Apache Airflow

Mingjun Zhong

Department of Computing Science

University of Aberdeen

Content

- Showing how data pipelines can be represented in workflows as graphs of tasks
- Understanding how Airflow fits into the ecosystem of workflow managers
- Determining when to use or not use Airflow

Apache Airflow is a workflow management tool

- A pipeline orchestration and workflow management tool
- A batch-oriented framework for building data pipelines.
- Enables building scheduled data pipelines using Python frameworks
- Glueing together various technologies/tools for DE
- Coordinating different (distributed) systems
- Orchestrate different components for processing data in data pipelines

Data pipelines

- Data pipelines consist of tasks or actions to be executed to achieve the desired result.

- Weather dashboard:
    1. Fetch weather forecast data from a weather API
    2. 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.
    3. Push the transformed data to the weather dashboard.
- Three different tasks; executed in order

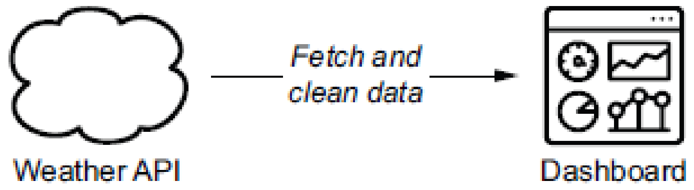
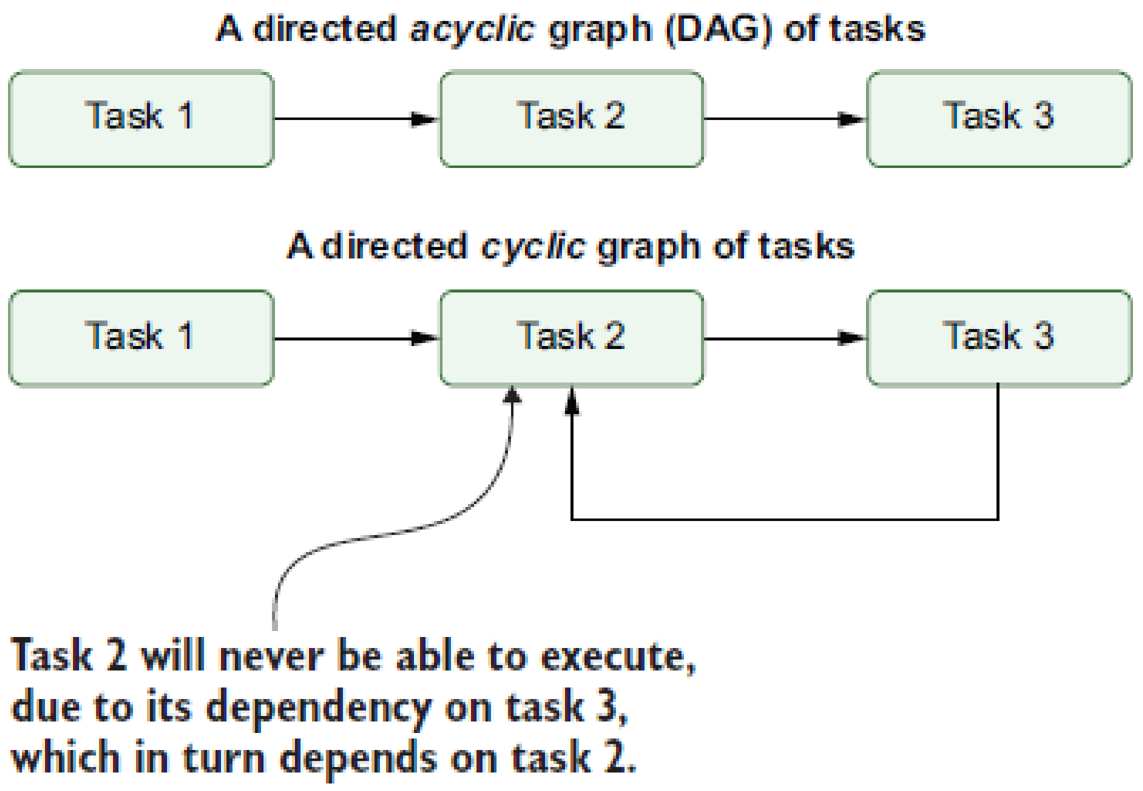


Figure 1.1 Overview of the weather dashboard use case, in which weather data is fetched from an external API and fed into a dynamic dashboard

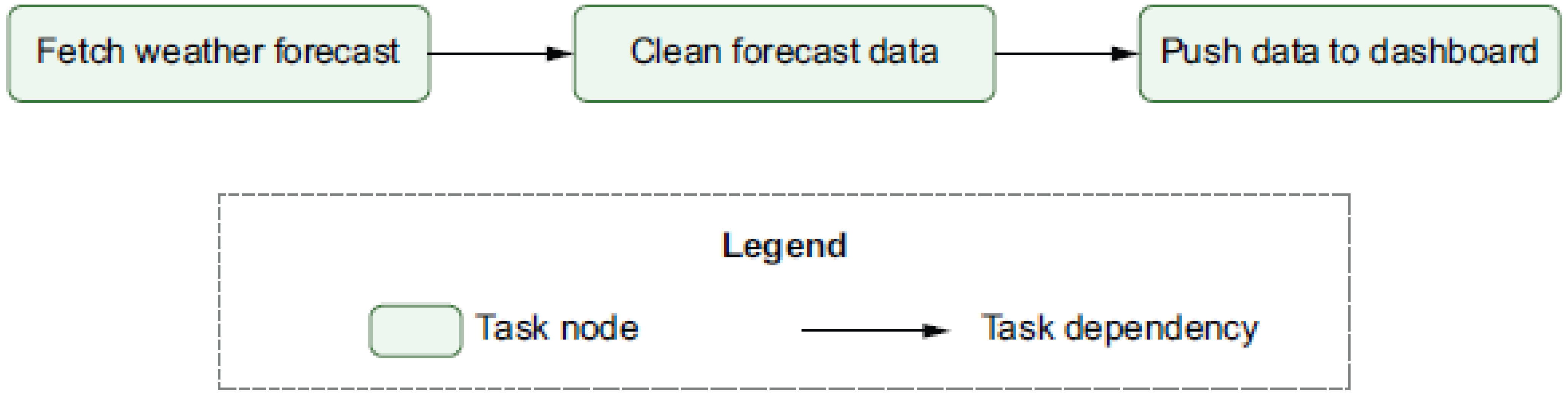
Data pipelines as graphs

- **Directed Acyclic Graph (DAG):** nodes and edges with directions.
    - Direction of the edge indicates the direction from parent to child nodes
    - No loops or cycles
- **Data pipelines:**
    - Tasks are nodes
    - Edges are dependencies between tasks: the order of tasks
- DAG is implemented by Airflow


A directed *acyclic* graph (DAG) of tasks
Task 1
Task 2
Task 3
A directed *cyclic* graph of tasks
Task 1
Task 2
Task 3
Task 2 will never be able to execute, due to its dependency on task 3, which in turn depends on task 2.

Weather dashboard pipeline

- The DAG shows the flow of the tasks in weather dashboard example

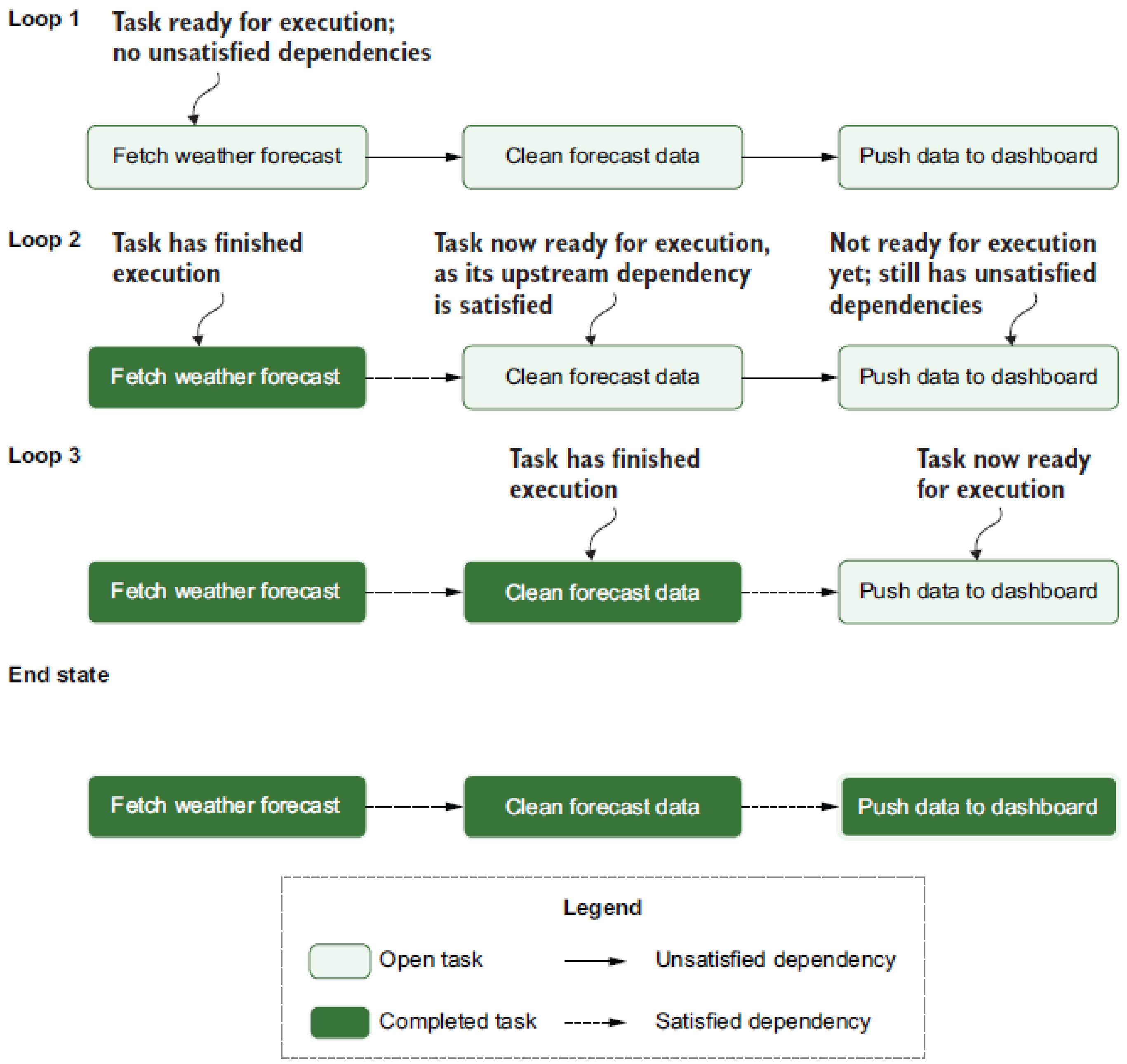


Executing a pipeline graph

- Once all tasks are defined, and the DAGs
- Algorithm for executing the pipeline graph can be
  1. For each open (= uncompleted) task in the graph, do the following:
     – For each edge pointing toward the task, check if the “upstream” task on the other end of the edge has been completed.
     – If all upstream tasks have been completed, add the task under consideration to a queue of tasks to be executed.
  2. Execute the tasks in the execution queue, marking them completed once they finish performing their work.
  3. Jump back to step 1 and repeat until all tasks in the graph have been completed.

Weather dashboard

Using the DAG structure to execute tasks in the data pipeline in the correct order: depicts each task's state during each of the loops through the algorithm, demonstrating how this leads to the completed execution of the pipeline (end state)

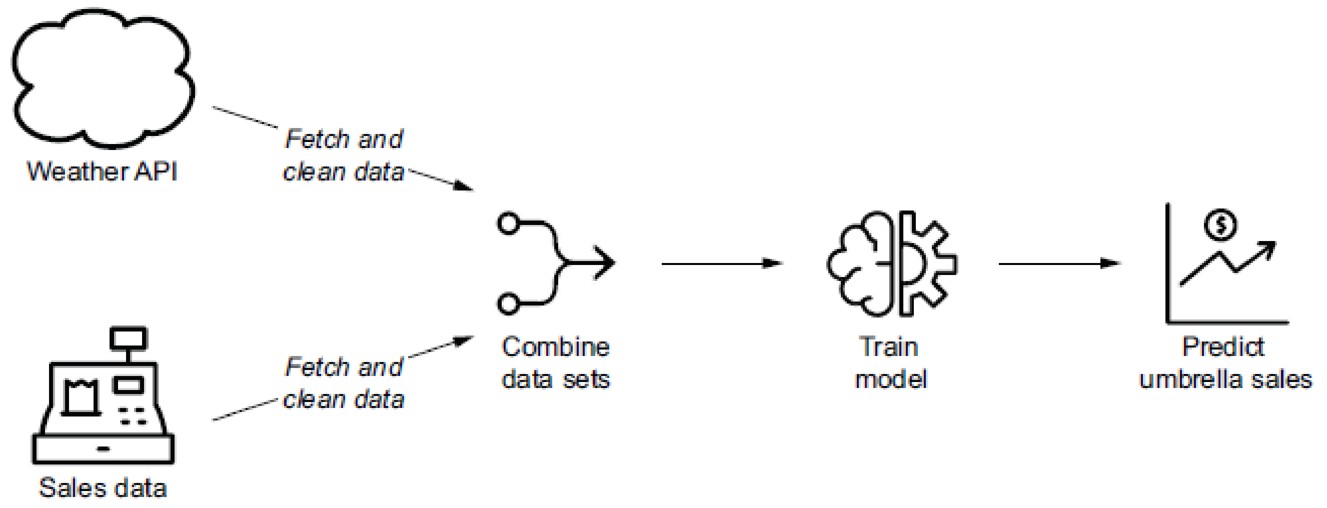


Pipeline graphs vs sequential scripts

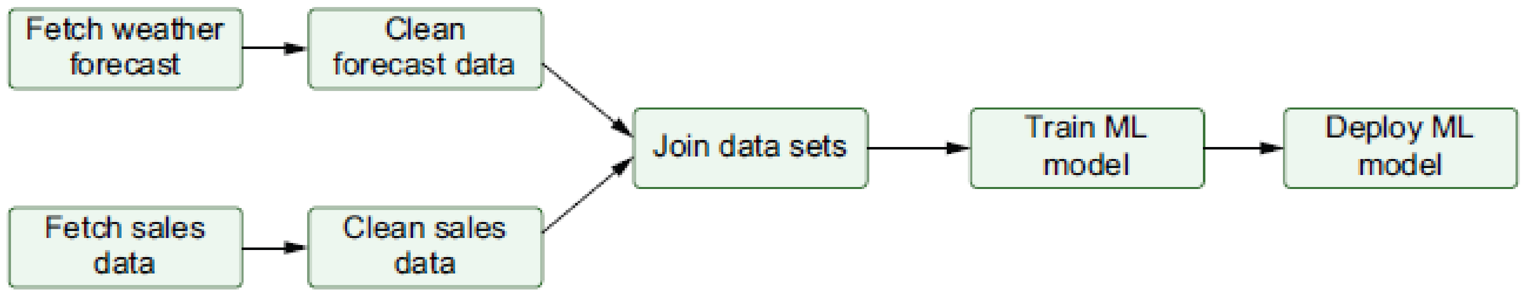
- The weather dashboard example suggests sequential scripts
- Graphs have advantages over sequential scripts:
  - parallel vs sequential
  - Small incremental tasks vs monolithic scripts
- An umbrella company wants to sell umbrellas based on weather by building the following Machine Learning model:
  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.

Pipeline graphs vs sequential scripts

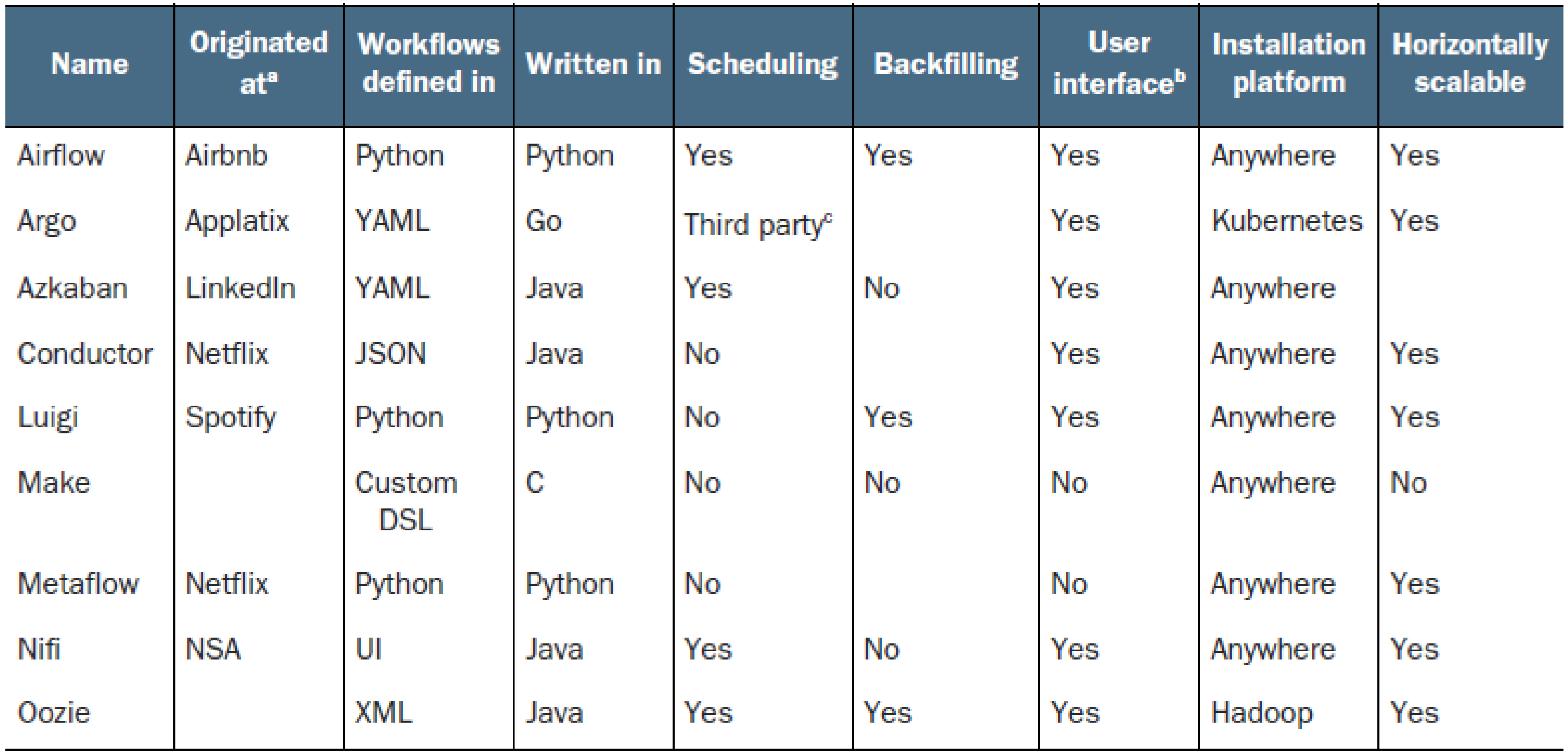
Use case



DAG



Orchestration & 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

Apache Airflow

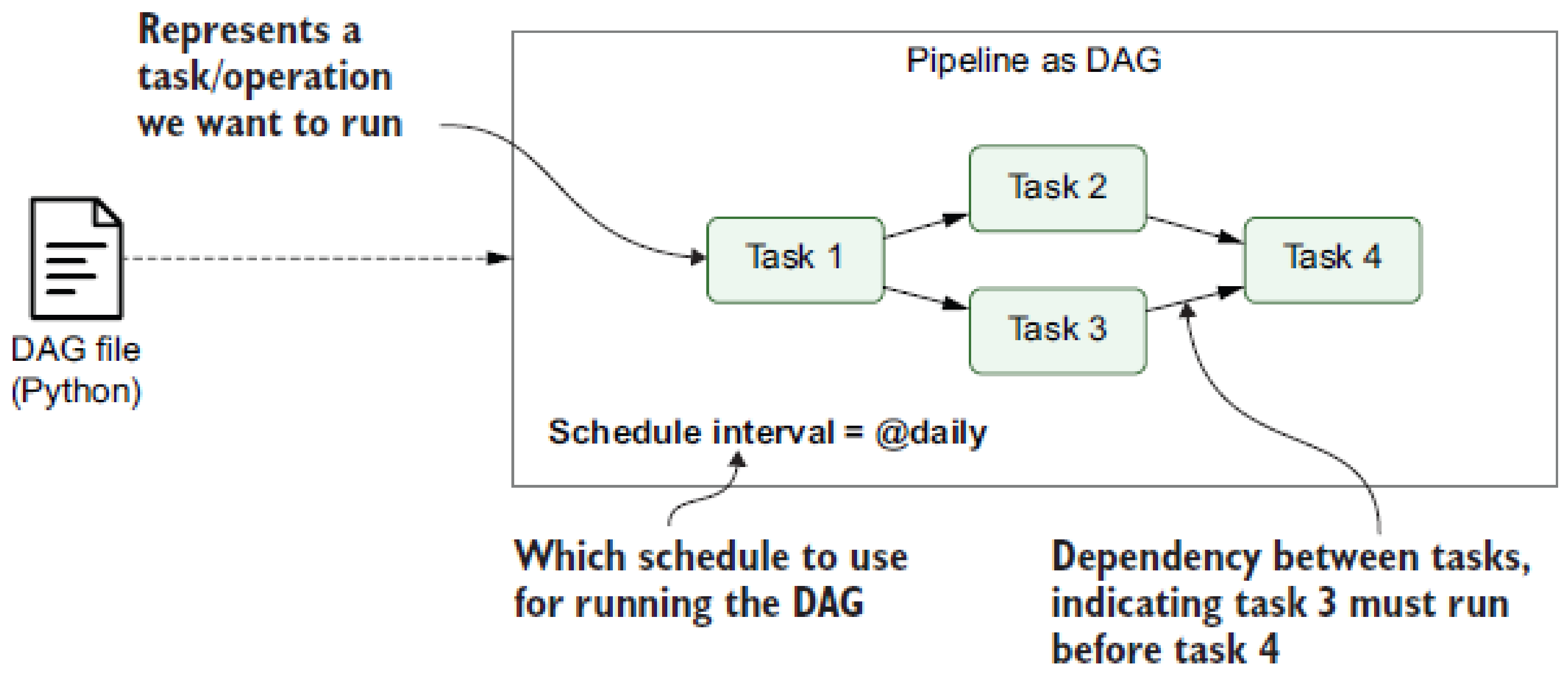
- Open source
- Define pipelines or workflows as DAGs of tasks
- Define DAGs using Python code in DAG files – essential Python scripts
- Each DAG file describes the set of tasks and the dependencies between them
- DAG files contain additional metadata about the DAG: telling Airflow how and when it should be executed

Apache Airflow

- Programming flexibility: using Python to program any tasks
- Can dynamically generate optional tasks depending on conditions
- Can customize to fit your complex pipelines
- Tasks can execute any operation that you can implement in Python

Apache Airflow

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


Represents a task/operation we want to run
Pipeline as DAG
Task 1
Task 2
Task 3
Task 4
DAG file (Python)
Schedule interval = @daily
Which schedule to use for running the DAG
Dependency between tasks, indicating task 3 must run before task 4

Scheduling and executing pipelines

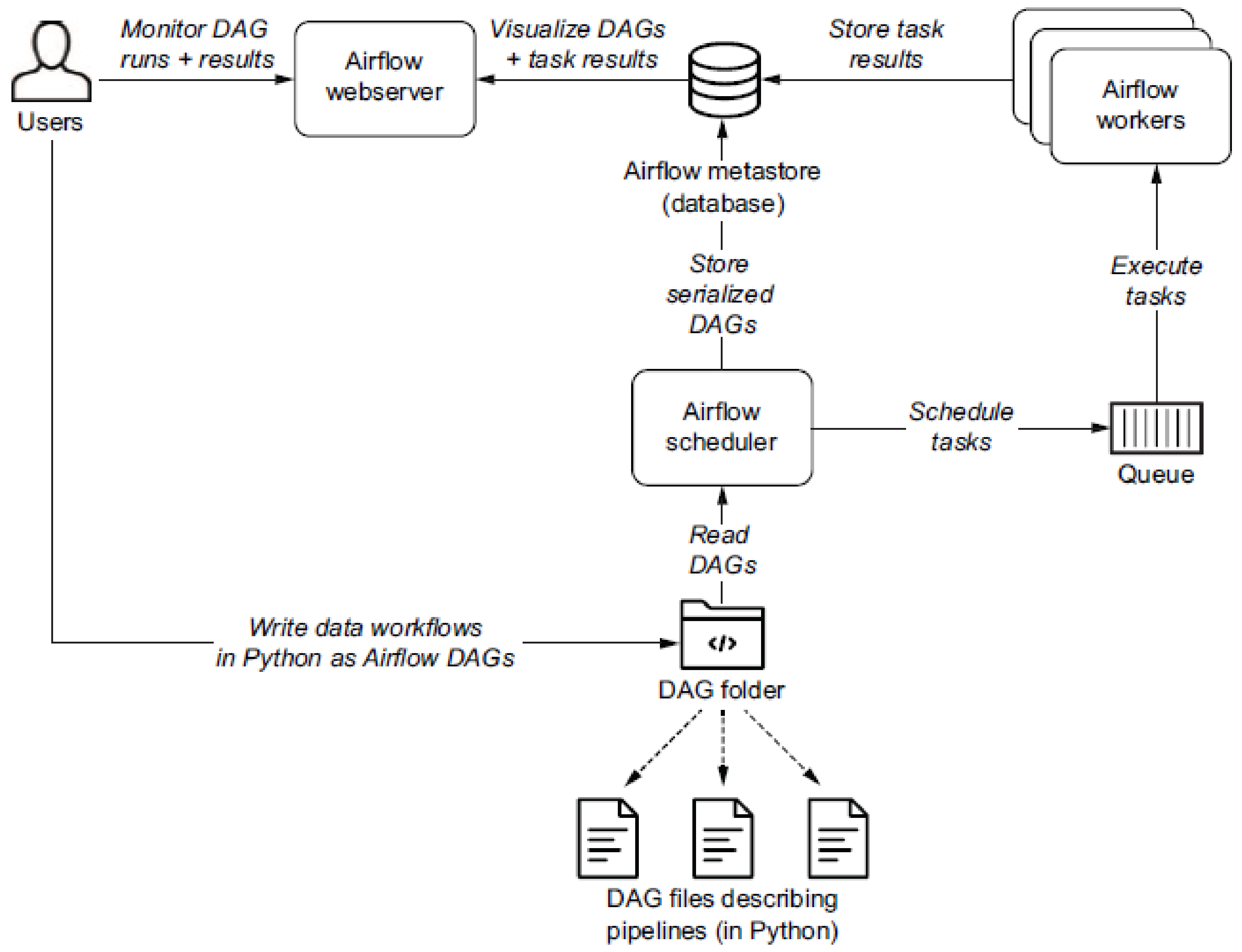
- Define a schedule interval for each DAG, after the structure of pipeline is defined

- Tell Airflow to execute your DAG every hour, every day, every week, etc.

Scheduling and executing pipelines

- Overall process for running Airflow DAGs, three main components:
  - **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.

The main components in Airflow

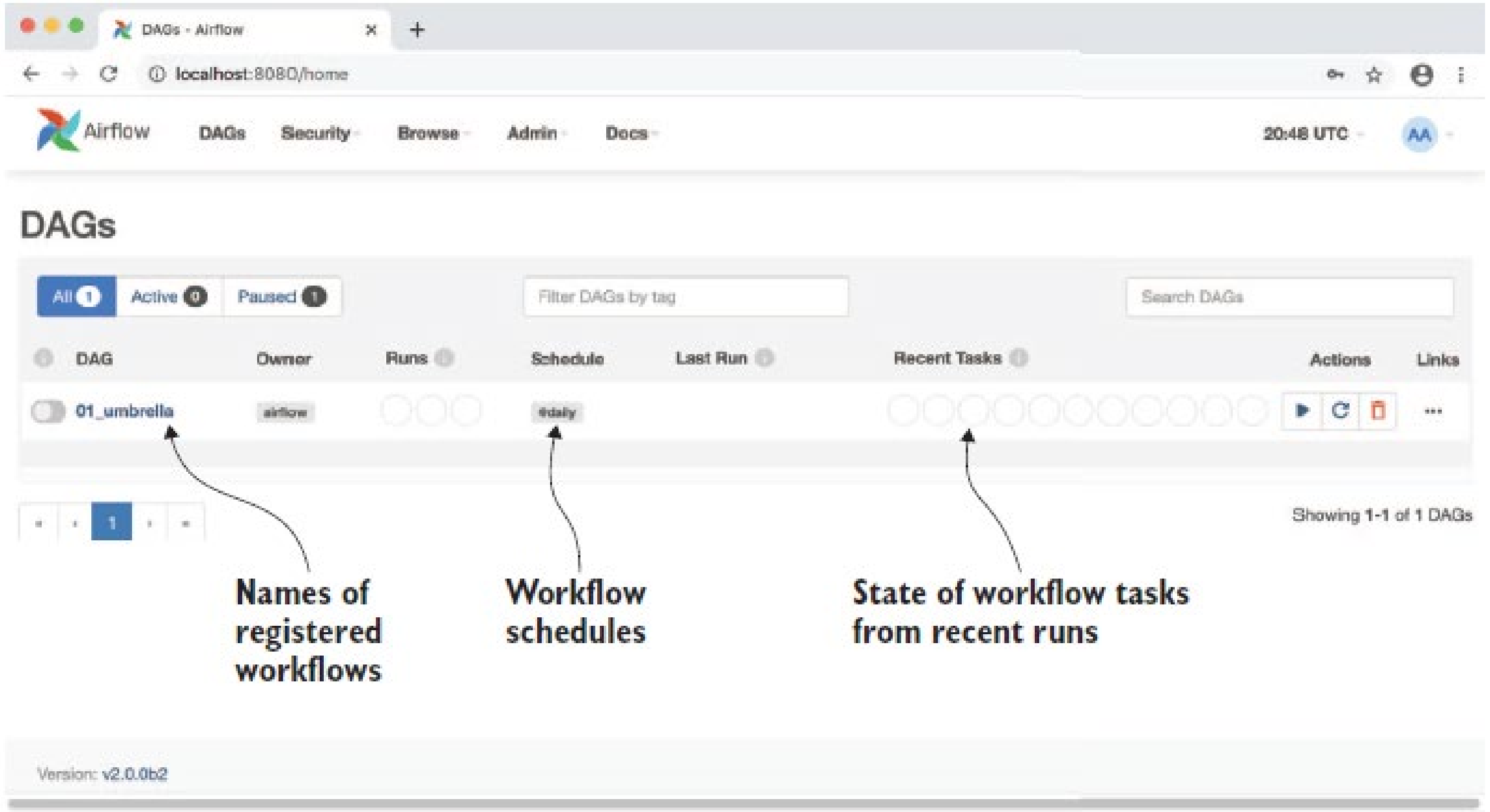

Users
Monitor DAG runs + results
Airflow webserver
Visualize DAGs + task results
Store task results
Airflow workers
Airflow metastore (database)
Store serialized DAGs
Execute tasks
Airflow scheduler
Schedule tasks
Queue
Read DAGs
Write data workflows in Python as Airflow DAGs
DAG folder
DAG files describing pipelines (in Python)

The schedule

- The heart of Airflow to run the following steps:
    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.

Monitoring and handling failures

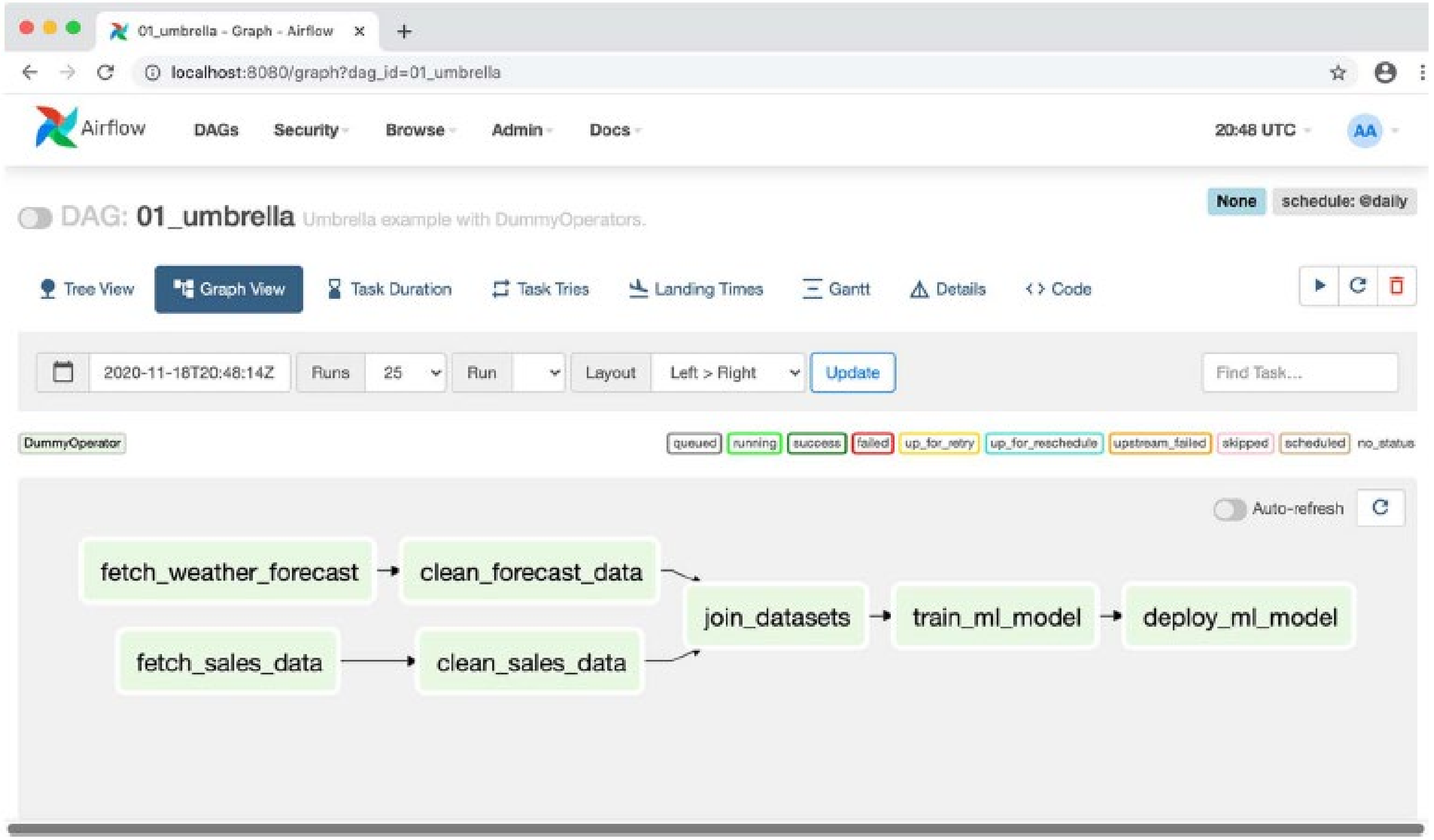
- Airflow provides web interface for viewing DAGs and monitoring the results of DAG runs



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

Monitoring and handling failures

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


localhost:8080/graph?dag_id=01_umbrella

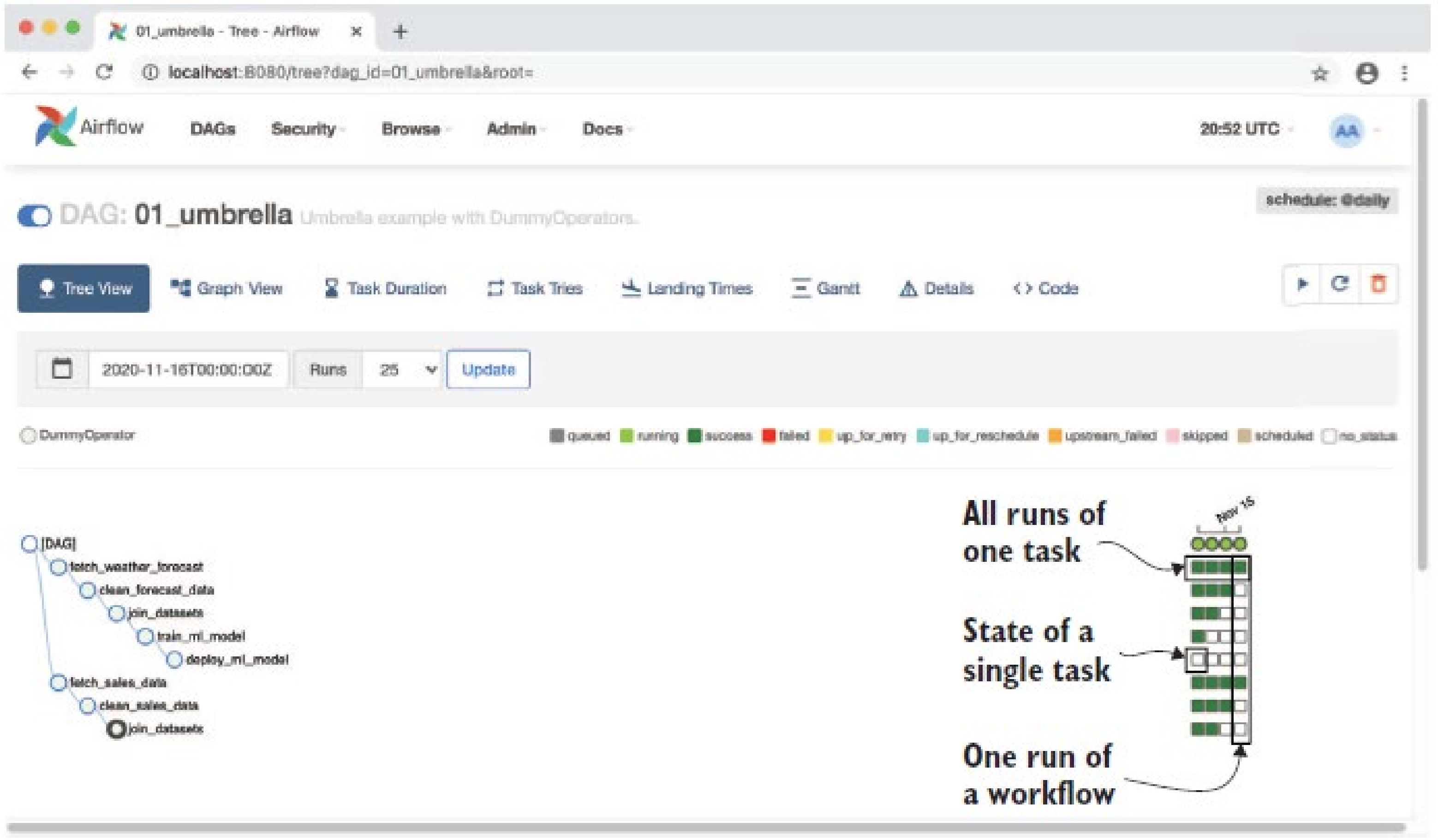
DAG: 01_umbrella Umbrella example with DummyOperators.

None schedule: @daily

2020-11-18T20:48:14Z Runs 25 Run Layout Left > Right

Monitoring and handling failures

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 (which you can see in the e-book version) indicate the result of the corresponding task. Users can also click on the task “squares” for moredetails about a given task instance, or to reset the state of a task so that it can be rerun by Airflow, if desired.



Incremental loading and backfilling

- Incremental loading:
  - Scheduling defines the intervals to run DAGs
  - So scheduling semantics provide details about the past and future schedule intervals
  - Each DAG is run in each intervals
  - So can build incremental pipelines: only the data at that time point is processed – don't need to run the entire data
- Backfilling:
  - running a DAG (Directed Acyclic Graph) for historical dates, effectively re-running tasks to process data for a specific past period.

Reasons to choose Airflow

- 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.

Reasons not to choose Airflow

- 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.

Reference

All the materials were taken from the following book:

- Chapter 1 in "*Data Pipelines with Apache Airflow*", by Bas Harenslak & Julian de Ruiter, April 2021, ISBN 9781617296901, Manning Publications