)
- Recognize quotes when parsing urls in logs (#40508)

Doc Only Changes

- Add notes about passing secrets via environment variables (#40519)
- Revamp some confusing log messages (#40334)
- Add more precise description of masking sensitive field names (#40512)
- Add slightly more detailed guidance about upgrading to the docs (#40227)
- Metrics allow_list complete example (#40120)
- Add warning to deprecated api docs that access control isn't applied (#40129)
- Simpler command to check local scheduler is alive (#40074)
- Add a note and an example clarifying the usage of DAG-level params (#40541)
- Fix highlight of example code in dags.rst (#40114)
- Add warning about the PostgresOperator being deprecated (#40662)
- Updating airflow download links to CDN based links (#40618)
- Fix import statement for DatasetOrTimetable example (#40601)
- Further clarify triage process (#40536)
- Fix param order in PythonOperator docstring (#40122)
- Update serializers.rst to mention that bytes are not supported (#40597)

Miscellaneous

- Upgrade build installers and dependencies (#40177)
- Bump braces from 3.0.2 to 3.0.3 in /airflow/www (#40180)
- Upgrade to another version of trove-classifier (new CUDA classifiers) (#40564)
- Rename “try_number” increments that are unrelated to the airflow concept (#39317)
- Update trove classifiers to the latest version as build dependency (#40542)
- Upgrade to latest version of hatchling as build dependency (#40387)
- Fix bug in SchedulerJobRunner._process_executor_events (#40563)
- Remove logging for “blocked” events (#40446)

3.16.9 Airflow 2.9.2 (2024-06-10)

Significant Changes

No significant changes.

Bug Fixes

- Fix bug that makes `AirflowSecurityManagerV2` leave transactions in the `idle in transaction` state (#39935)
- Fix alembic auto-generation and rename mismatching constraints (#39032)
- Add the `existing_nullable` to the downgrade side of the migration (#39374)
- Fix Mark Instance state buttons stay disabled if user lacks permission (#37451). (#38732)
- Use `SKIP LOCKED` instead of `NOWAIT` in mini scheduler (#39745)
- Remove DAG Run Add option from FAB view (#39881)
- Add `max_consecutive_failed_dag_runs` in API spec (#39830)
- Fix `example_branch_operator` failing in python 3.12 (#39783)
- Fetch served logs also when task attempt is up for retry and no remote logs available (#39496)
- Change dataset URI validation to raise warning instead of error in Airflow 2.9 (#39670)
- Visible DAG RUN doesn't point to the same dag run id (#38365)
- Refactor `SafeDogStatsdLogger` to use `get_validator` to enable pattern matching (#39370)
- Fix custom actions in security manager `has_access` (#39421)
- Fix HTTP 500 Internal Server Error if DAG is triggered with bad params (#39409)
- Fix static file caching is disabled in Airflow Webserver. (#39345)
- Fix `TaskHandlerWithCustomFormatter` now adds prefix only once (#38502)
- Do not provide deprecated `execution_date` in `@apply_lineage` (#39327)
- Add missing `conn_id` to string representation of `ObjectStoragePath` (#39313)
- Fix `sqlalchemy_engine_args` config example (#38971)
- Add Cache-Control “no-store” to all dynamically generated content (#39550)

Miscellaneous

- Limit `yandex` provider to avoid `mypy` errors (#39990)
- Warn on mini scheduler failures instead of debug (#39760)
- Change type definition for `provider_info_cache` decorator (#39750)
- Better typing for `BaseOperator defer` (#39742)
- More typing in `TimeSensor` and `TimeSensorAsync` (#39696)
- Re-raise exception from strict dataset URI checks (#39719)
- Fix stacklevel for `_log_state` helper (#39596)
- Resolve SA warnings in migrations scripts (#39418)
- Remove unused index `idx_last_scheduling_decision` on `dag_run` table (#39275)

Doc Only Changes

- Provide extra tip on labeling DynamicTaskMapping (#39977)
- Improve visibility of links / variables / other configs in Configuration Reference (#39916)
- Remove ‘legacy’ definition for CronDataIntervalTimetable (#39780)
- Update plugins.rst examples to use pyproject.toml over setup.py (#39665)
- Fix nit in pg set-up doc (#39628)
- Add Matomo to Tracking User Activity docs (#39611)
- Fix Connection.get -> Connection.get_connection_from_secrets (#39560)
- Adding note for provider dependencies (#39512)
- Update docker-compose command (#39504)
- Update note about restarting triggerer process (#39436)
- Updating S3LogLink with an invalid bucket link (#39424)
- Update testing_packages.rst (#38996)
- Add multi-team diagrams (#38861)

3.16.10 Airflow 2.9.1 (2024-05-03)

Significant Changes

Stackdriver logging bugfix requires Google provider 10.17.0 or later (#38071)

If you use Stackdriver logging, you must use Google provider version 10.17.0 or later. Airflow 2.9.1 now passes `gcp_log_name` to the `StackdriverTaskHandler` instead of `name`, and this will fail on earlier provider versions.

This fixes a bug where the log name configured in `[logging] remove_base_log_folder` was overridden when Airflow configured logging, resulting in task logs going to the wrong destination.

Bug Fixes

- Make task log messages include run_id (#39280)
- Copy menu_item href for nav bar (#39282)
- Fix trigger kwarg encryption migration (#39246, #39361, #39374)
- Add workaround for datetime-local input in firefox (#39261)
- Add Grid button to Task Instance view (#39223)
- Get served logs when remote or executor logs not available for non-running task try (#39177)
- Fixed side effect of menu filtering causing disappearing menus (#39229)
- Use grid view for Task Instance’s log_url (#39183)
- Improve task filtering UX (#39119)
- Improve rendered_template ux in react dag page (#39122)
- Graph view improvements (#38940)
- Check that the dataset<>task exists before trying to render graph (#39069)
- Hostname was “redacted”, not “redact”; remove it when there is no context (#39037)

- Check whether `AUTH_ROLE_PUBLIC` is set in `check_authentication` (#39012)
- Move rendering of `map_index_template` so it renders for failed tasks as long as it was defined before the point of failure (#38902)
- Undeprecate `BaseXCom.get_one` method for now (#38991)
- Add `inherit_cache` attribute for `CreateTableAs` custom SA Clause (#38985)
- Don't wait for `DagRun` lock in mini scheduler (#38914)
- Fix calendar view with no DAG Run (#38964)
- Changed the background color of external task in graph (#38969)
- Fix dag run selection (#38941)
- Fix `SAWarning` ‘Coercing Subquery object into a select() for use in IN()’ (#38926)
- Fix implicit cartesian product in `AirflowSecurityManagerV2` (#38913)
- Fix problem that links in legacy log view can not be clicked (#38882)
- Fix dag run link params (#38873)
- Use async db calls in `WorkflowTrigger` (#38689)
- Fix audit log events filter (#38719)
- Use `methodtools.lru_cache` instead of `functools.lru_cache` in class methods (#37757)
- Raise deprecated warning in `airflow dags backfill` only if `-I` / `--ignore-first-depends-on-past` provided (#38676)

Miscellaneous

- `TriggerDagRunOperator` deprecate `execution_date` in favor of `logical_date` (#39285)
- Force to use Airflow Deprecation warnings categories on `@deprecated` decorator (#39205)
- Add warning about run/import Airflow under the Windows (#39196)
- Update `is_authorized_custom_view` from auth manager to handle custom actions (#39167)
- Add in Trove classifiers Python 3.12 support (#39004)
- Use debug level for `minischeduler` skip (#38976)
- Bump `undici` from 5.28.3 to 5.28.4 in `/airflow/www` (#38751)

Doc Only Changes

- Fix supported k8s version in docs (#39172)
- Dynamic task mapping `PythonOperator` `op_kwargs` (#39242)
- Add link to `user` and `role` commands (#39224)
- Add `k8s 1.29` to supported version in docs (#39168)
- Data aware scheduling docs edits (#38687)
- Update `DagBag` class docstring to include all params (#38814)
- Correcting an example taskflow example (#39015)
- Remove decorator from rendering fields example (#38827)

3.16.11 Airflow 2.9.0 (2024-04-08)

Significant Changes

Following Listener API methods are considered stable and can be used for production system (were experimental feature in older Airflow versions) (#36376):

Lifecycle events:

- `on_starting`
- `before_stopping`

DagRun State Change Events:

- `on_dag_run_running`
- `on_dag_run_success`
- `on_dag_run_failed`

TaskInstance State Change Events:

- `on_task_instance_running`
- `on_task_instance_success`
- `on_task_instance_failed`

Support for Microsoft SQL-Server for Airflow Meta Database has been removed (#36514)

After discussion and a voting process, the Airflow's PMC members and Committers have reached a resolution to no longer maintain MsSQL as a supported Database Backend.

As of Airflow 2.9.0 support of MsSQL has been removed for Airflow Database Backend.

A migration script which can help migrating the database *before* upgrading to Airflow 2.9.0 is available in [airflow-mssql-migration repo on Github](#). Note that the migration script is provided without support and warranty.

This does not affect the existing provider packages (operators and hooks), DAGs can still access and process data from MsSQL.

Dataset URIs are now validated on input (#37005)

Datasets must use a URI that conform to rules laid down in AIP-60, and the value will be automatically normalized when the DAG file is parsed. See [documentation on Datasets](#) for a more detailed description on the rules.

You may need to change your Dataset identifiers if they look like a URI, but are used in a less mainstream way, such as relying on the URI's auth section, or have a case-sensitive protocol name.

The method `get_permitted_menu_items` in `BaseAuthManager` has been renamed `filter_permitted_menu_items` (#37627)

Add REST API actions to Audit Log events (#37734)

The Audit Log event name for REST API events will be prepended with `api.` or `ui.`, depending on if it came from the Airflow UI or externally.

Official support for Python 3.12 (#38025)

There are a few caveats though:

- Pendulum2 does not support Python 3.12. For Python 3.12 you need to use Pendulum 3
- Minimum SQLAlchemy version supported when Pandas is installed for Python 3.12 is 1.4.36 released in April 2022. Airflow 2.9.0 increases the minimum supported version of SQLAlchemy to 1.4.36 for all Python versions.

Not all Providers support Python 3.12. At the initial release of Airflow 2.9.0 the following providers are released without support for Python 3.12:

- apache.beam - pending on Apache Beam support for 3.12
- papermill - pending on Releasing Python 3.12 compatible papermill client version including this merged issue

Prevent large string objects from being stored in the Rendered Template Fields (#38094)

There's now a limit to the length of data that can be stored in the Rendered Template Fields. The limit is set to 4096 characters. If the data exceeds this limit, it will be truncated. You can change this limit by setting the [core]max_template_field_length configuration option in your airflow config.

Change xcom table column value type to longblob for MySQL backend (#38401)

Xcom table column value type has changed from blob to longblob. This will allow you to store relatively big data in Xcom but process can take a significant amount of time if you have a lot of large data stored in Xcom.

To downgrade from revision: b4078ac230a1, ensure that you don't have Xcom values larger than 65,535 bytes. Otherwise, you'll need to clean those rows or run `airflow db clean xcom` to clean the Xcom table.

Stronger validation for key parameter defaults in taskflow context variables (#38015)

As for the taskflow implementation in conjunction with context variable defaults invalid parameter orders can be generated, it is now not accepted anymore (and validated) that taskflow functions are defined with defaults other than None. If you have done this before you most likely will see a broken DAG and a error message like `Error message: Context key parameter my_param can't have a default other than None.`

New Features

- Allow users to write dag_id and task_id in their national characters, added display name for dag / task (v2) (#38446)
- Prevent large objects from being stored in the RTIF (#38094)
- Use current time to calculate duration when end date is not present. (#38375)
- Add average duration mark line in task and dagrun duration charts. (#38214, #38434)
- Add button to manually create dataset events (#38305)
- Add Matomo as an option for analytics_tool. (#38221)
- Experimental: Support custom weight_rule implementation to calculate the TI priority_weight (#38222)
- Adding ability to automatically set DAG to off after X times it failed sequentially (#36935)
- Add dataset conditions to next run datasets modal (#38123)
- Add task log grouping to UI (#38021)
- Add dataset_expression to grid dag details (#38121)
- Introduce mechanism to support multiple executor configuration (#37635)

- Add color formatting for ANSI chars in logs from task executions (#37985)
- Add the dataset_expression as part of DagModel and DAGDetailSchema (#37826)
- Allow longer rendered_map_index (#37798)
- Inherit the run_ordering from DatasetTriggeredTimetable for DatasetOrTimeSchedule (#37775)
- Implement AIP-60 Dataset URI formats (#37005)
- Introducing Logical Operators for dataset conditional logic (#37101)
- Add post endpoint for dataset events (#37570)
- Show custom instance names for a mapped task in UI (#36797)
- Add excluded/include events to get_event_logs api (#37641)
- Add datasets to dag graph (#37604)
- Show dataset events above task/run details in grid view (#37603)
- Introduce new config variable to control whether DAG processor outputs to stdout (#37439)
- Make Datasets hashable (#37465)
- Add conditional logic for dataset triggering (#37016)
- Implement task duration page in react. (#35863)
- Add queuedEvent endpoint to get/delete DatasetDagRunQueue (#37176)
- Support multiple XCom output in the BaseOperator (#37297)
- AIP-58: Add object storage backend for xcom (#37058)
- Introduce DatasetOrTimeSchedule (#36710)
- Add on_skipped_callback to BaseOperator (#36374)
- Allow override of hovered navbar colors (#36631)
- Create new Metrics with Tagging (#36528)
- Add support for openlineage to AFS and common.io (#36410)
- Introduce @task.bash TaskFlow decorator (#30176, #37875)

Improvements

- More human friendly “show tables” output for db cleanup (#38654)
- Improve trigger assign_unassigned by merging alive_triggerer_ids and get_sorted_triggers queries (#38664)
- Add exclude/include events filters to audit log (#38506)
- Clean up unused triggers in a single query for all dialects except MySQL (#38663)
- Update Confirmation Logic for Config Changes on Sensitive Environments Like Production (#38299)
- Improve datasets graph UX (#38476)
- Only show latest dataset event timestamp after last run (#38340)
- Add button to clear only failed tasks in a dagrun. (#38217)
- Delete all old dag pages and redirect to grid view (#37988)
- Check task attribute before use in sentry.add_tagging() (#37143)

- Mysql change xcom value col type for MySQL backend (#38401)
- ExternalPythonOperator use version from `sys.version_info` (#38377)
- Replace too broad exceptions into the Core (#38344)
- Add CLI support for bulk pause and resume of DAGs (#38265)
- Implement methods on TaskInstancePydantic and DagRunPydantic (#38295, #38302, #38303, #38297)
- Made filters bar collapsible and add a full screen toggle (#38296)
- Encrypt all trigger attributes (#38233, #38358, #38743)
- Upgrade react-table package. Use with Audit Log table (#38092)
- Show if dag page filters are active (#38080)
- Add try number to mapped instance (#38097)
- Add retries to job heartbeat (#37541)
- Add REST API events to Audit Log (#37734)
- Make current working directory as templated field in BashOperator (#37968)
- Add calendar view to react (#37909)
- Add `run_id` column to log table (#37731)
- Add `tryNumber` to grid task instance tooltip (#37911)
- Session is not used in `_do_render_template_fields` (#37856)
- Improve MappedOperator property types (#37870)
- Remove provide_session decorator from TaskInstancePydantic methods (#37853)
- Ensure the “airflow.task” logger used for TaskInstancePydantic and TaskInstance (#37857)
- Better error message for internal api call error (#37852)
- Increase tooltip size of dag grid view (#37782) (#37805)
- Use named loggers instead of root logger (#37801)
- Add Run Duration in React (#37735)
- Avoid non-recommended usage of logging (#37792)
- Improve DateTimeTrigger typing (#37694)
- Make sure all unique `run_ids` render a task duration bar (#37717)
- Add Dag Audit Log to React (#37682)
- Add log event for auto pause (#38243)
- Better message for exception for templated base operator fields (#37668)
- Clean up webserver endpoints adding to audit log (#37580)
- Filter datasets graph by `dag_id` (#37464)
- Use new exception type inheriting BaseException for SIGTERMs (#37613)
- Refactor dataset class inheritance (#37590)
- Simplify checks for package versions (#37585)
- Filter Datasets by associated `dag_ids` (GET /datasets) (#37512)

- Enable “airflow tasks test” to run deferrable operator (#37542)
- Make datasets list/graph width adjustable (#37425)
- Speedup determine installed airflow version in `ExternalPythonOperator` (#37409)
- Add more task details from rest api (#37394)
- Add confirmation dialog box for DAG run actions (#35393)
- Added shutdown color to the STATE_COLORS (#37295)
- Remove legacy dag details page and redirect to grid (#37232)
- Order XCom entries by map index in API (#37086)
- Add `data_interval_start` and `data_interval_end` in `dagrun` create API endpoint (#36630)
- Making links in task logs as hyperlinks by preventing HTML injection (#36829)
- Improve `ExternalTaskSensor` Async Implementation (#36916)
- Make Datasets `Pathlike` (#36947)
- Simplify query for orphaned tasks (#36566)
- Add deferrable param in `FileSensor` (#36840)
- Run Trigger Page: Configurable number of recent configs (#36878)
- Merge `nowait` and `skip_locked` into `with_row_locks` (#36889)
- Return the specified field when get `dag/dagRun` in the REST API (#36641)
- Only iterate over the items if debug is enabled for `DagFileProcessorManager` (#36761)
- Add a fuzzy/regex pattern-matching for metric allow and block list (#36250)
- Allow custom columns in cli dags list (#35250)
- Make it possible to change the default cron timetable (#34851)
- Some improvements to Airflow IO code (#36259)
- Improve `TaskInstance` typing hints (#36487)
- Remove dependency of `Connexion` from auth manager interface (#36209)
- Refactor `ExternalDagLink` to not create ad hoc `TaskInstances` (#36135)

Bug Fixes

- Load providers configuration when gunicorn workers start (#38795)
- Fix grid header rendering (#38720)
- Add a task instance dependency for mapped dependencies (#37498)
- Improve stability of `remove_task_decorator` function (#38649)
- Mark more fields on API as dump-only (#38616)
- Fix `total_entries` count on the event logs endpoint (#38625)
- Add padding to bottom of log block. (#38610)
- Properly serialize nested attrs classes (#38591)
- Fixing the `tz` in next run ID info (#38482)

- Show abandoned tasks in Grid View (#38511)
- Apply task instance mutation hook consistently (#38440)
- Override chakra styles to keep dropdowns in filter bar (#38456)
- Store duration in seconds and scale to handle case when a value in the series has a larger unit than the preceding durations. (#38374)
- Don't allow defaults other than None in context parameters, and improve error message (#38015)
- Make postgresql default engine args comply with SA 2.0 (#38362)
- Add return statement to yield within a while loop in triggers (#38389)
- Ensure `__exit__` is called in decorator context managers (#38383)
- Make the method `BaseAuthManager.is_authorized_custom_view` abstract (#37915)
- Add upper limit to planned calendar events calculation (#38310)
- Fix Scheduler in daemon mode doesn't create PID at the specified location (#38117)
- Properly serialize `TaskInstancePydantic` and `DagRunPydantic` (#37855)
- Fix graph task state border color (#38084)
- Add back methods removed in security manager (#37997)
- Don't log "403" from worker serve-logs as "Unknown error". (#37933)
- Fix execution data validation error in `/get_logs_with_metadata` endpoint (#37756)
- Fix task duration selection (#37630)
- Refrain from passing `encoding` to the SQL engine in SQLAlchemy v2 (#37545)
- Fix 'implicitly coercing SELECT object to scalar subquery' in latest dag run statement (#37505)
- Clean up typing with `max_execution_date` query builder (#36958)
- Optimize `max_execution_date` query in single dag case (#33242)
- Fix list dags command for `get_dagmodel` is None (#36739)
- Load `consuming_dags` attr eagerly before dataset listener (#36247)

Miscellaneous

- Remove display of param from the UI (#38660)
- Update log level to debug from warning about `scheduled_duration` metric (#38180)
- Use `importlib_metadata` with compat to Python 3.10/3.12 stdlib (#38366)
- Refactored `__new__` magic method of `BaseOperatorMeta` to avoid bad mixing classic and decorated operators (#37937)
- Use `sys.version_info` for determine Python Major.Minor (#38372)
- Add missing deprecated Fab auth manager (#38376)
- Remove unused loop variable from airflow package (#38308)
- Adding max consecutive failed dag runs info in UI (#38229)
- Bump minimum version of `blinker` add where it requires (#38140)
- Bump follow-redirects from 1.15.4 to 1.15.6 in `/airflow/www` (#38156)

- Bump Cryptography to > 39.0.0 (#38112)
- Add Python 3.12 support (#36755, #38025, #36595)
- Avoid use of `assert` outside of the tests (#37718)
- Update ObjectStoragePath for universal_pathlib>=v0.2.2 (#37930)
- Resolve G004: Logging statement uses f-string (#37873)
- Update build and install dependencies. (#37910)
- Bump sanitize-html from 2.11.0 to 2.12.1 in /airflow/www (#37833)
- Update to latest installer versions. (#37754)
- Deprecate smtp configs in airflow settings / local_settings (#37711)
- Deprecate PY* constants into the airflow module (#37575)
- Remove usage of deprecated `flask._request_ctx_stack` (#37522)
- Remove redundant `login` attribute in `airflow.__init__.py` (#37565)
- Upgrade to FAB 4.3.11 (#37233)
- Remove SCHEDULED_DEPS which is no longer used anywhere since 2.0.0 (#37140)
- Replace `datetime.datetime.utcnow` by `airflow.utils.timezone.utcnow` in core (#35448)
- Bump aiohttp min version to avoid CVE-2024-23829 and CVE-2024-23334 (#37110)
- Move config related to FAB auth manager to FAB provider (#36232)
- Remove MSSQL support form Airflow core (#36514)
- Remove `is_authorized_cluster_activity` from auth manager (#36175)
- Create FAB provider and move FAB auth manager in it (#35926)

Doc Only Changes

- Improve timetable documentation (#38505)
- Reorder OpenAPI Spec tags alphabetically (#38717)
- Update UI screenshots in the documentation (#38680, #38403, #38438, #38435)
- Remove section as it's no longer true with dataset expressions PR (#38370)
- Refactor DatasetOrTimeSchedule timetable docs (#37771)
- Migrate executor docs to respective providers (#37728)
- Add directive to render a list of URI schemes (#37700)
- Add doc page with providers deprecations (#37075)
- Add a cross reference to security policy (#37004)
- Improve AIRFLOW__WEBSERVER__BASE_URL docs (#37003)
- Update faq.rst with (hopefully) clearer description of start_date (#36846)
- Update public interface doc re operators (#36767)
- Add exception to templates ref list (#36656)
- Add auth manager interface as public interface (#36312)

- Reference fab provider documentation in Airflow documentation (#36310)
- Create auth manager documentation (#36211)
- Update permission docs (#36120)
- Docstring improvement to `_covers_every_hour` (#36081)
- Add note that task instance, dag and lifecycle listeners are non-experimental (#36376)

3.16.12 Airflow 2.8.4 (2024-03-25)

Significant Changes

No significant changes.

Bug Fixes

- Fix incorrect serialization of `FixedTimezone` (#38139)
- Fix excessive permission changing for log task handler (#38164)
- Fix task instances list link (#38096)
- Fix a bug where scheduler heartrate parameter was not used (#37992)
- Add padding to prevent grid horizontal scroll overlapping tasks (#37942)
- Fix hash caching in `ObjectStoragePath` (#37769)

Miscellaneous

- Limit `importlib_resources` as it breaks `pytest_rewrites` (#38095, #38139)
- Limit `pandas` to <2.2 (#37748)
- Bump `croniter` to fix an issue with 29 Feb cron expressions (#38198)

Doc Only Changes

- Tell users what to do if their scanners find issues in the image (#37652)
- Add a section about debugging in Docker Compose with PyCharm (#37940)
- Update deferrable docs to clarify kwargs when trigger resumes operator (#38122)

3.16.13 Airflow 2.8.3 (2024-03-11)

Significant Changes

The `smtp` provider is now pre-installed when you install Airflow. (#37713)

Bug Fixes

- Add “MENU” permission in auth manager (#37881)
- Fix `external_executor_id` being overwritten (#37784)
- Make more `MappedOperator` members modifiable (#37828)
- Set parsing context `dag_id` in `dag test` command (#37606)

Miscellaneous

- Remove useless methods from security manager (#37889)
- Improve code coverage for TriggerRuleDep (#37680)
- The SMTP provider is now preinstalled when installing Airflow (#37713)
- Bump min versions of openapi validators (#37691)
- Properly include `airflow_pre_installed_providers.txt` artifact (#37679)

Doc Only Changes

- Clarify lack of sync between workers and scheduler (#37913)
- Simplify some docs around `airflow_local_settings` (#37835)
- Add section about local settings configuration (#37829)
- Fix docs of `BranchDayOfWeekOperator` (#37813)
- Write to secrets store is not supported by design (#37814)
- ERD generating doc improvement (#37808)
- Update incorrect config value (#37706)
- Update security model to clarify Connection Editing user's capabilities (#37688)
- Fix ImportError on examples dags (#37571)

3.16.14 Airflow 2.8.2 (2024-02-26)

Significant Changes

The `allowed_deserialization_classes` flag now follows a glob pattern (#36147).

For example if one wants to add the class `airflow.tests.custom_class` to the `allowed_deserialization_classes` list, it can be done by writing the full class name (`airflow.tests.custom_class`) or a pattern such as the ones used in glob search (e.g., `airflow.*`, `airflow.tests.*`).

If you currently use a custom regexp path make sure to rewrite it as a glob pattern.

Alternatively, if you still wish to match it as a regexp pattern, add it under the new list `allowed_deserialization_classes_regex` instead.

The `audit_logs` permissions have been updated for heightened security (#37501).

This was done under the policy that we do not want users like Viewer, Ops, and other users apart from Admin to have access to `audit_logs`. The intention behind this change is to restrict users with less permissions from viewing user details like First Name, Email etc. from the `audit_logs` when they are not permitted to.

The impact of this change is that the existing users with non admin rights won't be able to view or access the `audit_logs`, both from the Browse tab or from the DAG run.

AirflowTimeoutError is no longer except by default through Exception (#35653).

The `AirflowTimeoutError` is now inheriting `BaseException` instead of `AirflowException->``Exception```. See <https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

This prevents code catching `Exception` from accidentally catching `AirflowTimeoutError` and continuing to run. `AirflowTimeoutError` is an explicit intent to cancel the task, and should not be caught in attempts to handle the error and return some default value.

Catching `AirflowTimeoutError` is still possible by explicitly `except``ing ``AirflowTimeoutError` or `BaseException`. This is discouraged, as it may allow the code to continue running even after such cancellation requests. Code that previously depended on performing strict cleanup in every situation after catching `Exception` is advised to use `finally` blocks or context managers. To perform only the cleanup and then automatically re-raise the exception. See similar considerations about catching `KeyboardInterrupt` in <https://docs.python.org/3/library/exceptions.html#KeyboardInterrupt>

Bug Fixes

- Sort dag processing stats by last_runtime (#37302)
- Allow pre-population of trigger form values via URL parameters (#37497)
- Base date for fetching dag grid view must include selected run_id (#34887)
- Check permissions for ImportError (#37468)
- Move IMPORT_ERROR from DAG related permissions to view related permissions (#37292)
- Change AirflowTaskTimeout to inherit BaseException (#35653)
- Revert “Fix future DagRun rarely triggered by race conditions when max_active_runs reached its upper limit. (#31414)” (#37596)
- Change margin to padding so first task can be selected (#37527)
- Fix Airflow serialization for namedtuple (#37168)
- Fix bug with clicking url-unsafe tags (#37395)
- Set deterministic and new getter for Treeview function (#37162)
- Fix permissions of parent folders for log file handler (#37310)
- Fix permission check on DAGs when access_entity is specified (#37290)
- Fix the value of dateTimeAttrFormat constant (#37285)
- Resolve handler close race condition at triggerer shutdown (#37206)
- Fixing status icon alignment for various views (#36804)
- Remove superfluous @Sentry.enrich_errors (#37002)
- Use execution_date= param as a backup to base date for grid view (#37018)
- Handle SystemExit raised in the task. (#36986)
- Revoking audit_log permission from all users except admin (#37501)
- Fix broken regex for allowed_deserialization_classes (#36147)
- Fix the bug that affected the DAG end date. (#36144)
- Adjust node width based on task name length (#37254)
- fix: PythonVirtualenvOperator crashes if any python_callable function is defined in the same source as DAG (#37165)
- Fix collapsed grid width, line up selected bar with gantt (#37205)
- Adjust graph node layout (#37207)

- Revert the sequence of initializing configuration defaults (#37155)
- Displaying “actual” try number in TaskInstance view (#34635)
- Bugfix Triggering DAG with parameters is mandatory when `show_trigger_form_if_no_params` is enabled (#37063)
- Secret masker ignores passwords with special chars (#36692)
- Fix DagRuns with UPSTREAM_FAILED tasks get stuck in the backfill. (#36954)
- Disable dryrun auto-fetch (#36941)
- Fix copy button on a DAG run’s config (#36855)
- Fix bug introduced by replacing spaces by + in `run_id` (#36877)
- Fix webserver always redirecting to home page if user was not logged in (#36833)
- REST API set description on POST to `/variables` endpoint (#36820)
- Sanitize the `conn_id` to disallow potential script execution (#32867)
- Fix task id copy button copying wrong id (#34904)
- Fix security manager inheritance in fab provider (#36538)
- Avoid `pendulum.from_timestamp` usage (#37160)

Miscellaneous

- Install latest docker CLI instead of specific one (#37651)
- Bump `undici` from 5.26.3 to 5.28.3 in `/airflow/www` (#37493)
- Add Python 3.12 exclusions in `providers/pyproject.toml` (#37404)
- Remove `markdown` from core dependencies (#37396)
- Remove unused `pageSize` method. (#37319)
- Add more `itertools` as dependency of `common-sql` (#37359)
- Replace other Python 3.11 and 3.12 deprecations (#37478)
- Include `airflow_pre_installed_providers.txt` into `sdist` distribution (#37388)
- Turn Pydantic into an optional dependency (#37320)
- Limit `universal-pathlib` to < 0.2.0 (#37311)
- Allow running airflow against sqlite in-memory DB for tests (#37144)
- Add description to `queue_when` (#36997)
- Updated `config.yml` for environment variable `sqlalchemy_connect_args` (#36526)
- Bump min version of `Alembic` to 1.13.1 (#36928)
- Limit `flask-session` to < 0.6 (#36895)

Doc Only Changes

- Fix upgrade docs to reflect true CLI flags available (#37231)
- Fix a bug in fundamentals doc (#37440)
- Add redirect for deprecated page (#37384)

- Fix the `otel` config descriptions (#37229)
- Update `Objectstore` tutorial with `prereqs` section (#36983)
- Add more precise description on avoiding generic package/module names (#36927)
- Add airflow version substitution into Docker Compose Howto (#37177)
- Add clarification about DAG author capabilities to security model (#37141)
- Move docs for cron basics to Authoring and Scheduling section (#37049)
- Link to release notes in the upgrade docs (#36923)
- Prevent templated field logic checks in `__init__` of operators automatically (#33786)

3.16.15 Airflow 2.8.1 (2024-01-19)

Significant Changes

Target version for core dependency pendulum package set to 3 (#36281).

Support for `pendulum` 2.1.2 will be saved for a while, presumably until the next feature version of Airflow. It is advised to upgrade user code to use `pendulum` 3 as soon as possible.

`Pendulum` 3 introduced some subtle incompatibilities that you might rely on in your code - for example default rendering of dates is missing T in the rendered date representation, which is not ISO8601 compliant. If you rely on the default rendering of dates, you might need to adjust your code to use `isoformat()` method to render dates in ISO8601 format.

Airflow packaging specification follows modern Python packaging standards (#36537).

We standardized Airflow dependency configuration to follow latest development in Python packaging by using `pyproject.toml`. Airflow is now compliant with those accepted PEPs:

- [PEP-440 Version Identification and Dependency Specification](#)
- [PEP-517 A build-system independent format for source trees](#)
- [PEP-518 Specifying Minimum Build System Requirements for Python Projects](#)
- [PEP-561 Distributing and Packaging Type Information](#)
- [PEP-621 Storing project metadata in `pyproject.toml`](#)
- [PEP-660 Editable installs for `pyproject.toml` based builds \(wheel based\)](#)
- [PEP-685 Comparison of extra names for optional distribution dependencies](#)

Also we implement multiple license files support coming from Draft, not yet accepted (but supported by `hatchling`)
PEP: * [PEP 639 Improving License Clarity with Better Package Metadata](#)

This has almost no noticeable impact on users if they are using modern Python packaging and development tools, generally speaking Airflow should behave as it did before when installing it from PyPI and it should be much easier to install it for development purposes using `pip install -e ".[devel]"`.

The differences from the user side are:

- Airflow extras now get extras normalized to `-` (following PEP-685) instead of `_` and `.` (as it was before in some extras). When you install airflow with such extras (for example `dbt.core` or `all_dbs`) you should use `-` instead of `_` and `..`.

In most modern tools this will work in backwards-compatible way, but in some old version of those tools you might need to replace `_` and `.` with `-`. You can also get warnings that the extra you are installing does not exist - but usually this warning is harmless and the extra is installed anyway. It is, however, recommended to change to use `-` in extras in your dependency specifications for all Airflow extras.

- Released airflow package does not contain `devel`, `devel-*`, `doc` and `docs-gen` extras. Those extras are only available when you install Airflow from sources in `--editable` mode. This is because those extras are only used for development and documentation building purposes and are not needed when you install Airflow for production use. Those dependencies had unspecified and varying behaviour for released packages anyway and you were not supposed to use them in released packages.
- The `a11` and `a11-*` extras were not always working correctly when installing Airflow using constraints because they were also considered as development-only dependencies. With this change, those dependencies are now properly handling constraints and they will install properly with constraints, pulling the right set of providers and dependencies when constraints are used.

Graphviz dependency is now an optional one, not required one (#36647).

The `graphviz` dependency has been problematic as Airflow required dependency - especially for ARM-based installations. Graphviz packages require binary graphviz libraries - which is already a limitation, but they also require to install graphviz Python bindings to be build and installed. This does not work for older Linux installation but - more importantly - when you try to install Graphviz libraries for Python 3.8, 3.9 for ARM M1 MacBooks, the packages fail to install because Python bindings compilation for M1 can only work for Python 3.10+.

This is not a breaking change technically - the CLIs to render the DAGs is still there and IF you already have `graphviz` installed, it will continue working as it did before. The only problem when it does not work is where you do not have `graphviz` installed it will raise an error and inform that you need it.

Graphviz will remain to be installed for most users:

- the Airflow Image will still contain `graphviz` library, because it is added there as extra
- when previous version of Airflow has been installed already, then `graphviz` library is already installed there and Airflow will continue working as it did

The only change will be a new installation of new version of Airflow from the scratch, where `graphviz` will need to be specified as extra or installed separately in order to enable DAG rendering option.

Bug Fixes

- Fix airflow-scheduler exiting with code 0 on exceptions (#36800)
- Fix Callback exception when a removed task is the last one in the `taskinstance` list (#36693)
- Allow anonymous user edit/show resource when set `AUTH_ROLE_PUBLIC=admin` (#36750)
- Better error message when sqlite URL uses relative path (#36774)
- Explicit string cast required to force integer-type `run_ids` to be passed as strings instead of integers (#36756)
- Add log lookup exception for empty op subtypes (#35536)
- Remove unused index on task instance (#36737)
- Fix check on subclass for `typing.Union` in `_infer_multiple_outputs` for Python 3.10+ (#36728)
- Make sure `multiple_outputs` is inferred correctly even when using `TypedDict` (#36652)
- Add back FAB constant in legacy security manager (#36719)
- Fix AttributeError when using `Dagrun.update_state` (#36712)
- Do not let `EventsTimetable` schedule past events if `catchup=False` (#36134)
- Support encryption for triggers parameters (#36492)
- Fix the type hint for `tis_query` in `_process_executor_events` (#36655)
- Redirect to index when user does not have permission to access a page (#36623)

- Avoid using dict as default value in `call_regular_interval` (#36608)
- Remove option to set a task instance to running state in UI (#36518)
- Fix details tab not showing when using dynamic task mapping (#36522)
- Raise error when DagRun fails while running `dag test` (#36517)
- Refactor `_manage_executor_state` by refreshing TIs in batch (#36502)
- Add flask config: `MAX_CONTENT_LENGTH` (#36401)
- Fix `get_leaves` calculation for teardown in nested group (#36456)
- Stop serializing timezone-naive datetime to timezone-aware datetime with UTC tz (#36379)
- Make `kubernetes` decorator type annotation consistent with operator (#36405)
- Fix Webserver returning 500 for POST requests to `api/dag/*/dagrun` from anonymous user (#36275)
- Fix the required access for `get_variable` endpoint (#36396)
- Fix datetime reference in `DAG.is_fixed_time_schedule` (#36370)
- Fix AirflowSkipException message raised by BashOperator (#36354)
- Allow `PythonVirtualenvOperator.skip_on_exit_code` to be zero (#36361)
- Increase width of `execution_date` input in `trigger.html` (#36278)
- Fix logging for pausing DAG (#36182)
- Stop deserializing pickle when `enable_xcom_pickling` is False (#36255)
- Check DAG read permission before accessing DAG code (#36257)
- Enable mark task as failed/success always (#36254)
- Create latest log dir symlink as relative link (#36019)
- Fix Python-based decorators templating (#36103)

Miscellaneous

- Rename concurrency label to max active tasks (#36691)
- Restore function scoped `httpx` import in `file_task_handler` for performance (#36753)
- Add support of Pendulum 3 (#36281)
- Standardize airflow build process and switch to `hatchling` build backend (#36537)
- Get rid of `pyarrow-hotfix` for CVE-2023-47248 (#36697)
- Make `graphviz` dependency optional (#36647)
- Announce MSSQL support end in Airflow 2.9.0, add migration script hints (#36509)
- Set min `pandas` dependency to 1.2.5 for all providers and airflow (#36698)
- Bump follow-redirects from 1.15.3 to 1.15.4 in `/airflow/www` (#36700)
- Provide the `logger_name` param to `base_hook` in order to override the logger name (#36674)
- Fix run type icon alignment with run type text (#36616)
- Follow `BaseHook` connection fields method signature in `FSHook` (#36444)
- Remove redundant `docker` decorator type annotations (#36406)

- Straighten typing in workday timetable (#36296)
- Use `batch_is_authorized_dag` to check if user has permission to read DAGs (#36279)
- Replace deprecated `get_accessible_dag_ids` and use `get_readable_dags` in `get_dag_warnings` (#36256)

Doc Only Changes

- Metrics tagging documentation (#36627)
- In docs use `logical_date` instead of deprecated `execution_date` (#36654)
- Add section about live-upgrading Airflow (#36637)
- Replace `numpy` example with practical exercise demonstrating top-level code (#35097)
- Improve and add more complete description in the architecture diagrams (#36513)
- Improve the error message displayed when there is a webserver error (#36570)
- Update `dags.rst` with information on DAG pausing (#36540)
- Update installation prerequisites after upgrading to Debian Bookworm (#36521)
- Add description on the ways how users should approach DB monitoring (#36483)
- Add branching based on mapped task group example to `dynamic-task-mapping.rst` (#36480)
- Add further details to replacement documentation (#36485)
- Use cards when describing priority weighting methods (#36411)
- Update `metrics.rst` for param `dagrun.schedule_delay` (#36404)
- Update admonitions in Python operator doc to reflect sentiment (#36340)
- Improve `audit_logs.rst` (#36213)
- Remove Redshift mention from the list of managed Postgres backends (#36217)

3.16.16 Airflow 2.8.0 (2023-12-18)

Significant Changes

Raw HTML code in DAG docs and DAG params descriptions is disabled by default (#35460)

To ensure that no malicious javascript can be injected with DAG descriptions or trigger UI forms by DAG authors a new parameter `webserver.allow_raw_html_descriptions` was added with default value of `False`. If you trust your DAG authors code and want to allow using raw HTML in DAG descriptions and params, you can restore the previous behavior by setting the configuration value to `True`.

To ensure Airflow is secure by default, the raw HTML support in trigger UI has been super-seeded by markdown support via the `description_md` attribute. If you have been using `description_html` please migrate to `description_md`. The `custom_html_form` is now deprecated.

New Features

- AIP-58: Add Airflow ObjectStore (AFS) ([AIP-58](#))
- Add XCom tab to Grid (#35719)
- Add “literal” wrapper to disable field templating (#35017)
- Add task context logging feature to allow forwarding messages to task logs (#32646, #32693, #35857)
- Add Listener hooks for Datasets (#34418, #36247)

- Allow override of navbar text color (#35505)
- Add lightweight serialization for deltalake tables (#35462)
- Add support for serialization of iceberg tables (#35456)
- `prev_end_date_success` method access (#34528)
- Add task parameter to set custom logger name (#34964)
- Add pyspark decorator (#35247)
- Add trigger as a valid option for the db clean command (#34908)
- Add decorators for external and venv python branching operators (#35043)
- Allow PythonVenvOperator using other index url (#33017)
- Add Python Virtualenv Operator Caching (#33355)
- Introduce a generic export for containerized executor logging (#34903)
- Add ability to clear downstream tis in `List Task Instances` view (#34529)
- Attribute `clear_number` to track DAG run being cleared (#34126)
- Add BranchPythonVirtualenvOperator (#33356)
- Allow PythonVenvOperator using other index url (#33017)
- Add CLI notification commands to providers (#33116)
- Use dropdown instead of buttons when there are more than 10 retries in log tab (#36025)

Improvements

- Add `multiselect` to run state in grid view (#35403)
- Fix warning message in `Connection.get_hook` in case of `ImportError` (#36005)
- Add `processor_subdir` to `import_error` table to handle multiple dag processors (#35956)
- Consolidate the call of `change_state` to fail or success in the core executors (#35901)
- Relax mandatory requirement for `start_date` when `schedule=None` (#35356)
- Use `ExitStack` to manage mutation of `secrets_backend_list` in `dag.test` (#34620)
- improved visibility of tasks in `ActionModal` for `taskinstance` (#35810)
- Create directories based on `AIRFLOW_CONFIG` path (#35818)
- Implements JSON-string connection representation generator (#35723)
- Move `BaseOperatorLink` into the separate module (#35032)
- Set `mark_end_on_close` after `set_context` (#35761)
- Move external logs links to top of react logs page (#35668)
- Change terminal mode to `cbreak` in `execute_interactive` and handle `SIGINT` (#35602)
- Make raw HTML descriptions configurable (#35460)
- Allow email field to be templated (#35546)
- Hide logical date and run id in trigger UI form (#35284)
- Improved instructions for adding dependencies in TaskFlow (#35406)

- Add optional exit code to list import errors (#35378)
- Limit query result on DB rather than client in `synchronize_log_template` function (#35366)
- Allow description to be passed in when using variables CLI (#34791)
- Allow optional defaults in required fields with manual triggered dags (#31301)
- Permitting airflow kerberos to run in different modes (#35146)
- Refactor commands to unify daemon context handling (#34945)
- Add extra fields to plugins endpoint (#34913)
- Add description to pools view (#34862)
- Move cli's Connection export and Variable export command print logic to a separate function (#34647)
- Extract and reuse `get_kerberos_principle` func from `get_kerberos_principle` (#34936)
- Change type annotation for `BaseOperatorLink.operators` (#35003)
- Optimise and migrate to SA2-compatible syntax for `TaskReschedule` (#33720)
- Consolidate the permissions name in `SlaMissModelView` (#34949)
- Add debug log saying what's being run to `EventScheduler` (#34808)
- Increase log reader stream loop sleep duration to 1 second (#34789)
- Resolve pydantic deprecation warnings re `update_forward_refs` (#34657)
- Unify mapped task group lookup logic (#34637)
- Allow filtering event logs by attributes (#34417)
- Make connection login and password TEXT (#32815)
- Ban import `Dataset` from `airflow` package in codebase (#34610)
- Use `airflow.datasets.Dataset` in examples and tests (#34605)
- Enhance task status visibility (#34486)
- Simplify DAG trigger UI (#34567)
- Ban import `AirflowException` from `airflow` (#34512)
- Add descriptions for airflow resource config parameters (#34438)
- Simplify trigger name expression (#34356)
- Move definition of `Pod*Exceptions` to `pod_generator` (#34346)
- Add deferred tasks to the `cluster_activity` view Pools Slots (#34275)
- heartbeat failure log message fix (#34160)
- Rename variables for dag runs (#34049)
- Clarify `new_state` in OpenAPI spec (#34056)
- Remove `version` top-level element from docker compose files (#33831)
- Remove generic trigger cancelled error log (#33874)
- Use `NOT EXISTS` subquery instead of `tuple_not_in_condition` (#33527)
- Allow context key args to not provide a default (#33430)
- Order triggers by - TI priority_weight when assign unassigned triggers (#32318)

- Add metric `triggerer_heartbeat` (#33320)
- Allow `airflow variables export` to print to stdout (#33279)
- Workaround failing deadlock when running backfill (#32991)
- add `dag_run_ids` and `task_ids` filter for the batch task instance API endpoint (#32705)
- Configurable health check threshold for triggerer (#33089)
- Rework provider manager to treat Airflow core hooks like other provider hooks (#33051)
- Ensure DAG-level references are filled on unmap (#33083)
- Affix webserver access_denied warning to be configurable (#33022)
- Add support for arrays of different data types in the Trigger Form UI (#32734)
- Add a mechanism to warn if executors override existing CLI commands (#33423)

Bug Fixes

- Account for change in UTC offset when calculating next schedule (#35887)
- Add read access to pools for viewer role (#35352)
- Fix gantt chart queued duration when `queued_dttm` is greater than `start_date` for deferred tasks (#35984)
- Avoid crushing container when directory is not found on rm (#36050)
- Update `reset_user_sessions` to work from either CLI or web (#36056)
- Fix UI Grid error when DAG has been removed. (#36028)
- Change Trigger UI to use HTTP POST in web ui (#36026)
- Fix airflow db shell needing an extra key press to exit (#35982)
- Change dag grid `overscroll` behaviour to auto (#35717)
- Run triggers inline with dag test (#34642)
- Add `borderWidthRight` to grid for Firefox scrollbar (#35346)
- Fix for infinite recursion due to `secrets_masker` (#35048)
- Fix write `processor_subdir` in `serialized_dag` table (#35661)
- Reload configuration for standalone dag file processor (#35725)
- Long custom operator name overflows in graph view (#35382)
- Add `try_number` to extra links query (#35317)
- Prevent assignment of non JSON serializable values to `DagRun.conf dict` (#35096)
- Numeric values in DAG details are incorrectly rendered as timestamps (#35538)
- Fix Scheduler and triggerer crashes in daemon mode when statsd metrics are enabled (#35181)
- Infinite UI redirection loop after deactivating an active user (#35486)
- Bug fix `fetch_callback` of Partial Subset DAG (#35256)
- Fix `DagRun` data interval for `DeltaDataIntervalTimetable` (#35391)
- Fix query in `get_dag_by_pickle` util function (#35339)
- Fix `TriggerDagRunOperator` failing to trigger subsequent runs when `reset_dag_run=True` (#35429)

- Fix weight_rule property type in `mappedoperator` (#35257)
- Bugfix/prevent concurrency with cached venv (#35258)
- Fix dag serialization (#34042)
- Fix py/url-redirection by replacing `request.referrer` by `get_redirect()` (#34237)
- Fix updating variables during variable imports (#33932)
- Use Literal from `airflow.typing_compat` in Airflow core (#33821)
- Always use Literal from `typing_extensions` (#33794)

Miscellaneous

- Change default MySQL client to MariaDB (#36243)
- Mark daskexecutor provider as removed (#35965)
- Bump FAB to 4.3.10 (#35991)
- Mark daskexecutor provider as removed (#35965)
- Rename `Connection.to_json_dict` to `Connection.to_dict` (#35894)
- Upgrade to Pydantic v2 (#35551)
- Bump moto version to >= 4.2.9 (#35687)
- Use `pyarrow-hotfix` to mitigate CVE-2023-47248 (#35650)
- Bump axios from 0.26.0 to 1.6.0 in /airflow/www/ (#35624)
- Make docker decorator's type annotation consistent with operator (#35568)
- Add default to `navbar_text_color` and `rm` condition in style (#35553)
- Avoid initiating session twice in `dag_next_execution` (#35539)
- Work around typing issue in examples and providers (#35494)
- Enable TCH004 and TCH005 rules (#35475)
- Humanize log output about retrieved DAG(s) (#35338)
- Switch from Black to Ruff formatter (#35287)
- Upgrade to Flask Application Builder 4.3.9 (#35085)
- D401 Support (#34932, #34933)
- Use `requires_access` to check read permission on dag instead of checking it explicitly (#34940)
- Deprecate lazy import `AirflowException` from airflow (#34541)
- View util refactoring on mapped stuff use cases (#34638)
- Bump postcss from 8.4.25 to 8.4.31 in /airflow/www (#34770)
- Refactor Sqlalchemy queries to 2.0 style (#34763, #34665, #32883, #35120)
- Change to lazy loading of io in pandas serializer (#34684)
- Use `airflow.models.dag.DAG` in examples (#34617)
- Use `airflow.exceptions.AirflowException` in core (#34510)
- Check that `dag_ids` passed in request are consistent (#34366)

- Refactors to make code better (#34278, #34113, #34110, #33838, #34260, #34409, #34377, #34350)
- Suspend qubole provider (#33889)
- Generate Python API docs for Google ADS (#33814)
- Improve importing in modules (#33812, #33811, #33810, #33806, #33807, #33805, #33804, #33803, #33801, #33799, #33800, #33797, #33798, #34406, #33808)
- Upgrade Elasticsearch to 8 (#33135)

Doc Only Changes

- Add support for tabs (and other UX components) to docs (#36041)
- Replace architecture diagram of Airflow with diagrams-generated one (#36035)
- Add the section describing the security model of DAG Author capabilities (#36022)
- Enhance docs for zombie tasks (#35825)
- Reflect drop/add support of DB Backends versions in documentation (#35785)
- More detail on mandatory task arguments (#35740)
- Indicate usage of the `re2regex` engine in the `.airflowignore` documentation. (#35663)
- Update `best-practices.rst` (#35692)
- Update `dag-run.rst` to mention Airflow's support for extended cron syntax through croniter (#35342)
- Update `webserver.rst` to include information of supported OAuth2 providers (#35237)
- Add back `dag_run` to docs (#35142)
- Fix `rst` code block format (#34708)
- Add typing to concrete taskflow examples (#33417)
- Add concrete examples for accessing context variables from TaskFlow tasks (#33296)
- Fix links in security docs (#33329)

3.16.17 Airflow 2.7.3 (2023-11-06)

Significant Changes

No significant changes.

Bug Fixes

- Fix pre-mature evaluation of tasks in mapped task group (#34337)
- Add TriggerRule missing value in rest API (#35194)
- Fix Scheduler crash looping when dagrun creation fails (#35135)
- Fix test connection with `codemirror` and extra (#35122)
- Fix usage of cron-descriptor since BC in v1.3.0 (#34836)
- Fix `get_plugin_info` for class based listeners. (#35022)
- Some improvements/fixes for `dag_run` and `task_instance` endpoints (#34942)
- Fix the dags count filter in webserver home page (#34944)

- Return only the TIs of the readable dags when ~ is provided as a dag_id (#34939)
- Fix triggerer thread crash in daemon mode (#34931)
- Fix wrong plugin schema (#34858)
- Use DAG timezone in TimeSensorAsync (#33406)
- Mark tasks with all_skipped trigger rule as skipped if any task is in upstream_failed state (#34392)
- Add read only validation to read only fields (#33413)

Misc/Internal

- Improve testing harness to separate DB and non-DB tests (#35160, #35333)
- Add pytest db_test markers to our tests (#35264)
- Add pip caching for faster build (#35026)
- Upper bound pendulum requirement to <3.0 (#35336)
- Limit sentry_sdk to 1.33.0 (#35298)
- Fix subtle bug in mocking processor_agent in our tests (#35221)
- Bump @babel/traverse from 7.16.0 to 7.23.2 in /airflow/www (#34988)
- Bump undici from 5.19.1 to 5.26.3 in /airflow/www (#34971)
- Remove unused set from SchedulerJobRunner (#34810)
- Remove warning about max_tis per query > parallelism (#34742)
- Improve modules import in Airflow core by moving some of them into a type-checking block (#33755)
- Fix tests to respond to Python 3.12 handling of utcnow in sentry-sdk (#34946)
- Add connexion<3.0 upper bound (#35218)
- Limit Airflow to < 3.12 (#35123)
- update moto version (#34938)
- Limit WTForms to below 3.1.0 (#34943)

Doc Only Changes

- Fix variables substitution in Airflow Documentation (#34462)
- Added example for defaults in conn.extras (#35165)
- Update datasets.rst issue with running example code (#35035)
- Remove mysql-connector-python from recommended MySQL driver (#34287)
- Fix syntax error in task dependency set_downstream example (#35075)
- Update documentation to enable test connection (#34905)
- Update docs errors.rst - Mention sentry “transport” configuration option (#34912)
- Update dags.rst to put SubDag deprecation note right after the SubDag section heading (#34925)
- Add info on getting variables and config in custom secrets backend (#34834)
- Document BaseExecutor interface in more detail to help users in writing custom executors (#34324)
- Fix broken link to airflow_local_settings.py template (#34826)

- Fixes python_callable function assignment context kwargs example in params.rst (#34759)
- Add missing multiple_outputs=True param in the TaskFlow example (#34812)
- Remove extraneous '>' in provider section name (#34813)
- Fix imports in extra link documentation (#34547)

3.16.18 Airflow 2.7.2 (2023-10-12)

Significant Changes

No significant changes

Bug Fixes

- Check if the lower of provided values are sensitives in config endpoint (#34712)
- Add support for ZoneInfo and generic UTC to fix datetime serialization (#34683, #34804)
- Fix AttributeError: ‘Select’ object has no attribute ‘count’ during the airflow db migrate command (#34348)
- Make dry run optional for patch task instance (#34568)
- Fix non deterministic datetime deserialization (#34492)
- Use iterative loop to look for mapped parent (#34622)
- Fix is_parent_mapped value by checking if any of the parent taskgroup is mapped (#34587)
- Avoid top-level airflow import to avoid circular dependency (#34586)
- Add more exemptions to lengthy metric list (#34531)
- Fix dag warning endpoint permissions (#34355)
- Fix task instance access issue in the batch endpoint (#34315)
- Correcting wrong time showing in grid view (#34179)
- Fix www cluster_activity view not loading due to standaloneDagProcessor templating (#34274)
- Set loglevel=DEBUG in ‘Not syncing DAG-level permissions’ (#34268)
- Make param validation consistent for DAG validation and triggering (#34248)
- Ensure details panel is shown when any tab is selected (#34136)
- Fix issues related to access_control={} (#34114)
- Fix not found ab_user table in the CLI session (#34120)
- Fix FAB-related logging format interpolation (#34139)
- Fix query bug in next_run_datasets_summary endpoint (#34143)
- Fix for TaskGroup toggles for duplicated labels (#34072)
- Fix the required permissions to clear a TI from the UI (#34123)
- Reuse _run_task_session in mapped render_template_fields (#33309)
- Fix scheduler logic to plan new dag runs by ignoring manual runs (#34027)
- Add missing audit logs for Flask actions add, edit and delete (#34090)
- Hide Irrelevant Dag Processor from Cluster Activity Page (#33611)
- Remove infinite animation for pinwheel, spin for 1.5s (#34020)

- Restore rendering of provider configuration with `version_added` (#34011)

Doc Only Changes

- Clarify audit log permissions (#34815)
- Add explanation for Audit log users (#34814)
- Import `AUTH_REMOTE_USER` from FAB in WSGI middleware example (#34721)
- Add information about drop support MsSQL as DB Backend in the future (#34375)
- Document how to use the system's timezone database (#34667)
- Clarify what landing time means in doc (#34608)
- Fix screenshot in dynamic task mapping docs (#34566)
- Fix class reference in Public Interface documentation (#34454)
- Clarify `var.value.get` and `var.json.get` usage (#34411)
- Schedule default value description (#34291)
- Docs for triggered_dataset_event (#34410)
- Add DagRun events (#34328)
- Provide tabular overview about trigger form param types (#34285)
- Add link to Amazon Provider Configuration in Core documentation (#34305)
- Add “security infrastructure” paragraph to security model (#34301)
- Change links to SQLAlchemy 1.4 (#34288)
- Add SBOM entry in security documentation (#34261)
- Added more example code for XCom push and pull (#34016)
- Add state utils to Public Airflow Interface (#34059)
- Replace markdown style link with rst style link (#33990)
- Fix broken link to the “UPDATING.md” file (#33583)

Misc/Internal

- Update min-sqlalchemy version to account for latest features used (#34293)
- Fix SessionExemptMixin spelling (#34696)
- Restrict `astroid` version < 3 (#34658)
- Fail dag test if defer without triggerer (#34619)
- Fix connections exported output (#34640)
- Don't run isort when creating new alembic migrations (#34636)
- Deprecate numeric type python version in PythonVirtualEnvOperator (#34359)
- Refactor `os.path.splitext` to `Path.*` (#34352, #33669)
- Replace `= by is` for type comparison (#33983)
- Refactor integer division (#34180)
- Refactor: Simplify comparisons (#34181)

- Refactor: Simplify string generation (#34118)
- Replace unnecessary dict comprehension with dict() in core (#33858)
- Change “not all” to “any” for ease of readability (#34259)
- Replace assert by if...raise in code (#34250, #34249)
- Move default timezone to except block (#34245)
- Combine similar if logic in core (#33988)
- Refactor: Consolidate import and usage of random (#34108)
- Consolidate importing of os.path.* (#34060)
- Replace sequence concatenation by unpacking in Airflow core (#33934)
- Refactor unneeded ‘continue’ jumps around the repo (#33849, #33845, #33846, #33848, #33839, #33844, #33836, #33842)
- Remove [project] section from `pyproject.toml` (#34014)
- Move the try outside the loop when this is possible in Airflow core (#33975)
- Replace loop by any when looking for a positive value in core (#33985)
- Do not create lists we don’t need (#33519)
- Remove useless string join from core (#33969)
- Add TCH001 and TCH002 rules to pre-commit to detect and move type checking modules (#33865)
- Add cancel_trigger_ids to to_cancel dequeue in batch (#33944)
- Avoid creating unnecessary list when parsing stats datadog tags (#33943)
- Replace dict.items by dict.values when key is not used in core (#33940)
- Replace lambdas with comprehensions (#33745)
- Improve modules import in Airflow core by some of them into a type-checking block (#33755)
- Refactor: remove unused state - SHUTDOWN (#33746, #34063, #33893)
- Refactor: Use in-place `.sort()` (#33743)
- Use literal dict instead of calling dict() in Airflow core (#33762)
- remove unnecessary map and rewrite it using list in Airflow core (#33764)
- Replace lambda by a def method in Airflow core (#33758)
- Replace type func by `isinstance` in fab_security manager (#33760)
- Replace single quotes by double quotes in all Airflow modules (#33766)
- Merge multiple `isinstance` calls for the same object in a single call (#33767)
- Use a single statement with multiple contexts instead of nested statements in core (#33769)
- Refactor: Use f-strings (#33734, #33455)
- Refactor: Use `random.choices` (#33631)
- Use `str.splitlines()` to split lines (#33592)
- Refactor: Remove useless str() calls (#33629)
- Refactor: Improve detection of duplicates and list sorting (#33675)

- Simplify conditions on `len()` (#33454)

3.16.19 Airflow 2.7.1 (2023-09-07)

Significant Changes

CronTriggerTimetable is now less aggressive when trying to skip a run (#33404)

When setting `catchup=False`, `CronTriggerTimetable` no longer skips a run if the scheduler does not query the timetable immediately after the previous run has been triggered.

This should not affect scheduling in most cases, but can change the behaviour if a DAG is paused-unpaused to manually skip a run. Previously, the timetable (with `catchup=False`) would only start a run after a DAG is unpause, but with this change, the scheduler would try to look at little bit back to schedule the previous run that covers a part of the period when the DAG was paused. This means you will need to keep a DAG paused longer (namely, for the entire cron period to pass) to really skip a run.

Note that this is also the behaviour exhibited by various other cron-based scheduling tools, such as anacron.

conf.set() becomes case insensitive to match conf.get() behavior (#33452)

Also, `conf.get()` will now break if used with non-string parameters.

`conf.set(section, key, value)` used to be case sensitive, i.e. `conf.set("SECTION", "KEY", value)` and `conf.set("section", "key", value)` were stored as two distinct configurations. This was inconsistent with the behavior of `conf.get(section, key)`, which was always converting the section and key to lower case.

As a result, configuration options set with upper case characters in the section or key were unreachable. That's why we are now converting section and key to lower case in `conf.set` too.

We also changed a bit the behavior of `conf.get()`. It used to allow objects that are not strings in the section or key. Doing this will now result in an exception. For instance, `conf.get("section", 123)` needs to be replaced with `conf.get("section", "123")`.

Bug Fixes

- Ensure that tasks wait for running indirect setup (#33903)
- Respect “soft_fail” for core async sensors (#33403)
- Differentiate 0 and unset as a default param values (#33965)
- Raise 404 from Variable PATCH API if variable is not found (#33885)
- Fix MappedTaskGroup tasks not respecting upstream dependency (#33732)
- Add limit 1 if required first value from query result (#33672)
- Fix UI DAG counts including deleted DAGs (#33778)
- Fix cleaning zombie RESTARTING tasks (#33706)
- SECURITY_MANAGER_CLASS should be a reference to class, not a string (#33690)
- Add back `get_url_for_login` in security manager (#33660)
- Fix 2.7.0 db migration job errors (#33652)
- Set context inside templates (#33645)
- Treat dag-defined access_control as authoritative if defined (#33632)
- Bind engine before attempting to drop archive tables (#33622)

- Add a fallback in case no first name and last name are set (#33617)
- Sort data before groupby in TIS duration calculation (#33535)
- Stop adding values to rendered templates UI when there is no dagrun (#33516)
- Set strict to True when parsing dates in webserver views (#33512)
- Use `dialect.name` in custom SA types (#33503)
- Do not return ongoing dagrun when a `end_date` is less than `utcnow` (#33488)
- Fix a bug in `formatDuration` method (#33486)
- Make `conf.set` case insensitive (#33452)
- Allow timetable to slightly miss catchup cutoff (#33404)
- Respect `soft_fail` argument when `poke` is called (#33401)
- Create a new method used to resume the task in order to implement specific logic for operators (#33424)
- Fix DagFileProcessor interfering with dags outside its `processor_subdir` (#33357)
- Remove the unnecessary `
` text in Provider's view (#33326)
- Respect `soft_fail` argument when ExternalTaskSensor runs in deferrable mode (#33196)
- Fix handling of default value and serialization of Param class (#33141)
- Check if the dynamically-added index is in the table schema before adding (#32731)
- Fix rendering the mapped parameters when using `expand_kwargs` method (#32272)
- Fix dependencies for celery and opentelemetry for Python 3.8 (#33579)

Misc/Internal

- Bring back Pydantic 1 compatibility (#34081, #33998)
- Use a trimmed version of README.md for PyPI (#33637)
- Upgrade to Pydantic 2 (#33956)
- Reorganize `devel_only` extra in Airflow's setup.py (#33907)
- Bumping FAB to 4.3.4 in order to fix issues with filters (#33931)
- Add minimum requirement for `sqlalchemy` to 1.4.24 (#33892)
- Update `version_added` field for configs in config file (#33509)
- Replace `OrderedDict` with plain dict (#33508)
- Consolidate import and usage of `itertools` (#33479)
- Static check fixes (#33462)
- Import `utc` from `datetime` and normalize its import (#33450)
- D401 Support (#33352, #33339, #33337, #33336, #33335, #33333, #33338)
- Fix some missing type hints (#33334)
- D205 Support - Stragglers (#33301, #33298, #33297)
- Refactor: Simplify code (#33160, #33270, #33268, #33267, #33266, #33264, #33292, #33453, #33476, #33567, #33568, #33480, #33753, #33520, #33623)
- Fix Pydantic warning about `orm_mode` rename (#33220)

- Add MySQL 8.1 to supported versions. (#33576)
- Remove Pydantic limitation for version < 2 (#33507)

Doc only changes

- Add documentation explaining template_ext (and how to override it) (#33735)
- Explain how users can check if python code is top-level (#34006)
- Clarify that DAG authors can also run code in DAG File Processor (#33920)
- Fix broken link in Modules Management page (#33499)
- Fix secrets backend docs (#33471)
- Fix config description for base_log_folder (#33388)

3.16.20 Airflow 2.7.0 (2023-08-18)

Significant Changes

Remove Python 3.7 support (#30963)

As of now, Python 3.7 is no longer supported by the Python community. Therefore, to use Airflow 2.7.0, you must ensure your Python version is either 3.8, 3.9, 3.10, or 3.11.

Old Graph View is removed (#32958)

The old Graph View is removed. The new Graph View is the default view now.

The trigger UI form is skipped in web UI if no parameters are defined in a DAG (#33351)

If you are using `dag_run.conf` dictionary and web UI JSON entry to run your DAG you should either:

- Add params to your DAG
- Enable the new configuration `show_trigger_form_if_no_params` to bring back old behaviour

The “db init”, “db upgrade” commands and “[database] load_default_connections” configuration options are deprecated (#33136).

Instead, you should use “airflow db migrate” command to create or upgrade database. This command will not create default connections. In order to create default connections you need to run “airflow connections create-default-connections” explicitly, after running “airflow db migrate”.

In case of SMTP SSL connection, the context now uses the “default” context (#33070)

The “default” context is Python’s `default_ssl_context` instead of previously used “none”. The `default_ssl_context` provides a balance between security and compatibility but in some cases, when certificates are old, self-signed or misconfigured, it might not work. This can be configured by setting “ssl_context” in “email” configuration of Airflow.

Setting it to “none” brings back the “none” setting that was used in Airflow 2.6 and before, but it is not recommended due to security reasons as this setting disables validation of certificates and allows MITM attacks.

Disable default allowing the testing of connections in UI, API and CLI (#32052)

For security reasons, the test connection functionality is disabled by default across Airflow UI, API and CLI. The availability of the functionality can be controlled by the `test_connection` flag in the `core` section of the Airflow configuration (`airflow.cfg`). It can also be controlled by the environment variable `AIRFLOW__CORE__TEST_CONNECTION`.

The following values are accepted for this config param: 1. **Disabled**: Disables the test connection functionality and disables the Test Connection button in the UI.

This is also the default value set in the Airflow configuration. 2. **Enabled**: Enables the test connection functionality and activates the Test Connection button in the UI.

3. **Hidden**: Disables the test connection functionality and hides the Test Connection button in UI.

For more information on capabilities of users, see the documentation: https://airflow.apache.org/docs/apache-airflow/stable/security/security_model.html#capabilities-of-authenticated-ui-users It is strongly advised to **not** enable the feature until you make sure that only highly trusted UI/API users have “edit connection” permissions.

The `xcomEntries` API disables support for the `deserialize` flag by default (#32176)

For security reasons, the `/dags/*/dagRuns/*/taskInstances/*/xcomEntries/*` API endpoint now disables the `deserialize` option to deserialize arbitrary XCom values in the webserver. For backward compatibility, server admins may set the `[api] enable_xcom_deserialize_support` config to `True` to enable the flag and restore backward compatibility.

However, it is strongly advised to **not** enable the feature, and perform deserialization at the client side instead.

Change of the default Celery application name (#32526)

Default name of the Celery application changed from `airflow.executors.celery_executor` to `airflow.providers.celery.executors.celery_executor`.

You should change both your configuration and Health check command to use the new name:

- in configuration (`celery_app_name` configuration in `celery` section) use `airflow.providers.celery.executors.celery_executor`
- in your Health check command use `airflow.providers.celery.executors.celery_executor.app`

The default value for `scheduler.max_tis_per_query` is changed from 512 to 16 (#32572)

This change is expected to make the Scheduler more responsive.

`scheduler.max_tis_per_query` needs to be lower than `core.parallelism`. If both were left to their default value previously, the effective default value of `scheduler.max_tis_per_query` was 32 (because it was capped at `core.parallelism`).

To keep the behavior as close as possible to the old config, one can set `scheduler.max_tis_per_query = 0`, in which case it'll always use the value of `core.parallelism`.

Some executors have been moved to corresponding providers (#32767)

In order to use the executors, you need to install the providers:

- for Celery executors you need to install `apache-airflow-providers-celery` package $\geq 3.3.0$
- for Kubernetes executors you need to install `apache-airflow-providers-cncf-kubernetes` package $\geq 7.4.0$
- For Dask executors you need to install `apache-airflow-providers-daskexecutor` package in any version

You can achieve it also by installing airflow with `[celery]`, `[cncf.kubernetes]`, `[daskexecutor]` extras respectively.

Users who base their images on the `apache/airflow` reference image (not slim) should be unaffected - the base reference image comes with all the three providers installed.

Improvement Changes

PostgreSQL only improvement: Added index on taskinstance table (#30762)

This index seems to have great positive effect in a setup with tens of millions such rows.

New Features

- Add OpenTelemetry to Airflow ([AIP-49](#))
- Trigger Button - Implement Part 2 of AIP-50 (#31583)
- Removing Executor Coupling from Core Airflow ([AIP-51](#))
- Automatic setup and teardown tasks ([AIP-52](#))
- OpenLineage in Airflow ([AIP-53](#))
- Experimental: Add a cache to Variable and Connection when called at dag parsing time (#30259)
- Enable pools to consider deferred tasks (#32709)
- Allows to choose SSL context for SMTP connection (#33070)
- New gantt tab (#31806)
- Load plugins from providers (#32692)
- Add `BranchExternalPythonOperator` (#32787, #33360)
- Add option for storing configuration description in providers (#32629)
- Introduce Heartbeat Parameter to Allow Per-LocalTaskJob Configuration (#32313)
- Add Executors discovery and documentation (#32532)
- Add JobState for job state constants (#32549)
- Add config to disable the ‘deserialize’ XCom API flag (#32176)
- Show task instance in web UI by custom operator name (#31852)
- Add `default_deferrable` config (#31712)
- Introducing `AirflowClusterPolicySkipDag` exception (#32013)
- Use `reactflow` for datasets graph (#31775)
- Add an option to load the dags from db for command tasks run (#32038)
- Add version of `chain` which doesn’t require matched lists (#31927)
- Use `operator_name` instead of `task_type` in UI (#31662)
- Add `--retry` and `--retry-delay` to `airflow db check` (#31836)
- Allow skipped task state `task_instance_schema.py` (#31421)
- Add a new config for celery result_backend engine options (#30426)
- UI Add Cluster Activity Page (#31123, #32446)

- Adding keyboard shortcuts to common actions (#30950)
- Adding more information to kubernetes executor logs (#29929)
- Add support for configuring custom alembic file (#31415)
- Add running and failed status tab for DAGs on the UI (#30429)
- Add multi-select, proposals and labels for trigger form (#31441)
- Making webserver config customizable (#29926)
- Render DAGCode in the Grid View as a tab (#31113)
- Add rest endpoint to get option of configuration (#31056)
- Add section query param in get config rest API (#30936)
- Create metrics to track Scheduled->Queued->Running task state transition times (#30612)
- Mark Task Groups as Success/Failure (#30478)
- Add CLI command to list the provider trigger info (#30822)
- Add Fail Fast feature for DAGs (#29406)

Improvements

- Improve graph nesting logic (#33421)
- Configurable health check threshold for triggerer (#33089, #33084)
- add dag_run_ids and task_ids filter for the batch task instance API endpoint (#32705)
- Ensure DAG-level references are filled on unmap (#33083)
- Add support for arrays of different data types in the Trigger Form UI (#32734)
- Always show gantt and code tabs (#33029)
- Move listener success hook to after SQLAlchemy commit (#32988)
- Rename db upgrade to db migrate and add connections create-default-connections (#32810, #33136)
- Remove old gantt chart and redirect to grid views gantt tab (#32908)
- Adjust graph zoom based on selected task (#32792)
- Call listener on_task_instance_running after rendering templates (#32716)
- Display execution_date in graph view task instance tooltip. (#32527)
- Allow configuration to be contributed by providers (#32604, #32755, #32812)
- Reduce default for max TIs per query, enforce <= parallelism (#32572)
- Store config description in Airflow configuration object (#32669)
- Use isdisjoint instead of not intersection (#32616)
- Speed up calculation of leaves and roots for task groups (#32592)
- Kubernetes Executor Load Time Optimizations (#30727)
- Save DAG parsing time if dag is not schedulable (#30911)
- Updates health check endpoint to include dag_processor status. (#32382)
- Disable default allowing the testing of connections in UI, API and CLI (#32052, #33342)

- Fix config var types under the scheduler section (#32132)
- Allow to sort Grid View alphabetically (#32179)
- Add hostname to triggerer metric [triggers.running] (#32050)
- Improve DAG ORM cleanup code (#30614)
- TriggerDagRunOperator: Add wait_for_completion to template_fields (#31122)
- Open links in new tab that take us away from Airflow UI (#32088)
- Only show code tab when a task is not selected (#31744)
- Add descriptions for celery and dask cert configs (#31822)
- PythonVirtualenvOperator termination log in alert (#31747)
- Migration of all DAG details to existing grid view dag details panel (#31690)
- Add a diagram to help visualize timer metrics (#30650)
- Celery Executor load time optimizations (#31001)
- Update code style for airflow db commands to SQLAlchemy 2.0 style (#31486)
- Mark uses of md5 as “not-used-for-security” in FIPS environments (#31171)
- Add pydantic support to serde (#31565)
- Enable search in note column in DagRun and TaskInstance (#31455)
- Save scheduler execution time by adding new Index idea for dag_run (#30827)
- Save scheduler execution time by caching dags (#30704)
- Support for sorting DAGs by Last Run Date in the web UI (#31234)
- Better typing for Job and JobRunners (#31240)
- Add sorting logic by created_date for fetching triggers (#31151)
- Remove DAGs.can_create on access control doc, adjust test fixture (#30862)
- Split Celery logs into stdout/stderr (#30485)
- Decouple metrics clients and validators into their own modules (#30802)
- Description added for pagination in get_log api (#30729)
- Optimize performance of scheduling mapped tasks (#30372)
- Add sentry transport configuration option (#30419)
- Better message on deserialization error (#30588)

Bug Fixes

- Remove user sessions when resetting password (#33347)
- Gantt chart: Use earliest/oldest ti dates if different than dag run start/end (#33215)
- Fix virtualenv detection for Python virtualenv operator (#33223)
- Correctly log when there are problems trying to chmod airflow.cfg (#33118)
- Pass app context to webserver_config.py (#32759)
- Skip served logs for non-running task try (#32561)

- Fix reload gunicorn workers (#32102)
- Fix future DagRun rarely triggered by race conditions when `max_active_runs` reached its upper limit. (#31414)
- Fix BaseOperator `get_task_instances` query (#33054)
- Fix issue with using the various state enum value in logs (#33065)
- Use string concatenation to prepend base URL for `log_url` (#33063)
- Update graph nodes with operator style attributes (#32822)
- Affix webserver `access_denied` warning to be configurable (#33022)
- Only load task action modal if user can edit (#32992)
- OpenAPI Spec fix nullable alongside `$ref` (#32887)
- Make the decorators of `PythonOperator` sub-classes extend its decorator (#32845)
- Fix check if `virtualenv` is installed in `PythonVirtualenvOperator` (#32939)
- Unwrap Proxy before checking `__iter__` in `is_container()` (#32850)
- Override base log folder by using task handler's `base_log_folder` (#32781)
- Catch arbitrary exception from `run_job` to prevent zombie scheduler (#32707)
- Fix `depends_on_past` work for dynamic tasks (#32397)
- Sort `extra_links` for predictable order in UI. (#32762)
- Fix prefix group false graph (#32764)
- Fix bad delete logic for dagruns (#32684)
- Fix bug in `prune_dict` where empty dict and list would be removed even in strict mode (#32573)
- Add explicit browsers list and correct rel for blank target links (#32633)
- Handle returned None when `multiple_outputs` is True (#32625)
- Fix returned value when `ShortCircuitOperator` condition is falsy and there is not downstream tasks (#32623)
- Fix returned value when `ShortCircuitOperator` condition is falsy (#32569)
- Fix rendering of `dagRunTimeout` (#32565)
- Fix permissions on `/blocked` endpoint (#32571)
- Bugfix, prevent force of unpause on trigger DAG (#32456)
- Fix data interval in `cli.dags.trigger` command output (#32548)
- Strip whitespaces from airflow connections form (#32292)
- Add `timedelta` support for applicable arguments of sensors (#32515)
- Fix incorrect default on `readonly` property in our API (#32510)
- Add `xcom map_index` as a filter to `xcom` endpoint (#32453)
- Fix CLI commands when custom timetable is used (#32118)
- Use `WebEncoder` to encode `DagRun.conf` in `DagRun`'s list view (#32385)
- Fix logic of the `skip_all_except` method (#31153)
- Ensure dynamic tasks inside dynamic task group only marks the (#32354)
- Handle the cases that `webserver.expose_config` is set to non-sensitive-only instead of boolean value (#32261)

- Add retry functionality for handling process termination caused by database network issues (#31998)
- Adapt Notifier for sla_miss_callback (#31887)
- Fix XCOM view (#31807)
- Fix for “Filter dags by tag” flickering on initial load of dags.html (#31578)
- Fix where expanding `resizer` would not expand grid view (#31581)
- Fix MappedOperator-BaseOperator attr sync check (#31520)
- Always pass named `type_` arg to `drop_constraint` (#31306)
- Fix bad `drop_constraint` call in migrations (#31302)
- Resolving problems with redesigned grid view (#31232)
- Support `requirepass` redis sentinel (#30352)
- Fix webserver crash when calling get /config (#31057)

Misc/Internal

- Modify pathspec version restriction (#33349)
- Refactor: Simplify code in `dag_processing` (#33161)
- For now limit Pydantic to < 2.0.0 (#33235)
- Refactor: Simplify code in models (#33181)
- Add elasticsearch group to pre-2.7 defaults (#33166)
- Refactor: Simplify dict manipulation in airflow/cli (#33159)
- Remove redundant dict.keys() call (#33158)
- Upgrade ruff to latest 0.0.282 version in pre-commits (#33152)
- Move openlineage configuration to provider (#33124)
- Replace State by TaskInstanceState in Airflow executors (#32627)
- Get rid of Python 2 numeric relics (#33050)
- Remove legacy dag code (#33058)
- Remove legacy task instance modal (#33060)
- Remove old graph view (#32958)
- Move CeleryExecutor to the celery provider (#32526, #32628)
- Move all k8S classes to `cncf.kubernetes` provider (#32767, #32891)
- Refactor existence-checking SQL to helper (#32790)
- Extract Dask executor to new daskexecutor provider (#32772)
- Remove atlas configuration definition (#32776)
- Add Redis task handler (#31855)
- Move writing configuration for webserver to main (webserver limited) (#32766)
- Improve getting the query count in Airflow API endpoints (#32630)
- Remove click upper bound (#32634)

- Add D400 pydocstyle check - core Airflow only (#31297)
- D205 Support (#31742, #32575, #32213, #32212, #32591, #32449, #32450)
- Bump word-wrap from 1.2.3 to 1.2.4 in /airflow/www (#32680)
- Strong-type all single-state enum values (#32537)
- More strong typed state conversion (#32521)
- SQL query improvements in utils/db.py (#32518)
- Bump semver from 6.3.0 to 6.3.1 in /airflow/www (#32506)
- Bump jsonschema version to 4.18.0 (#32445)
- Bump stylelint from 13.13.1 to 15.10.1 in /airflow/www (#32435)
- Bump tough-cookie from 4.0.0 to 4.1.3 in /airflow/www (#32443)
- upgrade flask-appbuilder (#32054)
- Support Pydantic 2 (#32366)
- Limit click until we fix mypy issues (#32413)
- A couple of minor cleanups (#31890)
- Replace State usages with strong-typed enums (#31735)
- Upgrade ruff to 0.272 (#31966)
- Better error message when serializing callable without name (#31778)
- Improve the views module a bit (#31661)
- Remove asynctest (#31664)
- Refactor sqlalchemy queries to 2.0 style (#31569, #31772, #32350, #32339, #32474, #32645)
- Remove Python 3.7 support (#30963)
- Bring back min-airflow-version for preinstalled providers (#31469)
- Docstring improvements (#31375)
- Improve typing in SchedulerJobRunner (#31285)
- Upgrade ruff to 0.0.262 (#30809)
- Upgrade to MyPy 1.2.0 (#30687)

Docs only changes

- Clarify UI user types in security model (#33021)
- Add links to DAGRun / DAG / Task in templates-ref.rst (#33013)
- Add docs of how to test for DAG Import Errors (#32811)
- Clean-up of our new security page (#32951)
- Cleans up Extras reference page (#32954)
- Update Dag trigger API and command docs (#32696)
- Add deprecation info to the Airflow modules and classes docstring (#32635)
- Formatting installation doc to improve readability (#32502)

- Fix triggerer HA doc (#32454)
- Add type annotation to code examples (#32422)
- Document cron and delta timetables (#32392)
- Update index.rst doc to correct grammar (#32315)
- Fixing small typo in python.py (#31474)
- Separate out and clarify policies for providers (#30657)
- Fix docs: add an “apache” prefix to pip install (#30681)

3.16.21 Airflow 2.6.3 (2023-07-10)

Significant Changes

Default allowed pattern of a run_id has been changed to ^[A-Za-z0-9_.~:+-]+\$ (#32293).

Previously, there was no validation on the run_id string. There is now a validation regex that can be set by configuring `allowed_run_id_pattern` in `scheduler` section.

Bug Fixes

- Use linear time regular expressions (#32303)
- Fix triggerers alive check and add a new conf for triggerer heartbeat rate (#32123)
- Catch the exception that triggerer initialization failed (#31999)
- Hide sensitive values from extra in connection edit form (#32309)
- Sanitize `DagRun.run_id` and allow flexibility (#32293)
- Add triggerer canceled log (#31757)
- Fix try number shown in the task view (#32361)
- Retry transactions on occasional deadlocks for rendered fields (#32341)
- Fix behaviour of `LazyDictWithCache` when import fails (#32248)
- Remove `executor_class` from Job - fixing backfill for custom executors (#32219)
- Fix bugged singleton implementation (#32218)
- Use `mapIndex` to display extra links per mapped task. (#32154)
- Ensure that main triggerer thread exits if the async thread fails (#32092)
- Use `re2` for matching untrusted regex (#32060)
- Render list items in rendered fields view (#32042)
- Fix hashing of `dag_dependencies` in serialized dag (#32037)
- Return `None` if an `XComArg` fails to resolve in a `multiple_outputs` Task (#32027)
- Check for DAG ID in query param from url as well as kwargs (#32014)
- Flash an error message instead of failure in `rendered-templates` when map index is not found (#32011)
- Fix `ExternalTaskSensor` when there is no task group TIs for the current execution date (#32009)
- Fix number param html type in trigger template (#31980, #31946)
- Fix masking nested variable fields (#31964)

- Fix `operator_extra_links` property serialization in mapped tasks (#31904)
- Decode old-style nested Xcom value (#31866)
- Add a check for trailing slash in webserver base_url (#31833)
- Fix connection uri parsing when the host includes a scheme (#31465)
- Fix database session closing with `xcom_pull` and `inlets` (#31128)
- Fix DAG's `on_failure_callback` is not invoked when task failed during testing dag. (#30965)
- Fix airflow module version check when using `ExternalPythonOperator` and debug logging level (#30367)

Misc/Internal

- Fix `task.sensor` annotation in type stub (#31954)
- Limit Pydantic to < 2.0.0 until we solve 2.0.0 incompatibilities (#32312)
- Fix Pydantic 2 pickiness about model definition (#32307)

Doc only changes

- Add explanation about tag creation and cleanup (#32406)
- Minor updates to docs (#32369, #32315, #32310, #31794)
- Clarify Listener API behavior (#32269)
- Add information for users who ask for requirements (#32262)
- Add links to DAGRun / DAG / Task in Templates Reference (#32245)
- Add comment to warn off a potential wrong fix (#32230)
- Add a note that we'll need to restart triggerer to reflect any trigger change (#32140)
- Adding missing hyperlink to the tutorial documentation (#32105)
- Added difference between Deferrable and Non-Deferrable Operators (#31840)
- Add comments explaining need for special “trigger end” log message (#31812)
- Documentation update on Plugin updates. (#31781)
- Fix SemVer link in security documentation (#32320)
- Update security model of Airflow (#32098)
- Update references to restructured documentation from Airflow core (#32282)
- Separate out advanced logging configuration (#32131)
- Add `™` to Airflow in prominent places (#31977)

3.16.22 Airflow 2.6.2 (2023-06-17)

Significant Changes

No significant changes.

Bug Fixes

- Cascade update of TaskInstance to TaskMap table (#31445)
- Fix Kubernetes executors detection of deleted pods (#31274)
- Use keyword parameters for migration methods for mssql (#31309)
- Control permissibility of driver config in extra from airflow.cfg (#31754)
- Fixing broken links in openapi/v1.yaml (#31619)
- Hide old alert box when testing connection with different value (#31606)
- Add TriggererStatus to OpenAPI spec (#31579)
- Resolving issue where Grid won't un-collapse when Details is collapsed (#31561)
- Fix sorting of tags (#31553)
- Add the missing `map_index` to the `xcom` key when skipping downstream tasks (#31541)
- Fix airflow users delete CLI command (#31539)
- Include triggerer health status in Airflow /health endpoint (#31529)
- Remove dependency already registered for this task warning (#31502)
- Use `kube_client` over default `CoreV1Api` for deleting pods (#31477)
- Ensure min backoff in base sensor is at least 1 (#31412)
- Fix `max_active_tis_per_dagrun` for Dynamic Task Mapping (#31406)
- Fix error handling when pre-importing modules in DAGs (#31401)
- Fix dropdown default and adjust tutorial to use 42 as default for proof (#31400)
- Fix crash when clearing run with task from normal to mapped (#31352)
- Make `BaseJobRunner` a generic on the job class (#31287)
- Fix `url_for_asset` fallback and 404 on DAG Audit Log (#31233)
- Don't present an undefined execution date (#31196)
- Added spinner activity while the logs load (#31165)
- Include rediss to the list of supported URL schemes (#31028)
- Optimize scheduler by skipping “non-schedulable” DAGs (#30706)
- Save scheduler execution time during search for queued dag_runs (#30699)
- Fix ExternalTaskSensor to work correctly with task groups (#30742)
- Fix DAG.access_control can't sync when clean access_control (#30340)
- Fix failing get_safe_url tests for latest Python 3.8 and 3.9 (#31766)
- Fix typing for POST user endpoint (#31767)
- Fix wrong update for nested group default args (#31776)
- Fix overriding `default_args` in nested task groups (#31608)
- Mark `[secrets]` `backend_kwargs` as a sensitive config (#31788)
- Executor events are not always “exited” here (#30859)
- Validate connection IDs (#31140)

Misc/Internal

- Add Python 3.11 support (#27264)
- Replace unicodedcsv with standard csv library (#31693)
- Bring back unicodedcsv as dependency of Airflow (#31814)
- Remove found_descendents param from get_flat_relative_ids (#31559)
- Fix typing in external task triggers (#31490)
- Wording the next and last run DAG columns better (#31467)
- Skip auto-document things with :meta private: (#31380)
- Add an example for sql_alchemy_connect_args conf (#31332)
- Convert dask upper-binding into exclusion (#31329)
- Upgrade FAB to 4.3.1 (#31203)
- Added metavar and choices to –state flag in airflow dags list-jobs CLI for suggesting valid state arguments. (#31308)
- Use only one line for tmp dir log (#31170)
- Rephrase comment in setup.py (#31312)
- Add fullname to owner on logging (#30185)
- Make connection id validation consistent across interface (#31282)
- Use single source of truth for sensitive config items (#31820)

Doc only changes

- Add docstring and signature for _read_remote_logs (#31623)
- Remove note about triggerer being 3.7+ only (#31483)
- Fix version support information (#31468)
- Add missing BashOperator import to documentation example (#31436)
- Fix task.branch error caused by incorrect initial parameter (#31265)
- Update callbacks documentation (errors and context) (#31116)
- Add an example for dynamic task mapping with non-TaskFlow operator (#29762)
- Few doc fixes - links, grammar and wording (#31719)
- Add description in a few more places about adding airflow to pip install (#31448)
- Fix table formatting in docker build documentation (#31472)
- Update documentation for constraints installation (#31882)

3.16.23 Airflow 2.6.1 (2023-05-16)

Significant Changes

Clarifications of the external Health Check mechanism and using Job classes (#31277).

In the past SchedulerJob and other *Job classes are known to have been used to perform external health checks for Airflow components. Those are, however, Airflow DB ORM related classes. The DB models and database structure

of Airflow are considered as internal implementation detail, following [public interface](#)). Therefore, they should not be used for external health checks. Instead, you should use the `airflow jobs check` CLI command (introduced in Airflow 2.1) for that purpose.

Bug Fixes

- Fix calculation of health check threshold for SchedulerJob (#31277)
- Fix timestamp parse failure for k8s executor pod tailing (#31175)
- Make sure that DAG processor job row has filled value in `job_type` column (#31182)
- Fix section name reference for `api_client_retry_configuration` (#31174)
- Ensure the KPO runs pod mutation hooks correctly (#31173)
- Remove worrying log message about redaction from the OpenLineage plugin (#31149)
- Move `interleave_timestamp_parser` config to the logging section (#31102)
- Ensure that we check worker for served logs if no local or remote logs found (#31101)
- Fix `MappedTaskGroup` import in taskinstance file (#31100)
- Format `DagBag.dagbag_report()` Output (#31095)
- Mask task attribute on task detail view (#31125)
- Fix template error when iterating None value and fix params documentation (#31078)
- Fix `apache-hive` extra so it installs the correct package (#31068)
- Fix issue with zip files in DAGs folder when pre-importing Airflow modules (#31061)
- Move `TaskInstanceKey` to a separate file to fix circular import (#31033, #31204)
- Fix deleting `DagRuns` and `TaskInstances` that have a note (#30987)
- Fix `airflow providers get` command output (#30978)
- Fix Pool schema in the OpenAPI spec (#30973)
- Add support for dynamic tasks with template fields that contain `pandas.DataFrame` (#30943)
- Use the Task Group explicitly passed to ‘partial’ if any (#30933)
- Fix `order_by` request in list DAG rest api (#30926)
- Include node height/width in center-on-task logic (#30924)
- Remove print from dag trigger command (#30921)
- Improve task group UI in new graph (#30918)
- Fix mapped states in grid view (#30916)
- Fix problem with displaying graph (#30765)
- Fix backfill `KeyError` when `try_number` out of sync (#30653)
- Re-enable clear and setting state in the `TaskInstance` UI (#30415)
- Prevent `DagRun`’s `state` and `start_date` from being reset when clearing a task in a running `DagRun` (#30125)

Misc/Internal

- Upper bind task until they solve a side effect in their test suite (#31259)
- Show task instances affected by clearing in a table (#30633)
- Fix missing models in API documentation (#31021)

Doc only changes

- Improve description of the `dag_processing.processes` metric (#30891)
- Improve Quick Start instructions (#30820)
- Add section about missing task logs to the FAQ (#30717)
- Mount the `config` directory in docker compose (#30662)
- Update `version_added` config field for `might_contain_dag` and `metrics_allow_list` (#30969)

3.16.24 Airflow 2.6.0 (2023-04-30)

Significant Changes

Default permissions of file task handler log directories and files has been changed to “owner + group” writeable (#29506).

Default setting handles case where impersonation is needed and both users (airflow and the impersonated user) have the same group set as main group. Previously the default was also other-writeable and the user might choose to use the other-writeable setting if they wish by configuring `file_task_handler_new_folder_permissions` and `file_task_handler_new_file_permissions` in logging section.

SLA callbacks no longer add files to the dag processor manager’s queue (#30076)

This stops SLA callbacks from keeping the dag processor manager permanently busy. It means reduced CPU, and fixes issues where SLAs stop the system from seeing changes to existing dag files. Additional metrics added to help track queue state.

The `cleanup()` method in `BaseTrigger` is now defined as asynchronous (following `async/await`) pattern (#30152).

This is potentially a breaking change for any custom trigger implementations that override the `cleanup()` method and uses synchronous code, however using synchronous operations in cleanup was technically wrong, because the method was executed in the main loop of the Triggerer and it was introducing unnecessary delays impacting other triggers. The change is unlikely to affect any existing trigger implementations.

The gauge `scheduler.tasks.running` no longer exist (#30374)

The gauge has never been working and its value has always been 0. Having an accurate value for this metric is complex so it has been decided that removing this gauge makes more sense than fixing it with no certainty of the correctness of its value.

Consolidate handling of tasks stuck in queued under new `task_queued_timeout` config (#30375)

Logic for handling tasks stuck in the queued state has been consolidated, and the all configurations responsible for timing out stuck queued tasks have been deprecated and merged into `[scheduler] task_queued_timeout`. The configurations that have been deprecated are `[kubernetes] worker_pods_pending_timeout`, `[celery] stalled_task_timeout`, and `[celery] task_adoption_timeout`. If any of these configurations are set, the

longest timeout will be respected. For example, if [celery] staled_task_timeout is 1200, and [scheduler] task_queued_timeout is 600, Airflow will set [scheduler] task_queued_timeout to 1200.

Improvement Changes

Display only the running configuration in configurations view (#28892)

The configurations view now only displays the running configuration. Previously, the default configuration was displayed at the top but it was not obvious whether this default configuration was overridden or not. Subsequently, the non-documented endpoint /configuration?raw=true is deprecated and will be removed in Airflow 3.0. The HTTP response now returns an additional Deprecation header. The /config endpoint on the REST API is the standard way to fetch Airflow configuration programmatically.

Explicit skipped states list for ExternalTaskSensor (#29933)

ExternalTaskSensor now has an explicit skipped_states list

Miscellaneous Changes

Handle OverflowError on exponential backoff in next_run_calculation (#28172)

Maximum retry task delay is set to be 24h (86400s) by default. You can change it globally via core.max_task_retry_delay parameter.

Move Hive macros to the provider (#28538)

The Hive Macros (hive.max_partition, hive.closest_ds_partition) are available only when Hive Provider is installed. Please install Hive Provider > 5.1.0 when using those macros.

Updated app to support configuring the caching hash method for FIPS v2 (#30675)

Various updates for FIPS-compliance when running Airflow in Python 3.9+. This includes a new webserver option, caching_hash_method, for changing the default flask caching method.

New Features

- AIP-50 Trigger DAG UI Extension with Flexible User Form Concept (#27063,#29376)
- Skip PythonVirtualenvOperator task when it returns a provided exit code (#30690)
- rename skip_exit_code to skip_on_exit_code and allow providing multiple codes (#30692)
- Add skip_on_exit_code also to ExternalPythonOperator (#30738)
- Add max_active_tis_per_dagrun for Dynamic Task Mapping (#29094)
- Add serializer for pandas dataframe (#30390)
- Deferrable TriggerDagRunOperator (#30292)
- Add command to get DAG Details via CLI (#30432)
- Adding ContinuousTimetable and support for @continuous schedule_interval (#29909)
- Allow customized rules to check if a file has dag (#30104)
- Add a new Airflow conf to specify a SSL ca cert for Kubernetes client (#30048)
- Bash sensor has an explicit retry code (#30080)
- Add filter task upstream/downstream to grid view (#29885)

- Add testing a connection via Airflow CLI (#29892)
- Support deleting the local log files when using remote logging (#29772)
- Blocklist to disable specific metric tags or metric names (#29881)
- Add a new graph inside of the grid view (#29413)
- Add database `check_migrations` config (#29714)
- add output format arg for `cli.dags.trigger` (#29224)
- Make json and yaml available in templates (#28930)
- Enable tagged metric names for existing Statsd metric publishing events | influxdb-statsd support (#29093)
- Add arg `--yes` to `db export-archived` command. (#29485)
- Make the policy functions pluggable (#28558)
- Add `airflow db drop-archived` command (#29309)
- Enable individual trigger logging (#27758)
- Implement new filtering options in graph view (#29226)
- Add triggers for ExternalTask (#29313)
- Add command to export purged records to CSV files (#29058)
- Add `FileTrigger` (#29265)
- Emit DataDog statsd metrics with metadata tags (#28961)
- Add some statsd metrics for dataset (#28907)
- Add `--overwrite` option to `connections import` CLI command (#28738)
- Add general-purpose “notifier” concept to DAGs (#28569)
- Add a new conf to wait `past_deps` before skipping a task (#27710)
- Add Flink on K8s Operator (#28512)
- Allow Users to disable SwaggerUI via configuration (#28354)
- Show mapped task groups in graph (#28392)
- Log FileTaskHandler to work with KubernetesExecutor’s `multi_namespace_mode` (#28436)
- Add a new config for adapting masked secrets to make it easier to prevent secret leakage in logs (#28239)
- List specific config section and its values using the cli (#28334)
- KubernetesExecutor `multi_namespace_mode` can use namespace list to avoid requiring cluster role (#28047)
- Automatically save and allow restore of recent DAG run configs (#27805)
- Added `exclude_microseconds` to cli (#27640)

Improvements

- Rename most `pod_id` usage to `pod_name` in KubernetesExecutor (#29147)
- Update the error message for invalid use of poke-only sensors (#30821)
- Update log level in scheduler critical section edge case (#30694)
- AIP-51 Removing Executor Coupling from Core Airflow ([AIP-51](#))

- Add multiple exit code handling in skip logic for BashOperator (#30739)
- Updated app to support configuring the caching hash method for FIPS v2 (#30675)
- Preload airflow imports before dag parsing to save time (#30495)
- Improve task & run actions UX in grid view (#30373)
- Speed up TaskGroups with caching property of group_id (#30284)
- Use the engine provided in the session (#29804)
- Type related import optimization for Executors (#30361)
- Add more type hints to the code base (#30503)
- Always use self.appbuilder.get_session in security managers (#30233)
- Update SQLAlchemy select() to new style (#30515)
- Refactor out xcom constants from models (#30180)
- Add exception class name to DAG-parsing error message (#30105)
- Rename statsd_allow_list and statsd_block_list to metrics_*_list (#30174)
- Improve serialization of tuples and sets (#29019)
- Make cleanup method in trigger an async one (#30152)
- Lazy load serialization modules (#30094)
- SLA callbacks no longer add files to the dag_processing manager queue (#30076)
- Add task.trigger rule to grid_data (#30130)
- Speed up log template sync by avoiding ORM (#30119)
- Separate cli_parser.py into two modules (#29962)
- Explicit skipped states list for ExternalTaskSensor (#29933)
- Add task state hover highlighting to new graph (#30100)
- Store grid tabs in url params (#29904)
- Use custom Connexion resolver to load lazily (#29992)
- Delay Kubernetes import in secret masker (#29993)
- Delay ConnectionModelView init until it's accessed (#29946)
- Scheduler, make stale DAG deactivation threshold configurable instead of using dag processing timeout (#29446)
- Improve grid view height calculations (#29563)
- Avoid importing executor during conf validation (#29569)
- Make permissions for FileTaskHandler group-writeable and configurable (#29506)
- Add colors in help outputs of Airflow CLI commands #28789 (#29116)
- Add a param for get_dags endpoint to list only unpause dags (#28713)
- Expose updated_at filter for dag run and task instance endpoints (#28636)
- Increase length of user identifier columns (#29061)
- Update gantt chart UI to display queued state of tasks (#28686)
- Add index on log.dttm (#28944)

- Display only the running configuration in configurations view (#28892)
- Cap dropdown menu size dynamically (#28736)
- Added JSON linter to connection edit / add UI for field extra. On connection edit screen, existing extra data will be displayed indented (#28583)
- Use labels instead of pod name for pod log read in k8s exec (#28546)
- Use time not tries for queued & running re-checks. (#28586)
- CustomTTYColoredFormatter should inherit TimezoneAware formatter (#28439)
- Improve past depends handling in Airflow CLI tasks.run command (#28113)
- Support using a list of callbacks in `on_*_callback`/`sla_miss_callbacks` (#28469)
- Better table name validation for db clean (#28246)
- Use object instead of array in config.yml for config template (#28417)
- Add markdown rendering for task notes. (#28245)
- Show mapped task groups in grid view (#28208)
- Add `renamed` and `previous_name` in config sections (#28324)
- Speed up most Users/Role CLI commands (#28259)
- Speed up Airflow role list command (#28244)
- Refactor serialization (#28067, #30819, #30823)
- Allow longer pod names for k8s executor / KPO (#27736)
- Updates health check endpoint to include `triggerer` status (#27755)

Bug Fixes

- Fix static_folder for cli app (#30952)
- Initialize plugins for cli appbuilder (#30934)
- Fix dag file processor heartbeat to run only if necessary (#30899)
- Fix KubernetesExecutor sending state to scheduler (#30872)
- Count mapped upstream only if all are finished (#30641)
- ExternalTaskSensor: add `external_task_group_id` to `template_fields` (#30401)
- Improve url detection for task instance details (#30779)
- Use material icons for dag import error banner (#30771)
- Fix misc grid/graph view UI bugs (#30752)
- Add a collapse grid button (#30711)
- Fix d3 dependencies (#30702)
- Simplify logic to resolve tasks stuck in queued despite stalled_task_timeout (#30375)
- When clearing task instances try to get associated DAGs from database (#29065)
- Fix mapped tasks partial arguments when DAG default args are provided (#29913)
- Deactivate DAGs deleted from within zip files (#30608)

- Recover from `too old resource version` exception by retrieving the latest `resource_version` (#30425)
- Fix possible race condition when refreshing DAGs (#30392)
- Use custom validator for OpenAPI request body (#30596)
- Fix `TriggerDagRunOperator` with deferrable parameter (#30406)
- Speed up dag runs deletion (#30330)
- Do not use template literals to construct html elements (#30447)
- Fix deprecation warning in `example_sensor_decorator` DAG (#30513)
- Avoid logging sensitive information in triggerer job log (#30110)
- Add a new parameter for base sensor to catch the exceptions in poke method (#30293)
- Fix dag run conf encoding with non-JSON serializable values (#28777)
- Added fixes for Airflow to be usable on Windows Task-Workers (#30249)
- Force DAG last modified time to UTC (#30243)
- Fix `EmptySkipOperator` in example dag (#30269)
- Make the webserver startup respect `update_fab_perms` (#30246)
- Ignore error when changing log folder permissions (#30123)
- Disable ordering `DagRuns` by note (#30043)
- Fix reading logs from finished KubernetesExecutor worker pod (#28817)
- Mask out non-access bits when comparing file modes (#29886)
- Remove Run task action from UI (#29706)
- Fix log tailing issues with legacy log view (#29496)
- Fixes to how DebugExecutor handles sensors (#28528)
- Ensure that `pod_mutation_hook` is called before logging the pod name (#28534)
- Handle OverflowError on exponential backoff in `next_run_calculation` (#28172)

Misc/Internal

- Make eager upgrade additional dependencies optional (#30811)
- Upgrade to pip 23.1.1 (#30808)
- Remove protobuf limitation from eager upgrade (#30182)
- Remove protobuf limitation from eager upgrade (#30182)
- Deprecate `skip_exit_code` in `BashOperator` (#30734)
- Remove gauge `scheduler.tasks.running` (#30374)
- Bump json5 to 1.0.2 and eslint-plugin-import to 2.27.5 in `/airflow/www` (#30568)
- Add tests to `PythonOperator` (#30362)
- Add `asgiref` as a core dependency (#30527)
- Discovery safe mode toggle comment clarification (#30459)
- Upgrade moment-timezone package to fix Tehran tz (#30455)

- Bump loader-utils from 2.0.0 to 2.0.4 in /airflow/www (#30319)
- Bump babel-loader from 8.1.0 to 9.1.0 in /airflow/www (#30316)
- DagBag: Use `dag.fileloc` instead of `dag.full_filepath` in exception message (#30610)
- Change log level of serialization information (#30239)
- Minor DagRun helper method cleanup (#30092)
- Improve type hinting in stats.py (#30024)
- Limit `importlib-metadata` backport to < 5.0.0 (#29924)
- Align cncf provider file names with AIP-21 (#29905)
- Upgrade FAB to 4.3.0 (#29766)
- Clear ExecutorLoader cache in tests (#29849)
- Lazy load Task Instance logs in UI (#29827)
- added warning log for max page limit exceeding api calls (#29788)
- Aggressively cache entry points in process (#29625)
- Don't use `importlib.metadata` to get Version for speed (#29723)
- Upgrade Mypy to 1.0 (#29468)
- Rename db `export-cleaned` to `db export-archived` (#29450)
- listener: simplify API by replacing SQLAlchemy event-listening by direct calls (#29289)
- No multi-line log entry for bash env vars (#28881)
- Switch to ruff for faster static checks (#28893)
- Remove horizontal lines in TI logs (#28876)
- Make allowed_deserialization_classes more intuitive (#28829)
- Propagate logs to stdout when in k8s executor pod (#28440, #30860)
- Fix code readability, add docstrings to json_client (#28619)
- AIP-51 - Misc. Compatibility Checks (#28375)
- Fix `is_local` for LocalKubernetesExecutor (#28288)
- Move Hive macros to the provider (#28538)
- Rerun flaky PinotDB integration test (#28562)
- Add pre-commit hook to check session default value (#28007)
- Refactor `get_mapped_group_summaries` for web UI (#28374)
- Add support for k8s 1.26 (#28320)
- Replace `freezegun` with `time-machine` (#28193)
- Completed D400 for `airflow/kubernetes/*` (#28212)
- Completed D400 for multiple folders (#27969)
- Drop k8s 1.21 and 1.22 support (#28168)
- Remove unused `task_queue` attr from k8s scheduler class (#28049)
- Completed D400 for multiple folders (#27767, #27768)

Doc only changes

- Add instructions on how to avoid accidental airflow upgrade/downgrade (#30813)
- Add explicit information about how to write task logs (#30732)
- Better explanation on how to log from tasks (#30746)
- Use correct import path for Dataset (#30617)
- Create `audit_logs.rst` (#30405)
- Adding taskflow API example for sensors (#30344)
- Add clarification about timezone aware dags (#30467)
- Clarity params documentation (#30345)
- Fix unit for task duration metric (#30273)
- Update `dag-run.rst` for dead links of cli commands (#30254)
- Add Write efficient Python code section to Reducing DAG complexity (#30158)
- Allow to specify which connection, variable or config are being looked up in the backend using `*_lookup_pattern` parameters (#29580)
- Add Documentation for notification feature extension (#29191)
- Clarify that executor interface is public but instances are not (#29200)
- Add Public Interface description to Airflow documentation (#28300)
- Add documentation for task group mapping (#28001)
- Some fixes to metrics doc (#30290)

3.16.25 Airflow 2.5.3 (2023-04-01)

Significant Changes

No significant changes.

Bug Fixes

- Fix DagProcessorJob integration for standalone dag-processor (#30278)
- Fix proper termination of gunicorn when it hangs (#30188)
- Fix XCom.get_one exactly one exception text (#30183)
- Correct the VARCHAR size to 250. (#30178)
- Revert fix for `on_failure_callback` when task receives a SIGTERM (#30165)
- Move read only property to `DagState` to fix generated docs (#30149)
- Ensure that `dag.partial_subset` doesn't mutate task group properties (#30129)
- Fix inconsistent returned value of `airflow dags next-execution` cli command (#30117)
- Fix `www/utils/dag_run_link` redirection (#30098)
- Fix `TriggerRuleDep` when the mapped tasks count is 0 (#30084)
- Dag processor manager, add `retry_db_transaction` to `_fetch_callbacks` (#30079)
- Fix db clean command for mysql db (#29999)

- Avoid considering EmptyOperator in mini scheduler (#29979)
- Fix some long known Graph View UI problems (#29971, #30355, #30360)
- Fix dag docs toggle icon initial angle (#29970)
- Fix tags selection in DAGs UI (#29944)
- Including airflow/example_dags/sql/sample.sql in MANIFEST.in (#29883)
- Fixing broken filter in /taskinstance/list view (#29850)
- Allow generic param dicts (#29782)
- Fix update_mask in patch variable route (#29711)
- Strip markup from app_name if instance_name_has_markup = True (#28894)

Misc/Internal

- Revert “Also limit importlib on Python 3.9 (#30069)” (#30209)
- Add custom_operator_name to @task.sensor tasks (#30131)
- Bump webpack from 5.73.0 to 5.76.0 in /airflow/www (#30112)
- Formatted config (#30103)
- Remove upper bound limit of astroid (#30033)
- Remove accidentally merged vendor daemon patch code (#29895)
- Fix warning in airflow tasks test command regarding absence of data_interval (#27106)

Doc only changes

- Adding more information regarding top level code (#30040)
- Update workday example (#30026)
- Fix some typos in the DAGs docs (#30015)
- Update set-up-database.rst (#29991)
- Fix some typos on the kubernetes documentation (#29936)
- Fix some punctuation and grammar (#29342)

3.16.26 Airflow 2.5.2 (2023-03-15)

Significant Changes

The date-time fields passed as API parameters or Params should be RFC3339-compliant (#29395)

In case of API calls, it was possible that “+” passed as part of the date-time fields were not URL-encoded, and such date-time fields could pass validation. Such date-time parameters should now be URL-encoded (as %2B).

In case of parameters, we still allow IS8601-compliant date-time (so for example it is possible that ‘ ‘ was used instead of T separating date from time and no timezone was specified) but we raise deprecation warning.

Default for [webserver] expose_hostname changed to False (#29547)

The default for [webserver] expose_hostname has been set to False, instead of True. This means administrators must opt-in to expose webserver hostnames to end users.

Bug Fixes

- Fix validation of date-time field in API and Parameter schemas (#29395)
- Fix grid logs for large logs (#29390)
- Fix `on_failure_callback` when task receives a SIGTERM (#29743)
- Update min version of python-daemon to fix containerd file limits (#29916)
- POST `/dagRuns` API should 404 if dag not active (#29860)
- DAG list sorting lost when switching page (#29756)
- Fix Scheduler crash when clear a previous run of a normal task that is now a mapped task (#29645)
- Convert moment with timezone to UTC instead of raising an exception (#29606)
- Fix clear dag run openapi spec responses by adding additional return type (#29600)
- Don't display empty rendered attrs in Task Instance Details page (#29545)
- Remove section check from get-value command (#29541)
- Do not show version/node in UI traceback for unauthenticated user (#29501)
- Make `prev_logical_date` variable offset-aware (#29454)
- Fix nested fields rendering in mapped operators (#29451)
- Datasets, next_run_datasets, remove unnecessary timestamp filter (#29441)
- Edgemodifier refactoring w/ labels in TaskGroup edge case (#29410)
- Fix Rest API update user output (#29409)
- Ensure Serialized DAG is deleted (#29407)
- Persist DAG and task doc values in TaskFlow API if explicitly set (#29399)
- Redirect to the origin page with all the params (#29212)
- Fixing Task Duration view in case of manual DAG runs only (#22015) (#29195)
- Remove poke method to fall back to parent implementation (#29146)
- PR: Introduced fix to run tasks on Windows systems (#29107)
- Fix warning in migrations about old config. (#29092)
- Emit dagrun failed duration when timeout (#29076)
- Handling error on cluster policy itself (#29056)
- Fix kerberos authentication for the REST API. (#29054)
- Fix leak sensitive field via V1EnvVar on exception (#29016)
- Sanitize url_for arguments before they are passed (#29039)
- Fix dag run trigger with a note. (#29228)
- Write action log to DB when DAG run is triggered via API (#28998)
- Resolve all variables in pickled XCom iterator (#28982)
- Allow URI without authority and host blocks in `airflow connections add` (#28922)
- Be more selective when adopting pods with KubernetesExecutor (#28899)
- KubenetesExecutor sends state even when successful (#28871)

- Annotate KubernetesExecutor pods that we don't delete (#28844)
- Throttle streaming log reads (#28818)
- Introduce dag processor job (#28799)
- Fix #28391 manual task trigger from UI fails for k8s executor (#28394)
- Logging poke info when external dag is not none and task_id and task_ids are none (#28097)
- Fix inconsistencies in checking edit permissions for a DAG (#20346)

Misc/Internal

- Add a check for not templateable fields (#29821)
- Removed continue for not in (#29791)
- Move extra links position in grid view (#29703)
- Bump `undici` from 5.9.1 to 5.19.1 (#29583)
- Change `expose_hostname` default to false (#29547)
- Change permissions of config/password files created by airflow (#29495)
- Use newer setuptools v67.2.0 (#29465)
- Increase max height for grid view elements (#29367)
- Clarify description of worker control config (#29247)
- Bump `ua-parser-js` from 0.7.31 to 0.7.33 in /airflow/www (#29172)
- Remove upper bound limitation for `pytest` (#29086)
- Check for `run_id` url param when linking to `graph/gantt` views (#29066)
- Clarify graph view dynamic task labels (#29042)
- Fixing import error for dataset (#29007)
- Update how PythonSensor returns values from `python_callable` (#28932)
- Add dep context description for better log message (#28875)
- Bump `swagger-ui-dist` from 3.52.0 to 4.1.3 in /airflow/www (#28824)
- Limit `importlib-metadata` backport to < 5.0.0 (#29924, #30069)

Doc only changes

- Update pipeline.rst - Fix query in `merge_data()` task (#29158)
- Correct argument name of Workday timetable in timetable.rst (#29896)
- Update ref anchor for env var link in Connection how-to doc (#29816)
- Better description for limit in api (#29773)
- Description of `dag_processing.last_duration` (#29740)
- Update docs re: `template_fields` typing and subclasses (#29725)
- Fix formatting of Dataset inlet/outlet note in TaskFlow concepts (#29678)
- Specific use-case: adding packages via requirements.txt in compose (#29598)
- Detect is 'docker-compose' existing (#29544)

- Add Landing Times entry to UI docs (#29511)
- Improve health checks in example docker-compose and clarify usage (#29408)
- Remove `notes` param from TriggerDagRunOperator docstring (#29298)
- Use `schedule` param rather than `timetable` in Timetables docs (#29255)
- Add trigger process to Airflow Docker docs (#29203)
- Update `set-up-database.rst` (#29104)
- Several improvements to the Params doc (#29062)
- Email Config docs more explicit env var examples (#28845)
- Listener plugin example added (#27905)

3.16.27 Airflow 2.5.1 (2023-01-20)

Significant Changes

Trigger gevent monkeypatching via environment variable (#28283)

If you are using gevent for your webserver deployment and used local settings to `monkeypatch gevent`, you might want to replace local settings patching with an `_AIRFLOW_PATCH_GEVENT` environment variable set to 1 in your webserver. This ensures gevent patching is done as early as possible.

Bug Fixes

- Fix masking of non-sensitive environment variables (#28802)
- Remove swagger-ui extra from connexion and install `swagger-ui-dist` via npm package (#28788)
- Fix `UIAlert` `should_show` when `AUTH_ROLE_PUBLIC` set (#28781)
- Only patch single label when adopting pod (#28776)
- Update CSRF token to expire with session (#28730)
- Fix “airflow tasks render” cli command for mapped task instances (#28698)
- Allow XComArgs for `external_task_ids` of ExternalTaskSensor (#28692)
- Row-lock TIs to be removed during mapped task expansion (#28689)
- Handle ConnectionReset exception in Executor cleanup (#28685)
- Fix description of output redirection for `access_log` for gunicorn (#28672)
- Add back join to zombie query that was dropped in #28198 (#28544)
- Fix calendar view for CronTriggerTimeTable dags (#28411)
- After running the DAG the employees table is empty. (#28353)
- Fix `DetachedInstanceError` when finding zombies in Dag Parsing process (#28198)
- Nest header blocks in divs to fix dagid copy nit on `dag.html` (#28643)
- Fix UI caret direction (#28624)
- Guard not-yet-expanded ti in trigger rule dep (#28592)
- Move TI `setNote` endpoints under TaskInstance in OpenAPI (#28566)
- Consider previous run in CronTriggerTimetable (#28532)

- Ensure correct log dir in file task handler (#28477)
- Fix bad pods pickled in executor_config (#28454)
- Add `ensure_ascii=False` in trigger dag run API (#28451)
- Add setters to MappedOperator on_*_callbacks (#28313)
- Fix `ti._try_number` for deferred and up_for_reschedule tasks (#26993)
- separate `callModal` from `dag.js` (#28410)
- A manual run can't look like a scheduled one (#28397)
- Dont show task/run durations when there is no start_date (#28395)
- Maintain manual scroll position in task logs (#28386)
- Correctly select a mapped task's "previous" task (#28379)
- Trigger gevent monkeypatching via environment variable (#28283)
- Fix db clean warnings (#28243)
- Make arguments 'offset' and 'length' not required (#28234)
- Make live logs reading work for "other" k8s executors (#28213)
- Add custom pickling hooks to `LazyXComAccess` (#28191)
- fix next run datasets error (#28165)
- Ensure that warnings from `@dag` decorator are reported in dag file (#28153)
- Do not warn when airflow dags tests command is used (#28138)
- Ensure the `dagbag_size` metric decreases when files are deleted (#28135)
- Improve run/task grid view actions (#28130)
- Make `BaseJob.most_recent_job` favor "running" jobs (#28119)
- Don't emit FutureWarning when code not calling old key (#28109)
- Add `airflow.api.auth.backend.session` to backend sessions in compose (#28094)
- Resolve false warning about calling `conf.get` on moved item (#28075)
- Return list of tasks that will be changed (#28066)
- Handle bad zip files nicely when parsing DAGs. (#28011)
- Prevent double loading of providers from local paths (#27988)
- Fix deadlock when chaining multiple empty mapped tasks (#27964)
- fix: `current_state` method on `TaskInstance` doesn't filter by `map_index` (#27898)
- Don't log CLI actions if db not initialized (#27851)
- Make sure we can get out of a faulty scheduler state (#27834)
- `dagrun`, `next_dagruns_to_examine`, add MySQL index hint (#27821)
- Handle DAG disappearing mid-flight when dag verification happens (#27720)
- fix: continue checking sla (#26968)
- Allow generation of connection URI to work when no conn type (#26765)

Misc/Internal

- Remove limit for `dnspython` after eventlet got fixed (#29004)
- Limit `dnspython` to < 2.3.0 until eventlet incompatibility is solved (#28962)
- Add automated version replacement in example dag indexes (#28090)
- Cleanup and do housekeeping with plugin examples (#28537)
- Limit SQLAlchemy to below 2.0 (#28725)
- Bump `json5` from 1.0.1 to 1.0.2 in /airflow/www (#28715)
- Fix some docs on using sensors with taskflow (#28708)
- Change Architecture and OperatingSystem classes into Enums (#28627)
- Add doc-strings and small improvement to email util (#28634)
- Fix `Connection.get_extra` type (#28594)
- navbar, cap dropdown size, and add scroll bar (#28561)
- Emit warnings for `conf.get*` from the right source location (#28543)
- Move MyPY plugins of ours to dev folder (#28498)
- Add retry to `purge_inactive_dag_warnings` (#28481)
- Re-enable Plyvel on ARM as it now builds cleanly (#28443)
- Add SIGUSR2 handler for LocalTaskJob and workers to aid debugging (#28309)
- Convert `test_task_command` to Pytest and unquarantine tests in it (#28247)
- Make invalid characters exception more readable (#28181)
- Bump decode-uri-component from 0.2.0 to 0.2.2 in /airflow/www (#28080)
- Use asserts instead of exceptions for executor not started (#28019)
- Simplify dataset subgraph logic (#27987)
- Order TIs by `map_index` (#27904)
- Additional info about Segmentation Fault in LocalTaskJob (#27381)

Doc only changes

- Mention mapped operator in cluster policy doc (#28885)
- Slightly improve description of Dynamic DAG generation preamble (#28650)
- Restructure Docs (#27235)
- Update scheduler docs about low priority tasks (#28831)
- Clarify that versioned constraints are fixed at release time (#28762)
- Clarify about docker compose (#28729)
- Adding an example dag for dynamic task mapping (#28325)
- Use docker compose v2 command (#28605)
- Add `AIRFLOW_PROJ_DIR` to docker-compose example (#28517)
- Remove outdated Optional Provider Feature outdated documentation (#28506)
- Add documentation for [core] mp_start_method config (#27993)

- Documentation for the LocalTaskJob return code counter (#27972)
- Note which versions of Python are supported (#27798)

3.16.28 Airflow 2.5.0 (2022-12-02)

Significant Changes

airflow dags test no longer performs a backfill job (#26400)

In order to make `airflow dags test` more useful as a testing and debugging tool, we no longer run a backfill job and instead run a “local task runner”. Users can still backfill their DAGs using the `airflow dags backfill` command.

Airflow config section kubernetes renamed to kubernetes_executor (#26873)

KubernetesPodOperator no longer considers any core kubernetes config params, so this section now only applies to kubernetes executor. Renaming it reduces potential for confusion.

AirflowException is now thrown as soon as any dependent tasks of ExternalTaskSensor fails (#27190)

ExternalTaskSensor no longer hangs indefinitely when `failed_states` is set, an `execute_date_fn` is used, and some but not all of the dependent tasks fail. Instead, an `AirflowException` is thrown as soon as any of the dependent tasks fail. Any code handling this failure in addition to timeouts should move to caching the `AirflowException` BaseClass and not only the `AirflowSensorTimeout` subclass.

The Airflow config option scheduler.deactivate_stale_dags_interval has been renamed to scheduler.parsing_cleanup_interval (#27828).

The old option will continue to work but will issue deprecation warnings, and will be removed entirely in Airflow 3.

New Features

- TaskRunner: notify of component start and finish (#27855)
- Add DagRun state change to the Listener plugin system(#27113)
- Metric for raw task return codes (#27155)
- Add logic for XComArg to pull specific map indexes (#27771)
- Clear TaskGroup (#26658, #28003)
- Add critical section query duration metric (#27700)
- Add: #23880 :: Audit log for `AirflowModelViews(Variables/Connection)` (#24079, #27994, #27923)
- Add postgres 15 support (#27444)
- Expand tasks in mapped group at run time (#27491)
- reset commits, clean submodules (#27560)
- scheduler_job, add metric for scheduler loop timer (#27605)
- Allow datasets to be used in taskflow (#27540)
- Add expanded_ti_count to ti context (#27680)
- Add user comment to task instance and dag run (#26457, #27849, #27867)
- Enable copying DagRun JSON to clipboard (#27639)

- Implement extra controls for SLAs (#27557)
- add dag parsed time in DAG view (#27573)
- Add max_wait for exponential_backoff in BaseSensor (#27597)
- Expand tasks in mapped group at parse time (#27158)
- Add disable retry flag on backfill (#23829)
- Adding sensor decorator (#22562)
- Api endpoint update ti (#26165)
- Filtering datasets by recent update events (#26942)
- Support Is /not Null filter for value is None on webui (#26584)
- Add search to datasets list (#26893)
- Split out and handle ‘params’ in mapped operator (#26100)
- Add authoring API for TaskGroup mapping (#26844)
- Add one_done trigger rule (#26146)
- Create a more efficient airflow dag test command that also has better local logging (#26400)
- Support add/remove permissions to roles commands (#26338)
- Auto tail file logs in Web UI (#26169)
- Add triggerer info to task instance in API (#26249)
- Flag to deserialize value on custom XCom backend (#26343)

Improvements

- Allow depth-first execution (#27827)
- UI: Update offset height if data changes (#27865)
- Improve TriggerRuleDep typing and readability (#27810)
- Make views requiring session, keyword only args (#27790)
- Optimize TI.xcom_pull() with explicit task_ids and map_indexes (#27699)
- Allow hyphens in pod id used by k8s executor (#27737)
- optimise task instances filtering (#27102)
- Use context managers to simplify log serve management (#27756)
- Fix formatting leftovers (#27750)
- Improve task deadlock messaging (#27734)
- Improve “sensor timeout” messaging (#27733)
- Replace urlparse with urlsplit (#27389)
- Align TaskGroup semantics to AbstractOperator (#27723)
- Add new files to parsing queue on every loop of dag processing (#27060)
- Make Kubernetes Executor & Scheduler resilient to error during PMH execution (#27611)
- Separate dataset deps into individual graphs (#27356)
- Use log.exception where more economical than log.error (#27517)

- Move validation branch_task_ids into SkipMixin (#27434)
- Coerce LazyXComAccess to list when pushed to XCom (#27251)
- Update cluster-policies.rst docs (#27362)
- Add warning if connection type already registered within the provider (#27520)
- Activate debug logging in commands with –verbose option (#27447)
- Add classic examples for Python Operators (#27403)
- change .first() to .scalar() (#27323)
- Improve reset_dag_run description (#26755)
- Add examples and howtos about sensors (#27333)
- Make grid view widths adjustable (#27273)
- Sorting plugins custom menu links by category before name (#27152)
- Simplify DagRun.verify_integrity (#26894)
- Add mapped task group info to serialization (#27027)
- Correct the JSON style used for Run config in Grid View (#27119)
- No extra__conn_type__ prefix required for UI behaviors (#26995)
- Improve dataset update blurb (#26878)
- Rename kubernetes config section to kubernetes_executor (#26873)
- decode params for dataset searches (#26941)
- Get rid of the DAGRun details page & rely completely on Grid (#26837)
- Fix scheduler crashloopbackoff when using hostname_callable (#24999)
- Reduce log verbosity in KubernetesExecutor. (#26582)
- Don't iterate tis list twice for no reason (#26740)
- Clearer code for PodGenerator.deserialize_model_file (#26641)
- Don't import kubernetes unless you have a V1Pod (#26496)
- Add updated_at column to DagRun and Ti tables (#26252)
- Move the deserialization of custom XCom Backend to 2.4.0 (#26392)
- Avoid calculating all elements when one item is needed (#26377)
- Add __future__.annotations automatically by isort (#26383)
- Handle list when serializing expand_kwargs (#26369)
- Apply PEP-563 (Postponed Evaluation of Annotations) to core airflow (#26290)
- Add more weekday operator and sensor examples #26071 (#26098)
- Align TaskGroup semantics to AbstractOperator (#27723)

Bug Fixes

- Gracefully handle whole config sections being renamed (#28008)
- Add allow list for imports during deserialization (#27887)
- Soft delete datasets that are no longer referenced in DAG schedules or task outlets (#27828)
- Redirect to home view when there are no valid tags in the URL (#25715)
- Refresh next run datasets info in dags view (#27839)
- Make MappedTaskGroup depend on its expand inputs (#27876)
- Make DagRun state updates for paused DAGs faster (#27725)
- Don't explicitly set include_examples to False on task run command (#27813)
- Fix menu border color (#27789)
- Fix backfill queued task getting reset to scheduled state. (#23720)
- Fix clearing child dag mapped tasks from parent dag (#27501)
- Handle json encoding of V1Pod in task callback (#27609)
- Fix ExternalTaskSensor can't check zipped dag (#27056)
- Avoid re-fetching DAG run in TriggerDagRunOperator (#27635)
- Continue on exception when retrieving metadata (#27665)
- External task sensor fail fix (#27190)
- Add the default None when pop actions (#27537)
- Display parameter values from serialized dag in trigger dag view. (#27482, #27944)
- Move TriggerDagRun conf check to execute (#27035)
- Resolve trigger assignment race condition (#27072)
- Update google_analytics.html (#27226)
- Fix some bug in web ui dags list page (auto-refresh & jump search null state) (#27141)
- Fixed broken URL for docker-compose.yaml (#26721)
- Fix xcom arg.py .zip bug (#26636)
- Fix 404 taskInstance errors and split into two tables (#26575)
- Fix browser warning of improper thread usage (#26551)
- template rendering issue fix (#26390)
- Clear autoregistered DAGs if there are any import errors (#26398)
- Fix from airflow import version lazy import (#26239)
- allow scroll in triggered dag runs modal (#27965)

Misc/Internal

- Remove is_mapped attribute (#27881)
- Simplify FAB table resetting (#27869)
- Fix old-style typing in Base Sensor (#27871)
- Switch (back) to late imports (#27730)

- Completed D400 for multiple folders (#27748)
- simplify notes accordion test (#27757)
- completed D400 for airflow/callbacks/* airflow/cli/* (#27721)
- Completed D400 for airflow/api_connexion/* directory (#27718)
- Completed D400 for airflow/listener/* directory (#27731)
- Completed D400 for airflow/lineage/* directory (#27732)
- Update API & Python Client versions (#27642)
- Completed D400 & D401 for airflow/api/* directory (#27716)
- Completed D400 for multiple folders (#27722)
- Bump minimatch from 3.0.4 to 3.0.8 in /airflow/www (#27688)
- Bump loader-utils from 1.4.1 to 1.4.2 ``in ``/airflow/www (#27697)
- Disable nested task mapping for now (#27681)
- bump alembic minimum version (#27629)
- remove unused code.html (#27585)
- Enable python string normalization everywhere (#27588)
- Upgrade dependencies in order to avoid backtracking (#27531)
- Strengthen a bit and clarify importance of triaging issues (#27262)
- Deduplicate type hints (#27508)
- Add stub ‘yield’ to BaseTrigger.run (#27416)
- Remove upper-bound limit to dask (#27415)
- Limit Dask to under 2022.10.1 (#27383)
- Update old style typing (#26872)
- Enable string normalization for docs (#27269)
- Slightly faster up/downgrade tests (#26939)
- Deprecate use of core get_kube_client in PodManager (#26848)
- Add memray files to gitignore / dockerignore (#27001)
- Bump sphinx and sphinx-autoapi (#26743)
- Simplify RTIF.delete_old_records() (#26667)
- migrate last react files to typescript (#26112)
- Work around pyupgrade edge cases (#26384)

Doc only changes

- Document dag_file_processor_timeouts metric as deprecated (#27067)
- Drop support for PostgreSQL 10 (#27594)
- Update index.rst (#27529)
- Add note about pushing the lazy XCom proxy to XCom (#27250)
- Fix BaseOperator link (#27441)

- [docs] best-practices add use variable with template example. (#27316)
- docs for custom view using plugin (#27244)
- Update graph view and grid view on overview page (#26909)
- Documentation fixes (#26819)
- make consistency on markup title string level (#26696)
- Add documentation to dag test function (#26713)
- Fix broken URL for `docker-compose.yaml` (#26726)
- Add a note against use of top level code in timetable (#26649)
- Fix example_datasets dag names (#26495)
- Update docs: zip-like effect is now possible in task mapping (#26435)
- changing to task decorator in docs from classic operator use (#25711)

3.16.29 Airflow 2.4.3 (2022-11-14)

Significant Changes

Make RotatingFilehandler used in DagProcessor non-caching (#27223)

In case you want to decrease cache memory when `CONFIG_PROCESSOR_MANAGER_LOGGER=True`, and you have your local settings created before, you can update `processor_manager_handler` to use `airflow.utils.log.non_caching_file_handler.NonCachingRotatingFileHandler` handler instead of `logging.RotatingFileHandler`.

Bug Fixes

- Fix double logging with some task logging handler (#27591)
- Replace FAB url filtering function with Airflow's (#27576)
- Fix mini scheduler expansion of mapped task (#27506)
- SLAMiss is nullable and not always given back when pulling task instances (#27423)
- Fix behavior of `_` when searching for DAGs (#27448)
- Fix getting the dag/task ids from BaseExecutor (#27550)
- Fix SQLAlchemy primary key black-out error on DDRQ (#27538)
- Fix IntegrityError during webserver startup (#27297)
- Add case insensitive constraint to username (#27266)
- Fix python external template keys (#27256)
- Reduce extraneous task log requests (#27233)
- Make RotatingFilehandler used in DagProcessor non-caching (#27223)
- Listener: Set task on SQLAlchemy TaskInstance object (#27167)
- Fix dags list page auto-refresh & jump search null state (#27141)
- Set `executor.job_id` to `BackfillJob.id` for backfills (#27020)

Misc/Internal

- Bump loader-utils from 1.4.0 to 1.4.1 in /airflow/www (#27552)
- Reduce log level for k8s TCP_KEEPALIVE etc warnings (#26981)

Doc only changes

- Use correct executable in docker compose docs (#27529)
- Fix wording in DAG Runs description (#27470)
- Document that `KubernetesExecutor` overwrites container args (#27450)
- Fix `BaseOperator` links (#27441)
- Correct timer units to seconds from milliseconds. (#27360)
- Add missed import in the Trigger Rules example (#27309)
- Update SLA wording to reflect it is relative to Dag Run start. (#27111)
- Add `kerberos` environment variables to the docs (#27028)

3.16.30 Airflow 2.4.2 (2022-10-23)

Significant Changes

Default for [webserver] expose_stacktrace changed to False (#27059)

The default for [webserver] expose_stacktrace has been set to False, instead of True. This means administrators must opt-in to expose tracebacks to end users.

Bug Fixes

- Make tracebacks opt-in (#27059)
- Add missing AUTOINC/SERIAL for FAB tables (#26885)
- Add separate error handler for 405(Method not allowed) errors (#26880)
- Don't re-patch pods that are already controlled by current worker (#26778)
- Handle mapped tasks in task duration chart (#26722)
- Fix task duration cumulative chart (#26717)
- Avoid 500 on dag redirect (#27064)
- Filter dataset dependency data on webserver (#27046)
- Remove double collection of dags in `airflow dags reserialize` (#27030)
- Fix auto refresh for graph view (#26926)
- Don't overwrite connection extra with invalid json (#27142)
- Fix next run dataset modal links (#26897)
- Change dag audit log sort by date from asc to desc (#26895)
- Bump min version of jinja2 (#26866)
- Add missing colors to `state_color_mapping` jinja global (#26822)
- Fix running debuggers inside `airflow tasks test` (#26806)
- Fix warning when using xcomarg dependencies (#26801)

- demote Removed state in priority for displaying task summaries (#26789)
- Ensure the log messages from operators during parsing go somewhere (#26779)
- Add restarting state to TaskState Enum in REST API (#26776)
- Allow retrieving error message from data.detail (#26762)
- Simplify origin string cleaning (#27143)
- Remove DAG parsing from StandardTaskRunner (#26750)
- Fix non-hidden cumulative chart on duration view (#26716)
- Remove TaskFail duplicates check (#26714)
- Fix airflow tasks run –local when dags_folder differs from that of processor (#26509)
- Fix yarn warning from d3-color (#27139)
- Fix version for a couple configurations (#26491)
- Revert “No grid auto-refresh for backfill dag runs (#25042)” (#26463)
- Retry on Airflow Schedule DAG Run DB Deadlock (#26347)

Misc/Internal

- Clean-ups around task-mapping code (#26879)
- Move user-facing string to template (#26815)
- add icon legend to datasets graph (#26781)
- Bump sphinx and sphinx-autoapi (#26743)
- Simplify RTIF.delete_old_records() (#26667)
- Bump FAB to 4.1.4 (#26393)

Doc only changes

- Fixed triple quotes in task group example (#26829)
- Documentation fixes (#26819)
- make consistency on markup title string level (#26696)
- Add a note against use of top level code in timetable (#26649)
- Fix broken URL for docker-compose.yaml (#26726)

3.16.31 Airflow 2.4.1 (2022-09-30)

Significant Changes

No significant changes.

Bug Fixes

- When rendering template, unmap task in context (#26702)
- Fix scroll overflow for ConfirmDialog (#26681)
- Resolve deprecation warning re Table.exists() (#26616)
- Fix XComArg zip bug (#26636)

- Use COALESCE when ordering runs to handle NULL (#26626)
- Check user is active (#26635)
- No missing user warning for public admin (#26611)
- Allow MapXComArg to resolve after serialization (#26591)
- Resolve warning about DISTINCT ON query on dags view (#26608)
- Log warning when secret backend kwargs is invalid (#26580)
- Fix grid view log try numbers (#26556)
- Template rendering issue in passing `templates_dict` to task decorator (#26390)
- Fix Deferrable stuck as `scheduled` during backfill (#26205)
- Suppress `SQLALCHEMY_TRACK_MODIFICATIONS` warning in db init (#26617)
- Correctly set `json_provider_class` on Flask app so it uses our encoder (#26554)
- Fix WSGI root app (#26549)
- Fix deadlock when mapped task with removed upstream is rerun (#26518)
- ExecutorConfigType should be `cacheable` (#26498)
- Fix proper joining of the path for logs retrieved from celery workers (#26493)
- DAG Deps extends `base_template` (#26439)
- Don't update backfill run from the scheduler (#26342)

Doc only changes

- Clarify owner links document (#26515)
- Fix invalid RST in dataset concepts doc (#26434)
- Document the `non-sensitive-only` option for `expose_config` (#26507)
- Fix `example_datasets` dag names (#26495)
- Zip-like effect is now possible in task mapping (#26435)
- Use task decorator in docs instead of classic operators (#25711)

3.16.32 Airflow 2.4.0 (2022-09-19)

Significant Changes

Data-aware Scheduling and Dataset concept added to Airflow

New to this release of Airflow is the concept of Datasets to Airflow, and with it a new way of scheduling dags: data-aware scheduling.

This allows DAG runs to be automatically created as a result of a task “producing” a dataset. In some ways this can be thought of as the inverse of `TriggerDagRunOperator`, where instead of the producing DAG controlling which DAGs get created, the consuming DAGs can “listen” for changes.

A dataset is identified by a URI:

```
from airflow import Dataset

# The URI doesn't have to be absolute
dataset = Dataset(uri="my-dataset")
# Or you can use a scheme to show where it lives.
dataset2 = Dataset(uri="s3://bucket/prefix")
```

To create a DAG that runs whenever a Dataset is updated use the new `schedule` parameter (see below) and pass a list of 1 or more Datasets:

```
with DAG(dag_id='dataset-consumer', schedule=[dataset]):
    ...
```

And to mark a task as producing a dataset pass the `dataset(s)` to the `outlets` attribute:

```
@task(outlets=[dataset])
def my_task():
    ...

# Or for classic operators
BashOperator(task_id="update-ds", bash_command=..., outlets=[dataset])
```

If you have the producer and consumer in different files you do not need to use the same `Dataset` object, two `Dataset()`s created with the same URI are equal.

Datasets represent the abstract concept of a dataset, and (for now) do not have any direct read or write capability - in this release we are adding the foundational feature that we will build upon.

For more info on Datasets please see [Datasets documentation](#).

Expanded dynamic task mapping support

Dynamic task mapping now includes support for `expand_kwargs`, `zip` and `map`.

For more info on dynamic task mapping please see [Dynamic Task Mapping](#).

DAGS used in a context manager no longer need to be assigned to a module variable (#23592)

Previously you had to assign a DAG to a module-level variable in order for Airflow to pick it up. For example this

```
with DAG(dag_id="example") as dag:
    ...

@dag
def dag_maker():
    ...

dag2 = dag_maker()
```

can become

```
with DAG(dag_id="example"):
    ...
```

(continues on next page)

(continued from previous page)

```
@dag
def dag_maker(): ...
```

```
dag_maker()
```

If you want to disable the behaviour for any reason then set `auto_register=False` on the dag:

```
# This dag will not be picked up by Airflow as it's not assigned to a variable
with DAG(dag_id="example", auto_register=False):
    ...
```

Deprecation of `schedule_interval` and `timetable` arguments (#25410)

We added new DAG argument `schedule` that can accept a cron expression, `timedelta` object, `timetable` object, or list of dataset objects. Arguments `schedule_interval` and `timetable` are deprecated.

If you previously used the `@daily` cron preset, your DAG may have looked like this:

```
with DAG(
    dag_id="my_example",
    start_date=datetime(2021, 1, 1),
    schedule_interval="@daily",
):
    ...
```

Going forward, you should use the `schedule` argument instead:

```
with DAG(
    dag_id="my_example",
    start_date=datetime(2021, 1, 1),
    schedule="@daily",
):
    ...
```

The same is true if you used a custom timetable. Previously you would have used the `timetable` argument:

```
with DAG(
    dag_id="my_example",
    start_date=datetime(2021, 1, 1),
    timetable=EventsTimetable(event_dates=[pendulum.datetime(2022, 4, 5)]),
):
    ...
```

Now you should use the `schedule` argument:

```
with DAG(
    dag_id="my_example",
    start_date=datetime(2021, 1, 1),
    schedule=EventsTimetable(event_dates=[pendulum.datetime(2022, 4, 5)]),
):
    ...
```

Removal of experimental Smart Sensors (#25507)

Smart Sensors were added in 2.0 and deprecated in favor of Deferrable operators in 2.2, and have now been removed.

airflow.contrib packages and deprecated modules are dynamically generated (#26153, #26179, #26167)

The `airflow.contrib` packages and deprecated modules from Airflow 1.10 in `airflow.hooks`, `airflow.operators`, `airflow.sensors` packages are now dynamically generated modules and while users can continue using the deprecated contrib classes, they are no longer visible for static code check tools and will be reported as missing. It is recommended for the users to move to the non-deprecated classes.

DBApiHook and SQLSensor have moved (#24836)

DBApiHook and SQLSensor have been moved to the `apache-airflow-providers-common-sql` provider.

DAG runs sorting logic changed in grid view (#25090)

The ordering of DAG runs in the grid view has been changed to be more “natural”. The new logic generally orders by data interval, but a custom ordering can be applied by setting the DAG to use a custom timetable.

New Features

- Add Data-aware Scheduling ([AIP-48](#))
- Add `@task.short_circuit` TaskFlow decorator (#25752)
- Make `execution_date_or_run_id` optional in `tasks test` command (#26114)
- Automatically register DAGs that are used in a context manager (#23592, #26398)
- Add option of sending DAG parser logs to stdout. (#25754)
- Support multiple `DagProcessors` parsing files from different locations. (#25935)
- Implement `ExternalPythonOperator` (#25780)
- Make `execution_date` optional for command `dags test` (#26111)
- Implement `expand_kwargs()` against a literal list (#25925)
- Add trigger rule tooltip (#26043)
- Add `conf` parameter to CLI for airflow dags test (#25900)
- Include scheduled slots in pools view (#26006)
- Add `output` property to `MappedOperator` (#25604)
- Add roles delete command to cli (#25854)
- Add Airflow specific warning classes (#25799)
- Add support for `TaskGroup` in `ExternalTaskSensor` (#24902)
- Add `@task.kubernetes` taskflow decorator (#25663)
- Add a way to import Airflow without side-effects (#25832)
- Let timetables control generated run_ids. (#25795)
- Allow per-timetable ordering override in grid view (#25633)
- Grid logs for mapped instances (#25610, #25621, #25611)

- Consolidate to one `schedule` param (#25410)
- DAG regex flag in backfill command (#23870)
- Adding support for owner links in the Dags view UI (#25280)
- Ability to clear a specific DAG Run's task instances via REST API (#23516)
- Possibility to document DAG with a separate markdown file (#25509)
- Add parsing context to DAG Parsing (#25161)
- Implement `CronTriggerTimetable` (#23662)
- Add option to mask sensitive data in UI configuration page (#25346)
- Create new databases from the ORM (#24156)
- Implement `XComArg.zip(*xcom_args)` (#25176)
- Introduce `sla_miss` metric (#23402)
- Implement `map()` semantic (#25085)
- Add override method to `TaskGroupDecorator` (#25160)
- Implement `expand_kwargs()` (#24989)
- Add parameter to turn off SQL query logging (#24570)
- Add `DagWarning` model, and a check for missing pools (#23317)
- Add Task Logs to Grid details panel (#24249)
- Added small health check server and endpoint in scheduler (#23905)
- Add built-in External Link for `ExternalTaskMarker` operator (#23964)
- Add default task retry delay config (#23861)
- Add clear `DagRun` endpoint. (#23451)
- Add support for timezone as string in cron interval timetable (#23279)
- Add auto-refresh to dags home page (#22900, #24770)

Improvements

- Add more weekday operator and sensor examples #26071 (#26098)
- Add `subdir` parameter to dags reserialize command (#26170)
- Update zombie message to be more descriptive (#26141)
- Only send an `SlaCallbackRequest` if the DAG is scheduled (#26089)
- Promote `Operator.output` more (#25617)
- Upgrade API files to typescript (#25098)
- Less hacky double-rendering prevention in mapped task (#25924)
- Improve Audit log (#25856)
- Remove mapped operator validation code (#25870)
- More `DAG(schedule=...)` improvements (#25648)
- Reduce `operator_name` dupe in serialized JSON (#25819)
- Make grid view group/mapped summary UI more consistent (#25723)

- Remove useless statement in `task_group_to_grid` (#25654)
- Add optional data interval to `CronTriggerTimetable` (#25503)
- Remove unused code in `/grid` endpoint (#25481)
- Add and document description fields (#25370)
- Improve Airflow logging for operator Jinja template processing (#25452)
- Update core example DAGs to use `@task.branch` decorator (#25242)
- Update DAG `audit_log` route (#25415)
- Change `stdout` and `stderr` access mode to append in commands (#25253)
- Remove `getTasks` from Grid view (#25359)
- Improve taskflow type hints with `ParamSpec` (#25173)
- Use tables in grid details panes (#25258)
- Explicitly list `@dag` arguments (#25044)
- More typing in `SchedulerJob` and `TaskInstance` (#24912)
- Patch `getfqdn` with more resilient version (#24981)
- Replace all NBSP characters by whitespaces (#24797)
- Re-serialize all DAGs on `airflow db upgrade` (#24518)
- Rework contract of `try_adopt_task_instances` method (#23188)
- Make `expand()` error vague so it's not misleading (#24018)
- Add enum validation for `[webserver]analytics_tool` (#24032)
- Add `dtm` searchable field in audit log (#23794)
- Allow more parameters to be piped through via `execute_in_subprocess` (#23286)
- Use `func.count` to count rows (#23657)
- Remove stale serialized dags (#22917)
- AIP45 Remove dag parsing in airflow run local (#21877)
- Add support for queued state in `DagRun` update endpoint. (#23481)
- Add fields to `dagrun` endpoint (#23440)
- Use `sqlalchemy_conn` for celery result backend when `result_backend` is not set (#24496)

Bug Fixes

- Have consistent types between the ORM and the migration files (#24044, #25869)
- Disallow any dag tags longer than 100 char (#25196)
- Add the `dag_id` to `AirflowDagCycleException` message (#26204)
- Properly build URL to retrieve logs independently from system (#26337)
- For worker log servers only bind to IPV6 when dual stack is available (#26222)
- Fix `TaskInstance.task` not defined before `handle_failure` (#26040)
- Undo secrets backend config caching (#26223)
- Fix faulty executor config serialization logic (#26191)

- Show DAGs and Datasets menu links based on role permission (#26183)
- Allow setting TaskGroup tooltip via function docstring (#26028)
- Fix RecursionError on graph view of a DAG with many tasks (#26175)
- Fix backfill occasional deadlocking (#26161)
- Fix DagRun.start_date not set during backfill with --reset-dagruns True (#26135)
- Use label instead of id for dynamic task labels in graph (#26108)
- Don't fail DagRun when leaf mapped_task is SKIPPED (#25995)
- Add group prefix to decorated mapped task (#26081)
- Fix UI flash when triggering with dup logical date (#26094)
- Fix Make items nullable for TaskInstance related endpoints to avoid API errors (#26076)
- Fix BranchDateTimeOperator to be timezone-awareness-insensitive (#25944)
- Fix legacy timetable schedule interval params (#25999)
- Fix response schema for list-mapped-task-instance (#25965)
- Properly check the existence of missing mapped TIs (#25788)
- Fix broken auto-refresh on grid view (#25950)
- Use per-timetable ordering in grid UI (#25880)
- Rewrite recursion when parsing DAG into iteration (#25898)
- Find cross-group tasks in iter_mapped_dependants (#25793)
- Fail task if mapping upstream fails (#25757)
- Support / in variable get endpoint (#25774)
- Use cfg default_wrap value for grid logs (#25731)
- Add origin request args when triggering a run (#25729)
- Operator name separate from class (#22834)
- Fix incorrect data interval alignment due to assumption on input time alignment (#22658)
- Return None if an XComArg fails to resolve (#25661)
- Correct json arg help in airflow variables set command (#25726)
- Added MySQL index hint to use ti_state on find_zombies query (#25725)
- Only excluded actually expanded fields from render (#25599)
- Grid, fix toast for axios errors (#25703)
- Fix UI redirect (#26409)
- Require dag_id arg for dags list-runs (#26357)
- Check for queued states for dags auto-refresh (#25695)
- Fix upgrade code for the dag_owner_attributes table (#25579)
- Add map index to task logs api (#25568)
- Ensure that zombie tasks for dags with errors get cleaned up (#25550)
- Make extra link work in UI (#25500)

- Sync up plugin API schema and definition (#25524)
- First/last names can be empty (#25476)
- Refactor DAG pages to be consistent (#25402)
- Check `expand_kwargs()` input type before unmapping (#25355)
- Filter XCOM by key when calculating map lengths (#24530)
- Fix `ExternalTaskSensor` not working with dynamic task (#25215)
- Added exception catching to send default email if template file raises any exception (#24943)
- Bring `MappedOperator` members in sync with `BaseOperator` (#24034)

Misc/Internal

- Add automatically generated ERD schema for the `MetaData` DB (#26217)
- Mark serialization functions as internal (#26193)
- Remove remaining deprecated classes and replace them with PEP562 (#26167)
- Move `dag_edges` and `task_group_to_dict` to corresponding util modules (#26212)
- Lazily import many modules to improve import speed (#24486, #26239)
- FIX Incorrect typing information (#26077)
- Add missing contrib classes to deprecated dictionaries (#26179)
- Re-configure/connect the ORM after forking to run a DAG processor (#26216)
- Remove cattrs from lineage processing. (#26134)
- Removed deprecated contrib files and replace them with PEP-562 `getattr` (#26153)
- Make `BaseSerialization.serialize` “public” to other classes. (#26142)
- Change the template to use human readable `task_instance` description (#25960)
- Bump `moment-timezone` from `0.5.34` to `0.5.35` in `/airflow/www` (#26080)
- Fix Flask deprecation warning (#25753)
- Add CamelCase to generated operations types (#25887)
- Fix migration issues and tighten the CI upgrade/downgrade test (#25869)
- Fix type annotations in `SkipMixin` (#25864)
- Workaround setuptools editable packages path issue (#25848)
- Bump `undici` from `5.8.0` to `5.9.1` in `/airflow/www` (#25801)
- Add `custom_operator_name` attr to `_BranchPythonDecoratedOperator` (#25783)
- Clarify `filename_template` deprecation message (#25749)
- Use `ParamSpec` to replace `...` in `Callable` (#25658)
- Remove deprecated modules (#25543)
- Documentation on task mapping additions (#24489)
- Remove Smart Sensors (#25507)
- Fix `elasticsearch` test config to avoid warning on deprecated template (#25520)
- Bump `terser` from `4.8.0` to `4.8.1` in `/airflow/ui` (#25178)

- Generate `typescript` types from rest API docs (#25123)
- Upgrade utils files to `typescript` (#25089)
- Upgrade remaining context file to `typescript`. (#25096)
- Migrate files to `ts` (#25267)
- Upgrade grid Table component to `ts`. (#25074)
- Skip mapping against mapped `ti` if it returns None (#25047)
- Refactor `js` file structure (#25003)
- Move mapped kwargs introspection to separate type (#24971)
- Only assert stuff for mypy when type checking (#24937)
- Bump `moment` from 2.29.3 to 2.29.4 in `/airflow/www` (#24885)
- Remove “bad characters” from our codebase (#24841)
- Remove `xcom_push` flag from `BashOperator` (#24824)
- Move Flask hook registration to end of file (#24776)
- Upgrade more javascript files to `typescript` (#24715)
- Clean up task decorator type hints and docstrings (#24667)
- Preserve original order of providers’ connection extra fields in UI (#24425)
- Rename `charts.css` to `chart.css` (#24531)
- Rename `grid.css` to `chart.css` (#24529)
- Misc: create new process group by `set_new_process_group` utility (#24371)
- Airflow UI fix Prototype Pollution (#24201)
- Bump `moto` version (#24222)
- Remove unused `[github_enterprise]` from ref docs (#24033)
- Clean up f-strings in logging calls (#23597)
- Add limit for `JType1` (#23847)
- Simply json responses (#25518)
- Add min attrs version (#26408)

Doc only changes

- Add url prefix setting for `Celery` Flower (#25986)
- Updating deprecated configuration in examples (#26037)
- Fix wrong link for taskflow tutorial (#26007)
- Reorganize tutorials into a section (#25890)
- Fix concept doc for dynamic task map (#26002)
- Update code examples from “classic” operators to taskflow (#25845, #25657)
- Add instructions on manually fixing MySQL Charset problems (#25938)
- Prefer the local Quick Start in docs (#25888)
- Fix broken link to `Trigger Rules` (#25840)

- Improve docker documentation (#25735)
- Correctly link to Dag parsing context in docs (#25722)
- Add note on `task_instance_mutation_hook` usage (#25607)
- Note that TaskFlow API automatically passes data between tasks (#25577)
- Update DAG run to clarify when a DAG actually runs (#25290)
- Update tutorial docs to include a definition of operators (#25012)
- Rewrite the Airflow documentation home page (#24795)
- Fix `task-generated mapping` example (#23424)
- Add note on subtle logical date change in 2.2.0 (#24413)
- Add missing import in best-practices code example (#25391)

3.16.33 Airflow 2.3.4 (2022-08-23)

Significant Changes

Added new config [logging]log_formatter_class to fix timezone display for logs on UI (#24811)

If you are using a custom Formatter subclass in your [logging]logging_config_class, please inherit from `airflow.utils.log.timezone_aware.TimezoneAware` instead of `logging.Formatter`. For example, in your `custom_config.py`:

```
from airflow.utils.log.timezone_aware import TimezoneAware

# before
class YourCustomFormatter(logging.Formatter): ...

# after
class YourCustomFormatter(TimezoneAware): ...

AIRFLOW_FORMATTER = LOGGING_CONFIG["formatters"]["airflow"]
AIRFLOW_FORMATTER["class"] = "somewhere.your.custom_config.YourCustomFormatter"
# or use TimezoneAware class directly. If you don't have custom Formatter.
AIRFLOW_FORMATTER["class"] = "airflow.utils.log.timezone_aware.TimezoneAware"
```

Bug Fixes

- Disable `attrs` state management on `MappedOperator` (#24772)
- Serialize `pod_override` to JSON before pickling `executor_config` (#24356)
- Fix pid check (#24636)
- Rotate session id during login (#25771)
- Fix mapped sensor with reschedule mode (#25594)
- Cache the custom secrets backend so the same instance gets reused (#25556)
- Add right padding (#25554)
- Fix reducing mapped length of a mapped task at runtime after a clear (#25531)

- Fix `airflow db reset` when dangling tables exist (#25441)
- Change `disable_verify_ssl` behaviour (#25023)
- Set default task group in `dag.add_task` method (#25000)
- Removed interfering force of index. (#25404)
- Remove useless logging line (#25347)
- Adding mysql index hint to use index on `task_instance.state` in critical section query (#25673)
- Configurable umask to all daemonized processes. (#25664)
- Fix the errors raised when None is passed to template filters (#25593)
- Allow wildcared CORS origins (#25553)
- Fix “This Session’s transaction has been rolled back” (#25532)
- Fix Serialization error in `TaskCallbackRequest` (#25471)
- fix - resolve bash by absolute path (#25331)
- Add `__repr__` to `ParamsDict` class (#25305)
- Only load distribution of a name once (#25296)
- convert `TimeSensorAsync target_time` to utc on call time (#25221)
- call `updateNodeLabels` after `expandGroup` (#25217)
- Stop SLA callbacks gazumping other callbacks and DOS’ing the `DagProcessorManager` queue (#25147)
- Fix `invalidateQueries` call (#25097)
- `airflow/www/package.json`: Add name, version fields. (#25065)
- No grid auto-refresh for backfill dag runs (#25042)
- Fix tag link on dag detail page (#24918)
- Fix zombie task handling with multiple schedulers (#24906)
- Bind log server on worker to IPv6 address (#24755) (#24846)
- Add `%z` for `%(asctime)s` to fix timezone for logs on UI (#24811)
- `TriggerDagRunOperator.operator_extra_links` is attr (#24676)
- Send DAG timeout callbacks to processor outside of `prohibit_commit` (#24366)
- Don’t rely on current ORM structure for db clean command (#23574)
- Clear next method when clearing TIs (#23929)
- Two typing fixes (#25690)

Doc only changes

- Update set-up-database.rst (#24983)
- Fix syntax in mysql setup documentation (#24893 (#24939))
- Note how DAG policy works with `default_args` (#24804)
- Update PythonVirtualenvOperator Howto (#24782)
- Doc: Add hyperlinks to Github PRs for Release Notes (#24532)

Misc/Internal

- Remove depreciation warning when use default remote tasks logging handlers (#25764)
- clearer method name in scheduler_job.py (#23702)
- Bump cattrs version (#25689)
- Include missing mention of `external_executor_id` in `sql_engine_collation_for_ids` docs (#25197)
- Refactor DR.`task_instance_scheduling_decisions` (#24774)
- Sort operator extra links (#24992)
- Extends `resolve_xcom_backend` function level documentation (#24965)
- Upgrade FAB to 4.1.3 (#24884)
- Limit Flask to <2.3 in the wake of 2.2 breaking our tests (#25511)
- Limit astroid version to < 2.12 (#24982)
- Move javascript compilation to host (#25169)
- Bump typing-extensions and mypy for ParamSpec (#25088)

3.16.34 Airflow 2.3.3 (2022-07-09)

Significant Changes

We've upgraded Flask App Builder to a major version 4.* (#24399)

Flask App Builder is one of the important components of Airflow Webserver, as it uses a lot of dependencies that are essential to run the webserver and integrate it in enterprise environments - especially authentication.

The FAB 4.* upgrades a number of dependencies to major releases, which upgrades them to versions that have a number of security issues fixed. A lot of tests were performed to bring the dependencies in a backwards-compatible way, however the dependencies themselves implement breaking changes in their internals so it might be that some of those changes might impact the users in case they are using the libraries for their own purposes.

One important change that you likely will need to apply to Oauth configuration is to add `server_metadata_url` or `jwks_uri` and you can read about it more in [this issue](#).

Here is the list of breaking changes in dependencies that comes together with FAB 4:

- Flask from 1.X to 2.X [breaking changes](#)
- flask-jwt-extended 3.X to 4.X [breaking changes](#):
- Jinja2 2.X to 3.X [breaking changes](#):
- Werkzeug 1.X to 2.X [breaking changes](#)
- pyJWT 1.X to 2.X [breaking changes](#):
- Click 7.X to 8.X [breaking changes](#):
- itsdangerous 1.X to 2.X [breaking changes](#)

Bug Fixes

- Fix exception in mini task scheduler (#24865)
- Fix cycle bug with attaching label to task group (#24847)
- Fix timestamp defaults for `sensorinstance` (#24638)

- Move fallible `ti.task.dag` assignment back inside `try/except` block (#24533) (#24592)
- Add missing types to `FSHook` (#24470)
- Mask secrets in `stdout` for `airflow tasks test` (#24362)
- `DebugExecutor` use `ti.run()` instead of `ti._run_raw_task` (#24357)
- Fix bugs in URI constructor for MySQL connection (#24320)
- Missing `scheduleinterval` nullable true added in `openapi` (#24253)
- Unify `return_code` interface for task runner (#24093)
- Handle occasional deadlocks in trigger with retries (#24071)
- Remove special serde logic for mapped `op_kwargs` (#23860)
- `ExternalTaskSensor` respects `soft_fail` if the external task enters a `failed_state` (#23647)
- Fix `StatD` timing metric units (#21106)
- Add `cache_ok` flag to sqlalchemy TypeDecorators. (#24499)
- Allow for `LOGGING_LEVEL=DEBUG` (#23360)
- Fix grid date ticks (#24738)
- Debounce status highlighting in Grid view (#24710)
- Fix Grid vertical scrolling (#24684)
- don't try to render child rows for closed groups (#24637)
- Do not calculate grid root instances (#24528)
- Maintain grid view selection on filtering upstream (#23779)
- Speed up `grid_data` endpoint by 10x (#24284)
- Apply per-run log templates to log handlers (#24153)
- Don't crash scheduler if exec config has old k8s objects (#24117)
- `TI.log_url` fix for `map_index` (#24335)
- Fix migration `0080_2_0_2` - Replace null values before setting column not null (#24585)
- Patch `sqlalchemy_conn` if old Postgres schemes used (#24569)
- Seed `log_template` table (#24511)
- Fix deprecated `log_id_template` value (#24506)
- Fix toast messages (#24505)
- Add indexes for CASCADE deletes for `task_instance` (#24488)
- Return empty dict if Pod JSON encoding fails (#24478)
- Improve grid rendering performance with a custom tooltip (#24417, #24449)
- Check for `run_id` for grid group summaries (#24327)
- Optimize calendar view for cron scheduled DAGs (#24262)
- Use `get_hostname` instead of `socket.getfqdn` (#24260)
- Check that edge nodes actually exist (#24166)
- Fix `useTasks` crash on error (#24152)

- Do not fail re-queued TIs (#23846)
- Reduce grid view API calls (#24083)
- Rename Permissions to Permission Pairs. (#24065)
- Replace `use_task_execution_date` with `use_task_logical_date` (#23983)
- Grid fix details button truncated and small UI tweaks (#23934)
- Add TaskInstance State REMOVED to finished states and success states (#23797)
- Fix mapped task immutability after clear (#23667)
- Fix permission issue for dag that has dot in name (#23510)
- Fix closing connection `dbapi.get_pandas_df` (#23452)
- Check bag DAG `schedule_interval` match timetable (#23113)
- Parse error for task added to multiple groups (#23071)
- Fix flaky order of returned dag runs (#24405)
- Migrate `jsx` files that affect run/task selection to `tsx` (#24509)
- Fix links to sources for examples (#24386)
- Set proper Content-Type and charset on `grid_data` endpoint (#24375)

Doc only changes

- Update templates doc to mention `extras` and format Airflow Vars / Conns (#24735)
- Document built in Timetables (#23099)
- Alphabetizes two tables (#23923)
- Clarify that users should not use Maria DB (#24556)
- Add imports to deferring code samples (#24544)
- Add note about image regeneration in June 2022 (#24524)
- Small cleanup of `get_current_context()` chapter (#24482)
- Fix default 2.2.5 `log_id_template` (#24455)
- Update description of installing providers separately from core (#24454)
- Mention context variables and logging (#24304)

Misc/Internal

- Remove internet explorer support (#24495)
- Removing magic status code numbers from `api_connexion` (#24050)
- Upgrade FAB to 4.1.2 (#24619)
- Switch Markdown engine to `markdown-it-py` (#19702)
- Update `rich` to latest version across the board. (#24186)
- Get rid of `TimedJSONWebSignatureSerializer` (#24519)
- Update `flask-appbuilder authlib/ oauth` dependency (#24516)
- Upgrade to `webpack` 5 (#24485)

- Add typescript (#24337)
- The JWT claims in the request to retrieve logs have been standardized: we use nbf and aud claims for maturity and audience of the requests. Also “filename” payload field is used to keep log name. (#24519)
- Address all yarn test warnings (#24722)
- Upgrade to react 18 and chakra 2 (#24430)
- Refactor DagRun.verify_integrity (#24114)
- Upgrade FAB to 4.1.1 (#24399)
- We now need at least Flask-WTF 0.15 (#24621)

3.16.35 Airflow 2.3.2 (2022-06-04)

No significant changes.

Bug Fixes

- Run the check_migration loop at least once
- Fix grid view for mapped tasks (#24059)
- Icons in grid view for different DAG run types (#23970)
- Faster grid view (#23951)
- Disallow calling expand with no arguments (#23463)
- Add missing is_mapped field to Task response. (#23319)
- DagFileProcessorManager: Start a new process group only if current process not a session leader (#23872)
- Mask sensitive values for not-yet-running TIs (#23807)
- Add cascade to dag_tag to dag foreign key (#23444)
- Use --subdir argument value for standalone dag processor. (#23864)
- Highlight task states by hovering on legend row (#23678)
- Fix and speed up grid view (#23947)
- Prevent UI from crashing if grid task instances are null (#23939)
- Remove redundant register exit signals in dag-processor command (#23886)
- Add __wrapped__ property to _TaskDecorator (#23830)
- Fix UnboundLocalError when sql is empty list in DbApiHook (#23816)
- Enable clicking on DAG owner in autocomplete dropdown (#23804)
- Simplify flash message for _airflow_moved tables (#23635)
- Exclude missing tasks from the gantt view (#23627)

Doc only changes

- Add column names for DB Migration Reference (#23853)

Misc/Internal

- Remove pinning for `xmldict` (#23992)

3.16.36 Airflow 2.3.1 (2022-05-25)

Significant Changes

No significant changes.

Bug Fixes

- Automatically reschedule stalled queued tasks in `CeleryExecutor` (#23690)
- Fix expand/collapse all buttons (#23590)
- Grid view status filters (#23392)
- Expand/collapse all groups (#23487)
- Fix retrieval of deprecated non-config values (#23723)
- Fix secrets rendered in UI when task is not executed. (#22754)
- Fix provider import error matching (#23825)
- Fix regression in ignoring symlinks (#23535)
- Fix `dag-processor` fetch metadata database config (#23575)
- Fix auto upstream dep when expanding non-templated field (#23771)
- Fix task log is not captured (#23684)
- Add `reschedule` to the serialized fields for the `BaseSensorOperator` (#23674)
- Modify db clean to also catch the `ProgrammingError` exception (#23699)
- Remove titles from link buttons (#23736)
- Fix grid details header text overlap (#23728)
- Ensure `execution_timeout` as `timedelta` (#23655)
- Don't run pre-migration checks for downgrade (#23634)
- Add index for event column in log table (#23625)
- Implement `send_callback` method for `CeleryKubernetesExecutor` and `LocalKubernetesExecutor` (#23617)
- Fix `PythonVirtualenvOperator` `templated_fields` (#23559)
- Apply specific ID collation to `root_dag_id` too (#23536)
- Prevent `KubernetesJobWatcher` getting stuck on resource too old (#23521)
- Fix scheduler crash when expanding with mapped task that returned none (#23486)
- Fix broken dagrun links when many runs start at the same time (#23462)
- Fix: Exception when parsing log #20966 (#23301)
- Handle invalid date parsing in webserver views. (#23161)
- Pools with negative open slots should not block other pools (#23143)
- Move around overflow, position and padding (#23044)

- Change approach to finding bad rows to LEFT OUTER JOIN. (#23528)
- Only count bad refs when moved table exists (#23491)
- Visually distinguish task group summary (#23488)
- Remove color change for highly nested groups (#23482)
- Optimize 2.3.0 pre-upgrade check queries (#23458)
- Add backward compatibility for `core_sqlalchemy_conn_cmd` (#23441)
- Fix literal cross product expansion (#23434)
- Fix broken task instance link in xcom list (#23367)
- Fix connection test button (#23345)
- fix cli airflow dags show for mapped operator (#23339)
- Hide some task instance attributes (#23338)
- Don't show grid actions if server would reject with permission denied (#23332)
- Use `run_id` for `ti.mark_success_url` (#23330)
- Fix update user auth stats (#23314)
- Use `<Time />` in Mapped Instance table (#23313)
- Fix duplicated Kubernetes DeprecationWarnings (#23302)
- Store grid view selection in url params (#23290)
- Remove custom signal handling in Triggerer (#23274)
- Override pool for TaskInstance when pool is passed from cli. (#23258)
- Show warning if '/' is used in a DAG run ID (#23106)
- Use kubernetes queue in kubernetes hybrid executors (#23048)
- Add tags inside try block. (#21784)

Doc only changes

- Move `dag_processing.processor_timeouts` to counters section (#23393)
- Clarify that bundle extras should not be used for PyPi installs (#23697)
- Synchronize support for Postgres and K8S in docs (#23673)
- Replace DummyOperator references in docs (#23502)
- Add doc notes for keyword-only args for `expand()` and `partial()` (#23373)
- Document fix for broken elasticsearch logs with 2.3.0+ upgrade (#23821)

Misc/Internal

- Add typing for `airflow/configuration.py` (#23716)
- Disable Flower by default from docker-compose (#23685)
- Added postgres 14 to support versions(including breeze) (#23506)
- add K8S 1.24 support (#23637)
- Refactor code references from tree to grid (#23254)

3.16.37 Airflow 2.3.0 (2022-04-30)

For production docker image related changes, see the [Docker Image Changelog](#).

Significant Changes

Passing execution_date to XCom.set(), XCom.clear(), XCom.get_one(), and XCom.get_many() is deprecated (#19825)

Continuing the effort to bind TaskInstance to a DagRun, XCom entries are now also tied to a DagRun. Use the `run_id` argument to specify the DagRun instead.

Task log templates are now read from the metadata database instead of airflow.cfg (#20165)

Previously, a task's log is dynamically rendered from the `[core] log_filename_template` and `[elasticsearch] log_id_template` config values at runtime. This resulted in unfortunate characteristics, e.g. it is impractical to modify the config value after an Airflow instance is running for a while, since all existing task logs have been saved under the previous format and cannot be found with the new config value.

A new `log_template` table is introduced to solve this problem. This table is synchronized with the aforementioned config values every time Airflow starts, and a new field `log_template_id` is added to every DAG run to point to the format used by tasks (NULL indicates the first ever entry for compatibility).

Minimum kubernetes library version bumped from 3.0.0 to 21.7.0 (#20759)

Note

This is only about changing the `kubernetes` library, not the Kubernetes cluster. Airflow support for Kubernetes version is described in [Installation prerequisites](#).

No change in behavior is expected. This was necessary in order to take advantage of a [bugfix](#) concerning refreshing of Kubernetes API tokens with EKS, which enabled the removal of some [workaround code](#).

XCom now defined by run_id instead of execution_date (#20975)

As a continuation to the TaskInstance-DagRun relation change started in Airflow 2.2, the `execution_date` columns on XCom has been removed from the database, and replaced by an [association proxy](#) field at the ORM level. If you access Airflow's metadata database directly, you should rewrite the implementation to use the `run_id` column instead.

Note that Airflow's metadatabase definition on both the database and ORM levels are considered implementation detail without strict backward compatibility guarantees.

Non-JSON-serializable params deprecated (#21135).

It was previously possible to use dag or task param defaults that were not JSON-serializable.

For example this worked previously:

```
@dag.task(params={"a": [1, 2, 3], "b": pendulum.now()})
def datetime_param(value):
    print(value)

datetime_param("{{ params.a }} | {{ params.b }}")
```

Note the use of `set` and `datetime` types, which are not JSON-serializable. This behavior is problematic because to override these values in a dag run conf, you must use JSON, which could make these params non-overridable. Another problem is that the support for param validation assumes JSON. Use of non-JSON-serializable params will be removed in Airflow 3.0 and until then, use of them will produce a warning at parse time.

You must use `postgresql://` instead of `postgres://` in `sqlalchemy_conn` for SQLAlchemy 1.4.0+ (#21205)

When you use SQLAlchemy 1.4.0+, you need to use `postgresql://` as the scheme in the `sqlalchemy_conn`. In the previous versions of SQLAlchemy it was possible to use `postgres://`, but using it in SQLAlchemy 1.4.0+ results in:

```
>     raise exc.NoSuchModuleError(
        "Can't load plugin: %s:%s" % (self.group, name)
    )
E     sqlalchemy.exc.NoSuchModuleError: Can't load plugin: sqlalchemy.dialects:postgres
```

If you cannot change the scheme of your URL immediately, Airflow continues to work with SQLAlchemy 1.3 and you can downgrade SQLAlchemy, but we recommend updating the scheme. Details in the [SQLAlchemy Changelog](#).

`auth_backends` replaces `auth_backend` configuration setting (#21472)

Previously, only one backend was used to authorize use of the REST API. In 2.3 this was changed to support multiple backends, separated by comma. Each will be tried in turn until a successful response is returned.

`airflow.models.base.Operator` is removed (#21505)

Previously, there was an empty class `airflow.models.base.Operator` for “type hinting”. This class was never really useful for anything (everything it did could be done better with `airflow.models.baseoperator.BaseOperator`), and has been removed. If you are relying on the class’s existence, use `BaseOperator` (for concrete operators), `airflow.models.abstractoperator.AbstractOperator` (the base class of both `BaseOperator` and the AIP-42 `MappedOperator`), or `airflow.models.operator.Operator` (a union type `BaseOperator | MappedOperator` for type annotation).

Zip files in the DAGs folder can no longer have a .py extension (#21538)

It was previously possible to have any extension for zip files in the DAGs folder. Now `.py` files are going to be loaded as modules without checking whether it is a zip file, as it leads to less IO. If a `.py` file in the DAGs folder is a zip compressed file, parsing it will fail with an exception.

`auth_backends` includes `session` (#21640)

To allow the Airflow UI to use the API, the previous default authorization backend `airflow.api.auth.backend.deny_all` is changed to `airflow.api.auth.backend.session`, and this is automatically added to the list of API authorization backends if a non-default value is set.

Default templates for log filenames and elasticsearch log_id changed (#21734)

In order to support Dynamic Task Mapping the default templates for per-task instance logging has changed. If your config contains the old default values they will be upgraded-in-place.

If you are happy with the new config values you should *remove* the setting in `airflow.cfg` and let the default value be used. Old default values were:

- [core] log_filename_template: {{ ti.dag_id }}/{{ ti.task_id }}/{{ ts }}/{{ try_number }}.log
- [elasticsearch] log_id_template: {dag_id}-{task_id}-{execution_date}-{try_number}

[core] log_filename_template now uses “hive partition style” of dag_id=<id>/run_id=<id> by default, which may cause problems on some older FAT filesystems. If this affects you then you will have to change the log template.

If you have customized the templates you should ensure that they contain {{ ti.map_index }} if you want to use dynamically mapped tasks.

If after upgrading you find your task logs are no longer accessible, try adding a row in the log_template table with id=0 containing your previous log_id_template and log_filename_template. For example, if you used the defaults in 2.2.5:

```
INSERT INTO log_template (id, filename, elasticsearch_id, created_at) VALUES (0, '{{ ti.  
->dag_id }}/{{ ti.task_id }}/{{ ts }}/{{ try_number }}.log', '{dag_id}-{task_id}-  
->{execution_date}-{try_number}', NOW());
```

BaseOperatorLink's get_link method changed to take a ti_key keyword argument (#21798)

In v2.2 we “deprecated” passing an execution date to XCom.get methods, but there was no other option for operator links as they were only passed an execution_date.

Now in 2.3 as part of Dynamic Task Mapping (AIP-42) we will need to add map_index to the XCom row to support the “reduce” part of the API.

In order to support that cleanly we have changed the interface for BaseOperatorLink to take an TaskInstanceKey as the ti_key keyword argument (as execution_date + task is no longer unique for mapped operators).

The existing signature will be detected (by the absence of the ti_key argument) and continue to work.

ReadyToRescheduleDep now only runs when reschedule is True (#21815)

When a ReadyToRescheduleDep is run, it now checks whether the reschedule attribute on the operator, and always reports itself as *passed* unless it is set to *True*. If you use this dep class on your custom operator, you will need to add this attribute to the operator class. Built-in operator classes that use this dep class (including sensors and all subclasses) already have this attribute and are not affected.

The deps attribute on an operator class should be a class level attribute (#21815)

To support operator-mapping (AIP 42), the deps attribute on operator class must be a set at the class level. This means that if a custom operator implements this as an instance-level variable, it will not be able to be used for operator-mapping. This does not affect existing code, but we highly recommend you to restructure the operator’s dep logic in order to support the new feature.

Deprecation: Connection.extra must be JSON-encoded dict (#21816)

TLDR

From Airflow 3.0, the extra field in airflow connections must be a JSON-encoded Python dict.

What, why, and when?

Airflow's Connection is used for storing credentials. For storage of information that does not fit into user / password / host / schema / port, we have the `extra` string field. Its intention was always to provide for storage of arbitrary key-value pairs, like `no_host_key_check` in the SSH hook, or `keyfile_dict` in GCP.

But since the field is string, it's technically been permissible to store any string value. For example one could have stored the string value '`my-website.com`' and used this in the hook. But this is a very bad practice. One reason is intelligibility: when you look at the value for `extra`, you don't have any idea what its purpose is. Better would be to store `{"api_host": "my-website.com"}` which at least tells you *something* about the value. Another reason is extensibility: if you store the API host as a simple string value, what happens if you need to add more information, such as the API endpoint, or credentials? Then you would need to convert the string to a dict, and this would be a breaking change.

For these reason, starting in Airflow 3.0 we will require that the `Connection.extra` field store a JSON-encoded Python dict.

How will I be affected?

For users of providers that are included in the Airflow codebase, you should not have to make any changes because in the Airflow codebase we should not allow hooks to misuse the `Connection.extra` field in this way.

However, if you have any custom hooks that store something other than JSON dict, you will have to update it. If you do, you should see a warning any time that this connection is retrieved or instantiated (e.g. it should show up in task logs).

To see if you have any connections that will need to be updated, you can run this command:

```
airflow connections export - 2>&1 >/dev/null | grep 'non-JSON'
```

This will catch any warnings about connections that are storing something other than JSON-encoded Python dict in the `extra` field.

The `tree` default view setting has been renamed to `grid` (#22167)

If you set the `dag_default_view` config option or the `default_view` argument to `DAG()` to `tree` you will need to update your deployment. The old name will continue to work but will issue warnings.

Database configuration moved to new section (#22284)

The following configurations have been moved from `[core]` to the new `[database]` section. However when reading the new option, the old option will be checked to see if it exists. If it does a `DeprecationWarning` will be issued and the old option will be used instead.

- `sql_alchemy_conn`
- `sql_engine_encoding`
- `sql_engine_collation_for_ids`
- `sql_alchemy_pool_enabled`
- `sql_alchemy_pool_size`
- `sql_alchemy_max_overflow`
- `sql_alchemy_pool_recycle`
- `sql_alchemy_pool_pre_ping`
- `sql_alchemy_schema`

- `sqlalchemy_connect_args`
- `load_default_connections`
- `max_db_retries`

Remove requirement that custom connection UI fields be prefixed (#22607)

Hooks can define custom connection fields for their connection type by implementing method `get_connection_form_widgets`. These custom fields appear in the web UI as additional connection attributes, but internally they are stored in the connection `extra` dict field. For technical reasons, previously, when stored in the `extra` dict, the custom field's dict key had to take the form `extra__<conn type>__<field name>`. This had the consequence of making it more cumbersome to define connections outside of the UI, since the prefix `extra__<conn type>__` makes it tougher to read and work with. With #22607, we make it so that you can now define custom fields such that they can be read from and stored in `extra` without the prefix.

To enable this, update the dict returned by the `get_connection_form_widgets` method to remove the prefix from the keys. Internally, the providers manager will still use a prefix to ensure each custom field is globally unique, but the absence of a prefix in the returned widget dict will signal to the Web UI to read and store custom fields without the prefix. Note that this is only a change to the Web UI behavior; when updating your hook in this way, you must make sure that when your *hook* reads the `extra` field, it will also check for the prefixed value for backward compatibility.

The `webserver.X_FRAME_ENABLED` configuration works according to description now (#23222).

In Airflow 2.0.0 - 2.2.4 the `webserver.X_FRAME_ENABLED` parameter worked the opposite of its description, setting the value to “true” caused “X-Frame-Options” header to “DENY” (not allowing Airflow to be used in an iframe). When you set it to “false”, the header was not added, so Airflow could be embedded in an iframe. By default Airflow could not be embedded in an iframe.

In Airflow 2.2.5 there was a bug introduced that made it impossible to disable Airflow to work in iframe. No matter what the configuration was set, it was possible to embed Airflow in an iframe.

Airflow 2.3.0 restores the original meaning to the parameter. If you set it to “true” (default) Airflow can be embedded in an iframe (no header is added), but when you set it to “false” the header is added and Airflow cannot be embedded in an iframe.

New Features

- Add dynamic task mapping (AIP-42)
- New Grid View replaces Tree View (#18675)
- Templatized `requirements.txt` in Python Operators (#17349)
- Allow reuse of decorated tasks (#22941)
- Move the database configuration to a new section (#22284)
- Add `SmoothOperator` (#22813)
- Make operator’s `execution_timeout` configurable (#22389)
- Events Timetable (#22332)
- Support dag serialization with custom `ti_deps` rules (#22698)
- Support log download in task log view (#22804)
- support for continue backfill on failures (#22697)
- Add `dag-processor` cli command (#22305)
- Add possibility to create users in LDAP mode (#22619)

- Add `ignore_first_depends_on_past` for scheduled jobs (#22491)
- Update base sensor operator to support XCOM return value (#20656)
- Add an option for run id in the ui trigger screen (#21851)
- Enable JSON serialization for connections (#19857)
- Add REST API endpoint for bulk update of DAGs (#19758)
- Add queue button to click-on-DagRun interface. (#21555)
- Add `list-import-errors` to `airflow dags` command (#22084)
- Store callbacks in database if `standalone_dag_processor` config is True. (#21731)
- Add LocalKubernetesExecutor (#19729)
- Add `celery.task_timeout_error` metric (#21602)
- Airflow db downgrade cli command (#21596)
- Add `ALL_SKIPPED` trigger rule (#21662)
- Add db clean CLI command for purging old data (#20838)
- Add `celery_logging_level` (#21506)
- Support different timeout value for dag file parsing (#21501)
- Support generating SQL script for upgrades (#20962)
- Add option to compress Serialized dag data (#21332)
- Branch python operator decorator (#20860)
- Add Audit Log View to Dag View (#20733)
- Add missing StatsD metric for failing SLA Callback notification (#20924)
- Add `ShortCircuitOperator` configurability for respecting downstream trigger rules (#20044)
- Allow using Markup in page title in Webserver (#20888)
- Add Listener Plugin API that tracks TaskInstance state changes (#20443)
- Add context var hook to inject more env vars (#20361)
- Add a button to set all tasks to skipped (#20455)
- Cleanup pending pods (#20438)
- Add config to warn public deployment exposure in UI (#18557)
- Log filename template records (#20165)
- Added windows extensions (#16110)
- Showing approximate time until next `dag_run` in Airflow (#20273)
- Extend config window on UI (#20052)
- Add show dag dependencies feature to CLI (#19985)
- Add cli command for ‘`airflow dags reserialize`’ (#19471)
- Add missing description field to Pool schema(REST API) (#19841)
- Introduce DagRun action to change state to queued. (#19353)
- Add DAG run details page (#19705)

- Add role export/import to cli tools (#18916)
- Adding `dag_id_pattern` parameter to the `/dags` endpoint (#18924)

Improvements

- Show `schedule_interval/timetable` description in UI (#16931)
- Added column duration to DAG runs view (#19482)
- Enable use of custom conn extra fields without prefix (#22607)
- Initialize finished counter at zero (#23080)
- Improve logging of optional provider features messages (#23037)
- Meaningful error message in `resolve_template_files` (#23027)
- Update `ImportError` items instead of deleting and recreating them (#22928)
- Add option `--skip-init` to db reset command (#22989)
- Support importing connections from files with “`.yml`” extension (#22872)
- Support glob syntax in `.airflowignore` files (#21392) (#22051)
- Hide pagination when data is a single page (#22963)
- Support for sorting DAGs in the web UI (#22671)
- Speed up `has_access` decorator by ~200ms (#22858)
- Add `XComArg` to lazy-imported list of Airflow module (#22862)
- Add more fields to REST API `dags/dag_id/details` endpoint (#22756)
- Don’t show irrelevant/duplicated/“internal” Task attrs in UI (#22812)
- No need to load whole ti in `current_state` (#22764)
- Pickle dag exception string fix (#22760)
- Better verification of LocalExecutor’s parallelism option (#22711)
- log backfill exceptions to sentry (#22704)
- retry commit on MySQL deadlocks during backfill (#22696)
- Add more fields to REST API `get DAG(dags/dag_id)` endpoint (#22637)
- Use timetable to generate planned days for current year (#22055)
- Disable connection pool for celery worker (#22493)
- Make date picker label visible in trigger dag view (#22379)
- Expose `try_number` in airflow vars (#22297)
- Add generic connection type (#22310)
- Add a few more fields to the taskinstance finished log message (#22262)
- Pause auto-refresh if scheduler isn’t running (#22151)
- Show DagModel details. (#21868)
- Add `pip_install_options` to PythonVirtualenvOperator (#22158)
- Show import error for `airflow dags list` CLI command (#21991)
- Pause auto-refresh when page is hidden (#21904)

- Default args type check (#21809)
- Enhance magic methods on XComArg for UX (#21882)
- py files don't have to be checked `is_zipfiles` in `refresh_dag` (#21926)
- Fix TaskDecorator type hints (#21881)
- Add ‘Show record’ option for variables (#21342)
- Use DB where possible for quicker `airflow dag` subcommands (#21793)
- REST API: add rendered fields in task instance. (#21741)
- Change the default auth backend to session (#21640)
- Don't check if py DAG files are zipped during parsing (#21538)
- Switch XCom implementation to use `run_id` (#20975)
- Action log on Browse Views (#21569)
- Implement multiple API auth backends (#21472)
- Change logging level details of connection info in `get_connection()` (#21162)
- Support mssql in airflow db shell (#21511)
- Support config `worker_enable_remote_control` for celery (#21507)
- Log memory usage in `CgroupTaskRunner` (#21481)
- Modernize DAG-related URL routes and rename “tree” to “grid” (#20730)
- Move Zombie detection to `SchedulerJob` (#21181)
- Improve speed to run `airflow` by 6x (#21438)
- Add more SQL template fields renderers (#21237)
- Simplify fab has access lookup (#19294)
- Log context only for default method (#21244)
- Log trigger status only if at least one is running (#21191)
- Add optional features in providers. (#21074)
- Better `multiple_outputs` inferral for `@task.python` (#20800)
- Improve handling of string type and non-attribute `template_fields` (#21054)
- Remove un-needed deps/version requirements (#20979)
- Correctly specify overloads for TaskFlow API for type-hinting (#20933)
- Introduce `notification_sent` to SlaMiss view (#20923)
- Rewrite the task decorator as a composition (#20868)
- Add “Greater/Smaller than or Equal” to filters in the browse views (#20602) (#20798)
- Rewrite DAG run retrieval in task command (#20737)
- Speed up creation of DagRun for large DAGs (5k+ tasks) by 25-130% (#20722)
- Make native environment Airflow-flavored like sandbox (#20704)
- Better error when param value has unexpected type (#20648)
- Add filter by state in DagRun REST API (List Dag Runs) (#20485)

- Prevent exponential memory growth in Tasks with custom logging handler (#20541)
- Set default logger in logging Mixin (#20355)
- Reduce deprecation warnings from www (#20378)
- Add hour and minute to time format on x-axis of all charts using nvd3.lineChart (#20002)
- Add specific warning when Task asks for more slots than pool defined with (#20178)
- UI: Update duration column for better human readability (#20112)
- Use Viewer role as example public role (#19215)
- Properly implement DAG param dict copying (#20216)
- `ShortCircuitOperator` push XCom by returning python_callable result (#20071)
- Add clear logging to tasks killed due to a Dagrun timeout (#19950)
- Change log level for Zombie detection messages (#20204)
- Better confirmation prompts (#20183)
- Only execute TIs of running DagRuns (#20182)
- Check and run migration in commands if necessary (#18439)
- Log only when Zombies exists (#20118)
- Increase length of the email and username (#19932)
- Add more filtering options for TI's in the UI (#19910)
- Dynamically enable “Test Connection” button by connection type (#19792)
- Avoid littering postgres server logs with “could not obtain lock” with HA schedulers (#19842)
- Renamed `Connection.get_hook` parameter to make it the same as in `SqlSensor` and `SqlOperator`. (#19849)
- Add `hook_params` in `SqlSensor` using the latest changes from PR #18718. (#18431)
- Speed up webserver boot time by delaying provider initialization (#19709)
- Configurable logging of XCOM value in `PythonOperator` (#19378)
- Minimize production js files (#19658)
- Add `hook_params` in `BaseSqlOperator` (#18718)
- Add missing “end_date” to hash components (#19281)
- More friendly output of the airflow plugins command + add timetables (#19298)
- Add sensor default timeout config (#19119)
- Update `taskinstance` REST API schema to include `dag_run_id` field (#19105)
- Adding feature in bash operator to append the user defined env variable to system env variable (#18944)
- Duplicate Connection: Added logic to query if a connection id exists before creating one (#18161)

Bug Fixes

- Use inherited ‘trigger_tasks’ method (#23016)
- In DAG dependency detector, use class type instead of class name (#21706)
- Fix tasks being wrongly skipped by `schedule_after_task_execution` (#23181)
- Fix X-Frame enabled behaviour (#23222)

- Allow `extra` to be nullable in connection payload as per schema(REST API). (#23183)
- Fix `dag_id` extraction for dag level access checks in web ui (#23015)
- Fix timezone display for logs on UI (#23075)
- Include message in graph errors (#23021)
- Change trigger dropdown left position (#23013)
- Don't add planned tasks for legacy DAG runs (#23007)
- Add dangling rows check for `TaskInstance` references (#22924)
- Validate the input params in connection CLI command (#22688)
- Fix trigger event payload is not persisted in db (#22944)
- Drop “airflow moved” tables in command `db reset` (#22990)
- Add max width to task group tooltips (#22978)
- Add template support for `external_task_ids`. (#22809)
- Allow `DagParam` to hold falsy values (#22964)
- Fix regression in pool metrics (#22939)
- Priority order tasks even when using pools (#22483)
- Do not clear XCom when resuming from deferral (#22932)
- Handle invalid JSON metadata in `get_logs_with_metadata` endpoint. (#22898)
- Fix pre-upgrade check for rows dangling w.r.t. `dag_run` (#22850)
- Fixed backfill interference with scheduler (#22701)
- Support conf param override for backfill runs (#22837)
- Correctly interpolate pool name in `PoolSlotsAvailableDep` statuses (#22807)
- Fix `email_on_failure` with `render_template_as_native_obj` (#22770)
- Fix processor cleanup on `DagFileProcessorManager` (#22685)
- Prevent meta name clash for task instances (#22783)
- remove json parse for gantt chart (#22780)
- Check for missing dagrun should know version (#22752)
- Fixes `ScheduleInterval` spec (#22635)
- Fixing task status for non-running and non-committed tasks (#22410)
- Do not log the hook connection details even at DEBUG level (#22627)
- Stop crashing when empty logs are received from kubernetes client (#22566)
- Fix bugs about timezone change (#22525)
- Fix entire DAG stops when one task has `end_date` (#20920)
- Use logger to print message during task execution. (#22488)
- Make sure finalizers are not skipped during exception handling (#22475)
- update smart sensor docs and minor fix on `is_smart_sensor_compatible()` (#22386)
- Fix `run_id` k8s and elasticsearch compatibility with Airflow 2.1 (#22385)

- Allow to `except_skip` None on `BranchPythonOperator` (#20411)
- Fix incorrect datetime details (`DagRun` views) (#21357)
- Remove incorrect deprecation warning in secrets backend (#22326)
- Remove `RefreshConfiguration` workaround for K8s token refreshing (#20759)
- Masking extras in GET `/connections/<connection>` endpoint (#22227)
- Set `queued_dttm` when submitting task to directly to executor (#22259)
- Addressed some issues in the tutorial mentioned in discussion #22233 (#22236)
- Change default python executable to `python3` for docker decorator (#21973)
- Don't validate that Params are JSON when NOTSET (#22000)
- Add per-DAG delete permissions (#21938)
- Fix handling some None parameters in kubernetes 23 libs. (#21905)
- Fix handling of empty (None) tags in `bulk_write_to_db` (#21757)
- Fix DAG date range bug (#20507)
- Removed `request.referrer` from `views.py` (#21751)
- Make `DbApiHook` use `get_uri` from `Connection` (#21764)
- Fix some migrations (#21670)
- [de]serialize resources on task correctly (#21445)
- Add params `dag_id`, `task_id` etc to `XCom.serialize_value` (#19505)
- Update test connection functionality to use custom form fields (#21330)
- fix all “high” npm vulnerabilities (#21526)
- Fix bug incorrectly removing action from role, rather than permission. (#21483)
- Fix relationship join bug in FAB/SecurityManager with SQLA 1.4 (#21296)
- Use Identity instead of Sequence in SQLAlchemy 1.4 for MSSQL (#21238)
- Ensure `on_task_instance_running` listener can get at task (#21157)
- Return to the same place when triggering a DAG (#20955)
- Fix task ID deduplication in `@task_group` (#20870)
- Add downgrade to some FAB migrations (#20874)
- Only validate Params when DAG is triggered (#20802)
- Fix `airflow trigger cli` (#20781)
- Fix task instances iteration in a pool to prevent blocking (#20816)
- Allow depending to a `@task_group` as a whole (#20671)
- Use original task's `start_date` if a task continues after deferral (#20062)
- Disabled edit button in task instances list view page (#20659)
- Fix a package name import error (#20519) (#20519)
- Remove `execution_date` label when get cleanup pods list (#20417)
- Remove unneeded FAB REST API endpoints (#20487)

- Fix parsing of Cloudwatch log group arn containing slashes (#14667) (#19700)
- Sanity check for MySQL's TIMESTAMP column (#19821)
- Allow using default celery command group with executors subclassed from Celery-based executors. (#18189)
- Move `class_permission_name` to mixin so it applies to all classes (#18749)
- Adjust `trimmed_pod_id` and replace ‘.’ with ‘-’ (#19036)
- Pass `custom_headers` to `send_email` and `send_email_smtp` (#19009)
- Ensure `catchup=False` is used in example dags (#19396)
- Edit permalinks in OpenApi description file (#19244)
- Navigate directly to DAG when selecting from search typeahead list (#18991)
- [Minor] Fix padding on home page (#19025)

Doc only changes

- Update doc for DAG file processing (#23209)
- Replace changelog/updating with release notes and `towncrier` now (#22003)
- Fix wrong reference in `tracking-user-activity.rst` (#22745)
- Remove references to `rbac = True` from docs (#22725)
- Doc: Update description for executor-bound dependencies (#22601)
- Update `check-health.rst` (#22372)
- Stronger language about Docker Compose customizability (#22304)
- Update `logging-tasks.rst` (#22116)
- Add example config of `sqlalchemy_connect_args` (#22045)
- Update `best-practices.rst` (#22053)
- Add information on DAG pausing/deactivation/deletion (#22025)
- Add brief examples of integration test dags you might want (#22009)
- Run inclusive language check on CHANGELOG (#21980)
- Add detailed email docs for Sendgrid (#21958)
- Add docs for `db upgrade` / `db downgrade` (#21879)
- Update `modules_management.rst` (#21889)
- Fix UPDATING section on SQLAlchemy 1.4 scheme changes (#21887)
- Update TaskFlow tutorial doc to show how to pass “operator-level” args. (#21446)
- Fix doc - replace decreasing by increasing (#21805)
- Add another way to dynamically generate DAGs to docs (#21297)
- Add extra information about time synchronization needed (#21685)
- Update `debug.rst` docs (#21246)
- Replaces the usage of `postgres://` with `postgresql://` (#21205)
- Fix task execution process in `CeleryExecutor` docs (#20783)

Misc/Internal

- Bring back deprecated security manager functions (#23243)
- Replace usage of `DummyOperator` with `EmptyOperator` (#22974)
- Deprecate `DummyOperator` in favor of `EmptyOperator` (#22832)
- Remove unnecessary python 3.6 conditionals (#20549)
- Bump `moment` from 2.29.1 to 2.29.2 in /airflow/www (#22873)
- Bump `prismjs` from 1.26.0 to 1.27.0 in /airflow/www (#22823)
- Bump `nanoid` from 3.1.23 to 3.3.2 in /airflow/www (#22803)
- Bump `minimist` from 1.2.5 to 1.2.6 in /airflow/www (#22798)
- Remove dag parsing from db init command (#22531)
- Update our approach for executor-bound dependencies (#22573)
- Use `Airflow.Base.metadata` in FAB models (#22353)
- Limit docutils to make our documentation pretty again (#22420)
- Add Python 3.10 support (#22050)
- [FEATURE] add 1.22 1.23 K8S support (#21902)
- Remove pandas upper limit now that SQLA is 1.4+ (#22162)
- Patch `sqlalchemy_conn` if old postgres scheme used (#22333)
- Protect against accidental misuse of `XCom.get_value()` (#22244)
- Order filenames for migrations (#22168)
- Don't try to auto generate migrations for Celery tables (#22120)
- Require SQLAlchemy 1.4 (#22114)
- bump sphinx-jinja (#22101)
- Add compat shim for SQLAlchemy to avoid warnings (#21959)
- Rename `xcom.dagrun_id` to `xcom.dag_run_id` (#21806)
- Deprecate non-JSON `conn.extra` (#21816)
- Bump upper bound version of `jsonschema` to 5.0 (#21712)
- Deprecate helper utility `days_ago` (#21653)
- Remove `:type` lines now `sphinx-autoapi` supports type hints (#20951)
- Silence deprecation warning in tests (#20900)
- Use `DagRun.run_id` instead of `execution_date` when updating state of TIs (UI & REST API) (#18724)
- Add Context stub to Airflow packages (#20817)
- Update Kubernetes library version (#18797)
- Rename `PodLauncher` to `PodManager` (#20576)
- Removes Python 3.6 support (#20467)
- Add deprecation warning for non-json-serializable params (#20174)
- Rename `TaskMixin` to `DependencyMixin` (#20297)

- Deprecate passing execution_date to XCom methods (#19825)
- Remove get_readable_dags and get_editable_dags, and get_accessible_dags. (#19961)
- Remove postgres 9.6 support (#19987)
- Removed hardcoded connection types. Check if hook is instance of DbApiHook. (#19639)
- add kubernetes 1.21 support (#19557)
- Add FAB base class and set import_name explicitly. (#19667)
- Removes unused state transitions to handle auto-changing view permissions. (#19153)
- Chore: Use enum for __var and __type members (#19303)
- Use fab models (#19121)
- Consolidate method names between Airflow Security Manager and FAB default (#18726)
- Remove distutils usages for Python 3.10 (#19064)
- Removing redundant max_tis_per_query initialisation on SchedulerJob (#19020)
- Remove deprecated usage of init_role() from API (#18820)
- Remove duplicate code on dbapi hook (#18821)

3.16.38 Airflow 2.2.5, (2022-04-04)

Significant Changes

No significant changes.

Bug Fixes

- Check and disallow a relative path for sqlite (#22530)
- Fixed dask executor and tests (#22027)
- Fix broken links to celery documentation (#22364)
- Fix incorrect data provided to tries & landing times charts (#21928)
- Fix assignment of unassigned triggers (#21770)
- Fix triggerer --capacity parameter (#21753)
- Fix graph auto-refresh on page load (#21736)
- Fix filesystem sensor for directories (#21729)
- Fix stray order_by(TaskInstance.execution_date) (#21705)
- Correctly handle multiple '=' in LocalFileSystem secrets. (#21694)
- Log exception in local executor (#21667)
- Disable default_pool delete on web ui (#21658)
- Extends typing-extensions to be installed with python 3.8+ #21566 (#21567)
- Dispose unused connection pool (#21565)
- Fix logging JDBC SQL error when task fails (#21540)
- Filter out default configs when overrides exist. (#21539)
- Fix Resources __eq__ check (#21442)

- Fix `max_active_runs=1` not scheduling runs when `min_file_process_interval` is high (#21413)
- Reduce DB load incurred by Stale DAG deactivation (#21399)
- Fix race condition between triggerer and scheduler (#21316)
- Fix trigger dag redirect from task instance log view (#21239)
- Log traceback in trigger exceptions (#21213)
- A trigger might use a connection; make sure we mask passwords (#21207)
- Update `ExternalTaskSensorLink` to handle templated `external_dag_id` (#21192)
- Ensure `clear_task_instances` sets valid run state (#21116)
- Fix: Update custom connection field processing (#20883)
- Truncate stack trace to DAG user code for exceptions raised during execution (#20731)
- Fix duplicate trigger creation race condition (#20699)
- Fix Tasks getting stuck in scheduled state (#19747)
- Fix: Do not render undefined graph edges (#19684)
- Set `X-Frame-Options` header to DENY only if `X_FRAME_ENABLED` is set to true. (#19491)

Doc only changes

- adding `on_execute_callback` to callbacks docs (#22362)
- Add documentation on specifying a DB schema. (#22347)
- Fix postgres part of pipeline example of tutorial (#21586)
- Extend documentation for states of DAGs & tasks and update trigger rules docs (#21382)
- DB upgrade is required when updating Airflow (#22061)
- Remove misleading MSSQL information from the docs (#21998)

Misc

- Add the new Airflow Trove Classifier to setup.cfg (#22241)
- Rename `to_delete` to `to_cancel` in TriggerRunner (#20658)
- Update Flask-AppBuilder to 3.4.5 (#22596)

3.16.39 Airflow 2.2.4, (2022-02-22)

Significant Changes

Smart sensors deprecated

Smart sensors, an “early access” feature added in Airflow 2, are now deprecated and will be removed in Airflow 2.4.0. They have been superseded by Deferrable Operators, added in Airflow 2.2.0.

See [Migrating to Deferrable Operators](#) for details on how to migrate.

Bug Fixes

- Adding missing login provider related methods from Flask-Appbuilder (#21294)
- Fix slow DAG deletion due to missing `dag_id` index for job table (#20282)
- Add a session backend to store session data in the database (#21478)
- Show task status only for running dags or only for the last finished dag (#21352)
- Use compat data interval shim in log handlers (#21289)
- Fix mismatch in generated `run_id` and logical date of DAG run (#18707)
- Fix TriggerDagRunOperator extra link (#19410)
- Add possibility to create user in the Remote User mode (#19963)
- Avoid deadlock when rescheduling task (#21362)
- Fix the incorrect scheduling time for the first run of dag (#21011)
- Fix Scheduler crash when executing task instances of missing DAG (#20349)
- Deferred tasks does not cancel when DAG is marked fail (#20649)
- Removed duplicated `dag_run` join in `Dag.get_task_instances()` (#20591)
- Avoid unintentional data loss when deleting DAGs (#20758)
- Fix session usage in `/rendered-k8s` view (#21006)
- Fix `airflow dags backfill --reset-dagrains` errors when run twice (#21062)
- Do not set `TaskInstance.max_tries` in `refresh_from_task` (#21018)
- Don't require `dag_id` in body in dagrun REST API endpoint (#21024)
- Add Roles from Azure OAUTH Response in internal Security Manager (#20707)
- Allow Viewing DagRuns and TIs if a user has DAG “read” perms (#20663)
- Fix running `airflow dags test <dag_id> <execution_dt>` results in error when run twice (#21031)
- Switch to non-vendored latest connexion library (#20910)
- Bump flask-appbuilder to `>=3.3.4` (#20628)
- upgrade celery to `5.2.3` (#19703)
- Bump croniter from `<1.1` to `<1.2` (#20489)
- Avoid calling `DAG.following_schedule()` for `TaskInstance.get_template_context()` (#20486)
- Fix(standalone): Remove hardcoded Webserver port (#20429)
- Remove unnecessary logging in experimental API (#20356)
- Un-ignore DeprecationWarning (#20322)
- Deepcopying Kubernetes Secrets attributes causing issues (#20318)
- Fix(dag-dependencies): fix arrow styling (#20303)
- Adds retry on taskinstance retrieval lock (#20030)
- Correctly send timing metrics when using dogstatsd (fix `schedule_delay` metric) (#19973)
- Enhance `multiple_outputs` inference of dict typing (#19608)
- Fixing Amazon SES email backend (#18042)

- Pin MarkupSafe until we are able to upgrade Flask/Jinja (#21664)

Doc only changes

- Added explaining concept of logical date in DAG run docs (#21433)
- Add note about Variable precedence with env vars (#21568)
- Update error docs to include before_send option (#21275)
- Augment xcom docs (#20755)
- Add documentation and release policy on “latest” constraints (#21093)
- Add a link to the DAG model in the Python API reference (#21060)
- Added an enum param example (#20841)
- Compare taskgroup and subdag (#20700)
- Add note about reserved `params` keyword (#20640)
- Improve documentation on `Params` (#20567)
- Fix typo in MySQL Database creation code (Set up DB docs) (#20102)
- Add requirements.txt description (#20048)
- Clean up `default_args` usage in docs (#19803)
- Add docker-compose explanation to conn localhost (#19076)
- Update CSV ingest code for tutorial (#18960)
- Adds Pendulum 1.x -> 2.x upgrade documentation (#18955)
- Clean up dynamic `start_date` values from docs (#19607)
- Docs for multiple pool slots (#20257)
- Update upgrading.rst with detailed code example of how to resolve post-upgrade warning (#19993)

Misc

- Deprecate some functions in the experimental API (#19931)
- Deprecate smart sensors (#20151)

3.16.40 Airflow 2.2.3, (2021-12-21)

Significant Changes

No significant changes.

Bug Fixes

- Lazy Jinja2 context (#20217)
- Exclude `snowflake-sqlalchemy` v1.2.5 (#20245)
- Move away from legacy `importlib.resources` API (#19091)
- Move `setgid` as the first command executed in forked task runner (#20040)
- Fix race condition when starting `DagProcessorAgent` (#19935)
- Limit `httpx` to <0.20.0 (#20218)

- Log provider import errors as debug warnings (#20172)
- Bump minimum required alembic version (#20153)
- Fix log link in gantt view (#20121)
- fixing #19028 by moving chown to use sudo (#20114)
- Lift off upper bound for MarkupSafe (#20113)
- Fix infinite recursion on redact log (#20039)
- Fix db downgrades (#19994)
- Context class handles deprecation (#19886)
- Fix possible reference to undeclared variable (#19933)
- Validate DagRun state is valid on assignment (#19898)
- Workaround occasional deadlocks with MSSQL (#19856)
- Enable task run setting to be able reinitialize (#19845)
- Fix log endpoint for same task (#19672)
- Cast macro datetime string inputs explicitly (#19592)
- Do not crash with stacktrace when task instance is missing (#19478)
- Fix log timezone in task log view (#19342) (#19401)
- Fix: Add taskgroup tooltip to graph view (#19083)
- Rename execution date in forms and tables (#19063)
- Simplify “invalid TI state” message (#19029)
- Handle case of nonexistent file when preparing file path queue (#18998)
- Do not create dagruns for DAGs with import errors (#19367)
- Fix field relabeling when switching between conn types (#19411)
- KubernetesExecutor should default to template image if used (#19484)
- Fix task instance api cannot list task instances with None state (#19487)
- Fix IntegrityError in DagFileProcessor.manage_slas (#19553)
- Declare data interval fields as serializable (#19616)
- Relax timetable class validation (#19878)
- Fix labels used to find queued KubernetesExecutor pods (#19904)
- Fix moved data migration check for MySQL when replication is used (#19999)

Doc only changes

- Warn without tracebacks when example_dags are missing deps (#20295)
- Deferrable operators doc clarification (#20150)
- Ensure the example DAGs are all working (#19355)
- Updating core example DAGs to use TaskFlow API where applicable (#18562)
- Add xcom clearing behaviour on task retries (#19968)
- Add a short chapter focusing on adapting secret format for connections (#19859)

- Add information about supported OS-es for Apache Airflow (#19855)
- Update docs to reflect that changes to the `base_log_folder` require updating other configs (#19793)
- Disclaimer in `KubernetesExecutor` pod template docs (#19686)
- Add upgrade note on `execution_date -> run_id` (#19593)
- Expanding `.output` operator property information in TaskFlow tutorial doc (#19214)
- Add example SLA DAG (#19563)
- Add a proper example to patch DAG (#19465)
- Add DAG file processing description to Scheduler Concepts (#18954)
- Updating explicit arg example in TaskFlow API tutorial doc (#18907)
- Adds back documentation about context usage in `Python/@task` (#18868)
- Add release date for when an endpoint/field is added in the REST API (#19203)
- Better `pod_template_file` examples (#19691)
- Add description on how you can customize image entrypoint (#18915)
- Dags-in-image pod template example should not have dag mounts (#19337)

3.16.41 Airflow 2.2.2 (2021-11-15)

Significant Changes

No significant changes.

Bug Fixes

- Fix bug when checking for existence of a Variable (#19395)
- Fix Serialization when `relativedelta` is passed as `schedule_interval` (#19418)
- Fix moving of dangling TaskInstance rows for SQL Server (#19425)
- Fix task instance modal in gantt view (#19258)
- Fix serialization of `Params` with set data type (#19267)
- Check if job object is `None` before calling `.is_alive()` (#19380)
- Task should fail immediately when pod is unprocessable (#19359)
- Fix downgrade for a DB Migration (#19390)
- Only mark SchedulerJobs as failed, not any jobs (#19375)
- Fix message on “Mark as” confirmation page (#19363)
- Bugfix: Check next run exists before reading data interval (#19307)
- Fix MySQL db migration with default encoding/collation (#19268)
- Fix hidden tooltip position (#19261)
- `sqlite_default` Connection has been hard-coded to `/tmp`, use `gettempdir` instead (#19255)
- Fix Toggle Wrap on DAG code page (#19211)
- Clarify “dag not found” error message in CLI (#19338)
- Add Note to SLA regarding `schedule_interval` (#19173)

- Use `execution_date` to check for existing `DagRun` for `TriggerDagRunOperator` (#18968)
- Add explicit session parameter in `PoolSlotsAvailableDep` (#18875)
- FAB still requires `WTForms<3.0` (#19466)
- Fix missing dagruns when `catchup=True` (#19528)

Doc only changes

- Add missing parameter documentation for “timetable” (#19282)
- Improve Kubernetes Executor docs (#19339)
- Update image tag used in docker docs

3.16.42 Airflow 2.2.1 (2021-10-29)

Significant Changes

Param's default value for default removed

`Param`, introduced in Airflow 2.2.0, accidentally set the default value to `None`. This default has been removed. If you want `None` as your default, explicitly set it as such. For example:

```
Param(None, type=["null", "string"])
```

Now if you resolve a `Param` without a default and don't pass a value, you will get an `TypeError`. For Example:

```
Param().resolve() # raises TypeError
```

max_queued_runs_per_dag configuration has been removed

The `max_queued_runs_per_dag` configuration option in `[core]` section has been removed. Previously, this controlled the number of queued dagrun the scheduler can create in a dag. Now, the maximum number is controlled internally by the DAG's `max_active_runs`

Bug Fixes

- Fix Unexpected commit error in `SchedulerJob` (#19213)
- Add `DagRun.logical_date` as a property (#19198)
- Clear `ti.next_method` and `ti.next_kwargs` on task finish (#19183)
- Faster PostgreSQL db migration to Airflow 2.2 (#19166)
- Remove incorrect type comment in `Swagger2Specification._set_defaults` classmethod (#19065)
- Add `TriggererJob` to jobs check command (#19179, #19185)
- Hide tooltip when next run is `None` (#19112)
- Create TI context with data interval compat layer (#19148)
- Fix queued dag runs changes `catchup=False` behaviour (#19130, #19145)
- add detailed information to logging when a dag or a task finishes. (#19097)
- Warn about unsupported Python 3.10 (#19060)
- Fix catchup by limiting queued dagrun creation using `max_active_runs` (#18897)
- Prevent scheduler crash when serialized dag is missing (#19113)

- Don't install SQLAlchemy/Pendulum adapters for other DBs (#18745)
- Workaround libstdcpp TLS error (#19010)
- Change `ds`, `ts`, etc. back to use logical date (#19088)
- Ensure task state doesn't change when marked as failed/success/skipped (#19095)
- Relax packaging requirement (#19087)
- Rename trigger page label to Logical Date (#19061)
- Allow Param to support a default value of None (#19034)
- Upgrade old DAG/task param format when deserializing from the DB (#18986)
- Don't bake ENV and `_cmd` into tmp config for non-sudo (#18772)
- CLI: Fail `backfill` command before loading DAGs if missing args (#18994)
- BugFix: Null execution date on insert to `task_fail` violating NOT NULL (#18979)
- Try to move “dangling” rows in db upgrade (#18953)
- Row lock TI query in `SchedulerJob._process_executor_events` (#18975)
- Sentry before send fallback (#18980)
- Fix `XCom.delete` error in Airflow 2.2.0 (#18956)
- Check python version before starting triggerer (#18926)

Doc only changes

- Update access control documentation for TaskInstances and DagRuns (#18644)
- Add information about keepalives for managed Postgres (#18850)
- Doc: Add Callbacks Section to Logging & Monitoring (#18842)
- Group PATCH DAGRun together with other DAGRun endpoints (#18885)

3.16.43 Airflow 2.2.0 (2021-10-11)

Significant Changes

Note: Upgrading the database to 2.2.0 or later can take some time to complete, particularly if you have a large `task_instance` table.

`worker_log_server_port` configuration has been moved to the logging section.

The `worker_log_server_port` configuration option has been moved from `[celery]` section to `[logging]` section to allow for reuse between different executors.

pandas is now an optional dependency

Previously `pandas` was a core requirement so when you run `pip install apache-airflow` it looked for `pandas` library and installed it if it does not exist.

If you want to install `pandas` compatible with Airflow, you can use `[pandas]` extra while installing Airflow, example for Python 3.8 and Airflow 2.1.2:

```
pip install -U "apache-airflow[pandas]==2.1.2" \
--constraint https://raw.githubusercontent.com/apache/airflow/constraints-2.1.2/
constraints-3.8.txt"
```

none_failed_or_skipped trigger rule has been deprecated

`TriggerRule.NONE_FAILED_OR_SKIPPED` is replaced by `TriggerRule.NONE_FAILED_MIN_ONE_SUCCESS`. This is only name change, no functionality changes made. This change is backward compatible however `TriggerRule.NONE_FAILED_OR_SKIPPED` will be removed in next major release.

Dummy trigger rule has been deprecated

`TriggerRule.DUMMY` is replaced by `TriggerRule.ALWAYS`. This is only name change, no functionality changes made. This change is backward compatible however `TriggerRule.DUMMY` will be removed in next major release.

DAG concurrency settings have been renamed

[core] `dag_concurrency` setting in `airflow.cfg` has been renamed to [core] `max_active_tasks_per_dag` for better understanding.

It is the maximum number of task instances allowed to run concurrently in each DAG. To calculate the number of tasks that is running concurrently for a DAG, add up the number of running tasks for all DAG runs of the DAG.

This is configurable at the DAG level with `max_active_tasks` and a default can be set in `airflow.cfg` as [core] `max_active_tasks_per_dag`.

Before:

```
[core]
dag_concurrency = 16
```

Now:

```
[core]
max_active_tasks_per_dag = 16
```

Similarly, `DAG.concurrency` has been renamed to `DAG.max_active_tasks`.

Before:

```
dag = DAG(
    dag_id="example_dag",
    start_date=datetime(2021, 1, 1),
    catchup=False,
    concurrency=3,
)
```

Now:

```
dag = DAG(
    dag_id="example_dag",
    start_date=datetime(2021, 1, 1),
    catchup=False,
    max_active_tasks=3,
)
```

If you are using DAGs Details API endpoint, use `max_active_tasks` instead of `concurrency`.

Task concurrency parameter has been renamed

`BaseOperator.task_concurrency` has been deprecated and renamed to `max_active_tis_per_dag` for better understanding.

This parameter controls the number of concurrent running task instances across `dag_runs` per task.

Before:

```
with DAG(dag_id="task_concurrency_example"):
    BashOperator(task_id="t1", task_concurrency=2, bash_command="echo Hi")
```

After:

```
with DAG(dag_id="task_concurrency_example"):
    BashOperator(task_id="t1", max_active_tis_per_dag=2, bash_command="echo Hi")
```

processor_poll_interval config have been renamed to scheduler_idle_sleep_time

[scheduler] `processor_poll_interval` setting in `airflow.cfg` has been renamed to [scheduler] `scheduler_idle_sleep_time` for better understanding.

It controls the ‘time to sleep’ at the end of the Scheduler loop if nothing was scheduled inside `SchedulerJob`.

Before:

```
[scheduler]
processor_poll_interval = 16
```

Now:

```
[scheduler]
scheduler_idle_sleep_time = 16
```

Marking success/failed automatically clears failed downstream tasks

When marking a task success/failed in Graph View, its downstream tasks that are in failed/upstream_failed state are automatically cleared.

[core] store_dag_code has been removed

While DAG Serialization is a strict requirement since Airflow 2, we allowed users to control where the Webserver looked for when showing the **Code View**.

If [core] `store_dag_code` was set to `True`, the Scheduler stored the code in the DAG file in the DB (in `dag_code` table) as a plain string, and the webserver just read it from the same table. If the value was set to `False`, the webserver read it from the DAG file.

While this setting made sense for Airflow < 2, it caused some confusion to some users where they thought this setting controlled DAG Serialization.

From Airflow 2.2, Airflow will only look for DB when a user clicks on **Code View** for a DAG.

Clearing a running task sets its state to RESTARTING

Previously, clearing a running task sets its state to SHUTDOWN. The task gets killed and goes into FAILED state. After #16681, clearing a running task sets its state to RESTARTING. The task is eligible for retry without going into FAILED state.

Remove TaskInstance.log_filepath attribute

This method returned incorrect values for a long time, because it did not take into account the different logger configuration and task retries. We have also started supporting more advanced tools that don't use files, so it is impossible to determine the correct file path in every case e.g. Stackdriver doesn't use files but identifies logs based on labels. For this reason, we decided to delete this attribute.

If you need to read logs, you can use `airflow.utils.log.log_reader.TaskLogReader` class, which does not have the above restrictions.

If a sensor times out, it will not retry

Previously, a sensor is retried when it times out until the number of retries are exhausted. So the effective timeout of a sensor is `timeout * (retries + 1)`. This behaviour is now changed. A sensor will immediately fail without retrying if `timeout` is reached. If it's desirable to let the sensor continue running for longer time, set a larger `timeout` instead.

Default Task Pools Slots can be set using [core] default_pool_task_slot_count

By default tasks are running in `default_pool`. `default_pool` is initialized with 128 slots and user can change the number of slots through UI/CLI/API for an existing deployment.

For new deployments, you can use `default_pool_task_slot_count` setting in `[core]` section. This setting would not have any effect in an existing deployment where the `default_pool` already exists.

Previously this was controlled by `non_pooled_task_slot_count` in `[core]` section, which was not documented.

Webserver DAG refresh buttons removed

Now that the DAG parser syncs DAG permissions there is no longer a need for manually refreshing DAGs. As such, the buttons to refresh a DAG have been removed from the UI.

In addition, the `/refresh` and `/refresh_all` webserver endpoints have also been removed.

TaskInstances now require a DagRun

Under normal operation every `TaskInstance` row in the database would have `DagRun` row too, but it was possible to manually delete the `DagRun` and Airflow would still schedule the `TaskInstances`.

In Airflow 2.2 we have changed this and now there is a database-level foreign key constraint ensuring that every `TaskInstance` has a `DagRun` row.

Before updating to this 2.2 release you will have to manually resolve any inconsistencies (add back `DagRun` rows, or delete `TaskInstances`) if you have any "dangling" `TaskInstance` rows.

As part of this change the `clean_tis_without_dagrun_interval` config option under `[scheduler]` section has been removed and has no effect.

TaskInstance and TaskReschedule now define run_id instead of execution_date

As a part of the TaskInstance-DagRun relation change, the `execution_date` columns on TaskInstance and TaskReschedule have been removed from the database, and replaced by `association proxy` fields at the ORM level. If you access Airflow's metadatabase directly, you should rewrite the implementation to use the `run_id` columns instead.

Note that Airflow's metadatabase definition on both the database and ORM levels are considered implementation detail without strict backward compatibility guarantees.

DaskExecutor - Dask Worker Resources and queues

If dask workers are not started with complementary resources to match the specified queues, it will now result in an `AirflowException`, whereas before it would have just ignored the queue argument.

Logical date of a DAG run triggered from the web UI now have its sub-second component set to zero

Due to a change in how the logical date (`execution_date`) is generated for a manual DAG run, a manual DAG run's logical date may not match its time-of-trigger, but have its sub-second part zero-ed out. For example, a DAG run triggered on `2021-10-11T12:34:56.78901` would have its logical date set to `2021-10-11T12:34:56.00000`.

This may affect some logic that expects on this quirk to detect whether a run is triggered manually or not. Note that `dag_run.run_type` is a more authoritative value for this purpose. Also, if you need this distinction between automated and manually-triggered run for “next execution date” calculation, please also consider using the new data interval variables instead, which provide a more consistent behavior between the two run types.

New Features

- AIP-39: Add (customizable) Timetable class to Airflow for richer scheduling behaviour (#15397, #16030, #16352, #17030, #17122, #17414, #17552, #17755, #17989, #18084, #18088, #18244, #18266, #18420, #18434, #18421, #18475, #18499, #18573, #18522, #18729, #18706, #18742, #18786, #18804)
- AIP-40: Add Deferrable “Async” Tasks (#15389, #17564, #17565, #17601, #17745, #17747, #17748, #17875, #17876, #18129, #18210, #18214, #18552, #18728, #18414)
- Add a Docker Taskflow decorator (#15330, #18739)
- Add Airflow Standalone command (#15826)
- Display alert messages on dashboard from local settings (#18284)
- Advanced Params using json-schema (#17100)
- Ability to test connections from UI or API (#15795, #18750)
- Add Next Run to UI (#17732)
- Add default weight rule configuration option (#18627)
- Add a calendar field to choose the execution date of the DAG when triggering it (#16141)
- Allow setting specific cwd for BashOperator (#17751)
- Show import errors in DAG views (#17818)
- Add pre/post execution hooks [Experimental] (#17576)
- Added table to view providers in Airflow ui under admin tab (#15385)
- Adds secrets backend/logging/auth information to provider yaml (#17625)
- Add date format filters to Jinja environment (#17451)

- Introduce RESTARTING state (#16681)
- Webserver: Unpause DAG on manual trigger (#16569)
- API endpoint to create new user (#16609)
- Add `insert_args` for support transfer replace (#15825)
- Add recursive flag to glob in filesystem sensor (#16894)
- Add conn to jinja template context (#16686)
- Add `default_args` for TaskGroup (#16557)
- Allow adding duplicate connections from UI (#15574)
- Allow specifying multiple URLs via the CORS config option (#17941)
- Implement API endpoint for DAG deletion (#17980)
- Add DAG run endpoint for marking a dagrun success or failed (#17839)
- Add support for `kinit` options `[-f|-F]` and `[-a|-A]` (#17816)
- Queue support for DaskExecutor using Dask Worker Resources (#16829, #18720)
- Make auto refresh interval configurable (#18107)

Improvements

- Small improvements for Airflow UI (#18715, #18795)
- Rename `processor_poll_interval` to `scheduler_idle_sleep_time` (#18704)
- Check the allowed values for the logging level (#18651)
- Fix error on triggering a dag that doesn't exist using `dagrun_conf` (#18655)
- Add muldelete action to `TaskInstanceModelView` (#18438)
- Avoid importing DAGs during clean DB installation (#18450)
- Require `can_edit` on DAG privileges to modify TaskInstances and DagRuns (#16634)
- Make Kubernetes job description fit on one log line (#18377)
- Always draw borders if task instance state is null or undefined (#18033)
- Inclusive Language (#18349)
- Improved log handling for zombie tasks (#18277)
- Adding `Variable.update` method and improving detection of variable key collisions (#18159)
- Add note about params on trigger DAG page (#18166)
- Change `TaskInstance` and `TaskReschedule` PK from `execution_date` to `run_id` (#17719)
- Adding TaskGroup support in `BaseOperator.chain()` (#17456)
- Allow filtering DAGS by tags in the REST API (#18090)
- Optimize imports of Providers Manager (#18052)
- Adds capability of Warnings for incompatible community providers (#18020)
- Serialize the `template_ext` attribute to show it in UI (#17985)
- Add `robots.txt` and `X-Robots-Tag` header (#17946)
- Refactor `BranchDayOfWeekOperator`, `DayOfWeekSensor` (#17940)

- Update error message to guide the user into self-help mostly (#17929)
- Update to Celery 5 (#17397)
- Add links to provider's documentation (#17736)
- Remove Marshmallow schema warnings (#17753)
- Rename `none_failed_or_skipped` by `none_failed_min_one_success` trigger rule (#17683)
- Remove `[core] store_dag_code` & use DB to get Dag Code (#16342)
- Rename `task_concurrency` to `max_active_tis_per_dag` (#17708)
- Import Hooks lazily individually in providers manager (#17682)
- Adding support for multiple task-ids in the external task sensor (#17339)
- Replace `execution_date` with `run_id` in airflow tasks run command (#16666)
- Make output from users cli command more consistent (#17642)
- Open relative extra links in place (#17477)
- Move `worker_log_server_port` option to the logging section (#17621)
- Use gunicorn to serve logs generated by worker (#17591)
- Improve validation of Group id (#17578)
- Simplify 404 page (#17501)
- Add XCom.clear so it's hookable in custom XCom backend (#17405)
- Add deprecation notice for `SubDagOperator` (#17488)
- Support DAGS folder being in different location on scheduler and runners (#16860)
- Remove `/dagrun/create` and disable edit form generated by F.A.B (#17376)
- Enable specifying dictionary paths in `template_fields_renderers` (#17321)
- error early if virtualenv is missing (#15788)
- Handle connection parameters added to Extra and custom fields (#17269)
- Fix `airflow celery stop` to accept the pid file. (#17278)
- Remove DAG refresh buttons (#17263)
- Deprecate dummy trigger rule in favor of always (#17144)
- Be verbose about failure to import `airflow_local_settings` (#17195)
- Include exit code in `AirflowException str` when `BashOperator` fails. (#17151)
- Adding EdgeModifier support for `chain()` (#17099)
- Only allows supported field types to be used in custom connections (#17194)
- Secrets backend failover (#16404)
- Warn on Webserver when using `SQLite` or `SequentialExecutor` (#17133)
- Extend `init_containers` defined in `pod_override` (#17537)
- Client-side filter dag dependencies (#16253)
- Improve executor validation in CLI (#17071)
- Prevent running `airflow db init/upgrade` migrations and setup in parallel. (#17078)

- Update `chain()` and `cross_downstream()` to support `XComArgs` (#16732)
- Improve graph view refresh (#16696)
- When a task instance fails with exception, log it (#16805)
- Set process title for `serve-logs` and `LocalExecutor` (#16644)
- Rename `test_cycle` to `check_cycle` (#16617)
- Add schema as `DbApiHook` instance attribute (#16521, #17423)
- Improve compatibility with MSSQL (#9973)
- Add transparency for unsupported connection type (#16220)
- Call resource based fab methods (#16190)
- Format more dates with timezone (#16129)
- Replace deprecated `dag.sub_dag` with `dag.partial_subset` (#16179)
- Treat `AirflowSensorTimeout` as immediate failure without retrying (#12058)
- Marking success/failed automatically clears failed downstream tasks (#13037)
- Add close/open indicator for import dag errors (#16073)
- Add collapsible import errors (#16072)
- Always return a response in TI's `action_clear` view (#15980)
- Add cli command to delete user by email (#15873)
- Use resource and action names for FAB permissions (#16410)
- Rename DAG concurrency (`[core] dag_concurrency`) settings for easier understanding (#16267, #18730)
- Calendar UI improvements (#16226)
- Refactor: `SKIPPED` should not be logged again as `SUCCESS` (#14822)
- Remove version limits for `dnspython` (#18046, #18162)
- Accept custom run ID in `TriggerDagRunOperator` (#18788)

Bug Fixes

- Make REST API patch user endpoint work the same way as the UI (#18757)
- Properly set `start_date` for cleared tasks (#18708)
- Ensure `task_instance` exists before running update on its state(REST API) (#18642)
- Make `AirflowDateTimePickerWidget` a required field (#18602)
- Retry deadlocked transactions on deleting old rendered task fields (#18616)
- Fix `retry_exponential_backoff` divide by zero error when retry delay is zero (#17003)
- Improve how UI handles datetimes (#18611, #18700)
- Bugfix: `dag_bag.get_dag` should return `None`, not raise exception (#18554)
- Only show the task modal if it is a valid instance (#18570)
- Fix accessing rendered `{{ task.x }}` attributes from within templates (#18516)
- Add missing email type of connection (#18502)
- Don't use flash for "same-page" UI messages. (#18462)

- Fix task group tooltip (#18406)
- Properly fix dagrun update state endpoint (#18370)
- Properly handle ti state difference between executor and scheduler (#17819)
- Fix stuck “queued” tasks in KubernetesExecutor (#18152)
- Don’t permanently add zip DAGs to `sys.path` (#18384)
- Fix random deadlocks in MSSQL database (#18362)
- Deactivating DAGs which have been removed from files (#17121)
- When syncing dags to db remove `dag_tag` rows that are now unused (#8231)
- Graceful scheduler shutdown on error (#18092)
- Fix mini scheduler not respecting `wait_for_downstream` dep (#18338)
- Pass exception to `run_finished_callback` for Debug Executor (#17983)
- Make `XCom.get_one` return full, not abbreviated values (#18274)
- Use try/except when closing temporary file in `task_runner` (#18269)
- show next run if not none (#18273)
- Fix DB session handling in `XCom.set` (#18240)
- Fix `external_executor_id` not being set for manually run jobs (#17207)
- Fix deleting of zipped Dags in Serialized Dag Table (#18243)
- Return explicit error on user-add for duplicated email (#18224)
- Remove loading dots even when last run data is empty (#18230)
- Swap dag import error dropdown icons (#18207)
- Automatically create section when migrating config (#16814)
- Set encoding to utf-8 by default while reading task logs (#17965)
- Apply parent dag permissions to subdags (#18160)
- Change id collation for MySQL to case-sensitive (#18072)
- Logs task launch exception in `StandardTaskRunner` (#17967)
- Applied permissions to `self._error_file` (#15947)
- Fix blank dag dependencies view (#17990)
- Add missing menu access for dag dependencies and configurations pages (#17450)
- Fix passing Jinja templates in `DateTimeSensor` (#17959)
- Fixing bug which restricted the visibility of ImportErrors (#17924)
- Fix grammar in `traceback.html` (#17942)
- Fix `DagRunState` enum query for MySQLdb driver (#17886)
- Fixed button size in “Actions” group. (#17902)
- Only show import errors for DAGs a user can access (#17835)
- Show all import_errors from zip files (#17759)
- fix EXTRA_LOGGER_NAMES param and related docs (#17808)

- Use one interpreter for Airflow and gunicorn (#17805)
- Fix: Mysql 5.7 id utf8mb3 (#14535)
- Fix dag_processing.last_duration metric random holes (#17769)
- Automatically use utf8mb3_general_ci collation for MySQL (#17729)
- fix: filter condition of TaskInstance does not work #17535 (#17548)
- Dont use TaskInstance in CeleryExecutor.trigger_tasks (#16248)
- Remove locks for upgrades in MSSQL (#17213)
- Create virtualenv via python call (#17156)
- Ensure a DAG is acyclic when running `DAG.cli()` (#17105)
- Translate non-ascii characters (#17057)
- Change the logic of None comparison in `model_list` template (#16893)
- Have UI and POST /task_instances_state API endpoint have same behaviour (#16539)
- ensure task is skipped if missing sla (#16719)
- Fix direct use of `cached_property` module (#16710)
- Fix TI success confirm page (#16650)
- Modify return value check in python virtualenv jinja template (#16049)
- Fix dag dependency search (#15924)
- Make custom JSON encoder support `Decimal` (#16383)
- Bugfix: Allow clearing tasks with just `dag_id` and empty `subdir` (#16513)
- Convert port value to a number before calling test connection (#16497)
- Handle missing/null serialized DAG dependencies (#16393)
- Correctly set `dag.fileloc` when using the `@dag` decorator (#16384)
- Fix TI success/failure links (#16233)
- Correctly implement autocomplete early return in `airflow/www/views.py` (#15940)
- Backport fix to allow pickling of Loggers to Python 3.6 (#18798)
- Fix bug that Backfill job fail to run when there are tasks run into `reschedule` state (#17305, #18806)

Doc only changes

- Update `dagbag_size` documentation (#18824)
- Update documentation about bundle extras (#18828)
- Fix wrong Postgres `search_path` set up instructions (#17600)
- Remove `AIRFLOW_GID` from Docker images (#18747)
- Improve error message for BranchPythonOperator when no `task_id` to follow (#18471)
- Improve guidance to users telling them what to do on import timeout (#18478)
- Explain scheduler fine-tuning better (#18356)
- Added example JSON for airflow pools import (#18376)
- Add `sla_miss_callback` section to the documentation (#18305)

- Explain sentry default environment variable for subprocess hook (#18346)
- Refactor installation pages (#18282)
- Improves quick-start docker-compose warnings and documentation (#18164)
- Production-level support for MSSQL (#18382)
- Update non-working example in documentation (#18067)
- Remove default_args pattern + added get_current_context() use for Core Airflow example DAGs (#16866)
- Update max_tis_per_query to better render on the webpage (#17971)
- Adds Github Oauth example with team based authorization (#17896)
- Update docker.rst (#17882)
- Example xcom update (#17749)
- Add doc warning about connections added via env vars (#17915)
- fix wrong documents around upgrade-check.rst (#17903)
- Add Brent to Committers list (#17873)
- Improves documentation about modules management (#17757)
- Remove deprecated metrics from metrics.rst (#17772)
- Make sure “production-readiness” of docker-compose is well explained (#17731)
- Doc: Update Upgrade to v2 docs with Airflow 1.10.x EOL dates (#17710)
- Doc: Replace deprecated param from docstrings (#17709)
- Describe dag owner more carefully (#17699)
- Update note so avoid misinterpretation (#17701)
- Docs: Make `DAG.is_active` read-only in API (#17667)
- Update documentation regarding Python 3.9 support (#17611)
- Fix MySQL database character set instruction (#17603)
- Document overriding `XCom.clear` for data lifecycle management (#17589)
- Path correction in docs for airflow core (#17567)
- docs(celery): reworded, add actual multiple queues example (#17541)
- Doc: Add FAQ to speed up parsing with tons of dag files (#17519)
- Improve image building documentation for new users (#17409)
- Doc: Strip unnecessary arguments from MariaDB JIRA URL (#17296)
- Update warning about MariaDB and multiple schedulers (#17287)
- Doc: Recommend using same configs on all Airflow components (#17146)
- Move docs about masking to a new page (#17007)
- Suggest use of Env vars instead of Airflow Vars in best practices doc (#16926)
- Docs: Better description for `pod_template_file` (#16861)
- Add Aneesh Joseph as Airflow Committer (#16835)
- Docs: Added new pipeline example for the tutorial docs (#16548)

- Remove upstart from docs (#16672)
- Add new committers: Jed and TP (#16671)
- Docs: Fix `flask-ouathlib` to `flask-oauthlib` in Upgrading docs (#16320)
- Docs: Fix creating a connection docs (#16312)
- Docs: Fix url for Elasticsearch (#16275)
- Small improvements for README.md files (#16244)
- Fix docs for `dag_concurrency` (#16177)
- Check syntactic correctness for code-snippets (#16005)
- Add proper link for wheel packages in docs. (#15999)
- Add Docs for `default_pool` slots (#15997)
- Add memory usage warning in quick-start documentation (#15967)
- Update example KubernetesExecutor git-sync pod template file (#15904)
- Docs: Fix Taskflow API docs (#16574)
- Added new pipeline example for the tutorial docs (#16084)
- Updating the DAG docstring to include `render_template_as_native_obj` (#16534)
- Update docs on setting up SMTP (#16523)
- Docs: Fix API verb from POST to PATCH (#16511)

Misc/Internal

- Renaming variables to be consistent with code logic (#18685)
- Simplify strings previously split across lines (#18679)
- fix exception string of `BranchPythonOperator` (#18623)
- Add multiple roles when creating users (#18617)
- Move FABs base Security Manager into Airflow. (#16647)
- Remove unnecessary css state colors (#18461)
- Update boto3 to <1.19 (#18389)
- Improve coverage for `airflow.security.kerberos` module (#18258)
- Fix Amazon Kinesis test (#18337)
- Fix provider test accessing `importlib-resources` (#18228)
- Silence warnings in tests from using SubDagOperator (#18275)
- Fix usage of `range(len())` to `enumerate` (#18174)
- Test coverage on the autocomplete view (#15943)
- Add “packaging” to core requirements (#18122)
- Adds LoggingMixins to BaseTrigger (#18106)
- Fix building docs in main builds (#18035)
- Remove upper-limit on `tenacity` (#17593)
- Remove redundant `numpy` dependency (#17594)

- Bump mysql-connector-python to latest version (#17596)
- Make pandas an optional core dependency (#17575)
- Add more typing to airflow.utils.helpers (#15582)
- Chore: Some code cleanup in airflow/utils/db.py (#17090)
- Refactor: Remove processor_factory from DAG processing (#16659)
- Remove AbstractDagFileProcessorProcess from dag processing (#16816)
- Update TaskGroup typing (#16811)
- Update click to 8.x (#16779)
- Remove remaining Pylint disables (#16760)
- Remove duplicated try, there is already a try in create_session (#16701)
- Removes pylint from our toolchain (#16682)
- Refactor usage of unneeded function call (#16653)
- Add type annotations to setup.py (#16658)
- Remove SQLAlchemy <1.4 constraint (#16630) (Note: our dependencies still have a requirement on <1.4)
- Refactor dag.clear method (#16086)
- Use DAG_ACTIONS constant (#16232)
- Use updated _get_all_non_dag_permissions method (#16317)
- Add updated-name wrappers for built-in FAB methods (#16077)
- Remove TaskInstance.log_filepath attribute (#15217)
- Removes unnecessary function call in airflow/www/app.py (#15956)
- Move plyvel to google provider extra (#15812)
- Update permission migrations to use new naming scheme (#16400)
- Use resource and action names for FAB (#16380)
- Swap out calls to find_permission_view_menu for get_permission wrapper (#16377)
- Fix deprecated default for fab_logging_level to WARNING (#18783)
- Allow running tasks from UI when using CeleryKubernetesExecutor (#18441)

3.16.44 Airflow 2.1.4 (2021-09-18)

Significant Changes

No significant changes.

Bug Fixes

- Fix deprecation error message rather than silencing it (#18126)
- Limit the number of queued dagruns created by the Scheduler (#18065)
- Fix DagRun execution order from queued to running not being properly followed (#18061)
- Fix max_active_runs not allowing moving of queued dagruns to running (#17945)
- Avoid redirect loop for users with no permissions (#17838)

- Avoid endless redirect loop when user has no roles (#17613)
- Fix log links on graph TI modal (#17862)
- Hide variable import form if user lacks permission (#18000)
- Improve dag/task concurrency check (#17786)
- Fix Clear task instances endpoint resets all DAG runs bug (#17961)
- Fixes incorrect parameter passed to views (#18083) (#18085)
- Fix Sentry handler from LocalTaskJob causing error (#18119)
- Limit colorlog version (6.x is incompatible) (#18099)
- Only show Pause/Unpause tooltip on hover (#17957)
- Improve graph view load time for dags with open groups (#17821)
- Increase width for Run column (#17817)
- Fix wrong query on running tis (#17631)
- Add root to tree refresh url (#17633)
- Do not delete running DAG from the UI (#17630)
- Improve discoverability of Provider packages' functionality
- Do not let `create_dagrun` overwrite explicit `run_id` (#17728)
- Regression on pid reset to allow task start after heartbeat (#17333)
- Set task state to failed when pod is DELETED while running (#18095)
- Advises the kernel to not cache log files generated by Airflow (#18054)
- Sort adopted tasks in `_check_for_stalled_adopted_tasks` method (#18208)

Doc only changes

- Update version added fields in `airflow/config_templates/config.yml` (#18128)
- Improve the description of how to handle dynamic task generation (#17963)
- Improve cross-links to operators and hooks references (#17622)
- Doc: Fix replacing Airflow version for Docker stack (#17711)
- Make the providers operators/hooks reference much more usable (#17768)
- Update description about the new `connection-types` provider meta-data
- Suggest to use secrets backend for variable when it contains sensitive data (#17319)
- Separate Installing from sources section and add more details (#18171)
- Doc: Use `closer.lua` script for downloading sources (#18179)
- Doc: Improve installing from sources (#18194)
- Improves installing from sources pages for all components (#18251)

3.16.45 Airflow 2.1.3 (2021-08-23)

Significant Changes

No significant changes.

Bug Fixes

- Fix task retries when they receive `sigkill` and have retries and properly handle `sigterm` (#16301)
- Fix redacting secrets in context exceptions. (#17618)
- Fix race condition with `dagrun` callbacks (#16741)
- Add ‘queued’ to `DagRunState` (#16854)
- Add ‘queued’ state to `DagRun` (#16401)
- Fix external elasticsearch logs link (#16357)
- Add proper warning message when recorded PID is different from current PID (#17411)
- Fix running tasks with `default_ impersonation` config (#17229)
- Rescue if a `DagRun`’s DAG was removed from db (#17544)
- Fixed broken `json_client` (#17529)
- Handle and log exceptions raised during task callback (#17347)
- Fix CLI `kubernetes cleanup-pods` which fails on invalid label key (#17298)
- Show serialization exceptions in DAG parsing log (#17277)
- Fix: `TaskInstance` does not show `queued_by_job_id` & `external_executor_id` (#17179)
- Adds more explanatory message when `SecretsMasker` is not configured (#17101)
- Enable the use of `__init_subclass__` in subclasses of `BaseOperator` (#17027)
- Fix task instance retrieval in XCom view (#16923)
- Validate type of `priority_weight` during parsing (#16765)
- Correctly handle custom `deps` and `task_group` during DAG Serialization (#16734)
- Fix slow (cleared) tasks being adopted by Celery worker. (#16718)
- Fix calculating duration in tree view (#16695)
- Fix `AttributeError: datetime.timezone object has no attribute name` (#16599)
- Redact conn secrets in webserver logs (#16579)
- Change graph focus to top of view instead of center (#16484)
- Fail tasks in scheduler when executor reports they failed (#15929)
- fix(`smart_sensor`): Unbound variable errors (#14774)
- Add back missing permissions to `UserModelView` controls. (#17431)
- Better diagnostics and self-healing of docker-compose (#17484)
- Improve diagnostics message when users have `secret_key` misconfigured (#17410)
- Stop checking `execution_date` in `task_instance.refresh_from_db` (#16809)

Improvements

- Run mini scheduler in LocalTaskJob during task exit (#16289)
- Remove SQLAlchemy<1.4 constraint (#16630)
- Bump Jinja2 upper-bound from 2.12.0 to 4.0.0 (#16595)
- Bump dnspython (#16698)
- Updates to FlaskAppBuilder 3.3.2+ (#17208)
- Add State types for tasks and DAGs (#15285)
- Set Process title for Worker when using LocalExecutor (#16623)
- Move DagFileProcessor and DagFileProcessorProcess out of scheduler_job.py (#16581)

Doc only changes

- Fix inconsistencies in configuration docs (#17317)
- Fix docs link for using SQLite as Metadata DB (#17308)

Misc

- Switch back http provider after requests removes LGPL dependency (#16974)

3.16.46 Airflow 2.1.2 (2021-07-14)

Significant Changes

No significant changes.

Bug Fixes

- Only allow webserver to request from the worker log server (#16754)
- Fix “Invalid JSON configuration, must be a dict” bug (#16648)
- Fix CeleryKubernetesExecutor (#16700)
- Mask value if the key is token (#16474)
- Fix impersonation issue with LocalTaskJob (#16852)
- Resolve all npm vulnerabilities including bumping jQuery to 3.5 (#16440)

Misc

- Add Python 3.9 support (#15515)

3.16.47 Airflow 2.1.1 (2021-07-02)

Significant Changes

activate_dag_runs argument of the function clear_task_instances is replaced with dag_run_state

To achieve the previous default behaviour of `clear_task_instances` with `activate_dag_runs=True`, no change is needed. To achieve the previous behaviour of `activate_dag_runs=False`, pass `dag_run_state=False` instead. (The previous parameter is still accepted, but is deprecated)

`dag.set_dag_runs_state` is deprecated

The method `set_dag_runs_state` is no longer needed after a bug fix in PR: #15382. This method is now deprecated and will be removed in a future version.

Bug Fixes

- Don't crash attempting to mask secrets in dict with non-string keys (#16601)
- Always install sphinx_airflow_theme from PyPI (#16594)
- Remove limitation for elasticsearch library (#16553)
- Adding extra requirements for build and runtime of the PROD image. (#16170)
- Cattrs 1.7.0 released by the end of May 2021 break lineage usage (#16173)
- Removes unnecessary packages from setup_requires (#16139)
- Pins docutils to <0.17 until breaking behaviour is fixed (#16133)
- Improvements for Docker Image docs (#14843)
- Ensure that `dag_run.conf` is a dict (#15057)
- Fix CLI connections import and migrate logic from secrets to Connection model (#15425)
- Fix Dag Details start date bug (#16206)
- Fix DAG run state not updated while DAG is paused (#16343)
- Allow null value for operator field in task_instance schema(REST API) (#16516)
- Avoid recursion going too deep when redacting logs (#16491)
- Backfill: Don't create a DagRun if no tasks match task regex (#16461)
- Tree View UI for larger DAGs & more consistent spacing in Tree View (#16522)
- Correctly handle None returns from Query.scalar() (#16345)
- Adding `only_active` parameter to /dags endpoint (#14306)
- Don't show stale Serialized DAGs if they are deleted in DB (#16368)
- Make REST API List DAGs endpoint consistent with UI/CLI behaviour (#16318)
- Support remote logging in elasticsearch with filebeat 7 (#14625)
- Queue tasks with higher priority and earlier execution_date first. (#15210)
- Make task ID on legend have enough width and width of line chart to be 100%. (#15915)
- Fix normalize-url vulnerability (#16375)
- Validate retries value on init for better errors (#16415)
- add num_runs query param for tree refresh (#16437)
- Fix templated default/example values in config ref docs (#16442)
- Add passphrase and private_key to default sensitive field names (#16392)
- Fix tasks in an infinite slots pool were never scheduled (#15247)
- Fix Orphaned tasks stuck in CeleryExecutor as running (#16550)
- Don't fail to log if we can't redact something (#16118)
- Set max tree width to 1200 pixels (#16067)

- Fill the “job_id” field for `airflow task run` without `--local`/`--raw` for KubeExecutor (#16108)
- Fixes problem where conf variable was used before initialization (#16088)
- Fix apply defaults for task decorator (#16085)
- Parse recently modified files even if just parsed (#16075)
- Ensure that we don’t try to mask empty string in logs (#16057)
- Don’t die when masking `log.exception` when there is no exception (#16047)
- Restores `apply_defaults` import in `base_sensor_operator` (#16040)
- Fix auto-refresh in tree view When webserver ui is not in / (#16018)
- Fix `dag.clear()` to set multiple dags to running when necessary (#15382)
- Fix Celery executor getting stuck randomly because of `reset_signals` in multiprocessing (#15989)

3.16.48 Airflow 2.1.0 (2021-05-21)

Significant Changes

New “`deprecated_api`” extra

We have a new ‘`[deprecated_api]`’ extra that should be used when installing airflow when the deprecated API is going to be used. This is now an optional feature of Airflow now because it pulls in `requests` which (as of 14 May 2021) pulls LGPL chardet dependency.

The `http` provider is not installed by default

The `http` provider is now optional and not installed by default, until `chardet` becomes an optional dependency of `requests`. See [PR to replace chardet with charset-normalizer](#)

`@apply_default` decorator isn’t longer necessary

This decorator is now automatically added to all operators via the metaclass on `BaseOperator`

Change the configuration options for field masking

We’ve improved masking for sensitive data in Web UI and logs. As part of it, the following configurations have been changed:

- `hide_sensitive_variable_fields` option in `admin` section has been replaced by `hide_sensitive_var_conn_fields` section in `core` section,
- `sensitive_variable_fields` option in `admin` section has been replaced by `sensitive_var_conn_names` section in `core` section.

Deprecated `PodDefaults` and `add_xcom_sidecar` in `airflow.kubernetes.pod_generator`

We have moved `PodDefaults` from `airflow.kubernetes.pod_generator.PodDefaults` to `airflow.providers.cncf.kubernetes.utils.xcom_sidecar.PodDefaults` and moved `add_xcom_sidecar` from `airflow.kubernetes.pod_generator.PodGenerator.add_xcom_sidecar` to `airflow.providers.cncf.kubernetes.utils.xcom_sidecar.add_xcom_sidecar`. This change will allow us to modify the KubernetesPodOperator XCom functionality without requiring airflow upgrades.

Removed pod_launcher from core airflow

Moved the pod launcher from `airflow.kubernetes.pod_launcher` to `airflow.providers.cncf.kubernetes.utils.pod_launcher`

This will allow users to update the `pod_launcher` for the `KubernetesPodOperator` without requiring an airflow upgrade

Default [webserver] worker_refresh_interval is changed to 6000 seconds

The default value for `[webserver] worker_refresh_interval` was 30 seconds for Airflow <=2.0.1. However, since Airflow 2.0 DAG Serialization is a hard requirement and the Webserver used the serialized DAGs, there is no need to kill an existing worker and create a new one as frequently as 30 seconds.

This setting can be raised to an even higher value, currently it is set to 6000 seconds (100 minutes) to serve as a DagBag cache burst time.

default_queue configuration has been moved to the operators section.

The `default_queue` configuration option has been moved from `[celery]` section to `[operators]` section to allow for reuse between different executors.

New Features

- Add `PythonVirtualenvDecorator` to Taskflow API (#14761)
- Add `Taskgroup` decorator (#15034)
- Create a DAG Calendar View (#15423)
- Create cross-DAG dependencies view (#13199)
- Add rest API to query for providers (#13394)
- Mask passwords and sensitive info in task logs and UI (#15599)
- Add `SubprocessHook` for running commands from operators (#13423)
- Add DAG Timeout in UI page “DAG Details” (#14165)
- Add `WeekDayBranchOperator` (#13997)
- Add JSON linter to DAG Trigger UI (#13551)
- Add DAG Description Doc to Trigger UI Page (#13365)
- Add airflow webserver URL into SLA miss email. (#13249)
- Add read only REST API endpoints for users (#14735)
- Add files to generate Airflow’s Python SDK (#14739)
- Add dynamic fields to snowflake connection (#14724)
- Add read only REST API endpoint for roles and permissions (#14664)
- Add new datetime branch operator (#11964)
- Add Google leveldb hook and operator (#13109) (#14105)
- Add plugins endpoint to the REST API (#14280)
- Add `worker_pod_pending_timeout` support (#15263)
- Add support for labeling DAG edges (#15142)
- Add CUD REST API endpoints for Roles (#14840)

- Import connections from a file (#15177)
- A bunch of `template_fields_renderers` additions (#15130)
- Add REST API query sort and order to some endpoints (#14895)
- Add timezone context in new ui (#15096)
- Add query mutations to new UI (#15068)
- Add different modes to sort dag files for parsing (#15046)
- Auto refresh on Tree View (#15474)
- BashOperator to raise `AirflowSkipException` on exit code 99 (by default, configurable) (#13421) (#14963)
- Clear tasks by task ids in REST API (#14500)
- Support jinja2 native Python types (#14603)
- Allow celery workers without gossip or mingle modes (#13880)
- Add `airflow jobs check` CLI command to check health of jobs (Scheduler etc) (#14519)
- Rename `DateTimeBranchOperator` to `BranchDateTimeOperator` (#14720)

Improvements

- Add optional result handler callback to `DbApiHook` (#15581)
- Update Flask App Builder limit to recently released 3.3 (#15792)
- Prevent creating flask sessions on REST API requests (#15295)
- Sync DAG specific permissions when parsing (#15311)
- Increase maximum length of pool name on Tasks to 256 characters (#15203)
- Enforce READ COMMITTED isolation when using mysql (#15714)
- Auto-apply `apply_default` to subclasses of `BaseOperator` (#15667)
- Emit error on duplicated DAG ID (#15302)
- Update `KubernetesExecutor` pod templates to allow access to IAM permissions (#15669)
- More verbose logs when running `airflow db check-migrations` (#15662)
- When `one_success` mark task as failed if no success (#15467)
- Add an option to trigger a dag w/o changing conf (#15591)
- Add Airflow UI instance_name configuration option (#10162)
- Add a decorator to retry functions with DB transactions (#14109)
- Add return to `PythonVirtualenvOperator`'s execute method (#14061)
- Add `verify_ssl` config for kubernetes (#13516)
- Add description about `secret_key` when Webserver > 1 (#15546)
- Add Traceback in LogRecord in `JSONFormatter` (#15414)
- Add support for arbitrary json in conn uri format (#15100)
- Adds description field in variable (#12413) (#15194)
- Add logs to show last modified in SFTP, FTP and Filesystem sensor (#15134)
- Execute `on_failure_callback` when SIGTERM is received (#15172)

- Allow hiding of all edges when highlighting states (#15281)
- Display explicit error in case UID has no actual username (#15212)
- Serve logs with Scheduler when using Local or Sequential Executor (#15557)
- Deactivate trigger, refresh, and delete controls on dag detail view. (#14144)
- Turn off autocomplete for connection forms (#15073)
- Increase default `worker_refresh_interval` to 6000 seconds (#14970)
- Only show User's local timezone if it's not UTC (#13904)
- Suppress LOG/WARNING for a few tasks CLI for better CLI experience (#14567)
- Configurable API response (CORS) headers (#13620)
- Allow viewers to see all docs links (#14197)
- Update Tree View date ticks (#14141)
- Make the tooltip to Pause / Unpause a DAG clearer (#13642)
- Warn about precedence of env var when getting variables (#13501)
- Move `[celery] default_queue` config to `[operators] default_queue` to reuse between executors (#14699)

Bug Fixes

- Fix 500 error from `updateTaskInstancesState` API endpoint when `dry_run` not passed (#15889)
- Ensure that task preceding a `PythonVirtualenvOperator` doesn't fail (#15822)
- Prevent mixed case env vars from crashing processes like worker (#14380)
- Fixed type annotations in DAG decorator (#15778)
- Fix `on_failure_callback` when task receive SIGKILL (#15537)
- Fix dags table overflow (#15660)
- Fix changing the parent dag state on subdag clear (#15562)
- Fix reading from zip package to default to text (#13962)
- Fix wrong parameter for `drawDagStatsForDag` in `dags.html` (#13884)
- Fix `QueuedLocalWorker` crashing with EOFError (#13215)
- Fix typo in `NotPreviouslySkippedDep` (#13933)
- Fix parallelism after KubeExecutor pod adoption (#15555)
- Fix kube client on mac with keepalive enabled (#15551)
- Fixes wrong limit for dask for python>3.7 (should be <3.7) (#15545)
- Fix Task Adoption in `KubernetesExecutor` (#14795)
- Fix timeout when using XCom with `KubernetesPodOperator` (#15388)
- Fix deprecated provider aliases in "extras" not working (#15465)
- Fixed default XCom deserialization. (#14827)
- Fix `used_group_ids` in `dag.partial_subset` (#13700) (#15308)
- Further fix trimmed `pod_id` for `KubernetesPodOperator` (#15445)

- Bugfix: Invalid name when trimmed `pod_id` ends with hyphen in `KubernetesPodOperator` (#15443)
- Fix incorrect slots stats when `TI pool_slots > 1` (#15426)
- Fix DAG last run link (#15327)
- Fix `sync-perm` to work correctly when `update_fab_perms = False` (#14847)
- Fixes limits on Arrow for plexus test (#14781)
- Fix UI bugs in tree view (#14566)
- Fix AzureDataFactoryHook failing to instantiate its connection (#14565)
- Fix permission error on non-POSIX filesystem (#13121)
- Fix spelling in “ignorable” (#14348)
- Fix `get_context_data` doctest import (#14288)
- Correct typo in `GCSObjectsWithPrefixExistenceSensor` (#14179)
- Fix order of failed deps (#14036)
- Fix critical CeleryKubernetesExecutor bug (#13247)
- Fix four bugs in StackdriverTaskHandler (#13784)
- `func.sum` may return `Decimal` that break rest APIs (#15585)
- Persist tags params in pagination (#15411)
- API: Raise `AlreadyExists` exception when the `execution_date` is same (#15174)
- Remove duplicate call to `sync_metadata` inside `DagFileProcessorManager` (#15121)
- Extra `docker-py` update to resolve docker op issues (#15731)
- Ensure executors end method is called (#14085)
- Remove `user_id` from API schema (#15117)
- Prevent clickable bad links on disabled pagination (#15074)
- Acquire lock on db for the time of migration (#10151)
- Skip SLA check only if SLA is None (#14064)
- Print right version in airflow info command (#14560)
- Make `airflow info` work with pipes (#14528)
- Rework client-side script for connection form. (#14052)
- API: Add `CollectionInfo` in all Collections that have `total_entries` (#14366)
- Fix `task_instance_mutation_hook` when importing `airflow.models.dagrun` (#15851)

Doc only changes

- Fix docstring of `SqlSensor` (#15466)
- Small changes on “DAGs and Tasks documentation” (#14853)
- Add note on changes to configuration options (#15696)
- Add docs to the `markdownlint` and `yamllint` config files (#15682)
- Rename old “Experimental” API to deprecated in the docs. (#15653)
- Fix documentation error in `git_sync_template.yaml` (#13197)

- Fix doc link permission name (#14972)
- Fix link to Helm chart docs (#14652)
- Fix docstrings for Kubernetes code (#14605)
- docs: Capitalize & minor fixes (#14283) (#14534)
- Fixed reading from zip package to default to text. (#13984)
- An initial rework of the “Concepts” docs (#15444)
- Improve docstrings for various modules (#15047)
- Add documentation on database connection URI (#14124)
- Add Helm Chart logo to docs index (#14762)
- Create a new documentation package for Helm Chart (#14643)
- Add docs about supported logging levels (#14507)
- Update docs about tableau and salesforce provider (#14495)
- Replace deprecated doc links to the correct one (#14429)
- Refactor redundant doc url logic to use utility (#14080)
- docs: NOTICE: Updated 2016-2019 to 2016-now (#14248)
- Skip DAG perm sync during parsing if possible (#15464)
- Add picture and examples for Edge Labels (#15310)
- Add example DAG & how-to guide for sqlite (#13196)
- Add links to new modules for deprecated modules (#15316)
- Add note in Updating.md about FAB data model change (#14478)

Misc/Internal

- Fix `logging.exception` redundancy (#14823)
- Bump `stylelint` to remove vulnerable sub-dependency (#15784)
- Add resolution to force dependencies to use patched version of `lodash` (#15777)
- Update `croniter` to 1.0.x series (#15769)
- Get rid of Airflow 1.10 in `Breeze` (#15712)
- Run helm chart tests in parallel (#15706)
- Bump `ssri` from 6.0.1 to 6.0.2 in `/airflow/www` (#15437)
- Remove the limit on `Gunicorn` dependency (#15611)
- Better “dependency already registered” warning message for tasks #14613 (#14860)
- Pin `pandas-gbq` to <0.15.0 (#15114)
- Use Pip 21.* to install airflow officially (#15513)
- Bump `mysqlclient` to support the 1.4.x and 2.x series (#14978)
- Finish refactor of DAG resource name helper (#15511)
- Refactor/Cleanup Presentation of Graph Task and Path Highlighting (#15257)
- Standardize default fab perms (#14946)

- Remove `datepicker` for task instance detail view (#15284)
- Turn provider's import warnings into debug logs (#14903)
- Remove left-over fields from required in provider_info schema. (#14119)
- Deprecate `tableau extra` (#13595)
- Use built-in `cached_property` on Python 3.8 where possible (#14606)
- Clean-up JS code in UI templates (#14019)
- Bump elliptic from 6.5.3 to 6.5.4 in /airflow/www (#14668)
- Switch to f-strings using `flynt`. (#13732)
- use `jquery ready` instead of vanilla js (#15258)
- Migrate task instance log (`ti_log`) js (#15309)
- Migrate graph js (#15307)
- Migrate dags.html javascript (#14692)
- Removes unnecessary AzureContainerInstance connection type (#15514)
- Separate Kubernetes pod_launcher from core airflow (#15165)
- update remaining old import paths of operators (#15127)
- Remove broken and undocumented “demo mode” feature (#14601)
- Simplify configuration/legibility of Webpack entries (#14551)
- remove inline tree js (#14552)
- Js linting and inline migration for simple scripts (#14215)
- Remove use of repeated constant in AirflowConfigParser (#14023)
- Deprecate email credentials from environment variables. (#13601)
- Remove unused ‘context’ variable in `task_instance.py` (#14049)
- Disable `suppress_logs_and_warning` in cli when debugging (#13180)

3.16.49 Airflow 2.0.2 (2021-04-19)

Significant Changes

`Default [kubernetes] enable_tcp_keepalive is changed to True`

This allows Airflow to work more reliably with some environments (like Azure) by default.

`sync-perm` CLI no longer syncs DAG specific permissions by default

The `sync-perm` CLI command will no longer sync DAG specific permissions by default as they are now being handled during DAG parsing. If you need or want the old behavior, you can pass `--include-dags` to have `sync-perm` also sync DAG specific permissions.

Bug Fixes

- Bugfix: `TypeError` when Serializing & sorting iterable properties of DAGs (#15395)
- Fix missing `on_load` trigger for folder-based plugins (#15208)
- `kubernetes cleanup-pods` subcommand will only clean up Airflow-created Pods (#15204)

- Fix password masking in CLI action_logging (#15143)
- Fix url generation for TriggerDagRunOperatorLink (#14990)
- Restore base lineage backend (#14146)
- Unable to trigger backfill or manual jobs with Kubernetes executor. (#14160)
- Bugfix: Task docs are not shown in the Task Instance Detail View (#15191)
- Bugfix: Fix overriding pod_template_file in KubernetesExecutor (#15197)
- Bugfix: resources in executor_config breaks Graph View in UI (#15199)
- Fix celery executor bug trying to call len on map (#14883)
- Fix bug in airflow.stats timing that broke dogstatsd mode (#15132)
- Avoid scheduler/parser manager deadlock by using non-blocking IO (#15112)
- Re-introduce dagrun.schedule_delay metric (#15105)
- Compare string values, not if strings are the same object in Kube executor (#14942)
- Pass queue to BaseExecutor.execute_async like in airflow 1.10 (#14861)
- Scheduler: Remove TIs from starved pools from the critical path. (#14476)
- Remove extra/needless deprecation warnings from airflow.contrib module (#15065)
- Fix support for long dag_id and task_id in KubernetesExecutor (#14703)
- Sort lists, sets and tuples in Serialized DAGs (#14909)
- Simplify cleaning string passed to origin param (#14738) (#14905)
- Fix error when running tasks with Sentry integration enabled. (#13929)
- Webserver: Sanitize string passed to origin param (#14738)
- Fix losing duration < 1 secs in tree (#13537)
- Pin SQLAlchemy to <1.4 due to breakage of sqlalchemy-utils (#14812)
- Fix KubernetesExecutor issue with deleted pending pods (#14810)
- Default to Celery Task model when backend model does not exist (#14612)
- Bugfix: Plugins endpoint was unauthenticated (#14570)
- BugFix: fix DAG doc display (especially for TaskFlow DAGs) (#14564)
- BugFix: TypeError in airflow.kubernetes.pod_launcher's monitor_pod (#14513)
- Bugfix: Fix wrong output of tags and owners in dag detail API endpoint (#14490)
- Fix logging error with task error when JSON logging is enabled (#14456)
- Fix StatsD metrics not sending when using daemon mode (#14454)
- Gracefully handle missing start_date and end_date for DagRun (#14452)
- BugFix: Serialize max_retry_delay as a timedelta (#14436)
- Fix crash when user clicks on “Task Instance Details” caused by start_date being None (#14416)
- BugFix: Fix TaskInstance API call fails if a task is removed from running DAG (#14381)
- Scheduler should not fail when invalid executor_config is passed (#14323)
- Fix bug allowing task instances to survive when dagrun_timeout is exceeded (#14321)

- Fix bug where DAG timezone was not always shown correctly in UI tooltips (#14204)
- Use Lax for `cookie_samesite` when empty string is passed (#14183)
- [AIRFLOW-6076] fix `dag.cli()` KeyError (#13647)
- Fix running child tasks in a subdag after clearing a successful subdag (#14776)

Improvements

- Remove unused JS packages causing false security alerts (#15383)
- Change default of `[kubernetes].enable_tcp_keepalive` for new installs to True (#15338)
- Fixed #14270: Add error message in OOM situations (#15207)
- Better compatibility/diagnostics for arbitrary UID in docker image (#15162)
- Updates 3.6 limits for latest versions of a few libraries (#15209)
- Adds Blinker dependency which is missing after recent changes (#15182)
- Remove ‘conf’ from search_columns in DagRun View (#15099)
- More proper default value for namespace in K8S cleanup-pods CLI (#15060)
- Faster default role syncing during webserver start (#15017)
- Speed up webserver start when there are many DAGs (#14993)
- Much easier to use and better documented Docker image (#14911)
- Use `libyaml` C library when available. (#14577)
- Don’t create `unittest.cfg` when not running in unit test mode (#14420)
- Webserver: Allow Filtering TaskInstances by `queued_dttm` (#14708)
- Update Flask-AppBuilder dependency to allow 3.2 (and all 3.x series) (#14665)
- Remember expanded task groups in browser local storage (#14661)
- Add plain format output to cli tables (#14546)
- Make `airflow dags show` command display TaskGroups (#14269)
- Increase maximum size of `extra` connection field. (#12944)
- Speed up `clear_task_instances` by doing a single sql delete for `TaskReschedule` (#14048)
- Add more flexibility with FAB menu links (#13903)
- Add better description and guidance in case of sqlite version mismatch (#14209)

Doc only changes

- Add documentation create/update community providers (#15061)
- Fix mistake and typos in `airflow.utils.timezone` docstrings (#15180)
- Replace new url for Stable Airflow Docs (#15169)
- Docs: Clarify behavior of `delete_worker_pods_on_failure` (#14958)
- Create a documentation package for Docker image (#14846)
- Multiple minor doc (OpenAPI) fixes (#14917)
- Replace Graph View Screenshot to show Auto-refresh (#14571)

Misc/Internal

- Import Connection lazily in hooks to avoid cycles (#15361)
- Rename last_scheduler_run into last_parsed_time, and ensure it's updated in DB (#14581)
- Make TaskInstance.pool_slots not nullable with a default of 1 (#14406)
- Log migrations info in consistent way (#14158)

3.16.50 Airflow 2.0.1 (2021-02-08)

Significant Changes

Permission to view Airflow Configurations has been removed from User and Viewer role

Previously, Users with User or Viewer role were able to get/view configurations using the REST API or in the Web-server. From Airflow 2.0.1, only users with Admin or Op role would be able to get/view Configurations.

To allow users with other roles to view configuration, add can_read on Configurations permissions to that role.

Note that if [webserver] expose_config is set to False, the API will throw a 403 response even if the user has role with can_read on Configurations permission.

Default [celery] worker_concurrency is changed to 16

The default value for [celery] worker_concurrency was 16 for Airflow <2.0.0. However, it was unintentionally changed to 8 in 2.0.0.

From Airflow 2.0.1, we revert to the old default of 16.

Default [scheduler] min_file_process_interval is changed to 30

The default value for [scheduler] min_file_process_interval was 0, due to which the CPU Usage mostly stayed around 100% as the DAG files are parsed constantly.

From Airflow 2.0.0, the scheduling decisions have been moved from DagFileProcessor to Scheduler, so we can keep the default a bit higher: 30.

Bug Fixes

- Bugfix: Return XCom Value in the XCom Endpoint API (#13684)
- Bugfix: Import error when using custom backend and sqlalchemy_conn_secret (#13260)
- Allow PID file path to be relative when daemonize a process (scheduler, kerberos, etc) (#13232)
- Bugfix: no generic DROP CONSTRAINT in MySQL during airflow db upgrade (#13239)
- Bugfix: Sync Access Control defined in DAGs when running sync-perm (#13377)
- Stop sending Callback Requests if no callbacks are defined on DAG (#13163)
- BugFix: Dag-level Callback Requests were not run (#13651)
- Stop creating duplicate Dag File Processors (#13662)
- Filter DagRuns with Task Instances in removed State while Scheduling (#13165)
- Bump datatables.net from 1.10.21 to 1.10.22 in /airflow/www (#13143)
- Bump datatables.net JS to 1.10.23 (#13253)
- Bump dompurify from 2.0.12 to 2.2.6 in /airflow/www (#13164)

- Update minimum `cattrs` version (#13223)
- Remove inapplicable arg ‘output’ for CLI pools import/export (#13071)
- Webserver: Fix the behavior to deactivate the authentication option and add docs (#13191)
- Fix: add support for no-menu plugin views (#11742)
- Add `python-daemon` limit for Python 3.8+ to fix daemon crash (#13540)
- Change the default celery `worker_concurrency` to 16 (#13612)
- Audit Log records View should not contain link if `dag_id` is None (#13619)
- Fix invalid `continue_token` for cleanup list pods (#13563)
- Switches to latest version of snowflake connector (#13654)
- Fix backfill crash on task retry or reschedule (#13712)
- Setting `max_tis_per_query` to 0 now correctly removes the limit (#13512)
- Fix race conditions in task callback invocations (#10917)
- Fix webserver exiting when gunicorn master crashes (#13518)(#13780)
- Fix SQL syntax to check duplicate connections (#13783)
- `BaseBranchOperator` will push to xcom by default (#13704) (#13763)
- Fix Deprecation for `configuration.getsection` (#13804)
- Fix TaskNotFound in log endpoint (#13872)
- Fix race condition when using Dynamic DAGs (#13893)
- Fix: Linux/Chrome window bouncing in Webserver
- Fix db shell for sqlite (#13907)
- Only compare updated time when Serialized DAG exists (#13899)
- Fix dag run type enum query for `mysqldb` driver (#13278)
- Add authentication to lineage endpoint for experimental API (#13870)
- Do not add User role perms to custom roles. (#13856)
- Do not add `Website.can_read` access to default roles. (#13923)
- Fix invalid value error caused by long Kubernetes pod name (#13299)
- Fix DB Migration for SQLite to upgrade to 2.0 (#13921)
- Bugfix: Manual DagRun trigger should not skip scheduled runs (#13963)
- Stop loading Extra Operator links in Scheduler (#13932)
- Added missing return parameter in read function of `FileTaskHandler` (#14001)
- Bugfix: Do not try to create a duplicate Dag Run in Scheduler (#13920)
- Make `v1/config` endpoint respect webserver `expose_config` setting (#14020)
- Disable row level locking for Mariadb and MySQL <8 (#14031)
- Bugfix: Fix permissions to triggering only specific DAGs (#13922)
- Fix broken SLA Mechanism (#14056)
- Bugfix: Scheduler fails if task is removed at runtime (#14057)

- Remove permissions to read Configurations for User and Viewer roles (#14067)
- Fix DB Migration from 2.0.1rc1

Improvements

- Increase the default `min_file_process_interval` to decrease CPU Usage (#13664)
- Dispose connections when running tasks with `os.fork` & `CeleryExecutor` (#13265)
- Make function purpose clearer in `example_kubernetes_executor` example dag (#13216)
- Remove unused libraries - `flask-swagger`, `funcsig` (#13178)
- Display alternative tooltip when a Task has yet to run (no TI) (#13162)
- User werkzeug's own type conversion for request args (#13184)
- UI: Add `queued_by_job_id` & `external_executor_id` Columns to TI View (#13266)
- Make `json-merge-patch` an optional library and unpin it (#13175)
- Adds missing LDAP “extra” dependencies to ldap provider. (#13308)
- Refactor `setup.py` to better reflect changes in providers (#13314)
- Pin `pyjwt` and Add integration tests for Apache Pinot (#13195)
- Removes provider-imposed requirements from `setup.cfg` (#13409)
- Replace deprecated decorator (#13443)
- Streamline & simplify `__eq__` methods in models Dag and BaseOperator (#13449)
- Additional properties should be allowed in provider schema (#13440)
- Remove unused dependency - `contextdecorator` (#13455)
- Remove ‘typing’ dependency (#13472)
- Log migrations info in consistent way (#13458)
- Unpin `mysql-connector-python` to allow 8.0.22 (#13370)
- Remove thrift as a core dependency (#13471)
- Add `NotFound` response for DELETE methods in OpenAPI YAML (#13550)
- Stop Log Spamming when `[core] lazy_load_plugins` is `False` (#13578)
- Display message and docs link when no plugins are loaded (#13599)
- Unpin restriction for `colorlog` dependency (#13176)
- Add missing Dag Tag for Example DAGs (#13665)
- Support tables in DAG docs (#13533)
- Add `python3-openid` dependency (#13714)
- Add `__repr__` for Executors (#13753)
- Add description to hint if `conn_type` is missing (#13778)
- Upgrade Azure blob to v12 (#12188)
- Add extra field to `get_connection` REST endpoint (#13885)
- Make Smart Sensors DB Migration idempotent (#13892)
- Improve the error when DAG does not exist when running dag pause command (#13900)

- Update `airflow_local_settings.py` to fix an error message (#13927)
- Only allow passing JSON Serializable conf to `TriggerDagRunOperator` (#13964)
- Bugfix: Allow getting details of a DAG with null `start_date` (REST API) (#13959)
- Add params to the DAG details endpoint (#13790)
- Make the role assigned to anonymous users customizable (#14042)
- Retry critical methods in Scheduler loop in case of `OperationalError` (#14032)

Doc only changes

- Add Missing StatsD Metrics in Docs (#13708)
- Add Missing Email configs in Configuration doc (#13709)
- Add quick start for Airflow on Docker (#13660)
- Describe which Python versions are supported (#13259)
- Add note block to 2.x migration docs (#13094)
- Add documentation about `webserver_config.py` (#13155)
- Add missing version information to recently added configs (#13161)
- API: Use generic information in `UpdateMask` component (#13146)
- Add Airflow 2.0.0 to requirements table (#13140)
- Avoid confusion in doc for CeleryKubernetesExecutor (#13116)
- Update docs link in REST API spec (#13107)
- Add link to PyPI Repository to provider docs (#13064)
- Fix link to Airflow master branch documentation (#13179)
- Minor enhancements to Sensors docs (#13381)
- Use 2.0.0 in Airflow docs & Breeze (#13379)
- Improves documentation regarding providers and custom connections (#13375)(#13410)
- Fix malformed table in `production-deployment.rst` (#13395)
- Update `celery.rst` to fix broken links (#13400)
- Remove reference to scheduler `run_duration` param in docs (#13346)
- Set minimum SQLite version supported (#13412)
- Fix installation doc (#13462)
- Add docs about mocking variables and connections (#13502)
- Add docs about Flask CLI (#13500)
- Fix Upgrading to 2 guide to use rbac UI (#13569)
- Make docs clear that Auth can not be disabled for Stable API (#13568)
- Remove archived links from docs & add link for AIPs (#13580)
- Minor fixes in `upgrading-to-2.rst` (#13583)
- Fix Link in Upgrading to 2.0 guide (#13584)
- Fix heading for Mocking section in `best-practices.rst` (#13658)

- Add docs on how to use custom operators within plugins folder (#13186)
- Update docs to register Operator Extra Links (#13683)
- Improvements for database setup docs (#13696)
- Replace module path to Class with just Class Name (#13719)
- Update DAG Serialization docs (#13722)
- Fix link to Apache Airflow docs in webserver (#13250)
- Clarifies differences between extras and provider packages (#13810)
- Add information about all access methods to the environment (#13940)
- Docs: Fix FAQ on scheduler latency (#13969)
- Updated taskflow api doc to show dependency with sensor (#13968)
- Add deprecated config options to docs (#13883)
- Added a FAQ section to the Upgrading to 2 doc (#13979)

3.16.51 Airflow 2.0.0 (2020-12-18)

The full changelog is about 3,000 lines long (already excluding everything backported to 1.10) so please check [Airflow 2.0.0 Highlights Blog Post](#) instead.

Significant Changes

The 2.0 release of the Airflow is a significant upgrade, and includes substantial major changes, and some of them may be breaking. Existing code written for earlier versions of this project will require updates to use this version. Sometimes necessary configuration changes are also required. This document describes the changes that have been made, and what you need to do to update your usage.

If you experience issues or have questions, please file [an issue](#).

Major changes

This section describes the major changes that have been made in this release.

The experimental REST API is disabled by default

The experimental REST API is disabled by default. To restore these APIs while migrating to the stable REST API, set `enable_experimental_api` option in `[api]` section to `True`.

Please note that the experimental REST API do not have access control. The authenticated user has full access.

SparkJDBCHook default connection

For SparkJDBCHook default connection was `spark-default`, and for SparkSubmitHook it was `spark_default`. Both hooks now use the `spark_default` which is a common pattern for the connection names used across all providers.

Changes to output argument in commands

From Airflow 2.0, We are replacing `tabulate` with `rich` to render commands output. Due to this change, the `--output` argument will no longer accept formats of tabulate tables. Instead, it now accepts:

- `table` - will render the output in predefined table
- `json` - will render the output as a json

- `yaml` - will render the output as yaml

By doing this we increased consistency and gave users possibility to manipulate the output programmatically (when using json or yaml).

Affected commands:

- `airflow dags list`
- `airflow dags report`
- `airflow dags list-runs`
- `airflow dags list-jobs`
- `airflow connections list`
- `airflow connections get`
- `airflow pools list`
- `airflow pools get`
- `airflow pools set`
- `airflow pools delete`
- `airflow pools import`
- `airflow pools export`
- `airflow role list`
- `airflow providers list`
- `airflow providers get`
- `airflow providers hooks`
- `airflow tasks states-for-dag-run`
- `airflow users list`
- `airflow variables list`

Azure Wasb Hook does not work together with Snowflake hook

The WasbHook in Apache Airflow use a legacy version of Azure library. While the conflict is not significant for most of the Azure hooks, it is a problem for Wasb Hook because the blob folders for both libraries overlap. Installing both Snowflake and Azure extra will result in non-importable WasbHook.

Rename all to devel_all extra

The all extras were reduced to include only user-facing dependencies. This means that this extra does not contain development dependencies. If you were relying on all extra then you should use now `devel_all` or figure out if you need development extras at all.

Context variables `prev_execution_date_success` and `prev_execution_date_success` are now `pendulum.DateTime`

Rename policy to task_policy

Because Airflow introduced DAG level policy (`dag_policy`) we decided to rename existing `policy` function to `task_policy` to make the distinction more profound and avoid any confusion.

Users using cluster policy need to rename their `policy` functions in `airflow_local_settings.py` to `task_policy`.

Default value for [celery] operation_timeout has changed to 1.0

From Airflow 2, by default Airflow will retry 3 times to publish task to Celery broker. This is controlled by `[celery] task_publish_max_retries`. Because of this we can now have a lower Operation timeout that raises `AirflowTaskTimeout`. This generally occurs during network blips or intermittent DNS issues.

Adding Operators and Sensors via plugins is no longer supported

Operators and Sensors should no longer be registered or imported via Airflow's plugin mechanism – these types of classes are just treated as plain python classes by Airflow, so there is no need to register them with Airflow.

If you previously had a `plugins/my_plugin.py` and you used it like this in a DAG:

```
from airflow.operators.my_plugin import MyOperator
```

You should instead import it as:

```
from my_plugin import MyOperator
```

The name under `airflow.operators.` was the plugin name, where as in the second example it is the python module name where the operator is defined.

See <https://airflow.apache.org/docs/apache-airflow/stable/howto/custom-operator.html> for more info.

Importing Hooks via plugins is no longer supported

Importing hooks added in plugins via `airflow.hooks.<plugin_name>` is no longer supported, and hooks should just be imported as regular python modules.

```
from airflow.hooks.my_plugin import MyHook
```

You should instead import it as:

```
from my_plugin import MyHook
```

It is still possible (but not required) to “register” hooks in plugins. This is to allow future support for dynamically populating the Connections form in the UI.

See <https://airflow.apache.org/docs/apache-airflow/stable/howto/custom-operator.html> for more info.

The default value for [core] enable_xcom_pickling has been changed to False

The pickle type for XCom messages has been replaced to JSON by default to prevent RCE attacks. Note that JSON serialization is stricter than pickling, so for example if you want to pass raw bytes through XCom you must encode them using an encoding like base64. If you understand the risk and still want to use `pickling`, set `enable_xcom_pickling = True` in your Airflow config's `core` section.

Airflowignore of base path

There was a bug fixed in <https://github.com/apache/airflow/pull/11993> that the “airflowignore” checked the base path of the dag folder for forbidden dags, not only the relative part. This had the effect that if the base path contained the excluded word the whole dag folder could have been excluded. For example if the airflowignore file contained `x`, and the dags folder was `'/var/x/dags'`, then all dags in the folder would be excluded. The fix only matches the relative path only now which means that if you previously used full path as ignored, you should change it to relative one. For

example if your dag folder was ‘/var/dags/’ and your airflowignore contained ‘/var/dag/excluded/’, you should change it to ‘excluded/’.

ExternalTaskSensor provides all task context variables to execution_date_fn as keyword arguments

The old syntax of passing `context` as a dictionary will continue to work with the caveat that the argument must be named `context`. The following will break. To fix it, change `ctx` to `context`.

```
def execution_date_fn(execution_date, ctx): ...
```

`execution_date_fn` can take in any number of keyword arguments available in the task context dictionary. The following forms of `execution_date_fn` are all supported:

```
def execution_date_fn(dt): ...
```

```
def execution_date_fn(execution_date): ...
```

```
def execution_date_fn(execution_date, ds_nodash): ...
```

```
def execution_date_fn(execution_date, ds_nodash, dag): ...
```

The default value for [webserver] cookie_samesite has been changed to Lax

As recommended by Flask, the `[webserver] cookie_samesite` has been changed to `Lax` from `''` (empty string).

Changes to import paths

Formerly the core code was maintained by the original creators - Airbnb. The code that was in the contrib package was supported by the community. The project was passed to the Apache community and currently the entire code is maintained by the community, so now the division has no justification, and it is only due to historical reasons. In Airflow 2.0, we want to organize packages and move integrations with third party services to the `airflow.providers` package.

All changes made are backward compatible, but if you use the old import paths you will see a deprecation warning. The old import paths can be abandoned in the future.

According to AIP-21 `_operator` suffix has been removed from operators. A deprecation warning has also been raised for paths importing with the suffix.

The following table shows changes in import paths.

| Old path | New path |
|---|--|
| airflow.hooks.base_hook.BaseHook | airflow.hooks.base.BaseHook |
| airflow.hooks.dbapi_hook.DbApiHook | airflow.hooks.dbapi.DbApiHook |
| airflow.operators.dummy_operator. DummyOperator | airflow.operators.dummy.DummyOperator |
| airflow.operators.dagrun_operator. TriggerDagRunOperator | airflow.operators.trigger_dagrun. TriggerDagRunOperator |
| airflow.operators.branch_operator. BaseBranchOperator | airflow.operators.branch. BaseBranchOperator |
| airflow.operators.subdag_operator. SubDagOperator | airflow.operators.subdag.SubDagOperator |
| airflow.sensors.base_sensor_operator. BaseSensorOperator | airflow.sensors.base.BaseSensorOperator |
| airflow.sensors.date_time_sensor. DateTimeSensor | airflow.sensors.date_time.DateTimeSensor |
| airflow.sensors.external_task_sensor. ExternalTaskMarker | airflow.sensors.external_task. ExternalTaskMarker |
| airflow.sensors.external_task_sensor. ExternalTaskSensor | airflow.sensors.external_task. ExternalTaskSensor |
| airflow.sensors.sql_sensor.SqlSensor | airflow.sensors.sql.SqlSensor |
| airflow.sensors.time_delta_sensor. TimeDeltaSensor | airflow.sensors.time_delta.TimeDeltaSensor |
| airflow.contrib.sensors.weekday_sensor. DayOfWeekSensor | airflow.sensors.weekday.DayOfWeekSensor |

Database schema changes

In order to migrate the database, you should use the command `airflow db upgrade`, but in some cases manual steps are required.

Unique conn_id in connection table

Previously, Airflow allowed users to add more than one connection with the same `conn_id` and on access it would choose one connection randomly. This acted as a basic load balancing and fault tolerance technique, when used in conjunction with retries.

This behavior caused some confusion for users, and there was no clear evidence if it actually worked well or not.

Now the `conn_id` will be unique. If you already have duplicates in your metadata database, you will have to manage those duplicate connections before upgrading the database.

Not-nullable conn_type column in connection table

The `conn_type` column in the `connection` table must contain content. Previously, this rule was enforced by application logic, but was not enforced by the database schema.

If you made any modifications to the table directly, make sure you don't have null in the `conn_type` column.

Configuration changes

This release contains many changes that require a change in the configuration of this application or other application that integrate with it.

This section describes the changes that have been made, and what you need to do to.

airflow.contrib.utils.log has been moved

Formerly the core code was maintained by the original creators - Airbnb. The code that was in the contrib package was supported by the community. The project was passed to the Apache community and currently the entire code is maintained by the community, so now the division has no justification, and it is only due to historical reasons. In Airflow 2.0, we want to organize packages and move integrations with third party services to the `airflow.providers` package.

To clean up, the following packages were moved:

| Old package | New package |
|---|--|
| <code>airflow.contrib.utils.log</code> | <code>airflow.utils.log</code> |
| <code>airflow.utils.log.gcs_task_handler</code> | <code>airflow.providers.google.cloud.log.gcs_task_handler</code> |
| <code>airflow.utils.log.wasb_task_handler</code> | <code>airflow.providers.microsoft.azure.log.wasb_task_handler</code> |
| <code>airflow.utils.log.stackdriver_task_handler</code> | <code>airflow.providers.google.cloud.log.stackdriver_task_handler</code> |
| <code>airflow.utils.log.s3_task_handler</code> | <code>airflow.providers.amazon.aws.log.s3_task_handler</code> |
| <code>airflow.utils.log.es_task_handler</code> | <code>airflow.providers.elasticsearch.log.es_task_handler</code> |
| <code>airflow.utils.log.cloudwatch_task_handler</code> | <code>airflow.providers.amazon.aws.log.cloudwatch_task_handler</code> |

You should update the import paths if you are setting log configurations with the `logging_config_class` option. The old import paths still works but can be abandoned.

SendGrid emailer has been moved

Formerly the core code was maintained by the original creators - Airbnb. The code that was in the contrib package was supported by the community. The project was passed to the Apache community and currently the entire code is maintained by the community, so now the division has no justification, and it is only due to historical reasons.

To clean up, the `send_email` function from the `airflow.contrib.utils.sendgrid` module has been moved.

If your configuration file looks like this:

```
[email]
email_backend = airflow.contrib.utils.sendgrid.send_email
```

It should look like this now:

```
[email]
email_backend = airflow.providers.sendgrid.utils.emailer.send_email
```

The old configuration still works but can be abandoned.

Unify hostname_callable option in core section

The previous option used a colon(:) to split the module from function. Now the dot(.) is used.

The change aims to unify the format of all options that refer to objects in the `airflow.cfg` file.

Custom executors is loaded using full import path

In previous versions of Airflow it was possible to use plugins to load custom executors. It is still possible, but the configuration has changed. Now you don't have to create a plugin to configure a custom executor, but you need to provide the full path to the module in the `executor` option in the `core` section. The purpose of this change is to simplify the plugin mechanism and make it easier to configure executor.

If your module was in the path `my_acme_company.executors.MyCustomExecutor` and the plugin was called `my_plugin` then your configuration looks like this

```
[core]
executor = my_plugin.MyCustomExecutor
```

And now it should look like this:

```
[core]
executor = my_acme_company.executors.MyCustomExecutor
```

The old configuration is still works but can be abandoned at any time.

Use `CustomSQLAInterface` instead of `SQLAInterface` for custom data models.

From Airflow 2.0, if you want to define your own Flask App Builder data models you need to use `CustomSQLAInterface` instead of `SQLAInterface`.

For Non-RBAC replace:

```
from flask_appbuilder.models.sqla.interface import SQLAInterface

datamodel = SQLAInterface(your_data_model)
```

with RBAC (in 1.10):

```
from airflow.www_rbac.utils import CustomSQLAInterface

datamodel = CustomSQLAInterface(your_data_model)
```

and in 2.0:

```
from airflow.www.utils import CustomSQLAInterface

datamodel = CustomSQLAInterface(your_data_model)
```

Drop plugin support for `stat_name_handler`

In previous version, you could use plugins mechanism to configure `stat_name_handler`. You should now use the `stat_name_handler` option in `[scheduler]` section to achieve the same effect.

If your plugin looked like this and was available through the `test_plugin` path:

```
def my_stat_name_handler(stat):
    return stat

class AirflowTestPlugin(AirflowPlugin):
```

(continues on next page)

(continued from previous page)

```
name = "test_plugin"
stat_name_handler = my_stat_name_handler
```

then your airflow.cfg file should look like this:

```
[scheduler]
stat_name_handler=test_plugin.my_stat_name_handler
```

This change is intended to simplify the statsd configuration.

Logging configuration has been moved to new section

The following configurations have been moved from [core] to the new [logging] section.

- base_log_folder
- remote_logging
- remote_log_conn_id
- remote_base_log_folder
- encrypt_s3_logs
- logging_level
- fab_logging_level
- logging_config_class
- colored_console_log
- colored_log_format
- colored_formatter_class
- log_format
- simple_log_format
- task_log_prefix_template
- log_filename_template
- log_processor_filename_template
- dag_processor_manager_log_location
- task_log_reader

Metrics configuration has been moved to new section

The following configurations have been moved from [scheduler] to the new [metrics] section.

- statsd_on
- statsd_host
- statsd_port
- statsd_prefix
- statsd_allow_list
- stat_name_handler

- statsd_datadog_enabled
- statsd_datadog_tags
- statsd_custom_client_path

Changes to Elasticsearch logging provider

When JSON output to stdout is enabled, log lines will now contain the `log_id` & `offset` fields, this should make reading task logs from elasticsearch on the webserver work out of the box. Example configuration:

```
[logging]
remote_logging = True
[elasticsearch]
host = http://es-host:9200
write_stdout = True
json_format = True
```

Note that the webserver expects the log line data itself to be present in the `message` field of the document.

Remove `gcp_service_account_keys` option in `airflow.cfg` file

This option has been removed because it is no longer supported by the Google Kubernetes Engine. The new recommended service account keys for the Google Cloud management method is [Workload Identity](#).

Fernet is enabled by default

The fernet mechanism is enabled by default to increase the security of the default installation. In order to restore the previous behavior, the user must consciously set an empty key in the `fernet_key` option of section `[core]` in the `airflow.cfg` file.

At the same time, this means that the `apache-airflow[crypto]` extra-packages are always installed. However, this requires that your operating system has `libffi-dev` installed.

Changes to propagating Kubernetes worker annotations

`kubernetes_annotations` configuration section has been removed. A new key `worker_annotations` has been added to existing `kubernetes` section instead. That is to remove restriction on the character set for k8s annotation keys. All key/value pairs from `kubernetes_annotations` should now go to `worker_annotations` as a json. I.e. instead of e.g.

```
[kubernetes_annotations]
annotation_key = annotation_value
annotation_key2 = annotation_value2
```

it should be rewritten to

```
[kubernetes]
worker_annotations = { "annotation_key" : "annotation_value", "annotation_key2" :
˓→"annotation_value2" }
```

Remove run_duration

We should not use the `run_duration` option anymore. This used to be for restarting the scheduler from time to time, but right now the scheduler is getting more stable and therefore using this setting is considered bad and might cause an inconsistent state.

Rename pool statsd metrics

Used slot has been renamed to running slot to make the name self-explanatory and the code more maintainable.

This means `pool.used_slots.<pool_name>` metric has been renamed to `pool.running_slots.<pool_name>`. The `Used Slots` column in Pools Web UI view has also been changed to `Running Slots`.

Removal of Mesos Executor

The Mesos Executor is removed from the code base as it was not widely used and not maintained. [Mailing List Discussion on deleting it](#).

Change dag loading duration metric name

Change DAG file loading duration metric from `dag.loading-duration.<dag_id>` to `dag.loading-duration.<dag_file>`. This is to better handle the case when a DAG file has multiple DAGs.

Sentry is disabled by default

Sentry is disabled by default. To enable these integrations, you need set `sentry_on` option in `[sentry]` section to "True".

Simplified GCSTaskHandler configuration

In previous versions, in order to configure the service account key file, you had to create a connection entry. In the current version, you can configure `google_key_path` option in `[logging]` section to set the key file path.

Users using Application Default Credentials (ADC) need not take any action.

The change aims to simplify the configuration of logging, to prevent corruption of the instance configuration by changing the value controlled by the user - connection entry. If you configure a backend secret, it also means the webserver doesn't need to connect to it. This simplifies setups with multiple GCP projects, because only one project will require the Secret Manager API to be enabled.

Changes to the core operators/hooks

We strive to ensure that there are no changes that may affect the end user and your files, but this release may contain changes that will require changes to your DAG files.

This section describes the changes that have been made, and what you need to do to update your DAG File, if you use core operators or any other.

BaseSensorOperator now respects the trigger_rule of downstream tasks

Previously, BaseSensorOperator with setting `soft_fail=True` skips itself and skips all its downstream tasks unconditionally, when it fails i.e the `trigger_rule` of downstream tasks is not respected.

In the new behavior, the `trigger_rule` of downstream tasks is respected. User can preserve/achieve the original behaviour by setting the `trigger_rule` of each downstream task to `all_success`.

BaseOperator uses metaclass

BaseOperator class uses a BaseOperatorMeta as a metaclass. This meta class is based on abc.ABCMeta. If your custom operator uses different metaclass then you will have to adjust it.

Remove SQL support in BaseHook

Remove get_records and get_pandas_df and run from BaseHook, which only apply for SQL-like hook, If want to use them, or your custom hook inherit them, please use airflow.hooks.dbapi.DbApiHook

Assigning task to a DAG using bitwise shift (bit-shift) operators are no longer supported

Previously, you could assign a task to a DAG as follows:

```
dag = DAG("my_dag")
dummy = DummyOperator(task_id="dummy")

dag >> dummy
```

This is no longer supported. Instead, we recommend using the DAG as context manager:

```
with DAG("my_dag") as dag:
    dummy = DummyOperator(task_id="dummy")
```

Removed deprecated import mechanism

The deprecated import mechanism has been removed so the import of modules becomes more consistent and explicit.

For example: `from airflow.operators import BashOperator` becomes `from airflow.operators.bash_operator import BashOperator`

Changes to sensor imports

Sensors are now accessible via `airflow.sensors` and no longer via `airflow.operators.sensors`.

For example: `from airflow.operators.sensors import BaseSensorOperator` becomes `from airflow.sensors.base import BaseSensorOperator`

Skipped tasks can satisfy wait_for_downstream

Previously, a task instance with `wait_for_downstream=True` will only run if the downstream task of the previous task instance is successful. Meanwhile, a task instance with `depends_on_past=True` will run if the previous task instance is either successful or skipped. These two flags are close siblings yet they have different behavior. This inconsistency in behavior made the API less intuitive to users. To maintain consistent behavior, both successful or skipped downstream task can now satisfy the `wait_for_downstream=True` flag.

`airflow.utils.helpers.cross_downstream`

`airflow.utils.helpers.chain`

The `chain` and `cross_downstream` methods are now moved to `airflow.models.baseoperator` module from `airflow.utils.helpers` module.

The `baseoperator` module seems to be a better choice to keep closely coupled methods together. Helpers module is supposed to contain standalone helper methods that can be imported by all classes.

The `chain` method and `cross_downstream` method both use `BaseOperator`. If any other package imports any classes or functions from `helpers` module, then it automatically has an implicit dependency to `BaseOperator`. That can often lead to cyclic dependencies.

More information in [AIRFLOW-6392](#)

In Airflow < 2.0 you imported those two methods like this:

```
from airflow.utils.helpers import chain
from airflow.utils.helpers import cross_downstream
```

In Airflow 2.0 it should be changed to:

```
from airflow.models.baseoperator import chain
from airflow.models.baseoperator import cross_downstream
```

`airflow.operators.python.BranchPythonOperator`

`BranchPythonOperator` will now return a value equal to the `task_id` of the chosen branch, where previously it returned `None`. Since it inherits from `BaseOperator` it will do an `xcom_push` of this value if `do_xcom_push=True`. This is useful for downstream decision-making.

`airflow.sensors.sql_sensor.SqlSensor`

`SQLSensor` now consistent with python `bool()` function and the `allow_null` parameter has been removed.

It will resolve after receiving any value that is casted to `True` with python `bool(value)`. That changes the previous response receiving `NULL` or '`0`'. Earlier '`0`' has been treated as success criteria. `NULL` has been treated depending on value of `allow_null` parameter. But all the previous behaviour is still achievable setting param `success` to `lambda x: x is None or str(x) not in ('0', '')`.

`airflow.operators.trigger_dagrun.TriggerDagRunOperator`

The `TriggerDagRunOperator` now takes a `conf` argument to which a dict can be provided as `conf` for the `DagRun`. As a result, the `python_callable` argument was removed. PR: <https://github.com/apache/airflow/pull/6317>.

`airflow.operators.python.PythonOperator`

`provide_context` argument on the `PythonOperator` was removed. The signature of the callable passed to the `PythonOperator` is now inferred and argument values are always automatically provided. There is no need to explicitly provide or not provide the context anymore. For example:

```
def myfunc(execution_date):
    print(execution_date)

python_operator = PythonOperator(task_id="mytask", python_callable=myfunc, dag=dag)
```

Notice you don't have to set `provide_context=True`, variables from the task context are now automatically detected and provided.

All context variables can still be provided with a double-asterisk argument:

```
def myfunc(**context):
    print(context) # all variables will be provided to context
```

(continues on next page)

(continued from previous page)

```
python_operator = PythonOperator(task_id="mytask", python_callable=myfunc)
```

The task context variable names are reserved names in the callable function, hence a clash with `op_args` and `op_kwargs` results in an exception:

```
def myfunc(dag):
    # raises a ValueError because "dag" is a reserved name
    # valid signature example: myfunc(mydag)
    print("output")

python_operator = PythonOperator(
    task_id="mytask",
    op_args=[1],
    python_callable=myfunc,
)
```

The change is backwards compatible, setting `provide_context` will add the `provide_context` variable to the `kwargs` (but won't do anything).

PR: #5990

`airflow.providers.standard.sensors.filesystem.FileSensor`

`FileSensor` is now takes a glob pattern, not just a filename. If the filename you are looking for has *, ?, or [in it then you should replace these with [*], [?], and [[].

`airflow.operators.subdag_operator.SubDagOperator`

`SubDagOperator` is changed to use Airflow scheduler instead of backfill to schedule tasks in the subdag. User no longer need to specify the executor in `SubDagOperator`.

```
airflow.providers.google.cloud.operators.datastore.CloudDatastoreExportEntitiesOperator
airflow.providers.google.cloud.operators.datastore.CloudDatastoreImportEntitiesOperator
airflow.providers.cncf.kubernetes.operators.kubernetes_pod.KubernetesPodOperator
airflow.providers.ssh.operators.ssh.SSHOperator
airflow.providers.microsoft.winrm.operators.winrm.WinRMOperator
airflow.operators.bash.BashOperator
airflow.providers.docker.operators.docker.DockerOperator
airflow.providers.http.operators.http.SimpleHttpOperator
```

The `do_xcom_push` flag (a switch to push the result of an operator to xcom or not) was appearing in different incarnations in different operators. It's function has been unified under a common name (`do_xcom_push`) on `BaseOperator`. This way it is also easy to globally disable pushing results to xcom.

The following operators were affected:

- `DatastoreExportOperator` (Backwards compatible)

- DatastoreImportOperator (Backwards compatible)
- KubernetesPodOperator (Not backwards compatible)
- SSHOperator (Not backwards compatible)
- WinRMOperator (Not backwards compatible)
- BashOperator (Not backwards compatible)
- DockerOperator (Not backwards compatible)
- SimpleHttpOperator (Not backwards compatible)

See [AIRFLOW-3249](#) for details

`airflow.operators.latest_only_operator.LatestOnlyOperator`

In previous versions, the `LatestOnlyOperator` forcefully skipped all (direct and indirect) downstream tasks on its own. From this version on the operator will **only skip direct downstream** tasks and the scheduler will handle skipping any further downstream dependencies.

No change is needed if only the default trigger rule `all_success` is being used.

If the DAG relies on tasks with other trigger rules (i.e. `all_done`) being skipped by the `LatestOnlyOperator`, adjustments to the DAG need to be made to accommodate the change in behaviour, i.e. with additional edges from the `LatestOnlyOperator`.

The goal of this change is to achieve a more consistent and configurable cascading behaviour based on the `BaseBranchOperator` (see [AIRFLOW-2923](#) and [AIRFLOW-1784](#)).

Changes to the core Python API

We strive to ensure that there are no changes that may affect the end user, and your Python files, but this release may contain changes that will require changes to your plugins, DAG File or other integration.

Only changes unique to this provider are described here. You should still pay attention to the changes that have been made to the core (including core operators) as they can affect the integration behavior of this provider.

This section describes the changes that have been made, and what you need to do to update your Python files.

Removed sub-package imports from `airflow/__init__.py`

The imports `LoggingMixin`, `conf`, and `AirflowException` have been removed from `airflow/__init__.py`. All implicit references of these objects will no longer be valid. To migrate, all usages of each old path must be replaced with its corresponding new path.

| Old Path (Implicit Import) | New Path (Explicit Import) |
|---------------------------------------|---|
| <code>airflow.LoggingMixin</code> | <code>airflow.utils.log.logging_mixin.LoggingMixin</code> |
| <code>airflow.conf</code> | <code>airflow.configuration.conf</code> |
| <code>airflow.AirflowException</code> | <code>airflow.exceptions.AirflowException</code> |

Variables removed from the task instance context

The following variables were removed from the task instance context:

- `end_date`
- `latest_date`

- tables

`airflow.contrib.utils.Weekday`

Formerly the core code was maintained by the original creators - Airbnb. The code that was in the contrib package was supported by the community. The project was passed to the Apache community and currently the entire code is maintained by the community, so now the division has no justification, and it is only due to historical reasons.

To clean up, Weekday enum has been moved from `airflow.contrib.utils` into `airflow.utils` module.

`airflow.models.connection.Connection`

The connection module has new deprecated methods:

- `Connection.parse_from_uri`
- `Connection.log_info`
- `Connection.debug_info`

and one deprecated function:

- `parse_netloc_to_hostname`

Previously, users could create a connection object in two ways

```
conn_1 = Connection(conn_id="conn_a", uri="mysql://AAA/")
# or
conn_2 = Connection(conn_id="conn_a")
conn_2.parse_uri(uri="mysql://AAA/")
```

Now the second way is not supported.

`Connection.log_info` and `Connection.debug_info` method have been deprecated. Read each Connection field individually or use the default representation (`__repr__`).

The old method is still works but can be abandoned at any time. The changes are intended to delete method that are rarely used.

`airflow.models.dag.DAG.create_dagrun`

`DAG.create_dagrun` accepts `run_type` and does not require `run_id`. This change is caused by adding `run_type` column to `DagRun`.

Previous signature:

```
def create_dagrun(
    self,
    run_id,
    state,
    execution_date=None,
    start_date=None,
    external_trigger=False,
    conf=None,
    session=None,
): ...
```

current:

```
def create_dagrun(
    self,
    state,
    execution_date=None,
    run_id=None,
    start_date=None,
    external_trigger=False,
    conf=None,
    run_type=None,
    session=None,
): ...
```

If user provides `run_id` then the `run_type` will be derived from it by checking prefix, allowed types : `manual`, `scheduled`, `backfill` (defined by `airflow.utils.types.DagRunType`).

If user provides `run_type` and `execution_date` then `run_id` is constructed as `{run_type}__{execution_date.isoformat()}`.

Airflow should construct dagruns using `run_type` and `execution_date`, creation using `run_id` is preserved for user actions.

`airflow.models.dagrun.DagRun`

Use `DagRunType.SCHEDULED.value` instead of `DagRun.ID_PREFIX`

All the `run_id` prefixes for different kind of DagRuns have been grouped into a single enum in `airflow.utils.types.DagRunType`.

Previously, there were defined in various places, example as `ID_PREFIX` class variables for `DagRun`, `BackfillJob` and in `_trigger_dag` function.

Was:

```
>> from airflow.models.dagrun import DagRun
>> DagRun.ID_PREFIX
scheduled__
```

Replaced by:

```
>> from airflow.utils.types import DagRunType
>> DagRunType.SCHEDULED.value
scheduled
```

`airflow.utils.file.TemporaryDirectory`

We remove `airflow.utils.file.TemporaryDirectory` Since Airflow dropped support for Python < 3.5 there's no need to have this custom implementation of `TemporaryDirectory` because the same functionality is provided by `tempfile.TemporaryDirectory`.

Now users instead of `import from airflow.utils.files import TemporaryDirectory` should do `from tempfile import TemporaryDirectory`. Both context managers provide the same interface, thus no additional changes should be required.

airflow.AirflowMacroPlugin

We removed `airflow.AirflowMacroPlugin` class. The class was there in airflow package but it has not been used (apparently since 2015). It has been removed.

airflow.settings.CONTEXT_MANAGER_DAG

`CONTEXT_MANAGER_DAG` was removed from settings. Its role has been taken by `DagContext` in ‘`airflow.models.dag`’. One of the reasons was that settings should be rather static than store dynamic context from the DAG, but the main one is that moving the context out of settings allowed to untangle cyclic imports between DAG, `BaseOperator`, `SerializedDAG`, `SerializedBaseOperator` which was part of AIRFLOW-6010.

airflow.utils.log.logging_mixin.redirect_stderr

airflow.utils.log.logging_mixin.redirect_stdout

Function `redirect_stderr` and `redirect_stdout` from `airflow.utils.log.logging_mixin` module has been deleted because it can be easily replaced by the standard library. The functions of the standard library are more flexible and can be used in larger cases.

The code below

```
import logging

from airflow.utils.log.logging_mixin import redirect_stderr, redirect_stdout

logger = logging.getLogger("custom-logger")
with redirect_stdout(logger, logging.INFO), redirect_stderr(logger, logging.WARN):
    print("I love Airflow")
```

can be replaced by the following code:

```
from contextlib import redirect_stdout, redirect_stderr
import logging

from airflow.utils.log.logging_mixin import StreamLogWriter

logger = logging.getLogger("custom-logger")

with (
    redirect_stdout(StreamLogWriter(logger, logging.INFO)),
    redirect_stderr(StreamLogWriter(logger, logging.WARN)),
):
    print("I Love Airflow")
```

airflow.models.baseoperator.BaseOperator

Now, additional arguments passed to `BaseOperator` cause an exception. Previous versions of Airflow took additional arguments and displayed a message on the console. When the message was not noticed by users, it caused very difficult to detect errors.

In order to restore the previous behavior, you must set an `True` in the `allow_illegal_arguments` option of section `[operators]` in the `airflow.cfg` file. In the future it is possible to completely delete this option.

`airflow.models.dagbag.DagBag`

Passing `store_serialized_dags` argument to `DagBag.init` and accessing `DagBag.store_serialized_dags` property are deprecated and will be removed in future versions.

Previous signature:

```
def __init__(
    dag_folder=None,
    include_examples=conf.getboolean("core", "LOAD_EXAMPLES"),
    safe_mode=conf.getboolean("core", "DAG_DISCOVERY_SAFE_MODE"),
    store_serialized_dags=False,
): ...
```

current:

```
def __init__(
    dag_folder=None,
    include_examples=conf.getboolean("core", "LOAD_EXAMPLES"),
    safe_mode=conf.getboolean("core", "DAG_DISCOVERY_SAFE_MODE"),
    read_dags_from_db=False,
): ...
```

If you were using positional arguments, it requires no change but if you were using keyword arguments, please change `store_serialized_dags` to `read_dags_from_db`.

Similarly, if you were using `DagBag().store_serialized_dags` property, change it to `DagBag().read_dags_from_db`.

Changes in google provider package

We strive to ensure that there are no changes that may affect the end user and your Python files, but this release may contain changes that will require changes to your configuration, DAG Files or other integration e.g. custom operators.

Only changes unique to this provider are described here. You should still pay attention to the changes that have been made to the core (including core operators) as they can affect the integration behavior of this provider.

This section describes the changes that have been made, and what you need to do to update your if you use operators or hooks which integrate with Google services (including Google Cloud - GCP).

Direct impersonation added to operators communicating with Google services

Directly impersonating a service account has been made possible for operators communicating with Google services via new argument called `impersonation_chain` (`google_impersonation_chain` in case of operators that also communicate with services of other cloud providers). As a result, `GCSToS3Operator` no longer derivatives from `GCSListObjectsOperator`.

Normalize `gcp_conn_id` for Google Cloud

Previously not all hooks and operators related to Google Cloud use `gcp_conn_id` as parameter for GCP connection. There is currently one parameter which apply to most services. Parameters like `datastore_conn_id`, `bq_conn_id`, `google_cloud_storage_conn_id` and similar have been deprecated. Operators that require two connections are not changed.

Following components were affected by normalization:

- `airflow.providers.google.cloud.hooks.datastore.DatastoreHook`

- airflow.providers.google.cloud.hooks.bigquery.BigQueryHook
- airflow.providers.google.cloud.hooks.gcs.GoogleCloudStorageHook
- airflow.providers.google.cloud.operators.bigquery.BigQueryCheckOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryValueCheckOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryIntervalCheckOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryGetDataOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryDeleteDatasetOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryCreateEmptyDatasetOperator
- airflow.providers.google.cloud.operators.bigquery.BigQueryTableDeleteOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageCreateBucketOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageListOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageDownloadOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageDeleteOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageBucketCreateAclEntryOperator
- airflow.providers.google.cloud.operators.gcs.GoogleCloudStorageObjectCreateAclEntryOperator
- airflow.operators.sql_to_gcs.BaseSQLToGoogleCloudStorageOperator
- airflow.operators.adls_to_gcs.AdlsToGoogleCloudStorageOperator
- airflow.operators.gcs_to_s3.GoogleCloudStorageToS3Operator
- airflow.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator
- airflow.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator
- airflow.operators.local_to_gcs.FileToGoogleCloudStorageOperator
- airflow.operators.cassandra_to_gcs.CassandraToGoogleCloudStorageOperator
- airflow.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator

Changes to import paths and names of GCP operators and hooks

According to [AIP-21](#) operators related to Google Cloud has been moved from contrib to core. The following table shows changes in import paths.

| Old path | New path |
|---|----------|
| airflow.contrib.hooks.bigquery_hook.BigQueryHook | |
| airflow.contrib.hooks.datastore_hook.DatastoreHook | |
| airflow.contrib.hooks.gcp_bigtable_hook.BigtableHook | |
| airflow.contrib.hooks.gcp_cloud_build_hook.CloudBuildHook | |
| airflow.contrib.hooks.gcp_container_hook.GKEClusterHook | |
| airflow.contrib.hooks.gcp_compute_hook.GceHook | |
| airflow.contrib.hooks.gcp_dataflow_hook.DataFlowHook | |
| airflow.contrib.hooks.gcp_dataproc_hook.DataProcHook | |
| airflow.contrib.hooks.gcp_dlp_hook.CloudDLPHook | |
| airflow.contrib.hooks.gcp_function_hook.GcfHook | |

continues on next page

Table 106 – continued from previous page

| Old path | New path |
|---|----------|
| airflow.contrib.hooks.gcp_kms_hook.GoogleCloudKMSHook | |
| airflow.contrib.hooks.gcp_mlengine_hook.MLEngineHook | |
| airflow.contrib.hooks.gcp_natural_language_hook.CloudNaturalLanguageHook | |
| airflow.contrib.hooks.gcp_pubsub_hook.PubSubHook | |
| airflow.contrib.hooks.gcp_speech_to_text_hook.GCPSpeechToTextHook | |
| airflow.contrib.hooks.gcp_spanner_hook.CloudSpannerHook | |
| airflow.contrib.hooks.gcp_sql_hook.CloudSqlDatabaseHook | |
| airflow.contrib.hooks.gcp_sql_hook.CloudSqlHook | |
| airflow.contrib.hooks.gcp_tasks_hook.CloudTasksHook | |
| airflow.contrib.hooks.gcp_text_to_speech_hook.GCPTextToSpeechHook | |
| airflow.contrib.hooks.gcp_transfer_hook.GCPTransferServiceHook | |
| airflow.contrib.hooks.gcp_translate_hook.CloudTranslateHook | |
| airflow.contrib.hooks.gcp_video_intelligence_hook.CloudVideoIntelligenceHook | |
| airflow.contrib.hooks.gcp_vision_hook.CloudVisionHook | |
| airflow.contrib.hooks.gcs_hook.GoogleCloudStorageHook | |
| airflow.contrib.operators.adls_to_gcs.AdlsToGoogleCloudStorageOperator | |
| airflow.contrib.operators.bigquery_check_operator.BigQueryCheckOperator | |
| airflow.contrib.operators.bigquery_check_operator.BigQueryIntervalCheckOperator | |
| airflow.contrib.operators.bigquery_check_operator.BigQueryValueCheckOperator | |
| airflow.contrib.operators.bigquery_get_data.BigQueryGetDataOperator | |
| airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyDatasetOperator | |
| airflow.contrib.operators.bigquery_operator.BigQueryCreateEmptyTableOperator | |
| airflow.contrib.operators.bigquery_operator.BigQueryCreateExternalTableOperator | |
| airflow.contrib.operators.bigquery_operator.BigQueryDeleteDatasetOperator | |
| airflow.contrib.operators.bigquery_operator.BigQueryOperator | |
| airflow.contrib.operators.bigquery_table_delete_operator.BigQueryTableDeleteOperator | |
| airflow.contrib.operators.bigquery_to_bigquery.BigQueryToBigQueryOperator | |
| airflow.contrib.operators.bigquery_to_gcs.BigQueryToCloudStorageOperator | |
| airflow.contrib.operators.bigquery_to_mysql_operator.BigQueryToMySqlOperator | |
| airflow.contrib.operators.dataflow_operator.DataFlowJavaOperator | |
| airflow.contrib.operators.dataflow_operator.DataFlowPythonOperator | |
| airflow.contrib.operators.dataflow_operator.DataflowTemplateOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcHadoopOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcHiveOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcJobBaseOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcPigOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcPySparkOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcSparkOperator | |
| airflow.contrib.operators.dataproc_operator.DataProcSparkSqlOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocClusterCreateOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocClusterDeleteOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocClusterScaleOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocOperationBaseOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateInlineOperator | |
| airflow.contrib.operators.dataproc_operator.DataprocWorkflowTemplateInstantiateOperator | |
| airflow.contrib.operators.datastore_export_operator.DatastoreExportOperator | |
| airflow.contrib.operators.datastore_import_operator.DatastoreImportOperator | |
| airflow.contrib.operators.file_to_gcs.FileToGoogleCloudStorageOperator | |
| airflow.contrib.operators.gcp_bigtable_operator.BigtableClusterUpdateOperator | |
| airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceCreateOperator | |

continues on next page

Table 106 – continued from previous page

| Old path | New path |
|--|----------|
| airflow.contrib.operators.gcp_bigtable_operator.BigtableInstanceDeleteOperator | |
| airflow.contrib.operators.gcp_bigtable_operator.BigtableTableCreateOperator | |
| airflow.contrib.operators.gcp_bigtable_operator.BigtableTableDeleteOperator | |
| airflow.contrib.operators.gcp_bigtable_operator.BigtableTableWaitForReplicationSensor | |
| airflow.contrib.operators.gcp_cloud_build_operator.CloudBuildCreateBuildOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceBaseOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceInstanceGroupManagerUpdateTemplateOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceInstanceStartOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceInstanceStopOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceInstanceTemplateCopyOperator | |
| airflow.contrib.operators.gcp_compute_operator.GceSetMachineTypeOperator | |
| airflow.contrib.operators.gcp_container_operator.GKEClusterCreateOperator | |
| airflow.contrib.operators.gcp_container_operator.GKEClusterDeleteOperator | |
| airflow.contrib.operators.gcp_container_operator.GKEPodOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCancelDLPJobOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCreateDLPJobOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCreateDeidentifyTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCreateInspectTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCreateJobTriggerOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPCreateStoredInfoTypeOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeidentifyContentOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeleteDeidentifyTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeleteDlpJobOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeleteInspectTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeleteJobTriggerOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPDeleteStoredInfoTypeOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPGetDeidentifyTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPGetDlpJobOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPGetInspectTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPGetJobTripperOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPGetStoredInfoTypeOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPInspectContentOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListDeidentifyTemplatesOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListDlpJobsOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListInfoTypesOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListInspectTemplatesOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListJobTriggersOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPListStoredInfoTypesOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPRedactImageOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPReidentifyContentOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPUpdateDeidentifyTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPUpdateInspectTemplateOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPUpdateJobTriggerOperator | |
| airflow.contrib.operators.gcp_dlp_operator.CloudDLPUpdateStoredInfoTypeOperator | |
| airflow.contrib.operators.gcp_function_operator.GcfFunctionDeleteOperator | |
| airflow.contrib.operators.gcp_function_operator.GcfFunctionDeployOperator | |
| airflow.contrib.operators.gcp_natural_language_operator.CloudNaturalLanguageAnalyzeEntitiesOperator | |
| airflow.contrib.operators.gcp_natural_language_operator.CloudNaturalLanguageAnalyzeEntitySentimentOperator | |
| airflow.contrib.operators.gcp_natural_language_operator.CloudNaturalLanguageAnalyzeSentimentOperator | |
| airflow.contrib.operators.gcp_natural_language_operator.CloudNaturalLanguageClassifyTextOperator | |

continues on next page

Table 106 – continued from previous page

| Old path | New path |
|--|----------|
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeleteOperator | |
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseDeployOperator | |
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseQueryOperator | |
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDatabaseUpdateOperator | |
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeleteOperator | |
| airflow.contrib.operators.gcp_spanner_operator.CloudSpannerInstanceDeployOperator | |
| airflow.contrib.operators.gcp_speech_to_text_operator.GcpSpeechToTextRecognizeSpeechOperator | |
| airflow.contrib.operators.gcp_text_to_speech_operator.GcpTextToSpeechSynthesizeOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobCreateOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobDeleteOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceJobUpdateOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationCancelOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationGetOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationPauseOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationResumeOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GcpTransferServiceOperationsListOperator | |
| airflow.contrib.operators.gcp_transfer_operator.GoogleCloudStorageToGoogleCloudStorageTransferOperator | |
| airflow.contrib.operators.gcp_translate_operator.CloudTranslateTextOperator | |
| airflow.contrib.operators.gcp_translate_speech_operator.GcpTranslateSpeechOperator | |
| airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoExplicitContentOperator | |
| airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoLabelsOperator | |
| airflow.contrib.operators.gcp_video_intelligence_operator.CloudVideoIntelligenceDetectVideoShotsOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionAddProductToProductSetOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionAnnotateImageOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectDocumentTextOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImageLabelsOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectImageSafeSearchOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionDetectTextOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductCreateOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductDeleteOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductGetOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetCreateOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetDeleteOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetGetOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductSetUpdateOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionProductUpdateOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionReferenceImageCreateOperator | |
| airflow.contrib.operators.gcp_vision_operator.CloudVisionRemoveProductFromProductSetOperator | |
| airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageBucketCreateAclEntryOperator | |
| airflow.contrib.operators.gcs_acl_operator.GoogleCloudStorageObjectCreateAclEntryOperator | |
| airflow.contrib.operators.gcs_delete_operator.GoogleCloudStorageDeleteOperator | |
| airflow.contrib.operators.gcs_download_operator.GoogleCloudStorageDownloadOperator | |
| airflow.contrib.operators.gcs_list_operator.GoogleCloudStorageListOperator | |
| airflow.contrib.operators.gcs_operator.GoogleCloudStorageCreateBucketOperator | |
| airflow.contrib.operators.gcs_to_bq.GoogleCloudStorageToBigQueryOperator | |
| airflow.contrib.operators.gcs_to_gcs.GoogleCloudStorageToGoogleCloudStorageOperator | |
| airflow.contrib.operators.gcs_to_s3.GoogleCloudStorageToS3Operator | |
| airflow.contrib.operators.mlengine_operator.MLEngineBatchPredictionOperator | |
| airflow.contrib.operators.mlengine_operator.MLEngineModelOperator | |
| airflow.contrib.operators.mlengine_operator.MLEngineTrainingOperator | |

continues on next page

Table 106 – continued from previous page

| Old path | New path |
|---|----------|
| airflow.contrib.operators.mlengine_operator.MLEngineVersionOperator | |
| airflow.contrib.operators.mssql_to_gcs.MsSqlToGoogleCloudStorageOperator | |
| airflow.contrib.operators.mysql_to_gcs.MySqlToGoogleCloudStorageOperator | |
| airflow.contrib.operators.postgres_to_gcs_operator.PostgresToGoogleCloudStorageOperator | |
| airflow.contrib.operators.pubsub_operator.PubSubPublishOperator | |
| airflow.contrib.operators.pubsub_operator.PubSubSubscriptionCreateOperator | |
| airflow.contrib.operators.pubsub_operator.PubSubSubscriptionDeleteOperator | |
| airflow.contrib.operators.pubsub_operator.PubSubTopicCreateOperator | |
| airflow.contrib.operators.pubsub_operator.PubSubTopicDeleteOperator | |
| airflow.contrib.operators.sql_to_gcs.BaseSQLToGoogleCloudStorageOperator | |
| airflow.contrib.sensors.bigquery_sensor.BigQueryTableSensor | |
| airflow.contrib.sensors.gcp_transfer_sensor.GCPTransferServiceWaitForJobStatusSensor | |
| airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectSensor | |
| airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageObjectUpdatedSensor | |
| airflow.contrib.sensors.gcs_sensor.GoogleCloudStoragePrefixSensor | |
| airflow.contrib.sensors.gcs_sensor.GoogleCloudStorageUploadSessionCompleteSensor | |
| airflow.contrib.sensors.pubsub_sensor.PubSubPullSensor | |

Unify default conn_id for Google Cloud

Previously not all hooks and operators related to Google Cloud use `google_cloud_default` as a default `conn_id`. There is currently one default variant. Values like `google_cloud_storage_default`, `bq_default`, `google_cloud_datastore_default` have been deprecated. The configuration of existing relevant connections in the database have been preserved. To use those deprecated GCP `conn_id`, you need to explicitly pass their `conn_id` into operators/hooks. Otherwise, `google_cloud_default` will be used as GCP's `conn_id` by default.

```
airflow.providers.google.cloud.hooks.dataflow.DataflowHook  
airflow.providers.google.cloud.operators.dataflow.DataflowCreateJavaJobOperator  
airflow.providers.google.cloud.operators.dataflow.DataflowTemplatedJobStartOperator  
airflow.providers.google.cloud.operators.dataflow.DataflowCreatePythonJobOperator
```

To use `project_id` argument consistently across GCP hooks and operators, we did the following changes:

- **Changed order of arguments in DataflowHook.start_python_dataflow. Uses**
with positional arguments may break.
- **Changed order of arguments in DataflowHook.is_job_dataflow_running. Uses**
with positional arguments may break.
- **Changed order of arguments in DataflowHook.cancel_job. Uses**
with positional arguments may break.
- **Added optional project_id argument to DataflowCreateJavaJobOperator**
constructor.
- **Added optional project_id argument to DataflowTemplatedJobStartOperator**
constructor.
- **Added optional project_id argument to DataflowCreatePythonJobOperator**
constructor.

`airflow.providers.google.cloud.sensors.gcs.GCSUploadSessionCompleteSensor`

To provide more precise control in handling of changes to objects in underlying GCS Bucket the constructor of this sensor now has changed.

- Old Behavior: This constructor used to optionally take `previous_num_objects: int`.
- New replacement constructor kwarg: `previous_objects: Optional[Set[str]]`.

Most users would not specify this argument because the bucket begins empty and the user wants to treat any files as new.

Example of Updating usage of this sensor: Users who used to call:

```
GCSUploadSessionCompleteSensor(bucket='my_bucket', prefix='my_prefix',  
previous_num_objects=1)
```

Will now call:

```
GCSUploadSessionCompleteSensor(bucket='my_bucket', prefix='my_prefix',  
previous_num_objects={'.keep'})
```

Where ‘.keep’ is a single file at your prefix that the sensor should not consider new.

`airflow.providers.google.cloud.hooks.bigquery.BigQueryBaseCursor`

`airflow.providers.google.cloud.hooks.bigquery.BigQueryHook`

To simplify BigQuery operators (no need of `Cursor`) and standardize usage of hooks within all GCP integration methods from `BigQueryBaseCursor` were moved to `BigQueryHook`. Using them by from `Cursor` object is still possible due to preserved backward compatibility but they will raise `DeprecationWarning`. The following methods were moved:

| Old path | New path |
|--|--|
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.cancel_query | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.cancel_query |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. create_empty_dataset | airflow.providers.google.cloud. hooks.bigquery.BigQueryHook. create_empty_dataset |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. create_empty_table | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.create_empty_table |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. create_external_table | airflow.providers.google.cloud. hooks.bigquery.BigQueryHook. create_external_table |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.delete_dataset | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.delete_dataset |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.get_dataset | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.get_dataset |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. get_dataset_tables | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.get_dataset_tables |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. get_dataset_tables_list | airflow.providers.google.cloud. hooks.bigquery.BigQueryHook. get_dataset_tables_list |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. get_datasets_list | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.get_datasets_list |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.get_schema | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.get_schema |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.get_tabledata | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.get_tabledata |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.insert_all | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.insert_all |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.patch_dataset | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.patch_dataset |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.patch_table | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.patch_table |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. poll_job_complete | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.poll_job_complete |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.run_copy | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_copy |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.run_extract | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_extract |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. run_grant_dataset_view_access | airflow.providers.google.cloud. hooks.bigquery.BigQueryHook. run_grant_dataset_view_access |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.run_load | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_load |
| airflow.providers.google.cloud.hooks. bigquery.BigQueryBaseCursor.run_query | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_query |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. run_table_delete | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_table_delete |
| airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. run_table_upsert | airflow.providers.google.cloud.hooks. bigquery.BigQueryHook.run_table_upsert |
| 720 airflow.providers.google.cloud. hooks.bigquery.BigQueryBaseCursor. run_with_configuration | Chapter 3. Why not Airflow®? airflow.providers.google.cloud. hooks.bigquery.BigQueryHook. run_with_configuration |

`airflow.providers.google.cloud.hooks.bigquery.BigQueryBaseCursor`

Since BigQuery is the part of the GCP it was possible to simplify the code by handling the exceptions by usage of the `airflow.providers.google.common.hooks.base.GoogleBaseHook.catch_http_exception` decorator however it changes exceptions raised by the following methods:

- `airflow.providers.google.cloud.hooks.bigquery.BigQueryBaseCursor.run_table_delete` raises `AirflowException` instead of `Exception`.
- `airflow.providers.google.cloud.hooks.bigquery.BigQueryBaseCursor.create_empty_dataset` raises `AirflowException` instead of `ValueError`.
- `airflow.providers.google.cloud.hooks.bigquery.BigQueryBaseCursor.get_dataset` raises `AirflowException` instead of `ValueError`.

`airflow.providers.google.cloud.operators.bigquery.BigQueryCreateEmptyTableOperator`

`airflow.providers.google.cloud.operators.bigquery.BigQueryCreateEmptyDatasetOperator`

Idempotency was added to `BigQueryCreateEmptyTableOperator` and `BigQueryCreateEmptyDatasetOperator`. But to achieve that try / except clause was removed from `create_empty_dataset` and `create_empty_table` methods of `BigQueryHook`.

`airflow.providers.google.cloud.hooks.dataflow.DataflowHook`

`airflow.providers.google.cloud.hooks.mlengine.MLEngineHook`

`airflow.providers.google.cloud.hooks.pubsub.PubSubHook`

The change in GCP operators implies that GCP Hooks for those operators require now keyword parameters rather than positional ones in all methods where `project_id` is used. The methods throw an explanatory exception in case they are called using positional parameters.

Other GCP hooks are unaffected.

`airflow.providers.google.cloud.hooks.pubsub.PubSubHook`

`airflow.providers.google.cloud.operators.pubsub.PubSubTopicCreateOperator`

`airflow.providers.google.cloud.operators.pubsub.PubSubSubscriptionCreateOperator`

`airflow.providers.google.cloud.operators.pubsub.PubSubTopicDeleteOperator`

`airflow.providers.google.cloud.operators.pubsub.PubSubSubscriptionDeleteOperator`

`airflow.providers.google.cloud.operators.pubsub.PubSubPublishOperator`

`airflow.providers.google.cloud.sensors.pubsub.PubSubPullSensor`

In the `PubSubPublishOperator` and `PubSubHook.publish` method the `data` field in a message should be bytestring (utf-8 encoded) rather than base64 encoded string.

Due to the normalization of the parameters within GCP operators and hooks a parameters like `project` or `topic_project` are deprecated and will be substituted by parameter `project_id`. In `PubSubHook.create_subscription` hook method in the parameter `subscription_project` is replaced by `subscription_project_id`. Template fields are updated accordingly and old ones may not work.

It is required now to pass key-word only arguments to PubSub hook.

These changes are not backward compatible.

`airflow.providers.google.cloud.operators.kubernetes_engine.GKEStartPodOperator`

The `gcp_conn_id` parameter in `GKEPodOperator` is required. In previous versions, it was possible to pass the `None` value to the `gcp_conn_id` in the `GKEStartPodOperator` operator, which resulted in credentials being determined according to the [Application Default Credentials](#) strategy.

Now this parameter requires a value. To restore the previous behavior, configure the connection without specifying the service account.

Detailed information about connection management is available: [Google Cloud Connection](#).

`airflow.providers.google.cloud.hooks.gcs.GCSHook`

- The following parameters have been replaced in all the methods in `GCSHook`:
 - `bucket` is changed to `bucket_name`
 - `object` is changed to `object_name`
- The `maxResults` parameter in `GoogleCloudStorageHook.list` has been renamed to `max_results` for consistency.

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitPigJobOperator`

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitHiveJobOperator`

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitSparkSqlJobOperator`

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitSparkJobOperator`

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitHadoopJobOperator`

`airflow.providers.google.cloud.operators.dataproc.DataprocSubmitPySparkJobOperator`

The ‘properties’ and ‘jars’ properties for the Dataproc related operators (`DataprocXXXOperator`) have been renamed from `dataproc_xxxx_properties` and `dataproc_xxx_jars` to `dataproc_properties` and `dataproc_jars` respectively. Arguments for `dataproc_properties` `dataproc_jars`

`airflow.providers.google.cloud.operators.cloud_storage_transfer_service.CloudDataTransferServiceCreateJobOperator`

To obtain pylint compatibility the `filter` argument in `CloudDataTransferServiceCreateJobOperator` has been renamed to `request_filter`.

`airflow.providers.google.cloud.hooks.cloud_storage_transfer_service.CloudDataTransferServiceHook`

To obtain pylint compatibility the `filter` argument in `CloudDataTransferServiceHook`.
`list_transfer_job` and `CloudDataTransferServiceHook.list_transfer_operations` has been renamed to `request_filter`.

`airflow.providers.google.cloud.hooks.bigquery.BigQueryHook`

In general all hook methods are decorated with `@GoogleBaseHook.fallback_to_default_project_id` thus parameters to hook can only be passed via keyword arguments.

- `create_empty_table` method accepts now `table_resource` parameter. If provided all other parameters are ignored.
- `create_empty_dataset` will now use values from `dataset_reference` instead of raising error if parameters were passed in `dataset_reference` and as arguments to method. Additionally validation of `dataset_reference` is done using `Dataset.from_api_repr`. Exception and log messages has been changed.
- `update_dataset` requires now new `fields` argument (breaking change)
- `delete_dataset` has new signature (`dataset_id, project_id, ...`) previous one was (`project_id, dataset_id, ...`) (breaking change)
- `get_tabledata` returns list of rows instead of API response in dict format. This method is deprecated in favor of `list_rows`. (breaking change)

`airflow.providers.google.cloud.hooks.cloud_build.CloudBuildHook`

`airflow.providers.google.cloud.operators.cloud_build.CloudBuildCreateBuildOperator`

The `api_version` has been removed and will not be used since we migrate `CloudBuildHook` from using Discovery API to native google-cloud-build python library.

The `body` parameter in `CloudBuildCreateBuildOperator` has been deprecated.

Instead, you should pass body using the `build` parameter.

`airflow.providers.google.cloud.hooks.dataflow.DataflowHook.start_python_dataflow`

`airflow.providers.google.cloud.hooks.dataflow.DataflowHook.start_python_dataflow`

`airflow.providers.google.cloud.operators.dataflow.DataflowCreatePythonJobOperator`

Change python3 as Dataflow Hooks/Operators default interpreter

Now the `py_interpreter` argument for DataFlow Hooks/Operators has been changed from `python2` to `python3`.

`airflow.providers.google.common.hooks.base_google.GoogleBaseHook`

To simplify the code, the decorator `provide_gcp_credential_file` has been moved from the inner-class.

Instead of `@GoogleBaseHook._Decorators.provide_gcp_credential_file`, you should write `@GoogleBaseHook.provide_gcp_credential_file`

`airflow.providers.google.cloud.operators.dataproc.DataprocCreateClusterOperator`

It is highly recommended to have 1TB+ disk size for Dataproc to have sufficient throughput: <https://cloud.google.com/compute/docs/disks/performance>

Hence, the default value for `master_disk_size` in `DataprocCreateClusterOperator` has been changed from 500GB to 1TB.

Generating Cluster Config

If you are upgrading from Airflow 1.10.x and are not using **CLUSTER_CONFIG**, You can easily generate config using **make()** of `airflow.providers.google.cloud.operators.dataproc.ClusterGenerator`

This has been proved specially useful if you are using **metadata** argument from older API, refer [AIRFLOW-16911](#) for details.

eg. your cluster creation may look like this in **v1.10.x**

```
path = f"gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh"

create_cluster = DataprocClusterCreateOperator(
    task_id="create_dataproc_cluster",
    cluster_name="test",
    project_id="test",
    zone="us-central1-a",
    region="us-central1",
    master_machine_type="n1-standard-4",
    worker_machine_type="n1-standard-4",
    num_workers=2,
    storage_bucket="test_bucket",
    init_actions_uris=[path],
    metadata={"PIP_PACKAGES": "pyyaml requests pandas openpyxl"},
)
```

After upgrading to **v2.x.x** and using **CLUSTER_CONFIG**, it will look like followed:

```
path = f"gs://goog-dataproc-initialization-actions-us-central1/python/pip-install.sh"

CLUSTER_CONFIG = ClusterGenerator(
    project_id="test",
    zone="us-central1-a",
    master_machine_type="n1-standard-4",
    worker_machine_type="n1-standard-4",
    num_workers=2,
    storage_bucket="test",
    init_actions_uris=[path],
    metadata={"PIP_PACKAGES": "pyyaml requests pandas openpyxl"},
).make()

create_cluster_operator = DataprocClusterCreateOperator(
    task_id="create_dataproc_cluster",
    cluster_name="test",
    project_id="test",
    region="us-central1",
    cluster_config=CLUSTER_CONFIG,
)
```

`airflow.providers.google.cloud.operators.bigquery.BigQueryGetDatasetTablesOperator`

We changed signature of `BigQueryGetDatasetTablesOperator`.

Before:

```
def __init__(  
    dataset_id: str,  
    dataset_resource: dict,  
    # ...  
): ...
```

After:

```
def __init__(  
    dataset_resource: dict,  
    dataset_id: Optional[str] = None,  
    # ...  
): ...
```

Changes in amazon provider package

We strive to ensure that there are no changes that may affect the end user, and your Python files, but this release may contain changes that will require changes to your configuration, DAG Files or other integration e.g. custom operators.

Only changes unique to this provider are described here. You should still pay attention to the changes that have been made to the core (including core operators) as they can affect the integration behavior of this provider.

This section describes the changes that have been made, and what you need to do to update your if you use operators or hooks which integrate with Amazon services (including Amazon Web Service - AWS).

Migration of AWS components

All AWS components (hooks, operators, sensors, example DAGs) will be grouped together as decided in [AIP-21](#). Migrated components remain backwards compatible but raise a `DeprecationWarning` when imported from the old module. Migrated are:

| Old path | New path |
|---|--|
| airflow.hooks.S3_hook.S3Hook | airflow.providers.amazon.aws.hooks.s3.S3Hook |
| airflow.contrib.hooks.aws_athena_hook.AWSAthenaHook | airflow.providers.amazon.aws.hooks.athena.AWSAthenaHook |
| airflow.contrib.hooks.aws_lambda_hook.AwsLambdaHook | airflow.providers.amazon.aws.hooks.lambda_function.AwsLambdaHook |
| airflow.contrib.hooks.aws_sqs_hook.SQSHook | airflow.providers.amazon.aws.hooks.sqs.SQSHook |
| airflow.contrib.hooks.aws sns hook.AwsSnsHook | airflow.providers.amazon.aws.hooks.sns.AwsSnsHook |
| airflow.contrib.operators.aws_athena_operator.AWSAthenaOperator | airflow.providers.amazon.aws.operators.athena.AWSAthenaOperator |
| airflow.contrib.operators.awsbatch.AWSBatchOperator | airflow.providers.amazon.aws.operators.batch.AwsBatchOperator |
| airflow.contrib.operators.awsbatch.BatchProtocol | airflow.providers.amazon.aws.hooks.batch_client.AwsBatchProtocol |
| private attrs and methods on AWSBatchOperator | airflow.providers.amazon.aws.hooks.batch_client.AwsBatchClient |
| n/a | airflow.providers.amazon.aws.hooks.batch_waiters.AwsBatchWaiters |
| airflow.contrib.operators.aws_sqs_publish_operator.SQSPublishOperator | airflow.providers.amazon.aws.operators.sqs.SQSPublishOperator |
| airflow.contrib.operators.aws sns publish operator.SnsPublishOperator | airflow.providers.amazon.aws.operators.sns.SnsPublishOperator |
| airflow.contrib.sensors.aws_athena_sensor.AthenaSensor | airflow.providers.amazon.aws.sensors.athena.AthenaSensor |
| airflow.contrib.sensors.aws_sqs_sensor.SQSSensor | airflow.providers.amazon.aws.sensors.sqs.SQSSensor |

`airflow.providers.amazon.aws.hooks.emr.EmrHook`

`airflow.providers.amazon.aws.operators.emr_add_steps.EmrAddStepsOperator`

`airflow.providers.amazon.aws.operators.emr_create_job_flow.EmrCreateJobFlowOperator`

`airflow.providers.amazon.aws.operators.emr_terminate_job_flow.EmrTerminateJobFlowOperator`

The default value for the `aws_conn_id` was accidentally set to ‘`s3_default`’ instead of ‘`aws_default`’ in some of the emr operators in previous versions. This was leading to `EmrStepSensor` not being able to find their corresponding emr cluster. With the new changes in the `EmrAddStepsOperator`, `EmrTerminateJobFlowOperator` and `EmrCreateJobFlowOperator` this issue is solved.

`airflow.providers.amazon.aws.operators.batch.AwsBatchOperator`

The `AwsBatchOperator` was refactored to extract an `AwsBatchClient` (and inherit from it). The changes are mostly backwards compatible and clarify the public API for these classes; some private methods on `AwsBatchOperator` for polling a job status were relocated and renamed to surface new public methods on `AwsBatchClient` (and via inheritance on `AwsBatchOperator`). A couple of job attributes are renamed on an instance of `AwsBatchOperator`; these were mostly used like private attributes but they were surfaced in the public API, so any use of them needs to be updated as follows:

- `AwsBatchOperator().jobId -> AwsBatchOperator().job_id`

- `AwsBatchOperator().jobName -> AwsBatchOperator().job_name`

The `AwsBatchOperator` gets a new option to define a custom model for waiting on job status changes. The `AwsBatchOperator` can use a new `waiters` parameter, an instance of `AwsBatchWaiters`, to specify that custom job waiters will be used to monitor a batch job. See the latest API documentation for details.

`airflow.providers.amazon.aws.sensors.athena.AthenaSensor`

Replace parameter `max_retries` with `max_retries` to fix typo.

`airflow.providers.amazon.aws.hooks.s3.S3Hook`

Note: The order of arguments has changed for `check_for_prefix`. The `bucket_name` is now optional. It falls back to the `connection schema` attribute. The `delete_objects` now returns `None` instead of a response, since the method now makes multiple api requests when the keys list length is > 1000.

Changes in other provider packages

We strive to ensure that there are no changes that may affect the end user and your Python files, but this release may contain changes that will require changes to your configuration, DAG Files or other integration e.g. custom operators.

Only changes unique to providers are described here. You should still pay attention to the changes that have been made to the core (including core operators) as they can affect the integration behavior of this provider.

This section describes the changes that have been made, and what you need to do to update your if you use any code located in `airflow.providers` package.

Changed return type of `list_prefixes` and `list_keys` methods in `S3Hook`

Previously, the `list_prefixes` and `list_keys` methods returned `None` when there were no results. The behavior has been changed to return an empty list instead of `None` in this case.

Removed HipChat integration

HipChat has reached end of life and is no longer available.

For more information please see <https://community.atlassian.com/t5/Stride-articles/Stride-and-Hipchat-Cloud-have-reached-End-of-Life-updated/ba-p/940248>

`airflow.providers.salesforce.hooks.salesforce.SalesforceHook`

Replace parameter `sandbox` with `domain`. According to change in simple-salesforce package.

Rename `sign_in` function to `get_conn`.

`airflow.providers.apache.pinot.hooks.pinot.PinotAdminHook.create_segment`

Rename parameter name from `format` to `segment_format` in `PinotAdminHook` function `create_segment` for pylint compatible

`airflow.providers.apache.hive.hooks.hive.HiveMetastoreHook.get_partitions`

Rename parameter name from `filter` to `partition_filter` in `HiveMetastoreHook` function `get_partitions` for pylint compatible

`airflow.providers.ftp.hooks.ftp.FTPHook.list_directory`

Remove unnecessary parameter `nlist` in `FTPHook` function `list_directory` for pylint compatible

`airflow.providers.postgres.hooks.postgres.PostgresHook.copy_expert`

Remove unnecessary parameter `open` in `PostgresHook` function `copy_expert` for pylint compatible

`airflow.providers.opsgenie.operators.ops genie_alert.OpsgenieAlertOperator`

Change parameter name from `visibleTo` to `visible_to` in `OpsgenieAlertOperator` for pylint compatible

`airflow.providers imap.hooks imap.ImapHook`

`airflow.providers imap.sensors imap_attachment.ImapAttachmentSensor`

`ImapHook`:

- The order of arguments has changed for `has_mail_attachment`, `retrieve_mail_attachments` and `download_mail_attachments`.
- A new `mail_filter` argument has been added to each of those.

`airflow.providers.http.hooks.http.HttpHook`

The `HTTPHook` is now secured by default: `verify=True` (before: `verify=False`) This can be overwritten by using the `extra_options` param as `{'verify': False}`.

`airflow.providers.cloudant.hooks.cloudant.CloudantHook`

- upgraded cloudant version from `>=0.5.9,<2.0` to `>=2.0`
- removed the use of the `schema` attribute in the connection
- removed `db` function since the database object can also be retrieved by calling `cloudant_session['database_name']`

For example:

```
from airflow.providers.cloudant.hooks.cloudant import CloudantHook

with CloudantHook().get_conn() as cloudant_session:
    database = cloudant_session["database_name"]
```

See the [docs](#) for more information on how to use the new cloudant version.

`airflow.providers.snowflake`

When initializing a Snowflake hook or operator, the value used for `snowflake_conn_id` was always `snowflake_conn_id`, regardless of whether or not you specified a value for it. The default `snowflake_conn_id` value is now switched to `snowflake_default` for consistency and will be properly overridden when specified.

Other changes

This release also includes changes that fall outside any of the sections above.

Standardized “extra” requirements

We standardized the Extras names and synchronized providers package names with the main airflow extras.

We deprecated a number of extras in 2.0.

| Deprecated extras | New extras |
|---------------------------|------------------|
| atlas | apache.atlas |
| aws | amazon |
| azure | microsoft.azure |
| azure_blob_storage | microsoft.azure |
| azure_data_lake | microsoft.azure |
| azure_cosmos | microsoft.azure |
| azure_container_instances | microsoft.azure |
| cassandra | apache.cassandra |
| druid | apache.druid |
| gcp | google |
| gcp_api | google |
| hdfs | apache.hdfs |
| hive | apache.hive |
| kubernetes | cncf.kubernetes |
| mssql | microsoft.mssql |
| pinot | apache.pinot |
| webhdfs | apache.webhdfs |
| winrm | apache.winrm |

For example:

If you want to install integration for Apache Atlas, then instead of `pip install apache-airflow[atlas]` you should use `pip install apache-airflow[apache.atlas]`.

NOTE!

If you want to install integration for Microsoft Azure, then instead of

```
pip install 'apache-airflow[azure_blob_storage,azure_data_lake,azure_cosmos,azure_
˓→container_instances]'
```

you should run `pip install 'apache-airflow[microsoft.azure]'`

If you want to install integration for Amazon Web Services, then instead of `pip install 'apache-airflow[s3,emr]'`, you should execute `pip install 'apache-airflow[aws]'`

The deprecated extras will be removed in 3.0.

Simplify the response payload of endpoints /dag_stats and /task_stats

The response of endpoints `/dag_stats` and `/task_stats` help UI fetch brief statistics about DAGs and Tasks. The format was like

```
{
  "example_http_operator": [
    {
      "state": "success",
      "count": 0,
```

(continues on next page)

(continued from previous page)

```
        "dag_id": "example_http_operator",
        "color": "green"
    },
    {
        "state": "running",
        "count": 0,
        "dag_id": "example_http_operator",
        "color": "lime"
    }
]
}
```

The `dag_id` was repeated in the payload, which makes the response payload unnecessarily bigger.

Now the `dag_id` will not appear repeated in the payload, and the response format is like

```
{
    "example_http_operator": [
        {
            "state": "success",
            "count": 0,
            "color": "green"
        },
        {
            "state": "running",
            "count": 0,
            "color": "lime"
        }
    ]
}
```

3.17 Privacy Notice

The website follows the Privacy Policy of the Apache Software Foundation.

3.18 Project

3.18.1 History

Airflow was started in October 2014 by Maxime Beauchemin at Airbnb. It was open source from the very first commit and officially brought under the Airbnb GitHub and announced in June 2015.

The project joined the Apache Software Foundation's Incubator program in March 2016 and the Foundation announced [Apache Airflow as a Top-Level Project](#) in January 2019.

3.18.2 Committers

- Aizhamal Nurmamat kyzzy (@aijamalnk)
- Alex Guziel (@saguziel)
- Alex Van Boxel (@alexvanboxel)
- Amogh Desai (@amoghrajesh)
- Andrey Anshin (@taragolis)
- Aneesh Joseph (@aneesh-joseph)
- Arthur Wiedmer (@artwr)
- Ash Berlin-Taylor (@ashb)
- Bas Harenslak (@basph)
- Bolke de Bruin (@bolkedebruin)
- Brent Bovenzi (@bbovenzi)
- Buğra Öztürk (@bugraoz93)
- Chao-Han Tsai (@milton0825)
- Chris Riccomini (@criccomini)
- Dan Davydov (@aoen)
- Daniel Imberman (@dimberman)
- Daniel Standish (@dstandish)
- Dennis Ferruzzi (@ferruzzi)
- Elad Kalif (@eladkal)
- Ephraim Anierobi (@ephraimbuddy)
- Felix Uellendall (@feluelle)
- Fokko Driesprong (@fokko)
- Gopal Dirisala (@dirrao)
- Hitesh Shah (@hiteshs)
- Hussein Awala (@hussein-awala)
- Jakob Homan (@jghoman)
- James Timmins (@jhtimmins)
- Jarek Potiuk (@potiuk)
- Jed Cunningham (@jedcunningham)
- Jens Scheffler (@jscheffl)
- Jiajie Zhong (@zhongjiajie)
- Josh Fell (@josh-fell)
- Joshua Carp (@jmcarp)
- Joy Gao (@joygao)
- Kalyan Reddy (@rawwar)

- Kamil Breguła (@mik-laj)
- Karthikeyan Singaravelan (@tirkarthi)
- Kaxil Naik (@kaxil)
- Kengo Seki (@sekikn)
- Kevin Yang (@KevinYang21)
- Leah Cole (@leahecole)
- Maciej Obuchowski (@mobuchowski)
- Malthe Borch (@malthe)
- Maxime “Max” Beauchemin (@mistercrunch)
- Niko Oliveira (@o-nikolas)
- Pankaj Koti (@pankajkoti)
- Pankaj Singh (@pankajastro)
- Patrick Leo Tardif (@pltardif)
- Pavan Kumar Gopidesu (@gopidesupavan)
- Phani Kumar (@phanikumv)
- Pierre Jeambrun (@pierrejeambrun)
- Ping Zhang (@pingzh)
- Qian Yu (@yuqian90)
- Qingping Hou (@houqp)
- Rahul Vats (@vatsrahul1001)
- Rom Sharon (@romsharon98)
- Ry Walker (@ryw)
- Ryan Hamilton (@ryanahamilton)
- Ryan Hatter (@RNHTTR)
- Shahar Epstein (@shahar1)
- Shubham Raj (@shubhamraj-git)
- Siddharth “Sid” Anand (@r39132)
- Sumit Maheshwari (@msumit)
- Tao Feng (@feng-tao)
- Tomasz Urbaszek (@turbaszek)
- Tzu-ping Chung (@uranusjr)
- Utkarsh Sharma (@utkarsharma2)
- Vikram Koka (@vikramkoka)
- Vincent Beck (@vincbeck)
- Wei Lee (@Lee-W)
- Xiaodong Deng (@XD-DENG)

- Xinbin Huang (@xinbinhuang)
- Zhe You Liu (@jason810496)

For the full list of contributors, take a look at Airflow's GitHub Contributor page:

3.18.3 Resources & links

- Airflow's official documentation
- Mailing lists:
 - Developer's mailing list: dev-subscribe@airflow.apache.org
 - All commits mailing list: commits-subscribe@airflow.apache.org
 - Airflow users mailing list: users-subscribe@airflow.apache.org
- Issues on GitHub
- Slack (chat) Channel
- Airflow Improvement Proposals

3.19 License



Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity

(continues on next page)

(continued from previous page)

exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable

(continues on next page)

(continued from previous page)

(except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with

(continues on next page)

(continued from previous page)

- the conditions stated in this License.
5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
 6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
 7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
 8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
 9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

3.20 Operators and Hooks Reference

Here's the list of the operators and hooks which are available in this release in the apache-airflow package.

Airflow has many more integrations available for separate installation as apache-airflow-providers:index.

For details see: apache-airflow-providers:operators-and-hooks-ref/index.

Base:

| Module | Guides |
|--|--------|
| <code>airflow.hooks.base</code> | |
| <code>airflow.models.baseoperator</code> | |
| <code>airflow.sensors.base</code> | |

Operators:

| Operators | Guides |
|--|------------|
| <code>airflow.providers.standard.operators.bash</code> | How to use |
| <code>airflow.providers.standard.operators.python</code> | How to use |
| <code>airflow.providers.standard.operators.datetime</code> | How to use |
| <code>airflow.providers.standard.operators.empty</code> | |
| <code>airflow.providers.common.sql.operators.generic_transfer.GenericTransfer</code> | How to use |
| <code>airflow.providers.standard.operators.latest_only</code> | How to use |
| <code>airflow.providers.standard.operators.trigger_dagrun</code> | How to use |

Sensors:

| Sensors | Guides |
|---|------------|
| <code>airflow.providers.standard.sensors.bash</code> | How to use |
| <code>airflow.providers.standard.sensors.python</code> | How to use |
| <code>airflow.providers.standard.sensors.filesystem</code> | How to use |
| <code>airflow.providers.standard.sensors.date_time</code> | How to use |
| <code>airflow.providers.standard.sensors.external_task</code> | How to use |

Hooks:

| Hooks | Guides |
|--|--------|
| <code>airflow.providers.standard.hooks.filesystem</code> | |
| <code>airflow.providers.standard.hooks.subprocess</code> | |

3.21 Command Line Interface and Environment Variables Reference

3.21.1 Command Line Interface

Airflow has a very rich command line interface that allows for many types of operation on a DAG, starting services, and supporting development and testing.

 Note

For more information on usage CLI, see *Using the Command Line Interface*

Content

- *Positional Arguments*
- *Sub-commands*
 - *api-server*
 - *assets*
 - *backfill*
 - *cheat-sheet*
 - *config*
 - *connections*
 - *dag-processor*
 - *dags*
 - *db*
 - *info*
 - *jobs*
 - *kerberos*
 - *plugins*
 - *pools*
 - *providers*
 - *rotate-fernet-key*
 - *scheduler*
 - *standalone*
 - *tasks*
 - *triggerer*
 - *variables*
 - *version*

Providers that implement executors might contribute additional commands to the CLI. Here are the commands contributed by the community providers:

- Celery Executor and related CLI commands: apache-airflow-providers-celery:cli-ref
- Kubernetes Executor and related CLI commands: apache-airflow-providers-cncf-kubernetes:cli-ref
- Edge Executor and related CLI commands: apache-airflow-providers-edge3:cli-ref
- AWS and related CLI commands: apache-airflow-providers-amazon:cli-ref

- The users and roles CLI commands are described in FAB provider documentation apache-airflow-providers-fab:cli-ref

```
[91mUsage: [0m [37mairflow[0m [[36m-h[0m] [36mGROUP_OR_COMMAND[0m [36m...[0m]
```

Positional Arguments

GROUP_OR_COMMAND Possible choices: api-server, assets, backfill, cheat-sheet, config, connections, dag-processor, dags, db, info, jobs, kerberos, plugins, pools, providers, rotate-fernet-key, scheduler, standalone, tasks, triggerer, variables, version

Sub-commands

api-server

Start an Airflow API server instance

```
[37mairflow api-server[0m [[36m-h[0m] [[36m-A[0m [36mACCESS_LOGFILE[0m] [[36m--apps[0m_
→ [36mAPPS[0m] [[36m-D[0m] [[36m-d[0m] [[36m-H[0m [36mHOST[0m] [[36m-l[0m [36mLOG_
→ FILE[0m] [[36m--pid[0m [[36mPid[0m] [[36m-p[0m [36mPORT[0m] [[36m--proxy-headers[0m]_
→ [[36m--ssl-cert[0m [36mSSL_CERT[0m] [[36m--ssl-key[0m [36mSSL_KEY[0m]
[[36m--stderr[0m [36mSTDERR[0m] [[36m--stdout[0m [36mSTDOUT[0m] [[36m-
→ t[0m [36mWORKER_TIMEOUT[0m] [[36m-w[0m [36mWORKERS[0m]
```

Named Arguments

| | |
|-----------------------------|--|
| -A, --access-logfile | The logfile to store the access log. Use '-' to print to stdout Default: '-' |
| --apps | Applications to run (comma-separated). Default is all. Options: core, execution, all Default: 'all' |
| -D, --daemon | Daemonize instead of running in the foreground Default: False |
| -d, --dev | Start FastAPI in development mode Default: False |
| -H, --host | Set the host on which to run the API server Default: '0.0.0.0' |
| -l, --log-file | Location of the log file |
| --pid | PID file location |
| -p, --port | The port on which to run the API server Default: 8080 |
| --proxy-headers | Enable X-Forwarded-Proto, X-Forwarded-For, X-Forwarded-Port to populate remote address info. Default: False |
| --ssl-cert | Path to the SSL certificate for the webserver Default: '' |

| | |
|-----------------------------|---|
| --ssl-key | Path to the key to use with the SSL certificate |
| | Default: '' |
| --stderr | Redirect stderr to this file |
| --stdout | Redirect stdout to this file |
| -t, --worker-timeout | The timeout for waiting on API server workers |
| | Default: 120 |
| -w, --workers | Number of workers to run on the API server |
| | Default: 4 |

assets

Manage assets

```
[37mairflow assets[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|--|
| COMMAND | Possible choices: details, list, materialize |
|----------------|--|

Sub-commands

details

Show asset details

```
[37mairflow assets details[0m [[36m-h[0m] [[36m--alias[0m] [[36m--name[0m [36mNAME[0m] ↵ [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m--uri[0m [36mURI[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| --alias | Show asset alias |
| | Default: False |
| --name | Asset name |
| | Default: '' |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| --uri | Asset URI |
| | Default: '' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

list

List assets

```
[37mairflow assets list[0m [[36m-h[0m] [[36m--alias[0m] [[36m--columns[0m_
↳ [36mCOLUMNS[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| --alias | Show asset alias |
| | Default: False |
| --columns | List of columns to render. (default: ['name', 'uri', 'group', 'extra']) |
| | Default: ('name', 'uri', 'group', 'extra') |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

materialize

Materialize an asset

```
[37mairflow assets materialize[0m [[36m-h[0m] [[36m--name[0m [36mNAME[0m] [[36m-o[0m_
↳ [36m(table, json, yaml, plain)[0m] [[36m--uri[0m [36mURI[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| --name | Asset name |
| | Default: '' |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| --uri | Asset URI |
| | Default: '' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

backfill

Manage backfills

```
[37mairflow backfill[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

COMMAND Possible choices: create

Sub-commands

create

Run subsections of a DAG for a specified date range.

```
[37mairflow backfill create[0m [[36m-h[0m] [36m--dag-id[0m [36mDAG_ID[0m [[36m--dag-run-
→conf[0m [36mDAG_RUN_CONF[0m] [[36m--dry-run[0m] [36m--from-date[0m [36mFROM_DATE[0m_
→[[36m--max-active-runs[0m [36mMAX_ACTIVE_RUNS[0m]
[[36m--reprocess-behavior[0m [36m{none,completed,failed}[0m]_
→[[36m--run-backwards[0m] [36m--to-date[0m [36mTO_DATE[0m]
```

Named Arguments

| | |
|-----------------------------|---|
| --dag-id | The dag to backfill. |
| --dag-run-conf | JSON dag run configuration. |
| --dry-run | Perform a dry run Default: False |
| --from-date | Earliest logical date to backfill. |
| --max-active-runs | Max active runs for this backfill. |
| --reprocess-behavior | Possible choices: none, completed, failed When a run exists for the logical date, controls whether new runs will be created for the date. Default is none. |
| --run-backwards | If set, the backfill will run tasks from the most recent logical date first. Not supported if there are tasks that depend_on_past. Default: False |
| --to-date | Latest logical date to backfill |

cheat-sheet

Display cheat sheet

```
[37mairflow cheat-sheet[0m [[36m-h[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

config

View configuration

```
[37mairflow config[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

COMMAND Possible choices: get-value, lint, list, update

Sub-commands

get-value

Print the value of the configuration

```
[37mairflow config get-value[0m [[36m-h[0m] [[36m-v[0m] [36msection[0m [36moption[0m
```

Positional Arguments

section The section name

option The option name

Named Arguments

-v, --verbose Make logging output more verbose

Default: False

lint

lint options for the configuration changes while migrating from Airflow 2.x to Airflow 3.0

```
[37mairflow config lint[0m [[36m-h[0m] [[36m--ignore-option[0m [36mIGNORE_OPTION[0m]_
→[[36m--ignore-section[0m [36mIGNORE_SECTION[0m] [[36m--option[0m [36mOPTION[0m] [[36m--
→section[0m [36mSECTION[0m] [[36m-v[0m]
```

Named Arguments

--ignore-option The option name(s) to ignore to lint in the airflow config.

--ignore-section The section name(s) to ignore to lint in the airflow config.

--option The option name(s) to lint in the airflow config.

--section The section name(s) to lint in the airflow config.

-v, --verbose Make logging output more verbose

Default: False

list

List options for the configuration

```
[37mairflow config list[0m [[36m-h[0m] [[36m--color[0m [36m{on,auto,off}[0m] [[36m-c[0m]_
→[[36m-a[0m] [[36m-p[0m] [[36m-d[0m] [[36m-V[0m] [[36m-e[0m] [[36m-s[0m] [[36m--
→section[0m [36mSECTION[0m] [[36m-v[0m]
```

Named Arguments

| | |
|-------------------------------------|---|
| --color | Possible choices: on, auto, off Do emit colored output (default: auto) Default: 'auto' |
| -c, --comment-out-everything | Comment out all configuration options. Useful as starting point for new installation Default: False |
| -a, --defaults | Show only defaults - do not include local configuration, sources, includes descriptions, examples, variables. Comment out everything. Default: False |
| -p, --exclude-providers | Exclude provider configuration (they are included by default) Default: False |
| -d, --include-descriptions | Show descriptions for the configuration variables Default: False |
| -V, --include-env-vars | Show environment variable for each option Default: False |
| -e, --include-examples | Show examples for the configuration variables Default: False |
| -s, --include-sources | Show source of the configuration variable Default: False |
| --section | The section name |
| -v, --verbose | Make logging output more verbose Default: False |

update

update options for the configuration changes while migrating from Airflow 2.x to Airflow 3.0

```
[37mairflow config update[0m [[36m-h[0m] [[36m--all-recommendations[0m] [[36m--fix[0m] ↵
↳ [[36m--ignore-option[0m [36mIGNORE_OPTION[0m] [[36m--ignore-section[0m [36mIGNORE_
↳ SECTION[0m] [[36m--option[0m [36mOPTION[0m] [[36m--section[0m [36mSECTION[0m] [[36m-
↳ v[0m]
```

Named Arguments

| | |
|------------------------------|---|
| --all-recommendations | Include non-breaking (recommended) changes along with breaking ones. (Also use with --fix) |
| --fix | Automatically apply the configuration changes instead of performing a dry run. (Default: dry-run mode) |
| | Default: False |

| | |
|-------------------------|--|
| --ignore-option | The option name(s) to ignore to update in the airflow config. |
| --ignore-section | The section name(s) to ignore to update in the airflow config. |
| --option | The option name(s) to update in the airflow config. |
| --section | The section name(s) to update in the airflow config. |
| -v, --verbose | Make logging output more verbose Default: False |

connections

Manage connections

```
[37mairflow connections[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|--|
| COMMAND | Possible choices: add, create-default-connections, delete, export, get, import, list, test |
|----------------|--|

Sub-commands

add

Add a connection

```
[37mairflow connections add[0m [[36m-h[0m] [[36m--conn-description[0m [36mCONN_
↳ DESCRIPTION[0m] [[36m--conn-extra[0m [36mCONN_EXTRA[0m] [[36m--conn-host[0m [36mCONN_
↳ HOST[0m] [[36m--conn-json[0m [36mCONN_JSON[0m] [[36m--conn-login[0m [36mCONN_LOGIN[0m]
[[36m--conn-password[0m [36mCONN_PASSWORD[0m] [[36m--conn-
↳ port[0m [36mCONN_PORT[0m] [[36m--conn-schema[0m [36mCONN_SCHEMA[0m] [[36m--conn-
↳ type[0m [36mCONN_TYPE[0m] [[36m--conn-uri[0m [36mCONN_URI[0m]
[36mconn_id[0m
```

Positional Arguments

| | |
|----------------|---|
| conn_id | Connection id, required to get/add/delete/test a connection |
|----------------|---|

Named Arguments

| | |
|---------------------------|---|
| --conn-description | Connection description, optional when adding a connection |
| --conn-extra | Connection <i>Extra</i> field, optional when adding a connection |
| --conn-host | Connection host, optional when adding a connection |
| --conn-json | Connection JSON, required to add a connection using JSON representation |
| --conn-login | Connection login, optional when adding a connection |
| --conn-password | Connection password, optional when adding a connection |
| --conn-port | Connection port, optional when adding a connection |
| --conn-schema | Connection schema, optional when adding a connection |
| --conn-type | Connection type, required to add a connection without conn_uri |

--conn-uri Connection URI, required to add a connection without conn_type

create-default-connections

Creates all the default connections from all the providers

```
[37mairflow connections create-default-connections[0m [[36m-h[0m] [[36m-v[0m]
```

Named Arguments

-v, --verbose Make logging output more verbose

Default: False

delete

Delete a connection

```
[37mairflow connections delete[0m [[36m-h[0m] [[36m--color[0m [36m{on,auto,off}[0m]_
↪ [[36m-v[0m] [36mconn_id[0m
```

Positional Arguments

conn_id Connection id, required to get/add/delete/test a connection

Named Arguments

--color Possible choices: on, auto, off

Do emit colored output (default: auto)

Default: 'auto'

-v, --verbose Make logging output more verbose

Default: False

export

All connections can be exported in STDOUT using the following command: airflow connections export - The file format can be determined by the provided file extension. E.g., The following command will export the connections in JSON format: airflow connections export /tmp/connections.json The –file-format parameter can be used to control the file format. E.g., the default format is JSON in STDOUT mode, which can be overridden using: airflow connections export - –file-format yaml The –file-format parameter can also be used for the files, for example: airflow connections export /tmp/connections –file-format json. When exporting in *env* file format, you control whether URI format or JSON format is used to serialize the connection by passing *uri* or *json* with option –serialization-format.

```
[37mairflow connections export[0m [[36m-h[0m] [[36m--file-format[0m [36m{json,yaml,env}
↪ [[0m] [[36m--format[0m [36m{json,yaml,env}[0m] [[36m--serialization-format[0m [36m{json,
↪ uri}[0m] [[36m-v[0m] [36mfile[0m
```

Positional Arguments

file Output file path for exporting the connections

Named Arguments

| | |
|-------------------------------|---|
| --file-format | Possible choices: json, yaml, env File format for the export |
| --format | Possible choices: json, yaml, env Deprecated – use <code>--file-format</code> instead. File format to use for the export. |
| --serialization-format | Possible choices: json, uri When exporting as <code>.env</code> format, defines how connections should be serialized. Default is <code>uri</code> . |
| -v, --verbose | Make logging output more verbose Default: False |

get

Get a connection

```
[37mairflow connections get[0m [[36m-h[0m] [[36m--color[0m [36m{on,auto,off}[0m] [[36m-
˓o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m] [36mconn_id[0m
```

Positional Arguments

| | |
|----------------|---|
| conn_id | Connection id, required to get/add/delete/test a connection |
|----------------|---|

Named Arguments

| | |
|----------------------|--|
| --color | Possible choices: on, auto, off Do emit colored output (default: auto) Default: 'auto' |
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

import

Connections can be imported from the output of the export command. The filetype must by json, yaml or env and will be automatically inferred.

```
[37mairflow connections import[0m [[36m-h[0m] [[36m--overwrite[0m] [[36m-v[0m]_
˓[36mfile[0m
```

Positional Arguments

file Import connections from a file

Named Arguments

--overwrite Overwrite existing entries if a conflict occurs
Default: False

-v, --verbose Make logging output more verbose
Default: False

list

List connections

```
[37mairflow connections list[0m [[36m-h[0m] [[36m--conn-id[0m [36mCONN_ID[0m] [[36m-o[0m_
↳ [36m(table, json, yaml)[0m] [[36m-v[0m]
```

Named Arguments

--conn-id If passed, only items with the specified connection ID will be displayed

-o, --output Possible choices: table, json, yaml, plain
Output format. Allowed values: json, yaml, plain, table (default: table)
Default: 'table'

-v, --verbose Make logging output more verbose
Default: False

test

Test a connection

```
[37mairflow connections test[0m [[36m-h[0m] [[36m-v[0m] [36mconn_id[0m
```

Positional Arguments

conn_id Connection id, required to get/add/delete/test a connection

Named Arguments

-v, --verbose Make logging output more verbose
Default: False

dag-processor

Start a dag processor instance

```
[37mairflow dag-processor[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-D[0m_
↳ [[36m-l[0m [36mLOG_FILE[0m] [[36m-n[0m [36mNUM_RUNS[0m] [[36m--pid[0m [[36mPid[0m]]_
↳ [[36m--stderr[0m [36mSTDERR[0m] [[36m--stdout[0m [36mSTDOUT[0m] [[36m-v[0m]
```

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -D, --daemon | Daemonize instead of running in the foreground |
| | Default: False |
| -l, --log-file | Location of the log file |
| -n, --num-runs | Set the number of runs to execute before exiting |
| | Default: -1 |
| --pid | PID file location |
| --stderr | Redirect stderr to this file |
| --stdout | Redirect stdout to this file |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

dags

Manage DAGs

```
[37mairflow dags[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|---|
| COMMAND | Possible choices: delete, details, list, list-import-errors, list-jobs, list-runs, next-execution, pause, report, reserialize, show, show-dependencies, state, test, trigger, unpause |
|----------------|---|

Sub-commands

delete

Delete all DB records related to the specified DAG

```
[37mairflow dags delete[0m [[36m-h[0m] [[36m-v[0m] [[36m-y[0m] [36mdag_id[0m
```

Positional Arguments

| | |
|---------------|-------------------|
| dag_id | The id of the dag |
|---------------|-------------------|

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose |
| | Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! |
| | Default: False |

details

Get DAG details given a DAG id

```
[37mairflow dags details[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml)[0m]
→[[36m-v[0m] [36mdag_id[0m]
```

Positional Arguments

dag_id The id of the dag

Named Arguments

-o, --output Possible choices: table, json, yaml, plain
Output format. Allowed values: json, yaml, plain, table (default: table)
Default: 'table'

-v, --verbose Make logging output more verbose
Default: False

list

List all the DAGs

```
[37mairflow dags list[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m--columns[0m
→[36mCOLUMNS[0m] [[36m-l[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

-B, --bundle-name The name of the DAG bundle to use; may be provided more than once

--columns List of columns to render. (default: ['dag_id', 'fileloc', 'owner', 'is_paused'])
Default: ('dag_id', 'fileloc', 'owners', 'is_paused', 'bundle_name', 'bundle_version')

-l, --local Shows local parsed DAGs and their import errors, ignores content serialized in DB
Default: False

-o, --output Possible choices: table, json, yaml, plain
Output format. Allowed values: json, yaml, plain, table (default: table)
Default: 'table'

-v, --verbose Make logging output more verbose
Default: False

list-import-errors

List all the DAGs that have import errors

```
[37mairflow dags list-import-errors[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-
→1[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -l, --local | Shows local parsed DAGs and their import errors, ignores content serialized in DB |
| | Default: False |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

list-jobs

List the jobs

```
[37mairflow dags list-jobs[0m [[36m-h[0m] [[36m-d[0m [36mDAG_ID[0m] [[36m--limit[0m_
↳ [36mLIMIT[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m--state[0m_
↳ [36mrunning, success, restarting, failed[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -d, --dag-id | The id of the dag |
| --limit | Return a limited number of records |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| --state | Possible choices: running, success, restarting, failed |
| | Only list the jobs corresponding to the state |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

list-runs

List DAG runs given a DAG id. If state option is given, it will only search for all the dagruns with the given state. If no_backfill option is given, it will filter out all backfill dagruns for given dag id. If start_date is given, it will filter out all the dagruns that were executed before this date. If end_date is given, it will filter out all the dagruns that were executed after this date.

```
[37mairflow dags list-runs[0m [[36m-h[0m] [[36m-e[0m [36mEND_DATE[0m] [[36m--no-
↳ backfill[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-s[0m [36mSTART-
↳ DATE[0m] [[36m--state[0m [36mqueued, running, success, failed[0m] [[36m-v[0m] [36mdag-
↳ id[0m
```

Positional Arguments

dag_id The id of the dag

Named Arguments

| | |
|-------------------------|--|
| -e, --end-date | Override end_date YYYY-MM-DD |
| --no-backfill | filter all the backfill dagruns given the dag id |
| | Default: False |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -s, --start-date | Override start_date YYYY-MM-DD |
| --state | Possible choices: queued, running, success, failed |
| | Only list the DAG runs corresponding to the state |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

next-execution

Get the next logical datetimes of a DAG. It returns one execution unless the num-executions option is given

```
[37mairflow dags next-execution[0m [[36m-h[0m] [[36m-n[0m [36mNUM_EXECUTIONS[0m] [[36m-
˓→v[0m] [36mdag_id[0m
```

Positional Arguments

dag_id The id of the dag

Named Arguments

| | |
|-----------------------------|---|
| -n, --num-executions | The number of next logical date times to show |
| | Default: 1 |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

pause

Pause one or more DAGs. This command allows to halt the execution of specified DAGs, disabling further task scheduling. Use *-treat-dag-id-as-regex* to target multiple DAGs by treating the *-dag-id* as a regex pattern.

```
[37mairflow dags pause[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]-
˓→[[36m--treat-dag-id-as-regex[0m] [[36m-v[0m] [[36m-y[0m] [36mdag_id[0m
```

Positional Arguments

dag_id The id of the dag

Named Arguments

| | |
|--------------------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| --treat-dag-id-as-regex | if set, dag_id will be treated as regex instead of an exact string |
| | Default: False |
| -v, --verbose | Make logging output more verbose |
| | Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! |
| | Default: False |

report

Show DagBag loading report

```
[37mairflow dags report[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-o[0m_
↳[36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|--------------------------|--|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

reserialize

Reserialize DAGs in the metadata DB. This can be particularly useful if your serialized DAGs become out of sync with the Airflow version you are using.

```
[37mairflow dags reserialize[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-v[0m]
```

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

show

The `--imgcat` option only works in iTerm.

For more information, see: <https://www.iterm2.com/documentation-images.html>

The `--save` option saves the result to the indicated file.

The file format is determined by the file extension. For more information about supported format, see: <https://www.graphviz.org/doc/info/output.html>

If you want to create a PNG file then you should execute the following command: `airflow dags show <dag_id> --save output.png`

If you want to create a DOT file then you should execute the following command: `airflow dags show <dag_id> --save output.dot`

```
[37mairflow dags show[0m [[36m-h[0m] [[36m--imgcat[0m] [[36m-s[0m [36mSAVE[0m] [[36m-
→v[0m] [36mdag_id[0m
```

Positional Arguments

dag_id The id of the dag

Named Arguments

--imgcat Displays graph using the imgcat tool.

Default: False

-s, --save Saves the result to the indicated file.

-v, --verbose Make logging output more verbose

Default: False

show-dependencies

The `--imgcat` option only works in iTerm.

For more information, see: <https://www.iterm2.com/documentation-images.html>

The `--save` option saves the result to the indicated file.

The file format is determined by the file extension. For more information about supported format, see: <https://www.graphviz.org/doc/info/output.html>

If you want to create a PNG file then you should execute the following command: `airflow dags show-dependencies --save output.png`

If you want to create a DOT file then you should execute the following command: `airflow dags show-dependencies --save output.dot`

```
[37mairflow dags show-dependencies[0m [[36m-h[0m] [[36m--imgcat[0m] [[36m-s[0m_
→[36mSAVE[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|---|
| --imgcat | Displays graph using the imgcat tool. Default: False |
| -s, --save | Saves the result to the indicated file. |
| -v, --verbose | Make logging output more verbose Default: False |

state

Get the status of a dag run

```
[37mairflow dags state[0m [[36m-h[0m] [[36m-v[0m] [36mdag_id[0m [36mlogical_date_or_run_
-id[0m
```

Positional Arguments

| | |
|-------------------------------|---|
| dag_id | The id of the dag |
| logical_date_or_run_id | The logical date of the DAG or run_id of the DAGRun |

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

test

Execute one single DagRun for a given DAG and logical date.

You can test a DAG in three ways: 1. Using default bundle:

Unexpected indentation.

```
airflow dags test <DAG_ID>
```

2. Using a specific bundle if multiple DAG bundles are configured: airflow dags test <DAG_ID> -bundle-name <BUNDLE_NAME> (or -B <BUNDLE_NAME>)
3. Using a specific DAG file: airflow dags test <DAG_ID> -dagfile-path <PATH> (or -f <PATH>)

The `-imgcat-dagrun` option only works in iTerm.

For more information, see: <https://www.iterm2.com/documentation-images.html>

If `--save-dagrun` is used, then, after completing the backfill, saves the diagram for current DAG Run to the indicated file. The file format is determined by the file extension. For more information about supported format, see: <https://www.graphviz.org/doc/info/output.html>

If you want to create a PNG file then you should execute the following command: `airflow dags test <DAG_ID> <LOGICAL_DATE> --save-dagrun output.png`

If you want to create a DOT file then you should execute the following command: `airflow dags test <DAG_ID> <LOGICAL_DATE> --save-dagrun output.dot`

```
[37mairflow dags test[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-c[0m_
↳ [36mCONF[0m] [[36m-f[0m [36mDAGFILE_PATH[0m] [[36m--imgcat-dagrun[0m] [[36m--mark-
↳ success-pattern[0m [36mMARK_SUCCESS_PATTERN[0m] [[36m--save-dagrun[0m [36mSAVE_
↳ DAGRUN[0m] [[36m--show-dagrun[0m]
[[36m--use-executor[0m] [[36m-v[0m]
[36mdag_id[0m [[36mlogical_date[0m]
```

Positional Arguments

| | |
|---------------------|--|
| dag_id | The id of the dag |
| logical_date | The logical date of the DAG (optional) |

Named Arguments

| | |
|-------------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -c, --conf | JSON string that gets pickled into the DagRun's conf attribute |
| -f, --dagfile-path | Path to the dag file. Can be absolute or relative to current directory |
| --imgcat-dagrun | After completing the dag run, prints a diagram on the screen for the current DAG Run using the imgcat tool. Default: False |
| --mark-success-pattern | Don't run task_ids matching the regex <MARK_SUCCESS_PATTERN>, mark them as successful instead. Can be used to skip e.g. dependency check sensors or cleanup steps in local testing. |
| --save-dagrun | After completing the backfill, saves the diagram for current DAG Run to the indicated file. |
| --show-dagrun | After completing the backfill, shows the diagram for current DAG Run. The diagram is in DOT language Default: False |
| --use-executor | Use an executor to test the DAG. By default it runs the DAG without an executor. If set, it uses the executor configured in the environment. Default: False |
| -v, --verbose | Make logging output more verbose Default: False |

trigger

Trigger a new DAG run. If DAG is paused then dagrun state will remain queued, and the task won't run.

```
[37mairflow dags trigger[0m [[36m-h[0m] [[36m-c[0m [36mCONF[0m] [[36m-l[0m [36mLOGICAL_
↳ DATE[0m] [[36m--no-replace-microseconds[0m] [[36m-o[0m [36m(table, json, yaml,
↳ plain)[0m] [[36m-r[0m [36mRUN_ID[0m] [[36m-v[0m] [36mdag_id[0m]
```

Positional Arguments

dag_id The id of the dag

Named Arguments

| | |
|----------------------------------|--|
| -c, --conf | JSON string that gets pickled into the DagRun's conf attribute |
| -l, --logical-date | The logical date of the DAG |
| --no-replace-microseconds | whether microseconds should be zeroed Default: True |
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -r, --run-id | Helps to identify this run |
| -v, --verbose | Make logging output more verbose Default: False |

unpause

Resume one or more DAGs. This command allows to restore the execution of specified DAGs, enabling further task scheduling. Use *-treat-dag-id-as-regex* to target multiple DAGs treating the *-dag-id* as a regex pattern.

```
[37mairflow dags unpause[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]_
↳ [[36m--treat-dag-id-as-regex[0m] [[36m-v[0m] [[36m-y[0m] [36mdag_id[0m]
```

Positional Arguments

dag_id The id of the dag

Named Arguments

| | |
|--------------------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| --treat-dag-id-as-regex | if set, dag_id will be treated as regex instead of an exact string Default: False |
| -v, --verbose | Make logging output more verbose Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

db

Database operations

```
[37mairflow db[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|--|
| COMMAND | Possible choices: check, check-migrations, clean, downgrade, drop-archived, export-archived, migrate, reset, shell |
|----------------|--|

Sub-commands

check

Check if the database can be reached

```
[37mairflow db check[0m [[36m-h[0m] [[36m--retry[0m [36mRETRY[0m] [[36m--retry-delay[0m_
↳ [36mRETRY_DELAY[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| --retry | Retry database check upon failure Default: 0 |
| --retry-delay | Wait time between retries in seconds Default: 1 |
| -v, --verbose | Make logging output more verbose Default: False |

check-migrations

Check if migration have finished (or continually check until timeout)

```
[37mairflow db check-migrations[0m [[36m-h[0m] [[36m-t[0m [36mMIGRATION_WAIT_TIMEOUT[0m]_
↳ [[36m-v[0m]
```

Named Arguments

| | |
|-------------------------------------|--|
| -t, --migration-wait-timeout | timeout to wait for db to migrate Default: 60 |
| -v, --verbose | Make logging output more verbose Default: False |

clean

Purge old records in metastore tables

```
[37mairflow db clean[0m [[36m-h[0m] [36m--clean-before-timestamp[0m [36mCLEAN_BEFORE_-
↳ TIMESTAMP[0m [[36m--dry-run[0m] [[36m--skip-archive[0m] [[36m-t[0m [36mTABLES[0m]_
↳ [[36m-v[0m] [[36m-y[0m]
```

Named Arguments

| | |
|---------------------------------|---|
| --clean-before-timestamp | The date or timestamp before which data should be purged. If no timezone info is supplied then dates are assumed to be in airflow default timezone. Example: '2022-01-01 00:00:00+01:00' |
| --dry-run | Perform a dry run Default: False |
| --skip-archive | Don't preserve purged records in an archive table. Default: False |
| -t, --tables | Table names to perform maintenance on (use comma-separated list). Options: ['_xcom_archive', 'asset_event', 'callback_request', 'celery_taskmeta', 'celery_tasksetmeta', 'dag', 'dag_run', 'dag_version', 'deadline', 'import_error', 'job', 'log', 'sla_miss', 'task_instance', 'task_instance_history', 'task_reschedule', 'trigger', 'xcom'] |
| -v, --verbose | Make logging output more verbose Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

downgrade

Downgrade the schema of the metadata database. You must provide either *-to-revision* or *-to-version*. To print but not execute commands, use option *-show-sql-only*. If using options *-from-revision* or *-from-version*, you must also use *-show-sql-only*, because if actually *running* migrations, we should only migrate from the *current* Alembic revision.

```
[37mairflow db downgrade[0m [[36m-h[0m] [[36m--from-revision[0m [36mFROM_REVISION[0m]_
↳ [[36m--from-version[0m [36mFROM_VERSION[0m] [[36m-s[0m] [[36m-r[0m [36mTO_REVISION[0m]_
↳ [[36m-n[0m [36mTO_VERSION[0m] [[36m-v[0m] [[36m-y[0m]
```

Named Arguments

| | |
|----------------------------|---|
| --from-revision | (Optional) If generating sql, may supply a <i>from</i> Alembic revision |
| --from-version | (Optional) If generating sql, may supply a <i>from</i> version |
| -s, --show-sql-only | Don't actually run migrations; just print out sql scripts for offline migration. Required if using either <i>-from-revision</i> or <i>-from-version</i> . Default: False |
| -r, --to-revision | The Alembic revision to downgrade to. Note: must provide either <i>-to-revision</i> or <i>-to-version</i> . |
| -n, --to-version | (Optional) If provided, only run migrations up to this version. |
| -v, --verbose | Make logging output more verbose Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

drop-archived

Drop archived tables created through the db clean command

```
[37mairflow db drop-archived[0m [[36m-h[0m] [[36m-t[0m [36mTABLES[0m] [[36m-y[0m]
```

Named Arguments

| | |
|---------------------|---|
| -t, --tables | Table names to perform maintenance on (use comma-separated list). Options: ['_xcom_archive', 'asset_event', 'callback_request', 'celery_taskmeta', 'celery_tasksetmeta', 'dag', 'dag_run', 'dag_version', 'deadline', 'import_error', 'job', 'log', 'sla_miss', 'task_instance', 'task_instance_history', 'task_reschedule', 'trigger', 'xcom'] |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

export-archived

Export archived data from the archive tables

```
[37mairflow db export-archived[0m [[36m-h[0m] [[36m--drop-archives[0m] [[36m--export-  
-format[0m [36m{csv}[0m] [36m--output-path[0m [36mDIRPATH[0m [[36m-t[0m [36mTABLES[0m]_  
[[36m-y[0m]
```

Named Arguments

| | |
|------------------------|---|
| --drop-archives | Drop the archive tables after exporting. Use with caution. Default: False |
| --export-format | Possible choices: csv The file format to export the cleaned data Default: 'csv' |
| --output-path | The path to the output directory to export the cleaned data. This directory must exist. |
| -t, --tables | Table names to perform maintenance on (use comma-separated list). Options: ['_xcom_archive', 'asset_event', 'callback_request', 'celery_taskmeta', 'celery_tasksetmeta', 'dag', 'dag_run', 'dag_version', 'deadline', 'import_error', 'job', 'log', 'sla_miss', 'task_instance', 'task_instance_history', 'task_reschedule', 'trigger', 'xcom'] |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

migrate

Migrate the schema of the metadata database. Create the database if it does not exist To print but not execute commands, use option `--show-sql-only`. If using options `--from-revision` or `--from-version`, you must also use `--show-sql-only`, because if actually *running* migrations, we should only migrate from the *current* Alembic revision.

```
[37mairflow db migrate[0m [[36m-h[0m] [[36m--from-revision[0m [36mFROM_REVISION[0m]
↳ [[36m--from-version[0m [36mFROM_VERSION[0m] [[36m-s[0m] [[36m-r[0m [36mTO_REVISION[0m]
↳ [[36m-n[0m [36mTO_VERSION[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------------|---|
| --from-revision | (Optional) If generating sql, may supply a <i>from</i> Alembic revision |
| --from-version | (Optional) If generating sql, may supply a <i>from</i> version |
| -s, --show-sql-only | Don't actually run migrations; just print out sql scripts for offline migration. Required if using either <i>from-revision</i> or <i>from-version</i> . Default: False |
| -r, --to-revision | (Optional) If provided, only run migrations up to and including this Alembic revision. |
| -n, --to-version | (Optional) The airflow version to upgrade to. Note: must provide either <i>to-revision</i> or <i>to-version</i> . |
| -v, --verbose | Make logging output more verbose Default: False |

reset

Burn down and rebuild the metadata database

```
[37mairflow db reset[0m [[36m-h[0m] [[36m-s[0m] [[36m-v[0m] [[36m-y[0m]
```

Named Arguments

| | |
|------------------------|---|
| -s, --skip-init | Only remove tables; do not perform db init. Default: False |
| -v, --verbose | Make logging output more verbose Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

shell

Runs a shell to access the database

```
[37mairflow db shell[0m [[36m-h[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

info

Show information about current Airflow and environment

```
[37mairflow info[0m [[36m-h[0m] [[36m--anonymize[0m] [[36m--file-io[0m] [[36m-o[0m_
↳ [36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|---|
| --anonymize | Minimize any personal identifiable information. Use it when sharing output with others. |
| | Default: False |
| --file-io | Send output to file.io service and returns link. |
| | Default: False |
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

jobs

Manage jobs

```
[37mairflow jobs[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|-------------------------|
| COMMAND | Possible choices: check |
|----------------|-------------------------|

Sub-commands

check

Checks if job(s) are still alive

```
[37mairflow jobs check[0m [[36m-h[0m] [[36m--allow-multiple[0m] [[36m--hostname[0m_
↳ [36mHOSTNAME[0m] [[36m--job-type[0m [36m{SchedulerJob,TriggererJob,DagProcessorJob}
↳ [0m] [[36m--limit[0m [36mLIMIT[0m] [[36m--local[0m] [[36m-v[0m]
```

Named Arguments

| | |
|-------------------------|---|
| --allow-multiple | If passed, this command will be successful even if multiple matching alive jobs are found. |
| | Default: False |
| --hostname | The hostname of job(s) that will be checked. |
| --job-type | Possible choices: SchedulerJob, TriggererJob, DagProcessorJob The type of job(s) that will be checked. |

| | |
|----------------------|---|
| --limit | The number of recent jobs that will be checked. To disable limit, set 0. Default: 1 |
| --local | If passed, this command will only show jobs from the local host (those with a hostname matching what <i>hostname_callable</i> returns). Default: False |
| -v, --verbose | Make logging output more verbose Default: False |

examples: To check if the local scheduler is still working properly, run:

```
$ airflow jobs check --job-type SchedulerJob --local"
```

To check if any scheduler is running when you are using high availability, run:

```
$ airflow jobs check --job-type SchedulerJob --allow-multiple --limit 100
```

kerberos

Start a kerberos ticket renewer

```
[37mairflow kerberos[0m [[36m-h[0m] [[36m-D[0m] [[36m-k[0m [[36mKEYTAB[0m]] [[36m-l[0m_
↳ [36mLOG_FILE[0m] [[36m-o[0m] [[36m--pid[0m [[36mPid[0m]] [[36m--stderr[0m_
↳ [36mSTDERR[0m] [[36m--stdout[0m [[36mSTDOUT[0m] [[36m-v[0m] [[36mprincipal[0m]
```

Positional Arguments

| | |
|------------------|--------------------|
| principal | kerberos principal |
|------------------|--------------------|

Named Arguments

| | |
|-----------------------|--|
| -D, --daemon | Daemonize instead of running in the foreground Default: False |
| -k, --keytab | keytab Default: 'airflow.keytab' |
| -l, --log-file | Location of the log file |
| -o, --one-time | Run airflow kerberos one time instead of forever Default: False |
| --pid | PID file location |
| --stderr | Redirect stderr to this file |
| --stdout | Redirect stdout to this file |
| -v, --verbose | Make logging output more verbose Default: False |

plugins

Dump information about loaded plugins

```
[37mairflow plugins[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-  
↳v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

pools

Manage pools

```
[37mairflow pools[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|--|
| COMMAND | Possible choices: delete, export, get, import, list, set |
|----------------|--|

Sub-commands

delete

Delete pool

```
[37mairflow pools delete[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]_  
↳[[36m-v[0m] [36mNAME[0m
```

Positional Arguments

| | |
|-------------|-----------|
| NAME | Pool name |
|-------------|-----------|

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

export

Export all pools

```
[37mairflow pools export[0m [[36m-h[0m] [[36m-v[0m] [36mFILEPATH[0m]
```

Positional Arguments

| | |
|-----------------|-------------------------------|
| FILEPATH | Export all pools to JSON file |
|-----------------|-------------------------------|

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

get

Get pool size

```
[37mairflow pools get[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml)[0m] [[36m-
˓→v[0m] [36mNAME[0m]
```

Positional Arguments

| | |
|-------------|-----------|
| NAME | Pool name |
|-------------|-----------|

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

import

Import pools

```
[37mairflow pools import[0m [[36m-h[0m] [[36m-v[0m] [36mFILEPATH[0m]
```

Positional Arguments

| | |
|-----------------|--|
| FILEPATH | Import pools from JSON file. Example format: |
|-----------------|--|

```
{
    "pool_1": {"slots": 5, "description": "", "include_
˓→deferred": true},
    "pool_2": {"slots": 10, "description": "test", "include_
˓→deferred": false}
}
```

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

list

List pools

```
[37mairflow pools list[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]_
↪[[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

set

Configure pool

```
[37mairflow pools set[0m [[36m-h[0m] [[36m--include-deferred[0m] [[36m-o[0m [36m(table,_
↪json, yaml, plain)[0m] [[36m-v[0m] [36mNAME[0m [36mslots[0m [36mdescription[0m]
```

Positional Arguments

| | |
|--------------------|------------------|
| NAME | Pool name |
| slots | Pool slots |
| description | Pool description |

Named Arguments

| | |
|---------------------------|--|
| --include-deferred | Include deferred tasks in calculations for Pool Default: False |
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

providers

Display providers

```
[37mairflow providers [[36m-h[0m] [[36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|---|
| COMMAND | Possible choices: auth-managers, behaviours, configs, executors, get, hooks, lazy-loaded, links, list, logging, notifications, queues, secrets, triggers, widgets |
|----------------|---|

Sub-commands

auth-managers

Get information about auth managers provided

```
[37mairflow providers auth-managers [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, u
˓→plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

behaviours

Get information about registered connection types with custom behaviours

```
[37mairflow providers behaviours [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, u
˓→plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

configs

Get information about provider configuration

```
[37mairflow providers configs [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, u
˓→plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

executors

Get information about executors provided

```
[37mairflow providers executors[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,_
plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

get

Get detailed information about a provider

```
[37mairflow providers get[0m [[36m-h[0m] [[36m--color[0m [36m{on,auto,off}[0m] [[36m-
f[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m] [36mprovider_name[0m]
```

Positional Arguments

| | |
|----------------------|---|
| provider_name | Provider name, required to get provider information |
|----------------------|---|

Named Arguments

| | |
|---------------------|--|
| --color | Possible choices: on, auto, off Do emit colored output (default: auto) Default: 'auto' |
| -f, --full | Full information about the provider, including documentation information. Default: False |
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

hooks

List registered provider hooks

```
[37mairflow providers hooks[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]_
↳ [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

lazy-loaded

Checks that provider configuration is lazy loaded

```
[37mairflow providers lazy-loaded[0m [[36m-h[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -v, --verbose | Make logging output more verbose Default: False |
|----------------------|--|

links

List extra links registered by the providers

```
[37mairflow providers links[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]_
↳ [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

list

List installed providers

```
[37mairflow providers list[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]
↳ [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

logging

Get information about task logging handlers provided

```
[37mairflow providers logging[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,
↳ plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

notifications

Get information about notifications provided

```
[37mairflow providers notifications[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,
↳ plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

queues

Get information about queues provided

```
[37mairflow providers queues[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,  
- plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

secrets

Get information about secrets backends provided

```
[37mairflow providers secrets[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,  
- plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

triggers

List registered provider triggers

```
[37mairflow providers triggers[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,  
- plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

widgets

Get information about registered connection form widgets

```
[37mairflow providers widgets[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m] [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain |
| | Output format. Allowed values: json, yaml, plain, table (default: table) |
| | Default: 'table' |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

rotate-fernet-key

Rotate all encrypted connection credentials and variables; see <https://airflow.apache.org/docs/apache-airflow/stable/howto/secure-connections.html#rotating-encryption-keys>

```
[37mairflow rotate-fernet-key[0m [[36m-h[0m]
```

scheduler

Start a scheduler instance

```
[37mairflow scheduler[[36m-h[0m] [[36m-D[0m] [[36m-l[0m [36mLOG_FILE[0m] [[36m-n[0m[36mNUM_RUNS[0m] [[36m--pid[0m [[36mPid[0m] [[36m-s[0m] [[36m--stderr[0m[[36mSTDERR[0m] [[36m--stdout[0m [36mSTDOUT[0m] [[36m-v[0m]
```

Named Arguments

| | |
|------------------------------|---|
| -D, --daemon | Daemonize instead of running in the foreground |
| | Default: False |
| -l, --log-file | Location of the log file |
| -n, --num-runs | Set the number of runs to execute before exiting |
| | Default: -1 |
| --pid | PID file location |
| -s, --skip-serve-logs | Don't start the serve logs process along with the workers |
| | Default: False |
| --stderr | Redirect stderr to this file |
| --stdout | Redirect stdout to this file |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

Signals:

- SIGUSR2: Dump a snapshot of task state being tracked by the executor.

Example:

```
pskill -f -USR2 "airflow scheduler"
```

standalone

Run an all-in-one copy of Airflow

```
[37mairflow standalone[0m [[36m-h[0m]
```

tasks

Manage tasks

```
[37mairflow tasks[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|---|
| COMMAND | Possible choices: clear, failed-deps, list, render, state, states-for-dag-run, test |
|----------------|---|

Sub-commands**clear**

Clear a set of task instance, as if they never ran

```
[37mairflow tasks clear[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-R[0m] [[36m-
-d[0m] [[36m-e[0m [36mEND_DATE[0m] [[36m-f[0m] [[36m-r[0m] [[36m-s[0m [36mSTART_-
-DATE[0m] [[36m-t[0m [36mTASK_REGEX[0m] [[36m-u[0m] [[36m-v[0m] [[36m-y[0m] [36mdag_-
-id[0m
```

Positional Arguments

| | |
|---------------|-------------------|
| dag_id | The id of the dag |
|---------------|-------------------|

Named Arguments

| | |
|---------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -R, --dag-regex | Search dag_id as regex instead of exact string |
| | Default: False |
| -d, --downstream | Include downstream tasks |
| | Default: False |
| -e, --end-date | Override end_date YYYY-MM-DD |
| -f, --only-failed | Only failed jobs |
| | Default: False |
| -r, --only-running | Only running jobs |
| | Default: False |
| -s, --start-date | Override start_date YYYY-MM-DD |

| | |
|-------------------------|--|
| -t, --task-regex | The regex to filter specific task_ids (optional) |
| -u, --upstream | Include upstream tasks Default: False |
| -v, --verbose | Make logging output more verbose Default: False |
| -y, --yes | Do not prompt to confirm. Use with care! Default: False |

failed-deps

Returns the unmet dependencies for a task instance from the perspective of the scheduler. In other words, why a task instance doesn't get scheduled and then queued by the scheduler, and then run by an executor.

```
[37mairflow tasks failed-deps[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m--map-
-+index[0m [36mMAP_INDEX[0m] [[36m-v[0m] [36mdag_id[0m [36mtask_id[0m [36mlogical_date_
-+or_run_id[0m
```

Positional Arguments

| | |
|-------------------------------|---|
| dag_id | The id of the dag |
| task_id | The id of the task |
| logical_date_or_run_id | The logical date of the DAG or run_id of the DAGRun |

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| --map-index | Mapped task index Default: -1 |
| -v, --verbose | Make logging output more verbose Default: False |

list

List the tasks within a DAG

```
[37mairflow tasks list[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-v[0m] [36mdag_
-+id[0m
```

Positional Arguments

| | |
|---------------|-------------------|
| dag_id | The id of the dag |
|---------------|-------------------|

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| -v, --verbose | Make logging output more verbose Default: False |

render

Render a task instance's template(s)

```
[37mairflow tasks render[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m--map-
˓index[0m [36mMAP_INDEX[0m] [[36m-v[0m] [36mdag_id[0m [36mtask_id[0m [36mlogical_date_
˓or_run_id[0m
```

Positional Arguments

| | |
|-------------------------------|---|
| dag_id | The id of the dag |
| task_id | The id of the task |
| logical_date_or_run_id | The logical date of the DAG or run_id of the DAGRun |

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| --map-index | Mapped task index |
| | Default: -1 |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

state

Get the status of a task instance

```
[37mairflow tasks state[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m--map-
˓index[0m [36mMAP_INDEX[0m] [[36m-v[0m] [36mdag_id[0m [36mtask_id[0m [36mlogical_date_
˓or_run_id[0m
```

Positional Arguments

| | |
|-------------------------------|---|
| dag_id | The id of the dag |
| task_id | The id of the task |
| logical_date_or_run_id | The logical date of the DAG or run_id of the DAGRun |

Named Arguments

| | |
|--------------------------|---|
| -B, --bundle-name | The name of the DAG bundle to use; may be provided more than once |
| --map-index | Mapped task index |
| | Default: -1 |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

states-for-dag-run

Get the status of all task instances in a dag run

```
[37mairflow tasks states-for-dag-run[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml,  
plain)[0m] [[36m-v[0m] [36mdag_id[0m [36mlogical_date_or_run_id[0m]
```

Positional Arguments

dag_id The id of the dag

logical_date_or_run_id The logical date of the DAG or run_id of the DAGRun

Named Arguments

-o, --output Possible choices: table, json, yaml, plain

Output format. Allowed values: json, yaml, plain, table (default: table)

Default: 'table'

-v, --verbose Make logging output more verbose

Default: False

test

Test a task instance. This will run a task without checking for dependencies or recording its state in the database

```
[37mairflow tasks test[0m [[36m-h[0m] [[36m-B[0m [36mBUNDLE_NAME[0m] [[36m-n[0m] [[36m--  
env-vars[0m [36mENV_VARS[0m] [[36m--map-index[0m [36mMAP_INDEX[0m] [[36m-m[0m] [[36m-  
t[0m [36mTASK_PARAMS[0m] [[36m-v[0m] [36mdag_id[0m [36mtask_id[0m [[36mlogical_date_or_  
run_id[0m]
```

Positional Arguments

dag_id The id of the dag

task_id The id of the task

logical_date_or_run_id The logical date of the DAG or run_id of the DAGRun (optional)

Named Arguments

-B, --bundle-name The name of the DAG bundle to use; may be provided more than once

-n, --dry-run Perform a dry run for each task. Only renders Template Fields for each task, nothing else

Default: False

--env-vars Set env var in both parsing time and runtime for each of entry supplied in a JSON dict

--map-index Mapped task index

Default: -1

-m, --post-mortem Open debugger on uncaught exception

Default: False

| | |
|--------------------------|--------------------------------------|
| -t, --task-params | Sends a JSON params dict to the task |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

triggerer

Start a triggerer instance

```
[37mairflow triggerer[0m [[36m-h[0m] [[36m--capacity[0m [36mCAPACITY[0m] [[36m-D[0m]
↳ [[36m-1[0m [36mLOG_FILE[0m] [[36m--pid[0m [[36mPid[0m]] [[36m-s[0m] [[36m--stderr[0m_
↳ [36mSTDERR[0m] [[36m--stdout[0m [36mSTDOUT[0m] [[36m-v[0m]
```

Named Arguments

| | |
|------------------------------|---|
| --capacity | The maximum number of triggers that a Triggerer will run at one time. |
| -D, --daemon | Daemonize instead of running in the foreground |
| | Default: False |
| -l, --log-file | Location of the log file |
| --pid | PID file location |
| -s, --skip-serve-logs | Don't start the serve logs process along with the workers |
| | Default: False |
| --stderr | Redirect stderr to this file |
| --stdout | Redirect stdout to this file |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

variables

Manage variables

```
[37mairflow variables[0m [[36m-h[0m] [36mCOMMAND[0m [36m...[0m
```

Positional Arguments

| | |
|----------------|--|
| COMMAND | Possible choices: delete, export, get, import, list, set |
|----------------|--|

Sub-commands

delete

Delete variable

```
[37mairflow variables delete[0m [[36m-h[0m] [[36m-v[0m] [36mkey[0m
```

Positional Arguments

| | |
|------------|--------------|
| key | Variable key |
|------------|--------------|

Named Arguments

| | |
|----------------------|----------------------------------|
| -v, --verbose | Make logging output more verbose |
| | Default: False |

export

All variables can be exported in STDOUT using the following command: airflow variables export -

```
[37mairflow variables export[0m [[36m-h[0m] [[36m-v[0m] [36mfile[0m]
```

Positional Arguments

| | |
|-------------|-----------------------------------|
| file | Export all variables to JSON file |
|-------------|-----------------------------------|

Named Arguments

| | |
|----------------------|----------------------------------|
| -v, --verbose | Make logging output more verbose |
| | Default: False |

get

Get variable

```
[37mairflow variables get[0m [[36m-h[0m] [[36m-d[0m [36mVAL[0m] [[36m-j[0m] [[36m-v[0m]_u  
↳ [36mkey[0m
```

Positional Arguments

| | |
|------------|--------------|
| key | Variable key |
|------------|--------------|

Named Arguments

| | |
|----------------------|---|
| -d, --default | Default value returned if variable does not exist |
| -j, --json | Deserialize JSON variable |
| | Default: False |
| -v, --verbose | Make logging output more verbose |
| | Default: False |

import

Import variables

```
[37mairflow variables import[0m [[36m-h[0m] [[36m-a[0m [36m{overwrite, fail, skip}[0m]_u  
↳ [[36m-v[0m] [36mfile[0m
```

Positional Arguments

| | |
|-------------|---------------------------------|
| file | Import variables from JSON file |
|-------------|---------------------------------|

Named Arguments

| | |
|-------------------------------------|---|
| -a, --action-on-existing-key | Possible choices: overwrite, fail, skip Action to take if we encounter a variable key that already exists. Default: 'overwrite' |
| -v, --verbose | Make logging output more verbose Default: False |

list

List variables

```
[37mairflow variables list[0m [[36m-h[0m] [[36m-o[0m [36m(table, json, yaml, plain)[0m]
↳ [[36m-v[0m]
```

Named Arguments

| | |
|----------------------|--|
| -o, --output | Possible choices: table, json, yaml, plain Output format. Allowed values: json, yaml, plain, table (default: table) Default: 'table' |
| -v, --verbose | Make logging output more verbose Default: False |

set

Set variable

```
[37mairflow variables set[0m [[36m-h[0m] [[36m--description[0m [36mDESCRIPTION[0m] [[36m-
↳ j[0m] [[36m-v[0m] [36mkey[0m [36mVALUE[0m]
```

Positional Arguments

| | |
|--------------|----------------|
| key | Variable key |
| VALUE | Variable value |

Named Arguments

| | |
|----------------------|--|
| --description | Variable description, optional when setting a variable |
| -j, --json | Serialize JSON variable Default: False |
| -v, --verbose | Make logging output more verbose Default: False |

version

Show the version

```
[37mairflow version[0m [[36m-h[0m]
```

3.21.2 Environment Variables

AIRFLOW_{SECTION}__{KEY}

Sets options in the Airflow configuration. This takes priority over the value in the `airflow.cfg` file.

Replace the `{SECTION}` placeholder with any section and the `{KEY}` placeholder with any key in that specified section.

For example, if you want to set the `dags_folder` options in `[core]` section, then you should set the `AIRFLOW__CORE__DAGS_FOLDER` environment variable.

For more information, see: *Setting Configuration Options*.

AIRFLOW_{SECTION}__{KEY}_CMD

For any specific key in a section in Airflow, execute the command the key is pointing to. The result of the command is used as a value of the `AIRFLOW_{SECTION}__{KEY}` environment variable.

This is only supported by the following config options:

- `sqlalchemy_conn` in `[database]` section
- `fernet_key` in `[core]` section
- `broker_url` in `[celery]` section
- `flower_basic_auth` in `[celery]` section
- `result_backend` in `[celery]` section
- `password` in `[atlas]` section
- `smtp_password` in `[smtp]` section
- `secret_key` in `[webserver]` section

AIRFLOW_{SECTION}__{KEY}_SECRET

For any specific key in a section in Airflow, retrieve the secret from the configured secrets backend. The returned value will be used as the value of the `AIRFLOW_{SECTION}__{KEY}` environment variable.

See [Secrets Backends](#) for more information on available secrets backends.

This form of environment variable configuration is only supported for the same subset of config options as `AIRFLOW_{SECTION}__{KEY}_CMD`

AIRFLOW_CONFIG

The path to the Airflow configuration file.

AIRFLOW_CONN_{CONN_ID}

Defines a new connection with the name `{CONN_ID}` using the URI value.

For example, if you want to create a connection named `PROXY_POSTGRES_TCP`, you can create a key `AIRFLOW_CONN_PROXY_POSTGRES_TCP` with the connection URI as the value.

For more information, see: *Storing connections in environment variables*.

AIRFLOW_HOME

The root directory for the Airflow content. This is the default parent directory for Airflow assets such as dags and logs.

AIRFLOW_VAR_{KEY}

Defines an Airflow variable. Replace the {KEY} placeholder with the variable name.

For more information, see: *Managing Variables*.

3.22 Templates reference

Variables, macros and filters can be used in templates (see the *Jinja Templating* section)

The following come for free out of the box with Airflow. Additional custom macros can be added globally through *Plugins*, or at a DAG level through the `DAG.user_defined_macros` argument.

3.22.1 Variables

The Airflow engine passes a few variables by default that are accessible in all templates

| Variable | Type | Description |
|---|---|--|
| <code>{{ data_interval_start }}</code> | <code>pendulum.DateTime</code> | Start of the data interval. Added in version 2.2. |
| <code>{{ data_interval_end }}</code> | <code>pendulum.DateTime</code> | End of the data interval. Added in version 2.2. |
| <code>{{ logical_date }}</code> | <code>pendulum.DateTime</code> | A date-time that logically identifies the current DAG run. This value does not contain any semantics, but is simply a value for identification. Use <code>data_interval_start</code> and <code>data_interval_end</code> instead if you want a value that has real-world semantics, such as to get a slice of rows from the database based on timestamps. |
| <code>{{ exception }}</code> | <code>None str Exception KeyboardInterrupt</code> | Error occurred while running task instance. |
| <code>{{ prev_data_interval_start }}</code> | <code>pendulum.DateTime None</code> | Start of the data interval of the prior successful <i>DagRun</i> . Added in version 2.2. |

continues on next page

Table 107 – continued from previous page

| Variable | Type | Description |
|---|---------------------------------------|---|
| <code>{{ prev_data_interval_end_suc }}</code> | <code>pendulum.DateTime None</code> | End of the data interval of the prior successful <i>DagRun</i> . Added in version 2.2. |
| <code>{{ prev_start_date_success }}</code> | <code>pendulum.DateTime None</code> | Start date from prior successful <i>DagRun</i> (if available). |
| <code>{{ prev_end_date_success }}</code> | <code>pendulum.DateTime None</code> | End date from prior successful <i>DagRun</i> (if available). |
| <code>{{ start_date }}</code> | <code>pendulum.DateTime</code> | Datetime of when current task has been started. |
| <code>{{ inlets }}</code> | list | List of inlets declared on the task. |
| <code>{{ inlet_events }}</code> | <code>dict[str, ...]</code> | Access past events of inlet assets. See <i>Assets</i> . Added in version 2.10. |
| <code>{{ outlets }}</code> | list | List of outlets declared on the task. |
| <code>{{ outlet_events }}</code> | <code>dict[str, ...]</code> | Accessors to attach information to asset events that will be emitted by the current task. See <i>Assets</i> . Added in version 2.10. |
| <code>{{ dag }}</code> | DAG | The currently running <i>DAG</i> . You can read more about dags in <i>Dags</i> . |
| <code>{{ task }}</code> | BaseOperator | The currently running <i>BaseOperator</i> . You can read more about Tasks in <i>Operators</i> |
| <code>{{ task_reschedule_count }}</code> | int | How many times current task has been rescheduled. Relevant to mode="reschedule" sensors. |
| <code>{{ macros }}</code> | | A reference to the macros package. See <i>Macros</i> below. |
| <code>{{ task_instance }}</code> | TaskInstance | The currently running <i>TaskInstance</i> . |
| <code>{{ ti }}</code> | TaskInstance | Same as <code>{{ task_instance }}</code> . |
| <code>{{ params }}</code> | <code>dict[str, Any]</code> | The user-defined params. This can be overridden by the mapping passed to <code>trigger_dag -c</code> if <code>dag_run_conf_overrides_params</code> is enabled in <code>airflow.cfg</code> . |
| <code>{{ var.value }}</code> | | Airflow variables. See <i>Airflow Variables in Templates</i> below. |
| <code>{{ var.json }}</code> | | Airflow variables. See <i>Airflow Variables in Templates</i> below. |

continues on next page

Table 107 – continued from previous page

| Variable | Type | Description |
|--|-----------------------------|--|
| <code>{{ conn }}</code> | | Airflow connections. See <i>Airflow Connections in Templates</i> below. |
| <code>{{ task_instance_key_str }}</code> | str | A unique, human-readable key to the task instance. The format is <code>{dag_id}__{task_id}__{ds_nodash}</code> . |
| <code>{{ run_id }}</code> | str | The currently running <i>DagRun</i> run ID. |
| <code>{{ dag_run }}</code> | DagRun | The currently running <i>DagRun</i> . |
| <code>{{ test_mode }}</code> | bool | Whether the task instance was run by the <code>airflow test</code> CLI. |
| <code>{{ map_index_template }}</code> | None str | Template used to render the expanded task instance of a mapped task. Setting this value will be reflected in the rendered result. |
| <code>{{ expanded_ti_count }}</code> | int None | Number of task instances that a mapped task was expanded into. If the current task is not mapped, this should be None. Added in version 2.5. |
| <code>{{ triggering_asset_events }}</code> | dict[str, list[AssetEvent]] | If in an Asset Scheduled DAG, a map of Asset URI to a list of triggering <i>AssetEvent</i> (there may be more than one, if there are multiple Assets with different frequencies). Read more here <i>Assets</i> . Added in version 2.4. |

The following are only available when the DagRun has a `logical_date`

| Variable | Type | Description |
|--------------------------------------|------|--|
| <code>{{ ds }}</code> | str | The DAG run's logical date as YYYY-MM-DD. Same as <code>{{ logical_date ds }}</code> . |
| <code>{{ ds_nodash }}</code> | str | Same as <code>{{ logical_date ds_nodash }}</code> . |
| <code>{{ ts }}</code> | str | Same as <code>{{ logical_date ts }}</code> . Example: <code>2018-01-01T00:00:00+00:00</code> . |
| <code>{{ ts_nodash_with_tz }}</code> | str | Same as <code>{{ logical_date ts_nodash_with_tz }}</code> . Example: <code>20180101T000000+0000</code> . |
| <code>{{ ts_nodash }}</code> | str | Same as <code>{{ logical_date ts_nodash }}</code> . Example: <code>20180101T000000</code> . |

Note

The DAG run's logical date, and values derived from it, such as `ds` and `ts`, **should not** be considered unique in a DAG. Use `run_id` instead.

3.22.2 Accessing Airflow context variables from TaskFlow tasks

While `@task` decorated tasks don't support rendering jinja templates passed as arguments, all of the variables listed above can be accessed directly from tasks. The following code block is an example of accessing a `task_instance` object from its task:

```
from airflow.models.taskinstance import TaskInstance
from airflow.models.dagrun import DagRun

@task
def print_ti_info(task_instance: TaskInstance, dag_run: DagRun):
    print(f"Run ID: {task_instance.run_id}") # Run ID: scheduled_2023-08-
    ↪09T00:00:00+00:00
    print(f"Duration: {task_instance.duration}") # Duration: 0.972019
```

(continues on next page)

(continued from previous page)

```
print(f"DAG Run queued at: {dag_run.queued_at}") # 2023-08-10
→ 00:00:01+02:20
```

Note that you can access the object's attributes and methods with simple dot notation. Here are some examples of what is possible: `{{ task.owner }}`, `{{ task.task_id }}`, `{{ ti.hostname }}`, ... Refer to the models documentation for more information on the objects' attributes and methods.

3.22.3 Airflow Variables in Templates

The `var` template variable allows you to access Airflow Variables. You can access them as either plain-text or JSON. If you use JSON, you are also able to walk nested structures, such as dictionaries like: `{{ var.json.my_dict_var.key1 }}`.

It is also possible to fetch a variable by string if needed (for example your variable key contains dots) with `{{ var.value.get('my.var', 'fallback') }}` or `{{ var.json.get('my.dict.var', {'key1': 'val1'}) }}`. Defaults can be supplied in case the variable does not exist.

3.22.4 Airflow Connections in Templates

Similarly, Airflow Connections data can be accessed via the `conn` template variable. For example, you could use expressions in your templates like `{{ conn.my_conn_id.login }}`, `{{ conn.my_conn_id.password }}`, etc.

Just like with `var` it's possible to fetch a connection by string (e.g. `{{ conn.get('my_conn_id_+index).host }}`) or provide defaults (e.g. `{{ conn.get('my_conn_id', {"host": "host1", "login": "user1"}).host }}`).

Additionally, the `extras` field of a connection can be fetched as a Python Dictionary with the `extra_dejson` field, e.g. `conn.my_aws_conn_id.extra_dejson.region_name` would fetch `region_name` out of `extras`. This way, defaults in `extras` can be provided as well (e.g. `{{ conn.my_aws_conn_id.extra_dejson.get('region_name', 'Europe (Frankfurt)') }}`).

3.22.5 Filters

Airflow defines some Jinja filters that can be used to format values.

For example, using `{{ logical_date | ds }}` will output the `logical_date` in the `YYYY-MM-DD` format.

| Filter | Operates on | Description |
|--------------------------------|-----------------------|--|
| <code>ds</code> | <code>datetime</code> | Format the datetime as <code>YYYY-MM-DD</code> |
| <code>ds_nodash</code> | <code>datetime</code> | Format the datetime as <code>YYYYMMDD</code> |
| <code>ts</code> | <code>datetime</code> | Same as <code>.isoformat()</code> , Example: <code>2018-01-01T00:00:00+00:00</code> |
| <code>ts_nodash</code> | <code>datetime</code> | Same as <code>ts</code> filter without <code>-</code> , <code>:</code> or TimeZone info. Example: <code>20180101T000000</code> |
| <code>ts_nodash_with_tz</code> | <code>datetime</code> | As <code>ts</code> filter without <code>-</code> or <code>:</code> . Example <code>20180101T000000+0000</code> |

3.22.6 Macros

Macros are a way to expose objects to your templates and live under the `macros` namespace in your templates.

A few commonly used libraries and methods are made available.

| Variable | Description |
|-------------------------------|--|
| <code>macros.datetime</code> | The standard lib's <code>datetime.datetime</code> |
| <code>macros.timedelta</code> | The standard lib's <code>datetime.timedelta</code> |
| <code>macros.dateutil</code> | A reference to the <code>dateutil</code> package |
| <code>macros.time</code> | The standard lib's <code>time</code> |
| <code>macros.uuid</code> | The standard lib's <code>uuid</code> |
| <code>macros.random</code> | The standard lib's <code>random.random</code> |

Some Airflow specific macros are also defined:

`airflow.macros.random()` → x in the interval $[0, 1]$.

3.23 Airflow public API reference

It's a stub file. It will be converted automatically during the build process to the valid documentation by the Sphinx plugin. See: `airflow-core/docs/conf.py`

3.24 Configuration Reference

This page contains the list of all the available Airflow configurations that you can set in `airflow.cfg` file or using environment variables.

Use the same configuration across all the Airflow components. While each component does not require all, some configurations need to be same otherwise they would not work as expected. A good example for that is `secret_key` which should be same on the Webserver and Worker to allow Webserver to fetch logs from Worker.

The webserver key is also used to authorize requests to Celery workers when logs are retrieved. The token generated using the secret key has a short expiry time though - make sure that time on ALL the machines that you run Airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors when the logs are accessed.

 Note

For more information see *Setting Configuration Options*.

3.24.1 Provider-specific configuration options

Some of the providers have their own configuration options, you will find details of their configuration in the provider's documentation.

You can find all the provider configuration in configurations specific to providers

3.24.2 Airflow configuration options

Sections:

- *[api]*
- *[api_auth]*
- *[core]*
- *[dag_processor]*
- *[database]*
- *[email]*
- *[execution_api]*
- *[kerberos]*
- *[lineage]*
- *[logging]*
- *[metrics]*
- *[operators]*
- *[scheduler]*
- *[secrets]*
- *[sensors]*
- *[sentry]*
- *[smtp]*
- *[traces]*
- *[triggerer]*
- *[webserver]*
- *[workers]*

[api]**access_control_allow_headers**

Added in version 2.1.0.

Used in response to a preflight request to indicate which HTTP headers can be used when making the actual request. This header is the server side response to the browser's Access-Control-Request-Headers header.

Type

string

Default

''

Environment Variable

AIRFLOW__API__ACCESS_CONTROL_ALLOW_HEADERS

access_control_allow_methods

Added in version 2.1.0.

Specifies the method or methods allowed when accessing the resource.

Type

string

Default

''

Environment Variable

AIRFLOW__API__ACCESS_CONTROL_ALLOW_METHODS

access_control_allow_origins

Added in version 2.2.0.

Indicates whether the response can be shared with requesting code from the given origins. Separate URLs with space.

Type

string

Default

''

Environment Variable

AIRFLOW__API__ACCESS_CONTROL_ALLOW_ORIGINS

access_logfile

Log files for the api server. ‘-’ means log to stderr.

Type

string

Default

-

Environment Variable

AIRFLOW__API__ACCESS_LOGFILE

base_url

The base url of the API server. Airflow cannot guess what domain or CNAME you are using. If the Airflow console (the front-end) and the API server are on a different domain, this config should contain the API server endpoint.

Type

string

Default

None

Environment Variable

AIRFLOW__API__BASE_URL

Example

<https://my-airflow.company.com>

enable_xcom_deserialize_support

Added in version 2.7.0.

Indicates whether the **xcomEntries** endpoint supports the **deserialize** flag. If set to **False**, setting this flag in a request would result in a 400 Bad Request error.

Type

boolean

Default

False

Environment Variable

AIRFLOW__API__ENABLE_XCOM_DESERIALIZE_SUPPORT

expose_config

Expose the configuration file in the web server. Set to **non-sensitive-only** to show all values except those that have security implications. **True** shows all values. **False** hides the configuration completely.

Type

string

Default

False

Environment Variable

AIRFLOW__API__EXPOSE_CONFIG

fallback_page_limit

Added in version 2.0.0.

Used to set the default page limit when limit param is zero or not provided in API requests. Otherwise if positive integer is passed in the API requests as limit, the smallest number of user given limit or maximum page limit is taken as limit.

Type

integer

Default

50

Environment Variable

AIRFLOW__API__FALLBACK_PAGE_LIMIT

host

The ip specified when starting the api server

Type

string

Default

0.0.0.0

Environment Variable

AIRFLOW__API__HOST

maximum_page_limit

Added in version 2.0.0.

Used to set the maximum page limit for API requests. If limit passed as param is greater than maximum page limit, it will be ignored and maximum page limit value will be set as the limit

Type

integer

Default

100

Environment Variable

AIRFLOW__API__MAXIMUM_PAGE_LIMIT

port

The port on which to run the api server

Type

string

Default

8080

Environment Variable

AIRFLOW__API__PORT

ssl_cert

Paths to the SSL certificate and key for the api server. When both are provided SSL will be enabled. This does not change the api server port.

Type

string

Default

''

Environment Variable

AIRFLOW__API__SSL_CERT

ssl_key

Paths to the SSL certificate and key for the api server. When both are provided SSL will be enabled. This does not change the api server port.

Type

string

Default

''

Environment Variable

AIRFLOW__API__SSL_KEY

worker_timeout

Number of seconds the API server waits before timing out on a worker

Type

string

Default

120

Environment Variable

AIRFLOW__API__WORKER_TIMEOUT

workers

Number of workers to run on the API server

Type

string

Default

4

Environment Variable

AIRFLOW__API__WORKERS

[api_auth]

Settings relating to authentication on the Airflow APIs

jwt_algorithm

Added in version 3.0.0.

The algorithm name use when generating and validating JWT Task Identities.

This value must be appropriate for the given private key type.

If this is not specified Airflow makes some guesses as what algorithm is best based on the key type.

("HS512" if `jwt_secret` is set, otherwise a key-type specific guess)

Type

string

Default

None

Environment Variable

AIRFLOW__API_AUTH__JWT_ALGORITHM

Example

"EdDSA" or "HS512"

jwt_audience

Added in version 3.0.0.

The audience claim to use when generating and validating JWTs for the API.

This variable can be a single value, or a comma-separated string, in which case the first value is the one that will be used when generating, and the others are accepted at validation time.

Not required, but strongly encouraged.

See also *jwt_audience*

Type
string

Default
None

Environment Variable
AIRFLOW__API_AUTH__JWT_AUDIENCE

Example
my-unique-airflow-id

jwt_cli_expiration_time

Added in version 3.0.0.

Number in seconds until the JWTs used for authentication expires for CLI commands. When the token expires, all CLI calls using this token will fail on authentication.

Make sure that time on ALL the machines that you run airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors.

Type
integer

Default
3600

Environment Variable
AIRFLOW__API_AUTH__JWT_CLI_EXPIRATION_TIME

jwt_expiration_time

Added in version 3.0.0.

Number in seconds until the JWTs used for authentication expires. When the token expires, all API calls using this token will fail on authentication.

Make sure that time on ALL the machines that you run airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors.

See also *jwt_expiration_time*

Type
integer

Default
86400

Environment Variable
AIRFLOW__API_AUTH__JWT_EXPIRATION_TIME

jwt_issuer

Added in version 3.0.0.

Issuer of the JWT. This becomes the `iss` claim of generated tokens, and is validated on incoming requests.

Ideally this should be unique per individual airflow deployment

Not required, but strongly recommended to be set.

See also *jwt_audience*

Type
string

Default
None

Environment Variable
AIRFLOW__API_AUTH__JWT_ISSUER

Example
`http://my-airflow.mycompany.com`

jwt_kid

Added in version 3.0.0.

The Key ID to place in header when generating JWTs. Not used in the validation path.

If this is not specified the RFC7638 thumbprint of the private key will be used.

Ignored when `jwt_secret` is used.

Type
string

Default
None

Environment Variable
AIRFLOW__API_AUTH__JWT_KID

Example
`my-key-id`

jwt_leeway

Added in version 3.0.0.

Number of seconds leeway in validating expiry time of JWTs to account for clock skew between client and server

Type
integer

Default
10

Environment Variable
AIRFLOW__API_AUTH__JWT_LEEWAY

jwt_private_key_path

Added in version 3.0.0.

The path to a file containing a PEM-encoded private key use when generating Task Identity tokens in the executor.

Mutually exclusive with `jwt_secret`.

Type
string

Default

None

Environment Variable

AIRFLOW__API_AUTH__JWT_PRIVATE_KEY_PATH

Example

/path/to/private_key.pem

jwt_secret

Added in version 3.0.0.

Secret key used to encode and decode JWTs to authenticate to public and private APIs.

It should be as random as possible. However, when running more than 1 instances of API services, make sure all of them use the same `jwt_secret` otherwise calls will fail on authentication.

Mutually exclusive with `jwt_private_key_path`.

Type

string

Default

{JWT_SECRET_KEY}

Environment Variables

AIRFLOW__API_AUTH__JWT_SECRET

AIRFLOW__API_AUTH__JWT_SECRET_CMD

AIRFLOW__API_AUTH__JWT_SECRET_SECRET

trusted_jwks_url

Added in version 3.0.0.

The public signing keys of Task Execution token issuers to trust. It must contain the public key related to `jwt_private_key_path` else tasks will be unlikely to execute successfully.

Can be a local file path (without the `file://` prefix) or an http or https URL.

If a remote URL is given it will be polled periodically for changes.

Mutually exclusive with `jwt_secret`.

If a `jwt_private_key_path` is given but this setting is not set then the private key will be trusted. If this is provided it is your responsibility to ensure that the private key used for generation is in this list.

Type

string

Default

None

Environment Variable

AIRFLOW__API_AUTH__TRUSTED_JWKS_URL

Example

"`/path/to/public-jwks.json`" or "`https://my-issuer/.well-known/jwks.json`"

[core]**allowed_deserialization_classes**

Added in version 2.5.0.

Space-separated list of classes that may be imported during deserialization. Items can be glob expressions. Python built-in classes (like dict) are always allowed.

Type

string

Default

airflow.*

Environment Variable

AIRFLOW__CORE__ALLOWED_DESERIALIZATION_CLASSES

Example

airflow.* my_mod.my_other_mod.TheseClasses*

allowed_deserialization_classes_regex

Added in version 2.8.2.

Space-separated list of classes that may be imported during deserialization. Items are processed as regex expressions. Python built-in classes (like dict) are always allowed. This is a secondary option to [core] allowed_deserialization_classes.

Type

string

Default

''

Environment Variable

AIRFLOW__CORE__ALLOWED_DESERIALIZATION_CLASSES_REGEX

asset_manager_class

Added in version 3.0.0.

Class to use as asset manager.

Type

string

Default

None

Environment Variable

AIRFLOW__CORE__ASSET_MANAGER_CLASS

Example

airflow.assets.manager.AssetManager

asset_manager_kwargs

Added in version 3.0.0.

Kwargs to supply to asset manager.

Type
string

Default
None

Environment Variables

AIRFLOW__CORE__ASSET_MANAGER_KWARGS
AIRFLOW__CORE__ASSET_MANAGER_KWARGS_CMD
AIRFLOW__CORE__ASSET_MANAGER_KWARGS_SECRET

Example

```
{"some_param": "some_value"}
```

auth_manager

Added in version 2.7.0.

The auth manager class that airflow should use. Full import path to the auth manager class.

Type
string

Default
airflow.api_fastapi.auth.managers.simple.simple_auth_manager.
SimpleAuthManager

Environment Variable

AIRFLOW__CORE__AUTH_MANAGER

compress_serialized_dags

Added in version 2.3.0.

If True, serialized DAGs are compressed before writing to DB.

 **Note**

This will disable the DAG dependencies view

Type
string

Default
False

Environment Variable

AIRFLOW__CORE__COMPRESS_SERIALIZED_DAGS

daemon_umask

Added in version 2.3.4.

The default umask to use for process when run in daemon mode (scheduler, worker, etc.)

This controls the file-creation mode mask which determines the initial value of file permission bits for newly created files.

This value is treated as an octal-integer.

Type
string

Default
0o077

Environment Variable
AIRFLOW__CORE__DAEMON_UMASK

dag_discovery_safe_mode

Added in version 1.10.3.

If enabled, Airflow will only scan files containing both DAG and airflow (case-insensitive).

Type
string

Default
True

Environment Variable
AIRFLOW__CORE__DAG_DISCOVERY_SAFE_MODE

dag_ignore_file_syntax

Added in version 2.3.0.

The pattern syntax used in the .airflowignore files in the DAG directories. Valid values are regexp or glob.

Type
string

Default
glob

Environment Variable
AIRFLOW__CORE__DAG_IGNORE_FILE_SYNTAX

dag_run_conf_overrides_params

Whether to override params with dag_run.conf. If you pass some key-value pairs through airflow dags backfill -c or airflow dags trigger -c, the key-value pairs will override the existing ones in params.

Type
string

Default
True

Environment Variable
AIRFLOW__CORE__DAG_RUN_CONF_OVERRIDES_PARAMS

dagbag_import_error_traceback_depth

Added in version 2.0.0.

If tracebacks are shown, how many entries from the traceback should be shown

Type
integer

Default

2

Environment Variable

AIRFLOW__CORE__DAGBAG_IMPORT_ERROR_TRACEBACK_DEPTH

dagbag_import_error_tracebacks

Added in version 2.0.0.

Should a traceback be shown in the UI for dagbag import errors, instead of just the exception message

Type

boolean

Default

True

Environment Variable

AIRFLOW__CORE__DAGBAG_IMPORT_ERROR_TRACEBACKS

dagbag_import_timeout

How long before timing out a python file import

Type

float

Default

30.0

Environment Variable

AIRFLOW__CORE__DAGBAG_IMPORT_TIMEOUT

dags_are_paused_at_creation

Are DAGs paused by default at creation

Type

string

Default

True

Environment Variable

AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION

dags_folder

The folder where your airflow pipelines live, most likely a subfolder in a code repository. This path must be absolute.

Type

string

Default

{AIRFLOW_HOME}/dags

Environment Variable

AIRFLOW__CORE__DAGS_FOLDER

default_ impersonation

If set, tasks without a `run_as_user` argument will be run with this user Can be used to de-elevate a sudo user running Airflow when executing tasks

Type
string

Default
''

Environment Variable

AIRFLOW__CORE__DEFAULT_IMPERSONATION

default_pool_task_slot_count

Added in version 2.2.0.

Task Slot counts for `default_pool`. This setting would not have any effect in an existing deployment where the `default_pool` is already created. For existing deployments, users can change the number of slots using Webserver, API or the CLI

Type
string

Default
128

Environment Variable

AIRFLOW__CORE__DEFAULT_POOL_TASK_SLOT_COUNT

default_task_execution_timeout

Added in version 2.3.0.

The default task `execution_timeout` value for the operators. Expected an integer value to be passed into `timedelta` as seconds. If not specified, then the value is considered as None, meaning that the operators are never timed out by default.

Type
integer

Default
''

Environment Variable

AIRFLOW__CORE__DEFAULT_TASK_EXECUTION_TIMEOUT

default_task_retries

Added in version 1.10.6.

The number of retries each task is going to have by default. Can be overridden at dag or task level.

Type
string

Default
0

Environment Variable

AIRFLOW__CORE__DEFAULT_TASK_RETRIES

default_task_retry_delay

Added in version 2.4.0.

The number of seconds each task is going to wait by default between retries. Can be overridden at dag or task level.

Type

integer

Default

300

Environment Variable

AIRFLOW__CORE__DEFAULT_TASK_RETRY_DELAY

default_task_weight_rule

Added in version 2.2.0.

The weighting method used for the effective total priority weight of the task

Type

string

Default

downstream

Environment Variable

AIRFLOW__CORE__DEFAULT_TASK_WEIGHT_RULE

default_timezone

Default timezone in case supplied date times are naive can be *UTC* (default), *system*, or any *IANA* <<https://www.iana.org/time-zones>> timezone string (e.g. Europe/Amsterdam)

Type

string

Default

utc

Environment Variable

AIRFLOW__CORE__DEFAULT_TIMEZONE

execute_tasks_new_python_interpreter

Added in version 2.0.0.

Should tasks be executed via forking of the parent process

- **False:** Execute via forking of the parent process
- **True:** Spawning a new python process, slower than fork, but means plugin changes picked up by tasks straight away

See also

When are plugins (re)loaded?

Type

boolean

Default
False

Environment Variable
AIRFLOW__CORE__EXECUTE_TASKS_NEW_PYTHON_INTERPRETER

execution_api_server_url

Added in version 3.0.0.

The url of the execution api server. Default is {BASE_URL}/execution/ where {BASE_URL} is the base url of the API Server. If {BASE_URL} is not set, it will use http://localhost:8080 as the default base url.

Type
string

Default
None

Environment Variable
AIRFLOW__CORE__EXECUTION_API_SERVER_URL

executor

The executor class that airflow should use. Choices include LocalExecutor, CeleryExecutor, KubernetesExecutor or the full import path to the class when using a custom executor.

Type
string

Default
LocalExecutor

Environment Variable
AIRFLOW__CORE__EXECUTOR

fernet_key

Secret key to save connection passwords in the db

Type
string

Default
{FERNET_KEY}

Environment Variables
AIRFLOW__CORE__FERNET_KEY
AIRFLOW__CORE__FERNET_KEY_CMD
AIRFLOW__CORE__FERNET_KEY_SECRET

hide_sensitive_var_conn_fields

Added in version 2.1.0.

Hide sensitive **Variables** or **Connection extra json keys** from UI and task logs when set to True

 **Note**

Connection passwords are always hidden in logs

Type

boolean

Default

True

Environment Variable

AIRFLOW__CORE__HIDE_SENSITIVE_VAR_CONN_FIELDS

hostname_callable

Hostname by providing a path to a callable, which will resolve the hostname. The format is “package.function”.

For example, default value `airflow.utils.net.getfqdn` means that result from patched version of `socket.getfqdn()`, see related [CPython Issue](#).

No argument should be required in the function specified. If using IP address as hostname is preferred, use value `airflow.utils.net.get_host_ip_address`

Type

string

Default

`airflow.utils.net.getfqdn`

Environment Variable

AIRFLOW__CORE__HOSTNAME_CALLABLE

killed_task_cleanup_time

When a task is killed forcefully, this is the amount of time in seconds that it has to cleanup after it is sent a SIGTERM, before it is SIGKILLED

Type

string

Default

60

Environment Variable

AIRFLOW__CORE__KILLED_TASK_CLEANUP_TIME

lazy_discover_providers

Added in version 2.0.0.

By default Airflow providers are lazily-discovered (discovery and imports happen only when required). Set it to `False`, if you want to discover providers whenever ‘airflow’ is invoked via cli or loaded from module.

Type

boolean

Default

True

Environment Variable

AIRFLOW__CORE__LAZY_DISCOVER_PROVIDERS

lazy_load_plugins

Added in version 2.0.0.

By default Airflow plugins are lazily-loaded (only loaded when required). Set it to `False`, if you want to load plugins whenever ‘airflow’ is invoked via cli or loaded from module.

Type

boolean

Default

True

Environment Variable

AIRFLOW__CORE__LAZY_LOAD_PLUGINS

load_examples

Whether to load the DAG examples that ship with Airflow. It’s good to get started, but you probably want to set this to `False` in a production environment

Type

string

Default

True

Environment Variable

AIRFLOW__CORE__LOAD_EXAMPLES

max_active_runs_per_dag

The maximum number of active DAG runs per DAG. The scheduler will not create more DAG runs if it reaches the limit. This is configurable at the DAG level with `max_active_runs`, which is defaulted as [core] `max_active_runs_per_dag`.

Type

string

Default

16

Environment Variable

AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG

max_active_tasks_per_dag

Added in version 2.2.0.

The maximum number of task instances allowed to run concurrently in each DAG. To calculate the number of tasks that is running concurrently for a DAG, add up the number of running tasks for all DAG runs of the DAG. This is configurable at the DAG level with `max_active_tasks`, which is defaulted as [core] `max_active_tasks_per_dag`.

An example scenario when this would be useful is when you want to stop a new dag with an early start date from stealing all the executor slots in a cluster.

Type

string

Default

16

Environment Variable

AIRFLOW__CORE__MAX_ACTIVE_TASKS_PER_DAG

max_consecutive_failed_dag_runs_per_dag

Added in version 2.9.0.

(experimental) The maximum number of consecutive DAG failures before DAG is automatically paused. This is also configurable per DAG level with `max_consecutive_failed_dag_runs`, which is defaulted as [core] `max_consecutive_failed_dag_runs_per_dag`. If not specified, then the value is considered as 0, meaning that the dags are never paused out by default.

Type

string

Default

0

Environment Variable

AIRFLOW__CORE__MAX_CONSECUTIVE_FAILED_DAG_RUNS_PER_DAG

max_map_length

Added in version 2.3.0.

The maximum list/dict length an XCom can push to trigger task mapping. If the pushed list/dict has a length exceeding this value, the task pushing the XCom will be failed automatically to prevent the mapped tasks from clogging the scheduler.

Type

integer

Default

1024

Environment Variable

AIRFLOW__CORE__MAX_MAP_LENGTH

max_num_rendered_ti_fields_per_task

Added in version 1.10.10.

Maximum number of Rendered Task Instance Fields (Template Fields) per task to store in the Database. All the template_fields for each of Task Instance are stored in the Database. Keeping this number small may cause an error when you try to view Rendered tab in TaskInstance view for older tasks.

Type

integer

Default

30

Environment Variable

AIRFLOW__CORE__MAX_NUM_RENDERED_TI_FIELDS_PER_TASK

max_task_retry_delay

Added in version 2.6.0.

The maximum delay (in seconds) each task is going to wait by default between retries. This is a global setting and cannot be overridden at task or DAG level.

Type

integer

Default

86400

Environment Variable

AIRFLOW__CORE__MAX_TASK_RETRY_DELAY

max_templated_field_length

Added in version 2.9.0.

The maximum length of the rendered template field. If the value to be stored in the rendered template field exceeds this size, it's redacted.

Type

integer

Default

4096

Environment Variable

AIRFLOW__CORE__MAX_TEMPLATED_FIELD_LENGTH

might_contain_dag_callable

Added in version 2.6.0.

A callable to check if a python file has airflow dags defined or not and should return True if it has dags otherwise False. If this is not provided, Airflow uses its own heuristic rules.

The function should have the following signature

```
def func_name(file_path: str, zip_file: zipfile.ZipFile | None = None) -> bool: ...
```

Type

string

Default

airflow.utils.file.might_contain_dag_via_default_heuristic

Environment Variable

AIRFLOW__CORE__MIGHT_CONTAIN_DAG_CALLABLE

min_serialized_dag_fetch_interval

Added in version 1.10.12.

Fetching serialized DAG can not be faster than a minimum interval to reduce database read rate. This config controls when your DAGs are updated in the Webserver

Type

string

Default

10

Environment Variable

AIRFLOW__CORE__MIN_SERIALIZED_DAG_FETCH_INTERVAL

min_serialized_dag_update_interval

Added in version 1.10.7.

Updating serialized DAG can not be faster than a minimum interval to reduce database write rate.

Type

string

Default

30

Environment Variable

AIRFLOW__CORE__MIN_SERIALIZED_DAG_UPDATE_INTERVAL

mp_start_method

Added in version 2.0.0.

The name of the method used in order to start Python processes via the multiprocessing module. This corresponds directly with the options available in the Python docs: `multiprocessing.set_start_method` must be one of the values returned by `multiprocessing.get_all_start_methods()`.

Type

string

Default

None

Environment Variable

AIRFLOW__CORE__MP_START_METHOD

Example

fork

parallelism

This defines the maximum number of task instances that can run concurrently per scheduler in Airflow, regardless of the worker count. Generally this value, multiplied by the number of schedulers in your cluster, is the maximum number of task instances with the running state in the metadata database. The value must be larger or equal 1.

Type

string

Default

32

Environment Variable

AIRFLOW__CORE__PARALLELISM

plugins_folder

Path to the folder containing Airflow plugins

Type

string

Default

{AIRFLOW_HOME}/plugins

Environment Variable

AIRFLOW__CORE__PLUGINS_FOLDER

security

What security module to use (for example kerberos)

Type

string

Default

''

Environment Variable

AIRFLOW__CORE__SECURITY

sensitive_var_conn_names

Added in version 2.1.0.

A comma-separated list of extra sensitive keywords to look for in variables names or connection's extra JSON.

Type

string

Default

''

Environment Variable

AIRFLOW__CORE__SENSITIVE_VAR_CONN_NAMES

simple_auth_manager_all_admins

Added in version 3.0.0.

Whether to disable authentication and allow everyone as admin in the environment.

Type

string

Default

False

Environment Variable

AIRFLOW__CORE__SIMPLE_AUTH_MANAGER_ALL_ADMIN

simple_auth_manager_passwords_file

Added in version 3.0.0.

The json file where the simple auth manager stores passwords for the configured users. By default this is AIRFLOW_HOME/simple_auth_manager_passwords.json.generated.

Type
string

Default
None

Environment Variable

AIRFLOW__CORE__SIMPLE_AUTH_MANAGER_PASSWORDS_FILE

Example
/path/to/passwords.json

simple_auth_manager_users

Added in version 3.0.0.

The list of users and their associated role in simple auth manager. If the simple auth manager is used in your environment, this list controls who can access the environment.

List of user-role delimited with a comma. Each user-role is a colon delimited couple of username and role. Roles are predefined in simple auth managers: viewer, user, op, admin.

Type
string

Default
admin:admin

Environment Variable

AIRFLOW__CORE__SIMPLE_AUTH_MANAGER_USERS

Example
bob:admin,peter:viewer

task_success_overtime

Added in version 2.10.0.

Maximum possible time (in seconds) that task will have for execution of auxiliary processes (like listeners, mini scheduler...) after task is marked as success..

Type
integer

Default
20

Environment Variable

AIRFLOW__CORE__TASK_SUCCESS_OVERTIME

test_connection

Added in version 2.7.0.

The ability to allow testing connections across Airflow UI, API and CLI. Supported options: Disabled, Enabled, Hidden. Default: Disabled
Disabled - Disables the test connection functionality and disables the Test Connection button in UI.
Enabled - Enables the test connection functionality and shows the Test Connection button in UI.
Hidden - Disables the test connection functionality and hides the Test Connection button in UI.
Before setting this to Enabled, make sure that you review the users who are able to add/edit connections and ensure they are trusted. Connection testing can be done maliciously leading to undesired and insecure outcomes. See [Airflow Security Model: Capabilities of authenticated UI users](#) for more details.

Type
string

Default
Disabled

Environment Variable
`AIRFLOW__CORE__TEST_CONNECTION`

unit_test_mode

Turn unit test mode on (overwrites many configuration options with test values at runtime)

Type
string

Default
False

Environment Variable
`AIRFLOW__CORE__UNIT_TEST_MODE`

xcom_backend

Added in version 1.10.12.

Path to custom XCom class that will be used to store and resolve operators results

Type
string

Default
`airflow.sdk.execution_time.xcom.BaseXCom`

Environment Variable
`AIRFLOW__CORE__XCOM_BACKEND`

Example
`path.to.CustomXCom`

[dag_processor]

Configuration for the Airflow DAG processor. This includes, for example:

- DAG bundles, which allows Airflow to load DAGs from different sources
- **Parsing configuration, like:**
 - how often to refresh DAGs from those sources
 - how many files to parse concurrently

bundle_refresh_check_interval

How often the DAG processor should check if any DAG bundles are ready for a refresh, either by hitting the bundles `refresh_interval` or because another DAG processor has seen a newer version of the bundle. A low value means we check more frequently, and have a smaller window of time where DAG processors are out of sync with each other, parsing different versions of the same bundle.

Type
integer

Default

5

Environment Variable

AIRFLOW__DAG_PROCESSOR__BUNDLE_REFRESH_CHECK_INTERVAL

dag_bundle_config_list

Added in version 3.0.0.

List of backend configs. Must supply name, classpath, and kwargs for each backend.

By default, refresh_interval is set to [dag_processor] refresh_interval, but that can also be overridden in kwargs if desired.

The default is the dags folder dag bundle.

Note: As shown below, you can split your json config over multiple lines by indenting. See configparser documentation for an example: <https://docs.python.org/3/library/configparser.html#supported-ini-file-structure>.

Type

string

Default

```
[  
 {  
   "name": "dags-folder",  
   "classpath": "airflow.dag_processing.bundles.local.LocalDagBundle",  
   "kwargs": {}  
 }  
]
```

Environment Variable

AIRFLOW__DAG_PROCESSOR__DAG_BUNDLE_CONFIG_LIST

Example

```
[  
 {  
   "name": "my-git-repo",  
   "classpath": "airflow.providers.git.bundles.git.GitDagBundle",  
   "kwargs": {  
     "subdir": "dags",  
     "tracking_ref": "main",  
     "refresh_interval": 0  
   }  
 }  
]
```

dag_bundle_storage_path

Added in version 3.0.0.

String path to folder where Airflow bundles can store files locally. Not templated. If no path is provided, Airflow will use Path(tempfile.gettempdir()) / "airflow". This path must be absolute.

Type

string

Default

None

Environment Variable

AIRFLOW__DAG_PROCESSOR__DAG_BUNDLE_STORAGE_PATH

Example

/tmp/some-place

dag_file_processor_timeout

How long before timing out a DagFileProcessor, which processes a dag file

Type

string

Default

50

Environment Variable

AIRFLOW__DAG_PROCESSOR__DAG_FILE_PROCESSOR_TIMEOUT

disable_bundle_versioning

Always run tasks with the latest code. If set to True, the bundle version will not be stored on the dag run and therefore, the latest code will always be used.

Type

boolean

Default

False

Environment Variable

AIRFLOW__DAG_PROCESSOR__DISABLE_BUNDLE_VERSIONING

file_parsing_sort_mode

One of `modified_time`, `random_seeded_by_host` and `alphabetical`. The DAG processor will list and sort the dag files to decide the parsing order.

- `modified_time`: Sort by modified time of the files. This is useful on large scale to parse the recently modified DAGs first.
- `random_seeded_by_host`: Sort randomly across multiple DAG processors but with same order on the same host, allowing each processor to parse the files in a different order.
- `alphabetical`: Sort by filename

Type

string

Default

modified_time

Environment Variable

AIRFLOW__DAG_PROCESSOR__FILE_PARSING_SORT_MODE

max_callbacks_per_loop

The maximum number of callbacks that are fetched during a single loop.

Type
integer

Default
20

Environment Variable

AIRFLOW__DAG_PROCESSOR__MAX_CALLBACKS_PER_LOOP

min_file_process_interval

Number of seconds after which a DAG file is parsed. The DAG file is parsed every [dag_processor] min_file_process_interval number of seconds. Updates to DAGs are reflected after this interval. Keeping this number low will increase CPU usage.

Type
integer

Default
30

Environment Variable

AIRFLOW__DAG_PROCESSOR__MIN_FILE_PROCESS_INTERVAL

parsing_processes

The DAG processor can run multiple processes in parallel to parse dags. This defines how many processes will run.

Type
integer

Default
2

Environment Variable

AIRFLOW__DAG_PROCESSOR__PARSING_PROCESSES

print_stats_interval

How often should DAG processor stats be printed to the logs. Setting to 0 will disable printing stats

Type
integer

Default
30

Environment Variable

AIRFLOW__DAG_PROCESSOR__PRINT_STATS_INTERVAL

refresh_interval

How often (in seconds) to refresh, or look for new files, in a DAG bundle.

Type
integer

Default

300

Environment Variable

AIRFLOW__DAG_PROCESSOR__REFRESH_INTERVAL

stale_bundle_cleanup_age_threshold

Bundle versions used more recently than this threshold will not be removed. Recency of use is determined by when the task began running on the worker, that age is compared with this setting, given as time delta in seconds.

Type

integer

Default

21600

Environment Variable

AIRFLOW__DAG_PROCESSOR__STALE_BUNDLE_CLEANUP_AGE_THRESHOLD

stale_bundle_cleanup_interval

On shared workers, bundle copies accumulate in local storage as tasks run and version of the bundle changes. This setting represents the delta in seconds between checks for these stale bundles. Bundles which are older than *stale_bundle_cleanup_age_threshold* may be removed. But we always keep *stale_bundle_cleanup_min_versions* versions locally. Set to 0 or negative to disable.

Type

integer

Default

1800

Environment Variable

AIRFLOW__DAG_PROCESSOR__STALE_BUNDLE_CLEANUP_INTERVAL

stale_bundle_cleanup_min_versions

Minimum number of local bundle versions to retain on disk. Local bundle versions older than *stale_bundle_cleanup_age_threshold* will only be deleted we have more than *stale_bundle_cleanup_min_versions* versions accumulated on the worker.

Type

integer

Default

10

Environment Variable

AIRFLOW__DAG_PROCESSOR__STALE_BUNDLE_CLEANUP_MIN VERSIONS

stale_dag_threshold

How long (in seconds) to wait after we have re-parsed a DAG file before deactivating stale DAGs (DAGs which are no longer present in the expected files). The reason why we need this threshold is to account for the time between when the file is parsed and when the DAG is loaded. The absolute maximum that this could take is [dag_processor] *dag_file_processor_timeout*, but when you have a long timeout configured, it results in a significant delay in the deactivation of stale dags.

Type
integer

Default
50

Environment Variable

AIRFLOW__DAG_PROCESSOR__STALE_DAG_THRESHOLD

[database]

alembic_ini_file_path

Added in version 2.7.0.

Path to the `alembic.ini` file. You can either provide the file path relative to the Airflow home directory or the absolute path if it is located elsewhere.

Type
string

Default
`alembic.ini`

Environment Variable

AIRFLOW__DATABASE__ALEMBIC_INI_FILE_PATH

check_migrations

Added in version 2.6.0.

Whether to run alembic migrations during Airflow start up. Sometimes this operation can be expensive, and the users can assert the correct version through other means (e.g. through a Helm chart). Accepts True or False.

Type
string

Default
True

Environment Variable

AIRFLOW__DATABASE__CHECK.Migrations

external_db_managers

Added in version 3.0.0.

List of DB managers to use to migrate external tables in airflow database. The managers must inherit from `BaseDBManager`. If `FabAuthManager` is configured in the environment, `airflow.providers.fab.auth_manager.models.db.FABDBManager` is automatically added.

Type
string

Default
None

Environment Variable

AIRFLOW__DATABASE__EXTERNAL_DB_MANAGERS

Example

`airflow.providers.fab.auth_manager.models.db.FABDBManager`

max_db_retries

Added in version 2.3.0.

Number of times the code should be retried in case of DB Operational Errors. Not all transactions will be retried as it can cause undesired state. Currently it is only used in `DagFileProcessor.process_file` to retry `dagbag.sync_to_db`.

Type

integer

Default

3

Environment Variable

AIRFLOW__DATABASE__MAX_DB_RETRIES

migration_batch_size

Added in version 3.0.0.

The number of rows to process in each batch when performing a migration. This is useful for large tables to avoid locking and failure due to query timeouts.

Type

integer

Default

10000

Environment Variable

AIRFLOW__DATABASE__MIGRATION_BATCH_SIZE

sql_alchemy_conn

Added in version 2.3.0.

The SQLAlchemy connection string to the metadata database. SQLAlchemy supports many different database engines. See: [Set up a Database Backend: Database URI](#) for more details.

Type

string

Default

sqlite:///{{AIRFLOW_HOME}}/airflow.db

Environment Variables

AIRFLOW__DATABASE__SQLALCHEMY_CONN

AIRFLOW__DATABASE__SQLALCHEMY_CONN_CMD

AIRFLOW__DATABASE__SQLALCHEMY_CONN_SECRET

sql_alchemy_connect_args

Added in version 2.3.0.

Import path for connect args in SQLAlchemy. Defaults to an empty dict. This is useful when you want to configure db engine args that SQLAlchemy won't parse in connection string. This can be set by passing a dictionary containing the create engine parameters. For more details about passing create engine parameters (keepalives variables, timeout etc) in Postgres DB Backend see [Setting up a PostgreSQL Database](#) e.g `connect_args={"timeout":30}` can be defined in `airflow_local_settings.py` and can be imported as shown below

Type
string

Default
None

Environment Variable
AIRFLOW__DATABASE__SQL_ALCHEMY_CONNECT_ARGS

Example
airflow_local_settings.connect_args

sql_alchemy_engine_args

Added in version 2.3.0.

Extra engine specific keyword args passed to SQLAlchemy's create_engine, as a JSON-encoded value

Type
string

Default
None

Environment Variables
AIRFLOW__DATABASE__SQL_ALCHEMY_ENGINE_ARGS
AIRFLOW__DATABASE__SQL_ALCHEMY_ENGINE_ARGS_CMD
AIRFLOW__DATABASE__SQL_ALCHEMY_ENGINE_ARGS_SECRET

Example
{"arg1": true}

sql_alchemy_max_overflow

Added in version 2.3.0.

The maximum overflow size of the pool. When the number of checked-out connections reaches the size set in pool_size, additional connections will be returned up to this limit. When those additional connections are returned to the pool, they are disconnected and discarded. It follows then that the total number of simultaneous connections the pool will allow is **pool_size + max_overflow**, and the total number of “sleeping” connections the pool will allow is **pool_size**. max_overflow can be set to -1 to indicate no overflow limit; no limit will be placed on the total number of concurrent connections. Defaults to 10.

Type
string

Default
10

Environment Variable
AIRFLOW__DATABASE__SQL_ALCHEMY_MAX_OVERFLOW

sql_alchemy_pool_enabled

Added in version 2.3.0.

If SQLAlchemy should pool database connections.

Type
string

Default

True

Environment Variable

AIRFLOW__DATABASE__SQL_ALCHEMY_POOL_ENABLED

sql_alchemy_pool_pre_ping

Added in version 2.3.0.

Check connection at the start of each connection pool checkout. Typically, this is a simple statement like “SELECT 1”. See [SQLAlchemy Pooling: Disconnect Handling - Pessimistic](#) for more details.

Type

string

Default

True

Environment Variable

AIRFLOW__DATABASE__SQL_ALCHEMY_POOL_PRE_PING

sql_alchemy_pool_recycle

Added in version 2.3.0.

The SQLAlchemy pool recycle is the number of seconds a connection can be idle in the pool before it is invalidated. This config does not apply to sqlite. If the number of DB connections is ever exceeded, a lower config value will allow the system to recover faster.

Type

string

Default

1800

Environment Variable

AIRFLOW__DATABASE__SQL_ALCHEMY_POOL_RECYCLE

sql_alchemy_pool_size

Added in version 2.3.0.

The SQLAlchemy pool size is the maximum number of database connections in the pool. 0 indicates no limit.

Type

string

Default

5

Environment Variable

AIRFLOW__DATABASE__SQL_ALCHEMY_POOL_SIZE

sql_alchemy_schema

Added in version 2.3.0.

The schema to use for the metadata database. SQLAlchemy supports databases with the concept of multiple schemas.

Type

string

Default

''

Environment Variable

AIRFLOW__DATABASE__SQL_ALCHEMY_SCHEMA

sql_alchemy_session_maker

Added in version 2.10.0.

Important Warning: Use of `sql_alchemy_session_maker` Highly Discouraged Import path for function which returns '`sqlalchemy.orm.sessionmaker`'. Improper configuration of `sql_alchemy_session_maker` can lead to serious issues, including data corruption, unrecoverable application crashes. Please review the SQLAlchemy documentation for detailed guidance on proper configuration and best practices.

Type

string

Default

None

Environment Variable

AIRFLOW__DATABASE__SQLALCHEMY_SESSION_MAKER

Example

`airflow_local_settings._sessionmaker`

sql_engine_collation_for_ids

Added in version 2.3.0.

Collation for `dag_id`, `task_id`, `key`, `external_executor_id` columns in case they have different encoding. By default this collation is the same as the database collation, however for `mysql` and `mariadb` the default is `utf8mb3_bin` so that the index sizes of our index keys will not exceed the maximum size of allowed index when collation is set to `utf8mb4` variant, see [GitHub Issue Comment](#) for more details.

Type

string

Default

None

Environment Variable

AIRFLOW__DATABASE__SQL_ENGINE_COLLATION_FOR_IDS

sql_engine_encoding

Added in version 2.3.0.

The encoding for the databases

Type

string

Default

`utf-8`

Environment Variable

AIRFLOW__DATABASE__SQL_ENGINE_ENCODING

[email]

Configuration email backend and whether to send email alerts on retry or failure

`default_email_on_failure`

Added in version 2.0.0.

Whether email alerts should be sent when a task failed

Type

boolean

Default

True

Environment Variable

AIRFLOW__EMAIL__DEFAULT_EMAIL_ON_FAILURE

`default_email_on_retry`

Added in version 2.0.0.

Whether email alerts should be sent when a task is retried

Type

boolean

Default

True

Environment Variable

AIRFLOW__EMAIL__DEFAULT_EMAIL_ON_RETRY

`email_backend`

Email backend to use

Type

string

Default

airflow.utils.email.send_email_smtp

Environment Variable

AIRFLOW__EMAIL__EMAIL_BACKEND

`email_conn_id`

Added in version 2.1.0.

Email connection to use

Type

string

Default

smtp_default

Environment Variable

AIRFLOW__EMAIL__EMAIL_CONN_ID

from_email

Added in version 2.2.4.

Email address that will be used as sender address. It can either be raw email or the complete address in a format Sender Name <sender@email.com>

Type

string

Default

None

Environment Variable

AIRFLOW__EMAIL__FROM_EMAIL

Example

Airflow <airflow@example.com>

html_content_template

Added in version 2.0.1.

File that will be used as the template for Email content (which will be rendered using Jinja2). If not set, Airflow uses a base template.

See also

Email Configuration

Type

string

Default

None

Environment Variable

AIRFLOW__EMAIL__HTML_CONTENT_TEMPLATE

Example

/path/to/my_html_content_template_file

ssl_context

Added in version 2.7.0.

ssl context to use when using SMTP and IMAP SSL connections. By default, the context is “default” which sets it to `ssl.create_default_context()` which provides the right balance between compatibility and security, it however requires that certificates in your operating system are updated and that SMTP/IMAP servers of yours have valid certificates that have corresponding public keys installed on your machines. You can switch it to “none” if you want to disable checking of the certificates, but it is not recommended as it allows MITM (man-in-the-middle) attacks if your infrastructure is not sufficiently secured. It should only be set temporarily while you are fixing your certificate configuration. This can be typically done by upgrading to newer version of the operating system you run Airflow components on, by upgrading/refreshing proper certificates in the OS or by updating certificates for your mail servers.

Type

string

Default

default

Environment Variable

AIRFLOW__EMAIL__SSL_CONTEXT

Example

default

subject_template

Added in version 2.0.1.

File that will be used as the template for Email subject (which will be rendered using Jinja2). If not set, Airflow uses a base template.

 See also*Email Configuration***Type**

string

Default

None

Environment Variable

AIRFLOW__EMAIL__SUBJECT_TEMPLATE

Example

/path/to/my_subject_template_file

[execution_api]

Settings related to the Execution API server.

The ExecutionAPI also uses a lot of settings from the *[api_auth]* section.

jwt_audience

Added in version 3.0.0.

The audience claim to use when generating and validating JWTs for the Execution API.

This variable can be a single value, or a comma-separated string, in which case the first value is the one that will be used when generating, and the others are accepted at validation time.

Not required, but strongly encouraged

See also *jwt_audience*

Type

string

Default

urn:airflow.apache.org:task

Environment Variable

AIRFLOW__EXECUTION_API__JWT_AUDIENCE

jwt_expiration_time

Added in version 3.0.0.

Number in seconds until the JWT used for authentication expires. When the token expires, all API calls using this token will fail on authentication.

Make sure that time on ALL the machines that you run airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors.

Type

integer

Default

600

Environment Variable

AIRFLOW__EXECUTION_API__JWT_EXPIRATION_TIME

[kerberos]

ccache

Location of your ccache file once kinit has been performed.

Type

string

Default

/tmp/airflow_krb5_ccache

Environment Variable

AIRFLOW__KERBEROS__CCACHE

forwardable

Added in version 2.2.0.

Allow to disable ticket forwardability.

Type

boolean

Default

True

Environment Variable

AIRFLOW__KERBEROS__FORWARDABLE

include_ip

Added in version 2.2.0.

Allow to remove source IP from token, useful when using token behind NATted Docker host.

Type

boolean

Default

True

Environment Variable

AIRFLOW__KERBEROS__INCLUDE_IP

keytab

Designates the path to the Kerberos keytab file for the Airflow user

Type
string

Default
airflow.keytab

Environment Variable
AIRFLOW__KERBEROS__KEYTAB

kinit_path

Path to the kinit executable

Type
string

Default
kinit

Environment Variable
AIRFLOW__KERBEROS__KINIT_PATH

principal

gets augmented with fqdn

Type
string

Default
airflow

Environment Variable
AIRFLOW__KERBEROS__PRINCIPAL

reinit_frequency

Determines the frequency at which initialization or re-initialization processes occur.

Type
string

Default
3600

Environment Variable
AIRFLOW__KERBEROS__REINIT_FREQUENCY

[lineage]

backend

what lineage backend to use

Type
string

Default

''

Environment Variable

AIRFLOW__LINEAGE__BACKEND

[logging]

base_log_folder

Added in version 2.0.0.

The folder where airflow should store its log files. This path must be absolute. There are a few existing configurations that assume this is set to the default. If you choose to override this you may need to update the [logging] dag_processor_manager_log_location and [logging] dag_processor_child_process_log_directory settings as well.

Type

string

Default

{AIRFLOW_HOME}/logs

Environment Variable

AIRFLOW__LOGGING__BASE_LOG_FOLDER

celery_logging_level

Added in version 2.3.0.

Logging level for celery. If not set, it uses the value of logging_level

Supported values: CRITICAL, ERROR, WARNING, INFO, DEBUG.

Type

string

Default

''

Environment Variable

AIRFLOW__LOGGING__CELERY_LOGGING_LEVEL

celery_stdout_stderr_separation

Added in version 2.7.0.

By default Celery sends all logs into stderr. If enabled any previous logging handlers will get *removed*. With this option AirFlow will create new handlers and send low level logs like INFO and WARNING to stdout, while sending higher severity logs to stderr.

Type

boolean

Default

False

Environment Variable

AIRFLOW__LOGGING__CELERY_STDOUT_STDERR_SEPARATION

color_log_error_keywords

Added in version 2.10.0.

A comma separated list of keywords related to errors whose presence should display the line in red color in UI

Type

string

Default

error, exception

Environment Variable

AIRFLOW__LOGGING__COLOR_LOG_ERROR_KEYWORDS

color_log_warning_keywords

Added in version 2.10.0.

A comma separated list of keywords related to warning whose presence should display the line in yellow color in UI

Type

string

Default

warn

Environment Variable

AIRFLOW__LOGGING__COLOR_LOG_WARNING_KEYWORDS

colored_console_log

Added in version 2.0.0.

Flag to enable/disable Colored logs in Console Colour the logs when the controlling terminal is a TTY.

Type

string

Default

True

Environment Variable

AIRFLOW__LOGGING__COLORED_CONSOLE_LOG

colored_formatter_class

Added in version 2.0.0.

Specifies the class utilized by Airflow to implement colored logging

Type

string

Default

airflow.utils.log.colored_log.CustomTTYColoredFormatter

Environment Variable

AIRFLOW__LOGGING__COLORED_FORMATTER_CLASS

colored_log_format

Added in version 2.0.0.

Log format for when Colored logs is enabled

Type

string

Default

```
[%(blue)s%(asctime)s%(reset)s] %(blue)s%(filename)s:%%(lineno)d}
%{(log_color)s%(levelname)s%(reset)s - %(log_color)s%(message)s%(reset)s
```

Environment Variable

```
AIRFLOW__LOGGING__COLORED_LOG_FORMAT
```

dag_processor_child_process_log_directory

Determines the directory where logs for the child processes of the dag processor will be stored

Type

string

Default

```
{AIRFLOW_HOME}/logs/dag_processor
```

Environment Variable

```
AIRFLOW__LOGGING__DAG_PROCESSOR_CHILD_PROCESS_LOG_DIRECTORY
```

dag_processor_log_format

Added in version 2.4.0.

Format of Dag Processor Log line

Type

string

Default

```
[%(asctime)s] [SOURCE:DAG_PROCESSOR] %(filename)s:%%(lineno)d}
%{(levelname)s - %(message)s
```

Environment Variable

```
AIRFLOW__LOGGING__DAG_PROCESSOR_LOG_FORMAT
```

dag_processor_log_target

Added in version 2.4.0.

Where to send dag parser logs. If “file”, logs are sent to log files defined by child_process_log_directory.

Type

string

Default

file

Environment Variable

```
AIRFLOW__LOGGING__DAG_PROCESSOR_LOG_TARGET
```

delete_local_logs

Added in version 2.6.0.

Whether the local log files for GCS, S3, WASB, HDFS and OSS remote logging should be deleted after they are uploaded to the remote location.

Type

string

Default

False

Environment Variable

AIRFLOW__LOGGING__DELETE_LOCAL_LOGS

encrypt_s3_logs

Added in version 2.0.0.

Use server-side encryption for logs stored in S3

Type

string

Default

False

Environment Variable

AIRFLOW__LOGGING__ENCRYPT_S3_LOGS

extra_logger_names

Added in version 2.0.0.

A comma-separated list of third-party logger names that will be configured to print messages to consoles.

Type

string

Default

''

Environment Variable

AIRFLOW__LOGGING__EXTRA_LOGGER_NAMES

Example

fastapi,sqlalchemy

fab_logging_level

Added in version 2.0.0.

Logging level for Flask-appbuilder UI.

Supported values: CRITICAL, ERROR, WARNING, INFO, DEBUG.

Type

string

Default

WARNING

Environment Variable

AIRFLOW__LOGGING__FAB_LOGGING_LEVEL

file_task_handler_new_file_permissions

Added in version 2.6.0.

Permissions in the form or of octal string as understood by chmod. The permissions are important when you use impersonation, when logs are written by a different user than airflow. The most secure way of configuring it in this case is to add both users to the same group and make it the default group of both users. Group-writeable logs are default in airflow, but you might decide that you are OK with having the logs other-writeable, in which case you should set it to `0o666`. You might decide to add more security if you do not use impersonation and change it to `0o644` to make it only owner-writeable. You can also make it just readable only for owner by changing it to `0o600` if all the access (read/write) for your logs happens from the same user.

Type

string

Default

`0o664`

Environment Variable

AIRFLOW__LOGGING__FILE_TASK_HANDLER_NEW_FILE_PERMISSIONS

Example

`0o664`

file_task_handler_new_folder_permissions

Added in version 2.6.0.

Permissions in the form or of octal string as understood by chmod. The permissions are important when you use impersonation, when logs are written by a different user than airflow. The most secure way of configuring it in this case is to add both users to the same group and make it the default group of both users. Group-writeable logs are default in airflow, but you might decide that you are OK with having the logs other-writeable, in which case you should set it to `0o777`. You might decide to add more security if you do not use impersonation and change it to `0o755` to make it only owner-writeable. You can also make it just readable only for owner by changing it to `0o700` if all the access (read/write) for your logs happens from the same user.

Type

string

Default

`0o775`

Environment Variable

AIRFLOW__LOGGING__FILE_TASK_HANDLER_NEW_FOLDER_PERMISSIONS

Example

`0o775`

google_key_path

Added in version 2.0.0.

Path to Google Credential JSON file. If omitted, authorization based on [the Application Default Credentials](#) will be used.

Type

string

Default`''`**Environment Variable**`AIRFLOW__LOGGING__GOOGLE_KEY_PATH`**interleave_timestamp_parser**

Added in version 2.6.0.

We must parse timestamps to interleave logs between trigger and task. To do so, we need to parse timestamps in log files. In case your log format is non-standard, you may provide import path to callable which takes a string log line and returns the timestamp (datetime.datetime compatible).

Type`string`**Default**`None`**Environment Variable**`AIRFLOW__LOGGING__INTERLEAVE_TIMESTAMP_PARSER`**Example**`path.to.my_func`**log_filename_template**

Added in version 2.0.0.

Formatting for how airflow generates file names/paths for each task run.

Type`string`**Default**

```
dag_id={ ti.dag_id }/run_id={ ti.run_id }/task_id={ ti.task_id }/{% if  
ti.map_index >= 0 %}map_index={ ti.map_index }/{% endif %}attempt={  
try_number|default(ti.try_number) }.log
```

Environment Variable`AIRFLOW__LOGGING__LOG_FILENAME_TEMPLATE`**log_format**

Added in version 2.0.0.

Format of Log line

Type`string`**Default**

```
[%(asctime)s] {%(filename)s:%(lineno)d} %(levelname)s - %(message)s
```

Environment Variable`AIRFLOW__LOGGING__LOG_FORMAT`

log_formatter_class

Added in version 2.3.4.

Determines the formatter class used by Airflow for structuring its log messages. The default formatter class is timezone-aware, which means that timestamps attached to log entries will be adjusted to reflect the local timezone of the Airflow instance

Type

string

Default

`airflow.utils.log.timezone_aware.TimezoneAware`

Environment Variable

`AIRFLOW__LOGGING__LOG_FORMATTER_CLASS`

logging_config_class

Added in version 2.0.0.

Logging class Specify the class that will specify the logging configuration. This class has to be on the python classpath

Type

string

Default

`''`

Environment Variable

`AIRFLOW__LOGGING__LOGGING_CONFIG_CLASS`

Example

`my.path.default_local_settings.LOGGING_CONFIG`

logging_level

Added in version 2.0.0.

Logging level.

Supported values: CRITICAL, ERROR, WARNING, INFO, DEBUG.

Type

string

Default

`INFO`

Environment Variable

`AIRFLOW__LOGGING__LOGGING_LEVEL`

min_length_masked_secret

Added in version 3.0.0.

The minimum length of a secret to be masked in log messages. Secrets shorter than this length will not be masked.

Type

integer

Default

`5`

Environment Variable

AIRFLOW__LOGGING__MIN_LENGTH_MASKED_SECRET

remote_base_log_folder

Added in version 2.0.0.

Storage bucket URL for remote logging S3 buckets should start with **s3://** Cloudwatch log groups should start with **cloudwatch://** GCS buckets should start with **gs://** WASB buckets should start with **wasb** just to help Airflow select correct handler Stackdriver logs should start with **stackdriver://**

Type

string

Default

' '

Environment Variable

AIRFLOW__LOGGING__REMOTE_BASE_LOG_FOLDER

remote_log_conn_id

Added in version 2.0.0.

Users must supply an Airflow connection id that provides access to the storage location. Depending on your remote logging service, this may only be used for reading logs, not writing them.

Type

string

Default

' '

Environment Variable

AIRFLOW__LOGGING__REMOTE_LOG_CONN_ID

remote_logging

Added in version 2.0.0.

Airflow can store logs remotely in AWS S3, Google Cloud Storage or Elastic Search. Set this to True if you want to enable remote logging.

Type

string

Default

False

Environment Variable

AIRFLOW__LOGGING__REMOTE_LOGGING

remote_task_handler_kwargs

Added in version 2.6.0.

The `remote_task_handler_kwargs` param is loaded into a dictionary and passed to the `__init__` of remote task handler and it overrides the values provided by Airflow config. For example if you set `delete_local_logs=False` and you provide `{"delete_local_copy": true}`, then the local log files will be deleted after they are uploaded to remote location.

Type
string

Default
''

Environment Variables

AIRFLOW__LOGGING__REMOTE_TASK_HANDLER_KWARGS
AIRFLOW__LOGGING__REMOTE_TASK_HANDLER_KWARGS_CMD
AIRFLOW__LOGGING__REMOTE_TASK_HANDLER_KWARGS_SECRET

Example

```
{"delete_local_copy": true}
```

secret_mask_adapter

Added in version 2.6.0.

An import path to a function to add adaptations of each secret added with `airflow.sdk.execution_time.secrets_masker.mask_secret` to be masked in log messages. The given function is expected to require a single parameter: the secret to be adapted. It may return a single adaptation of the secret or an iterable of adaptations to each be masked as secrets. The original secret will be masked as well as any adaptations returned.

Type
string

Default
''

Environment Variable

AIRFLOW__LOGGING__SECRET_MASK_ADAPTER

Example

```
urllib.parse.quote
```

simple_log_format

Added in version 2.0.0.

Defines the format of log messages for simple logging configuration

Type
string

Default
%%(asctime)s %(levelname)s - %(message)s

Environment Variable

AIRFLOW__LOGGING__SIMPLE_LOG_FORMAT

task_log_prefix_template

Added in version 2.0.0.

Specify prefix pattern like mentioned below with stream handler `TaskHandlerWithCustomFormatter`

Type
string

Default
''

Environment Variable

AIRFLOW__LOGGING__TASK_LOG_PREFIX_TEMPLATE

Example

{ti.dag_id}-{ti.task_id}-{logical_date}-{ti.try_number}

task_log_reader

Added in version 2.0.0.

Name of handler to read task instance logs. Defaults to use `task` handler.**Type**

string

Default

task

Environment Variable

AIRFLOW__LOGGING__TASK_LOG_READER

trigger_log_server_port

Added in version 2.6.0.

Port to serve logs from for triggerer. See [logging] `worker_log_server_port` description for more info.**Type**

string

Default

8794

Environment Variable

AIRFLOW__LOGGING__TRIGGER_LOG_SERVER_PORT

worker_log_server_port

Added in version 2.2.0.

When you start an Airflow worker, Airflow starts a tiny web server subprocess to serve the workers local log files to the airflow main web server, who then builds pages and sends them to users. This defines the port on which the logs are served. It needs to be unused, and open visible from the main web server to connect into the workers.

Type

string

Default

8793

Environment Variable

AIRFLOW__LOGGING__WORKER_LOG_SERVER_PORT

[metrics]

StatsD integration settings.

metrics_allow_list

Added in version 2.6.0.

Configure an allow list (comma separated regex patterns to match) to send only certain metrics.

Type

string

Default

''

Environment Variable

AIRFLOW__METRICS__METRICS_ALLOW_LIST

Example

"scheduler,executor,dagrun,pool,triggerer,celery" or "^scheduler,^executor,heartbeat|timeout"

metrics_block_list

Added in version 2.6.0.

Configure a block list (comma separated regex patterns to match) to block certain metrics from being emitted. If [metrics] metrics_allow_list and [metrics] metrics_block_list are both configured, [metrics] metrics_block_list is ignored.

Type

string

Default

''

Environment Variable

AIRFLOW__METRICS__METRICS_BLOCK_LIST

Example

"scheduler,executor,dagrun,pool,triggerer,celery" or "^scheduler,^executor,heartbeat|timeout"

otel_debugging_on

Added in version 2.7.0.

If True, all metrics are also emitted to the console. Defaults to False.

Type

string

Default

False

Environment Variable

AIRFLOW__METRICS__OTEL_DEBUGGING_ON

otel_host

Added in version 2.6.0.

Specifies the hostname or IP address of the OpenTelemetry Collector to which Airflow sends metrics and traces.

Type
string

Default
localhost

Environment Variable
AIRFLOW__METRICS__OTEL_HOST

otel_interval_milliseconds

Added in version 2.6.0.

Defines the interval, in milliseconds, at which Airflow sends batches of metrics and traces to the configured OpenTelemetry Collector.

Type
integer

Default
60000

Environment Variable
AIRFLOW__METRICS__OTEL_INTERVAL_MILLISECONDS

otel_on

Added in version 2.6.0.

Enables sending metrics to OpenTelemetry.

Type
string

Default
False

Environment Variable
AIRFLOW__METRICS__OTEL_ON

otel_port

Added in version 2.6.0.

Specifies the port of the OpenTelemetry Collector that is listening to.

Type
string

Default
8889

Environment Variable
AIRFLOW__METRICS__OTEL_PORT

otel_prefix

Added in version 2.6.0.

The prefix for the Airflow metrics.

Type
string

Default
airflow

Environment Variable
AIRFLOW__METRICS__OTEL_PREFIX

otel_service

Added in version 2.10.3.

The default service name of traces.

Type
string

Default
Airflow

Environment Variable
AIRFLOW__METRICS__OTEL_SERVICE

otel_ssl_active

Added in version 2.7.0.

If True, SSL will be enabled. Defaults to False. To establish an HTTPS connection to the OpenTelemetry collector, you need to configure the SSL certificate and key within the OpenTelemetry collector's config.yml file.

Type
string

Default
False

Environment Variable
AIRFLOW__METRICS__OTEL_SSL_ACTIVE

stat_name_handler

Added in version 2.0.0.

A function that validate the StatsD stat name, apply changes to the stat name if necessary and return the transformed stat name.

The function should have the following signature

```
def func_name(stat_name: str) -> str: ...
```

Type
string

Default
''

Environment Variable
AIRFLOW__METRICS__STAT_NAME_HANDLER

statsd_custom_client_path

Added in version 2.0.0.

If you want to utilise your own custom StatsD client set the relevant module path below. Note: The module path must exist on your `PYTHONPATH` <<https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH>> for Airflow to pick it up

Type

string

Default

None

Environment Variable

`AIRFLOW__METRICS__STATSD_CUSTOM_CLIENT_PATH`

statsd_datadog_enabled

Added in version 2.0.0.

To enable datadog integration to send airflow metrics.

Type

string

Default

False

Environment Variable

`AIRFLOW__METRICS__STATSD_DATADOG_ENABLED`

statsd_datadog_metrics_tags

Added in version 2.6.0.

Set to `False` to disable metadata tags for some of the emitted metrics

Type

boolean

Default

True

Environment Variable

`AIRFLOW__METRICS__STATSD_DATADOG_METRICS_TAGS`

statsd_datadog_tags

Added in version 2.0.0.

List of datadog tags attached to all metrics(e.g: `key1:value1,key2:value2`)

Type

string

Default

''

Environment Variable

`AIRFLOW__METRICS__STATSD_DATADOG_TAGS`

statsd_disabled_tags

Added in version 2.6.0.

If you want to avoid sending all the available metrics tags to StatsD, you can configure a block list of prefixes (comma separated) to filter out metric tags that start with the elements of the list (e.g: job_id, run_id)

Type

string

Default

job_id,run_id

Environment Variable

AIRFLOW__METRICS__STATSD_DISABLED_TAGS

Example

job_id,run_id,dag_id,task_id

statsd_host

Added in version 2.0.0.

Specifies the host address where the StatsD daemon (or server) is running

Type

string

Default

localhost

Environment Variable

AIRFLOW__METRICS__STATSD_HOST

statsd_influxdb_enabled

Added in version 2.6.0.

To enable sending Airflow metrics with StatsD-Influxdb tagging convention.

Type

boolean

Default

False

Environment Variable

AIRFLOW__METRICS__STATSD_INFLUXDB_ENABLED

statsd_ipv6

Added in version 3.0.0.

Enables the statsd host to be resolved into IPv6 address

Type

string

Default

False

Environment Variable

AIRFLOW__METRICS__STATSD_IPV6

statsd_on

Added in version 2.0.0.

Enables sending metrics to StatsD.

Type

string

Default

False

Environment Variable

AIRFLOW__METRICS__STATSD_ON

statsd_port

Added in version 2.0.0.

Specifies the port on which the StatsD daemon (or server) is listening to

Type

string

Default

8125

Environment Variable

AIRFLOW__METRICS__STATSD_PORT

statsd_prefix

Added in version 2.0.0.

Defines the namespace for all metrics sent from Airflow to StatsD

Type

string

Default

airflow

Environment Variable

AIRFLOW__METRICS__STATSD_PREFIX

[operators]

default_cpus

Indicates the default number of CPU units allocated to each operator when no specific CPU request is specified in the operator's configuration

Type

string

Default

1

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_CPUS

default_deferrable

Added in version 2.7.0.

The default value of attribute “deferrable” in operators and sensors.

Type

boolean

Default

false

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_DEFERRABLE

default_disk

Indicates the default number of disk storage allocated to each operator when no specific disk request is specified in the operator’s configuration

Type

string

Default

512

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_DISK

default_gpus

Indicates the default number of GPUs allocated to each operator when no specific GPUs request is specified in the operator’s configuration

Type

string

Default

0

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_GPUS

default_owner

The default owner assigned to each new operator, unless provided explicitly or passed via `default_args`

Type

string

Default

airflow

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_OWNER

default_queue

Added in version 2.1.0.

Default queue that tasks get assigned to and that worker listen on.

Type
string

Default
default

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_QUEUE

default_ram

Indicates the default number of RAM allocated to each operator when no specific RAM request is specified in the operator's configuration

Type
string

Default
512

Environment Variable

AIRFLOW__OPERATORS__DEFAULT_RAM

[scheduler]

allowed_run_id_pattern

Added in version 2.6.3.

The run_id pattern used to verify the validity of user input to the run_id parameter when triggering a DAG. This pattern cannot change the pattern used by scheduler to generate run_id for scheduled DAG runs or DAG runs triggered without changing the run_id parameter.

Type
string

Default
^ [A-Za-z0-9_.~:+-]+\$

Environment Variable

AIRFLOW__SCHEDULER__ALLOWED_RUN_ID_PATTERN

catchup_by_default

Turn on scheduler catchup by setting this to True. Default behavior is unchanged and Command Line Backfills still work, but the scheduler will not do scheduler catchup if this is False, however it can be set on a per DAG basis in the DAG definition (catchup)

Type
boolean

Default
False

Environment Variable

AIRFLOW__SCHEDULER__CATCHUP_BY_DEFAULT

create_cron_data_intervals

Added in version 2.9.0.

Whether to create DAG runs that span an interval or one single point in time for cron schedules, when a cron string is provided to `schedule` argument of a DAG.

- True: **CronDataIntervalTimetable** is used, which is suitable for DAGs with well-defined data interval. You get contiguous intervals from the end of the previous interval up to the scheduled datetime.
- False: **CronTriggerTimetable** is used, which is closer to the behavior of cron itself.

Notably, for **CronTriggerTimetable**, the logical date is the same as the time the DAG Run will try to schedule, while for **CronDataIntervalTimetable**, the logical date is the beginning of the data interval, but the DAG Run will try to schedule at the end of the data interval.

See also

Differences between “trigger” and “data interval” timetables

Type

boolean

Default

False

Environment Variable

AIRFLOW__SCHEDULER__CREATE_CRON_DATA_INTERVALS

create_delta_data_intervals

Added in version 2.11.0.

Whether to create DAG runs that span an interval or one single point in time when a timedelta or relativedelta is provided to `schedule` argument of a DAG.

- True: **DeltaDataIntervalTimetable** is used, which is suitable for DAGs with well-defined data interval. You get contiguous intervals from the end of the previous interval up to the scheduled datetime.
- False: **DeltaTriggerTimetable** is used, which is suitable for DAGs that simply want to say e.g. “run this every day” and do not care about the data interval.

Notably, for **DeltaTriggerTimetable**, the logical date is the same as the time the DAG Run will try to schedule, while for **DeltaDataIntervalTimetable**, the logical date is the beginning of the data interval, but the DAG Run will try to schedule at the end of the data interval.

See also

Differences between “trigger” and “data interval” timetables

Type

boolean

Default

False

Environment Variable

AIRFLOW__SCHEDULER__CREATE_DELTA_DATA_INTERVALS

dag_stale_not_seen_duration

Added in version 2.4.0.

Time in seconds after which dags, which were not updated by Dag Processor are deactivated.

Type

integer

Default

600

Environment Variable

AIRFLOW__SCHEDULER__DAG_STALE_NOT_SEEN_DURATION

enable_health_check

Added in version 2.4.0.

When you start a scheduler, airflow starts a tiny web server subprocess to serve a health check if this is set to True

Type

boolean

Default

False

Environment Variable

AIRFLOW__SCHEDULER__ENABLE_HEALTH_CHECK

enable_tracemalloc

Added in version 3.0.0.

Whether to enable memory allocation tracing in the scheduler. If enabled, Airflow will start tracing memory allocation and log the top 10 memory usages at the error level upon receiving the signal SIGUSR1. This is an expensive operation and generally should not be used except for debugging purposes.

Type

boolean

Default

False

Environment Variable

AIRFLOW__SCHEDULER__ENABLE_TRACEMALLOC

ignore_first_depends_on_past_by_default

Added in version 2.3.0.

Setting this to True will make first task instance of a task ignore depends_on_past setting. A task instance will be considered as the first task instance of a task when there is no task instance in the DB with a logical_date earlier than it., i.e. no manual marking success will be needed for a newly added task to be scheduled.

Type

boolean

Default

True

Environment Variable

AIRFLOW__SCHEDULER__IGNORE_FIRST_DEPENDS_ON_PAST_BY_DEFAULT

job_heartbeat_sec

Task instances listen for external kill signal (when you clear tasks from the CLI or the UI), this defines the frequency at which they should listen (in seconds).

Type

float

Default

5

Environment Variable

AIRFLOW__SCHEDULER__JOB_HEARTBEAT_SEC

max_dagruns_per_loop_to_schedule

Added in version 2.0.0.

How many DagRuns should a scheduler examine (and lock) when scheduling and queuing tasks.

See also

Scheduler Configuration options

Type

integer

Default

20

Environment Variable

AIRFLOW__SCHEDULER__MAX_DAGRUNS_PER_LOOP_TO_SCHEDULE

max_dagruns_to_create_per_loop

Added in version 2.0.0.

Max number of DAGs to create DagRuns for per scheduler loop.

See also

Scheduler Configuration options

Type

integer

Default

10

Environment Variable

AIRFLOW__SCHEDULER__MAX_DAGRUNS_TO_CREATE_PER_LOOP

max_tis_per_query

This determines the number of task instances to be evaluated for scheduling during each scheduler loop. Set this to 0 to use the value of [core] parallelism

Type

integer

Default

16

Environment Variable

AIRFLOW__SCHEDULER__MAX_TIS_PER_QUERY

num_runs

Added in version 1.10.6.

The number of times to try to schedule each DAG file -1 indicates unlimited number

Type

integer

Default

-1

Environment Variable

AIRFLOW__SCHEDULER__NUM_RUNS

orphaned_tasks_check_interval

Added in version 2.0.0.

How often (in seconds) should the scheduler check for orphaned tasks and SchedulerJobs

Type

float

Default

300.0

Environment Variable

AIRFLOW__SCHEDULER__ORPHANED_TASKS_CHECK_INTERVAL

parsing_cleanup_interval

Added in version 2.5.0.

How often (in seconds) to check for stale DAGs (DAGs which are no longer present in the expected files) which should be deactivated, as well as assets that are no longer referenced and should be marked as orphaned.

Type

integer

Default

60

Environment Variable

AIRFLOW__SCHEDULER__PARSING_CLEANUP_INTERVAL

parsing_pre_import_modules

Added in version 2.6.0.

The scheduler reads dag files to extract the airflow modules that are going to be used, and imports them ahead of time to avoid having to re-do it for each parsing process. This flag can be set to `False` to disable this behavior in case an airflow module needs to be freshly imported each time (at the cost of increased DAG parsing time).

Type

boolean

Default

`True`

Environment Variable

`AIRFLOW__SCHEDULER__PARSING_PRE_IMPORT_MODULES`

pool_metrics_interval

Added in version 2.0.0.

How often (in seconds) should pool usage stats be sent to StatsD (if `statsd_on` is enabled)

Type

float

Default

`5.0`

Environment Variable

`AIRFLOW__SCHEDULER__POOL_METRICS_INTERVAL`

running_metrics_interval

Added in version 3.0.0.

How often (in seconds) should running task instance stats be sent to StatsD (if `statsd_on` is enabled)

Type

float

Default

`30.0`

Environment Variable

`AIRFLOW__SCHEDULER__RUNNING_METRICS_INTERVAL`

scheduler_health_check_server_host

Added in version 2.8.0.

When you start a scheduler, airflow starts a tiny web server subprocess to serve a health check on this host

Type

string

Default

`0.0.0.0`

Environment Variable

`AIRFLOW__SCHEDULER__SCHEDULER_HEALTH_CHECK_SERVER_HOST`

`scheduler_health_check_server_port`

Added in version 2.4.0.

When you start a scheduler, airflow starts a tiny web server subprocess to serve a health check on this port

Type
integer

Default
8974

Environment Variable

AIRFLOW__SCHEDULER__SCHEDULER_HEALTH_CHECK_SERVER_PORT

`scheduler_health_check_threshold`

Added in version 1.10.2.

If the last scheduler heartbeat happened more than [scheduler] `scheduler_health_check_threshold` ago (in seconds), scheduler is considered unhealthy. This is used by the health check in the `/health` endpoint and in `airflow jobs check` CLI for SchedulerJob.

Type
integer

Default
30

Environment Variable

AIRFLOW__SCHEDULER__SCHEDULER_HEALTH_CHECK_THRESHOLD

`scheduler_heartbeat_sec`

The scheduler constantly tries to trigger new tasks (look at the scheduler section in the docs for more information). This defines how often the scheduler should run (in seconds).

Type
integer

Default
5

Environment Variable

AIRFLOW__SCHEDULER__SCHEDULER_HEARTBEAT_SEC

`scheduler_idle_sleep_time`

Added in version 2.2.0.

Controls how long the scheduler will sleep between loops, but if there was nothing to do in the loop. i.e. if it scheduled something then it will start the next loop iteration straight away.

Type
float

Default
1

Environment Variable

AIRFLOW__SCHEDULER__SCHEDULER_IDLE_SLEEP_TIME

task_instance_heartbeat_sec

Added in version 2.7.0.

The frequency (in seconds) at which the LocalTaskJob should send heartbeat signals to the scheduler to notify it's still alive. If this value is set to 0, the heartbeat interval will default to the value of [scheduler] task_instance_heartbeat_timeout.

Type

integer

Default

0

Environment Variable

AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_SEC

task_instance_heartbeat_timeout

Local task jobs periodically heartbeat to the DB. If the job has not heartbeat in this many seconds, the scheduler will mark the associated task instance as failed and will re-schedule the task.

Type

integer

Default

300

Environment Variable

AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_TIMEOUT

task_instance_heartbeat_timeout_detection_interval

Added in version 2.3.0.

How often (in seconds) should the scheduler check for task instances whose heartbeats have timed out.

Type

float

Default

10.0

Environment Variable

AIRFLOW__SCHEDULER__TASK_INSTANCE_HEARTBEAT_TIMEOUT_DETECTION_INTERVAL

task_queued_timeout

Added in version 2.6.0.

Amount of time a task can be in the queued state before being retried or set to failed.

Type

float

Default

600.0

Environment Variable

AIRFLOW__SCHEDULER__TASK_QUEUED_TIMEOUT

task_queued_timeout_check_interval

Added in version 2.6.0.

How often to check for tasks that have been in the queued state for longer than [scheduler] task_queued_timeout.

Type

float

Default

120.0

Environment Variable

AIRFLOW__SCHEDULER__TASK_QUEUED_TIMEOUT_CHECK_INTERVAL

trigger_timeout_check_interval

Added in version 2.2.0.

How often to check for expired trigger requests that have not run yet.

Type

float

Default

15

Environment Variable

AIRFLOW__SCHEDULER__TRIGGER_TIMEOUT_CHECK_INTERVAL

use_job_schedule

Added in version 1.10.2.

Turn off scheduler use of cron intervals by setting this to False. DAGs submitted manually in the web UI or with trigger_dag will still run.

Type

boolean

Default

True

Environment Variable

AIRFLOW__SCHEDULER__USE_JOB_SCHEDULE

use_row_level_locking

Added in version 2.0.0.

Should the scheduler issue SELECT ... FOR UPDATE in relevant queries. If this is set to False then you should not run more than a single scheduler at once

Type

boolean

Default

True

Environment Variable

AIRFLOW__SCHEDULER__USE_ROW_LEVEL_LOCKING

dag_dir_list_interval (Deprecated)

Deprecated since version 3.0: The option has been moved to *dag_processor.refresh_interval*

[secrets]

backend

Added in version 1.10.10.

Full class name of secrets backend to enable (will precede env vars and metastore in search path)

Type

string

Default

''

Environment Variable

AIRFLOW__SECRETS__BACKEND

Example

```
airflow.providers.amazon.aws.secrets.systems_manager.  
SystemsManagerParameterStoreBackend
```

backend_kwargs

Added in version 1.10.10.

The `backend_kwargs` param is loaded into a dictionary and passed to `__init__` of secrets backend class. See documentation for the secrets backend you are using. JSON is expected.

Example for AWS Systems Manager ParameterStore: `{"connections_prefix": "/airflow/connections", "profile_name": "default"}`

Type

string

Default

''

Environment Variables

AIRFLOW__SECRETS__BACKEND_KWARGS

AIRFLOW__SECRETS__BACKEND_KWARGS_CMD

AIRFLOW__SECRETS__BACKEND_KWARGS_SECRET

cache_ttl_seconds

Added in version 2.7.0.

Note

This is an *experimental feature*.

When the cache is enabled, this is the duration for which we consider an entry in the cache to be valid. Entries are refreshed if they are older than this many seconds. It means that when the cache is enabled, this is the maximum amount of time you need to wait to see a Variable change take effect.

Type
integer

Default
900

Environment Variable
AIRFLOW__SECRETS__CACHE_TTL_SECONDS

use_cache

Added in version 2.7.0.

Note

This is an *experimental feature*.

Enables local caching of Variables, when parsing DAGs only. Using this option can make dag parsing faster if Variables are used in top level code, at the expense of longer propagation time for changes. Please note that this cache concerns only the DAG parsing step. There is no caching in place when DAG tasks are run.

Type
boolean

Default
False

Environment Variable
AIRFLOW__SECRETS__USE_CACHE

[sensors]

default_timeout

Added in version 2.3.0.

Sensor default timeout, 7 days by default ($7 * 24 * 60 * 60$).

Type
float

Default
604800

Environment Variable
AIRFLOW__SENSORS__DEFAULT_TIMEOUT

[sentry]

Sentry integration. Here you can supply additional configuration options based on the Python platform. See [Python / Configuration / Basic Options](#) for more details. Unsupported options: `integrations`, `in_app_include`, `in_app_exclude`, `ignore_errors`, `before_breadcrumb`, `transport`.

before_send

Added in version 2.2.0.

Dotted path to a before_send function that the sentry SDK should be configured to use.

Type
string

Default
None

Environment Variable
AIRFLOW__SENTRY__BEFORE_SEND

sentry_dsn

Added in version 1.10.6.

Type
string

Default
''

Environment Variables
AIRFLOW__SENTRY__SENTRY_DSN
AIRFLOW__SENTRY__SENTRY_DSN_CMD
AIRFLOW__SENTRY__SENTRY_DSN_SECRET

sentry_on

Added in version 2.0.0.

Enable error reporting to Sentry

Type
string

Default
false

Environment Variable
AIRFLOW__SENTRY__SENTRY_ON

[smtp]

If you want airflow to send emails on retries, failure, and you want to use the `airflow.utils.email.send_email_smtp` function, you have to configure an smtp server here

smtp_host

Specifies the host server address used by Airflow when sending out email notifications via SMTP.

Type
string

Default
localhost

Environment Variable
AIRFLOW__SMTP__SMTP_HOST

smtp_mail_from

Specifies the default **from** email address used when Airflow sends email notifications.

Type

string

Default

airflow@example.com

Environment Variable

AIRFLOW__SMTP__SMTP_MAIL_FROM

smtp_port

Defines the port number on which Airflow connects to the SMTP server to send email notifications.

Type

string

Default

25

Environment Variable

AIRFLOW__SMTP__SMTP_PORT

smtp_retry_limit

Added in version 2.0.0.

Defines the maximum number of times Airflow will attempt to connect to the SMTP server.

Type

integer

Default

5

Environment Variable

AIRFLOW__SMTP__SMTP_RETRY_LIMIT

smtp_ssl

Determines whether to use an SSL connection when talking to the SMTP server.

Type

string

Default

False

Environment Variable

AIRFLOW__SMTP__SMTP_SSL

smtp_starttls

Determines whether to use the STARTTLS command when connecting to the SMTP server.

Type

string

Default

True

Environment Variable

AIRFLOW__SMTP__SMTP_STARTTLS

smtp_timeout

Added in version 2.0.0.

Determines the maximum time (in seconds) the Apache Airflow system will wait for a connection to the SMTP server to be established.

Type

integer

Default

30

Environment Variable

AIRFLOW__SMTP__SMTP_TIMEOUT

[traces]

Distributed traces integration settings.

otel_debugging_on

Added in version 2.10.0.

If True, all traces are also emitted to the console. Defaults to False.

Type

string

Default

False

Environment Variable

AIRFLOW__TRACES__OTEL_DEBUGGING_ON

otel_host

Added in version 2.10.0.

Specifies the hostname or IP address of the OpenTelemetry Collector to which Airflow sends traces.

Type

string

Default

localhost

Environment Variable

AIRFLOW__TRACES__OTEL_HOST

otel_on

Added in version 2.10.0.

Enables sending traces to OpenTelemetry.

Type

string

Default

False

Environment Variable

AIRFLOW__TRACES__OTEL_ON

otel_port

Added in version 2.10.0.

Specifies the port of the OpenTelemetry Collector that is listening to.

Type

string

Default

8889

Environment Variable

AIRFLOW__TRACES__OTEL_PORT

otel_service

Added in version 2.10.0.

The default service name of traces.

Type

string

Default

Airflow

Environment Variable

AIRFLOW__TRACES__OTEL_SERVICE

otel_ssl_active

Added in version 2.10.0.

If True, SSL will be enabled. Defaults to False. To establish an HTTPS connection to the OpenTelemetry collector, you need to configure the SSL certificate and key within the OpenTelemetry collector's config.yml file.

Type

string

Default

False

Environment Variable

AIRFLOW__TRACES__OTEL_SSL_ACTIVE

[triggerer]

capacity

Added in version 2.2.0.

How many triggers a single Triggerer will run at once, by default.

Type

string

Default

1000

Environment Variable

AIRFLOW__TRIGGERER__CAPACITY

job_heartbeat_sec

Added in version 2.6.3.

How often to heartbeat the Triggerer job to ensure it hasn't been killed.

Type

float

Default

5

Environment Variable

AIRFLOW__TRIGGERER__JOB_HEARTBEAT_SEC

triggerer_health_check_threshold

Added in version 2.7.0.

If the last triggerer heartbeat happened more than [triggerer] triggerer_health_check_threshold ago (in seconds), triggerer is considered unhealthy. This is used by the health check in the /health endpoint and in airflow jobs check CLI for TriggererJob.

Type

float

Default

30

Environment Variable

AIRFLOW__TRIGGERER__TRIGGERER_HEALTH_CHECK_THRESHOLD

default_capacity (Deprecated)

Deprecated since version 3.0: The option has been moved to *triggerer.capacity*

[webserver]

access_denied_message

Added in version 2.7.0.

The message displayed when a user attempts to execute actions beyond their authorised privileges.

Type
string

Default
Access is Denied

Environment Variable
AIRFLOW__WEB SERVER__ACCESS_DENIED_MESSAGE

audit_view_excluded_events

Added in version 2.3.0.

Comma separated string of view events to exclude from dag audit view. All other events will be added minus the ones passed here. The audit logs in the db will not be affected by this parameter.

Type
string

Default
None

Environment Variable
AIRFLOW__WEB SERVER__AUDIT_VIEW_EXCLUDED_EVENTS

Example
cli_task_run,running,success

audit_view_included_events

Added in version 2.3.0.

Comma separated string of view events to include in dag audit view. If passed, only these events will populate the dag audit view. The audit logs in the db will not be affected by this parameter.

Type
string

Default
None

Environment Variable
AIRFLOW__WEB SERVER__AUDIT_VIEW_INCLUDED_EVENTS

Example
dagrun_cleared,failed

auto_refresh_interval

Added in version 2.2.0.

How frequently, in seconds, the DAG data will auto-refresh in graph or grid view when auto-refresh is turned on

Type
integer

Default
3

Environment Variable
AIRFLOW__WEB SERVER__AUTO_REFRESH_INTERVAL

default_wrap

Added in version 1.10.4.

Default setting for wrap toggle on DAG code and TI log views.

Type

boolean

Default

False

Environment Variable

AIRFLOW__WEBSERVER__DEFAULT_WRAP

enable_swagger_ui

Added in version 2.6.0.

Boolean for running SwaggerUI in the webserver.

Type

boolean

Default

True

Environment Variable

AIRFLOW__WEBSERVER__ENABLE_SWAGGER_UI

expose_hostname

Added in version 1.10.8.

Expose hostname in the web server

Type

string

Default

False

Environment Variable

AIRFLOW__WEBSERVER__EXPOSE_HOSTNAME

grid_view_sorting_order

Added in version 2.7.0.

Sorting order in grid view. Valid values are: `topological`, `hierarchical_alpha`

Type

string

Default

`topological`

Environment Variable

AIRFLOW__WEBSERVER__GRID_VIEW_SORTING_ORDER

hide_paused_dags_by_default

By default, the webserver shows paused DAGs. Flip this to hide paused DAGs by default

Type
string

Default
False

Environment Variable

AIRFLOW__WEBSERVER__HIDE_PAUSED_DAGS_BY_DEFAULT

instance_name

Added in version 2.1.0.

Sets a custom page title for the DAGs overview page and site title for all pages

Type
string

Default
None

Environment Variable

AIRFLOW__WEBSERVER__INSTANCE_NAME

instance_name_has_markup

Added in version 2.3.0.

Whether the custom page title for the DAGs overview page contains any Markup language

Type
boolean

Default
False

Environment Variable

AIRFLOW__WEBSERVER__INSTANCE_NAME_HAS_MARKUP

log_fetch_timeout_sec

The amount of time (in secs) webserver will wait for initial handshake while fetching logs from other worker machine

Type
string

Default
5

Environment Variable

AIRFLOW__WEBSERVER__LOG_FETCH_TIMEOUT_SEC

navbar_color

Define the color of navigation bar

Type
string

Default

#fff

Environment Variable

AIRFLOW__WEB SERVER__NAVBAR_COLOR

navbar_hover_color

Added in version 2.9.0.

Define the color of navigation bar links when hovered

Type

string

Default

#eee

Environment Variable

AIRFLOW__WEB SERVER__NAVBAR_HOVER_COLOR

navbar_text_color

Added in version 2.8.0.

Define the color of text in the navigation bar

Type

string

Default

#51504f

Environment Variable

AIRFLOW__WEB SERVER__NAVBAR_TEXT_COLOR

navbar_text_hover_color

Added in version 2.9.0.

Define the color of text in the navigation bar when hovered

Type

string

Default

#51504f

Environment Variable

AIRFLOW__WEB SERVER__NAVBAR_TEXT_HOVER_COLOR

page_size

Consistent page size across all listing views in the UI

Type

string

Default

50

Environment Variable

AIRFLOW__WEB SERVER__PAGE_SIZE

require_confirmation_dag_change

Added in version 2.9.0.

Require confirmation when changing a DAG in the web UI. This is to prevent accidental changes to a DAG that may be running on sensitive environments like production. When set to True, confirmation dialog will be shown when a user tries to Pause/Unpause, Trigger a DAG

Type

boolean

Default

False

Environment Variable

AIRFLOW__WEB SERVER__REQUIRE_CONFIRMATION_DAG_CHANGE

secret_key

Secret key used to run your api server. It should be as random as possible. However, when running more than 1 instances of webserver, make sure all of them use the same `secret_key` otherwise one of them will error with “CSRF session token is missing”. The webserver key is also used to authorize requests to Celery workers when logs are retrieved. The token generated using the secret key has a short expiry time though - make sure that time on ALL the machines that you run airflow components on is synchronized (for example using ntpd) otherwise you might get “forbidden” errors when the logs are accessed.

Type

string

Default

{SECRET_KEY}

Environment Variables

AIRFLOW__WEB SERVER__SECRET_KEY

AIRFLOW__WEB SERVER__SECRET_KEY_CMD

AIRFLOW__WEB SERVER__SECRET_KEY_SECRET

warn_deployment_exposure

Added in version 2.3.0.

Boolean for displaying warning for publicly viewable deployment

Type

boolean

Default

True

Environment Variable

AIRFLOW__WEB SERVER__WARN_DEPLOYMENT_EXPOSURE

x_frame_enabled

Added in version 1.10.8.

Allow the UI to be rendered in a frame

Type

boolean

Default

True

Environment Variable

AIRFLOW__WEBSERVER__X_FRAME_ENABLED

access_logfile (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.access_logfile*

expose_config (Deprecated)

Deprecated since version 3.0.1: The option has been moved to *api.expose_config*

web_server_host (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.host*

web_server_port (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.port*

web_server_ssl_cert (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.ssl_cert*

web_server_ssl_key (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.ssl_key*

web_server_worker_timeout (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.worker_timeout*

workers (Deprecated)

Deprecated since version 3.0: The option has been moved to *api.workers*

[workers]

Configuration related to workers that run Airflow tasks.

execution_api_retries

Added in version 3.0.0.

The maximum number of retry attempts to the execution API server.

Type
integer

Default
5

Environment Variable
AIRFLOW__WORKERS__EXECUTION_API_RETRIES

execution_api_retry_wait_max

Added in version 3.0.0.

The maximum amount of time (in seconds) to wait before retrying a failed API request.

Type
float

Default
90.0

Environment Variable
AIRFLOW__WORKERS__EXECUTION_API_RETRY_WAIT_MAX

execution_api_retry_wait_min

Added in version 3.0.0.

The minimum amount of time (in seconds) to wait before retrying a failed API request.

Type
float

Default
1.0

Environment Variable
AIRFLOW__WORKERS__EXECUTION_API_RETRY_WAIT_MIN

max_failed_heartbeats

Added in version 3.0.0.

The maximum number of consecutive failed heartbeats before terminating the task instance process.

Type
integer

Default
3

Environment Variable
AIRFLOW__WORKERS__MAX_FAILED_HEARTBEATS

min_heartbeat_interval

Added in version 3.0.0.

The minimum interval (in seconds) at which the worker checks the task instance's heartbeat status with the API server to confirm it is still alive.

Type
integer

Default
5

Environment Variable
AIRFLOW__WORKERS__MIN_HEARTBEAT_INTERVAL

secrets_backend

Added in version 3.0.0.

Full class name of secrets backend to enable for workers (will precede env vars backend)

Type
string

Default
''

Environment Variable
AIRFLOW__WORKERS__SECRETS_BACKEND

Example

```
airflow.providers.amazon.aws.secrets.systems_manager.  
SystemsManagerParameterStoreBackend
```

secrets_backend_kwargs

Added in version 3.0.0.

The secrets_backend_kwargs param is loaded into a dictionary and passed to `__init__` of secrets backend class. See documentation for the secrets backend you are using. JSON is expected.

Example for AWS Systems Manager ParameterStore: `{"connections_prefix": "/airflow/connections", "profile_name": "default"}`

Type
string

Default
''

Environment Variables
AIRFLOW__WORKERS__SECRETS_BACKEND_KWARGS
AIRFLOW__WORKERS__SECRETS_BACKEND_KWARGS_CMD
AIRFLOW__WORKERS__SECRETS_BACKEND_KWARGS_SECRET

3.25 Reference for package extras

3.25.1 Airflow distribution packages

With Airflow 3, Airflow is now split into several independent and isolated distribution packages on top of already existing providers and the dependencies are isolated and simplified across those distribution packages.

While the original installation methods via apache-airflow distribution package and extras still work as previously and it installs complete Airflow installation ready to serve as scheduler, webserver, triggerer and worker, the

apache-airflow package is now a meta-package that installs all the other distribution packages, it's also possible to install only the distribution packages that are needed for a specific component you want to run Airflow with.

The following distribution packages are available:

| Distribution package | Purpose | Optional extras |
|----------------------------|---|--|
| apache-airflow-core | This is the core distribution package that contains the Airflow scheduler, webserver, triggerer code. | <ul style="list-style-type: none"> Core extras that add optional functionality to Airflow core system - enhancing its functionality across multiple providers. Group <code>all</code> extra that installs all optional functionalities together. |
| apache-airflow-task-sdk | This is the distribution package that is needed to run tasks in the worker | <ul style="list-style-type: none"> No optional extras |
| apache-airflow-providers-* | Those are distribution packages that contain integrations of Airflow with external systems, 3rd-party software and services. Usually they provide operators, hooks, sensors, triggers, but also different types of extensions such as logging handlers, executors, and other functionalities that are tied to particular service or system. | <ul style="list-style-type: none"> Each provider distribution packages might have its own optional extras |
| apache-airflow | <p>This is the meta-distribution-package that installs (mandatory):</p> <ul style="list-style-type: none"> <code>apache-airflow-core</code> (always the same version as the <code>apache-airflow</code>) <code>apache-airflow-task-sdk</code> (latest) | <p> <ul style="list-style-type: none"> Any of the core extras Any of the provider packages via extras </p> <p>This is backwards-compatible with previous installation methods in Airflow 2.</p> <p>Group extras:</p> <ul style="list-style-type: none"> <code>all-core</code> - extra that installs all extras of the <code>apache-airflow-core</code> package <code>all</code> - extra that installs all core extras and all provider packages (without their optional extras). |

As mentioned above, Airflow has a number of optional “extras” that you can use to add features to your installation when you are installing Airflow. Those extras are a good way for the users to manage their installation, but also they are useful for contributors to Airflow when they want to contribute some of the features - including optional integrations of Airflow - via providers.

Here's the list of all the extra dependencies of Apache Airflow.

3.25.2 Core Airflow extras

These are core Airflow extras that extend capabilities of core Airflow. They usually do not install provider packages (with the exception of `celery` and `cncf.kubernetes` extras), they just install necessary python dependencies for the provided package.

| extra | install command | enables |
|-------------------|--|--|
| aiobotocore | <code>pip install 'apache-airflow[aiobotocore]'</code> | Support for asynchronous (deferrable) operators for Amazon integration |
| async | <code>pip install 'apache-airflow[async]'</code> | Async worker classes for Gunicorn |
| github-enterprise | <code>pip install 'apache-airflow[github-enterprise]'</code> | GitHub Enterprise auth backend |
| google-auth | <code>pip install 'apache-airflow[google-auth]'</code> | Google auth backend |
| graphviz | <code>pip install 'apache-airflow[graphviz]'</code> | Graphviz renderer for converting DAG to graphical output |
| kerberos | <code>pip install 'apache-airflow[kerberos]'</code> | Kerberos integration for Kerberized services (Hadoop, Presto, Trino) |
| ldap | <code>pip install 'apache-airflow[ldap]'</code> | LDAP authentication for users |
| leveldb | <code>pip install 'apache-airflow[leveldb]'</code> | Required for use leveldb extra in google provider |
| otel | <code>pip install 'apache-airflow[otel]'</code> | Required for OpenTelemetry metrics |
| pandas | <code>pip install 'apache-airflow[pandas]'</code> | Install Pandas library compatible with Airflow |
| polars | <code>pip install 'apache-airflow[polars]'</code> | Polars hooks and operators |
| rabbitmq | <code>pip install 'apache-airflow[rabbitmq]'</code> | RabbitMQ support as a Celery backend |
| sentry | <code>pip install 'apache-airflow[sentry]'</code> | Sentry service for application logging and monitoring |
| s3fs | <code>pip install 'apache-airflow[s3fs]'</code> | Support for S3 as Airflow FS |
| saml | <code>pip install 'apache-airflow[saml]'</code> | Support for SAML authentication in Airflow |
| standard | <code>pip install apache-airflow[standard]</code> | Standard hooks and operators |
| statsd | <code>pip install 'apache-airflow[statsd]'</code> | Needed by StatsD metrics |
| uv | <code>pip install 'apache-airflow[uv]'</code> | Install uv - fast, Rust-based package installer (experimental) |
| cloudpickle | <code>pip install apache-airflow[cloudpickle]</code> | Cloudpickle hooks and operators |

3.25.3 Providers extras

These providers extras are simply convenience extras to install providers so that you can install the providers with simple command - including provider package and necessary dependencies in single command, which allows PIP to resolve any conflicting dependencies. This is extremely useful for first time installation where you want to repeatably install version of dependencies which are ‘valid’ for both Airflow and providers installed.

For example the below command will install:

- apache-airflow
- apache-airflow-core
- apache-airflow-task-sdk
- apache-airflow-providers-amazon
- apache-airflow-providers-google
- apache-airflow-providers-apache-spark

with a consistent set of dependencies based on constraint files provided by Airflow Community at the time 3.1.0 version was released.

```
pip install apache-airflow[google,amazon,apache-spark]==3.1.0 \
--constraint "https://raw.githubusercontent.com/apache/airflow/constraints-3.1.0/
constraints-3.9.txt"
```

Note, that this will install providers in the versions that were released at the time of Airflow 3.1.0 release. You can later upgrade those providers manually if you want to use latest versions of the providers.

Also, those extras are ONLY available in the `apache-airflow` distribution package as they are a convenient way to install all the `airflow` packages together - similarly to what happened in Airflow 2. When you are installing `airflow-core` or `airflow-task-sdk` separately, if you want to install providers, you need to install them separately as `apache-airflow-providers-*` distribution packages.

Apache Software extras

These are extras that add dependencies needed for integration with other Apache projects (note that `apache.atlas` and `apache.webhdfs` do not have their own providers - they only install additional libraries that can be used in custom bash/python providers).

| extra | install command | enables |
|------------------|--|---|
| apache-atlas | <code>pip install 'apache-airflow[apache-atlas]'</code> | Apache Atlas |
| apache-beam | <code>pip install 'apache-airflow[apache-beam]'</code> | Apache Beam operators & hooks |
| apache-cassandra | <code>pip install 'apache-airflow[apache-cassandra]</code> | Cassandra related operators & hooks |
| apache-drill | <code>pip install 'apache-airflow[apache-drill]'</code> | Drill related operators & hooks |
| apache-druid | <code>pip install 'apache-airflow[apache-druid]'</code> | Druid related operators & hooks |
| apache-flink | <code>pip install 'apache-airflow[apache-flink]'</code> | Flink related operators & hooks |
| apache-hdfs | <code>pip install 'apache-airflow[apache-hdfs]'</code> | HDFS hooks and operators |
| apache-hive | <code>pip install 'apache-airflow[apache-hive]'</code> | All Hive related operators |
| apache-iceberg | <code>pip install 'apache-airflow[apache-iceberg]'</code> | Apache Iceberg hooks |
| apache-impala | <code>pip install 'apache-airflow[apache-impala]'</code> | All Impala related operators & hooks |
| apache-kafka | <code>pip install 'apache-airflow[apache-kafka]'</code> | All Kafka related operators & hooks |
| apache-kylin | <code>pip install 'apache-airflow[apache-kylin]'</code> | All Kylin related operators & hooks |
| apache-livy | <code>pip install 'apache-airflow[apache-livy]'</code> | All Livy related operators, hooks & sensors |
| apache-pig | <code>pip install 'apache-airflow[apache-pig]'</code> | All Pig related operators & hooks |
| apache-pinot | <code>pip install 'apache-airflow[apache-pinot]'</code> | All Pinot related hooks |
| apache-spark | <code>pip install 'apache-airflow[apache-spark]'</code> | All Spark related operators & hooks |
| apache-tinkerpop | <code>pip install apache-airflow[apache-tinkerpop]</code> | Apache-tinkerpop hooks and operators |
| apache-webhdfs | <code>pip install 'apache-airflow[apache-webhdfs]'</code> | HDFS hooks and operators |

External Services extras

These are extras that add dependencies needed for integration with external services - either cloud based or on-premises.

| | | extra | install command | enables |
|-----------------|--|---|-----------------|--|
| airbyte | | pip install 'apache-airflow[airbyte]' | | Airbyte hooks and operators |
| alibaba | | pip install 'apache-airflow[alibaba]' | | Alibaba Cloud |
| apprise | | pip install 'apache-airflow[apprise]' | | Apprise Notification |
| amazon | | pip install 'apache-airflow[amazon]' | | Amazon Web Services |
| asana | | pip install 'apache-airflow[asana]' | | Asana hooks and operators |
| atlassian-jira | | pip install 'apache-airflow[atlassian-jira]' | | Jira hooks and operators |
| microsoft-azure | | pip install 'apache-airflow[microsoft-azure]' | | Microsoft Azure |
| cloudant | | pip install 'apache-airflow[cloudant]' | | Cloudant hook |
| cohere | | pip install 'apache-airflow[cohere]' | | Cohere hook and operators |
| databricks | | pip install 'apache-airflow[databricks]' | | Databricks hooks and operators |
| datadog | | pip install 'apache-airflow[datadog]' | | Datadog hooks and sensors |
| dbt-cloud | | pip install 'apache-airflow[dbt-cloud]' | | dbt Cloud hooks and operators |
| dingding | | pip install 'apache-airflow[dingding]' | | Dingding hooks and sensors |
| discord | | pip install 'apache-airflow[discord]' | | Discord hooks and sensors |
| facebook | | pip install 'apache-airflow[facebook]' | | Facebook Social |
| github | | pip install 'apache-airflow[github]' | | GitHub operators and hook |
| google | | pip install 'apache-airflow[google]' | | Google Cloud |
| hashicorp | | pip install 'apache-airflow[hashicorp]' | | Hashicorp Services (Vault) |
| openai | | pip install 'apache-airflow[openai]' | | Open AI hooks and operators |
| opsgenie | | pip install 'apache-airflow[opsgenie]' | | OpsGenie hooks and operators |
| pagerduty | | pip install 'apache-airflow[pagerduty]' | | Pagerduty hook |
| pgvector | | pip install 'apache-airflow[pgvector]' | | pgvector operators and hook |
| pinecone | | pip install 'apache-airflow[pinecone]' | | Pinecone Operators and Hooks |
| qdrant | | pip install 'apache-airflow[qdrant]' | | Qdrant Operators and Hooks |
| salesforce | | pip install 'apache-airflow[salesforce]' | | Salesforce hook |
| sendgrid | | pip install 'apache-airflow[sendgrid]' | | Send email using sendgrid |
| segment | | pip install 'apache-airflow[segment]' | | Segment hooks and sensors |
| slack | | pip install 'apache-airflow[slack]' | | Slack hooks and operators |
| snowflake | | pip install 'apache-airflow[snowflake]' | | Snowflake hooks and operators |
| tableau | | pip install 'apache-airflow[tableau]' | | Tableau hooks and operators |
| tabular | | pip install 'apache-airflow[tabular]' | | Tabular hooks |
| telegram | | pip install 'apache-airflow[telegram]' | | Telegram hooks and operators |
| vertica | | pip install 'apache-airflow[vertica]' | | Vertica hook support as an Airflow backend |
| weaviate | | pip install 'apache-airflow[weaviate]' | | Weaviate hook and operators |
| yandex | | pip install 'apache-airflow[yandex]' | | Yandex.cloud hooks and operators |
| ydb | | pip install 'apache-airflow[ydb]' | | YDB hooks and operators |
| zendesk | | pip install 'apache-airflow[zendesk]' | | Zendesk hooks |

Locally installed software extras

These are extras that add dependencies needed for integration with other software packages installed usually as part of the deployment of Airflow. Some of those enable Airflow to use executors to run tasks with them - other than via the built-in LocalExecutor.

| extra | install command | brings | enables executors |
|-----------------|---|--|---|
| arangodb | <code>pip install 'apache-airflow[arangodb]'</code> | ArangoDB operators, sensors and hook | |
| celery | <code>pip install 'apache-airflow[celery]'</code> | Celery dependencies and sensor | CeleryExecutor, CeleryKubernetesExecutor |
| cnf-kubernetes | <code>pip install 'apache-airflow[cnfc-kubern</code> | Kubernetes client libraries, KubernetesPodOperator & friends | KubernetesExecutor, LocalKubernetesExecutor |
| docker | <code>pip install 'apache-airflow[docker]'</code> | Docker hooks and operators | |
| edge3 | <code>pip install 'apache-airflow[edge3]'</code> | Connect Edge Workers via HTTP to the scheduler | EdgeExecutor |
| elastic-search | <code>pip install 'apache-airflow[elasticsear</code> | Elasticsearch hooks and Log Han- | |
| exasol | <code>pip install 'apache-airflow[exasol]'</code> | Exasol hooks and operators | |
| fab | <code>pip install 'apache-airflow[fab]'</code> | FAB auth manager | |
| git | <code>pip install 'apache-airflow[git]'</code> | Git bundle and hook | |
| github | <code>pip install 'apache-airflow[github]'</code> | GitHub operators and hook | |
| influxdb | <code>pip install 'apache-airflow[influxdb]'</code> | Influxdb operators and hook | |
| jenkins | <code>pip install 'apache-airflow[jenkins]'</code> | Jenkins hooks and operators | |
| mongo | <code>pip install 'apache-airflow[mongo]'</code> | Mongo hooks and operators | |
| microsoft-mssql | <code>pip install 'apache-airflow[microsoft-m</code> | Microsoft SQL Server operators and hook. | |
| mysql | <code>pip install 'apache-airflow[mysql]'</code> | MySQL operators and hook | |
| neo4j | <code>pip install 'apache-airflow[neo4j]'</code> | Neo4j operators and hook | |
| odbc | <code>pip install 'apache-airflow[odbc]'</code> | ODBC data sources including MS SQL Server | |
| openfaas | <code>pip install 'apache-airflow[openfaas]'</code> | OpenFaaS hooks | |
| oracle | <code>pip install 'apache-airflow[oracle]'</code> | Oracle hooks and operators | |
| postgres | <code>pip install 'apache-airflow[postgres]'</code> | PostgreSQL operators and hook | |
| presto | <code>pip install 'apache-airflow[presto]'</code> | All Presto related operators & hooks | |
| redis | <code>pip install 'apache-airflow[redis]'</code> | Redis hooks and sensors | |
| samba | <code>pip install 'apache-airflow[samba]'</code> | Samba hooks and operators | |
| singularity | <code>pip install 'apache-airflow[singularity]</code> | Singularity container operator | |
| teradata | <code>pip install 'apache-airflow[teradata]'</code> | Teradata hooks and operators | |
| trino | <code>pip install 'apache-airflow[trino]'</code> | All Trino related operators & hooks | |

Other extras

These are extras that provide support for integration with external systems via some - usually - standard protocols.

The entries with * in the Preinstalled column indicate that those extras (providers) are always pre-installed when Airflow is installed.

| extra | install command | enables | Preinstalled |
|------------------|--|------------------------------------|--------------|
| common-compat | <code>pip install 'apache-airflow[common]</code> | Compatibility code for old Airflow | |
| common-io | <code>pip install 'apache-airflow[common]</code> | Core IO Operators | |
| common-messaging | <code>pip install 'apache-airflow[common]</code> | Core Messaging Operators | |
| common-sql | <code>pip install 'apache-airflow[common]</code> | Core SQL Operators | • |
| ftp | <code>pip install 'apache-airflow[ftp]</code> | FTP hooks and operators | • |
| grpc | <code>pip install 'apache-airflow[grpc]</code> | Grpc hooks and operators | |
| http | <code>pip install 'apache-airflow[http]</code> | HTTP hooks, operators and sensors | • |
| imap | <code>pip install 'apache-airflow[imap]</code> | IMAP hooks and sensors | • |
| jdbc | <code>pip install 'apache-airflow[jdbc]</code> | JDBC hooks and operators | |
| microsoft-psrp | <code>pip install 'apache-airflow[micro</code> | PSRP hooks and operators | |
| microsoft-winrm | <code>pip install 'apache-airflow[micro</code> | WinRM hooks and operators | |
| openlineage | <code>pip install 'apache-airflow[openl</code> | Sending OpenLineage events | |
| opensearch | <code>pip install 'apache-airflow[opens</code> | Opensearch hooks and operators | |
| papermill | <code>pip install 'apache-airflow[paper</code> | Papermill hooks and operators | |
| sftp | <code>pip install 'apache-airflow[sftp]</code> | SFTP hooks, operators and sensors | |
| smtp | <code>pip install 'apache-airflow[smtp]</code> | SMTP hooks and operators | |
| sqlite | <code>pip install 'apache-airflow[sqlit</code> | SQLite hooks and operators | • |
| ssh | <code>pip install 'apache-airflow[ssh]</code> | SSH hooks and operators | |

3.25.4 Group extras

The group extras are convenience extras. Such extra installs many optional dependencies together. It is not recommended to use it in production, but it is useful for CI, development and testing purposes.

| extra | install command | enables |
|----------|---|---|
| all | <code>pip install apache-airflow[all]</code> | All optional dependencies including all providers |
| all-core | <code>pip install apache-airflow[all-core]</code> | All optional core dependencies |

3.26 Reference for Database Migrations

Here's the list of all the Database Migrations that are executed via when you run `airflow db migrate`.

⚠ Warning

Those migration details are mostly used here to make the users aware when and what kind of migrations will be executed during migrations between specific Airflow versions. The intention here is that the “DB conscious” users might perform an analysis on the migrations and draw conclusions about the impact of the migrations on their Airflow database. Those users might also want to take a look at the *ERD Schema of the Database* document to understand how the internal DB of Airflow structure looks like. However, you should be aware that the structure is internal and you should not access the DB directly to retrieve or modify any data - you should use the *REST API* to do that instead.

| Revision ID | Revises ID | Airflow Version | Description |
|---------------------|--------------|-----------------|---|
| 29ce7909c52b (head) | 959e216a3abb | 3.0.0 | Change TI table to have unique UUID id/pk per attempt. |
| 959e216a3abb | 0e9519b56710 | 3.0.0 | Rename <code>is_active</code> to <code>is_stale</code> column in dag table. |
| 0e9519b56710 | ec62e120484d | 3.0.0 | Rename run_type from ‘dataset_triggered’ to ‘asset_triggered’ in dag table. |
| ec62e120484d | be2cc2f742cf | 3.0.0 | Add new otel span fields. |
| be2cc2f742cf | d469d27e2a64 | 3.0.0 | Support bundles in DagPriorityParsingRequest. |
| d469d27e2a64 | 16f7f5ee874e | 3.0.0 | Use <code>ti_id</code> as FK to TaskReschedule. |
| 16f7f5ee874e | cf87489a35df | 3.0.0 | Remove dag.default_view column. |
| cf87489a35df | 7645189f3479 | 3.0.0 | Use TI.id as primary key to TaskInstanceNote. |
| 7645189f3479 | e00344393f31 | 3.0.0 | Add try_id to TI and TIH. |
| e00344393f31 | 6a9e7a527a88 | 3.0.0 | remove external_trigger field. |
| 6a9e7a527a88 | 33b04e4bfa19 | 3.0.0 | Add DagRun run_after. |
| 33b04e4bfa19 | 8ea135928435 | 3.0.0 | add new task_instance field scheduled_dttm. |
| 8ea135928435 | e39a26ac59f6 | 3.0.0 | Add relative fileloc column. |
| e39a26ac59f6 | 38770795785f | 3.0.0 | remove pickled data from dagrun table. |
| 38770795785f | 5c9c0231baa2 | 3.0.0 | Add asset reference models. |
| 5c9c0231baa2 | 237cef8dfa1 | 3.0.0 | Remove processor_subdir. |
| 237cef8dfa1 | 038dc8bc6284 | 3.0.0 | Add deadline alerts table. |
| 038dc8bc6284 | e229247a6cb1 | 3.0.0 | update trigger_timeout column in task_instance table to UTC. |
| e229247a6cb1 | eed27faa34e3 | 3.0.0 | Add DagBundleModel. |
| eed27faa34e3 | 9fc3fc5de720 | 3.0.0 | Remove pickled data from xcom table. |
| 9fc3fc5de720 | 2b47dc6bc8df | 3.0.0 | Add references between assets and triggers. |
| 2b47dc6bc8df | d03e4a635aa3 | 3.0.0 | add dag versioning. |
| d03e4a635aa3 | d8cd3297971e | 3.0.0 | Drop DAG pickling. |
| d8cd3297971e | 5f57a45b8433 | 3.0.0 | Add last_heartbeat_at directly to TI. |
| 5f57a45b8433 | 486ac7936b78 | 3.0.0 | Drop task_fail table. |
| 486ac7936b78 | d59cbbef95eb | 3.0.0 | remove scheduler_lock column. |
| d59cbbef95eb | 05234396c6fc | 3.0.0 | Add UUID primary key to task_instance table. |

continues on next page

Table 109 – continued from previous page

| Revision ID | Revises ID | Airflow Version | Description |
|---------------------|--------------|-----------------|--|
| 05234396c6fc | 3a8972ecb8f9 | 3.0.0 | Rename dataset as asset. |
| 3a8972ecb8f9 | fb2d4922cd79 | 3.0.0 | Add exception_reason and logical_date to BackfillDagRun. |
| fb2d4922cd79 | 5a5d66100783 | 3.0.0 | Tweak AssetAliasModel to match AssetModel after AIP-76. |
| 5a5d66100783 | c3389cd7793f | 3.0.0 | Add AssetActive to track orphaning instead of a flag. |
| c3389cd7793f | 0d9e73a75ee4 | 3.0.0 | Add backfill to dag run model. |
| 0d9e73a75ee4 | 44eabb1904b4 | 3.0.0 | Add name and group fields to DatasetModel. |
| 44eabb1904b4 | 16cbc1c8c36 | 3.0.0 | Update dag_run_note.user_id and task_instance_note.user_id columns |
| 16cbc1c8c36 | 522625f6d606 | 3.0.0 | Remove redundant index. |
| 522625f6d606 | 1cdc775ca98f | 3.0.0 | Add tables for backfill. |
| 1cdc775ca98f | a2c32e6c7729 | 3.0.0 | Rename execution_date to logical_date. |
| a2c32e6c7729 | 0bfc26bc256e | 3.0.0 | Add triggered_by field to DagRun. |
| 0bfc26bc256e | d0f1c5954fa | 3.0.0 | Rename DagModel schedule_interval to timetable_summary. |
| d0f1c5954fa | 044f740568ec | 3.0.0 | Remove SubDAGs: is_subdag & root_dag_id columns from DAG |
| 044f740568ec | 5f2621c13b39 | 3.0.0 | Drop ab_user.id foreign key. |
| 5f2621c13b39 | 22ed7efa9da2 | 2.10.3 | Rename dag_schedule_dataset_alias_reference constraint names. |
| 22ed7efa9da2 | 8684e37832e6 | 2.10.0 | Add dag_schedule_dataset_alias_reference table. |
| 8684e37832e6 | 41b3bc7c0272 | 2.10.0 | Add dataset_alias_dataset association table. |
| 41b3bc7c0272 | ec3471c1e067 | 2.10.0 | Add try_number to audit log. |
| ec3471c1e067 | 05e19f3176be | 2.10.0 | Add dataset_alias_dataset_event. |
| 05e19f3176be | d482b7261ff9 | 2.10.0 | Add dataset_alias. |
| d482b7261ff9 | c4602ba06b4b | 2.10.0 | Add task_instance_history. |
| c4602ba06b4b | 677fdbb7fc54 | 2.10.0 | Added DagPriorityParsingRequest table. |
| 677fdbb7fc54 | 0fd0c178cbe8 | 2.10.0 | add new executor field to db. |
| 0fd0c178cbe8 | 686269002441 | 2.10.0 | Add indexes on dag_id column in referencing tables. |
| 686269002441 | bff083ad727d | 2.9.2 | Fix inconsistency between ORM and migration files. |
| bff083ad727d | 1949afb29106 | 2.9.2 | Remove idx_last_scheduling_decision index on last_scheduling |
| 1949afb29106 | ee1467d4aa35 | 2.9.0 | update trigger kwargs type and encrypt. |
| ee1467d4aa35 | b4078ac230a1 | 2.9.0 | add display name for dag and task instance. |
| b4078ac230a1 | 8e1c784a4fc7 | 2.9.0 | Change value column type to longblob in xcom table for mysql. |
| 8e1c784a4fc7 | ab34f260b71c | 2.9.0 | Adding max_consecutive_failed_dag_runs column to dag_model table |
| ab34f260b71c | d75389605139 | 2.9.0 | add dataset_expression in DagModel. |
| d75389605139 | 1fd565369930 | 2.9.0 | Add run_id to (Audit) log table and increase event name length. |
| 1fd565369930 | 88344c1d9134 | 2.9.0 | Add rendered_map_index to TaskInstance. |
| 88344c1d9134 | 10b52ebd31f7 | 2.8.1 | Drop unused TI index. |
| 10b52ebd31f7 | bd5dfbe21f88 | 2.8.0 | Add processor_subdir to ImportError. |
| bd5dfbe21f88 | f7bf2a57d0a6 | 2.8.0 | Make connection login/password TEXT. |
| f7bf2a57d0a6 | 375a816bbbf4 | 2.8.0 | Add owner_display_name to (Audit) Log table. |
| 375a816bbbf4 | 405de8318b3a | 2.8.0 | add new field 'clear_number' to dagrun. |
| 405de8318b3a | 788397e78828 | 2.7.0 | add include_deferred column to pool. |
| 788397e78828 | 937cbd173ca1 | 2.7.0 | Add custom_operator_name column. |
| 937cbd173ca1 (base) | None | 2.7.0 | Add index to task_instance table. |

3.27 ERD Schema of the Database

Here is the current Database schema diagram.

⚠ Warning

The ER diagram shows the snapshot of the database structure valid for Airflow version 3.1.0 and it should be treated as an internal detail. It might be changed at any time and you should not directly access the database to retrieve information from it or modify the data - you should use *Airflow public API reference* to do that instead. The main purpose of this diagram is to help with troubleshooting and understanding of the internal Airflow DB architecture in case you have any problems with the database - for example when dealing with problems with migrations. See also *Reference for Database Migrations* for list of detailed database migrations that are applied when running migration script and `db command` for the commands that you can use to manage the migrations.

PYTHON MODULE INDEX

a ??
airflow.config_templates.airflow_local_settings, airflow.example_dags.example_setup_teardown,
?? airflow.example_dags.example_setup_teardown_taskflow,
airflow.decorators, ?? ??
airflow.example_dags, ?? airflow.example_dags.example_simplest_dag, ??
airflow.example_dags.example_asset_alias, ?? airflow.example_dags.example_skip_dag, ??
airflow.example_dags.example_asset_alias_with_no_taskflow, ?? airflow.example_dags.example_task_group, ??
airflow.example_dags.example_asset_decorator, ?? airflow.example_dags.example_task_group_decorator,
??
airflow.example_dags.example_asset_with_watchers, ?? airflow.example_dags.example_time_delta_sensor_async,
airflow.example_dags.example_assets, ?? airflow.example_dags.example_trigger_target_dag,
airflow.example_dags.example_branch_labels, ?? airflow.example_dags.example_workday_timetable,
?? airflow.example_dags.example_branch_python_dop_operator_3, ??
airflow.example_dags.example_complex, ?? airflow.example_dags.example_xcomargs, ??
airflow.example_dags.example_custom_weight, ?? airflow.example_dags.libs, ??
airflow.example_dags.example_display_name, ?? airflow.example_dags.libs.helper, ??
airflow.example_dags.example_dynamic_task_mapping, ?? airflow.example_dags.plugins, ??
airflow.example_dags.example_dynamic_task_mapping, ?? airflow.example_dags.plugins.decreasing_priority_weight_st
?? airflow.example_dags.plugins.event_listener, ??
airflow.example_dags.example_inlet_event_extra, ?? airflow.example_dags.plugins.listener_plugin,
?? airflow.example_dags.example_kubernetes_executor, ?? airflow.example_dags.plugins.workday, ??
?? airflow.example_dags.example_kubernetes_executor, ?? airflow.example_dags.tutorial, ??
airflow.example_dags.example_latest_only_with_trigger, ?? airflow.example_dags.tutorial_dag, ??
?? airflow.example_dags.tutorial_objectstorage, ??
airflow.example_dags.example_local_kubernetes_executor, ?? airflow.example_dags.tutorial_taskflow_api,
?? airflow.example_dags.example_nested_branch_dag, ?? airflow.example_dags.tutorial_taskflow_api_virtualenv,
airflow.example_dags.example_outlet_event_extra, ?? airflow.example_dags.tutorial_taskflow_templates,
?? airflow.example_dags.example_params_trigger_ui, ?? airflow.exceptions, ??
airflow.example_dags.example_params_ui_tutorial, ?? airflow.hooks, ??
airflow.example_dags.example_params_ui_tutorial, ?? airflow.hooks.base, ??
airflow.example_dags.example_passing_params_via_test_command, ?? airflow.macros, ??

```
airflow.models.connection, ??  
airflow.models.dag, ??  
airflow.models.dagbag, ??  
airflow.models.dagrun, ??  
airflow.models.param, ??  
airflow.models.taskinstance, ??  
airflow.models.taskinstancekey, ??  
airflow.models.variable, ??  
airflow.models.xcom, ??  
airflow.policies, ??  
airflow.secrets, ??  
airflow.secrets.base_secrets, ??  
airflow.secrets.cache, ??  
airflow.secrets.environment_variables, ??  
airflow.secrets.local_filesystem, ??  
airflow.secrets.metastore, ??  
airflow.timetables, ??  
airflow.timetables.assets, ??  
airflow.timetables.base, ??  
airflow.timetables.datasets, ??  
airflow.timetables.events, ??  
airflow.timetables.interval, ??  
airflow.timetables.simple, ??  
airflow.timetables.trigger, ??  
airflow.triggers, ??  
airflow.triggers.base, ??  
airflow.triggers.testing, ??  
airflow.utils.state, ??
```