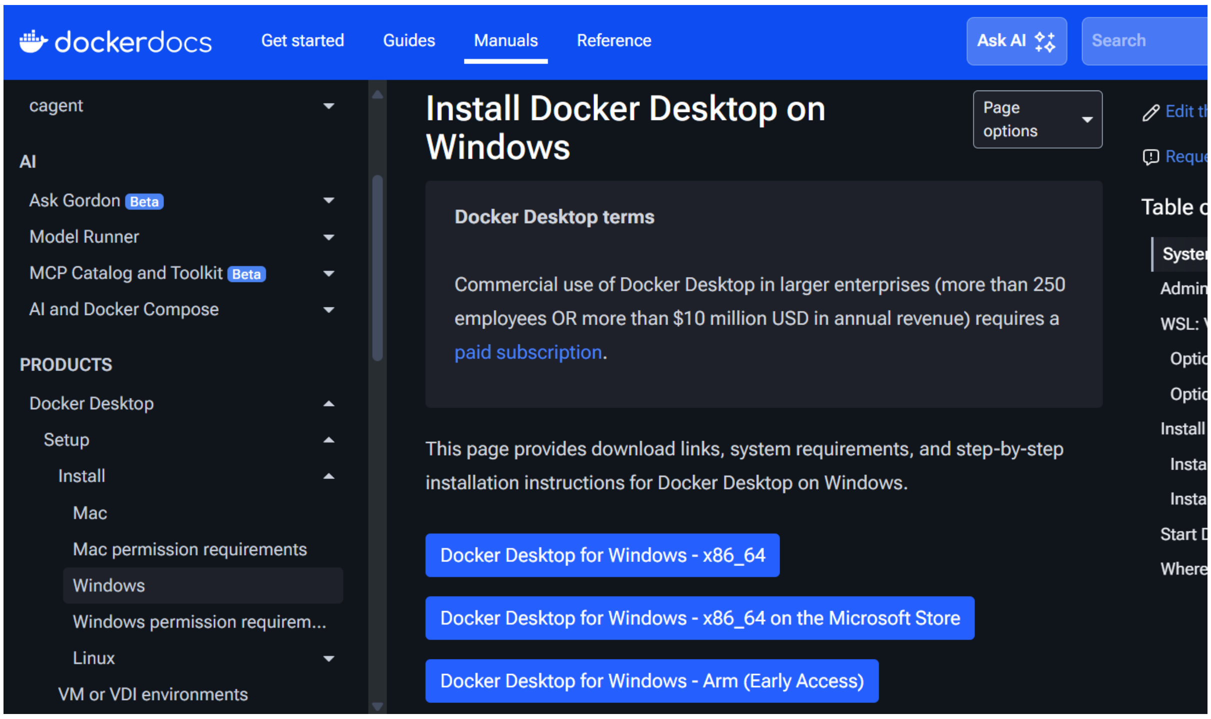
Airflow

Airflow installation

Step-1: Install Docker Windows | Docker Docs



Install Docker Desktop on Windows

Docker Desktop terms

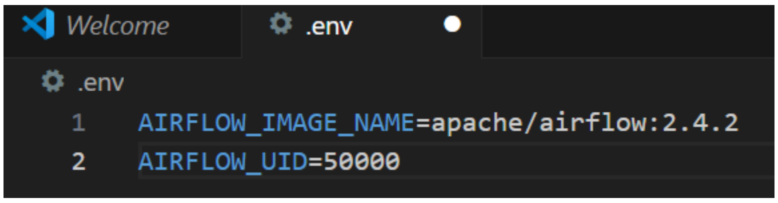
Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than $10 million USD in annual revenue) requires a paid subscription.

This page provides download links, system requirements, and step-by-step installation instructions for Docker Desktop on Windows.

Restart it

Step-2: This PC → users → your-name → create material folder → inside that you paste this file.

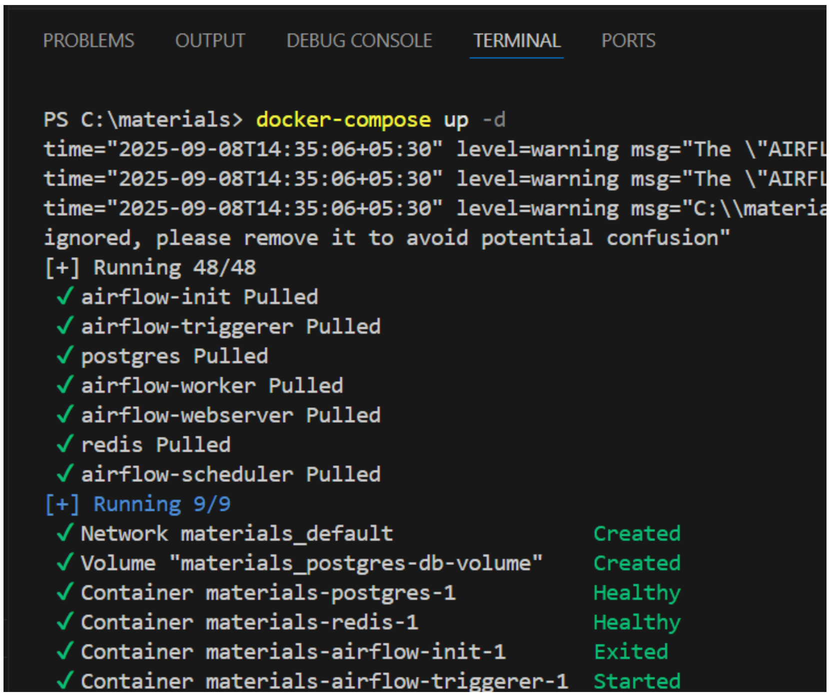
Step-3: open material folder in VS code → create **.env** file



```
AIRFLOW_IMAGE_NAME=apache/airflow:2.4.2
AIRFLOW_UID=50000
```

Step-4: open terminal and run command

docker-compose up -d

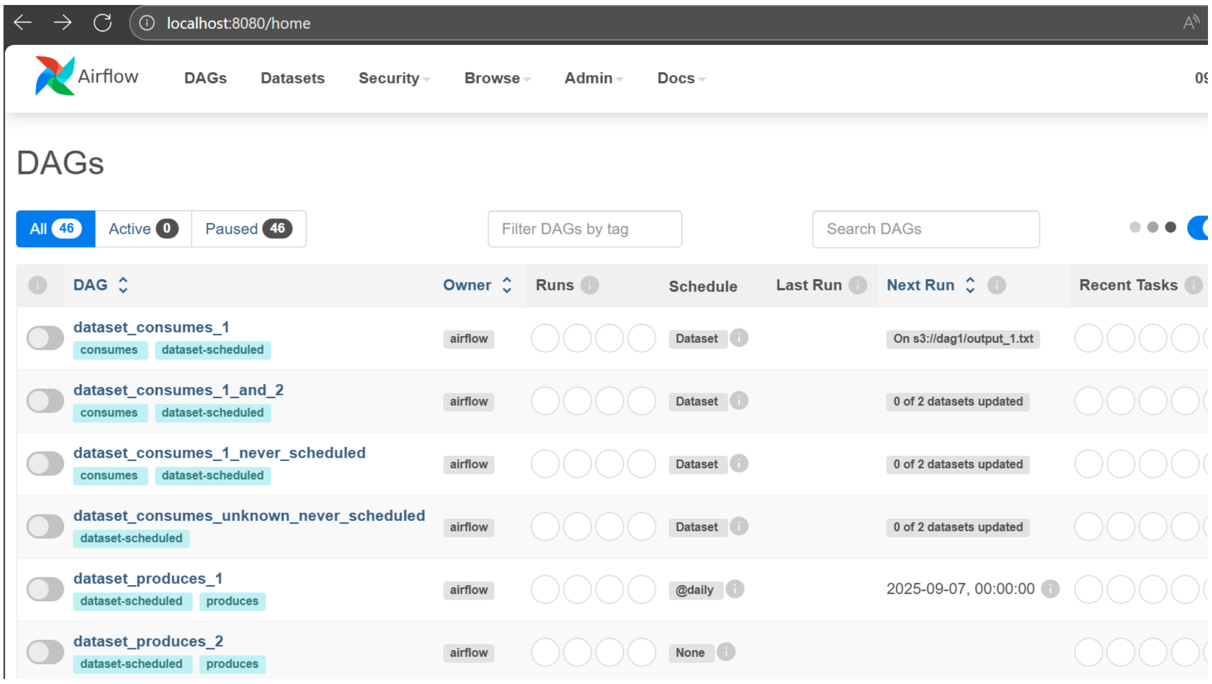


```
PS C:\materials> docker-compose up -d
time="2025-09-08T14:35:06+05:30" level=warning msg="The \"AIRFL
time="2025-09-08T14:35:06+05:30" level=warning msg="The \"AIRFL
time="2025-09-08T14:35:06+05:30" level=warning msg="C:\\materia
ignored, please remove it to avoid potential confusion"
[+] Running 48/48
 ✓ airflow-init Pulled
 ✓ airflow-triggerer Pulled
 ✓ postgres Pulled
 ✓ airflow-worker Pulled
 ✓ airflow-webserver Pulled
 ✓ redis Pulled
 ✓ airflow-scheduler Pulled
[+] Running 9/9
 ✓ Network materials_default              Created
 ✓ Volume "materials_postgres-db-volume"  Created
 ✓ Container materials-postgres-1         Healthy
 ✓ Container materials-redis-1            Healthy
 ✓ Container materials-airflow-init-1     Exited
 ✓ Container materials-airflow-triggerer-1  Started
```

Step-5: setup username and password

docker-compose run airflow-worker airflow users create --role Admin --username admin --email admin --firstname admin --lastname admin --password admin

step-6: http://localhost:8080



DAGs

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
dataset_consumes_1 (consumes, dataset-scheduled)	airflow		Dataset		On s3://dag1/output_1.txt	
dataset_consumes_1_and_2 (consumes, dataset-scheduled)	airflow		Dataset		0 of 2 datasets updated	
dataset_consumes_1_never_scheduled (consumes, dataset-scheduled)	airflow		Dataset		0 of 2 datasets updated	
dataset_consumes_unknown_never_scheduled (dataset-scheduled)	airflow		Dataset		0 of 2 datasets updated	
dataset_produces_1 (dataset-scheduled, produces)	airflow		@daily		2025-09-07, 00:00:00	
dataset_produces_2 (dataset-scheduled, produces)	airflow		None			

What is airflow?

- Apache airflow is an open-source platform for developing, scheduling and monitoring batch-oriented workflow.
- Airflow's extensible python framework enables you to build workflows.
- 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.

Apache Airflow is designed to build, schedule, and oversee complex workflows and dependencies.

The main advantage of **Airflow** is we can able to create the workflows has piece of **Python code** and also Airflow is **Extensible framework** allows us to connect with any latest technology and use it.

- Think of it as a **scheduler + orchestrator** for tasks.
- Workflows are defined as **DAGs (Directed Acyclic Graphs)** in Python.
- Each step in your data process (extract, transform, load, ML training, reporting, etc.) is a **task**.
- Airflow handles **dependencies, scheduling, retries, logging, and monitoring** of these tasks.

Workflows in Airflow are defined as DAGs (Directed Acyclic Graphs), where each node is a task and edges define dependencies.

Key Points:

- **Workflow as Code**: Workflows are written in Python as **DAGs** (Directed Acyclic Graphs).
- **Scheduling**: Automates running pipelines at specific intervals (hourly, daily, etc.).
- **Orchestration**: Manages dependencies between tasks, ensuring they run in the right order.
- **Monitoring**: Provides a rich **web UI** to track the status of tasks and retry failures.

- **Extensible**: Supports plugins and integrations (databases, cloud services, Spark, Hadoop, etc.).

Why we need Airflow?

Let's consider an **example,** suppose a **company-X** is running **2 or 3 pipelines** per day if an **error** encounters in the pipeline, we will **debug** the error easily and **rerun** the pipeline and resolve it.

What if we **need to run hundreds of data pipelines per day** and orchestrate them if an error occurs in meantime it takes longer time to resolve error and to debug it. As the data pipelines increases **it looks difficult for us to understand the dependencies** in between the **workflows involved in hundreds of data pipelines.**

Now we can able to **improve efficiency** easily using **Airflow** by **testing** before **deploying** the python code and debugging it with correct code and reduce time.

Workflows can also be stored in **version control** systems to **roll back** previous versions. We can validate our own required functionalities for a DAG by **writing specific tests.**

Airflow’s key features include:

- **Pipeline scheduling:** Automates workflow execution based on defined schedules.
- **Dependency management:** Ensures tasks run in the correct order.
- **Real-time monitoring:** Provides visibility into workflow health and performance.
- **Scalability**: Handles workflows at any scale, from small to enterprise-level.
- Dynamic pipelines: pipeline are written in python, enabling dynamic DAGs, custom logic, and integration with any code or API.

Why do we need Airflow? / What problems does it solve?

1. **Workflow Orchestration** – Manages dependencies between tasks (e.g., extract → transform → load).
2. **Scheduling** – Provides advanced scheduling beyond cron jobs (daily, hourly, event-driven).
3. **Scalability** – Can distribute workloads across multiple workers (good for large data pipelines).
4. **Monitoring & Alerting** – Has a UI to track job status, retries, and failures; supports alerting.
5. **Flexibility** – Pipelines are written in Python, making them dynamic and easy to maintain.
6. **Reliability** – Handles retries, backfills, and ensures workflows complete successfully.
7. **Integration** – Works with databases, cloud services (AWS/GCP/Azure), Spark, Hadoop, etc.

DAG stands for **Directed Acyclic Graph**.

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.

Definition

- Directed → task moves in one direction

  The tasks in the DAG have dependencies and are connected in a specific direction. The direction indicates the order in which to execute the tasks.

- Acyclic → no cycles/loops; task do not move in circle.

This means a DAG doesn't contain any cycles or loops. In other words, you can't create a circular dependency between tasks, as that would lead to an infinite loop.

- Graph (visual representation of different task) → collection of nodes (tasks) and edges (dependencies).

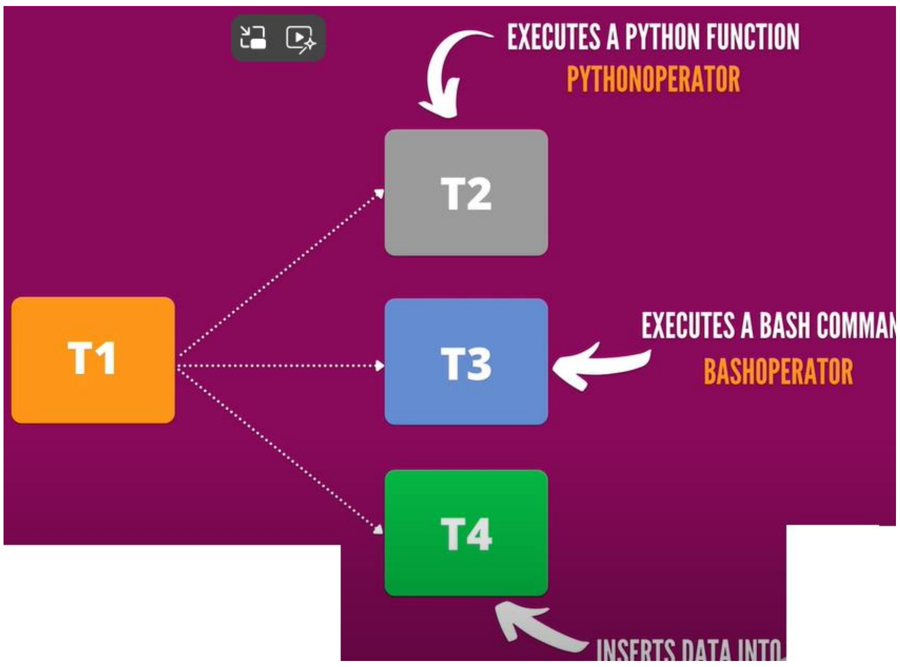
  DAG is a graph structure,

  - where the tasks are represented as nodes, and
  - the dependencies between tasks are represented as edges.

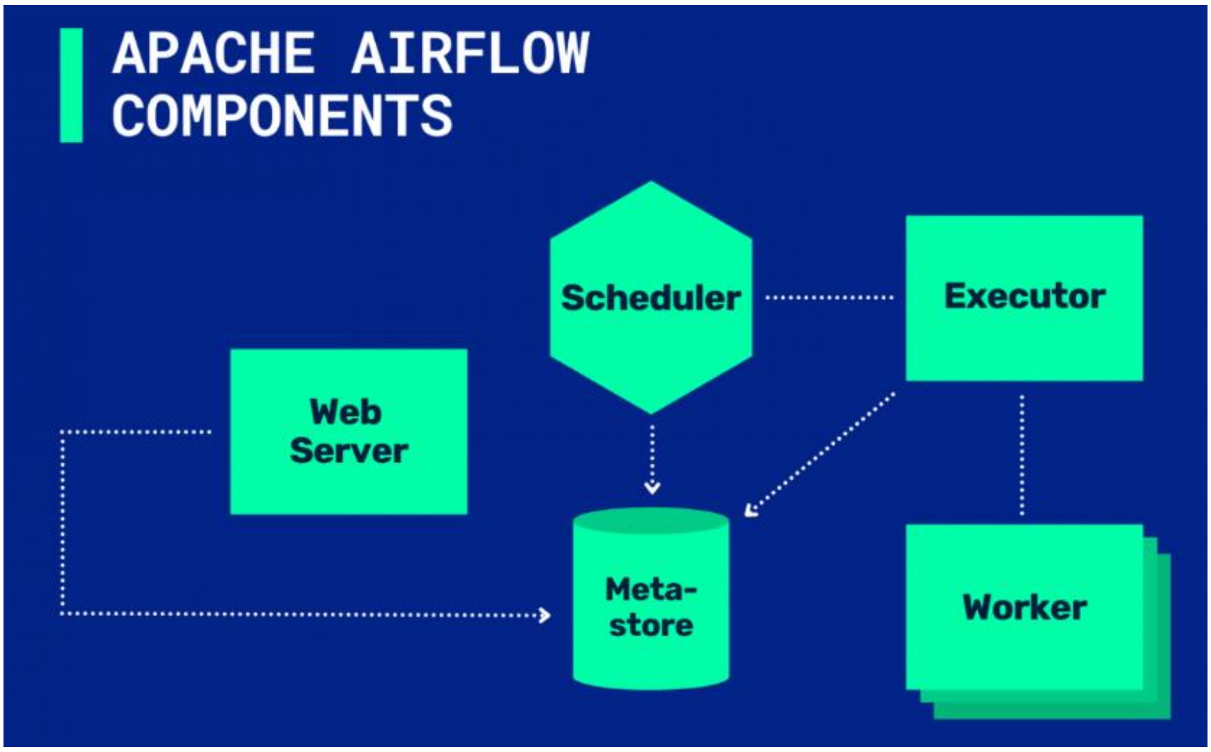
In Airflow, a DAG represents the entire workflow (your pipeline). Each task is a node, and dependencies between tasks form the edges.

In Airflow, you define a DAG using Python.

The dependencies between tasks are defined using shift operators (<< and >>) or by using the set_upstream and set_downstream methods. These dependency declarations specify the order in which tasks should execute based on their relationships.



Core components of Airflow



At its core, Airflow consists of the following architectural components:

1. **Scheduler**
2. **Executor**
3. **Workers**
4. **Metadata Database**
5. **Web Server**
6. **Task Queues**
7. **DAGs**

Scheduler

The scheduler is the brain of Airflow. Its role is to continuously monitor DAGs and schedule tasks based on the defined execution time and dependencies. The scheduler determines what tasks should run, when they should run, and submits them to the executor for execution.

In simple: continuously monitors all DAGs, schedules tasks based on timing and dependencies, and submits task for execution.

Executor

The executor is responsible for executing the tasks that the scheduler schedules. There are several types of executors available, depending on use case.

- **Sequential Executor**: simple executor, it executes one task at a time. Not suitable for large-scale workflows, often used for testing purposes or very small workflows where parallel execution isn’t necessary.
- **Local executor**: executes tasks on the same machine where the airflow instance runs. Ideal for small workloads. (runs tasks in parallel)
- **Celery Executor**: allows distributing tasks across multiple worker machines, making it suitable for large-scale workflows.
- **Kubernetes executor**: executes tasks in Kubernetes pods, providing excellent scalability and isolation.

Executors manage the execution of tasks and ensure their successful completion.

In short: Executes the tasks scheduled by the Scheduler. The Executor's configuration determines whether the tasks run locally or are distributed across workers.

Workers

Workers are the machines where the actual tasks are executed. They perform the processing by pulling tasks from the task queue and runs them.

Metadata Database

It stores all the metadata about DAGs, tasks, task instances, execution history, logs, and other configurations.

Airflow typically uses a relational database like PostgreSQL or MySQL for metadata storage.

This database helps airflow to keep track of which tasks are running, failed or completed, allowing it to retry failed tasks and manage complex workflows.

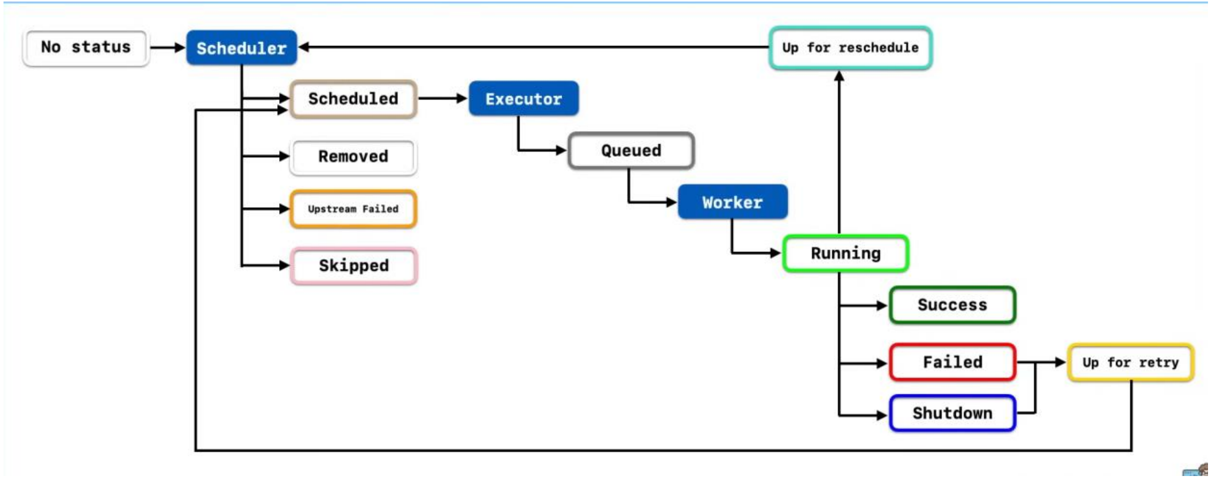
Web Server

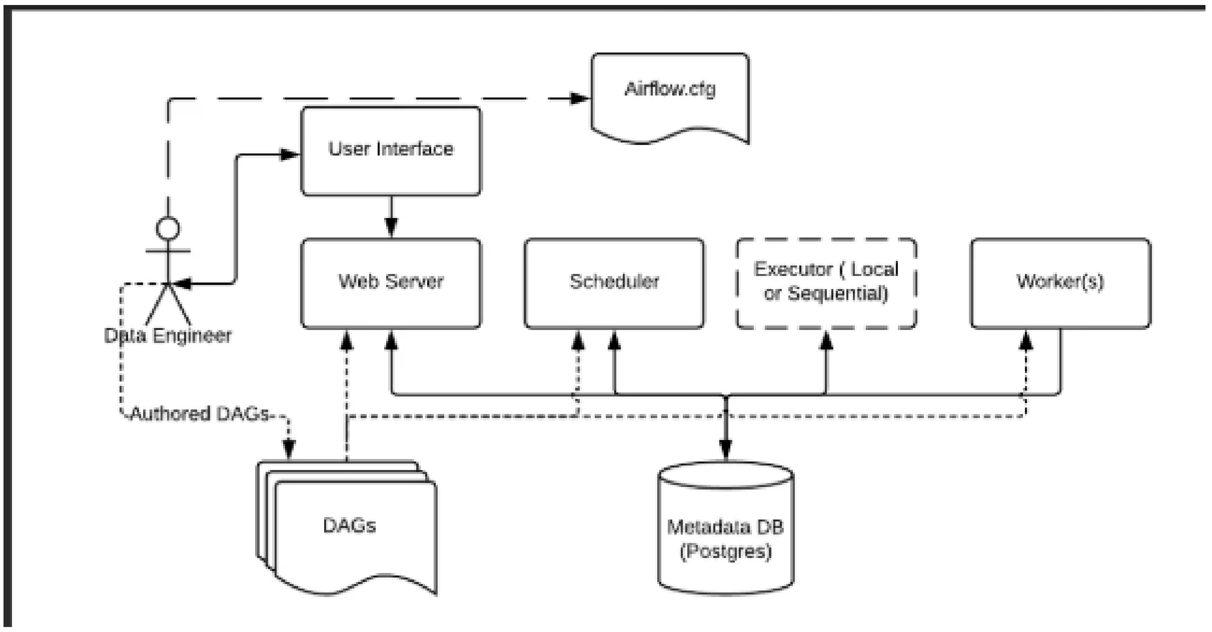
The Web Server component provides a user interface to interact with Airflow. It’s built using Flask and allows users to view DAGs, monitor task progress, check logs, and manage workflows. The intuitive UI enables easy management of workflows without writing additional code.

Some key features of the Web UI include:

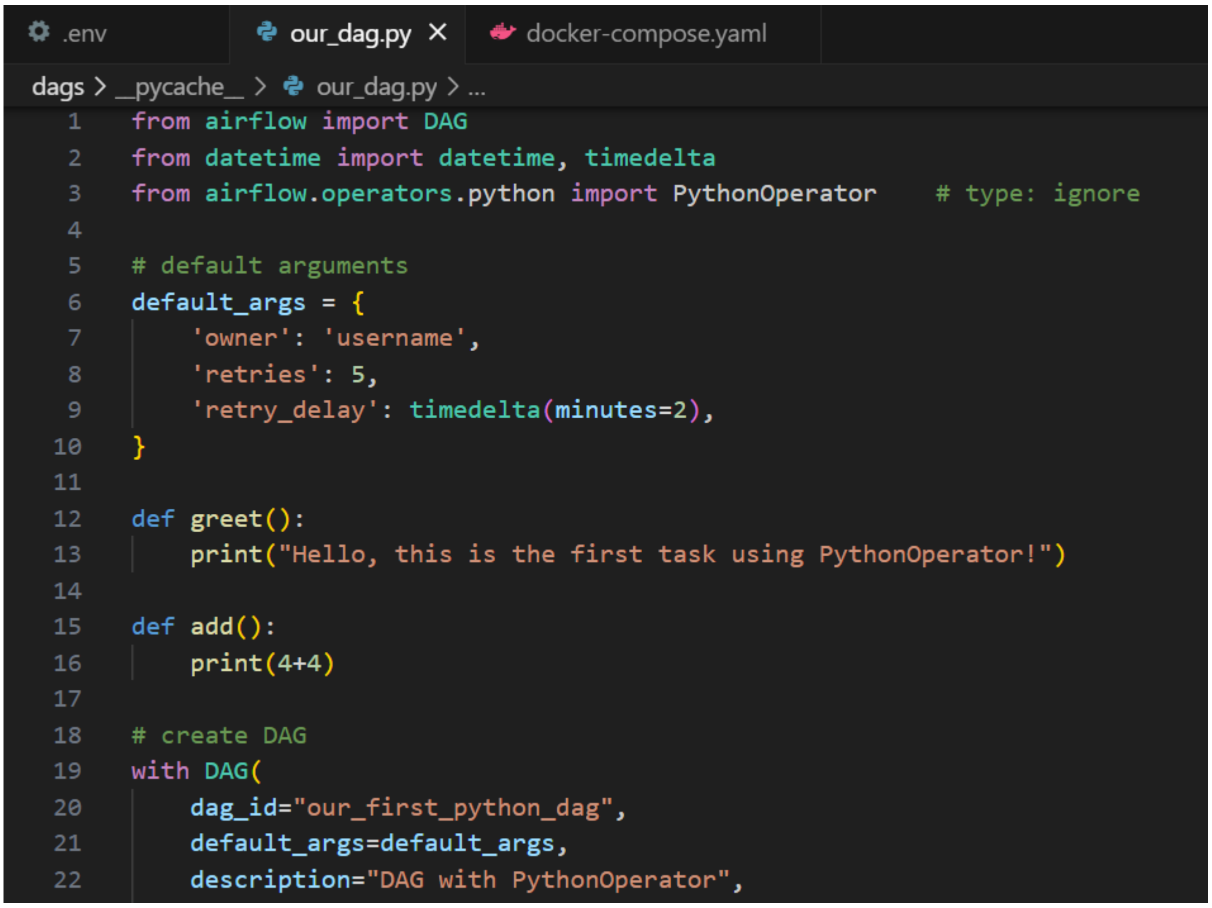
- **DAG Visualization**: Provides a graphical representation of DAGs and tasks.
- **Task Monitoring**: Shows real-time task execution status (running, queued, failed).
- **Logs Access**: Enables easy access to task logs for debugging.

Task lifecycle

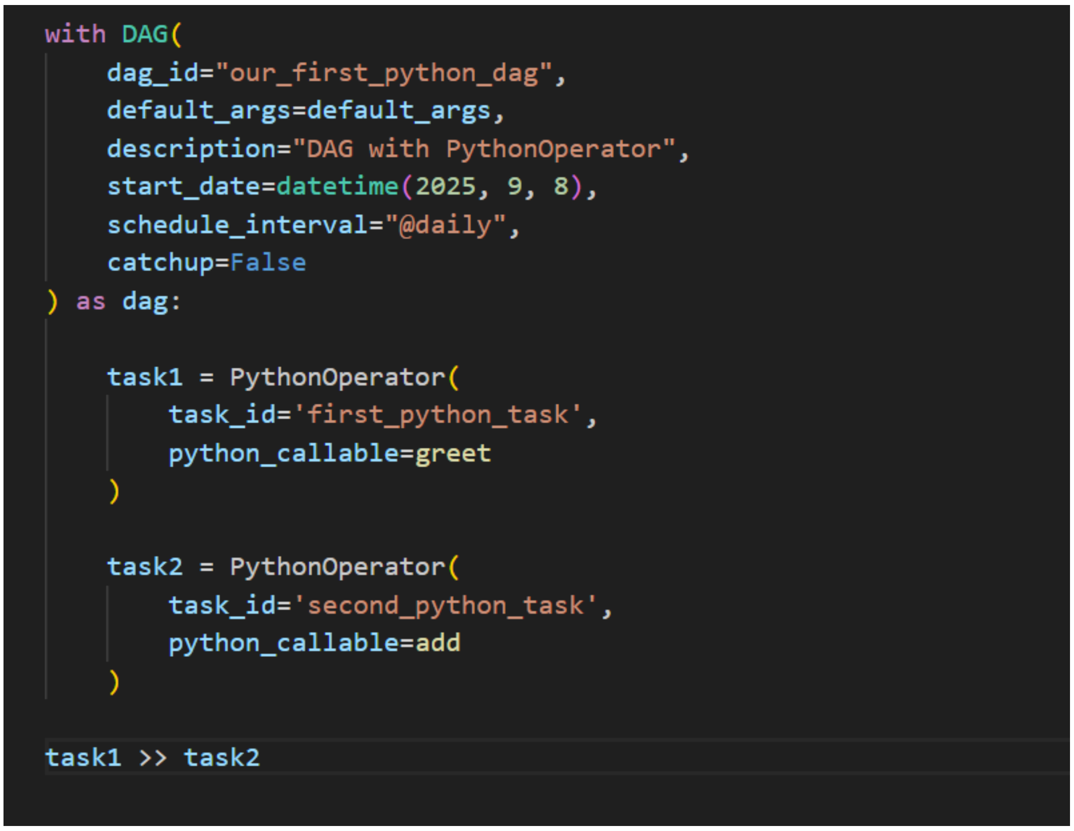




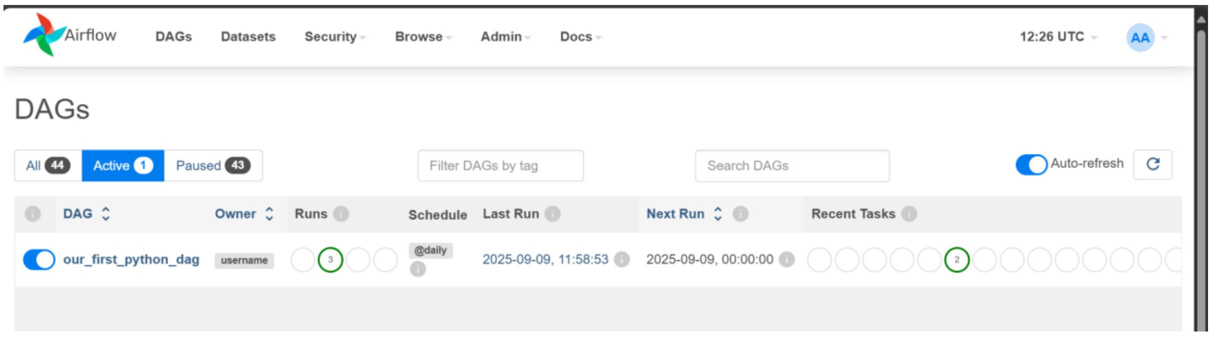
DAG using python operator



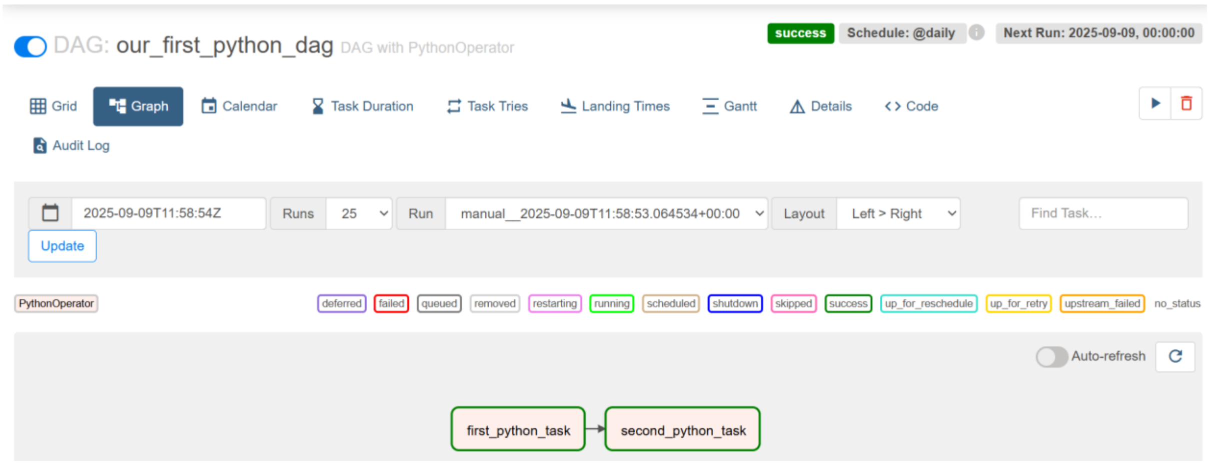
```python
from airflow import DAG
from datetime import datetime, timedelta
from airflow.operators.python import PythonOperator    # type: ignore

# default arguments
default_args = {
    'owner': 'username',
    'retries': 5,
    'retry_delay': timedelta(minutes=2),
}

def greet():
    print("Hello, this is the first task using PythonOperator!")

def add():
    print(4+4)

# create DAG
with DAG(
    dag_id="our_first_python_dag",
    default_args=default_args,
    description="DAG with PythonOperator",
```

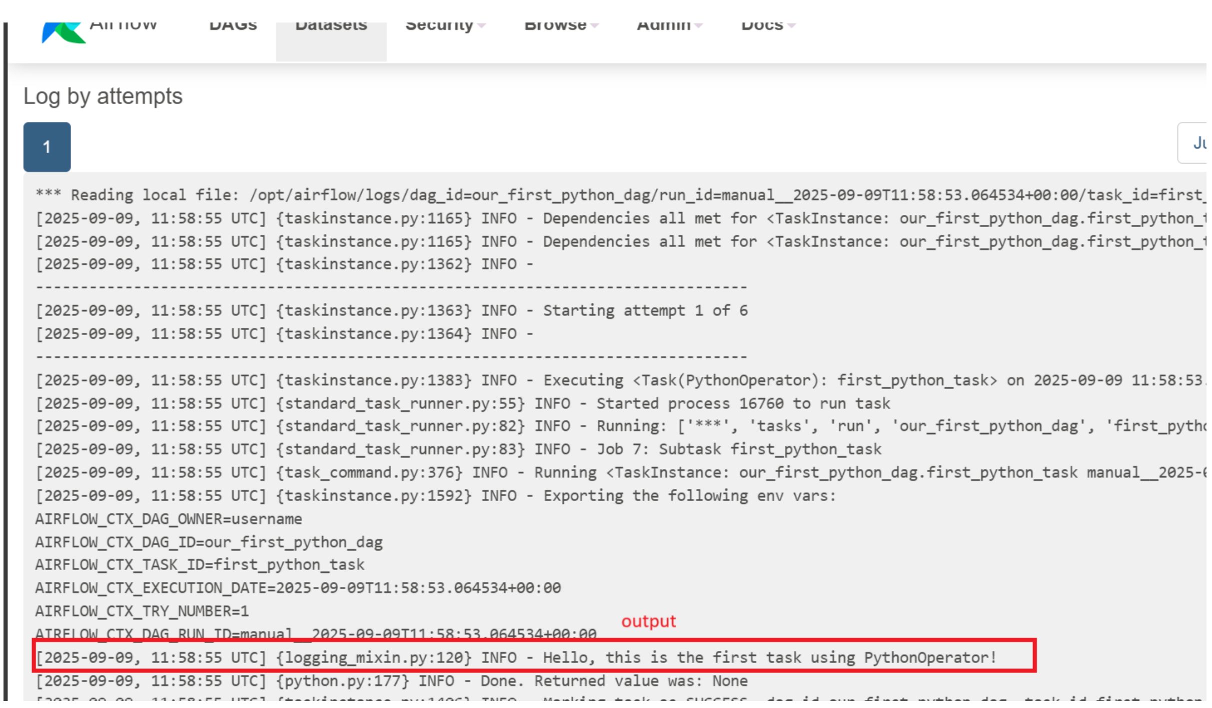


```python
with DAG(
    dag_id="our_first_python_dag",
    default_args=default_args,
    description="DAG with PythonOperator",
    start_date=datetime(2025, 9, 8),
    schedule_interval="@daily",
    catchup=False
) as dag:

    task1 = PythonOperator(
        task_id='first_python_task',
        python_callable=greet
    )

    task2 = PythonOperator(
        task_id='second_python_task',
        python_callable=add
    )

task1 >> task2
```



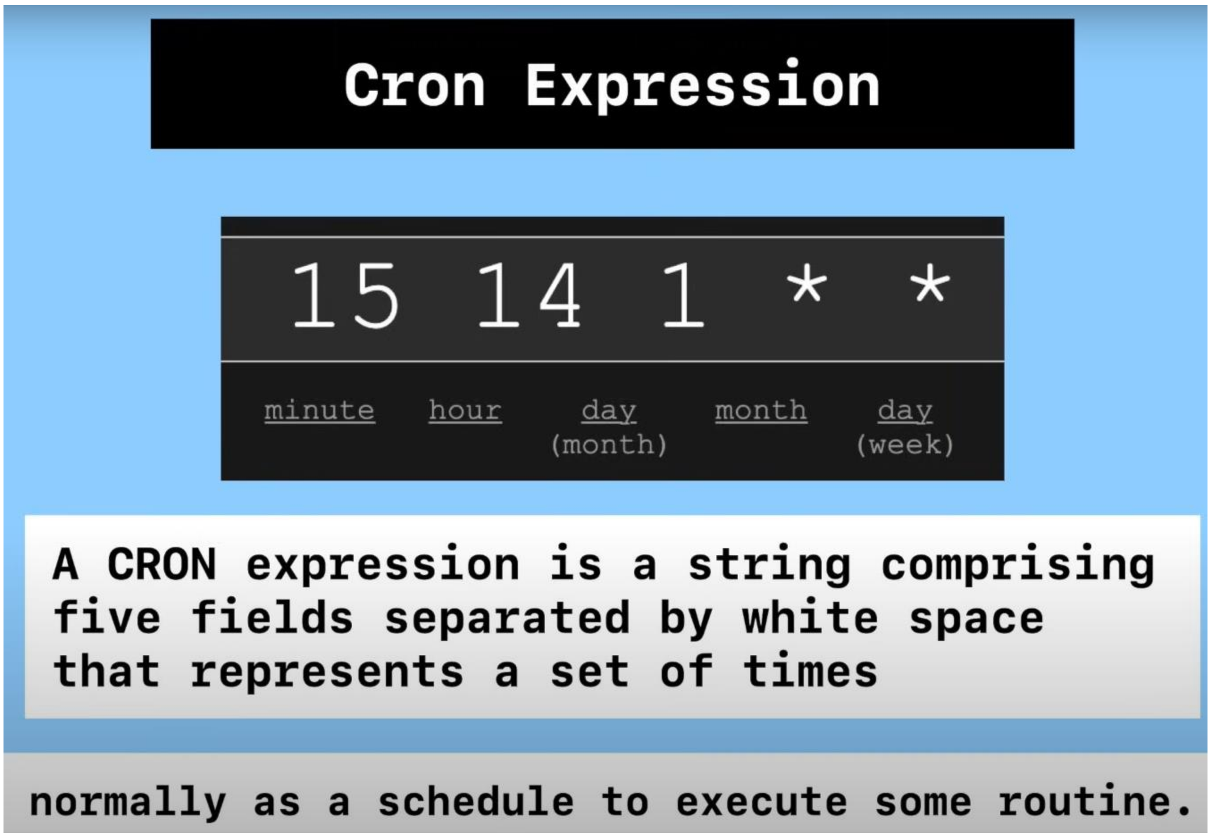
DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
our_first_python_dag	username	3	@daily	2025-09-09, 11:58:53	2025-09-09, 00:00:00	2





```
*** Reading local file: /opt/airflow/logs/dag_id=our_first_python_dag/run_id=manual__2025-09-09T11:58:53.064534+00:00/task_id=first_
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1165} INFO - Dependencies all met for <TaskInstance: our_first_python_dag.first_python_
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1165} INFO - Dependencies all met for <TaskInstance: our_first_python_dag.first_python_
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1362} INFO -
--------------------------------------------------------------------------------
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1363} INFO - Starting attempt 1 of 6
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1364} INFO -
--------------------------------------------------------------------------------
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1383} INFO - Executing <Task(PythonOperator): first_python_task> on 2025-09-09 11:58:53.
[2025-09-09, 11:58:55 UTC] {standard_task_runner.py:55} INFO - Started process 16760 to run task
[2025-09-09, 11:58:55 UTC] {standard_task_runner.py:82} INFO - Running: ['***', 'tasks', 'run', 'our_first_python_dag', 'first_pytho
[2025-09-09, 11:58:55 UTC] {standard_task_runner.py:83} INFO - Job 7: Subtask first_python_task
[2025-09-09, 11:58:55 UTC] {task_command.py:376} INFO - Running <TaskInstance: our_first_python_dag.first_python_task manual__2025-0
[2025-09-09, 11:58:55 UTC] {taskinstance.py:1592} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=username
AIRFLOW_CTX_DAG_ID=our_first_python_dag
AIRFLOW_CTX_TASK_ID=first_python_task
AIRFLOW_CTX_EXECUTION_DATE=2025-09-09T11:58:53.064534+00:00
AIRFLOW_CTX_TRY_NUMBER=1
AIRFLOW_CTX_DAG_RUN_ID=manual__2025-09-09T11:58:53.064534+00:00
[2025-09-09, 11:58:55 UTC] {logging_mixin.py:120} INFO - Hello, this is the first task using PythonOperator!
[2025-09-09, 11:58:55 UTC] {python.py:177} INFO - Done. Returned value was: None
```

Cron Expression



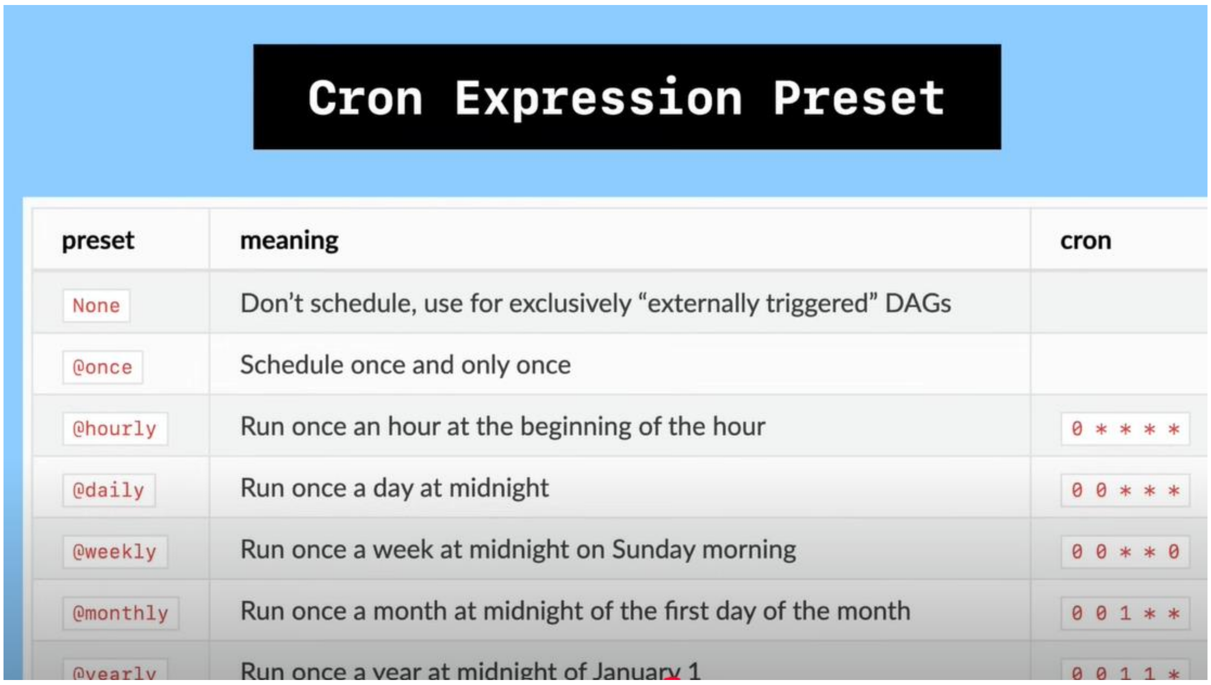
Cron Expression

```
15 14 1 * *
```

minute	hour	day (month)	month	day (week)
15	14	1	*	*

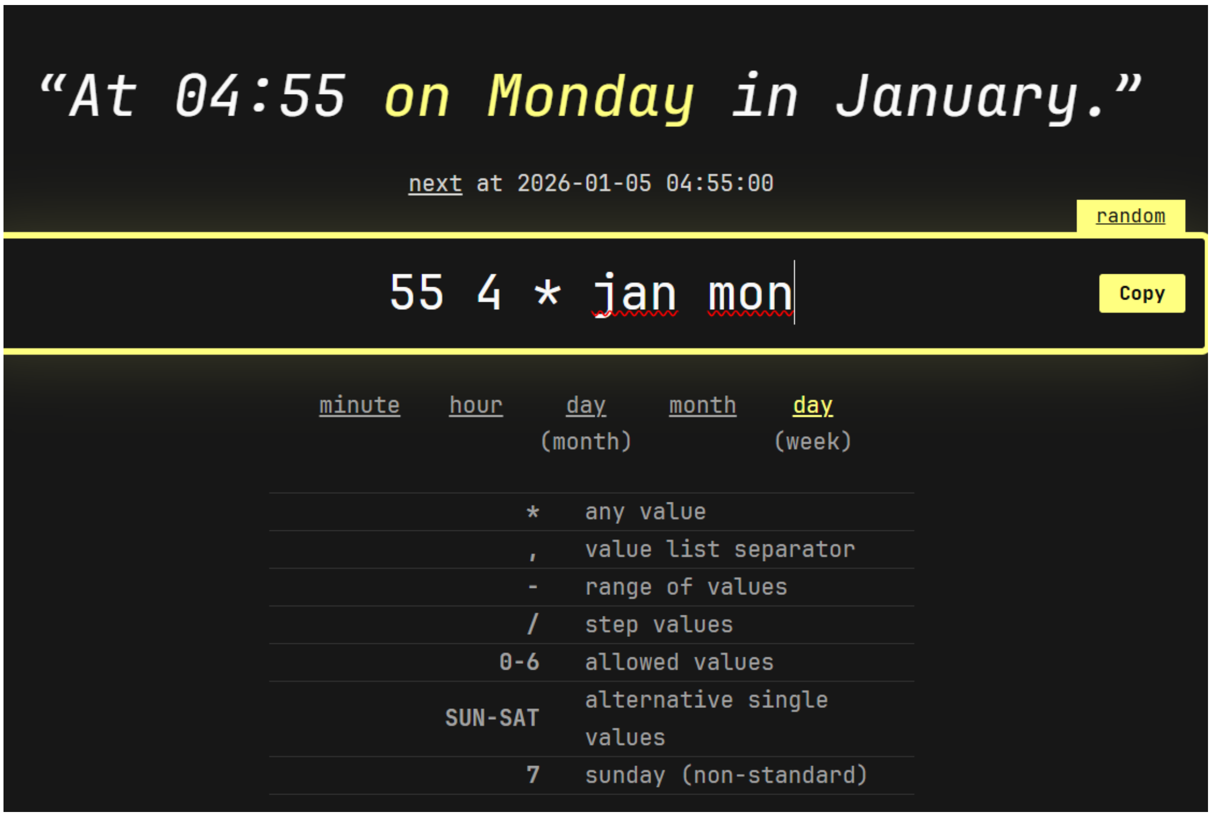
A CRON expression is a string comprising five fields separated by white space that represents a set of times

normally as a schedule to execute some routine.



Cron Expression Preset

preset	meaning	cron
`None`	Don't schedule, use for exclusively "externally triggered" DAGs	
`@once`	Schedule once and only once	
`@hourly`	Run once an hour at the beginning of the hour	`0 * * * *`
`@daily`	Run once a day at midnight	`0 0 * * *`
`@weekly`	Run once a week at midnight on Sunday morning	`0 0 * * 0`
`@monthly`	Run once a month at midnight of the first day of the month	`0 0 1 * *`
`@yearly`	Run once a year at midnight of January 1	`0 0 1 1 *`



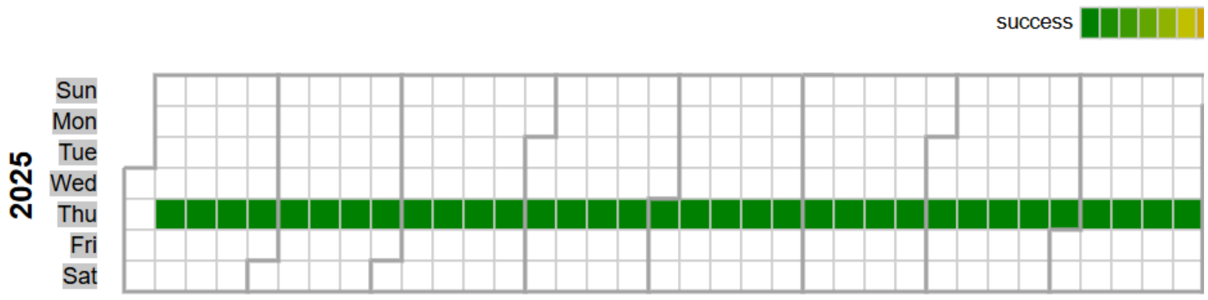
“At 04:55 on Monday in January.”

next at 2026-01-05 04:55:00

```
55 4 * jan mon
```

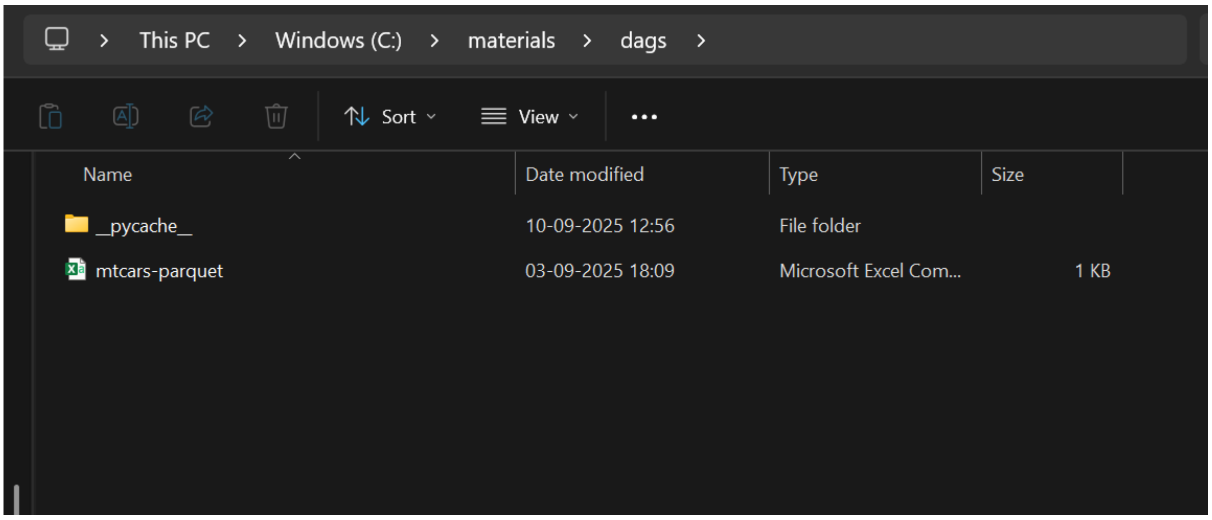
*	any value
,	value list separator
-	range of values
/	step values
0-6	allowed values
SUN-SAT	alternative single values
7	sunday (non-standard)

schedule_interval='0 0 * * Thu' → scheduled only on thursday



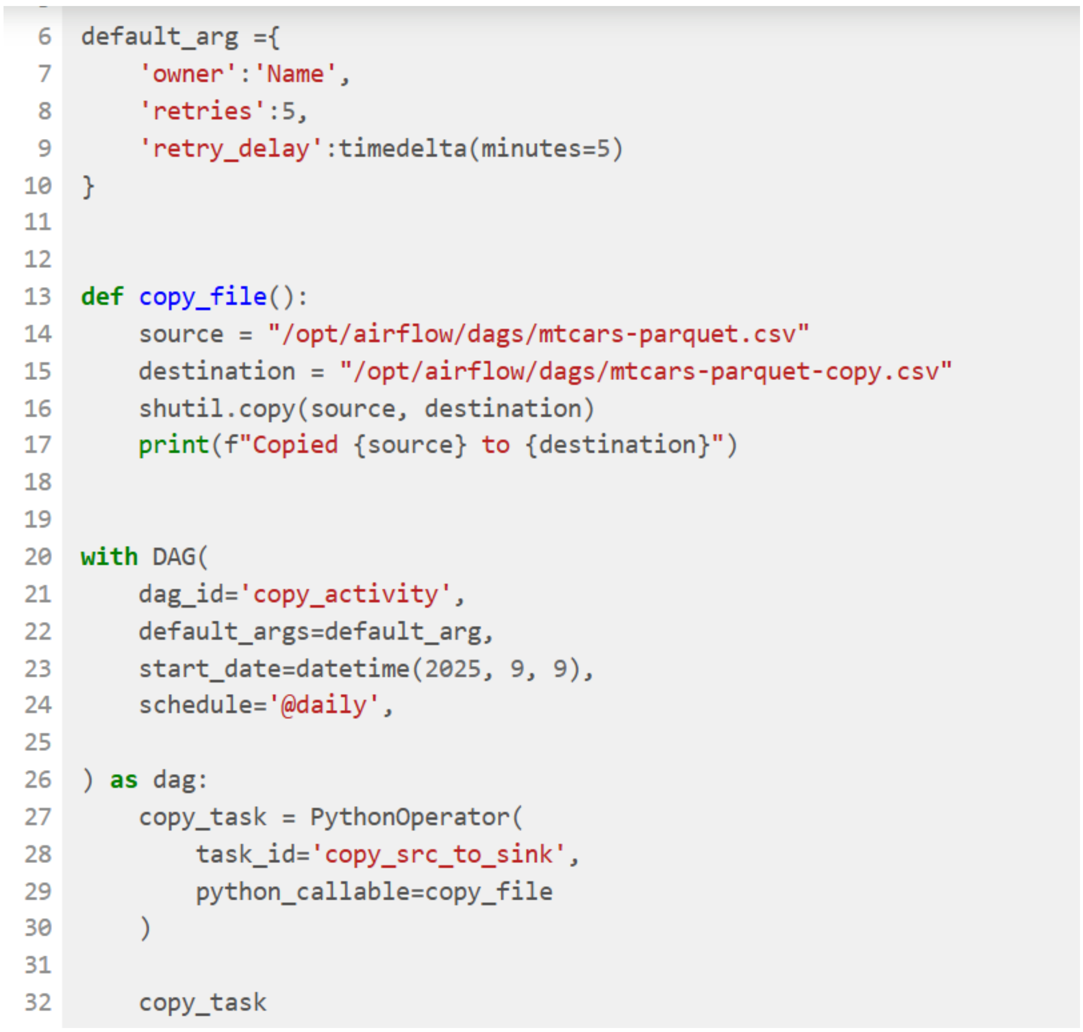
```python
from airflow import DAG
from airflow.operators.python import PythonOperator # type: ignore
from datetime import datetime, timedelta

default_args ={
    'owner':'yourname',
    'retries':5,
    'retry_delay':timedelta(minutes=5)
}

with DAG(
    dag_id="dag_with_crons_v02",
    default_args=default_args,
    description="DAG with crons",
    start_date=datetime(2025, 1, 8),
    schedule_interval='@daily'
) as dag:pass
```

Copy file from src to sink

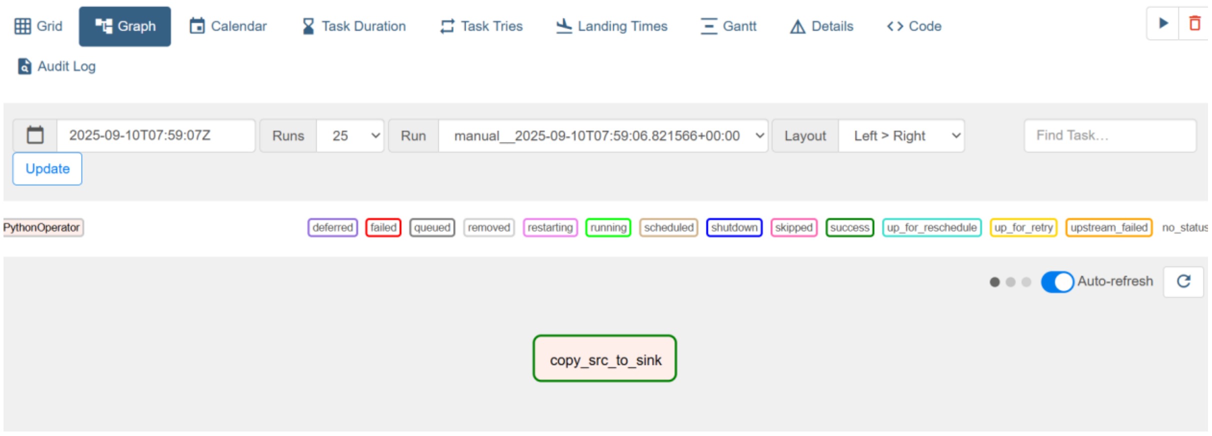


This PC > Windows (C:) > materials > dags >

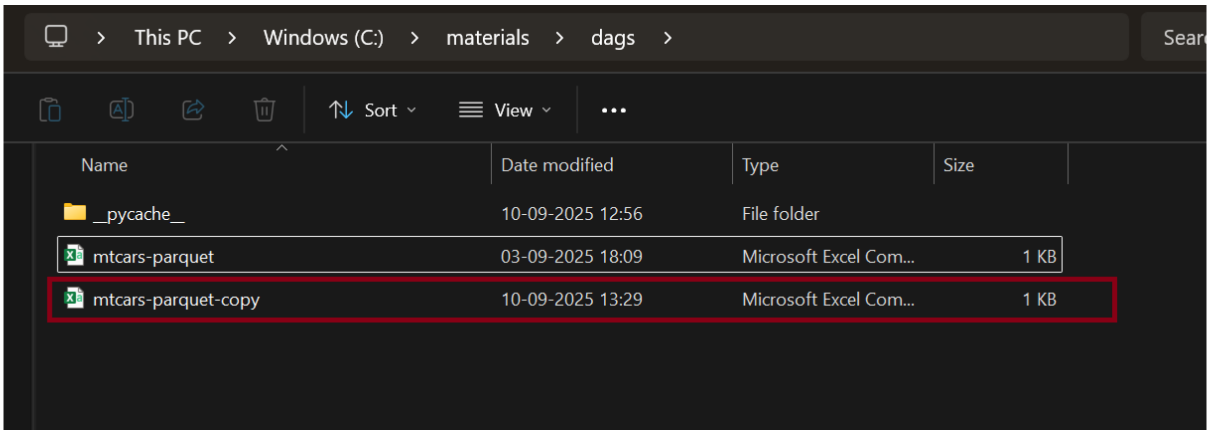
Name	Date modified	Type	Size
__pycache__	10-09-2025 12:56	File folder	
mtcars-parquet	03-09-2025 18:09	Microsoft Excel Com...	1 KB



```python
default_arg ={
    'owner':'Name',
    'retries':5,
    'retry_delay':timedelta(minutes=5)
}


def copy_file():
    source = "/opt/airflow/dags/mtcars-parquet.csv"
    destination = "/opt/airflow/dags/mtcars-parquet-copy.csv"
    shutil.copy(source, destination)
    print(f"Copied {source} to {destination}")


with DAG(
    dag_id='copy_activity',
    default_args=default_arg,
    start_date=datetime(2025, 9, 9),
    schedule='@daily',

) as dag:
    copy_task = PythonOperator(
        task_id='copy_src_to_sink',
        python_callable=copy_file
    )

    copy_task
```



2025-09-10T07:59:07Z Runs 25 Run manual__2025-09-10T07:59:06.821566+00:00 Layout Left > Right



Name	Date modified	Type	Size
pycache	10-09-2025 12:56	File folder	
mtcars-parquet	03-09-2025 18:09	Microsoft Excel Com...	1 KB
mtcars-parquet-copy	10-09-2025 13:29	Microsoft Excel Com...	1 KB

Case 1: **Default (catchup=True)**

start_date = datetime**(2024, 2, 2)**
schedule_interval = "@daily"
catchup = True # default

- Today is 2025-09-10.
- Airflow will try to create DAG runs for every day between **2024-02-02** → **2025-09-10**.
- That means hundreds of runs will be triggered (backlog).

Case 2: catchup=False

start_date = datetime(2024, 2, 2)
schedule_interval = "@daily"
catchup = False

- Today is **2025-09-10**.
- Airflow **does not create runs for the old dates**.
- Only a DAG run for **today (2025-09-10)** will be created (and from now onwards, it runs daily).

❖ With catchup=True → all past runs are scheduled.
❖ With catchup=False → only today's run (and future runs) is executed.

<u>Airflow AWS S3 sensor operation</u>

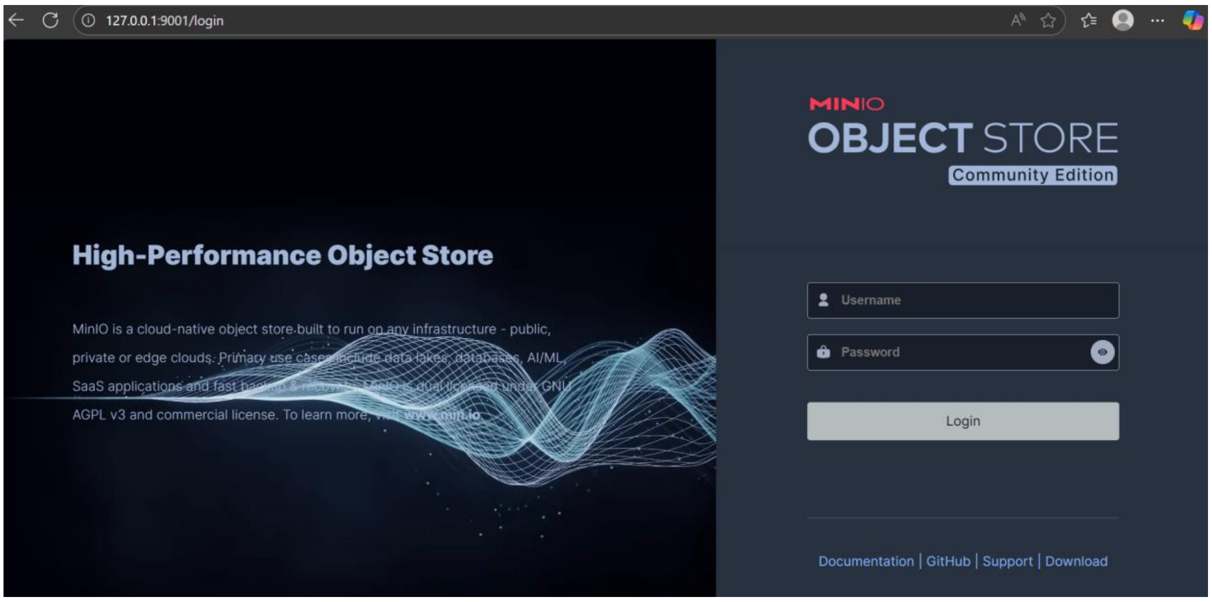
To create a community edition of S3, we use **minio**

Cmd

```
docker run -p 9000:9000 -p 9001:9001 --name minio -v
${HOME}/minio/data:/data -e "MINIO_ROOT_USER=ROOTNAME" -e
"MINIO_ROOT_PASSWORD=CHANGEME123" quay.io/minio/minio
server /data --console-address ":9001"
```

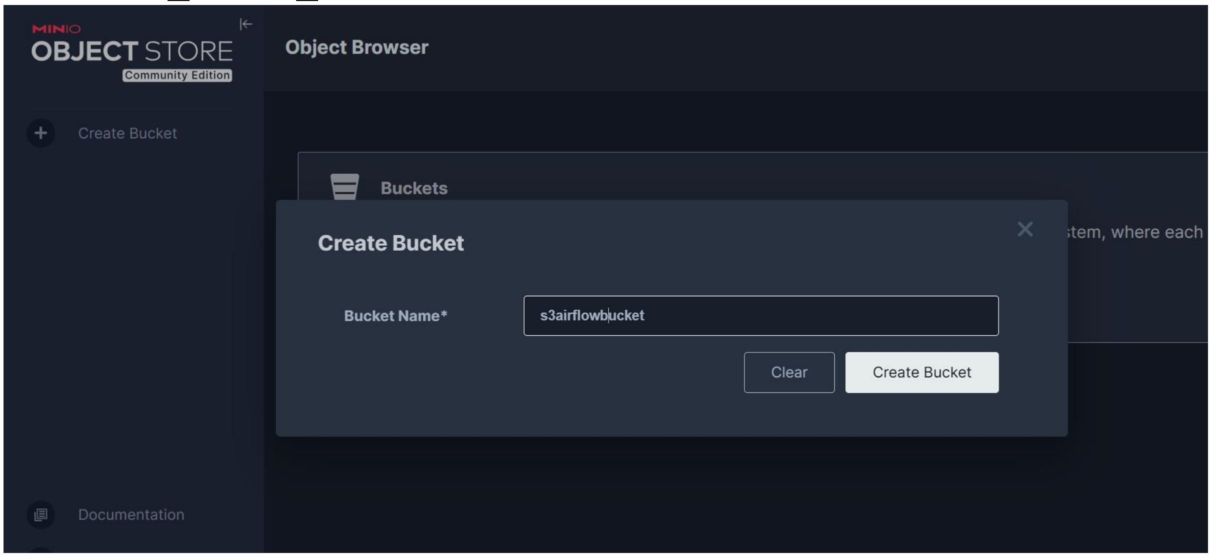
Gives

API: http://172.17.0.2:9000 http://127.0.0.1:9000
WebUI: http://172.17.0.2:9001 http://127.0.0.1:9001



MinIO is a cloud-native object store built to run on any infrastructure - public, private or edge clouds. Primary use cases include data lakes, databases, AI/ML, SaaS applications and fast backup & recovery. MinIO is dual-licensed under GNU AGPL v3 and commercial license. To learn more, visit www.min.io

-e "MINIO_ROOT_USER=ROOTNAME"
-e "MINIO_ROOT_PASSWORD=CHANGEME123"



Bucket Name* s3airflowbucket

Airflow sensor:

A Sensor in Airflow is a special operator that keeps checking for an external condition—like a file arrival, an API response, or another task's completion—before downstream tasks start. It prevents the workflow from running until condition satisfied.

Sensors are special operators that pause a DAG until a condition is met. Common types include File sensors (like S3KeySensor, FileSensor), ExternalTaskSensor for waiting on other DAGs, Time sensors like TimeDeltaSensor, Database sensors such as SqlSensor, and API sensors like HttpSensor.
We can also create **custom sensors** by subclassing **BaseSensorOperator**.

FileSensor – waits for a file on a local or shared filesystem.
S3keySensor – waits for a file to appear in an S3 bucket.

ExternalTaskSensor – waits for a specific task in another DAG to finish.
Dag Sensor- waits for an entire external DAG to succeed.

TimeDeltaSensor – waits for a fixed time interval.
DateTimeSensor – waits until a specific datetime is reached.

SqlSensor – runs a sql query repeatedly until it returns rows/condition is true.
HttpSensor – waits util an HTTP endpoint return a desired status or content.

Operator
operator in airflow is the definition or type of the work a task will perform inside a DAG.
DAG = workflow structure.
Task = an instance of work inside that DAG.
Operator = specifies *what kind* of work that task does.

Operator	**Purpose**
PythonOperator	Run a Python function.
BashOperator	Execute shell/bash commands.
EmailOperator	Send an email notification.
HttpOperator	Call an HTTP endpoint or API.
Sql/Postgres/BigQuery Operators	Run SQL queries on databases.
Sensor (a special operator)	Wait for a condition or file to arrive.
BranchPythonOperator	Conditional branching in a DAG.
DummyOperator	No-op placeholder (for dependencies or grouping).

SubDagOperator

- **Purpose**:
  Allows you to embed a **DAG inside another DAG** as a "sub-workflow."
- **How it works**:
  You define a separate DAG (the *sub-DAG*) and call it from the parent DAG using SubDagOperator.
- **Use case**:
  When you need to group a set of tasks that should run together and be treated as a single task in the main DAG.

TriggerDagRunOperator

- **Purpose**:
  Triggers **another DAG** (independent) from within a DAG run.
- **How it works**:
  Sends a request to start a completely separate DAG, optionally passing parameters.

- SubDagOperator runs a child DAG as part of the parent DAG—useful for grouping tasks but now largely replaced by TaskGroups.
- TriggerDagRunOperator starts an entirely separate DAG from within a task, often used for chaining independent workflows or passing runtime parameters.