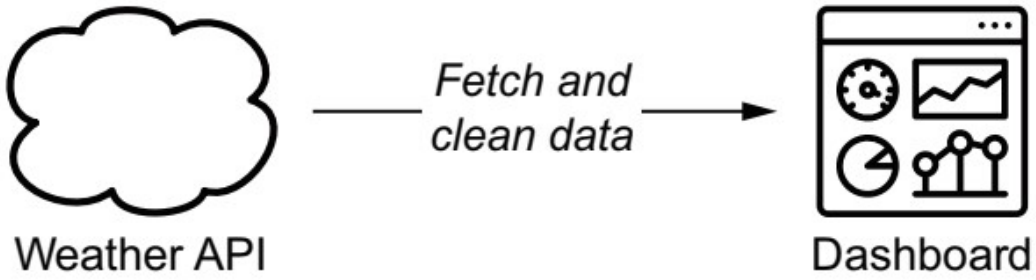
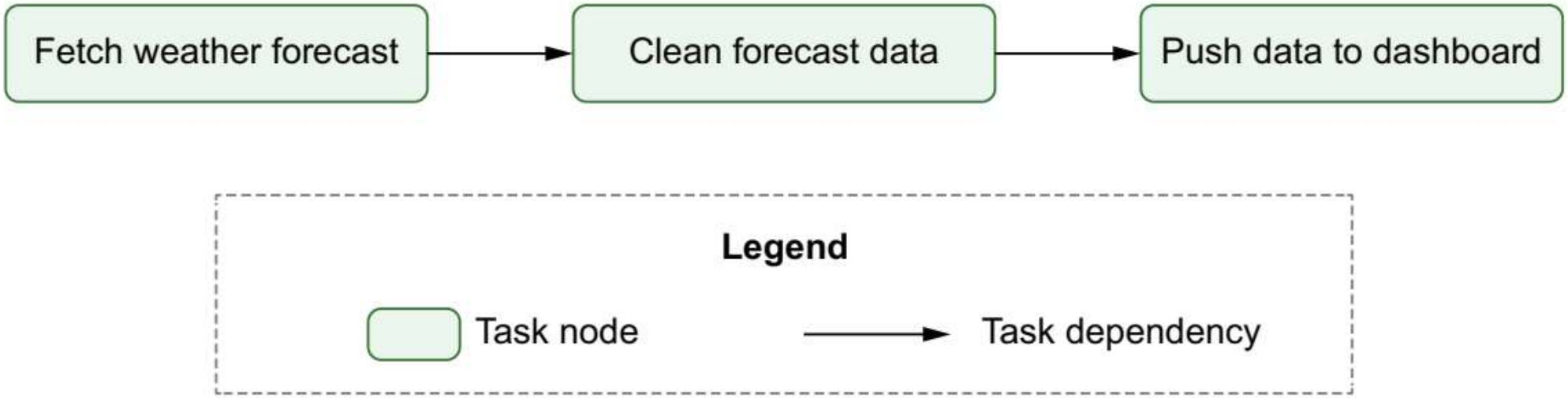
Airflow

Simple Complexity



More Complexity



DAG

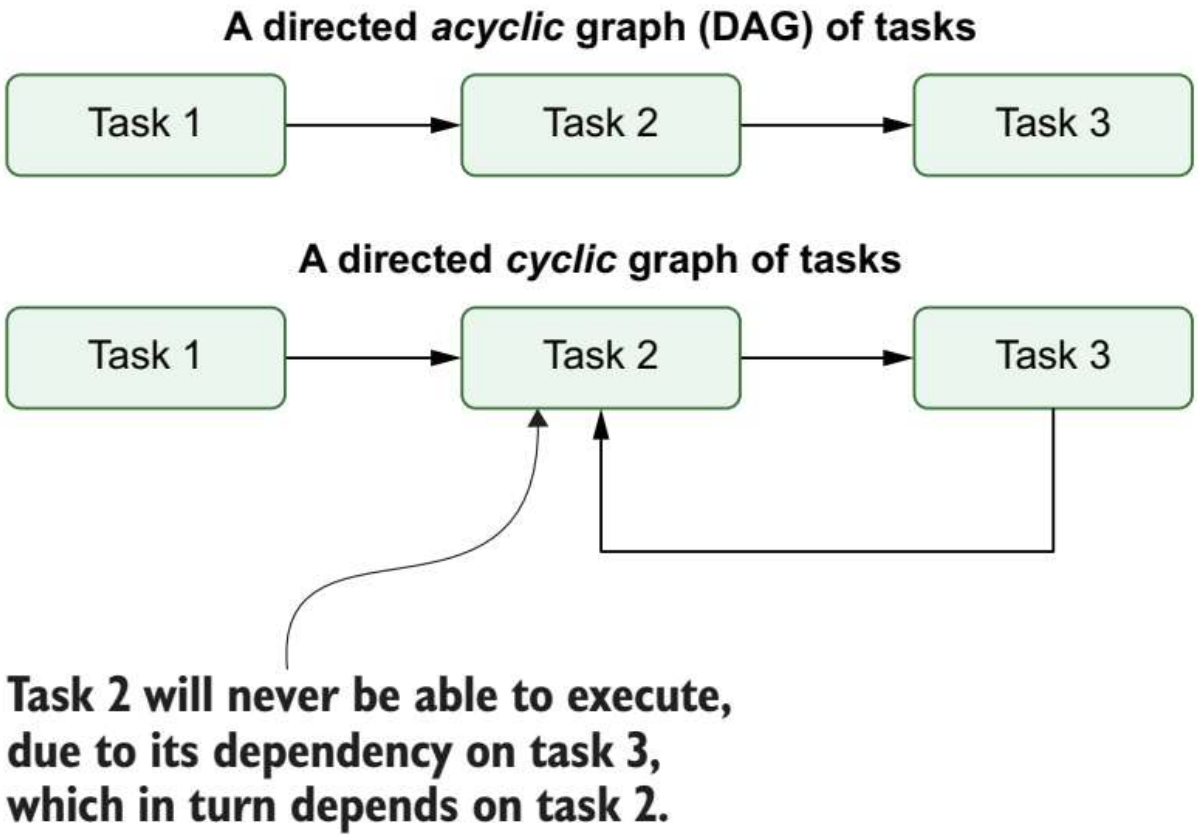
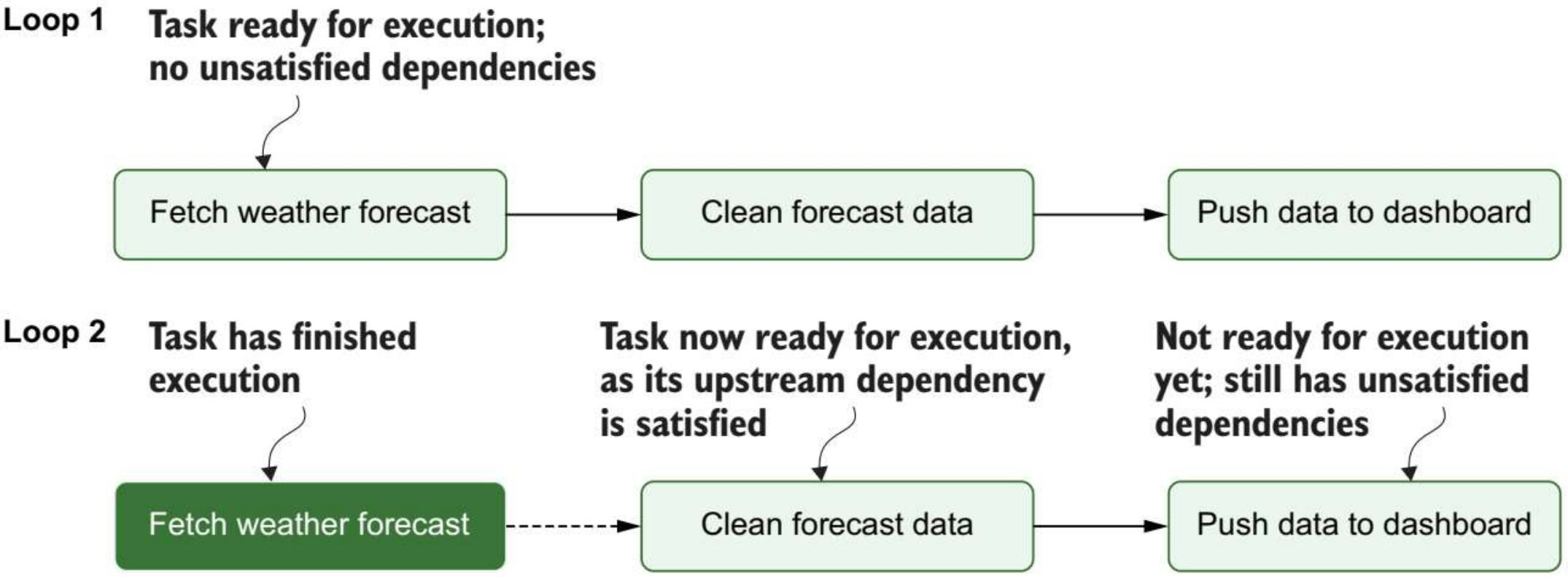


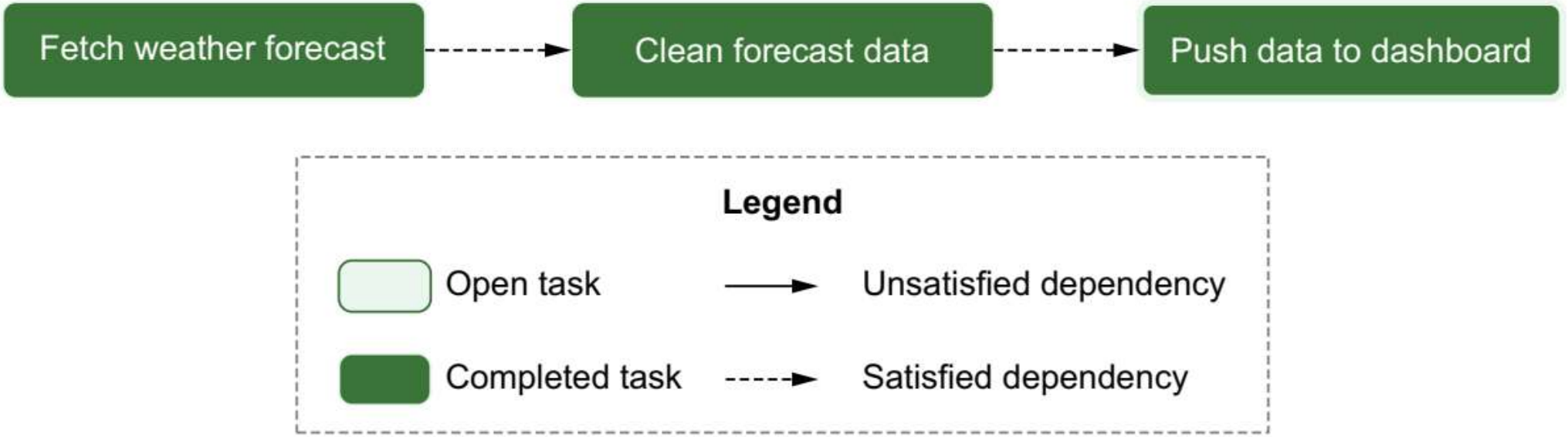
Figure 1.3 Cycles in graphs prevent task execution due to circular dependencies. In acyclic graphs (top), there is a clear path to execute the three different tasks. However, in cyclic graphs (bottom), there is no longer a clear execution path due to the interdependency between tasks 2 and 3.

How tasks are executed via DAG



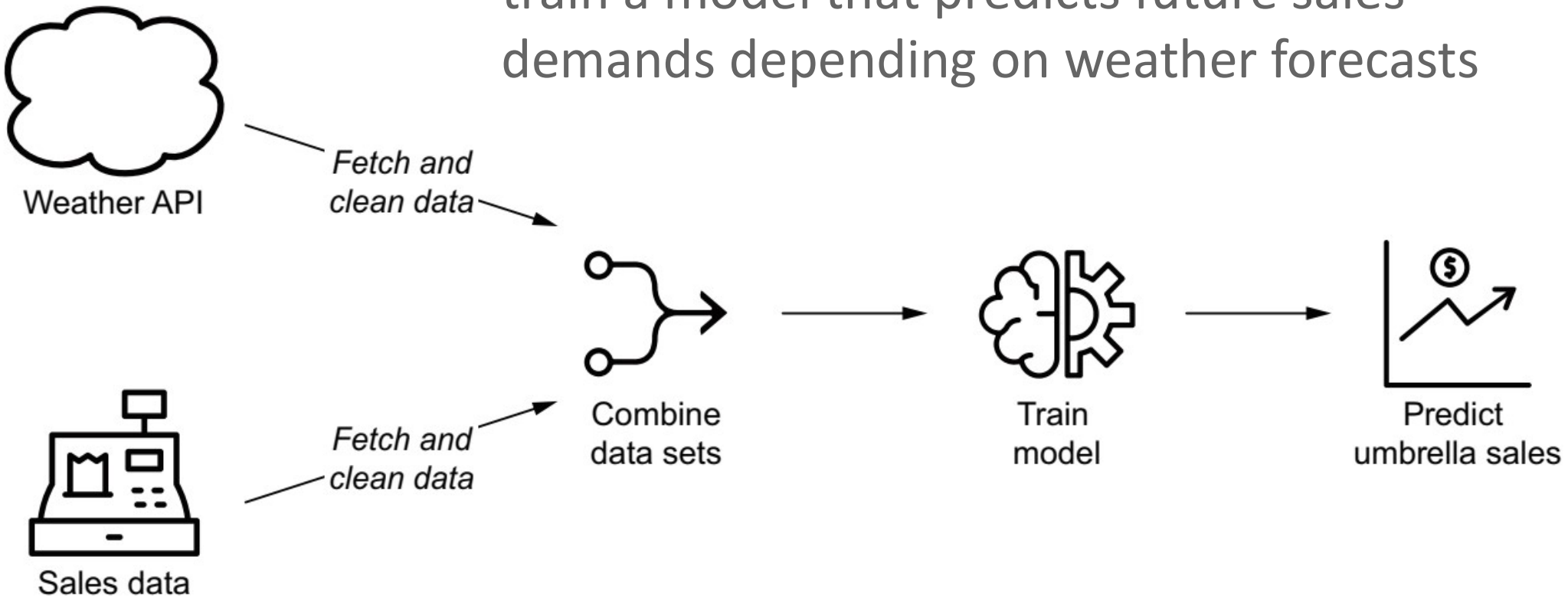
How tasks are executed via DAG

End state



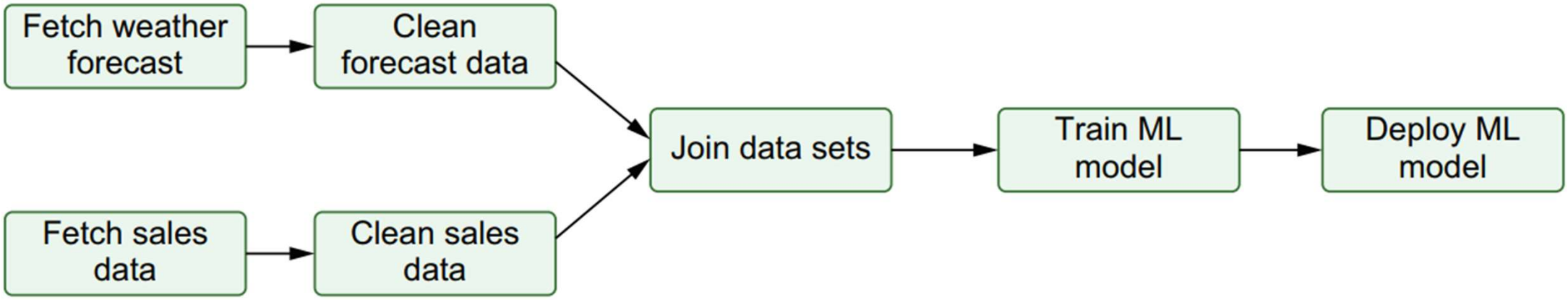
Overview of the umbrella demand use case

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.

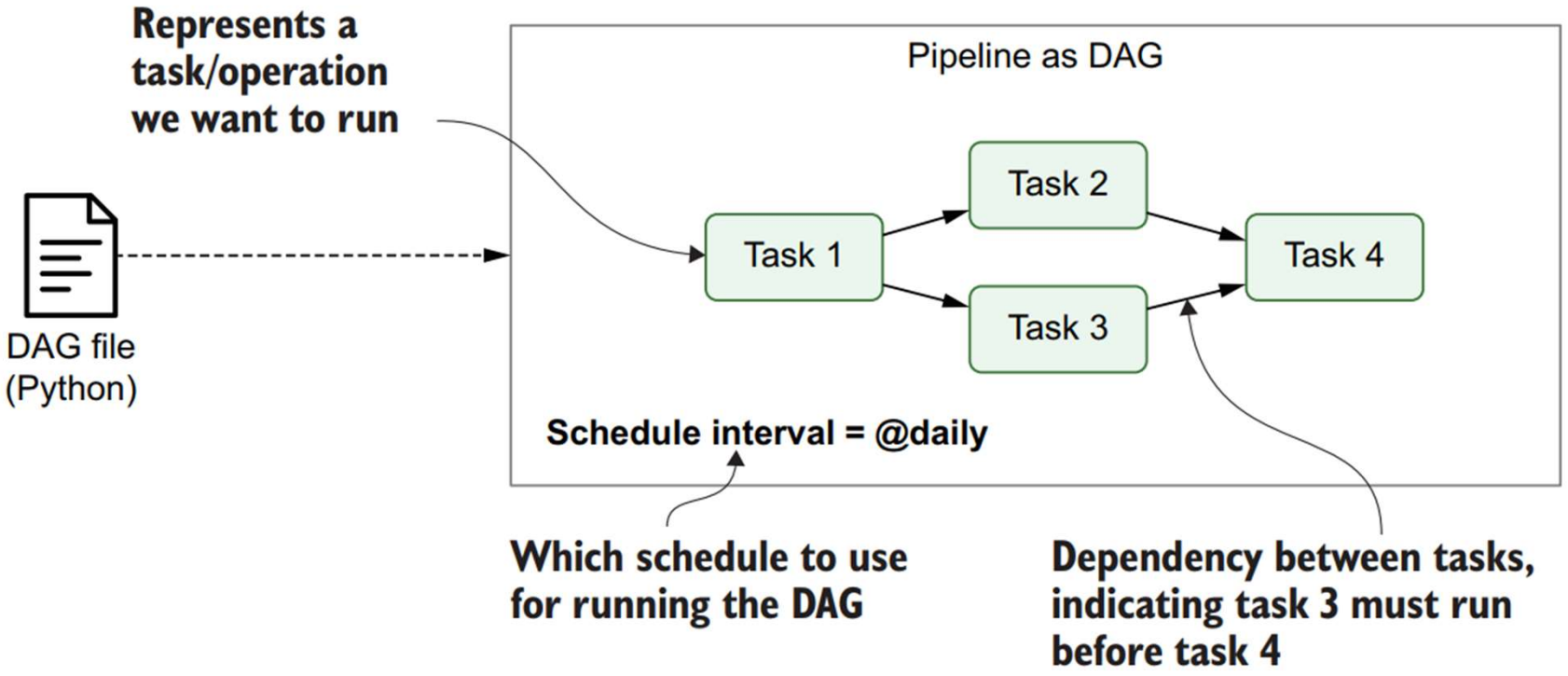


Overview of several well-known workflow managers

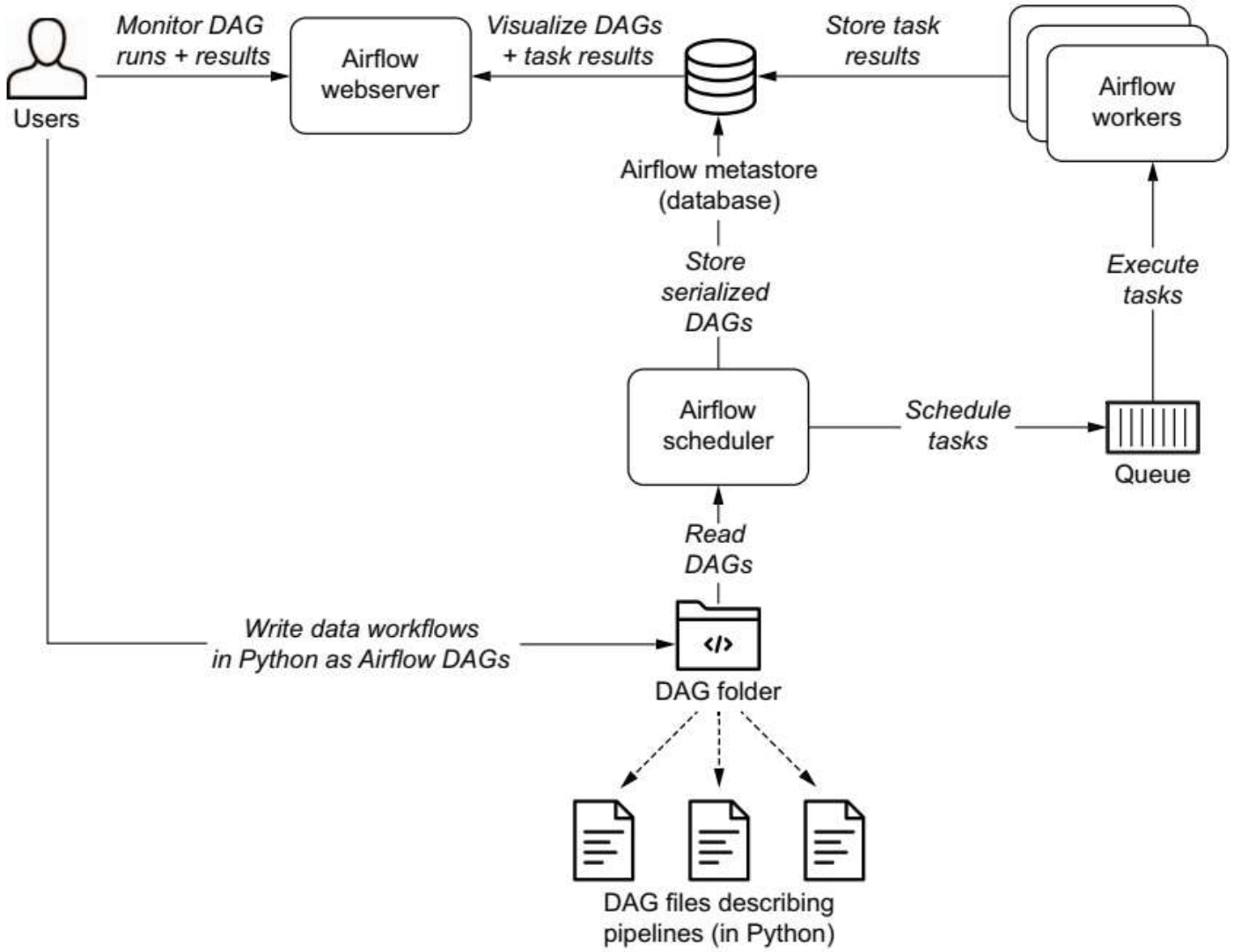
Name	Originated at[a]	Workflows defined in	Written in	Scheduling	Backfilling	User interface[b]	Installation platform	Horizontally scalable
Airflow	Airbnb	Python	Python	Yes	Yes	Yes	Anywhere	Yes
Argo	Applatix	YAML	Go	Third party[c]		Yes	Kubernetes	Yes
Azkaban	LinkedIn	YAML	Java	Yes	No	Yes	Anywhere	
Conductor	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

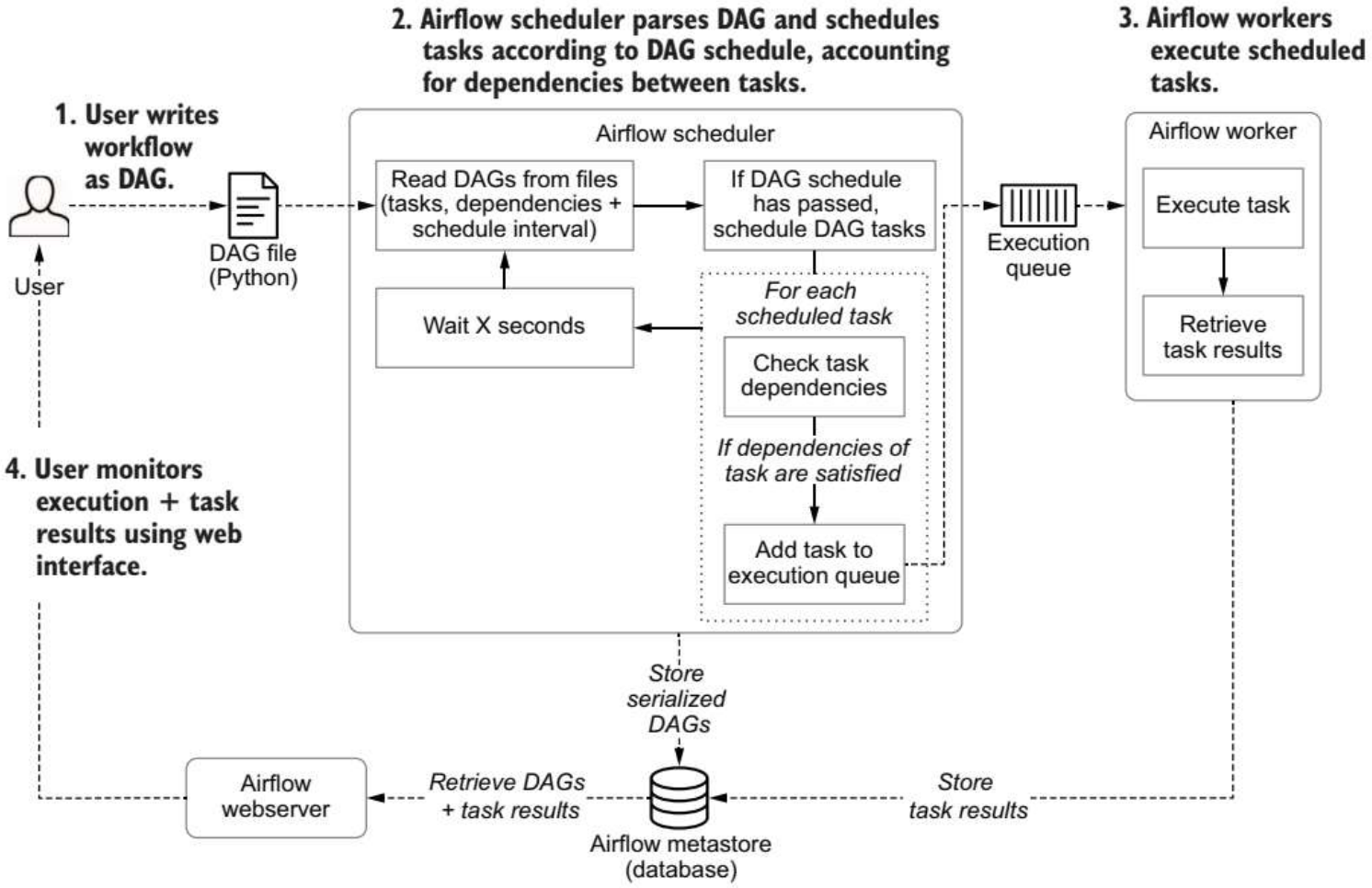
- Each DAG file typically defines one DAG, which describes the different tasks and their dependencies.
- Besides this, the DAG also defines a schedule interval that determines when the DAG is executed by Airflow



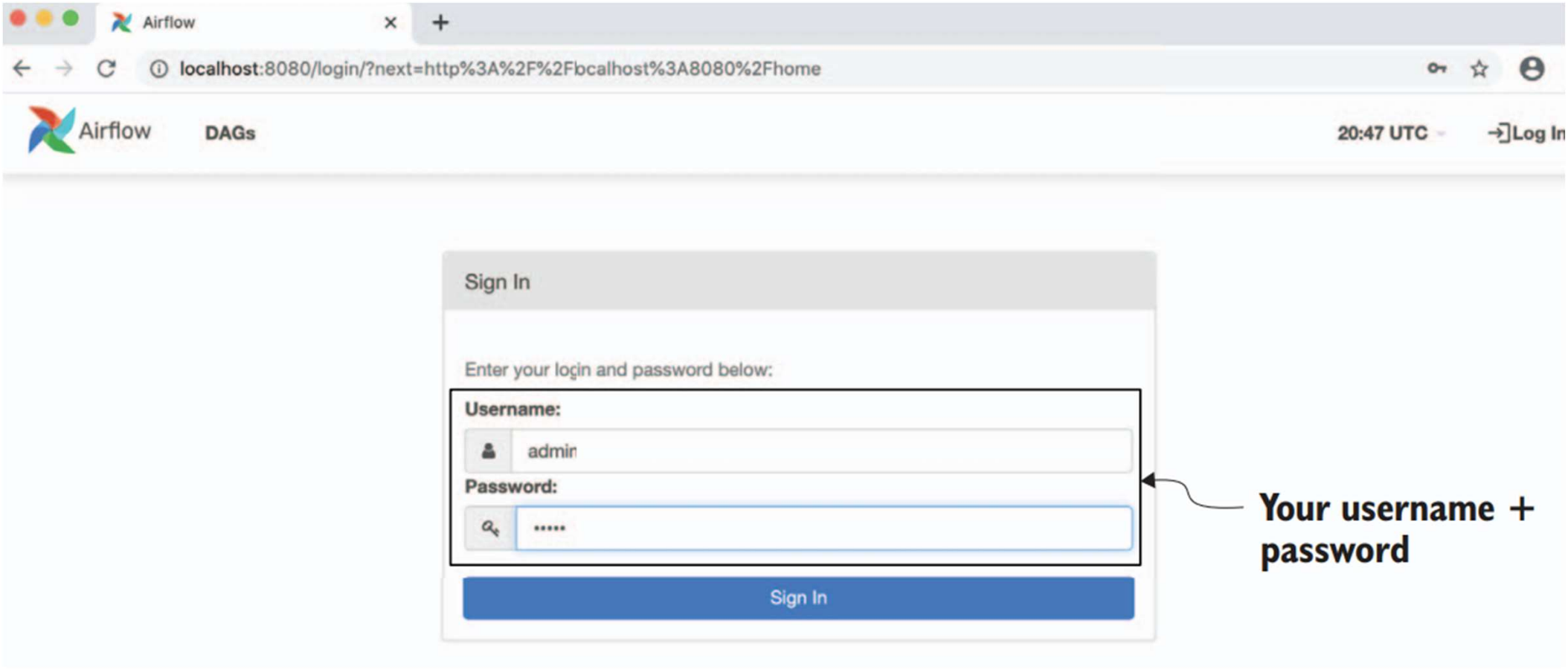
Overview of the main components involved in Airflow



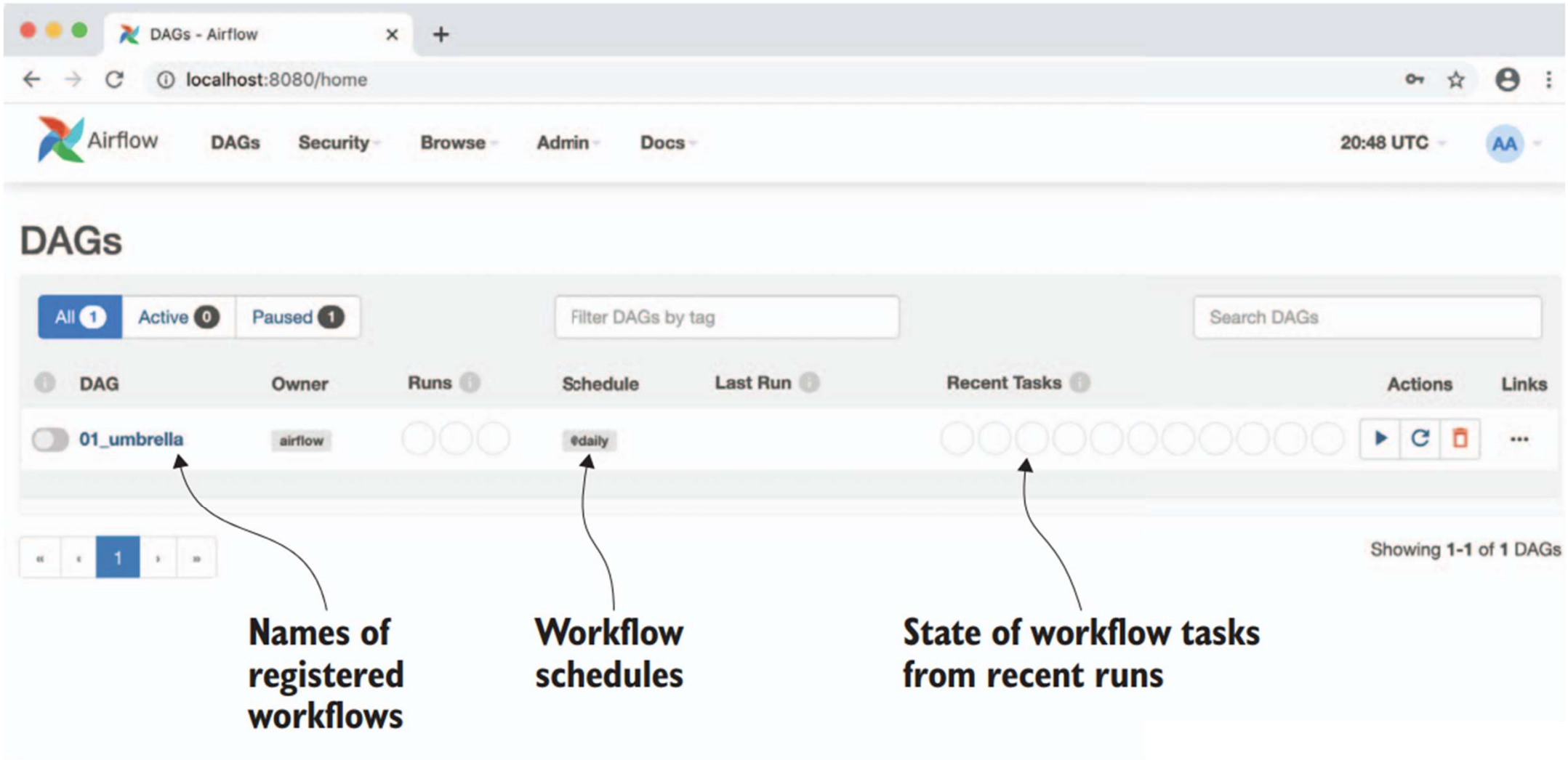
Schematic overview of the process involved in developing and executing pipelines as DAGs



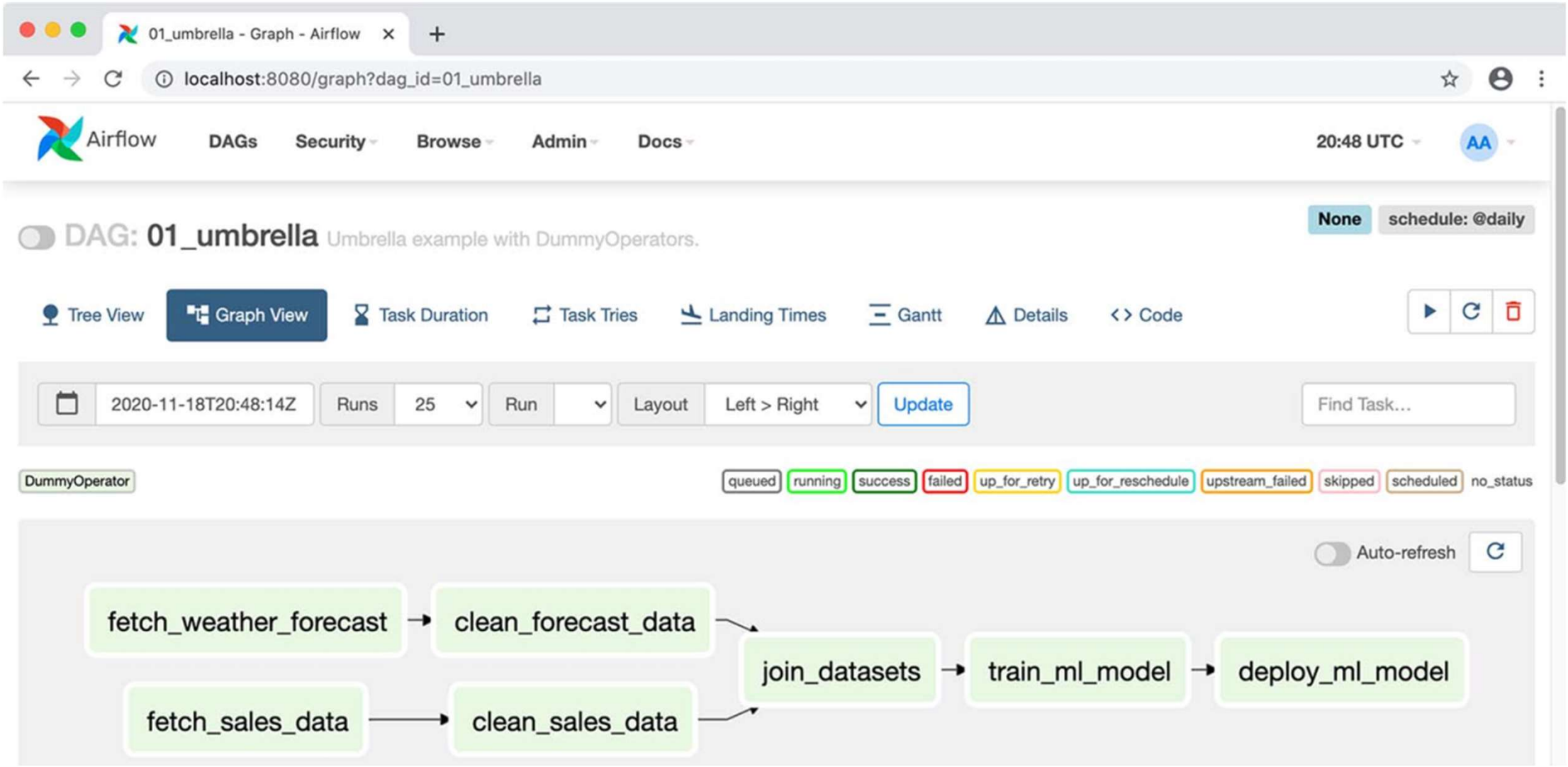
The login page for the Airflow web interface



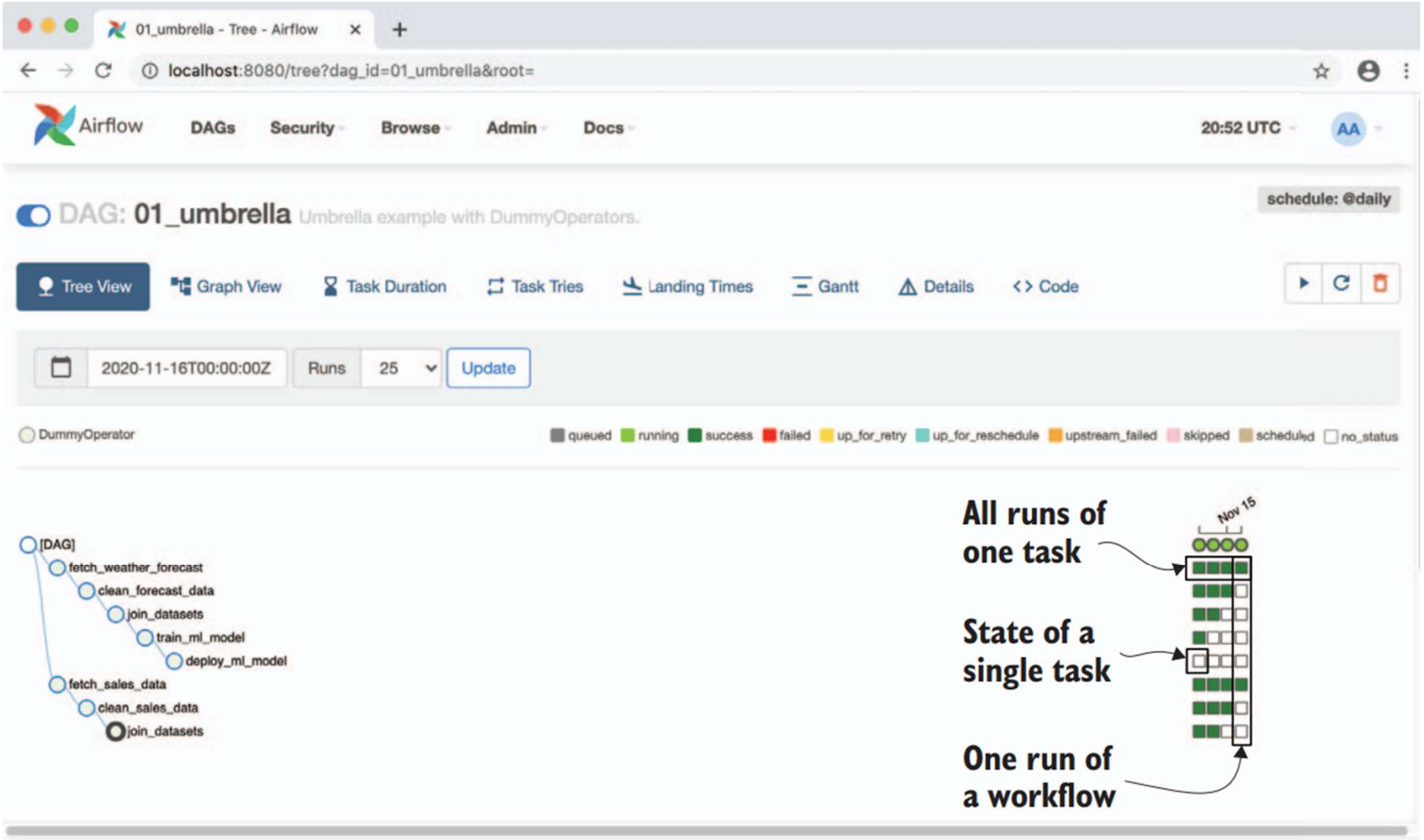
The main page of Airflow's web interface



The graph view in Airflow's web interface

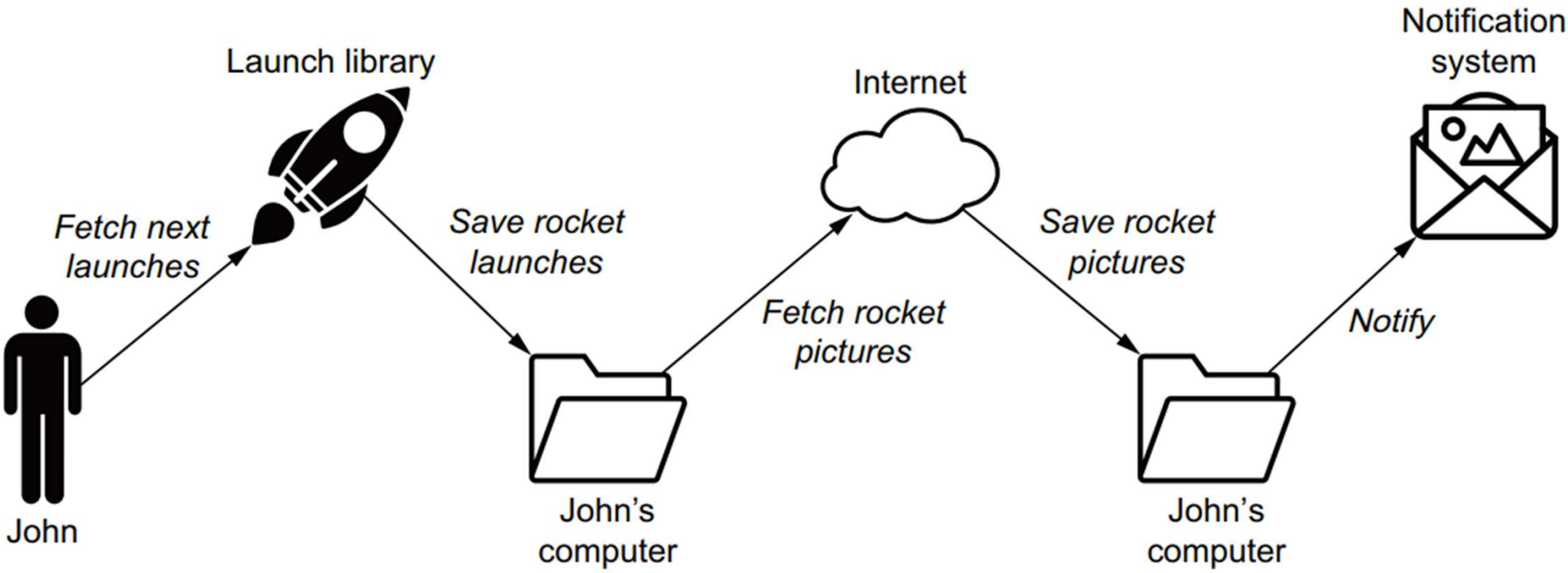


Airflow's tree view

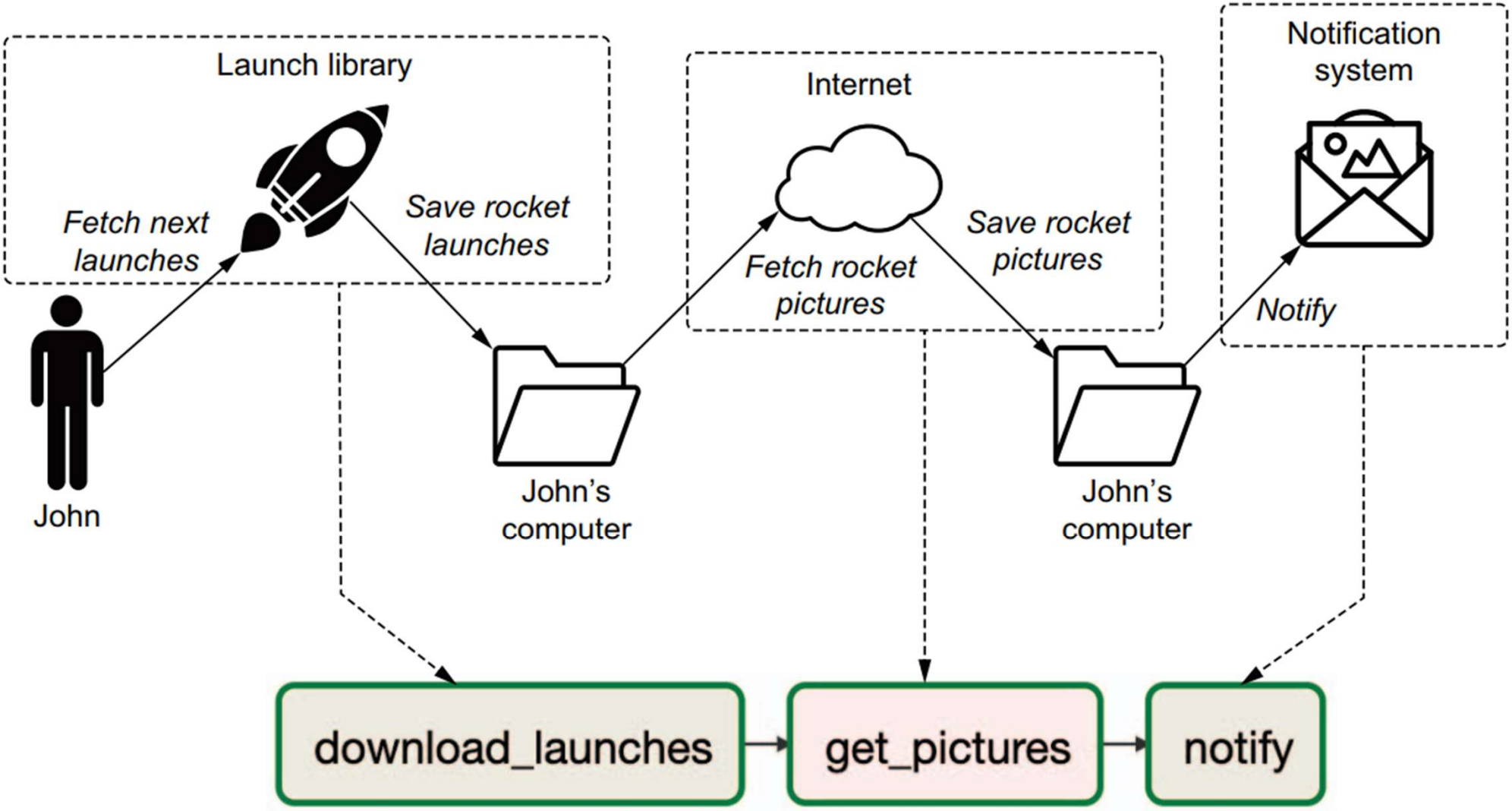


Anatomy of an Airflow DAG

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

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

```
)


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}.")
```

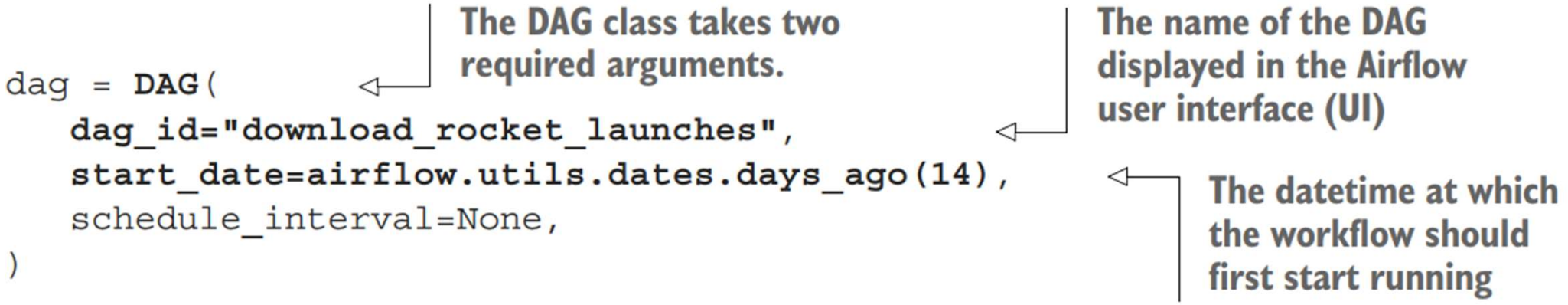
DAG for downloading and processing rocket launch data

```
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,
)
```

Instantiating a BashOperator to run a Bash command

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

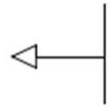
The name of the task

The Bash command to execute

Reference to the DAG variable

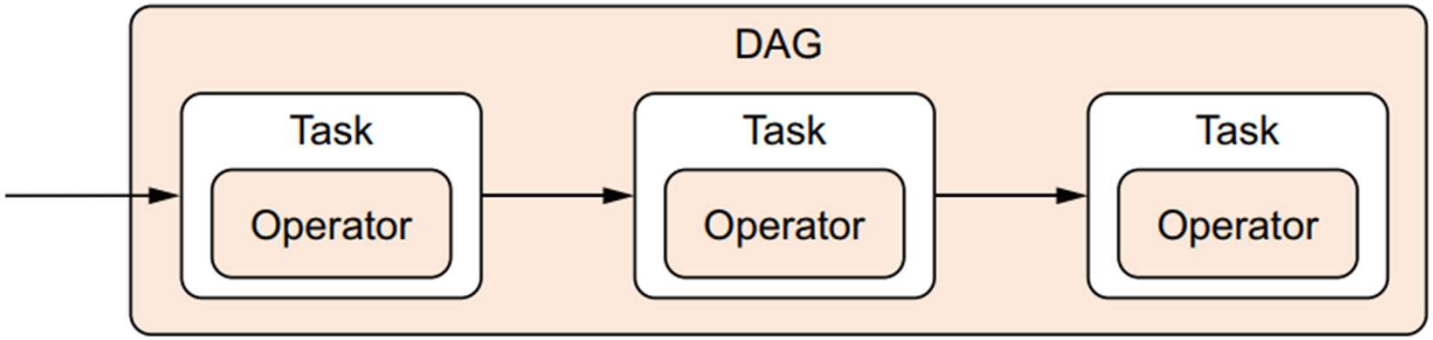
Defining the order of task execution

```
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():
    # 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}.")

get_pictures = PythonOperator(
    task_id="get_pictures",
    python_callable=_get_pictures,
    dag=dag,
)
```

Python function to call

Create pictures directory if it doesn't exist.

Open the result from the previous task.

Download each image.

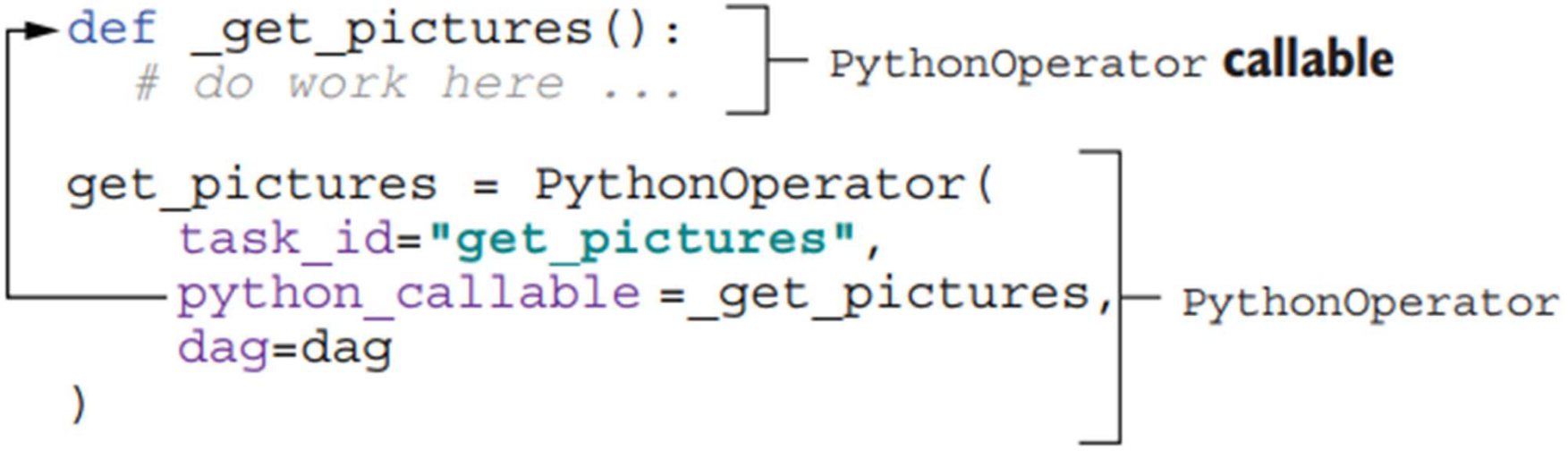
Store each image.

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

Instantiate a PythonOperator to call the Python function.

Point to the Python function to execute.

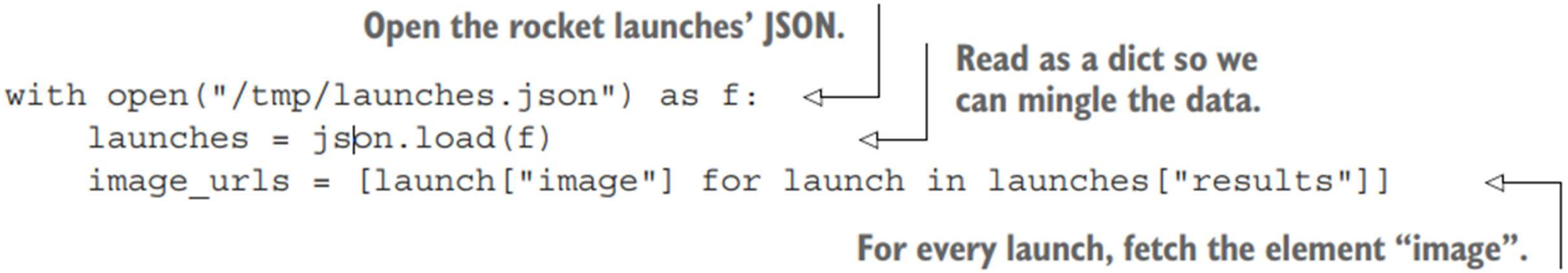
The python_callable argument in the PythonOperator points to a function to execute



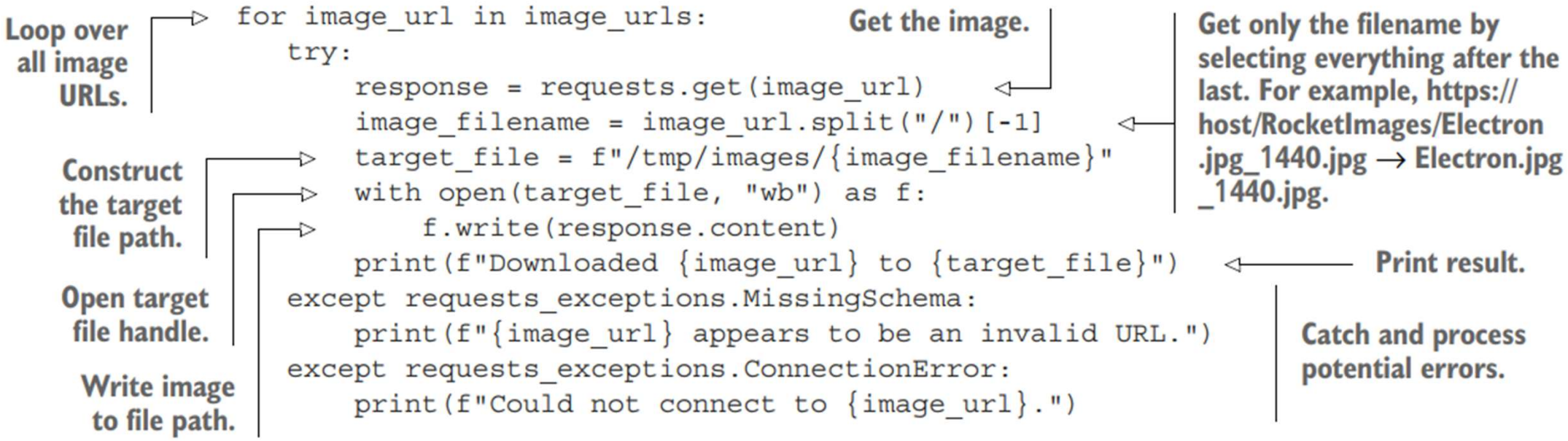
Ensures that the output directory exists and creates it if it doesn’t

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

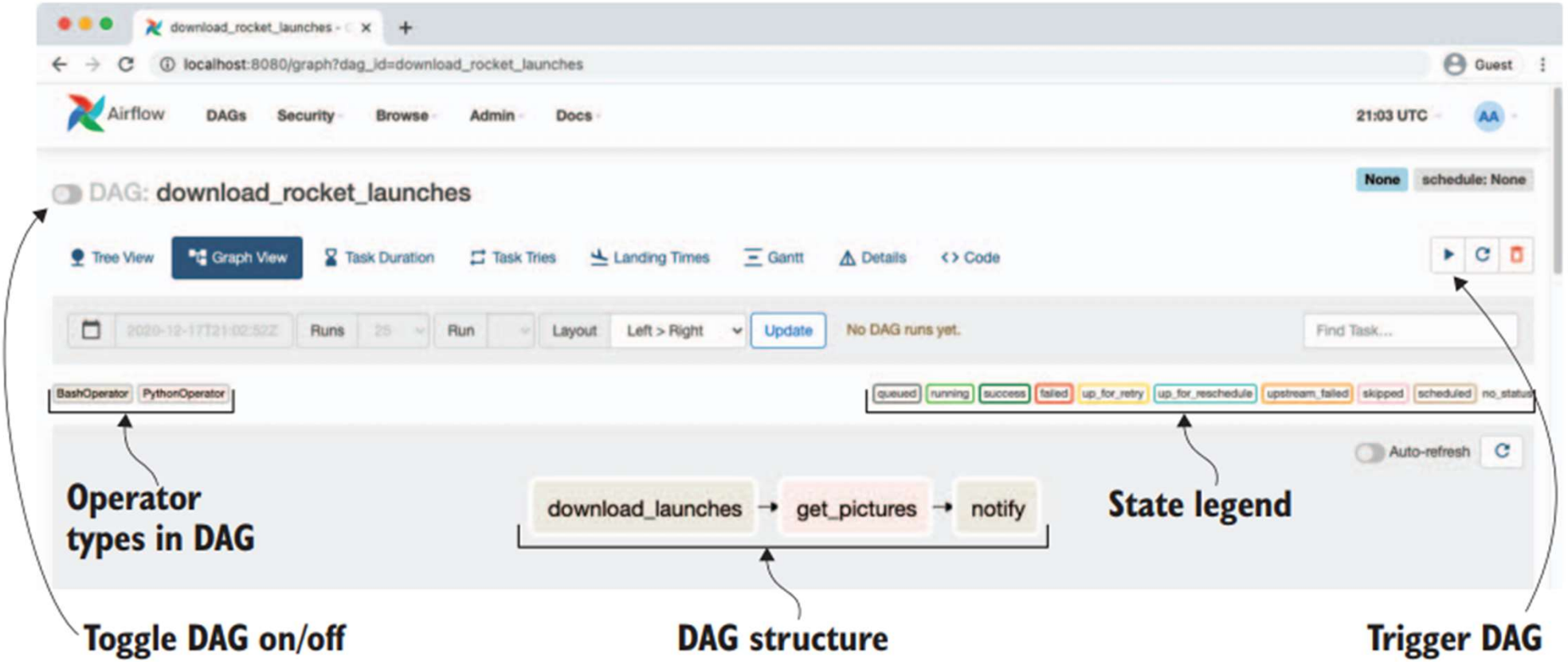
Extracts image URLs for every rocket launch



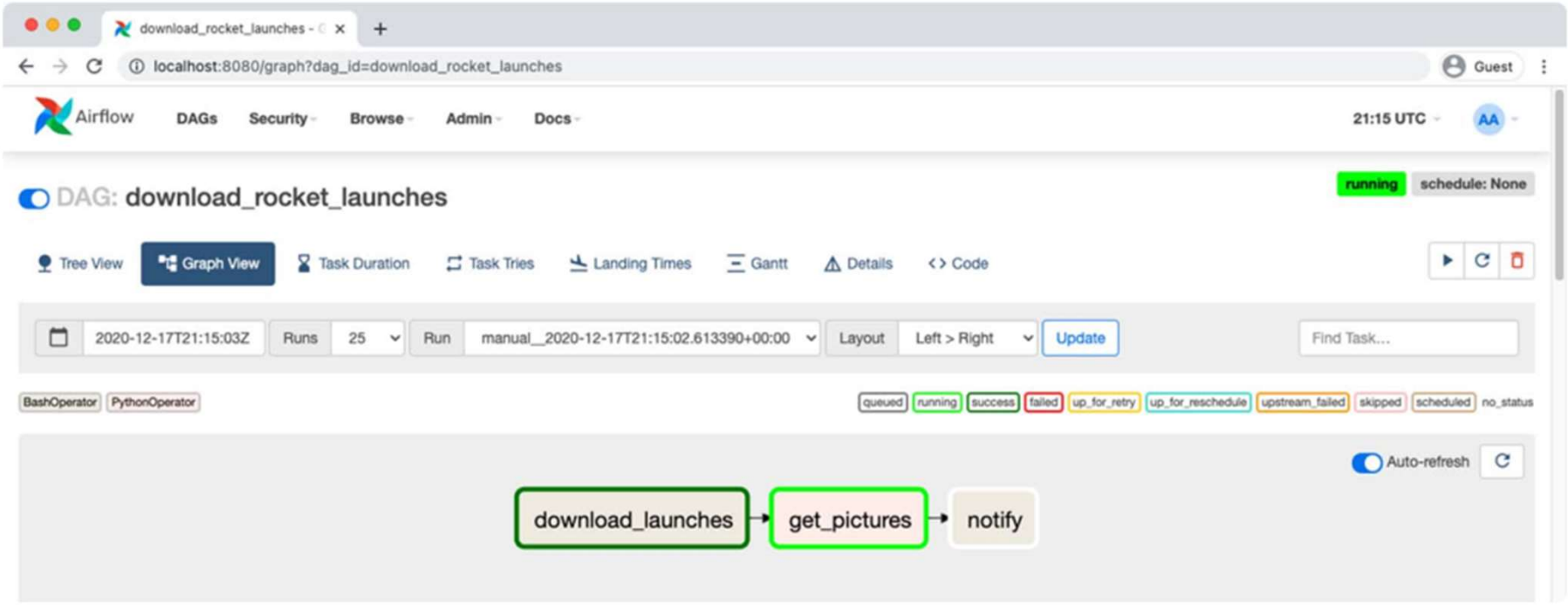
Downloads all images from the retrieved image URLs



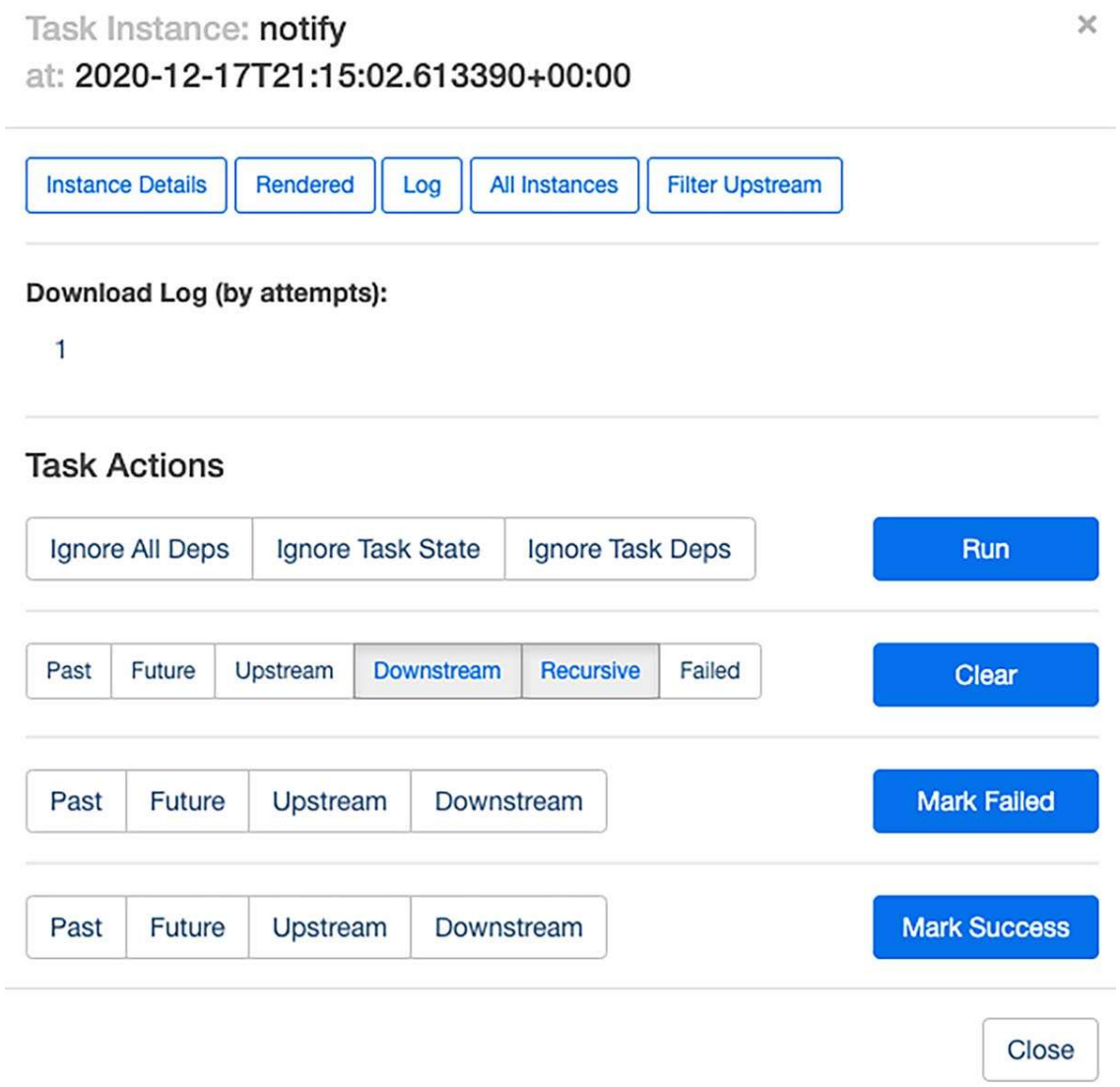
Airflow graph view



Graph view displaying a running DAG



Task pop-up options



Print statement displayed in logs

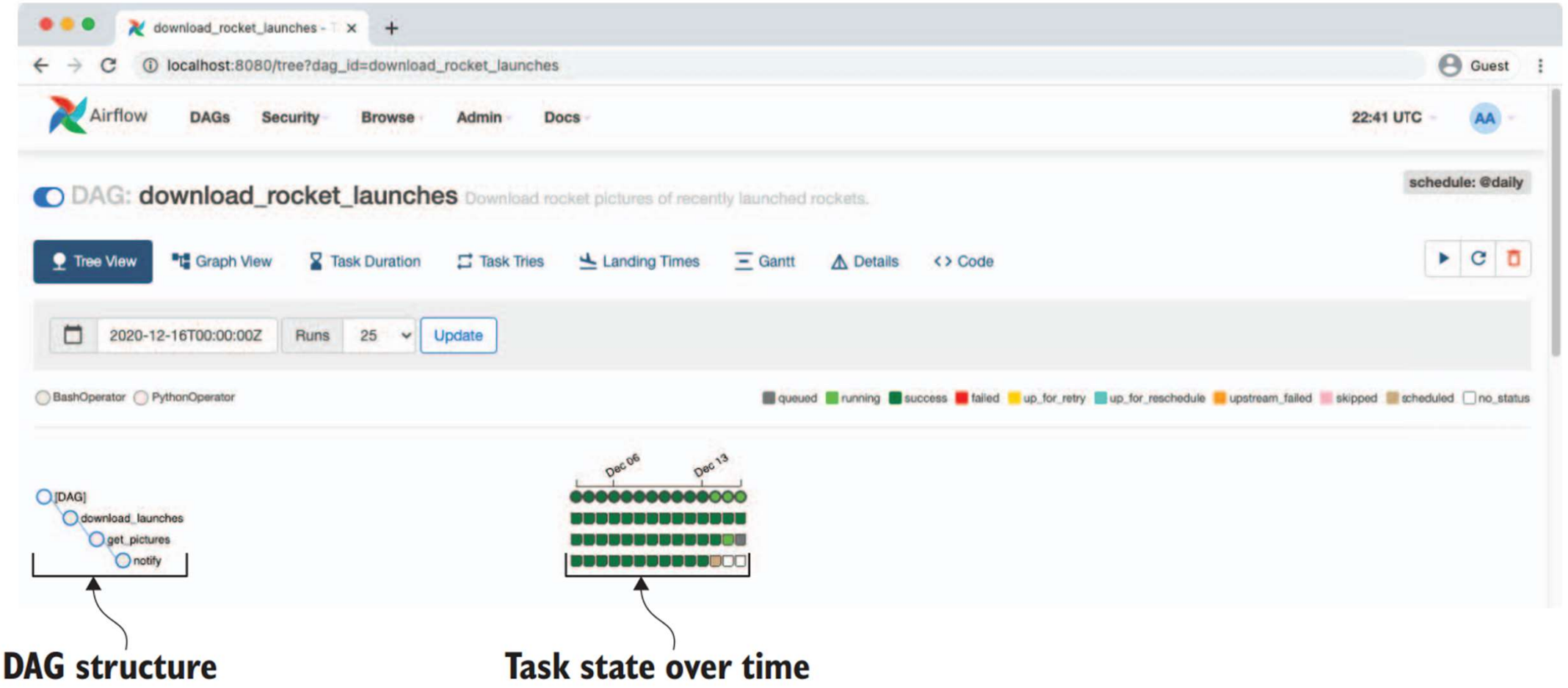
```
[2025-01-01 12:12:30,010] {local_task_job.py:118} INFO - Task exited with return code 0
[2025-01-01 12:12:30,031] {taskinstance.py:1282} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2025-01-01 12:12:30,051] {taskinstance.py:1332} INFO - Marking task as SUCCESS. dag_id=downstream_lockfile_launches, task_id=notify, execution
[2025-01-01 12:12:30,080] {base.py:111} INFO - Command exited with return code 0
[2025-01-01 12:12:30,080] {base.py:113} INFO - Three are now 5 images.
[2025-01-01 12:12:30,085] {base.py:103} INFO - Output:
[2025-01-01 12:12:30,094] {base.py:128} INFO - Running command: echo "There are now $(ls \tmp\images\ | wc -l) images."
 \tmp
[2025-01-01 12:12:30,094] {base.py:132} INFO - Tmp dir root location:
AIRFLOW_CTX_DAG_RUN_ID=manual__2025-01-01T12:05:01.013300+00:00
AIRFLOW_CTX_EXECUTION_DATE=2025-01-01T12:05:01.013300+00:00
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_DAG_ID=downstream_lockfile_launches
AIRFLOW_CTX_DAG_OWNER=airflow
[2025-01-01 12:12:30,093] {taskinstance.py:1530} INFO - Exporting the following env vars:
[2025-01-01 12:12:30,093] {logging_mixin.py:103} INFO - Running <TaskInstance: downstream_lockfile_launches.notify 2025-01-01T12:05:01.013300+00
[2025-01-01 12:12:30,038] {cfgnodegrid_task_runner.py:67} INFO - Job 8: Subtask notify
[2025-01-01 12:12:30,037] {cfgnodegrid_task_runner.py:57} INFO - Running: ['airflow', 'tasks', 'run', 'downstream_lockfile_launches', 'notify', '2
[2025-01-01 12:12:30,033] {cfgnodegrid_task_runner.py:54} INFO - Started process 1483 to run task
[2025-01-01 12:12:30,031] {taskinstance.py:1038} INFO - Executing <Task(BashOperator): notify> on 2025-01-01T12:05:01.013300+00:00
--------------------------------------------------------------------------------
[2025-01-01 12:12:30,053] {taskinstance.py:1037} INFO -
[2025-01-01 12:12:30,053] {taskinstance.py:1018} INFO - Starting attempt 1 of 1
--------------------------------------------------------------------------------
[2025-01-01 12:12:30,053] {taskinstance.py:1017} INFO -
[2025-01-01 12:12:30,053] {taskinstance.py:858} INFO - Dependencies all met for <TaskInstance: downstream_lockfile_launches.notify 2025-01-01T12:
[2025-01-01 12:12:30,017] {taskinstance.py:858} INFO - Dependencies all met for <TaskInstance: downstream_lockfile_launches.notify 2025-01-01T12:
*** Reading local file: \opt\airflow\logs\downstream_lockfile_launches\notify\2025-01-01T12:05:01.013300+00:00\1.log
```

Running a DAG once a day

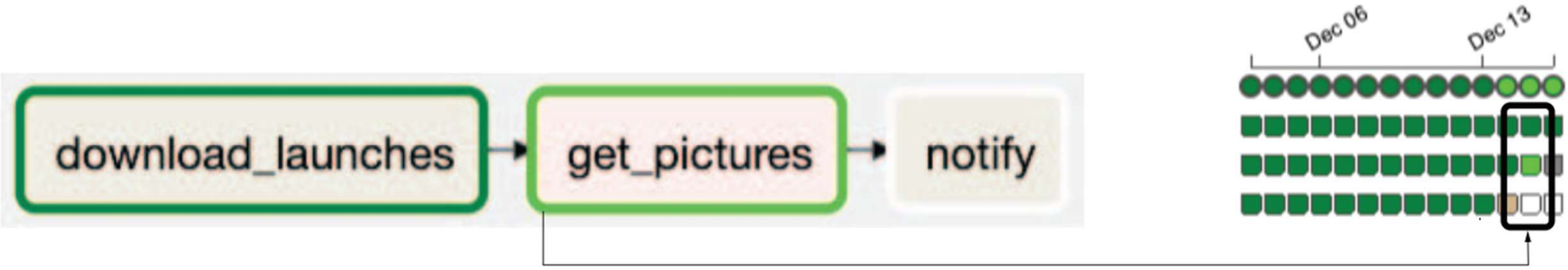
```
dag = DAG(
    dag_id="download_rocket_launches",
    start_date=airflow.utils.dates.days_ago(14),
    schedule_interval="@daily",
)
```

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

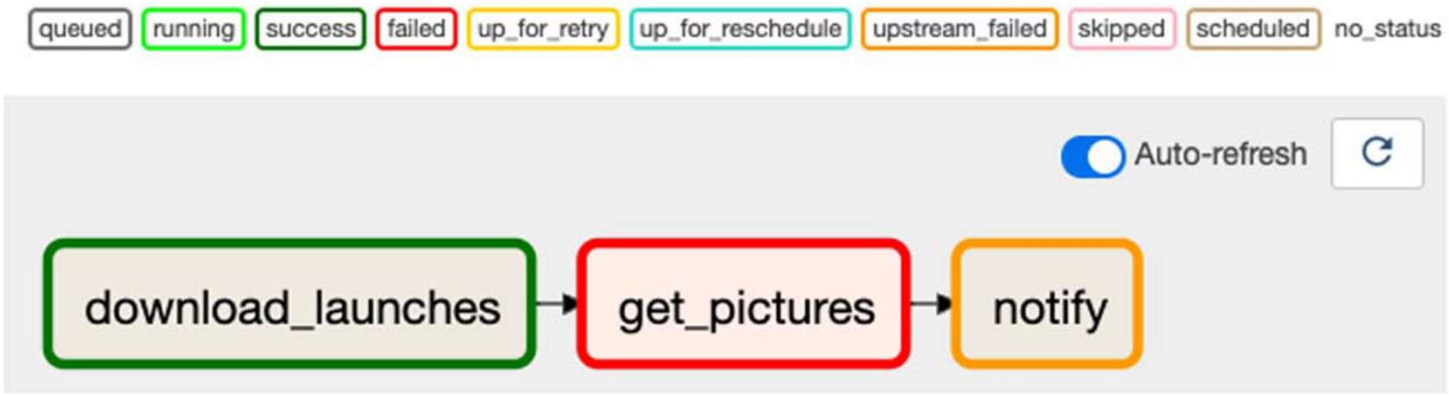
Airflow tree view

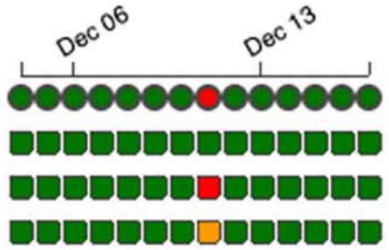


Relationship between graph view and tree view

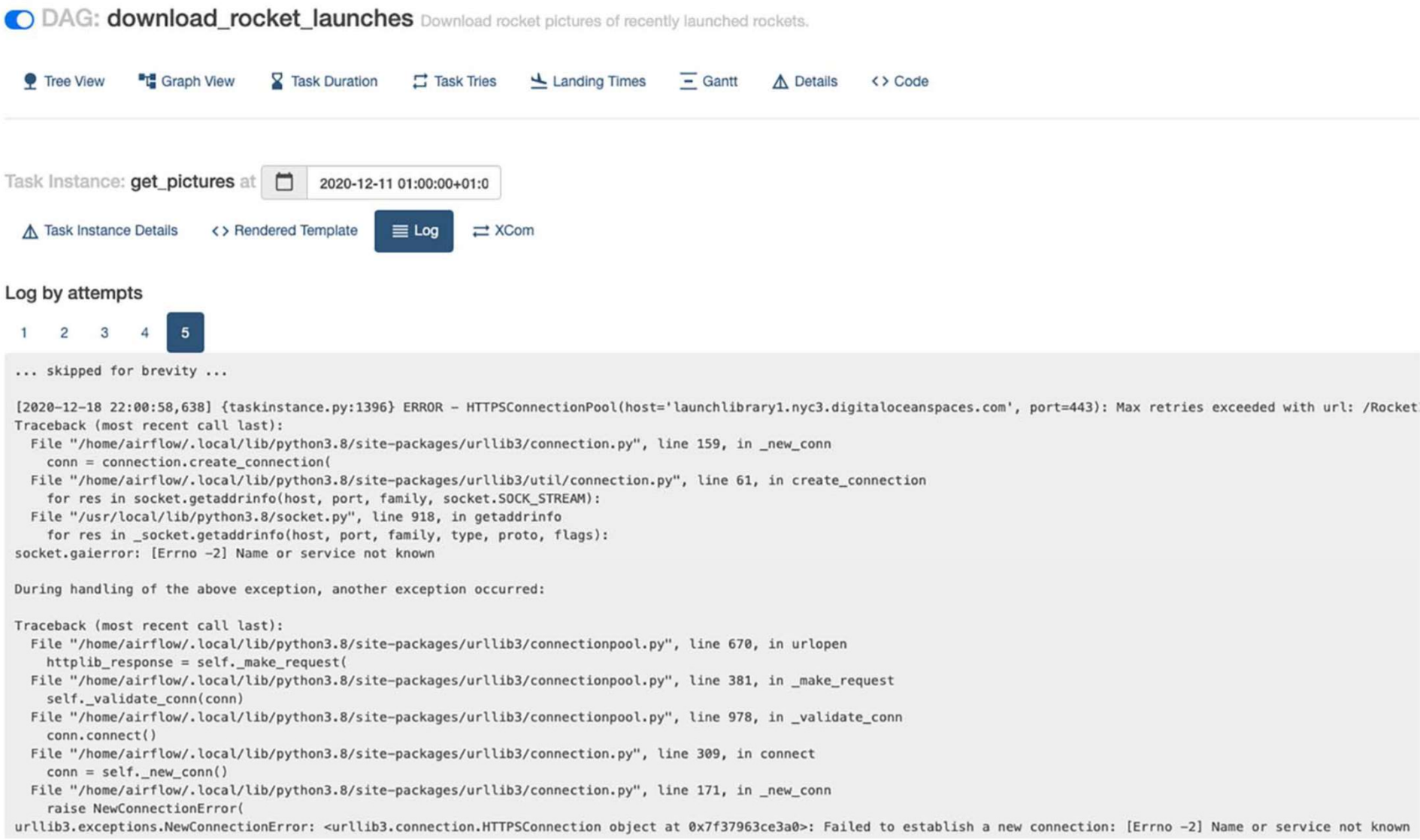


Failure displayed in graph view and tree view

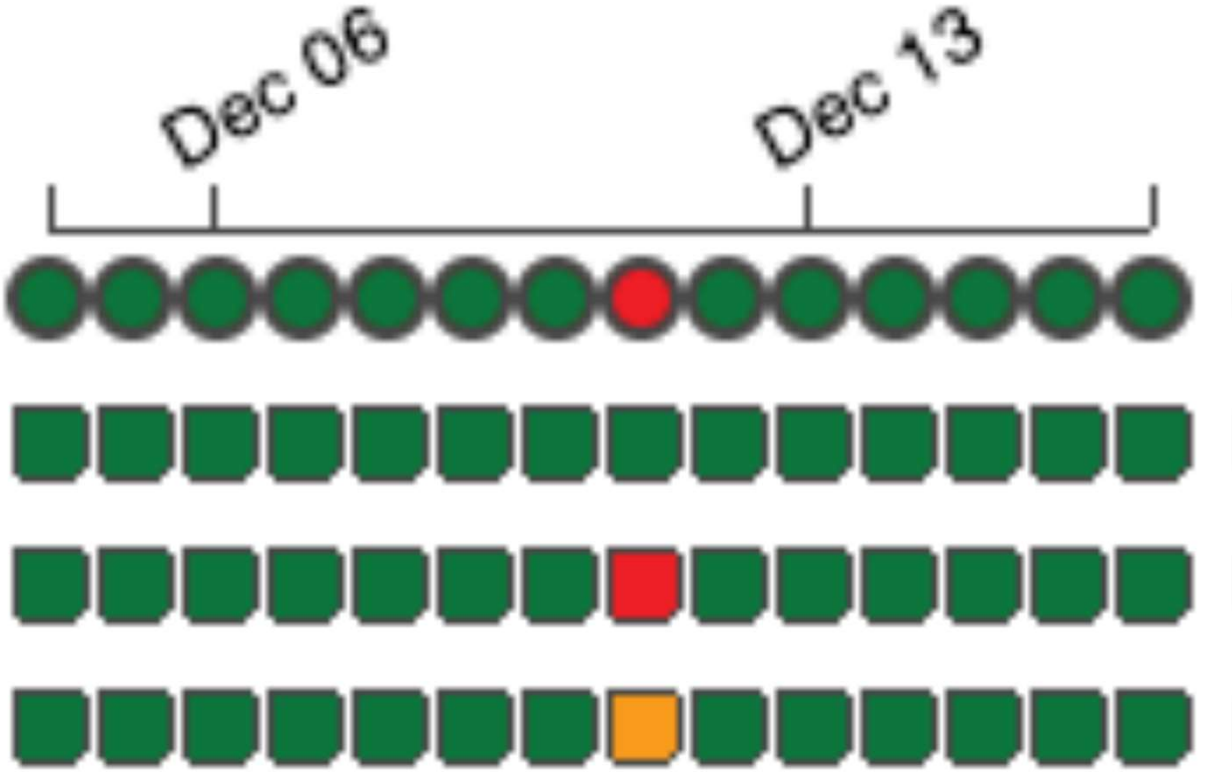




Stack trace of failed get_pictures task



Click on a failed task for options to clear it



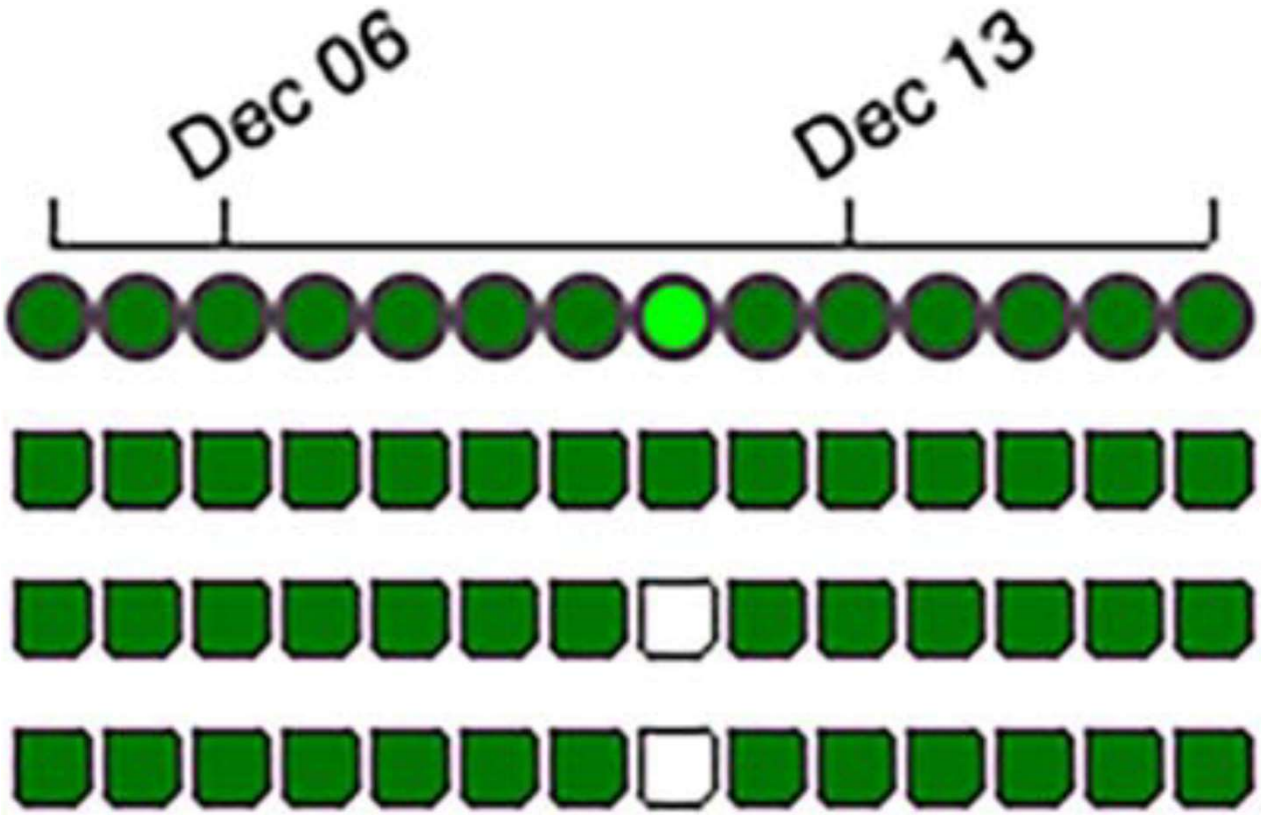
Clearing the state of get_pictures and successive tasks

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

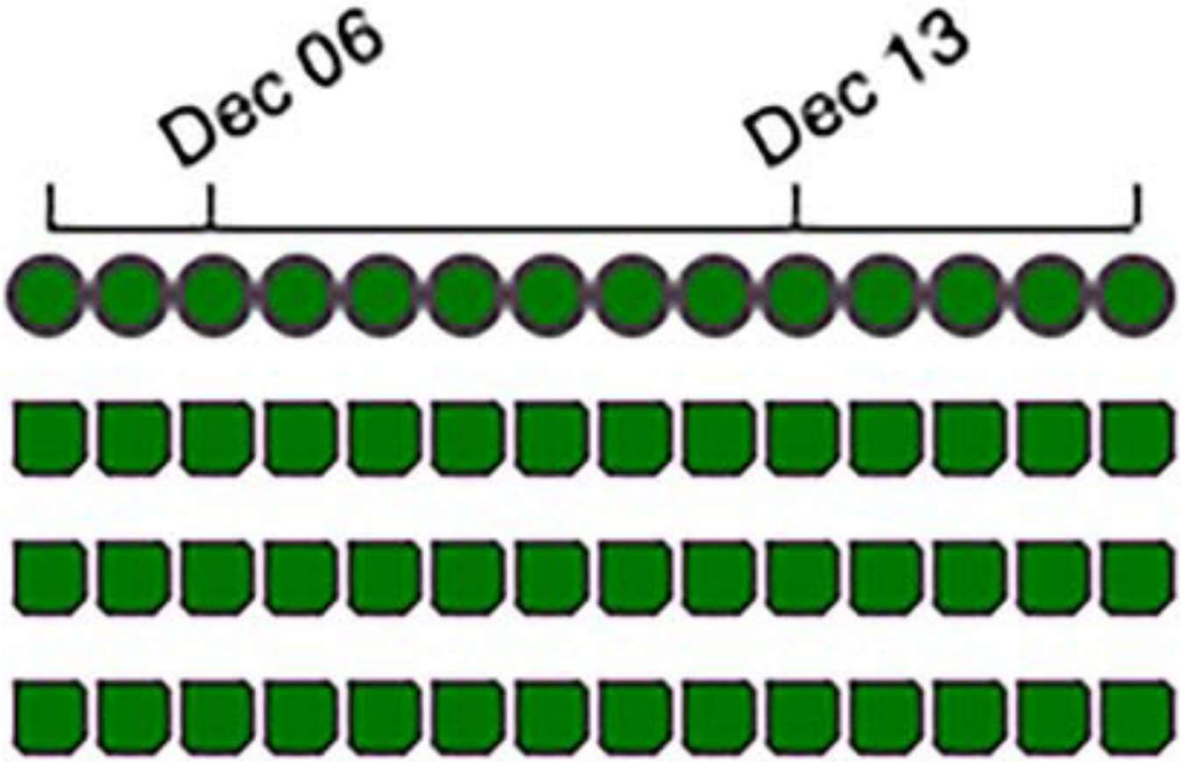
```
<TaskInstance: download_rocket_launches.get_pictures 2020-12-11 00:00:00+00:00 [failed]>
<TaskInstance: download_rocket_launches.notify 2020-12-11 00:00:00+00:00 [upstream_failed]>
```

OK! Cancel

Cleared tasks displayed in graph view



Successfully completed tasks after clearing failed tasks



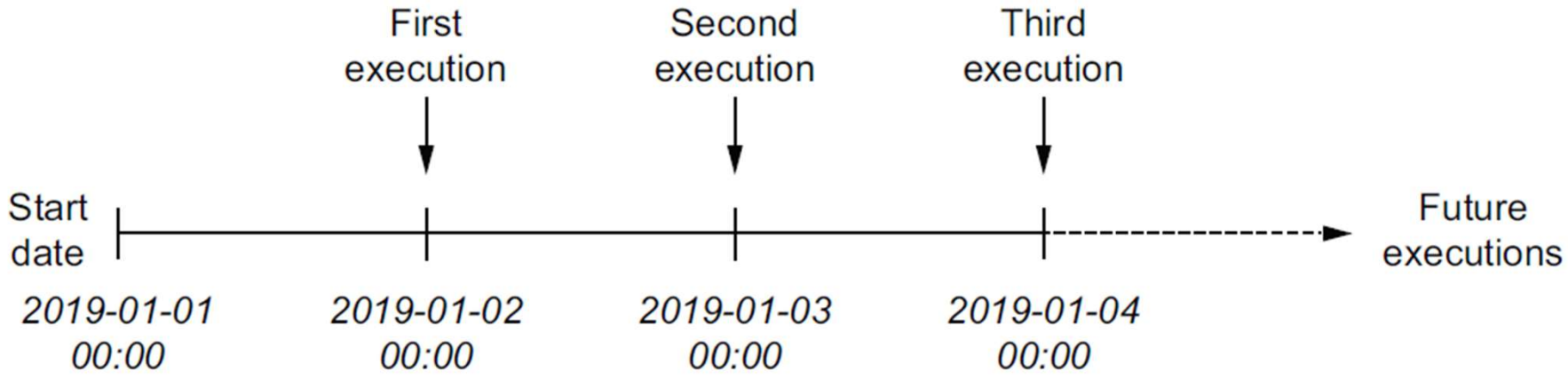
Scheduling in Airflow

Defining a daily schedule interval

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

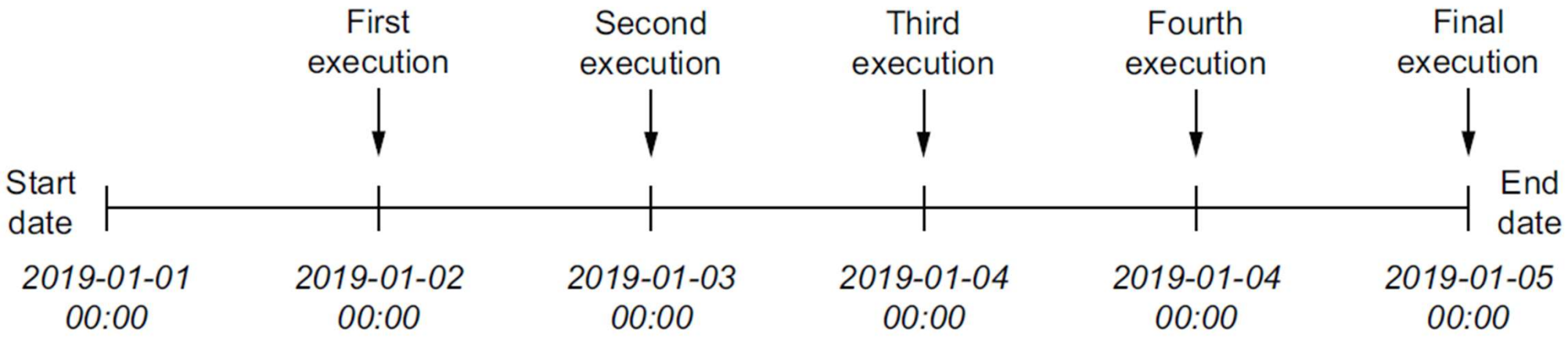
Schedule the DAG to run every day at midnight.

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

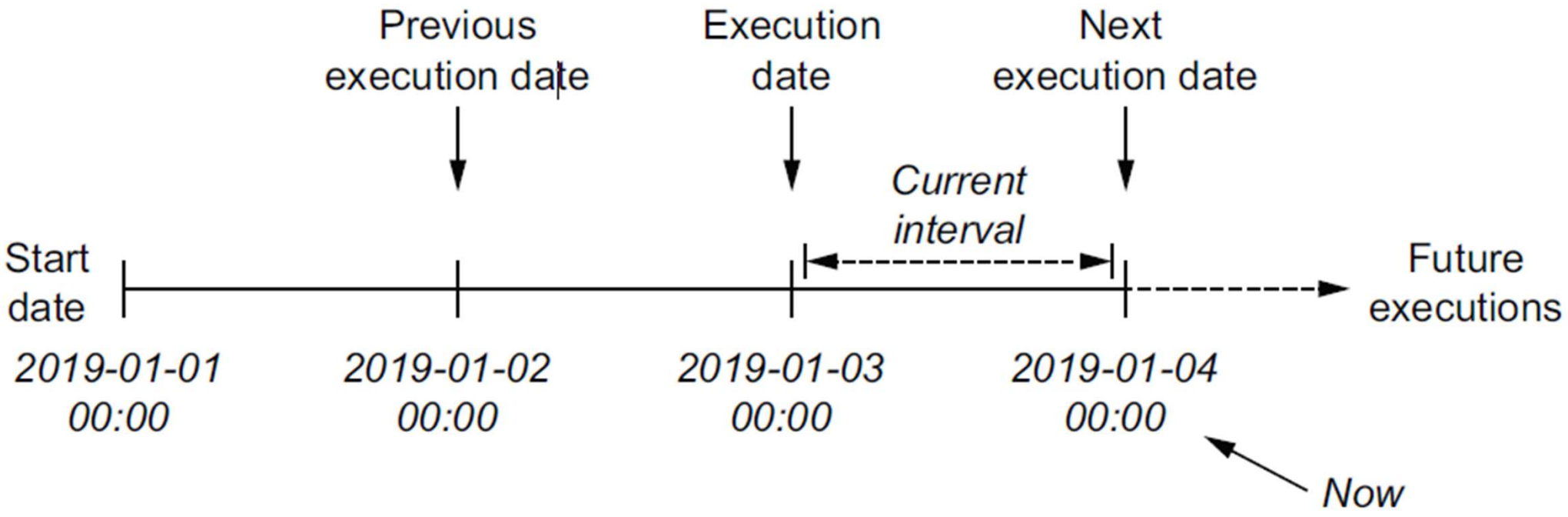
Preset	Meaning
`@once`	Schedule once and only once.
`@hourly`	Run once an hour at the beginning of the hour.
`@daily`	Run once a day at midnight.
`@weekly`	Run once a week at midnight on Sunday morning.
`@monthly`	Run once a month at midnight on the first day of the month.
`@yearly`	Run once a year at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

Defining a frequency-based schedule interval

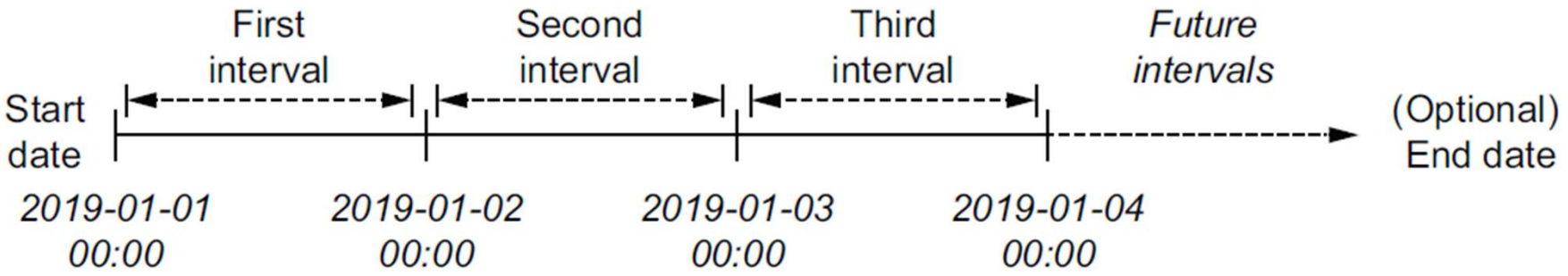
```
dag = DAG(
    dag_id="04_time_delta",
    schedule_interval=dt.timedelta(days=3),
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=1, day=5),
)
```

timedelta gives the ability to use frequency-based schedules.

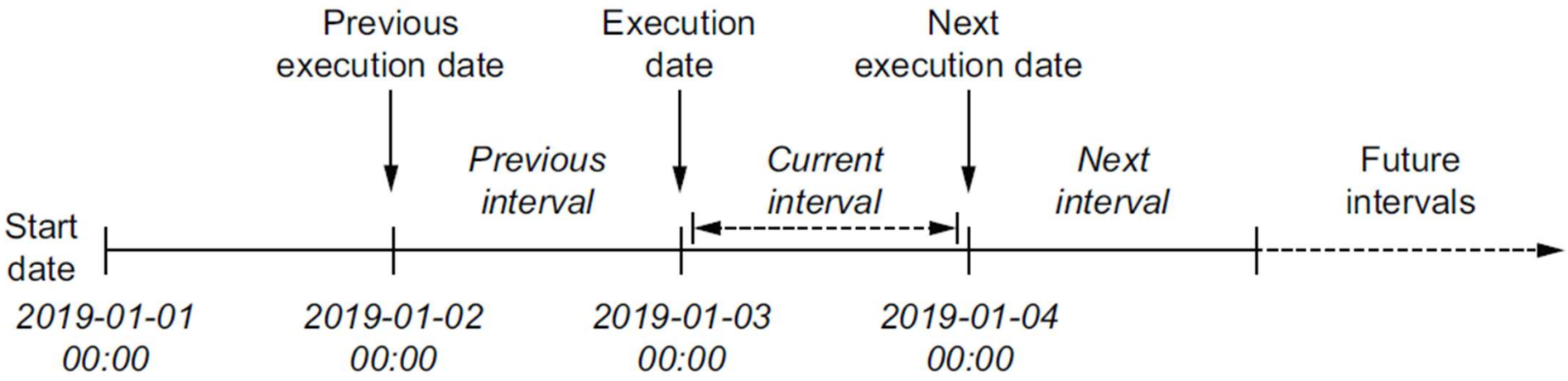
Dynamic time references using execution dates



Understanding Airflow's execution dates



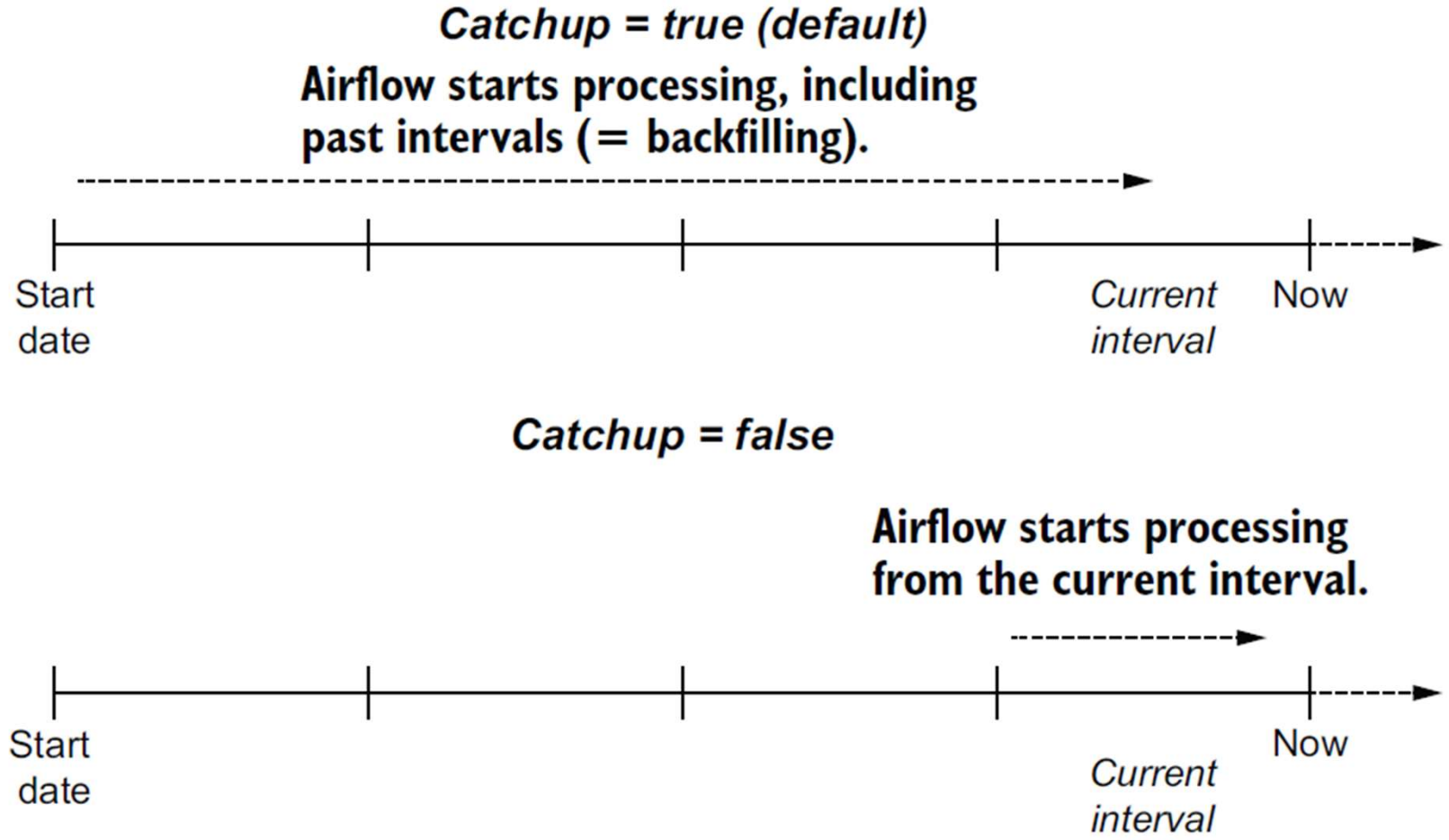
Execution Date insights



Using backfilling to fill in past gaps
- Disabling catchup to avoid running past runs

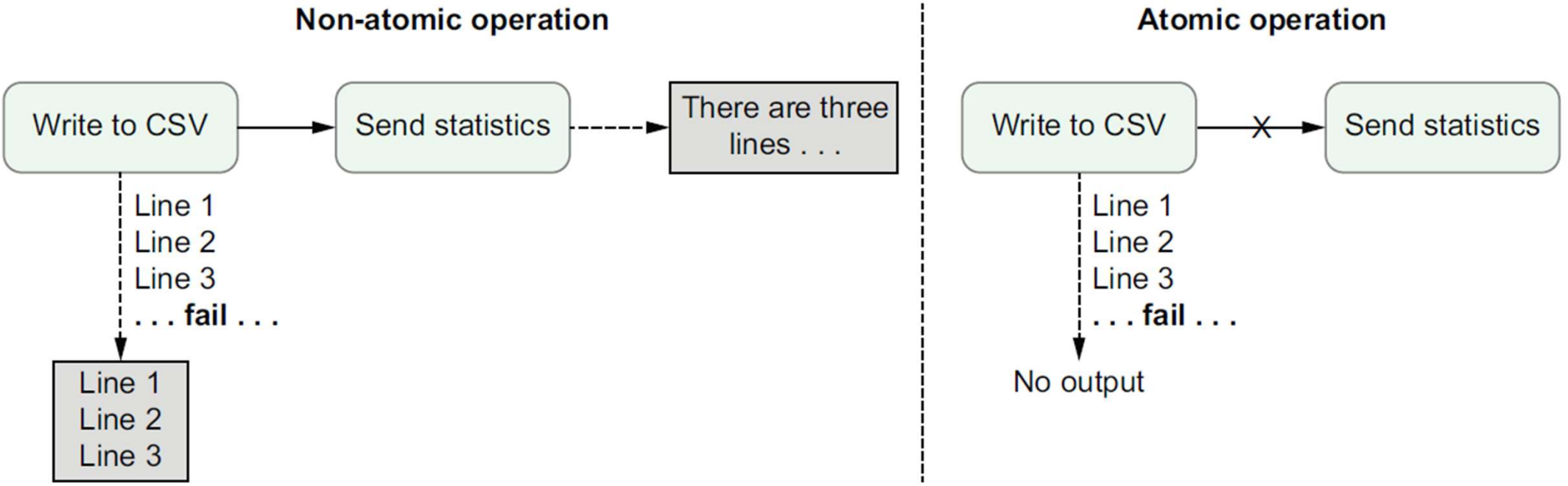
```
dag = DAG(
    dag_id="09_no_catchup",
    schedule_interval="@daily",
    start_date=dt.datetime(year=2019, month=1, day=1),
    end_date=dt.datetime(year=2019, month=1, day=5),
    catchup=False,
)
```

Backfilling in Airflow



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

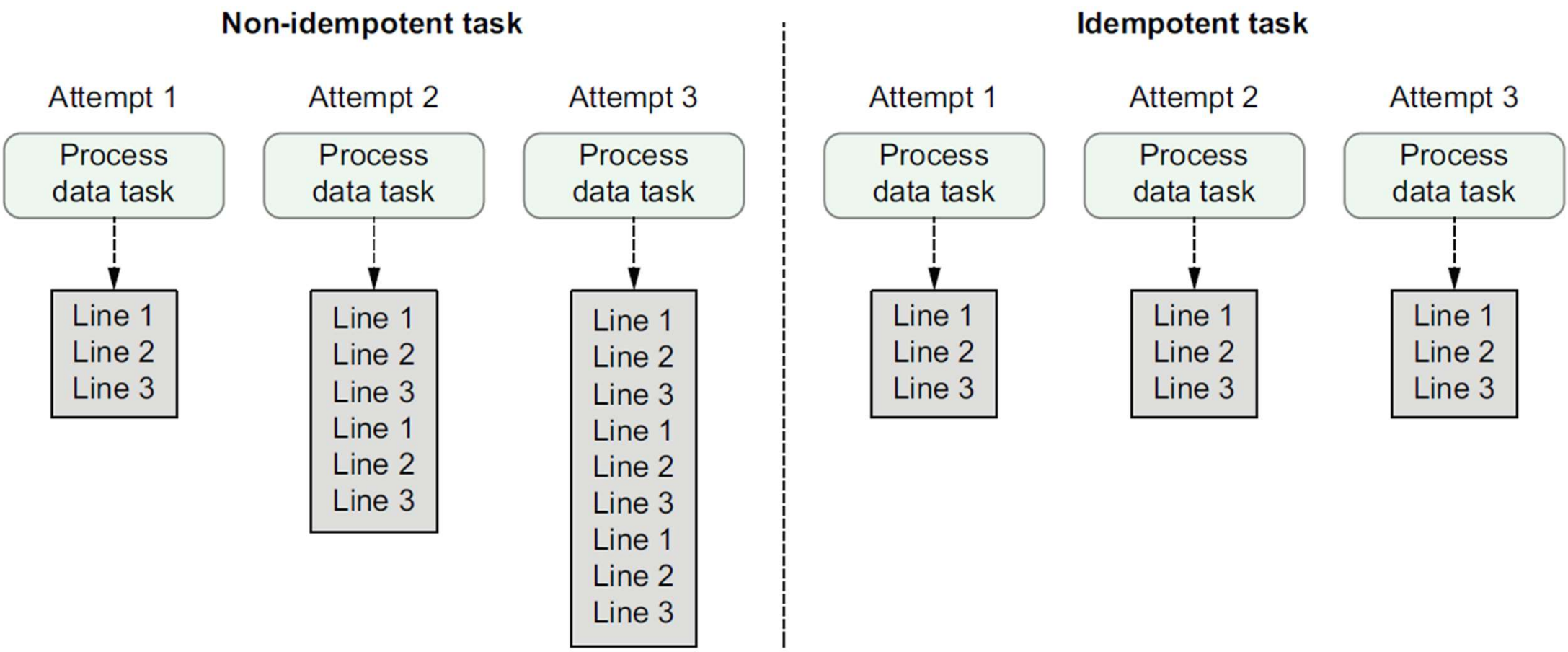
```
def _send_stats(email, **context):
    stats = pd.read_csv(context["templates_dict"]["stats_path"])
    email_stats(stats, email=email)


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