Apache Airflow in Practice

Mingjun Zhong

Department of Computing Science

University of Aberdeen

Content

- Running Airflow on your own machine
- Writing and running your first workflow
- Examining the first view at the Airflow interface
- Handling failed tasks in Airflow

Rocket launch data repository

- The repository tracks every rocket launch, including upcoming ones
- Launch Library 2: https://thespacedevs.com/llapi
- Free and open API
- We look at upcoming rocket launches: https://ll.thespacedevs.com/2.0.0/launch/upcoming/
- Data is represented as Json files

Rocket launch data

```
$ curl -L "https://ll.thespacedevs.com/2.0.0/launch/upcoming"

{
  ...
  "results": [
    {
      "id": "528b72ff-e47e-46a3-b7ad-23b2ffcec2f2",
      "url": "https://.../528b72ff-e47e-46a3-b7ad-23b2ffcec2f2/",
      "launch_library_id": 2103,
      "name": "Falcon 9 Block 5 | NROL-108",
      "net": "2020-12-19T14:00:00Z",
      "window_end": "2020-12-19T17:00:00Z",
      "window_start": "2020-12-19T14:00:00Z",
      "image": "https://spacelaunchnow-prod-
      east.nyc3.digitaloceanspaces.com/media/launch_images/falcon2520925_image
      _20201217060406.jpeg",
      "infographic": ".../falcon2520925_infographic_20201217162942.png",
      ...
    },
    {
      "id": "57c418cc-97ae-4d8e-b806-bb0e0345217f",
      "url": "https://.../57c418cc-97ae-4d8e-b806-bb0e0345217f/",
      "launch_library_id": null,
      "name": "Long March 8   | XJY-7 & others",
      "net": "2020-12-22T04:29:00Z",
      "window_end": "2020-12-22T05:03:00Z",
```

The response is a JSON document, as you can see by the structure.

Inspect the URL response with curl from the command line.

The square brackets indicate a list.

All values within these curly braces refer to one single rocket launch.

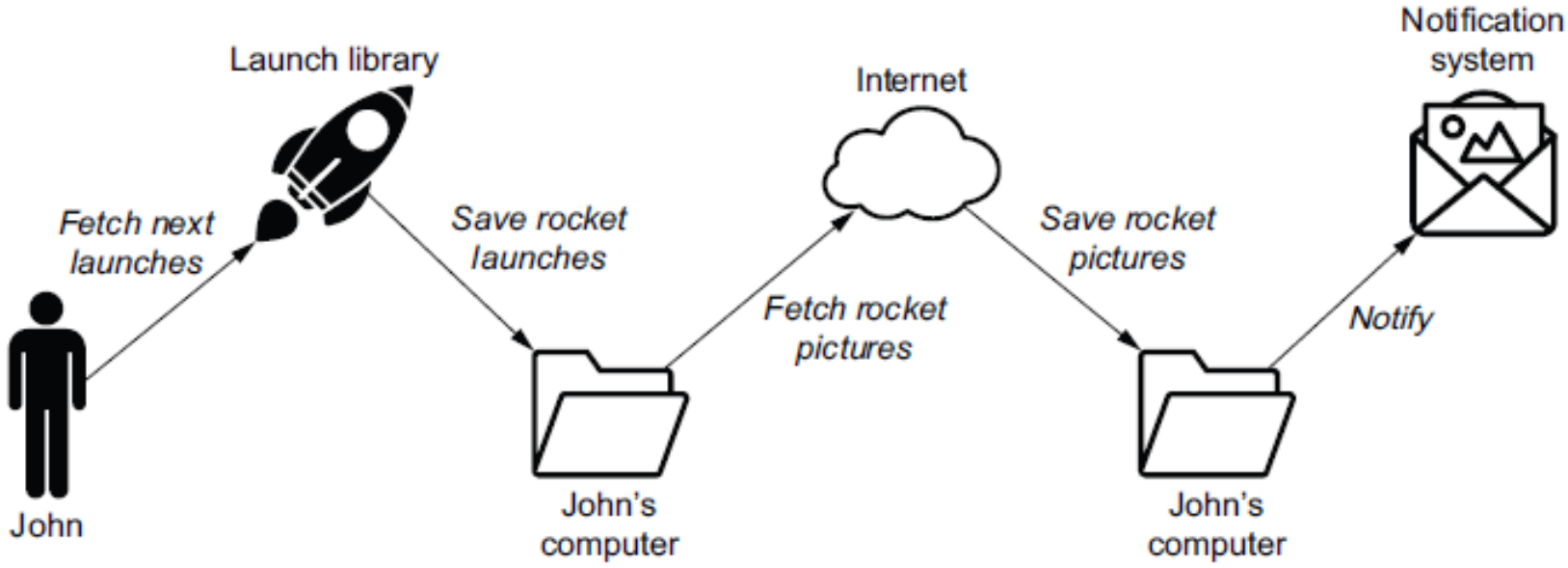
Here we see information such as rocket ID and start and end time of the rocket launch window.

A URL to an image of the launching rocket

Rocket launch

- Data in Json format; rocket launch info: ID, name, image URL
- We want to read, transform, and output the data (ETL)
- Only extract the images in this example

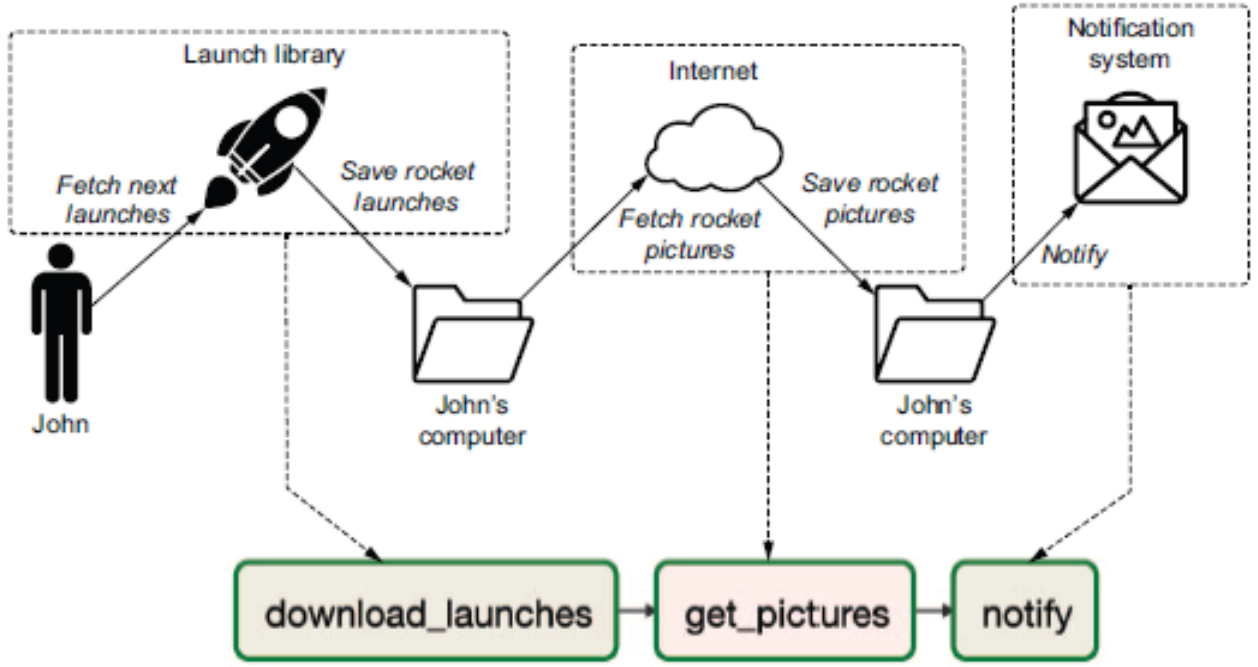
The data engineering model



The Airflow DAG

- The ETL model is defined.
- We will see how Airflow can split the DE model (assume a large job) into small tasks (assume small jobs)
- Tasks could be done in parallel and can run with different technologies

Mapping DE/ETL model into Airflow DAG



The Airflow DAG

- Now we go through the Python code to define the DAG for implementing the pipeline

Import the packages required

```
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
```

The Airflow DAG

- Instantiate a DAG object
- The starting point of any workflow
- All tasks within the workflow reference this DAG object so that Airflow knows which tasks belong to which DAG

```
BASE_PATH = "/opt/airflow/dags"  # Use a shared, Docker-visible path which
can be found in docker-compose.yaml

dag = DAG(
    dag_id = "rocket_launch", # Same to the filename; the name of DAG
    start_date=airflow.utils.dates.days_ago(14), # the date at which the DAG
should first start running

    schedule_interval=None, # at what interval the DAG should run; this is
None, so Airflow won't run automatically

)
```

The Airflow DAG

- The first task: apply bash to download the URL response with curl
- BashOperator is to run a Bash command
- Each operator performs only a single unit of work

```
download_launches = BashOperator(
    task_id="download_launches", # the name of task
    bash_command="curl -o /opt/airflow/dags/launches.json -L
'https://ll.thespacedevs.com/2.0.0/launch/upcoming'",
    dag=dag,
)
```

The Airflow DAG

- A Python function to parse response and download the rocket pictures from the URL link

```
def _get_pictures():
    # Ensure directory exists
    pathlib.Path("{BASE_PATH}/images").mkdir(parents=True, exist_ok=True)
    # Download all pictures in launches.json
    with open("/opt/airflow/dags/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"{BASE_PATH}/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}.")
```

The Airflow DAG

- The second task: call the Python function in the DAG with a PythonOperator

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

The Airflow DAG

- The third task: Using a BashOperator to notify the results

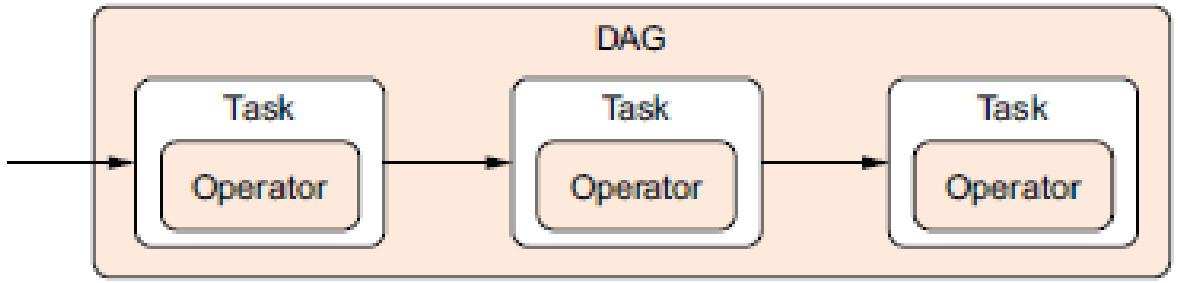
```
notify = BashOperator(
    task_id="notify",
    bash_command='echo "There are now $(ls {BASE_PATH}/images/ |
wc -l) images."',
    dag=dag,
)
```

The Airflow DAG

- Finally, execute the tasks in order
- “>>” defines the dependencies between tasks
- the latter can only be executed after the former is completed

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

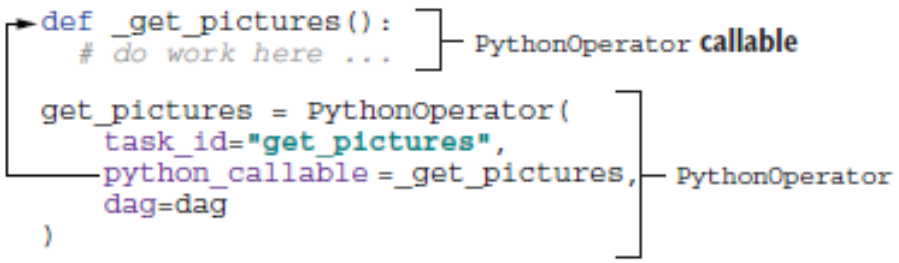
Tasks vs operators



- *Task* and *operator* are used interchangeably
- They both execute a piece of code
- As the user, we can view them as the same thing
- Airflow operators:
  - BashOperator – to run Bash scripts
  - PythonOperator – to run a Python function
  - EmailOperator – to send an email
  - SimpleHTTPOperator – to call an HTTP endpoint
- Task can be viewed as a wrapper around an operator that ensures the operator executes correctly
- Use focus on the work to be done by using operators; Airflow ensures correct execution of the work via tasks

Python Operators

- We define the operator itself
- The *python_callable* argument points to a callable, typically a function



Running a DAG in Airflow

- We will need to run the DAG and view it in the Airflow UI
- Two ways to get Airflow up and run it:
  - Running Airflow in a Python environment
    - You have to manage all the packages by yourself
    - Isolated Python environment using *Conda, pyenv, virtualenv*
  - Running Airflow in Docker containers
    - Docker containers provide isolated environment for Airflow
    - Isolated environment on the operating system level

- We demonstrate Airflow using a **Docker** container
- We will see the details in practical on how to install Docker and run Airflow

Running Airflow in Docker container

- Reference: https://airflow.apache.org/docs/apache-airflow/2.0.2/start/docker.html

1. Install Docker desktop on Windows:
   a. https://docs.docker.com/desktop/setup/install/windows-install/
   b. Download the installer
   c. Install docker by clicking .exe file

2. Deploy Apache Airflow on Docker Compose

   a.Download the docker-compose.yaml file:

https://airflow.apache.org/docs/apache-airflow/2.10.4/docker-compose.yaml

   b.Under the project folder: airflow, create the following directories:

   i)./dags – you can put your DAG files here

   ii)./logs – contains logs from task execution and scheduler

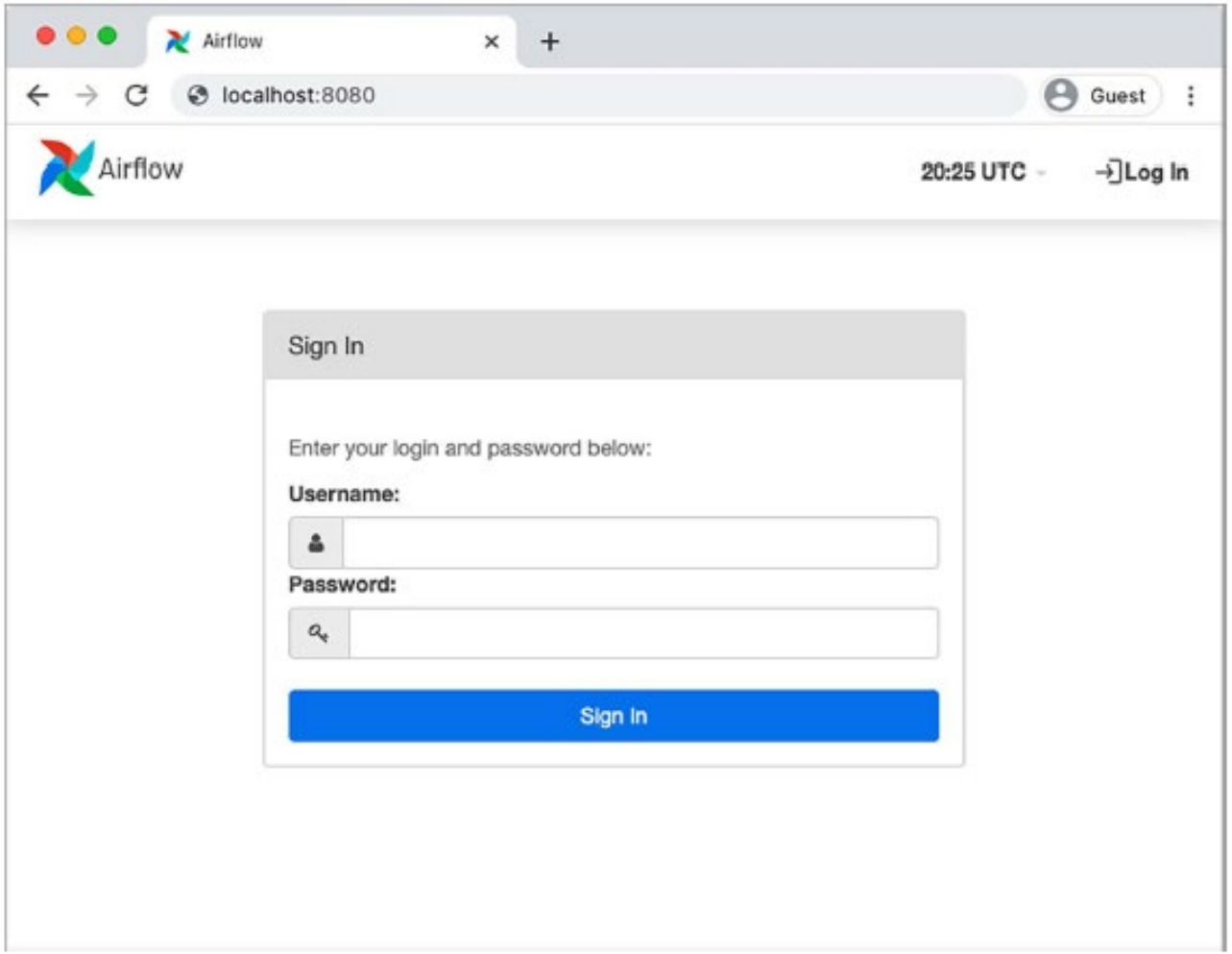
   iii)./plugins – you can put your custom plugins here

   iv)Move the *docker-compose.yaml* file to this folder

3. On all operating systems, you need to run database migrations and create the first user account. To do it, run: *docker-compose up airflow-init*
4. Running Airflow: Now you can start all services: ``docker-compose up''

Airflow UI

- Airflow UI is accessed by http://localhost:8080
- Username: airflow
- Password: airflow

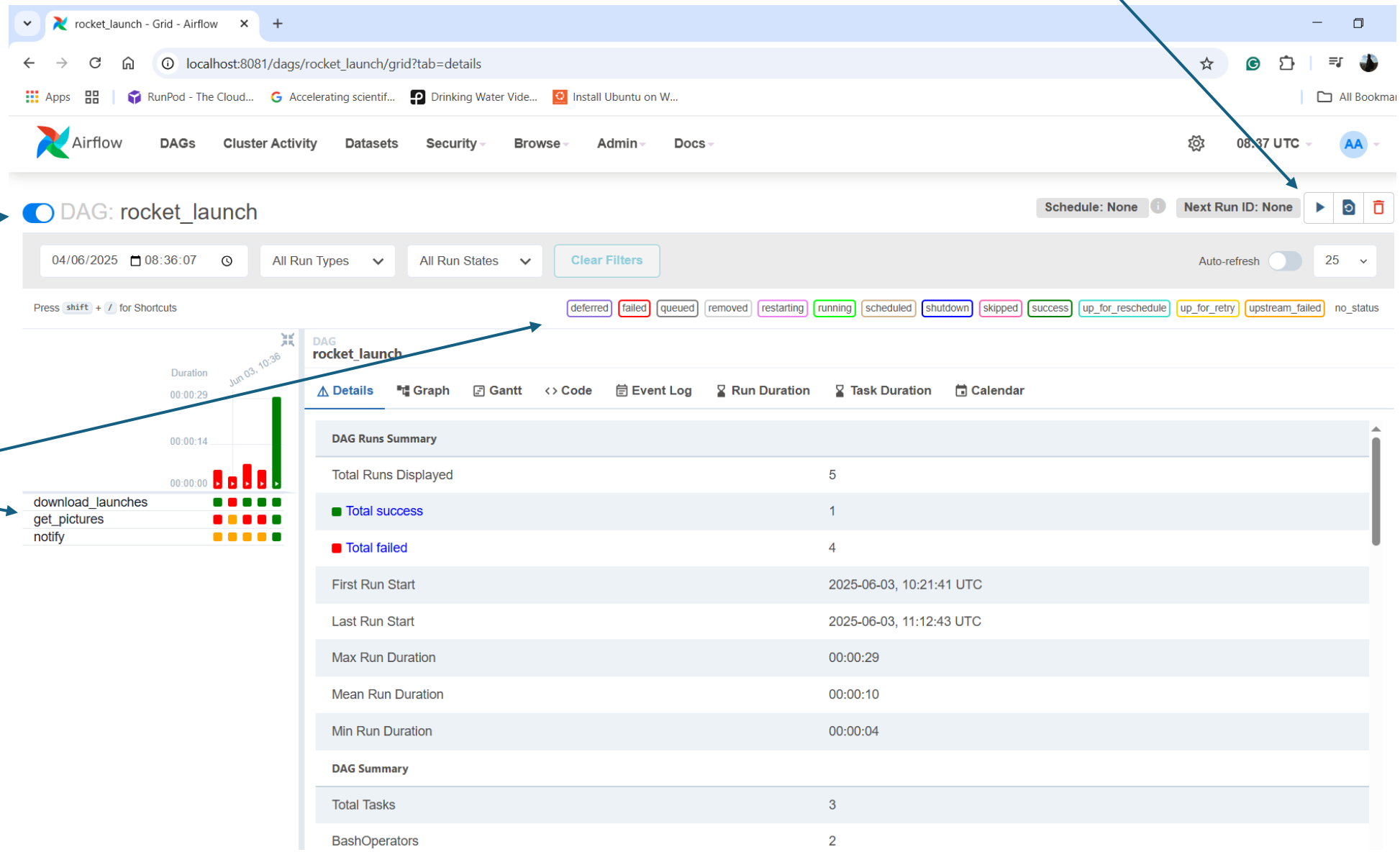


Airflow UI

Click to run the DAG

DAG

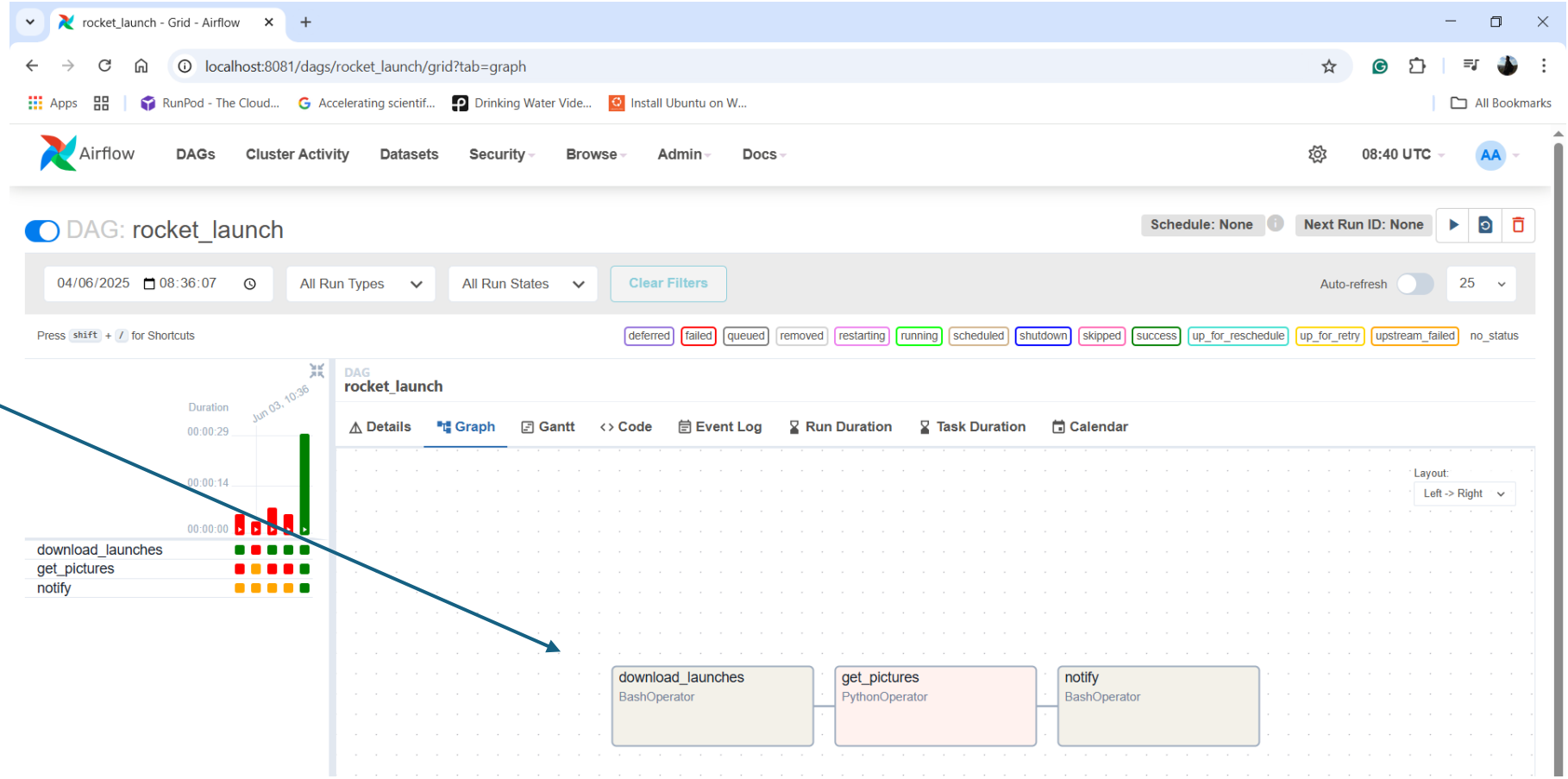
Tasks:
failed and success



Airflow UI

DAG

Tasks:
failed and success



Running at regular intervals

- We schedule a DAG to run at certain intervals: once an hour, day, or month
- This is executed by setting the argument in schedule_interval: e.g., schedule_interval = “@daily”
- Airflow will run the DAG once a day; you don’t need to trigger it manually

Handling failing tasks

- It’s often to have failed tasks
- The failed tasks are denoted by the red dots and seen under the details
- To fix the issues:
  - Inspect the log files under the directory /logs
  - Find the dag_id=rocket_launch
  - Then find the run_id=…
  - Open the log file and trace the errors
- You can also use the following bash command to see errors:
  - *docker-compose run airflow-worker airflow dags show rocket_launch*

Handling failing tasks

Error when running

attempt=1.log - Notepad

File Edit Format View Help

```
[2025-06-03T10:36:25.264+0000] {local_task_job_runner.py:123} INFO - ::group::Pre task execution logs
[2025-06-03T10:36:25.275+0000] {taskinstance.py:2613} INFO - Dependencies all met for dep_context=non-requeueable deps ti=<TaskInstance: rocket_launch.download_launches manual__2025-06-03T1
[2025-06-03T10:36:25.280+0000] {taskinstance.py:2613} INFO - Dependencies all met for dep_context=requeueable deps ti=<TaskInstance: rocket_launch.download_launches manual__2025-06-03T10:36
[2025-06-03T10:36:25.280+0000] {taskinstance.py:2866} INFO - Starting attempt 1 of 1
[2025-06-03T10:36:25.288+0000] {taskinstance.py:2889} INFO - Executing <Task(BashOperator): download_launches> on 2025-06-03 10:36:23.484540+00:00
[2025-06-03T10:36:25.293+0000] {warnings.py:112} WARNING - /home/***/.local/lib/python3.12/site-packages/***/task/task_runner/standard_task_runner.py:70: DeprecationWarning: This process (p
  pid = os.fork()

[2025-06-03T10:36:25.294+0000] {standard_task_runner.py:72} INFO - Started process 88 to run task
[2025-06-03T10:36:25.294+0000] {standard_task_runner.py:104} INFO - Running: ['***', 'tasks', 'run', 'rocket_launch', 'download_launches', 'manual__2025-06-03T10:36:23.484540+00:00', '--job
[2025-06-03T10:36:25.295+0000] {standard_task_runner.py:105} INFO - Job 24: Subtask download_launches
[2025-06-03T10:36:25.468+0000] {task_command.py:467} INFO - Running <TaskInstance: rocket_launch.download_launches manual__2025-06-03T10:36:23.484540+00:00 [running]> on host a6ad22ae3868
[2025-06-03T10:36:25.516+0000] {taskinstance.py:3132} INFO - Exporting env vars: AIRFLOW_CTX_DAG_OWNER='***' AIRFLOW_CTX_DAG_ID='rocket_launch' AIRFLOW_CTX_TASK_ID='download_launches' AIRFL
[2025-06-03T10:36:25.517+0000] {logging_mixin.py:190} INFO - Task instance is in running state
[2025-06-03T10:36:25.518+0000] {logging_mixin.py:190} INFO -  Previous state of the Task instance: queued
[2025-06-03T10:36:25.518+0000] {logging_mixin.py:190} INFO - Current task name:download_launches state:running start_date:2025-06-03 10:36:25.276328+00:00
[2025-06-03T10:36:25.518+0000] {logging_mixin.py:190} INFO - Dag name:rocket_launch and current dag run status:running
[2025-06-03T10:36:25.519+0000] {taskinstance.py:731} INFO - ::endgroup::
[2025-06-03T10:36:25.519+0000] {subprocess.py:78} INFO - Tmp dir root location: /tmp
[2025-06-03T10:36:25.520+0000] {subprocess.py:88} INFO - Running command: ['/usr/bin/bash', '-c', "curl -o {BASE_PATH}/launches.json -L 'https://ll.thespacedevs.com/2.0.0/launch/upcoming'"]
[2025-06-03T10:36:25.527+0000] {subprocess.py:99} INFO - Output:
[2025-06-03T10:36:25.533+0000] {subprocess.py:106} INFO -   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
[2025-06-03T10:36:25.533+0000] {subprocess.py:106} INFO -                                  Dload  Upload   Total   Spent    Left  Speed
[2025-06-03T10:36:26.266+0000] {subprocess.py:106} INFO -
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
[2025-06-03T10:36:26.455+0000] {subprocess.py:106} INFO - Warning: Failed to open the file {BASE_PATH}/launches.json: No such file or
[2025-06-03T10:36:26.457+0000] {subprocess.py:106} INFO - Warning: directory
[2025-06-03T10:36:26.458+0000] {subprocess.py:106} INFO -
  5 26836    5  1360    0     0   1472      0  0:00:18 --:--:--  0:00:18  1472
[2025-06-03T10:36:26.459+0000] {subprocess.py:106} INFO - curl: (23) Failure writing output to destination
[2025-06-03T10:36:26.459+0000] {subprocess.py:110} INFO - Command exited with return code 23
[2025-06-03T10:36:26.468+0000] {taskinstance.py:3311} ERROR - Task failed with exception
Traceback (most recent call last):
  File "/home/airflow/.local/lib/python3.12/site-packages/airflow/models/taskinstance.py", line 767, in _execute_task
    result = _execute_callable(context=context, **execute_callable_kwargs)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/airflow/.local/lib/python3.12/site-packages/airflow/models/taskinstance.py", line 733, in _execute_callable
    return ExecutionCallableRunner(
           ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/airflow/.local/lib/python3.12/site-packages/airflow/utils/operator_helpers.py", line 252, in run
    return self.func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/airflow/.local/lib/python3.12/site-packages/airflow/models/baseoperator.py", line 422, in wrapper
    return func(self, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Defining dependencies between tasks

- Explore how task dependencies are defined in Airflow
- How to implement more complex patterns:
  - Conditional tasks
  - Branches
  - Joins

Basic dependencies between tasks

- Linear dependencies:
  - Tasks that executed one after another
- Fan-in/fan-out dependencies:
  - One task is linked to multiple downstream tasks, or vice versa

Linear dependencies

- The *rocket-launch* is a linear dependency DAG
- Each task must be completed before going to the next
- The next task depends on the previous task
- If a task fails, the next task that depends on it won't be executed

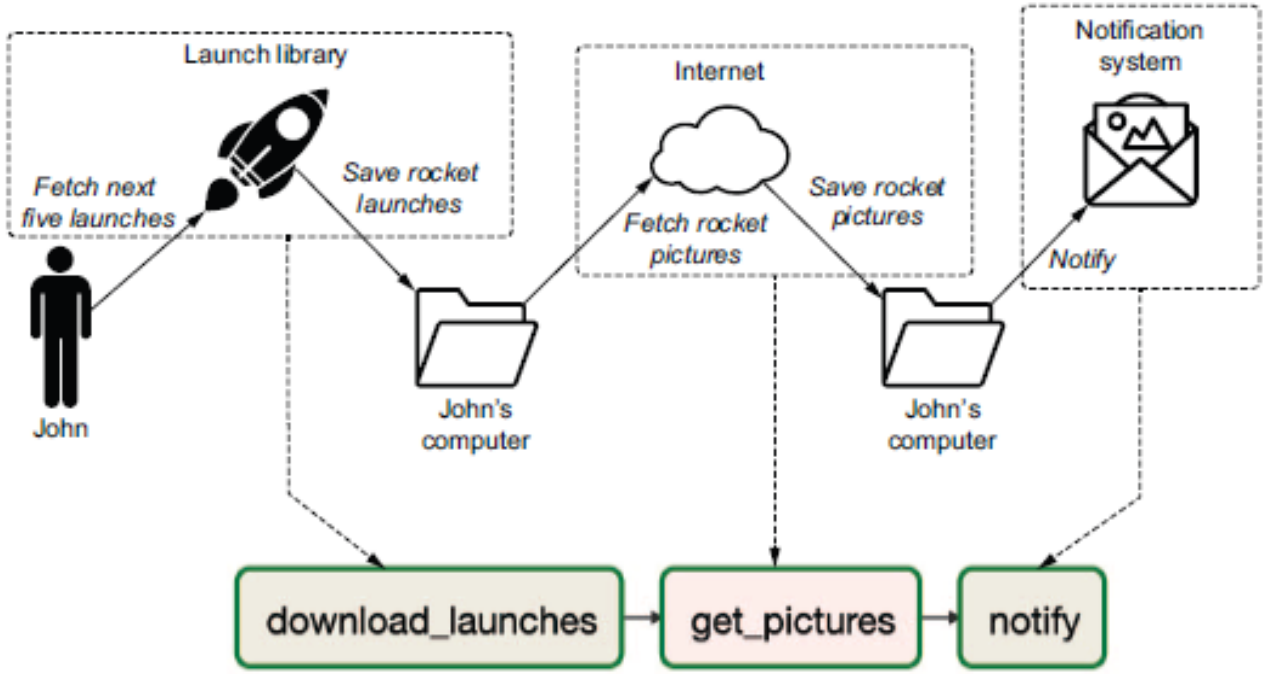
```
download_launches >> get_pictures
get_pictures >> notify
```

Set task dependencies one-by-one...

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

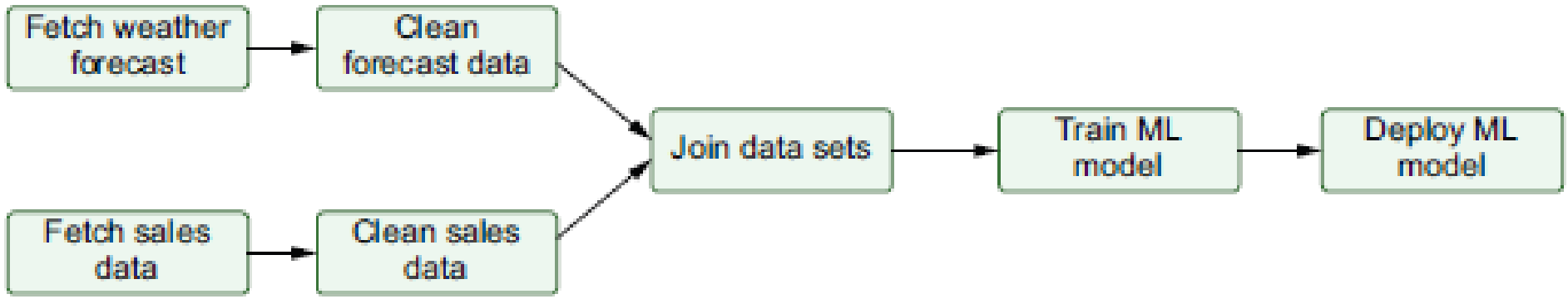
... or set multiple dependencies in one go.

```
download_launches = BashOperator(...)
get_pictures = PythonOperator(...)
notify = BashOperator(...)
```

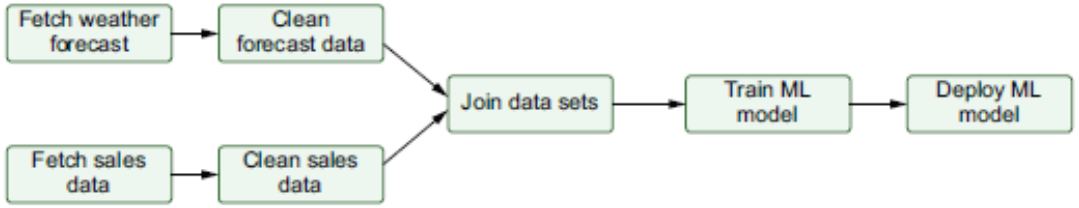


Fan-in/-out dependencies

- In addition to linear dependencies, Airflow's task dependencies can create more complex ones
- Umbrella example:
  - An umbrella company wants to predict sales using **weather data** and past **umbrella sales**
  - They want to use Machine Learning to do the prediction



Fan-in/-out dependencies



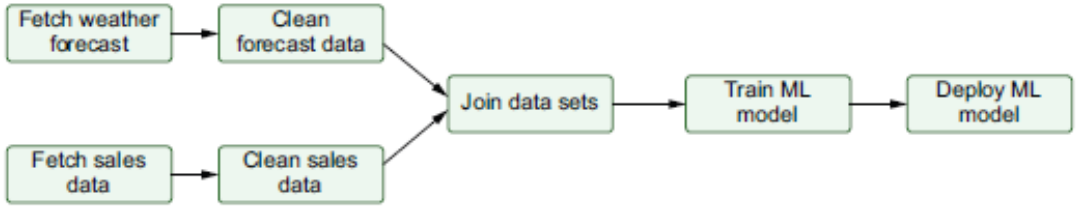
- Fetch weather >> clean, and fetch sales >> clean are independent:
  - fetch_weather >> clean_weather, fetch_sales >> clean_sales
- We can add a dummy task (not strictly necessary):
  - The dummy task kicks off both the fetch_weather and fetch_sales tasks
  - Fan-out dependency – linking one task to multiple downstream tasks

```
from airflow.operators.dummy import DummyOperator

start = DummyOperator(task_id="start")
start >> [fetch_weather, fetch_sales]
```

Create a dummy start task.

Create a fan-out (one-to-multiple) dependency.

Fan-in/-out dependencies



- Fan-in dependencies: cleaned weather and sales data are input to downstream task join_datasets

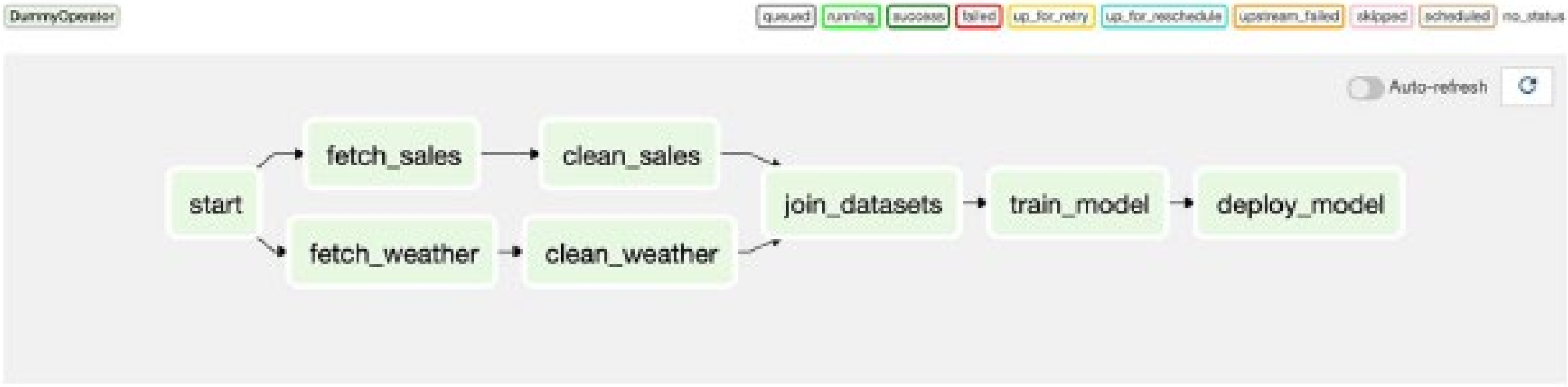
```
[clean_weather, clean_sales] >> join_datasets
```

- After fanning in with the join_datasets, all the downstream tasks are linear

```
join_datasets >> train_model >> deploy_model
```

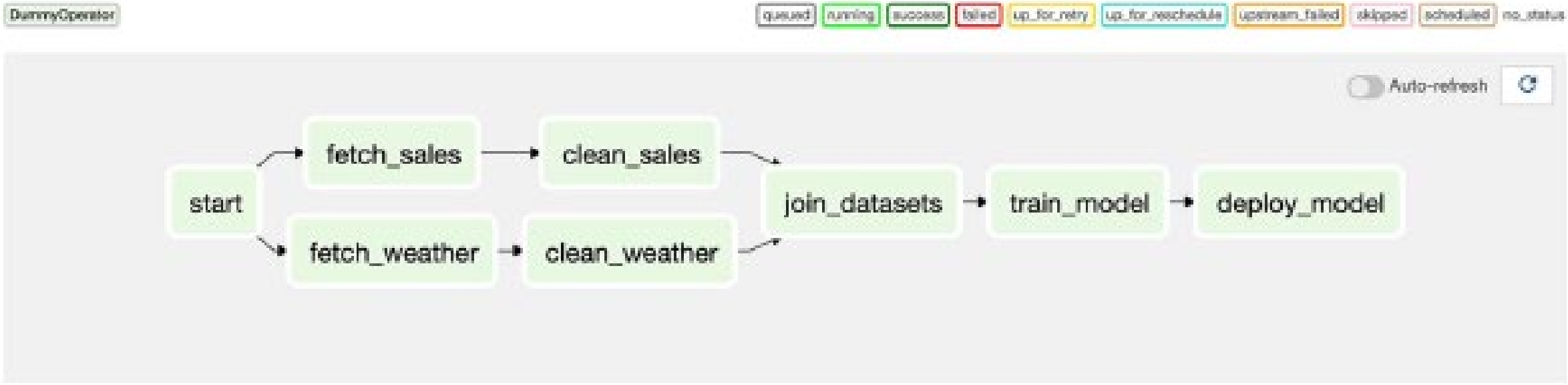
The overall DAG

- The execution of the DAG:
  - Firstly, Airflow will run the *start* task
  - After the start task is completed, it will initiate the fetch_sales and fetch_weather tasks, which will run in parallel
  - Completion of either of the fetch tasks will result in the start of the cleaning tasks
  - Only once the cleaning tasks have been completed, the join_datasets will start



Branching DAG

- Scenario:
  - Sales data will come from a different source after a particular date (e.g., two weeks later)
  - We may need a new system to fetch_sales and clean_sales



Branching DAG

- Solution one: with the same DAG, write separate Python methods to handle two different approaches

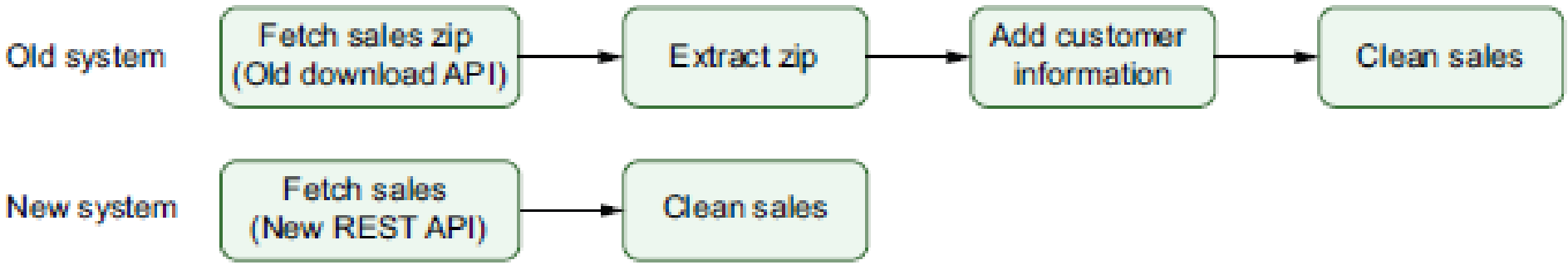
```
def _clean_sales(**context):
    if context["execution_date"] < ERP_CHANGE_DATE:
        _clean_sales_old(**context)
    else
        _clean_sales_new(**context)

...

clean_sales_data = PythonOperator(
    task_id="clean_sales",
    python_callable=_clean_sales,
)
```

```
def _fetch_sales(**context):
    if context["execution_date"] < ERP_CHANGE_DATE:
        _fetch_sales_old(**context)
    else:
        _fetch_sales_new(**context)
    ...
```

Branching DAG

- Solution one: with the same DAG, write separate Python methods to handle two different approaches

- Advantage: no need to change DAG; write flexible code
- Disadvantage: only for the same tasks; what if tasks are different? It's hard to see which code branch is used by Airflow during DAG run



Branching DAG

- Solution two:
  - to develop two distinct sets of tasks
  - DAG to choose which task to choose
- Building two tasks: create tasks for each system separately and link the respective tasks

```
fetch_sales_old = PythonOperator(...)
clean_sales_old = PythonOperator(...)

fetch_sales_new = PythonOperator(...)
clean_sales_new = PythonOperator(...)

fetch_sales_old >> clean_sales_old
fetch_sales_new >> clean_sales_new
```

Branching DAG

- Need to connect these tasks to the rest of DAG
- Use BranchPythonOperator:
  - Callable passed to BranchPythonOperator to return the ID of a downstream task
  - The returned ID indicates which tasks to execute

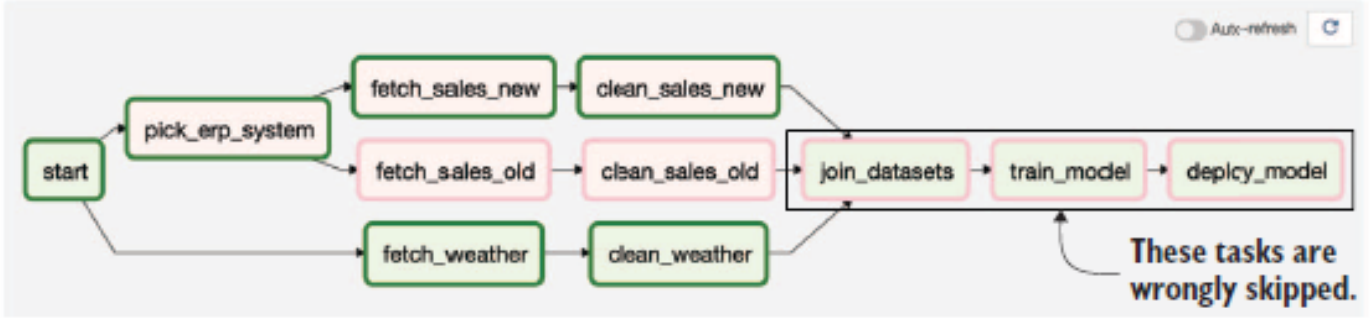
```
def _pick_erp_system(**context):
    ...

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)
```

Branching DAG

- Our example to choose old or new ERP to execute:
  - Airflow will execute “old” ERP tasks for execution dates occurring before the switch date; otherwise, the “new” ERP tasks

```
def _pick_erp_system(**context):
    if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)

pick_erp_system >> [fetch_sales_old, fetch_sales_new]
```

Branching DAG



- We can then connect all the tasks:

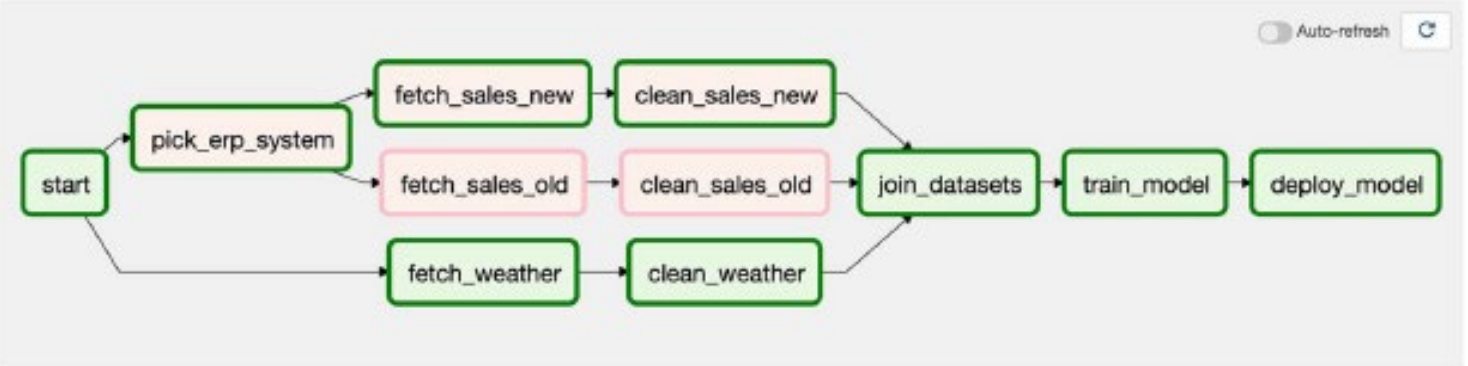
```
start_task >> pick_erp_system

[clean_sales_old, clean_sales_new] >> join_datasets
```

- However, this won't work: join_datasets will never be executed
- Join_datasets can only run, after both clean_sales_old and clean_sales_new are executed, but this would never happen

Branching DAG

- To fix the issue, we can change the argument *trigger_rules* in *join_datasets*
- The default is *trigger_rules=all_success*, but this never happens for this example
- We will use *none_failed*: the task will run as soon as all of its parents are done with executing and none have failed

```
join_datasets = PythonOperator(
    ...,
    trigger_rule="none_failed",
)
```



Branching DAG

- We can make it better in logic by adding a dummy task
- Want two edges to join_datasets, which better reflects the flow of our tasks

```
from airflow.operators.dummy import DummyOperator

join_branch = DummyOperator(
    task_id="join_erp_branch",
    trigger_rule="none_failed"
)

[clean_sales_old, clean_sales_new] >> join_branch
join_branch >> join_datasets
```

Reference

All the materials were taken from the following book:

- Chapters 2 & 5 in "*Data Pipelines with Apache Airflow*", by Bas Harenslak & Julian de Ruiter, April 2021, ISBN 9781617296901, Manning Publications