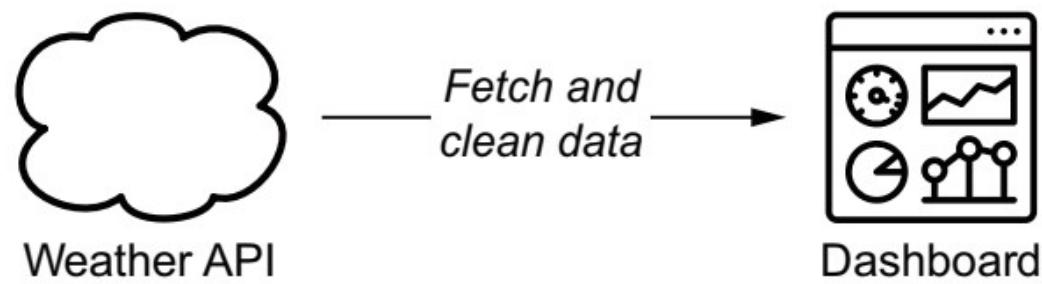
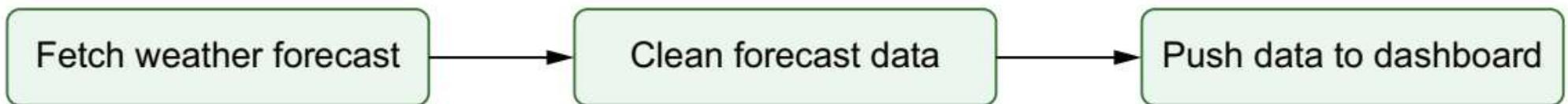


Airflow

Simple Complexity



More Complexity



DAG

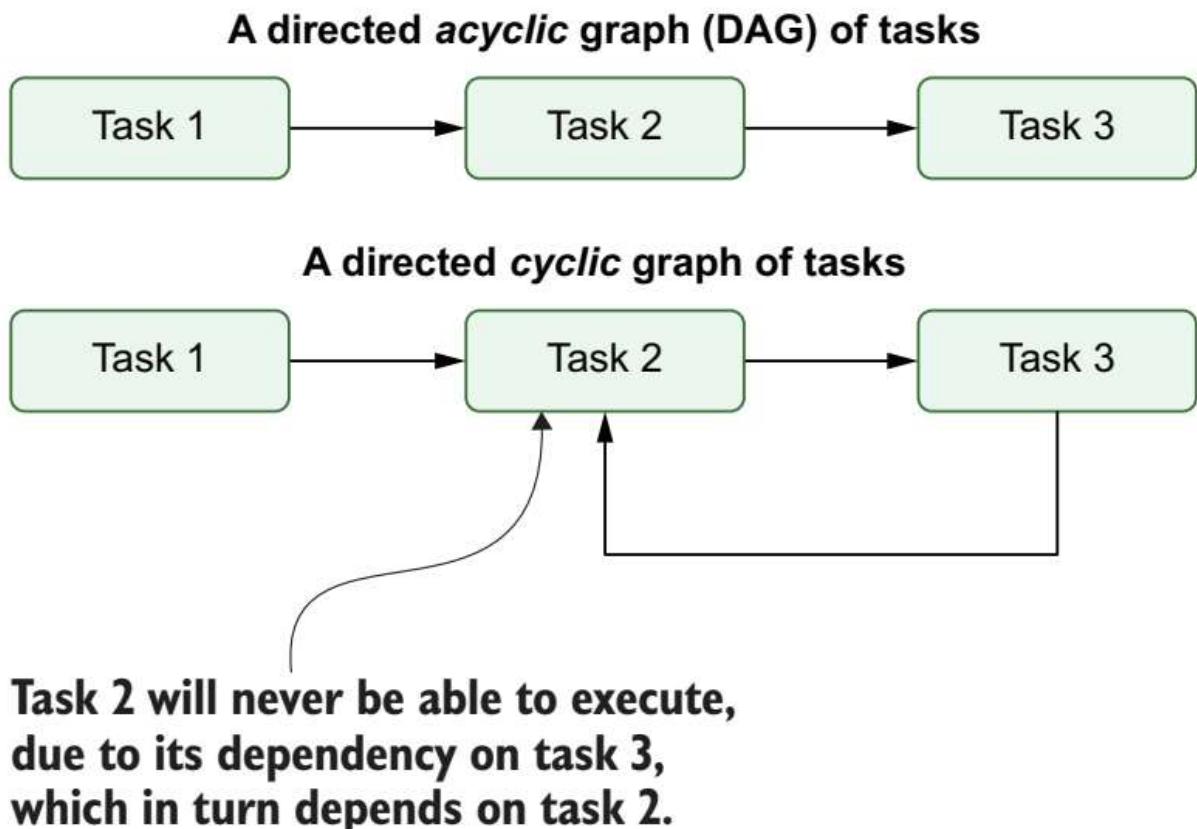
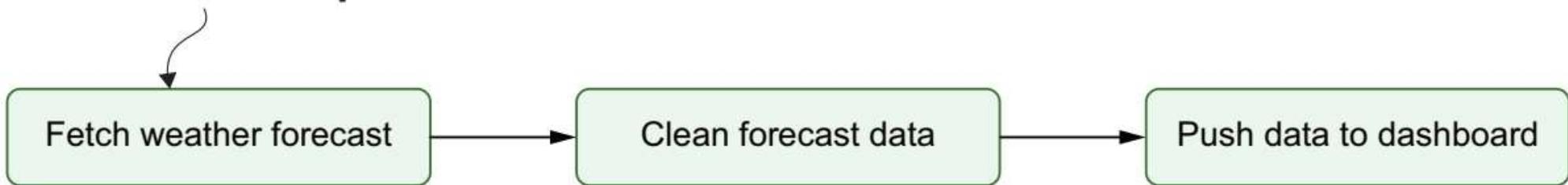


Figure 1.3 Cycles in graphs prevent task execution due to circular dependencies. In acyclic graphs (top), there is a clear path to execute the three different tasks. However, in cyclic graphs (bottom), there is no longer a clear execution path due to the interdependency between tasks 2 and 3.

How tasks are executed via DAG

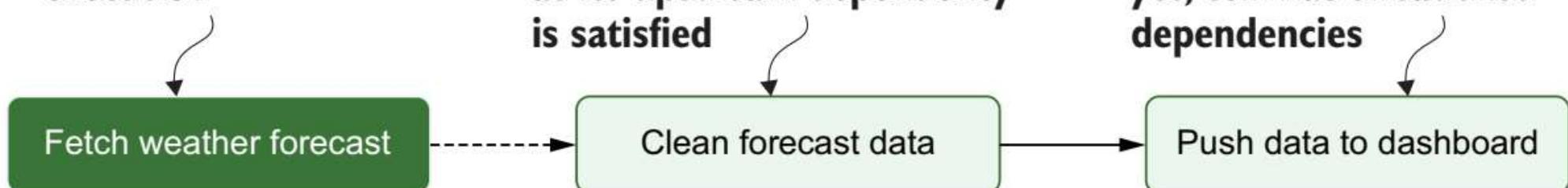
Loop 1 Task ready for execution; no unsatisfied dependencies



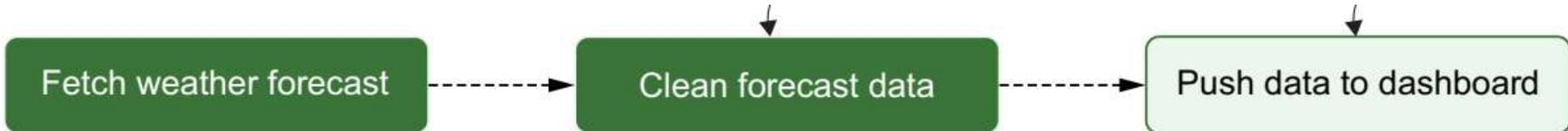
Loop 2 Task has finished execution

Task now ready for execution, as its upstream dependency is satisfied

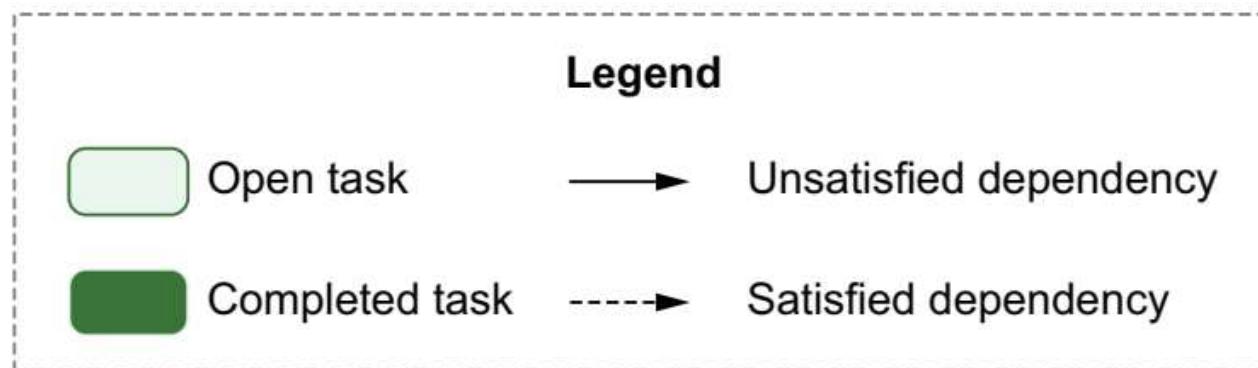
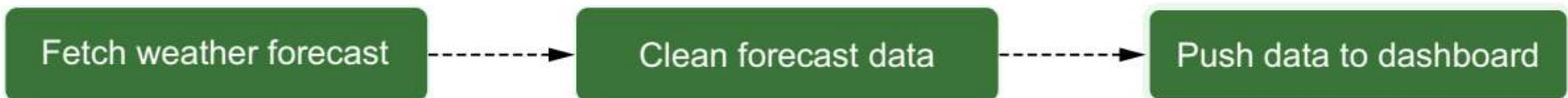
Not ready for execution yet; still has unsatisfied dependencies



How tasks are executed via DAG

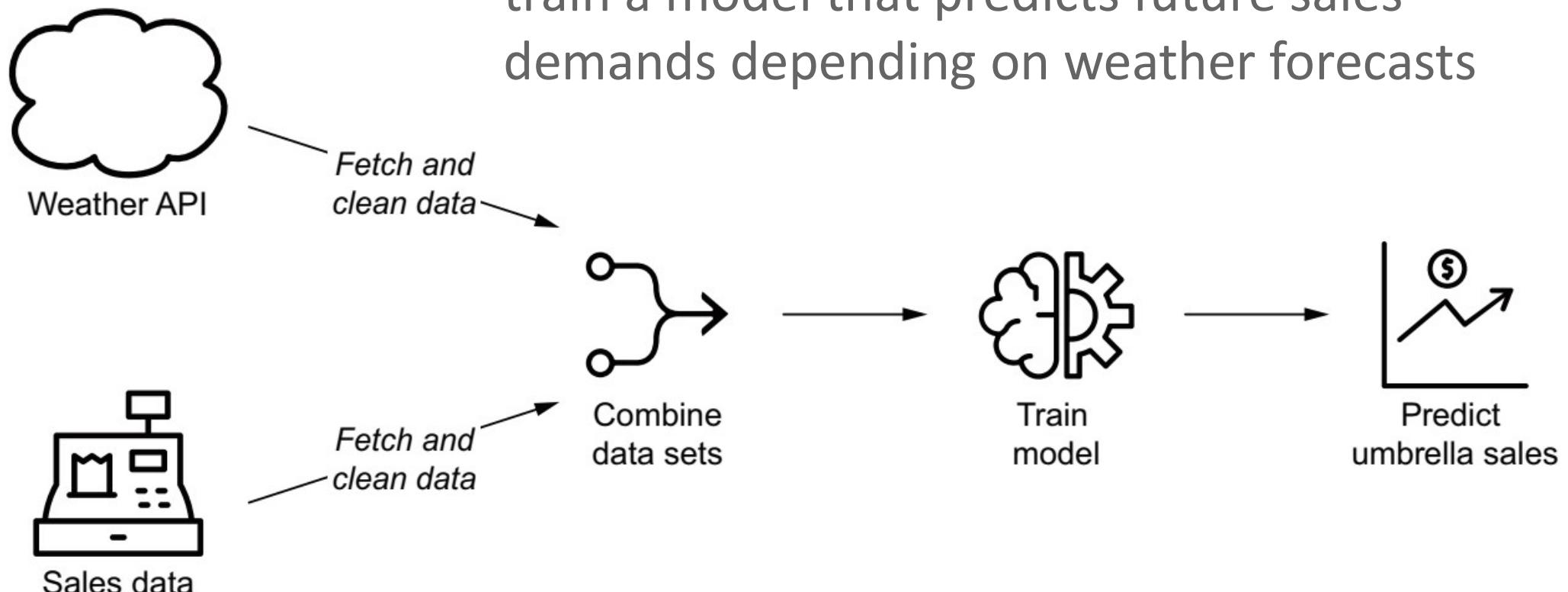


End state



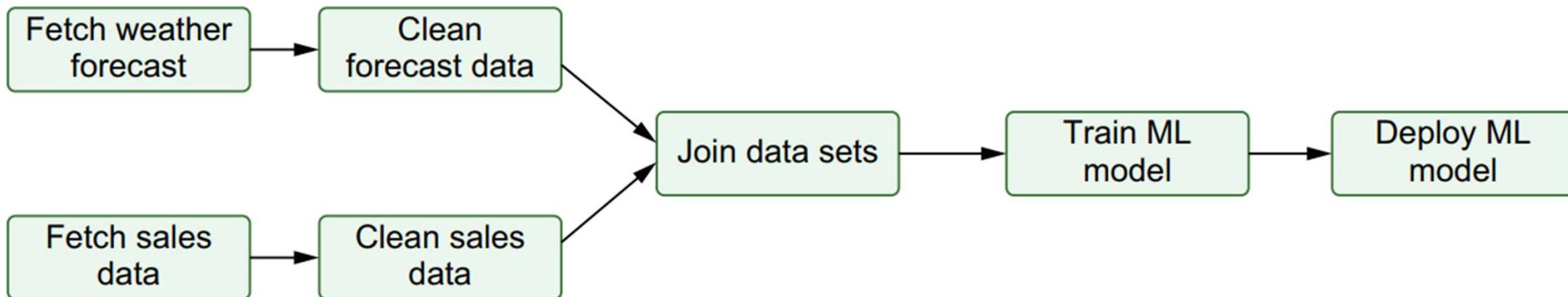
Overview of the umbrella demand use case

Historical weather and sales data are used to train a model that predicts future sales demands depending on weather forecasts



Independence between sales and weather tasks in the graph representation of the data pipeline for the umbrella demand forecast model

- The two sets of fetch/cleaning tasks are independent as they involve two different data sets (the weather and sales data sets).
- This independence is indicated by the lack of edges between the two sets of tasks.

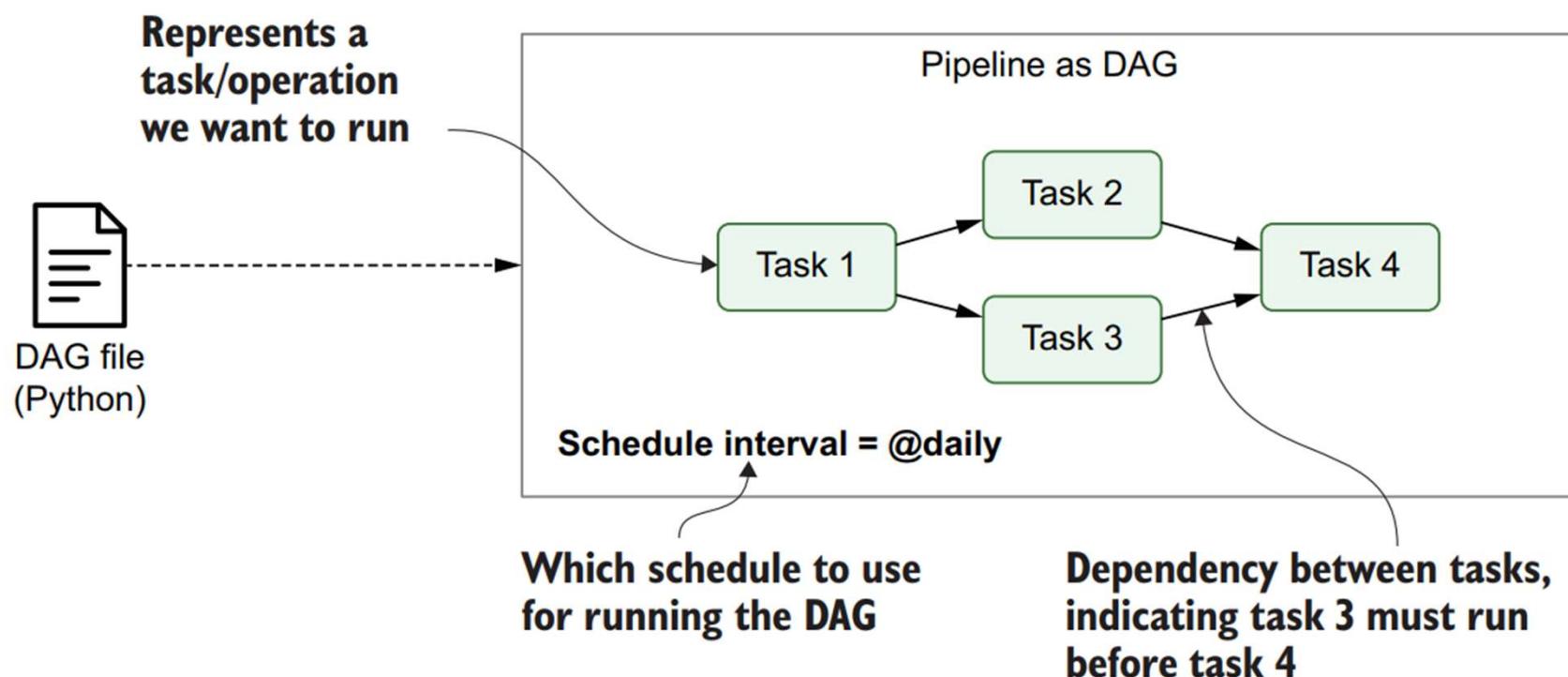


Overview of several well-known workflow managers

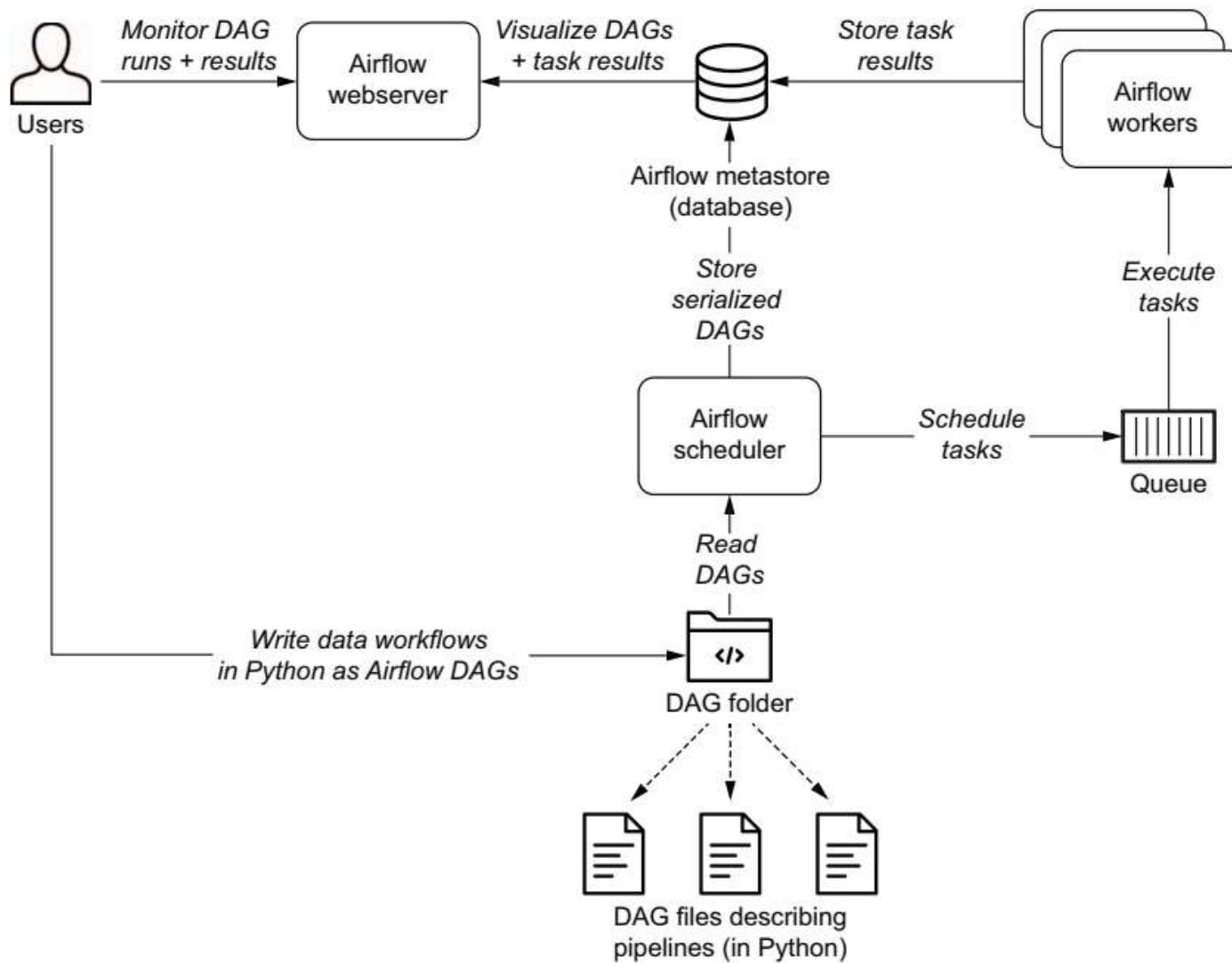
Name	Originated at ^a	Workflows defined in	Written in	Scheduling	Backfilling	User interface ^b	Installation platform	Horizontally scalable
Airflow	Airbnb	Python	Python	Yes	Yes	Yes	Anywhere	Yes
Argo	Applatix	YAML	Go	Third party ^c		Yes	Kubernetes	Yes
Azkaban	LinkedIn	YAML	Java	Yes	No	Yes	Anywhere	
Conductor	Netflix	JSON	Java	No		Yes	Anywhere	Yes
Luigi	Spotify	Python	Python	No	Yes	Yes	Anywhere	Yes
Make		Custom DSL	C	No	No	No	Anywhere	No
Metaflow	Netflix	Python	Python	No		No	Anywhere	Yes
Nifi	NSA	UI	Java	Yes	No	Yes	Anywhere	Yes
Oozie		XML	Java	Yes	Yes	Yes	Hadoop	Yes

Airflow pipelines are defined as DAGs using Python code in DAG files

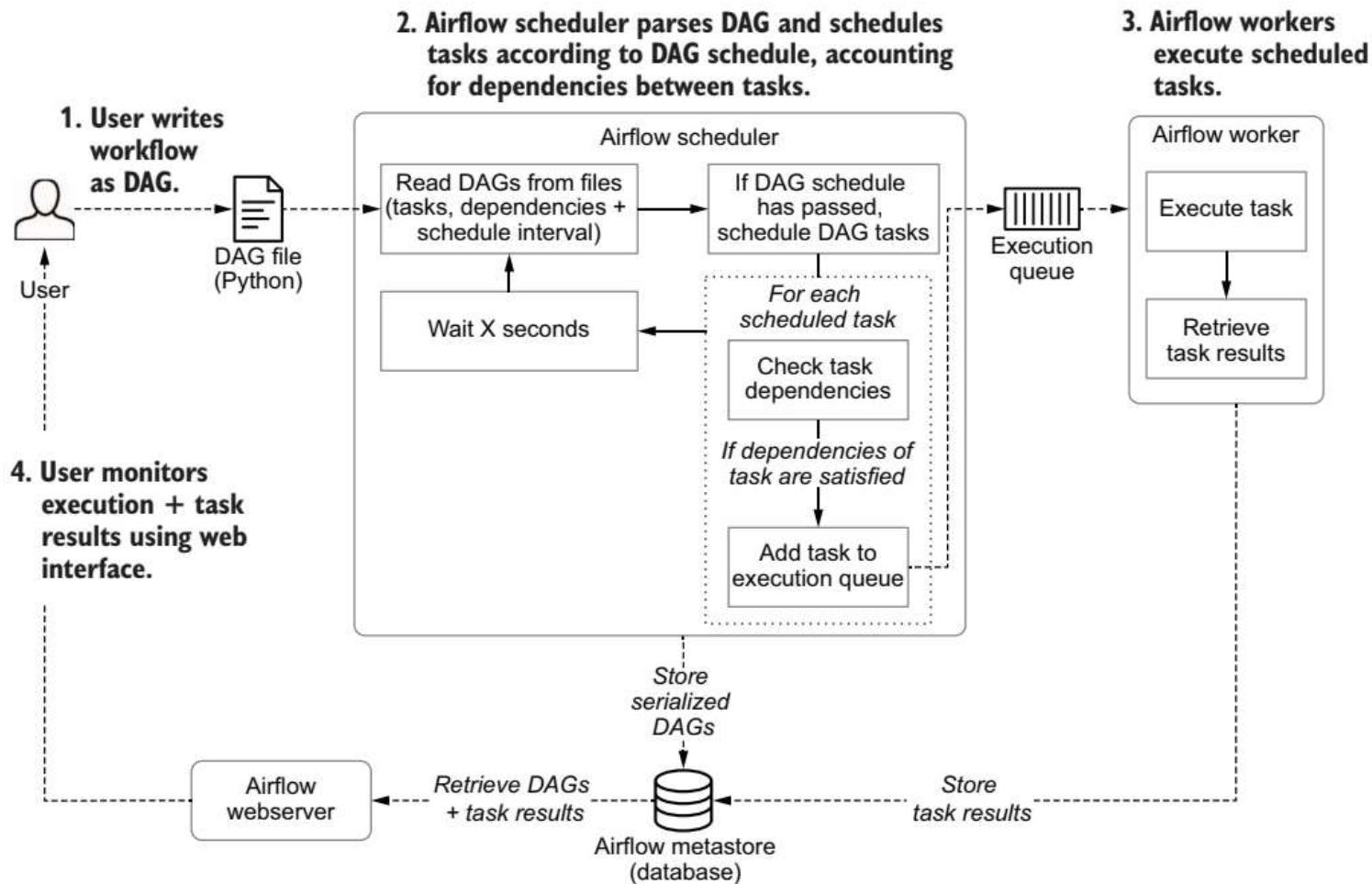
- Each DAG file typically defines one DAG, which describes the different tasks and their dependencies.
- Besides this, the DAG also defines a schedule interval that determines when the DAG is executed by Airflow



Overview of the main components involved in Airflow



Schematic overview of the process involved in developing and executing pipelines as DAGs



The login page for the Airflow web interface

A screenshot of a web browser window showing the Airflow login page. The title bar says "Airflow". The address bar shows "localhost:8080/login/?next=http%3A%2F%2Flocalhost%3A8080%2Fhome". The page itself has a header with the Airflow logo and "DAGs" link, a timestamp "20:47 UTC", and a "Log In" button. Below this is a "Sign In" form with fields for "Username" (containing "admin") and "Password" (containing "****"). A callout arrow points from the text "Your username + password" to the combined "Username" and "Password" input field.

Sign In

Enter your login and password below:

Username:
admin

Password:

Sign In

Your username + password

The main page of Airflow's web interface

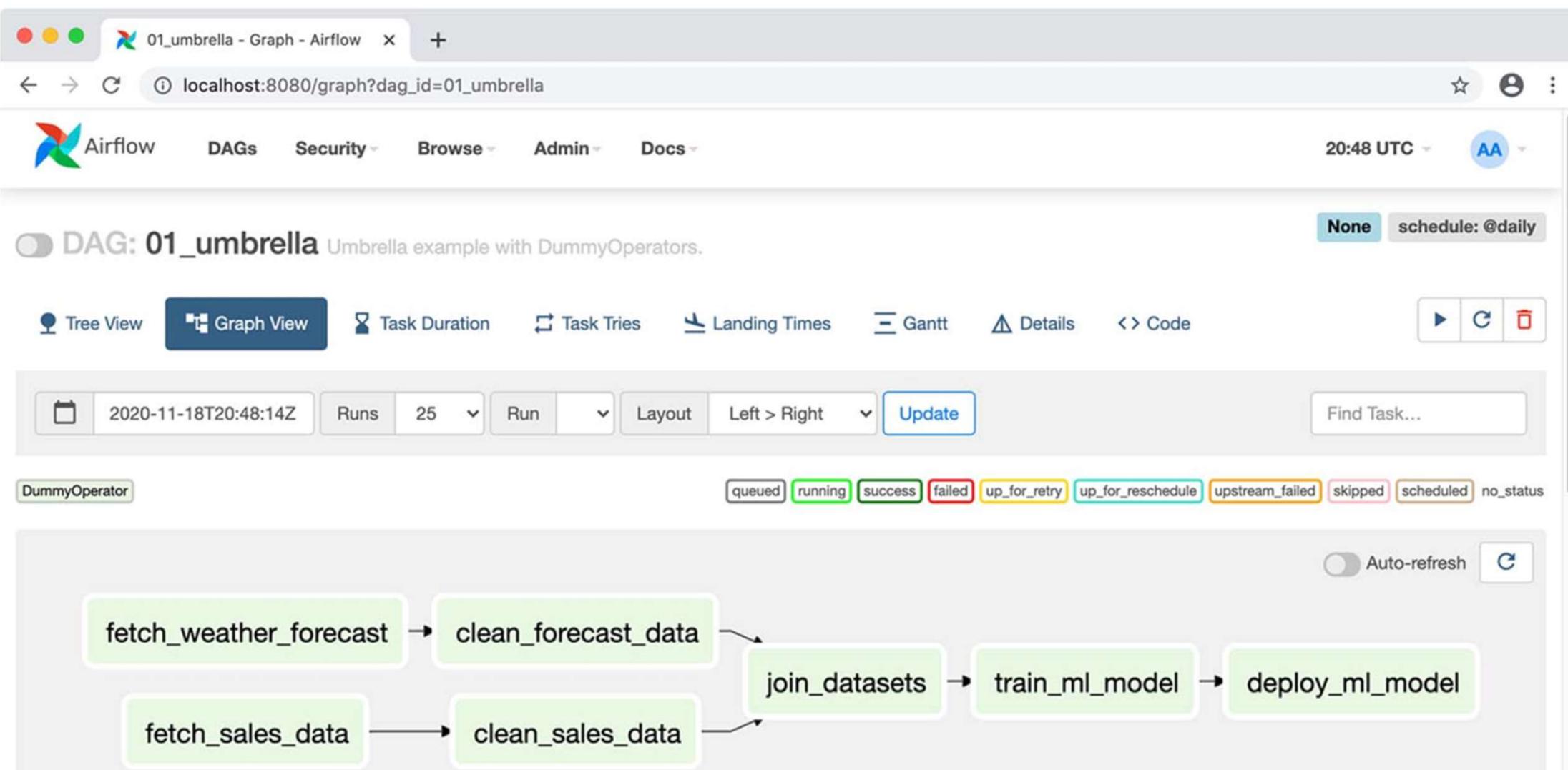
The screenshot shows the Airflow web interface at `localhost:8080/home`. The top navigation bar includes links for DAGs, Security, Browse, Admin, and Docs, along with a timestamp of 20:48 UTC and a user icon. The main section is titled "DAGs" and displays a single DAG named "01_umbrella". The DAG details include its owner ("airflow"), a schedule of "@daily", and recent task runs. A navigation bar at the bottom allows switching between pages 1, 2, and 3.

Names of registered workflows

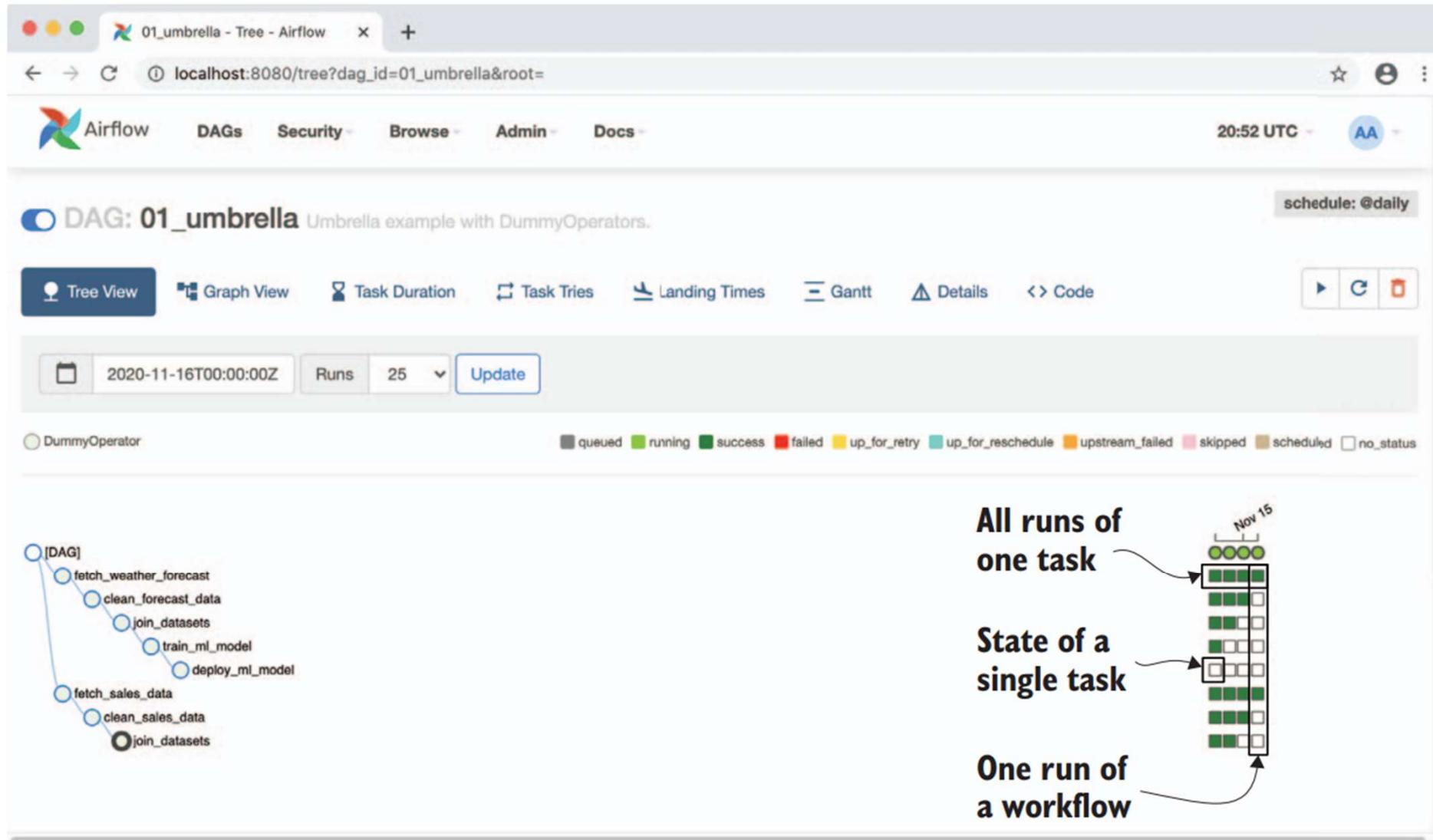
Workflow schedules

State of workflow tasks from recent runs

The graph view in Airflow's web interface

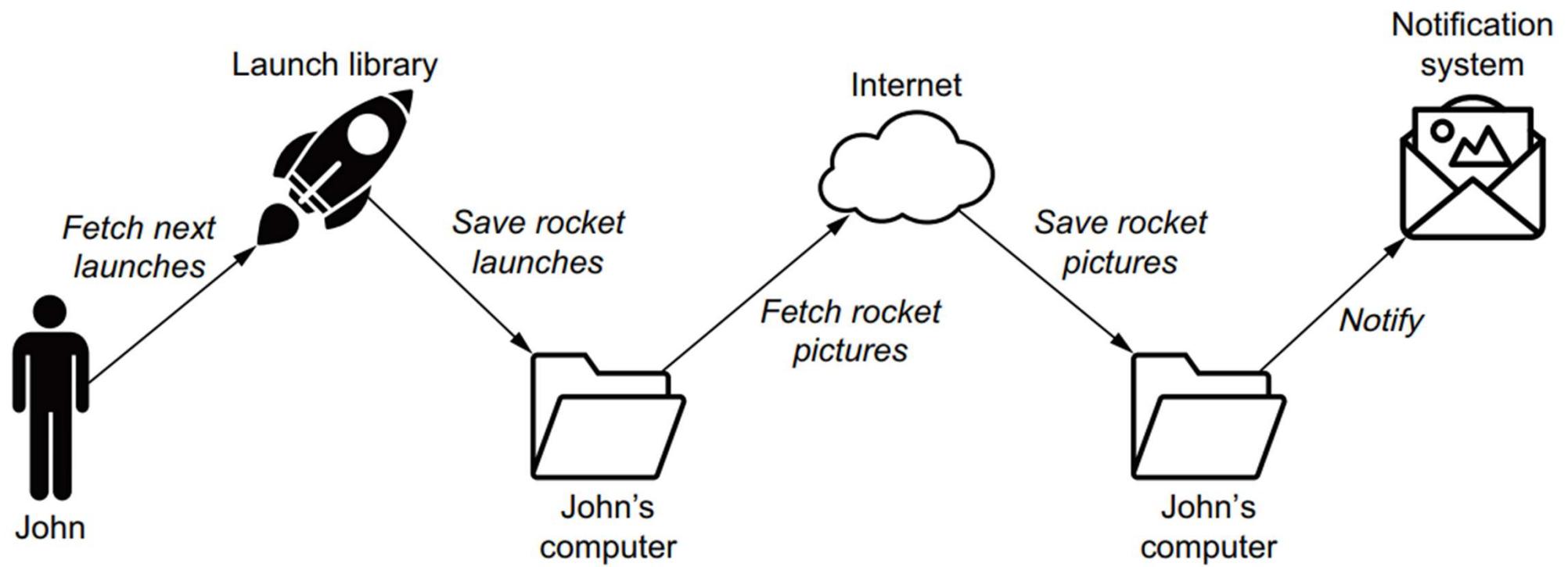


Airflow's tree view

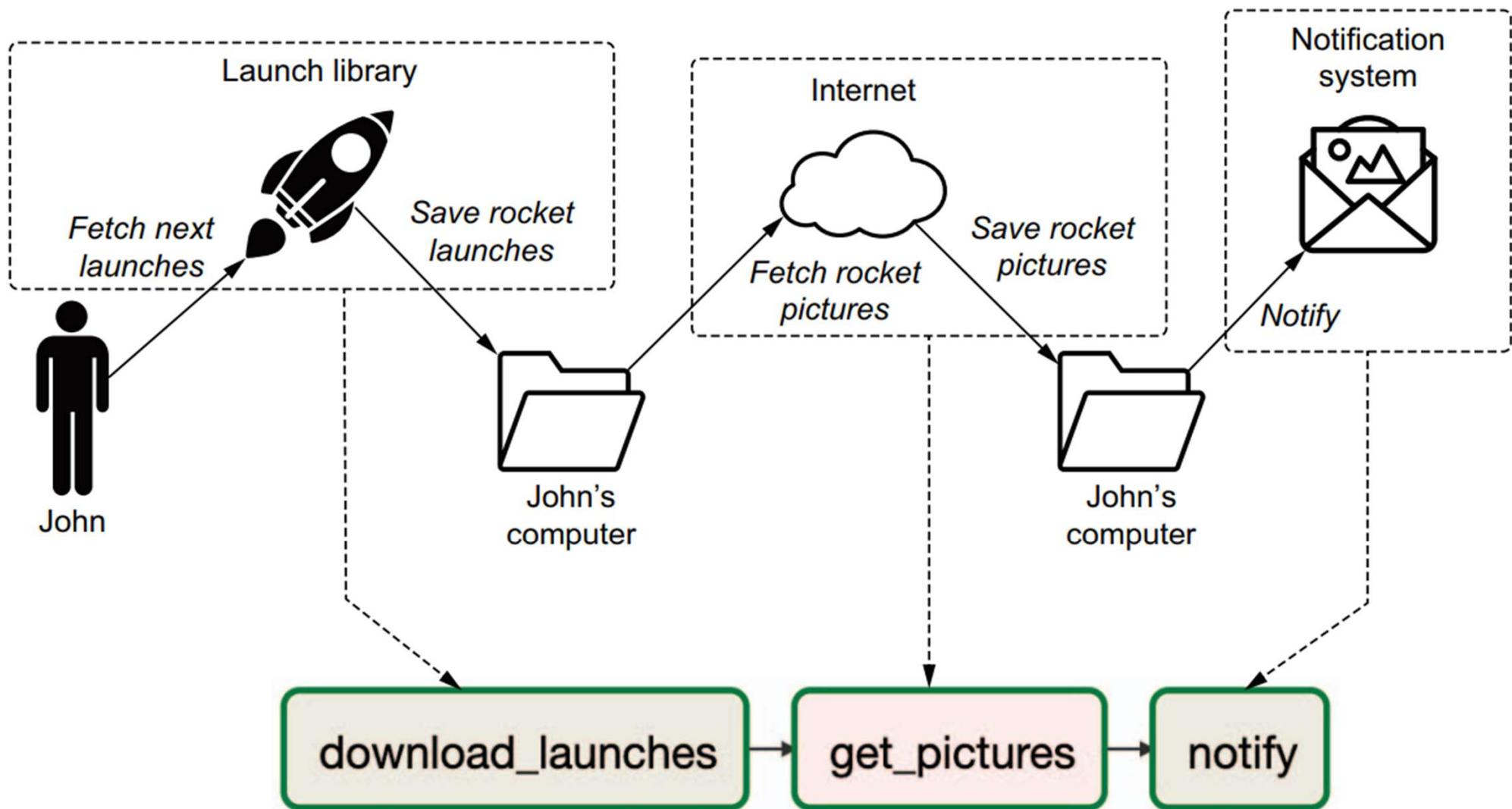


Anatomy of an Airflow DAG

Mental model of downloading rocket pictures



Mental model mapped to tasks in Airflow



DAG for downloading and processing rocket launch data

```
import json
import pathlib

import airflow
import requests
import requests.exceptions as requests_exceptions
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

dag = DAG(
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14),
    schedule_interval=None,
)

download_launches = BashOperator(
    task_id="download_launches",
    bash_command="curl -o /tmp/launches.json -L 'https://ll.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag,
)

def _get_pictures():
    # Ensure directory exists

```

The name of the DAG

At what interval the DAG should run

Instantiate a DAG object; this is the starting point of any workflow.

The date at which the DAG should first start running

Apply Bash to download the URL response with curl.

The name of the task

A Python function will parse the response and download all rocket pictures.

DAG for downloading and processing rocket launch data

```
)  
  
def _get_pictures():  
    # Ensure directory exists  
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)  
  
    # Download all pictures in launches.json  
    with open("/tmp/launches.json") as f:  
        launches = json.load(f)  
        image_urls = [launch["image"] for launch in launches["results"]]  
        for image_url in image_urls:  
            try:  
                response = requests.get(image_url)  
                image_filename = image_url.split("/")[-1]  
                target_file = f"/tmp/images/{image_filename}"  
                with open(target_file, "wb") as f:  
                    f.write(response.content)  
                print(f"Downloaded {image_url} to {target_file}")  
            except requests.exceptions.MissingSchema:  
                print(f"{image_url} appears to be an invalid URL.")  
            except requests.exceptions.ConnectionError:  
                print(f"Could not connect to {image_url}.")  
  
A Python function will parse  
the response and download  
all rocket pictures.
```

DAG for downloading and processing rocket launch data

```
get_pictures = PythonOperator(  
    task_id="get_pictures",  
    python_callable=_get_pictures,  
    dag=dag,  
)  
  
notify = BashOperator(  
    task_id="notify",  
    bash_command='echo "There are now $(ls /tmp/images/ | wc -l) images."',  
    dag=dag,  
)  
  
download_launches >> get_pictures >> notify
```

Call the Python function in the DAG with a PythonOperator.

Set the order of execution of tasks.

Instantiating a DAG object

```
dag = DAG(  
    dag_id="download_rocket_launches",  
    start_date=airflow.utils.dates.days_ago(14),  
    schedule_interval=None,  
)
```

The DAG class takes two required arguments.

The name of the DAG displayed in the Airflow user interface (UI)

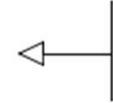
The datetime at which the workflow should first start running

Instantiating a BashOperator to run a Bash command

```
download_launches = BashOperator(  
    task_id="download_launches",           ← The name of  
                                            the task  
    bash_command="curl -o /tmp/launches.json 'https://  
        11.thespacedevs.com/2.0.0/launch/upcoming'",  
    dag=dag,                                ← The Bash command  
                                            to execute  
)  
                                         ← Reference to the  
DAG variable
```

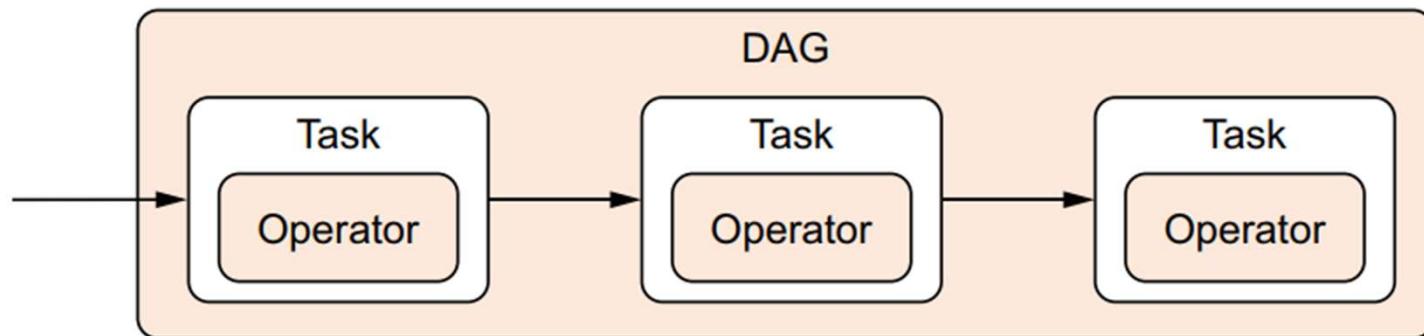
Defining the order of task execution

```
download_launches >> get_pictures >> notify
```



Arrows set the order
of execution of tasks.

DAGs and operators are used by Airflow users



Running a Python function using the PythonOperator

```
def _get_pictures():           ← Python function to call
    # Ensure directory exists
    pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)   ← Create pictures directory if it doesn't exist.

    # Download all pictures in launches.json
    with open("/tmp/launches.json") as f:   ← Open the result from the previous task.
        launches = json.load(f)
        image_urls = [launch["image"] for launch in launches["results"]]
        for image_url in image_urls:
            try:
                response = requests.get(image_url)   ← Download each image.
                image_filename = image_url.split("/") [-1]
                target_file = f"/tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)

                print(f"Downloaded {image_url} to {target_file}")   ← Print to stdout; this will be captured in Airflow logs.
            except requests.exceptions.MissingSchema:
                print(f"{image_url} appears to be an invalid URL.")
            except requests.exceptions.ConnectionError:
                print(f"Could not connect to {image_url}.")
```

Store each image.

Print to stdout; this will be captured in Airflow logs.

```
get_pictures = PythonOperator(   ← Instantiate a PythonOperator to call the Python function.
    task_id="get_pictures",
    python_callable=_get_pictures,   ← Point to the Python function to execute.
    dag=dag,
)
```

The `python_callable` argument in the `PythonOperator` points to a function to execute

```
→ def _get_pictures():    ] PythonOperator callable
    # do work here ...
    ]
get_pictures = PythonOperator(
    task_id="get_pictures",
    python_callable=_get_pictures,    ] PythonOperator
    dag=dag
)
    ]
```

Ensures that the output directory exists and creates it if it doesn't

```
# Ensure directory exists
pathlib.Path("/tmp/images").mkdir(parents=True, exist_ok=True)
```

Extracts image URLs for every rocket launch

```
Open the rocket launches' JSON.  
with open("/tmp/launches.json") as f:    ←  
    launches = json.load(f)                 ←  
    image_urls = [launch["image"] for launch in launches["results"]]  
    ← Read as a dict so we can mingle the data.  
    ← For every launch, fetch the element "image".
```

Downloads all images from the retrieved image URLs

```
for image_url in image_urls:
    try:
        response = requests.get(image_url)
        image_filename = image_url.split("/") [-1]
        target_file = f"/tmp/images/{image_filename}"
        with open(target_file, "wb") as f:
            f.write(response.content)
            print(f"Downloaded {image_url} to {target_file}")
    except requests.exceptions.MissingSchema:
        print(f"{image_url} appears to be an invalid URL.")
    except requests.exceptions.ConnectionError:
        print(f"Could not connect to {image_url}.")
```

Loop over all image URLs.

Get the image.

Get only the filename by selecting everything after the last. For example, `https://host/RocketImages/Electron.jpg_1440.jpg` → `Electron.jpg_1440.jpg`.

Construct the target file path.

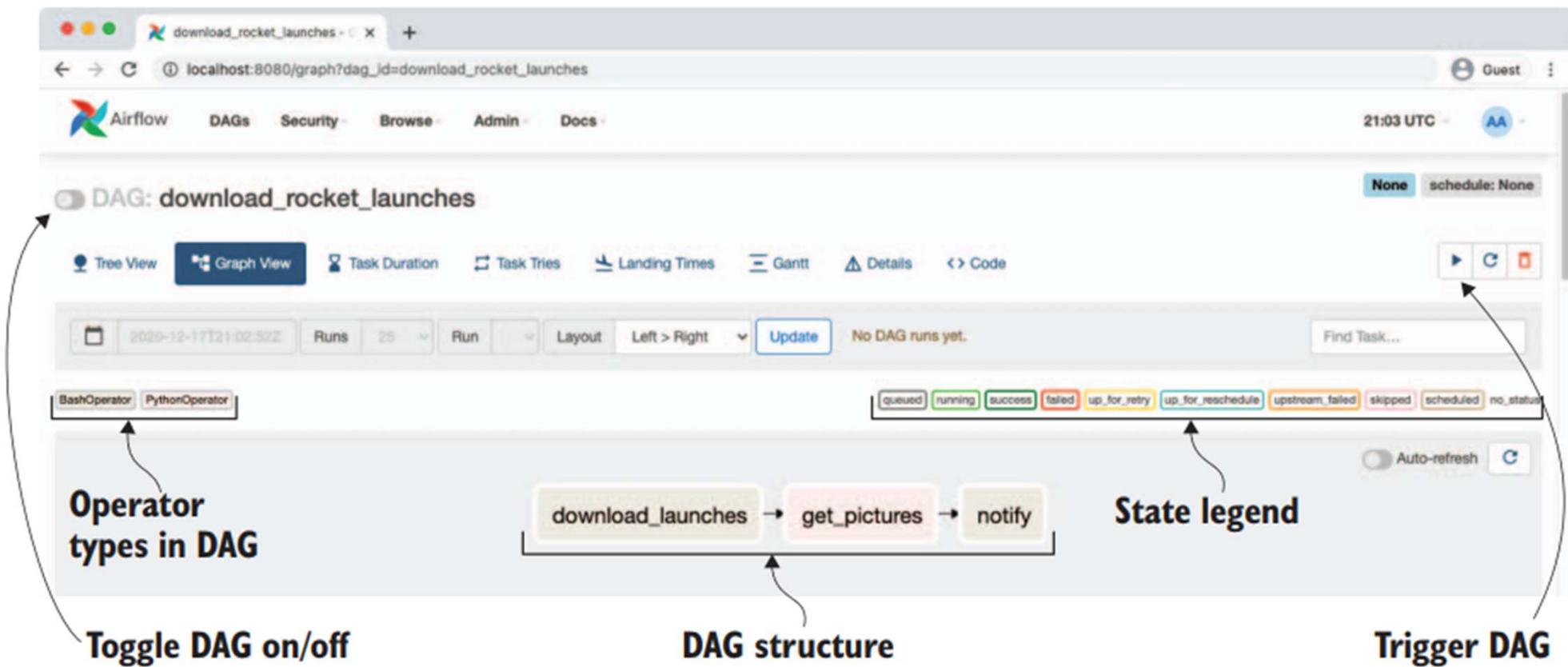
Open target file handle.

Write image to file path.

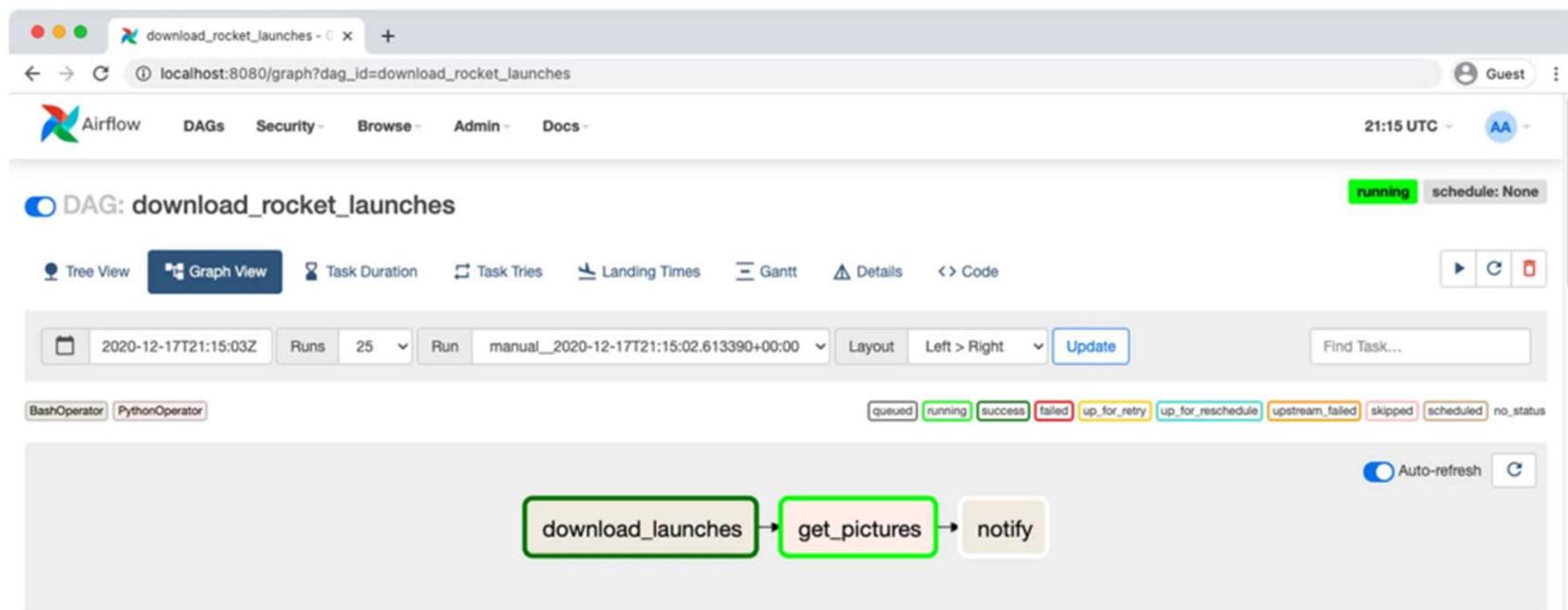
Print result.

Catch and process potential errors.

Airflow graph view



Graph view displaying a running DAG



Task pop-up options

Task Instance: **notify** ×
at: 2020-12-17T21:15:02.613390+00:00

[Instance Details](#) [Rendered](#) [Log](#) [All Instances](#) [Filter Upstream](#)

Download Log (by attempts):
1

Task Actions

[Ignore All Deps](#) [Ignore Task State](#) [Ignore Task Deps](#) [Run](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#) [Recursive](#) [Failed](#) [Clear](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#) [Mark Failed](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#) [Mark Success](#)

[Close](#)

Print statement displayed in logs

```
[2020-07-20T08:30:00Z] INFO {task_id:123} - Task executed with return code 0
[2020-07-20T08:30:00Z] INFO {task_id:123} - A command was successfully executed. Now performing check
[2020-07-20T08:30:00Z] INFO {task_id:123} - Marking task as SUCCESS. lock=domus\osad-locker\$_processes, task=task\$_notifl, execution=
[2020-07-20T08:30:00Z] INFO {task_id:123} - Command executed with return code 0
[2020-07-20T08:30:00Z] INFO {task_id:123} - There are now 2 warnings.
[2020-07-20T08:30:00Z] INFO {task_id:123} - Output:
[2020-07-20T08:30:00Z] INFO {task_id:123} - There are now 2 warnings.
echo "There are now 2 warnings | nc -z $) | rm -f"
\rm
[2020-07-20T08:30:00Z] INFO {task_id:123} - Task root location:
[2020-07-20T08:30:00Z] INFO {task_id:123} - /var/lib/docker/overlay2/53515a0525050505-exec5d5cfd5a05250505/mounts/d1_nua_dac_ctx_mkdir/via_task_id=task\$_notifl
[2020-07-20T08:30:00Z] INFO {task_id:123} - lock=domus\osad-locker\$_processes
[2020-07-20T08:30:00Z] INFO {task_id:123} - Exceeded maximum time for command execution: 00:00:00
[2020-07-20T08:30:00Z] INFO {task_id:123} - A command <task_id> was not yet completed: task\$_notifl
[2020-07-20T08:30:00Z] INFO {task_id:123} - Supertask not yet completed
[2020-07-20T08:30:00Z] INFO {task_id:123} - [root@domus osad]# task_id:123
[2020-07-20T08:30:00Z] INFO {task_id:123} - [root@domus osad]# task_id:123
[2020-07-20T08:30:00Z] INFO {task_id:123} - Stopped process 483 for task
[2020-07-20T08:30:00Z] INFO {task_id:123} - Executed <task_id> on task\$_notifl: task\$_notifl
-----[2020-07-20T08:30:05Z] INFO {task_id:123} - 
[2020-07-20T08:30:05Z] INFO {task_id:123} - Stopped attempt to do it
-----[2020-07-20T08:30:05Z] INFO {task_id:123} - 
[2020-07-20T08:30:05Z] INFO {task_id:123} - Dependence was met for <task_id>: task\$_notifl
[2020-07-20T08:30:05Z] INFO {task_id:123} - Dependence was met for <task_id>: task\$_notifl
[2020-07-20T08:30:05Z] INFO {task_id:123} - Dependence was met for <task_id>: task\$_notifl
*** Received task title: /obj\$\$task_id\$\$process\$_notifl\$\$task\$_notifl
[2020-07-20T08:30:05Z] INFO {task_id:123} -
```

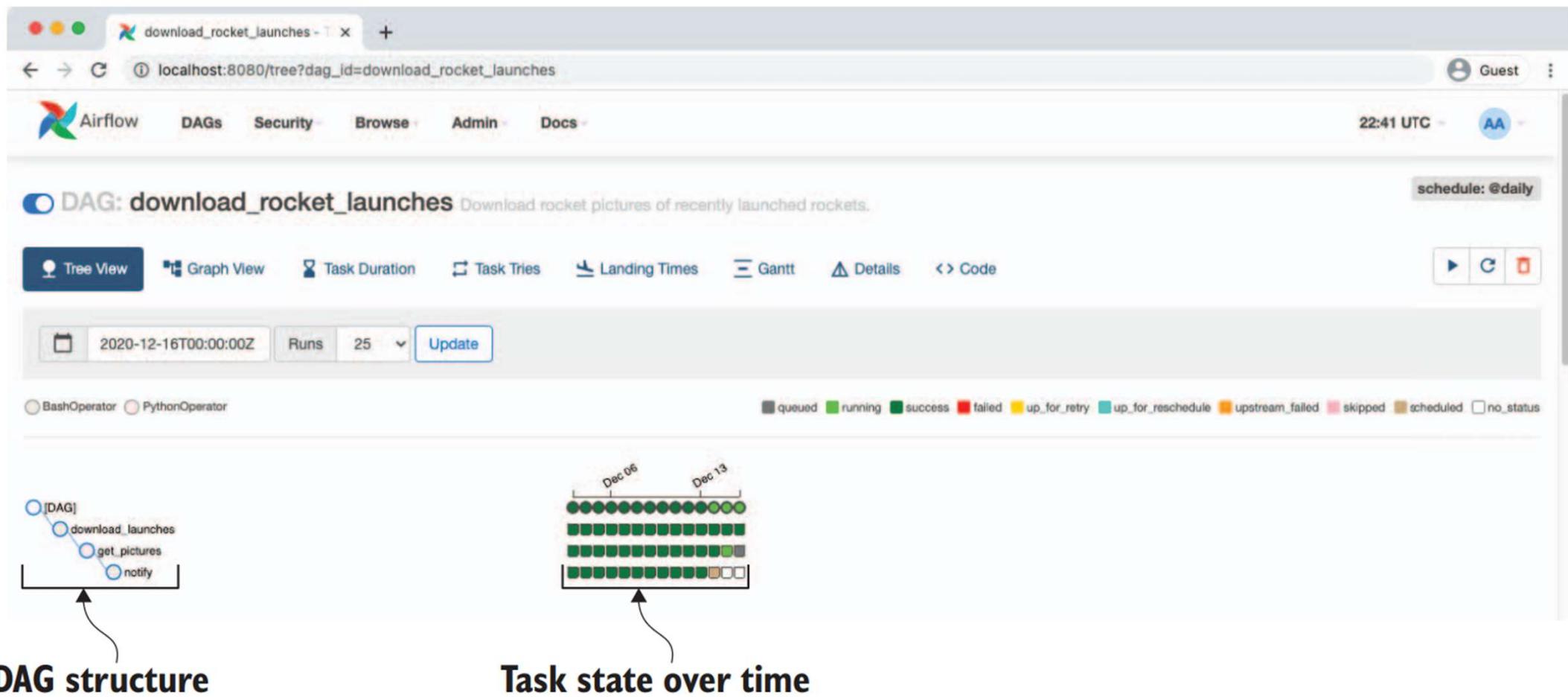
Running a DAG once a day

```
dag = DAG(  
    dag_id="download_rocket_launches",  
    start_date=airflow.utils.dates.days_ago(14),  
    schedule_interval="@daily",  
)
```

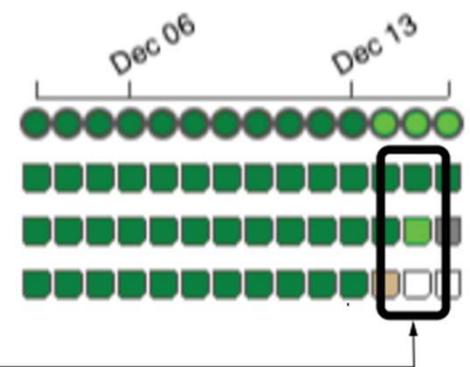
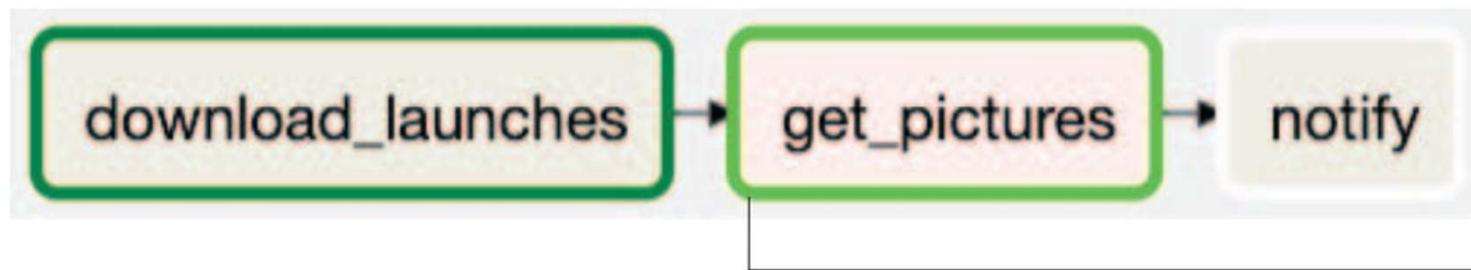


Airflow alias for 0 0
* * * (i.e., midnight)

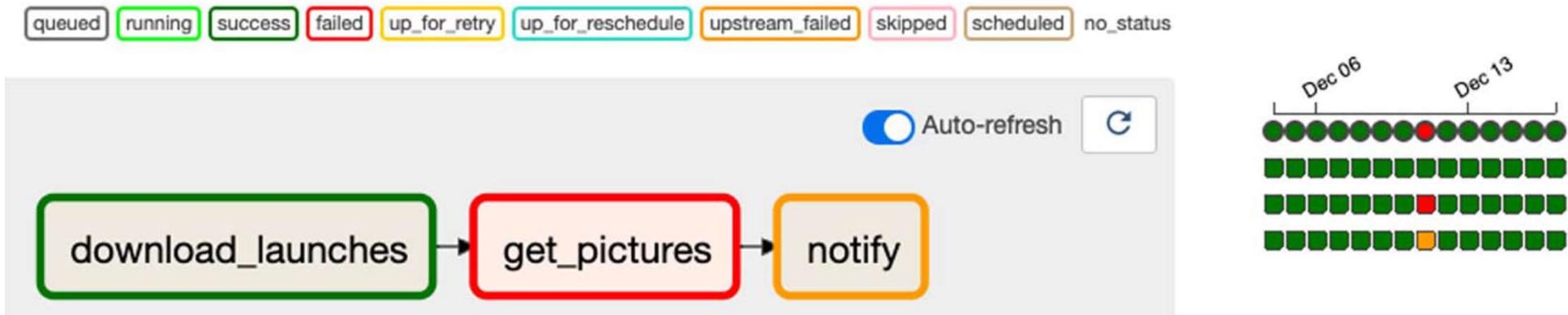
Airflow tree view



Relationship between graph view and tree view



Failure displayed in graph view and tree view



Stack trace of failed get_pictures task

DAG: download_rockets launches Download rocket pictures of recently launched rockets.

Tree View Graph View Task Duration Task Tries Landing Times Gantt Details Code

Task Instance: get_pictures at 2020-12-11 01:00:00+01:0

Task Instance Details Rendered Template Log XCom

Log by attempts

1 2 3 4 5

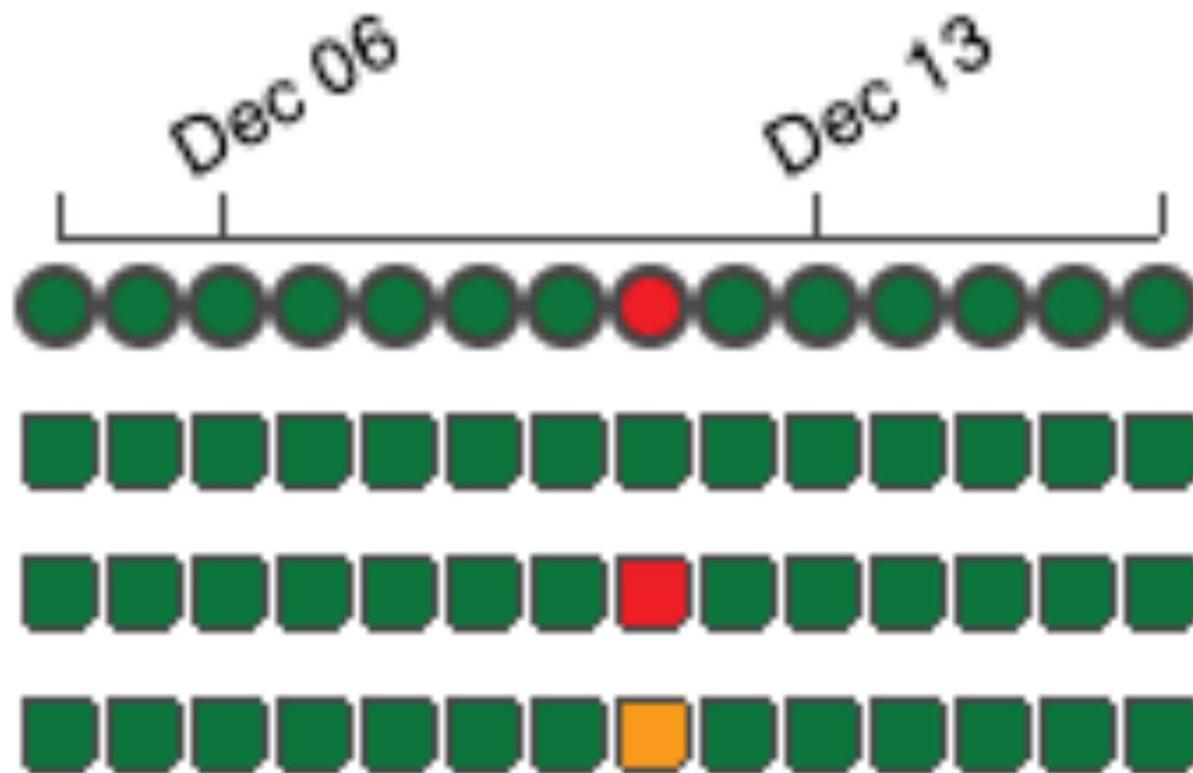
... skipped for brevity ...

```
[2020-12-18 22:00:58,638] {taskinstance.py:1396} ERROR - HTTPSConnectionPool(host='launchlibrary1.nyc3.digitaloceanspaces.com', port=443): Max retries exceeded with url: /Rocket
Traceback (most recent call last):
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connection.py", line 159, in _new_conn
    conn = connection.create_connection()
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/util/connection.py", line 61, in create_connection
    for res in socket.getaddrinfo(host, port, family, socket.SOCK_STREAM):
  File "/usr/local/lib/python3.8/socket.py", line 918, in getaddrinfo
    for res in _socket.getaddrinfo(host, port, family, type, proto, flags):
socket.gaierror: [Errno -2] Name or service not known

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connectionpool.py", line 670, in urlopen
    httplib_response = self._make_request(
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connectionpool.py", line 381, in _make_request
    self._validate_conn(conn)
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connectionpool.py", line 978, in _validate_conn
    conn.connect()
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connection.py", line 309, in connect
    conn = self._new_conn()
  File "/home/airflow/.local/lib/python3.8/site-packages/urllib3/connection.py", line 171, in _new_conn
    raise NewConnectionError(
urllib3.exceptions.NewConnectionError: <urllib3.connection.HTTPSConnection object at 0x7f37963ce3a0>: Failed to establish a new connection: [Errno -2] Name or service not known
```

Click on a failed task for options to clear it



Clearing the state of get_pictures and successive tasks

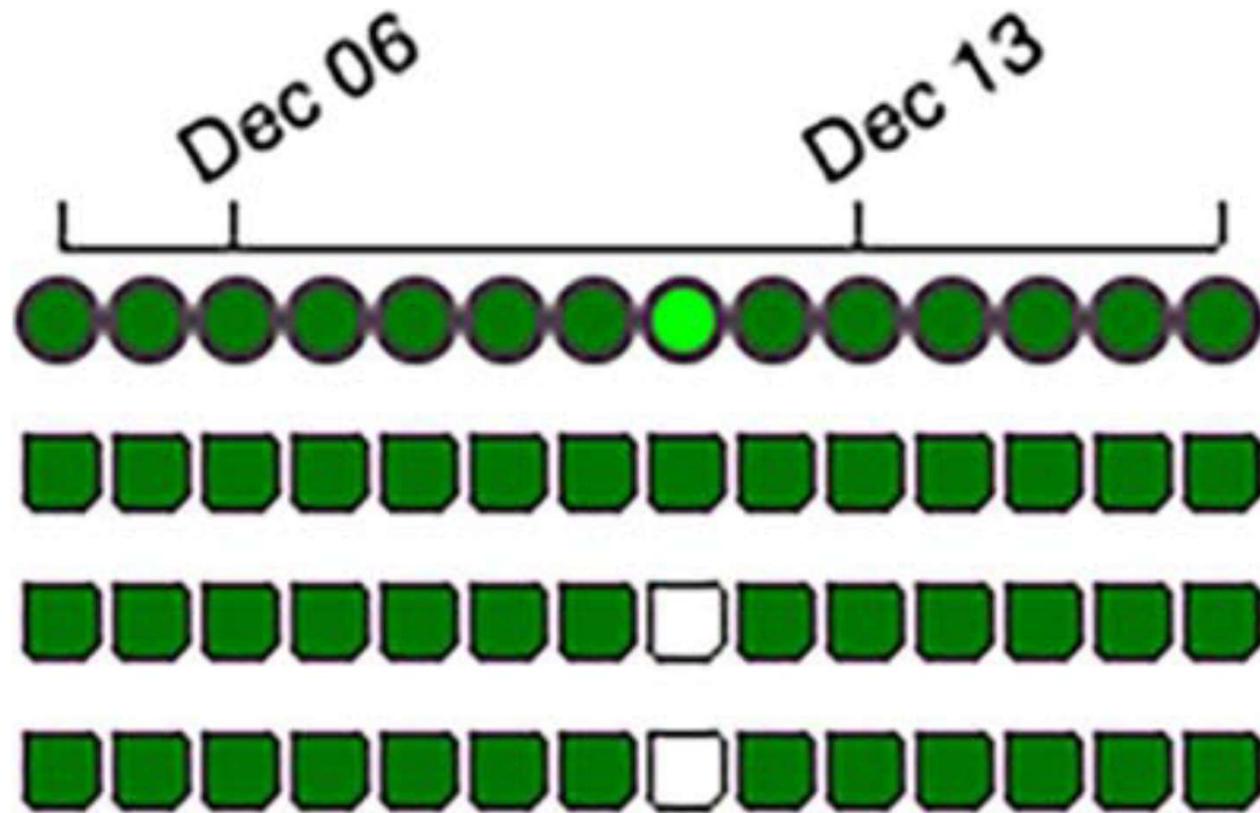
Here's the list of task instances you are about to clear:

```
<TaskInstance: download_rocket_launches.get_pictures 2020-12-11 00:00:00+00:00 [failed]>
<TaskInstance: download_rocket_launches.notify 2020-12-11 00:00:00+00:00 [upstream_failed]>
```

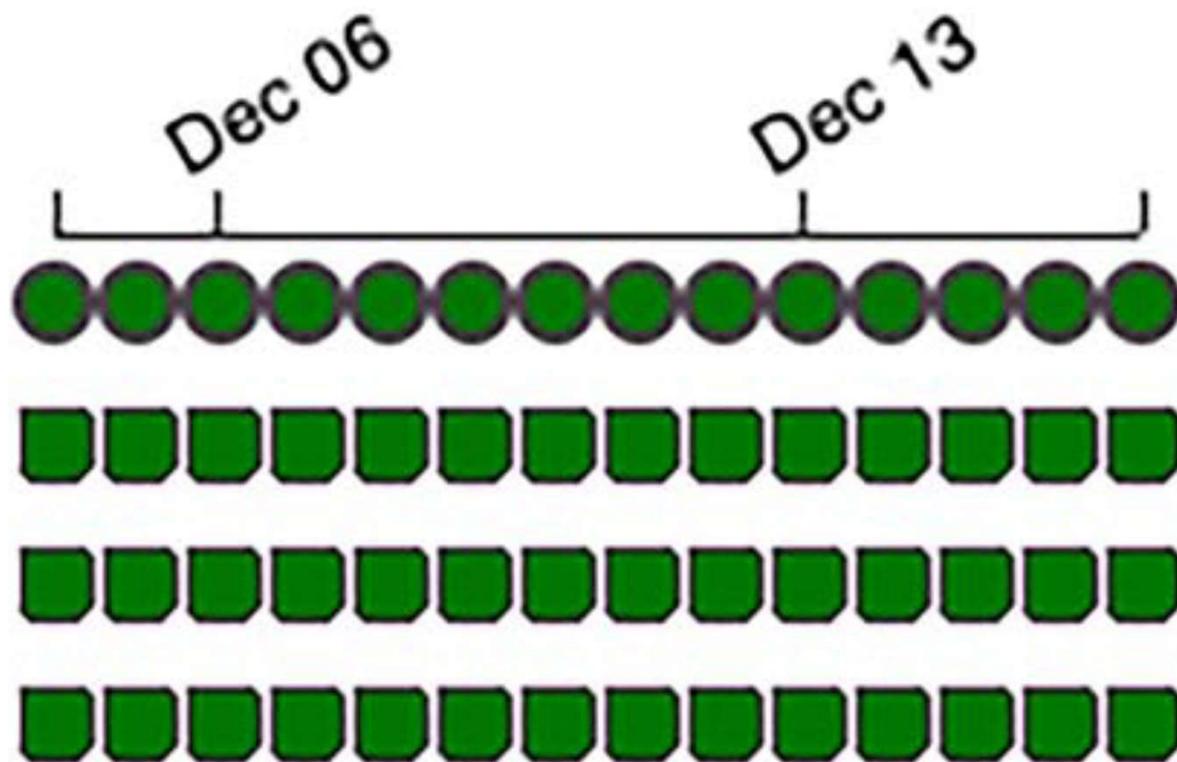
OK!

Cancel

Cleared tasks displayed in graph view



Successfully completed tasks after clearing failed tasks



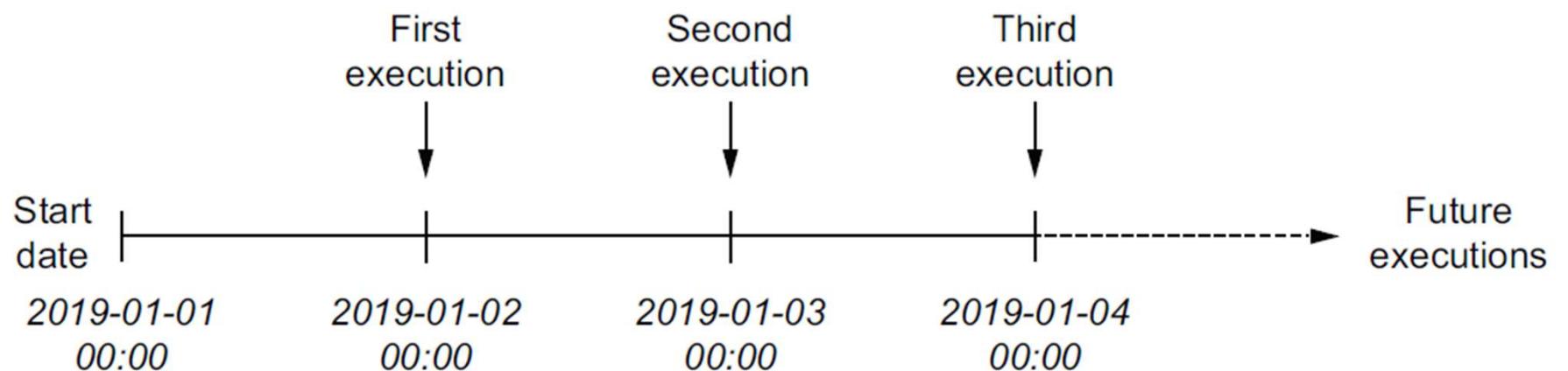
Scheduling in Airflow

Defining a daily schedule interval

```
dag = DAG(  
    dag_id="02_daily_schedule",  
    schedule_interval="@daily",  
    start_date=dt.datetime(2019, 1, 1),  
    ...  
)
```

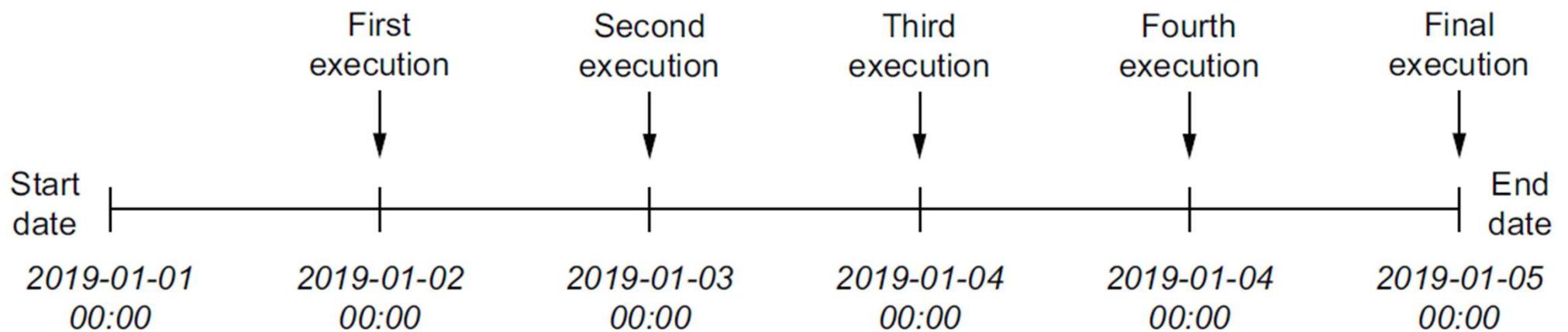
Schedule the DAG to run every day at midnight.

Date/time to start scheduling DAG runs



Schedule intervals with specified start and end dates

```
dag = DAG(  
    dag_id="03_with_end_date",  
    schedule_interval="@daily",  
    start_date=dt.datetime(year=2019, month=1, day=1),  
    end_date=dt.datetime(year=2019, month=1, day=5),  
)
```



Cron-based intervals

```
#          minute (0 - 59)
#          |         hour (0 - 23)
#          |         |        day of the month (1 - 31)
#          |         |         |       month (1 - 12)
#          |         |         |         |      day of the week (0 - 6) (Sunday to Saturday;
#          |         |         |         |           7 is also Sunday on some systems)
# * * * * *
```

- 0 * * * * = hourly (running on the hour)
- 0 0 * * * = daily (running at midnight)
- 0 0 * * 0 = weekly (running at midnight on Sunday)

Cron-based intervals

- 0 0 1 * * = midnight on the first of every month
 - 45 23 * * SAT = 23:45 every Saturday
-
- 0 0 * * MON,WED,FRI = run every Monday, Wednesday, Friday at midnight
 - 0 0 * * MON-FRI = run every weekday at midnight
 - 0 0,12 * * * = run every day at 00:00 and 12:00

Airflow presets for frequently used scheduling intervals

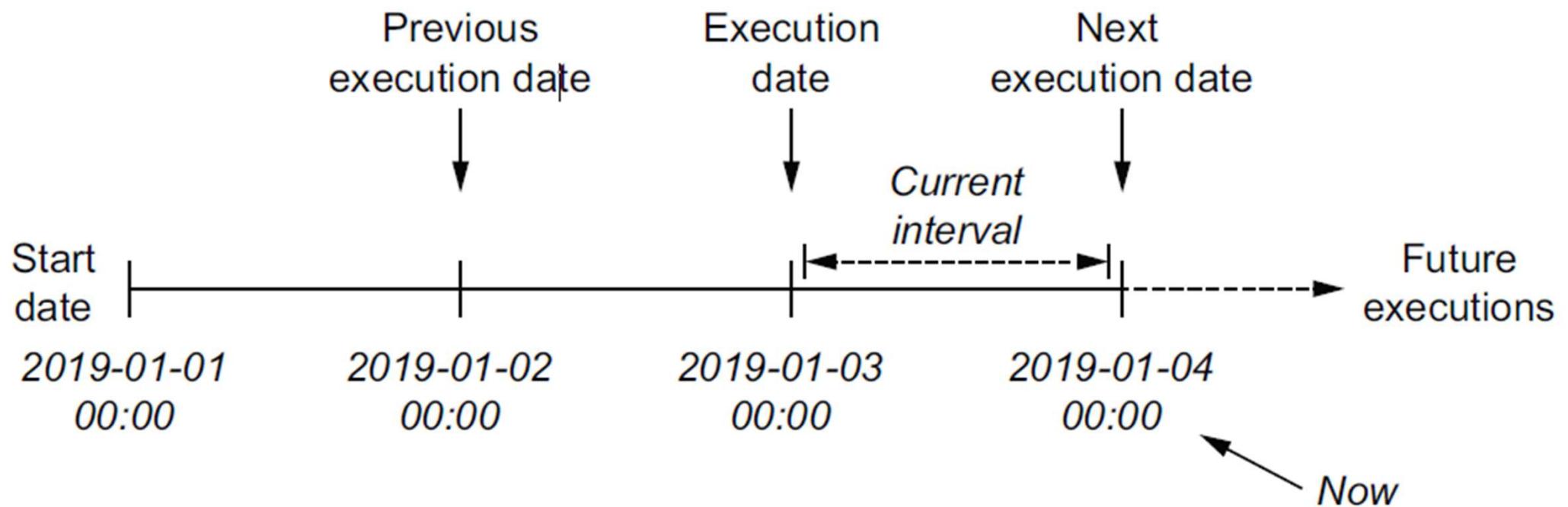
Preset	Meaning
@once	Schedule once and only once.
@hourly	Run once an hour at the beginning of the hour.
@daily	Run once a day at midnight.
@weekly	Run once a week at midnight on Sunday morning.
@monthly	Run once a month at midnight on the first day of the month.
@yearly	Run once a year at midnight on January 1.

Defining a frequency-based schedule interval

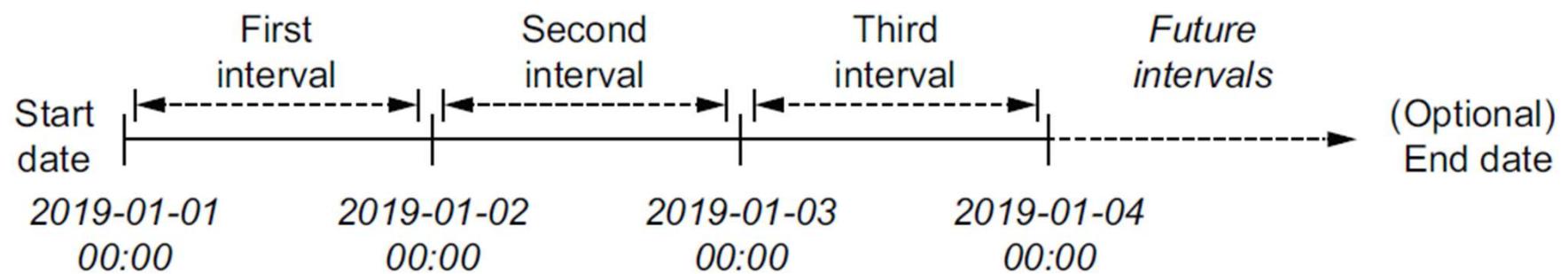
```
dag = DAG(  
    dag_id="04_time_delta",  
    schedule_interval=dt.timedelta(days=3),  
    start_date=dt.datetime(year=2019, month=1, day=1),  
    end_date=dt.datetime(year=2019, month=1, day=5),  
)
```

timedelta gives the ability
to use frequency-based
schedules.

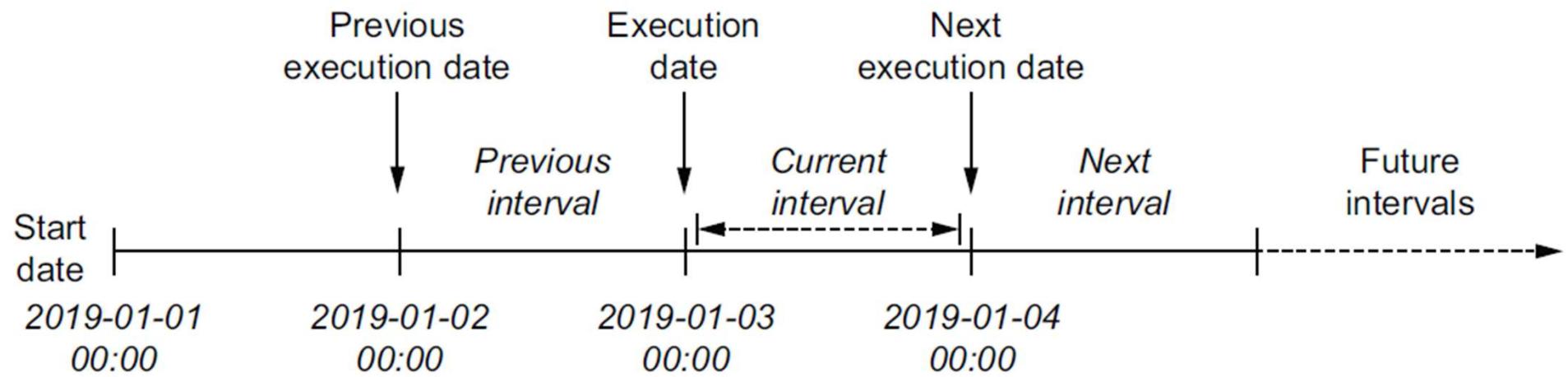
Dynamic time references using execution dates



Understanding Airflow's execution dates



Execution Date insights



Using backfilling to fill in past gaps

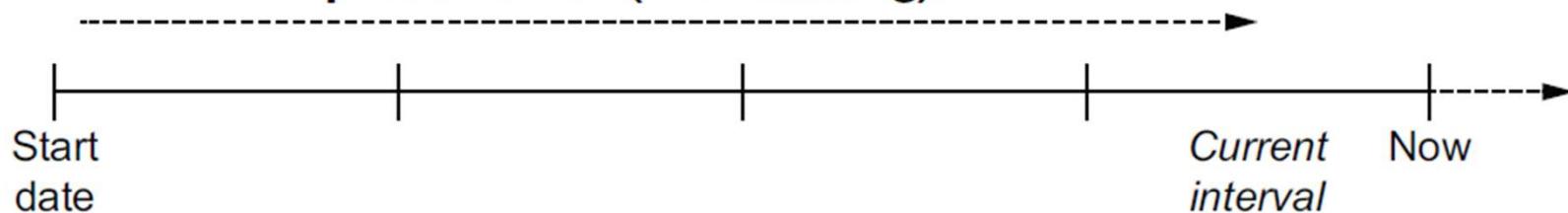
- Disabling catchup to avoid running past runs

```
dag = DAG(  
    dag_id="09_no_catchup",  
    schedule_interval="@daily",  
    start_date=dt.datetime(year=2019, month=1, day=1),  
    end_date=dt.datetime(year=2019, month=1, day=5),  
    catchup=False,  
)
```

Backfilling in Airflow

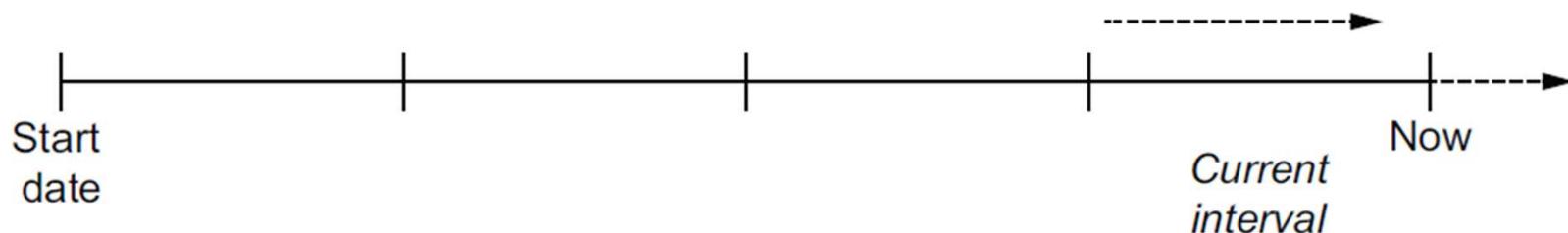
Catchup = true (default)

Airflow starts processing, including past intervals (= backfilling).



Catchup = false

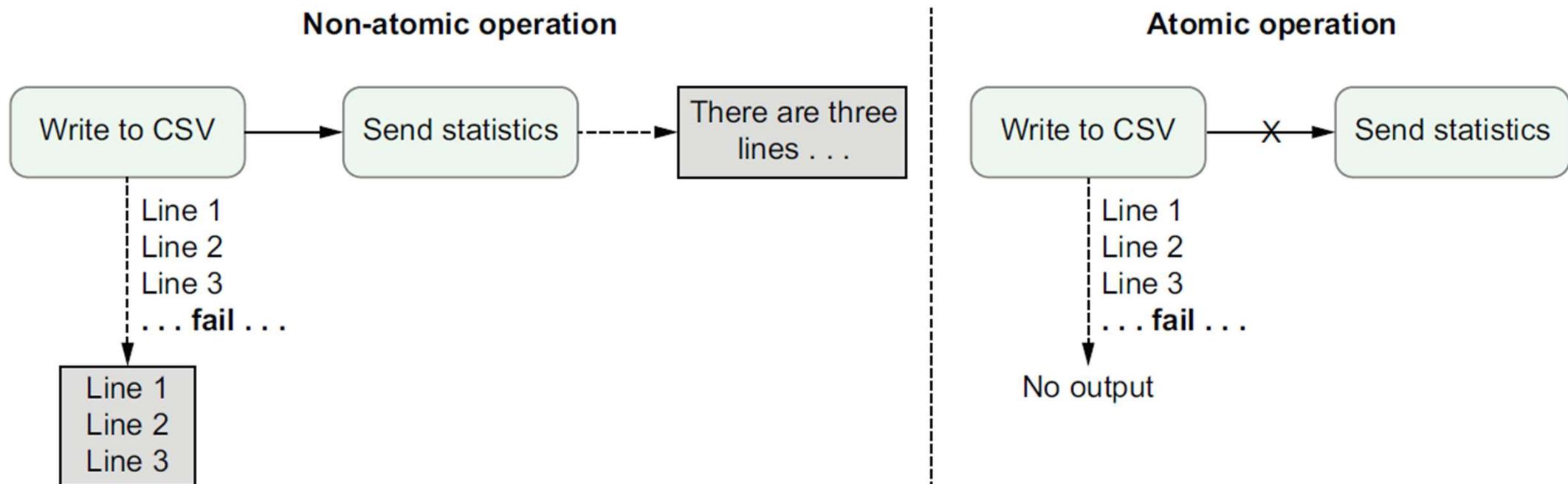
Airflow starts processing from the current interval.



Best practices for designing tasks

Atomicity

- Ensures either everything or nothing completes



Two jobs in one task, to break atomicity

```
def _calculate_stats(**context):
    """Calculates event statistics."""
    input_path = context["templates_dict"]["input_path"]
    output_path = context["templates_dict"]["output_path"]

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)

    email_stats(stats, email="user@example.com")
```

Sending an email after writing to CSV creates two pieces of work in a single function, which breaks the atomicity of the task.

Splitting into multiple tasks to improve atomicity

```
def _send_stats(email, **context):
    stats = pd.read_csv(context["templates_dict"]["stats_path"])
    email_stats(stats, email=email)           ←
send_stats = PythonOperator(
    task_id="send_stats",
    python_callable=_send_stats,
    op_kwargs={"email": "user@example.com"},
    templates_dict={"stats_path": "/data/stats/{{ds}}.csv"},
    dag=dag,
)
calculate_stats >> send_stats
```

Split off the `email_stats` statement into a separate task for atomicity.

Idempotency

- Existing implementation for fetching events

```
fetch_events = BashOperator(  
    task_id="fetch_events",  
    bash_command=  
        "mkdir -p /data/events && "  
        "curl -o /data/events/{{ds}}.json "      ←  
        "http://localhost:5000/events?"  
        "start_date={{ds}}&"  
        "end_date={{next_ds}}"  
) ,  
    dag=dag,  
)
```

Partitioning by setting
templated filename

Non-idempotent task

Attempt 1

Process
data task

Line 1
Line 2
Line 3

Attempt 2

Process
data task

Line 1
Line 2
Line 3
Line 1
Line 2
Line 3

Attempt 3

Process
data task

Line 1
Line 2
Line 3
Line 1
Line 2
Line 3
Line 1
Line 2
Line 3

Idempotent task

Attempt 1

Process
data task

Line 1
Line 2
Line 3

Attempt 2

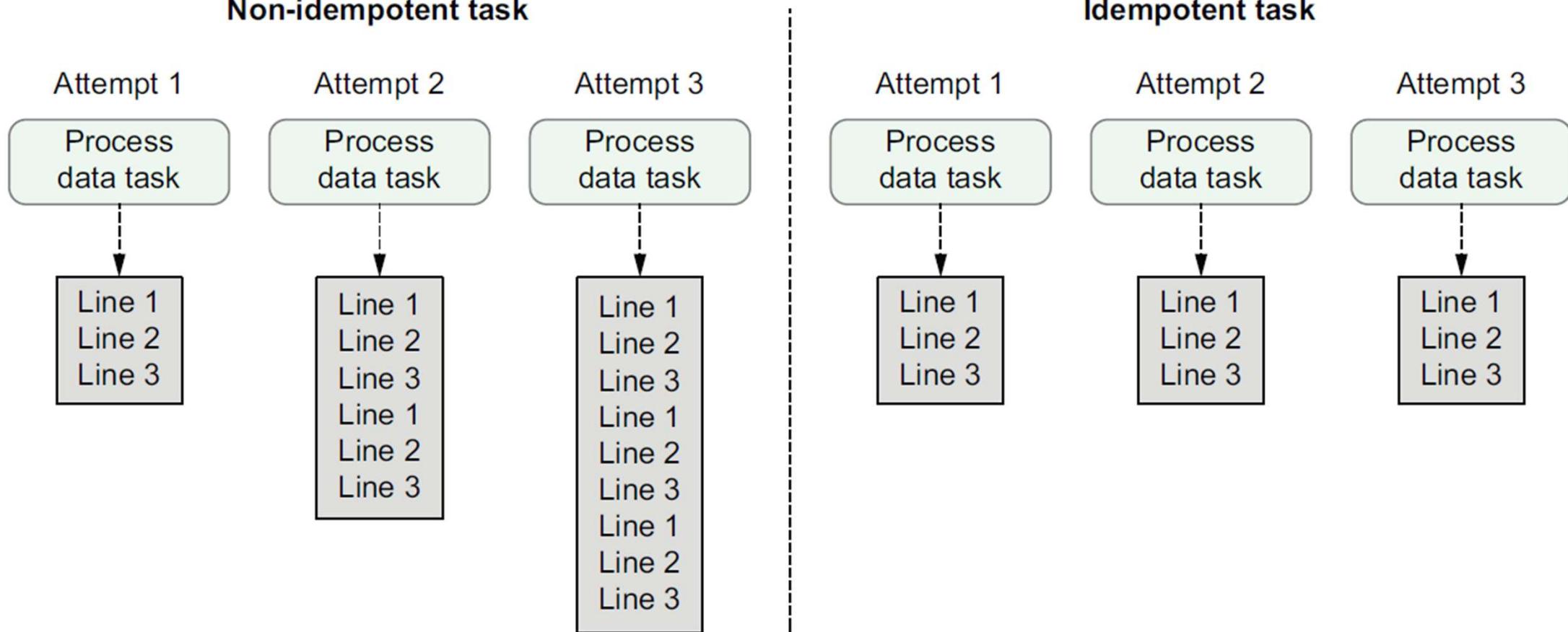
Process
data task

Line 1
Line 2
Line 3

Attempt 3

Process
data task

Line 1
Line 2
Line 3



Thanks