People and companies are continuously becoming more data-driven and are developing data pipelines as part of their daily business. Data volumes involved in these business processes have increased substantially over the years, from megabytes per day to gigabytes per minute. Though handling this data deluge may seem like a considerable challenge, these increasing data volumes can be managed with the appropriate tooling.

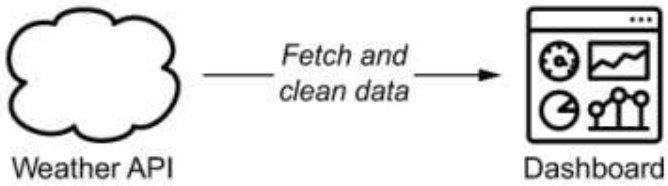
Apache Airflow, a batch-oriented framework for building data pipelines. Airflow's key feature is that it enables you to easily build scheduled data pipelines using a flexible Python framework, while also providing many building blocks that allow you to stitch together the many different technologies encountered in modern technological landscapes.

1.1 Introducing data pipelines

Data pipelines generally consist of several tasks or actions that need to be executed to achieve the desired result. For example, say we want to build a small weather dashboard that tells us what the weather will be like in the coming week (figure 1.1). To implement this live weather dashboard, we need to perform something like the following steps:

- Fetch weather forecast data from a weather API.
- Clean or otherwise transform the fetched data (e.g., converting temperatures from Fahrenheit to Celsius or vice versa), so that the data suits our purpose.
- Push the transformed data to the weather dashboard.

Figure 1.1 Overview of the weather dashboard use case, in which weather data is fetched from an external API and fed into a dynamic dashboard



As you can see, this relatively simple pipeline already consists of three different tasks that each perform part of the work. Moreover, these tasks need to be executed in a specific order, as it (for example) doesn't make sense to try transforming the data before fetching it. Similarly, we can't push any new data to the dashboard until it has undergone the required transformations. As such, we need to make sure that this implicit task order is also enforced when running this data process.

1.1.1 Data pipelines as graphs

One way to make dependencies between tasks more explicit is to draw the data pipeline as a graph. In this graph-based representation, tasks are represented as nodes in the graph, while dependencies between tasks are represented by directed edges between the task nodes. The direction of the edge indicates the direction of the dependency, with an edge pointing from task A to task B, indicating that task A needs to be completed before task B can start. Note that this type of graph is generally called a *directed graph*, due to the directions in the graph edges.

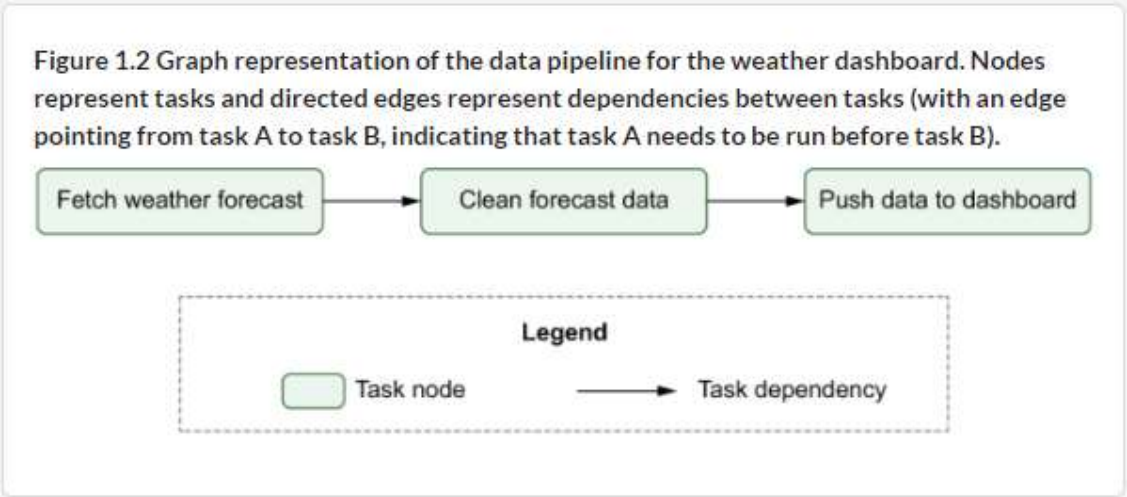
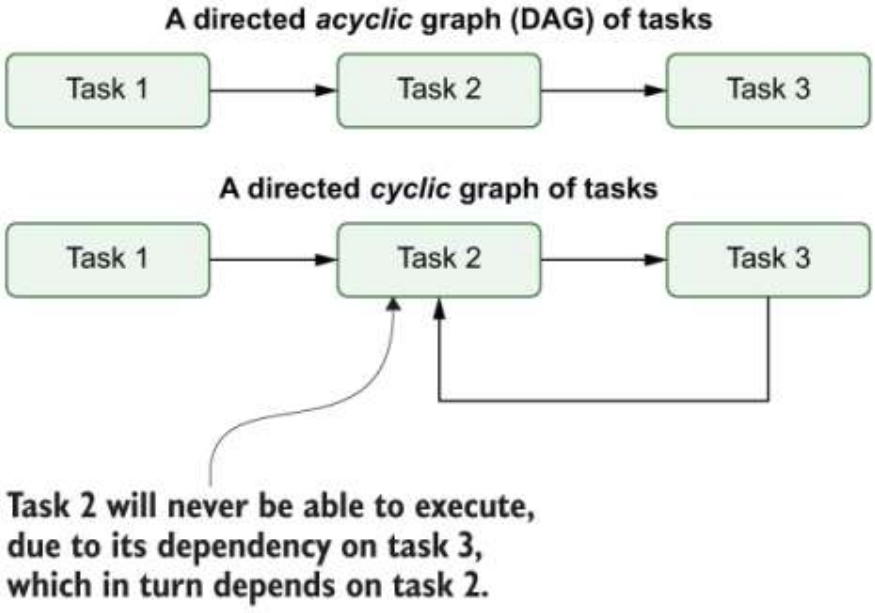


Figure 1.2 Graph representation of the data pipeline for the weather dashboard. Nodes represent tasks and directed edges represent dependencies between tasks (with an edge pointing from task A to task B, indicating that task A needs to be run before task B).

This type of graph is typically called a *directed acyclic graph* (DAG), as the graph contains *directed* edges and does not contain any loops or cycles (*acyclic*). This acyclic property is extremely important, as it prevents us from running into circular dependencies (figure 1.3) between tasks (where task A depends on task B and vice versa). These circular dependencies become problematic when trying to execute the graph, as we run into a situation where task 2 can only execute once task 3 has been completed, while task 3 can only execute once task 2 has been completed. This logical inconsistency leads to a deadlock type of situation, in which neither task 2 nor 3 can run, preventing us from executing the graph.

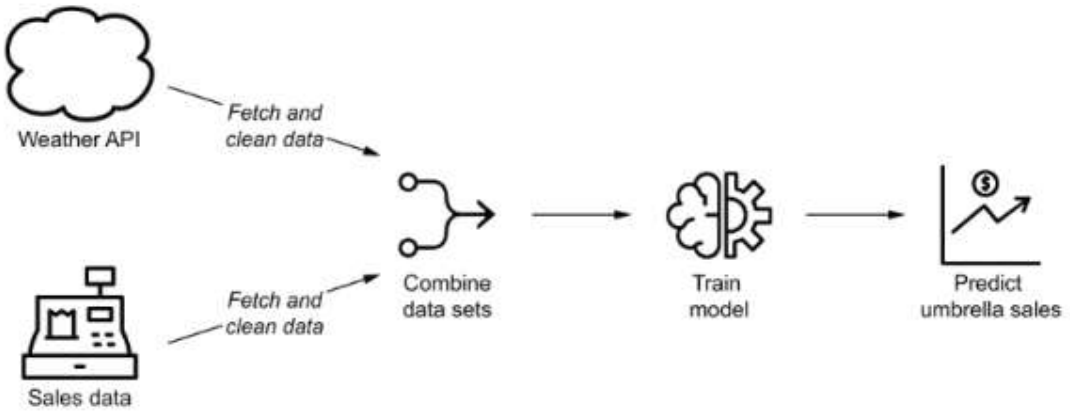
Figure 1.3 Cycles in graphs prevent task execution due to circular dependencies. In acyclic graphs (top), there is a clear path to execute the three different tasks. However, in cyclic graphs (bottom), there is no longer a clear execution path due to the interdependency between tasks 2 and 3.



1.1.3 Pipeline graphs vs. sequential scripts

Although the graph representation of a pipeline provides an intuitive overview of the tasks in the pipeline and their dependencies, you may find yourself wondering why we wouldn't just use a simple script to run this linear chain of three steps. To illustrate some advantages of the graph-based approach, let's jump to a slightly bigger example. In this new use case, we've been approached by the owner of an umbrella company, who was inspired by our weather dashboard and would like to try to use machine learning (ML) to increase the efficiency of their operation. To do so, the company owner would like us to implement a data pipeline that creates an ML model correlating umbrella sales with weather patterns. This model can then be used to predict how much demand there will be for the company's umbrellas in the coming weeks, depending on the weather forecasts for those weeks (figure 1.5).

Figure 1.5 Overview of the umbrella demand use case, in which historical weather and sales data are used to train a model that predicts future sales demands depending on weather forecasts



To build a pipeline for training the ML model, we need to implement something like the following steps:

1. Prepare the sales data by doing the following:
    - Fetching the sales data from the source system
    - Cleaning/transforming the sales data to fit requirements
2. Prepare the weather data by doing the following:
    - Fetching the weather forecast data from an API
    - Cleaning/transforming the weather data to fit requirements
3. Combine the sales and weather data sets to create the combined data set that can be used as input for creating a predictive ML model.
4. Train the ML model using the combined data set.
5. Deploy the ML model so that it can be used by the business.

This pipeline can be represented using the same graph-based representation that we used before, by drawing tasks as nodes and data dependencies between tasks as edges.

One important difference from our previous example is that the first steps of this pipeline (fetching and clearing the weather/sales data) are in fact independent of each other, as they involve two separate data sets. This is clearly illustrated by the two separate branches in the graph representation of the pipeline (figure 1.6),

which can be executed in parallel if we apply our graph execution algorithm, making better use of available resources and potentially decreasing the running time of a pipeline compared to executing the tasks sequentially.

Figure 1.6 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.

Another useful property of the graph-based representation is that it clearly separates pipelines into small incremental tasks rather than having one monolithic script or process that does all the work. Although having a single monolithic script may not initially seem like that much of a problem, it can introduce some inefficiencies when tasks in the pipeline fail, as we would have to rerun the entire script. In contrast, in the graph representation, we need only to rerun any failing tasks (and any downstream dependencies).

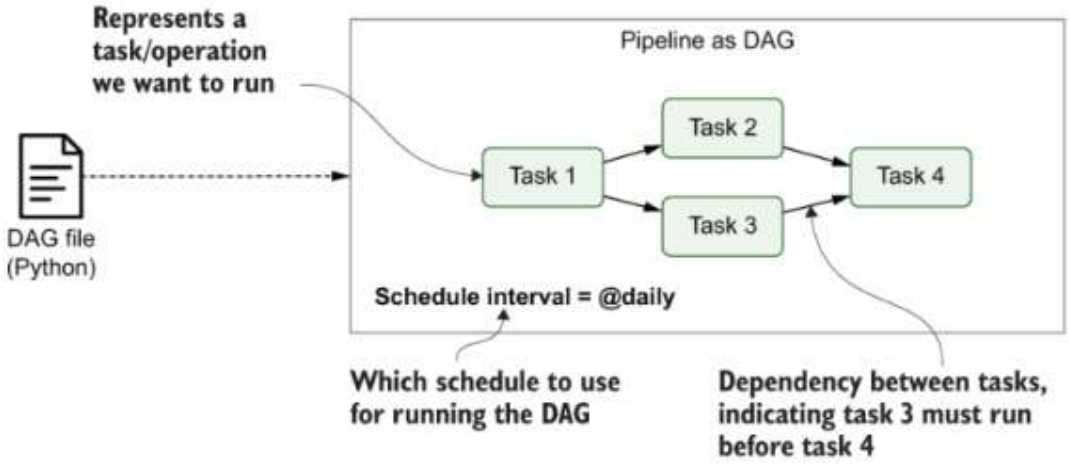
1.2 Introducing Airflow

Airflow, an open-source solution for developing and monitoring workflows. In this section, we'll provide a Birds eye view of what Airflow does, after which we'll jump into a more detailed examination of whether it is a good fit for your use case.

1.2.1 Defining pipelines flexibly in (Python) code

In Airflow, you define your DAGs using Python code in DAG files, which are essentially Python scripts that describe the structure of the corresponding DAG. As such, each DAG file typically describes the set of tasks for a given DAG and the dependencies between the tasks, which are then parsed by Airflow to identify the DAG structure (figure 1.7). Other than this, DAG files typically contain some additional metadata about the DAG telling Airflow how and when it should be executed, and so on.

Figure 1.7 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.



One advantage of defining Airflow DAGs in Python code is that this programmatic approach provides you with a lot of flexibility for building DAGs. For example, as we will see later in this book, you can use Python code to dynamically generate optional tasks depending on certain conditions or even generate entire DAGs based on external metadata or configuration files. This flexibility gives a great deal of customization in how you build your pipelines, allowing you to fit Airflow to your needs for building arbitrarily complex pipelines.

In addition to this flexibility, another advantage of Airflow's Python foundation is that tasks can execute any operation that you can implement in Python. Over time, this has led to the development of many Airflow extensions that enable you to execute tasks across a wide variety of systems, including external databases, big data technologies, and various cloud services, allowing you to build complex data pipelines bringing together data processes across many different systems.

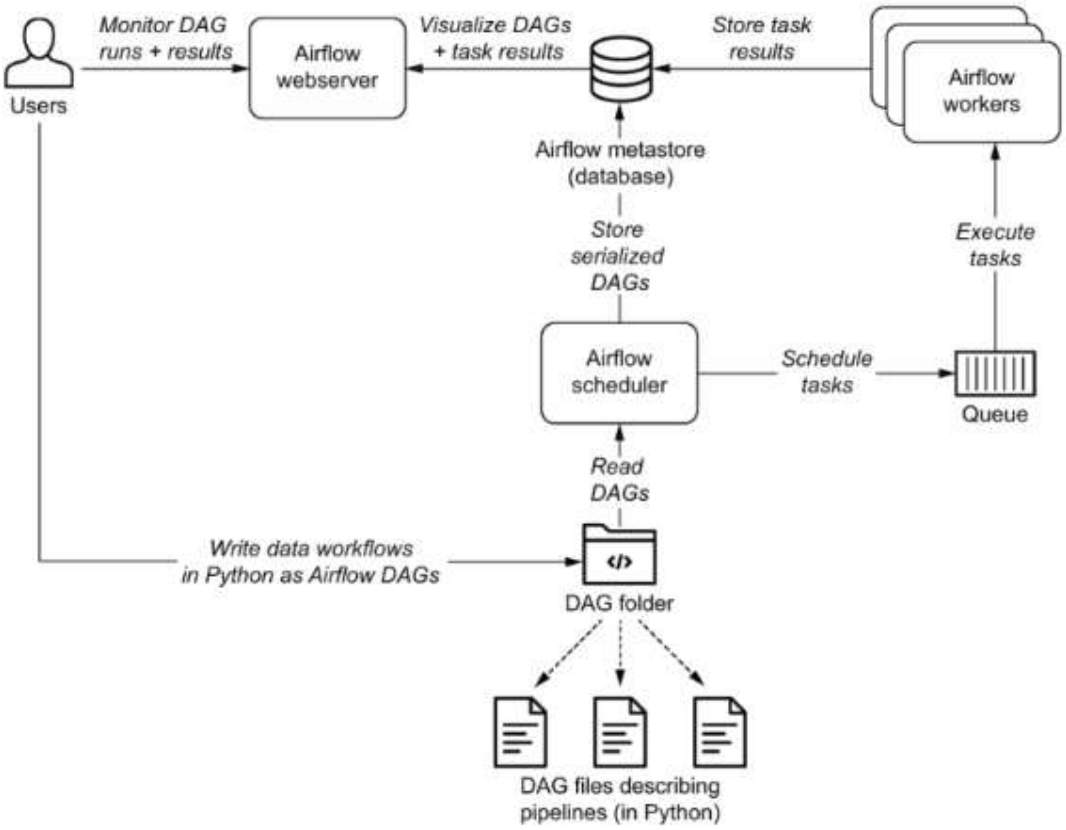
1.2.2 Scheduling and executing pipelines

Once you've defined the structure of your pipeline(s) as DAG(s), Airflow allows you to define a schedule interval for each DAG, which determines exactly when your pipeline is run by Airflow. This way, you can tell Airflow to execute your DAG every hour, every day, every week, and so on, or even use more complicated schedule intervals based on Cron-like expressions.

To see how Airflow executes your DAGs, let's briefly look at the overall process involved in developing and running Airflow DAGs. At a high level, Airflow is organized into three main components (figure 1.8):

- *The Airflow scheduler*—Parses DAGs, checks their schedule interval, and (if the DAGs' schedule has passed) starts scheduling the DAGs' tasks for execution by passing them to the Airflow workers.
- *The Airflow workers*—Pick up tasks that are scheduled for execution and execute them. As such, the workers are responsible for actually "doing the work."
- *The Airflow webserver*—Visualizes the DAGs parsed by the scheduler and provides the main interface for users to monitor DAG runs and their results.

Figure 1.8 Overview of the main components involved in Airflow (e.g., the Airflow webserver, scheduler, and workers)



The heart of Airflow is arguably the scheduler, as this is where most of the magic happens that determines when and how your pipelines are executed. At a high level, the scheduler runs through the following steps (figure 1.9):

1. Once users have written their workflows as DAGs, the files containing these DAGs are read by the scheduler to extract the corresponding tasks, dependencies, and schedule interval of each DAG.
2. For each DAG, the scheduler then checks whether the schedule interval for the DAG has passed since the last time it was read. If so, the tasks in the DAG are scheduled for execution.

3. For each scheduled task, the scheduler then checks whether the dependencies (= upstream tasks) of the task have been completed. If so, the task is added to the execution queue.
4. The scheduler waits for several moments before starting a new loop by jumping back to step 1.

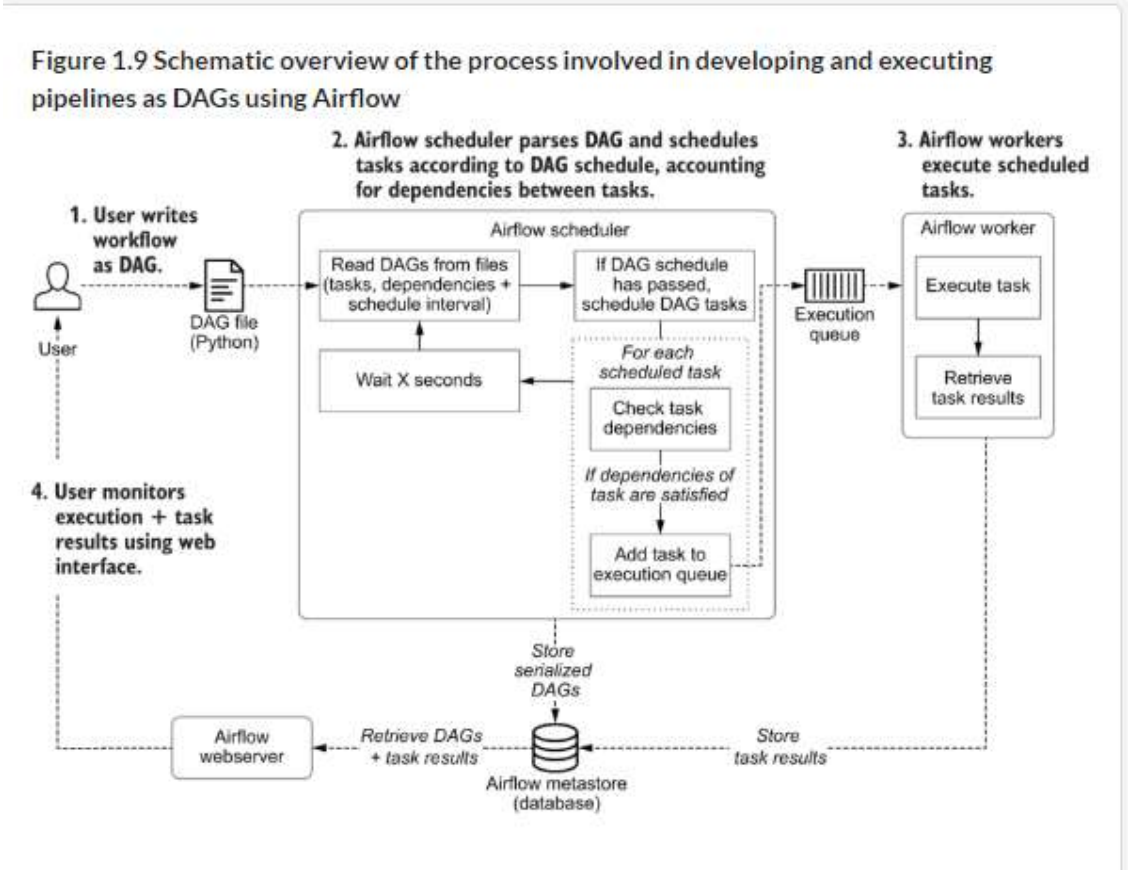


Figure 1.9 Schematic overview of the process involved in developing and executing pipelines as DAGs using Airflow

1.2.3 Monitoring and handling failures

Figure 1.10 The login page for the Airflow web interface.

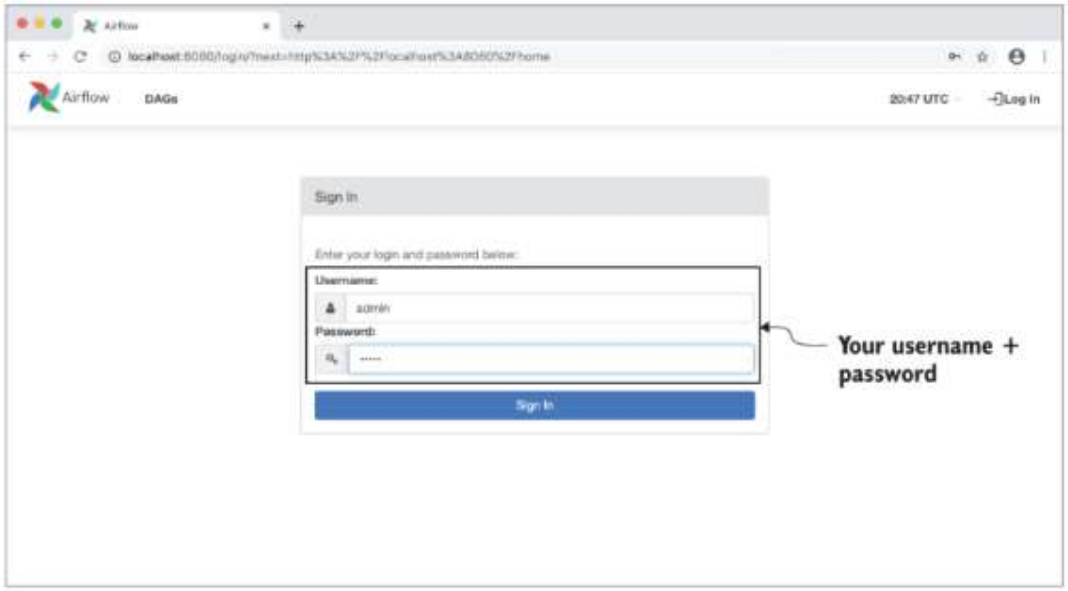


Figure 1.11 The main page of Airflow's web interface, showing an overview of the available DAGs and their recent results

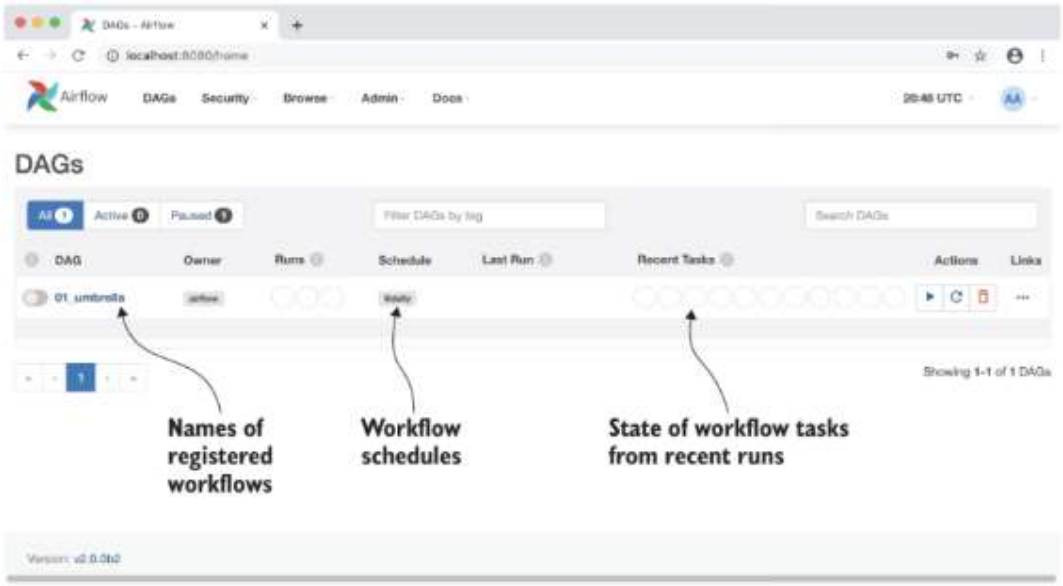


Figure 1.12 The graph view in Airflow's web interface, showing an overview of the tasks in an individual DAG and the dependencies between these tasks

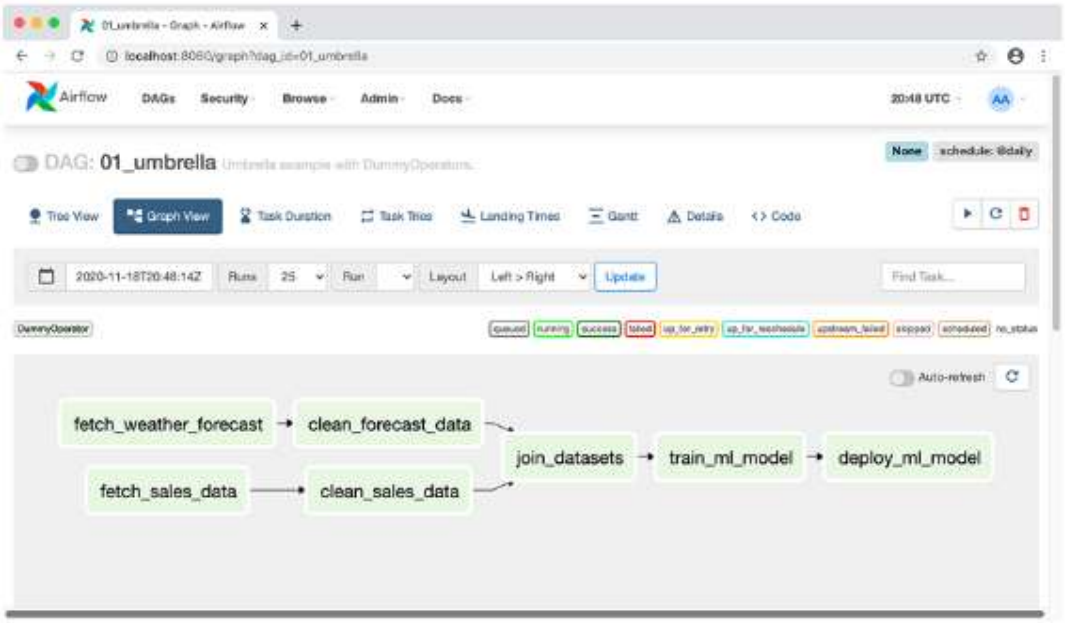
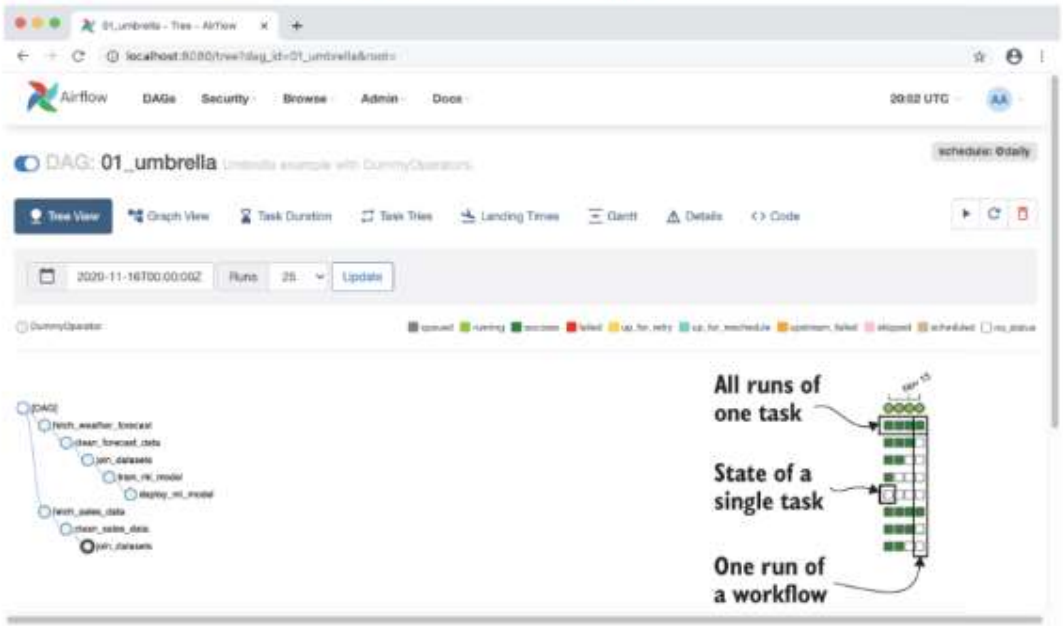


Figure 1.13 Airflow's tree view, showing the results of multiple runs of the umbrella sales model DAG (most recent + historical runs). The columns show the status of one execution of the DAG and the rows show the status of all executions of a single task. Colors indicate the result of the corresponding task. Users can also click on the task "squares" for more details about a given task instance, or to reset the state of a task so that it can be rerun by Airflow, if desired.



1.2.4 Incremental loading and backfilling

One powerful feature of Airflow's scheduling semantics is that the schedule intervals not only trigger DAGs at specific time points (similar to, for example, Cron), but also provide details about the last and (expected) next schedule intervals. This essentially allows you to divide time into discrete intervals (e.g., every day, week, etc.), and run your DAG for each of these intervals.

This property of Airflow's schedule intervals is invaluable for implementing efficient data pipelines, as it allows you to build incremental data pipelines. In these incremental pipelines, each DAG run processes only data for the corresponding time slot (the data's *delta*) instead of having to reprocess the entire data set every time. Especially for larger data sets, this can provide significant time and cost benefits by avoiding expensive recomputation of existing results.

Schedule intervals become even more powerful when combined with the concept of *backfilling*, which allows you to execute a new DAG for historical schedule intervals that occurred in the past. This feature allows you to easily create (or *backfill*) new data sets with historical data simply by running your DAG for these past schedule intervals. Moreover, by clearing the results of past runs, you

can also use this Airflow feature to easily rerun any historical tasks if you make changes to your task code, allowing you to easily reprocess an entire data set when needed.

1.3 When to use Airflow

1.3.1 Reasons to choose Airflow

In the past sections, we've already described several key features that make Airflow ideal for implementing batch-oriented data pipelines. In summary, these include the following:

- The ability to implement pipelines using Python code allows you to create arbitrarily complex pipelines using anything you can dream up in Python.
- The Python foundation of Airflow makes it easy to extend and add integrations with many different systems. In fact, the Airflow community has already developed a rich collection of extensions that allow Airflow to integrate with many different types of databases, cloud services, and so on.
- Rich scheduling semantics allow you to run your pipelines at regular intervals and build efficient pipelines that use incremental processing to avoid expensive recomputation of existing results.
- Features such as backfilling enable you to easily (re)process historical data, allowing you to recompute any derived data sets after making changes to your code.
- Airflow's rich web interface provides an easy view for monitoring the results of your pipeline runs and debugging any failures that may have occurred.

1.3.2 Reasons not to choose Airflow

Although Airflow has many rich features, several of Airflow's design choices may make it less suitable for certain cases. For example, some use cases that are not a good fit for Airflow include the following:

- Handling streaming pipelines, as Airflow is primarily designed to run recurring or batch-oriented tasks, rather than streaming workloads.
- Implementing highly dynamic pipelines, in which tasks are added/removed between every pipeline run. Although Airflow can implement this kind of dynamic behavior, the web interface will only show tasks that are still defined in the most recent version of the DAG. As such, Airflow favors pipelines that do not change in structure every time they run.
- Teams with little or no (Python) programming experience, as implementing DAGs in Python can be daunting with little Python experience. In such teams, using a workflow manager with a graphical interface (such as Azure Data Factory) or a static workflow definition may make more sense.
- Similarly, Python code in DAGs can quickly become complex for larger use cases. As such, implementing and maintaining Airflow DAGs require proper engineering rigor to keep things maintainable in the long run.

Lab 1- core_concepts.py

Lab 2- 01_umberella.py