Lecture Notes

Building Automated Data Pipelines with Apache Airflow

Data Pipelines

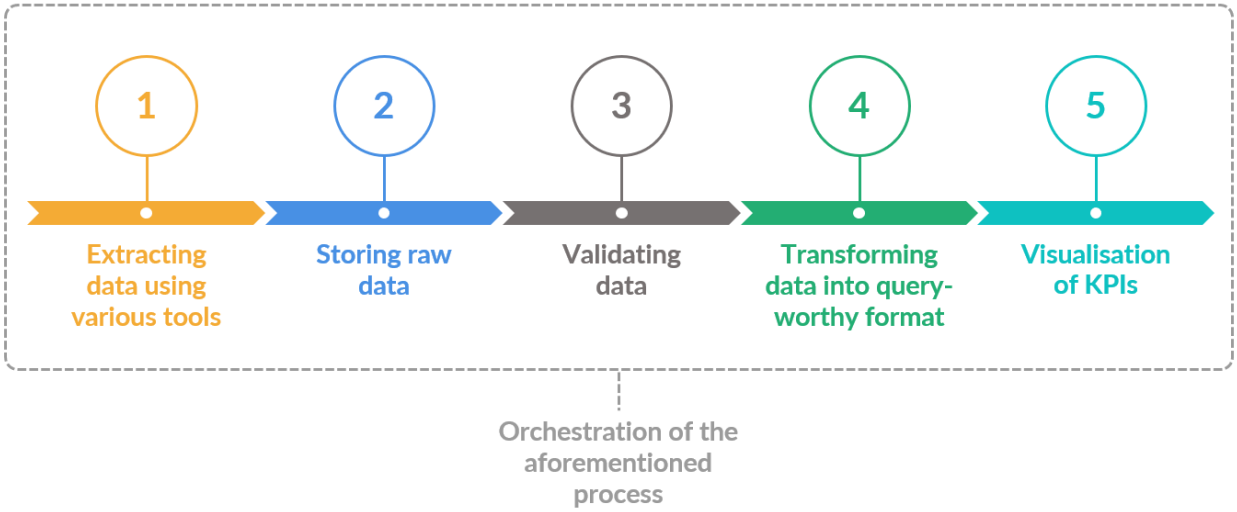
Before starting a discussion on the ways to automate large-scale data pipelines using Airflow, let's get a better understanding of data pipelines and what a typical data pipeline looks like.

Databases that are generally efficient for Online Transactional Processing (OLTP) are not best suited for Online Analytical Processing (OLAP). The first step in a scalable solution for data processing is to move the data from OLTP databases to OLAP systems.

This is where data pipelines come to our rescue.

We use data pipelines to move data from one system to another and possibly transform and validate this data in the process.

The phases involved in a typical data pipeline can be seen in the following graphic:



User Data Pipeline

Now, let's look at a real-life example to understand why these pipelines must be automated or orchestrated.

As you can see in the following diagram, the phases discussed in the earlier segment — Extraction > Storing raw data > Validating > Transforming > Visualising — can also be seen in the Uber example.

Following are the reasons for automating such pipelines:

- It can send notifications when processed data is available or if something fails.
- The reduced manual effort allows companies to focus on business logic.
- Tracking the performance of the different steps in the pipeline helps companies identify and resolve bottlenecks.
- Automating data pipelines allows the company to collect, process and economically use data in real time.

How to Automate a Data Pipeline?

Let's look at some possible solutions for data pipeline automation, such as manual orchestration, Cron jobs and Apache Oozie.

Manual automation is a very inefficient and impractical process in an industrial setting.

Cron jobs are easy to use/learn but are incapable of handling complex data pipelines. Moreover, these lack some essential features that a data orchestration tool must have.

Apache Oozie is a tool capable of handling large-scale data pipelines, but it has a steep learning curve and is becoming increasingly outdated every day.

As none of the above-mentioned solutions offers what we require, there is a need for a new automation tool with the following features:

- Scalable to orchestrate and schedule a large number of processes
- Efficient maintenance and monitoring of pipelines
- Richer UI to visualise the status and compare it against historical runs
- Retry and timeout feature for failed tasks
- Easy SLA handling
- Compatibility with various tools in the data engineering domain
- Strong community support

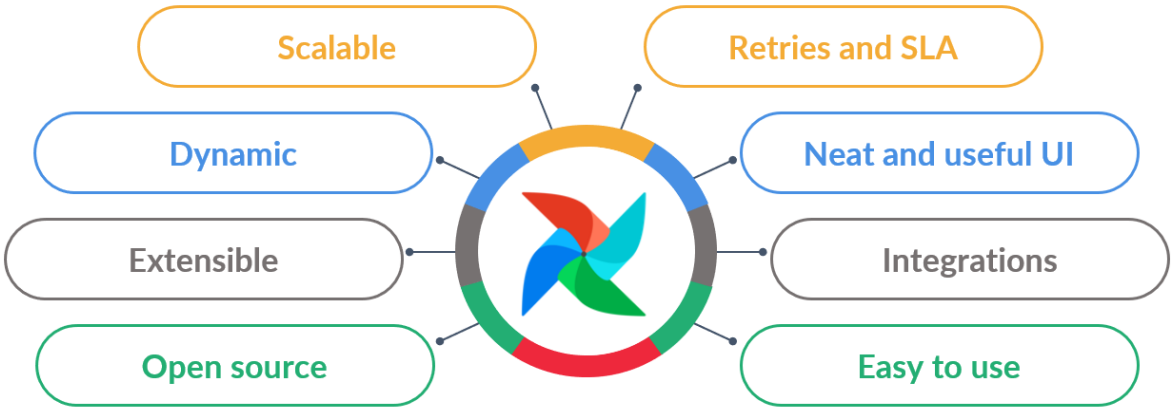
Introduction to Apache Airflow

The Airflow platform was built by Maxime Beauchemin at Airbnb in October 2014. The tool was built to address the company's increasing complex workflows or data pipelines and monitoring needs. It can programmatically create, schedule and monitor data pipelines.

Data pipelines in Airflow are:

- Implemented in the form of Directed Acyclic Graphs (DAGs),
- Created using Python code, and
- Can be generated dynamically.

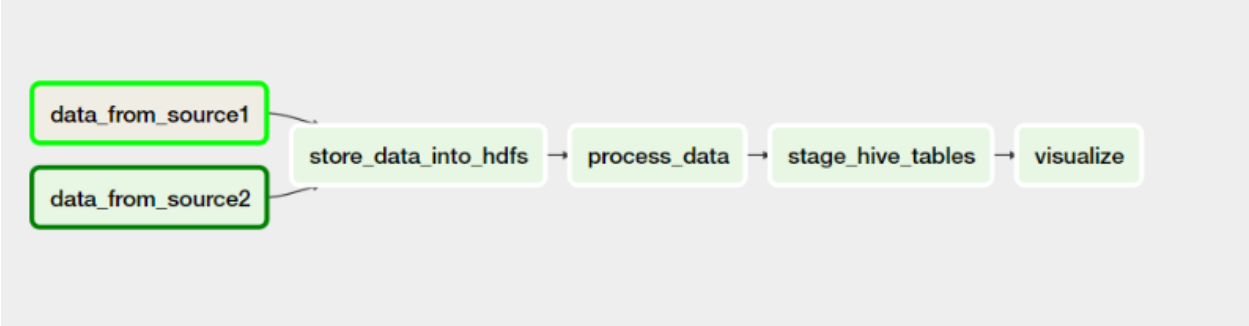
Following are some prominent features of Airflow:



DAGs : Data Pipelines in Airflow

DAGs or Directed Acyclic Graphs are what Airflow uses to implement its data pipelines.

The following diagram shows what a very simple DAG in Airflow looks like:



The properties of a DAG are as follows:

- It should be directed (i.e., all the edges have a direction indicating which task is dependent on which).
- It must be acyclic (i.e., it cannot contain cycles).
- DAGs are graph structures (i.e., a collection of vertices and edges).

Next, let's see how a DAG can be defined in Airflow.
As seen in the following sample code, DAGs are written in Python.

```
#Import modules for the DAG
```

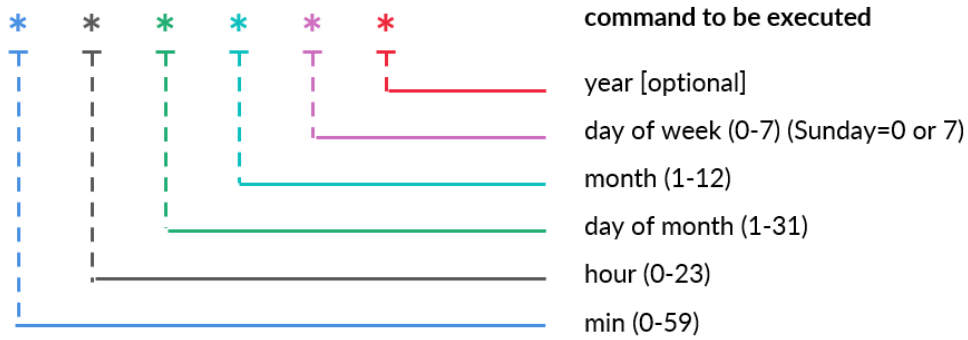
```
from datetime import datetime
from airflow import DAG

#Define a dictionary with the default arguments of a DAG
dag_default_configs = {
     'start_date': datetime(2016, 1, 1),
     'owner': 'airflow'
}

#Create DAG object using the DAG()
dag_object = DAG('my_dag',
           default_args = dag_default_configs,
           description='Sample DAG',
           schedule_interval='0 12 * * *')
```

As you can see in the sample code above, some important attributes of the DAG are as follows:

- ID - `'my_dag'`
- Description - `'Sample DAG'`
- Schedule - `'0 12 * * *'`
- Start Date - `datetime(2016, 1, 1)`
- Configs/default arguments - `dag_default_configs`

These are the important attributes of a DAG. You can specify many more such attributes to better address your requirements.

The schedule argument can be defined using a Cron expression. Following is the guide for the same:



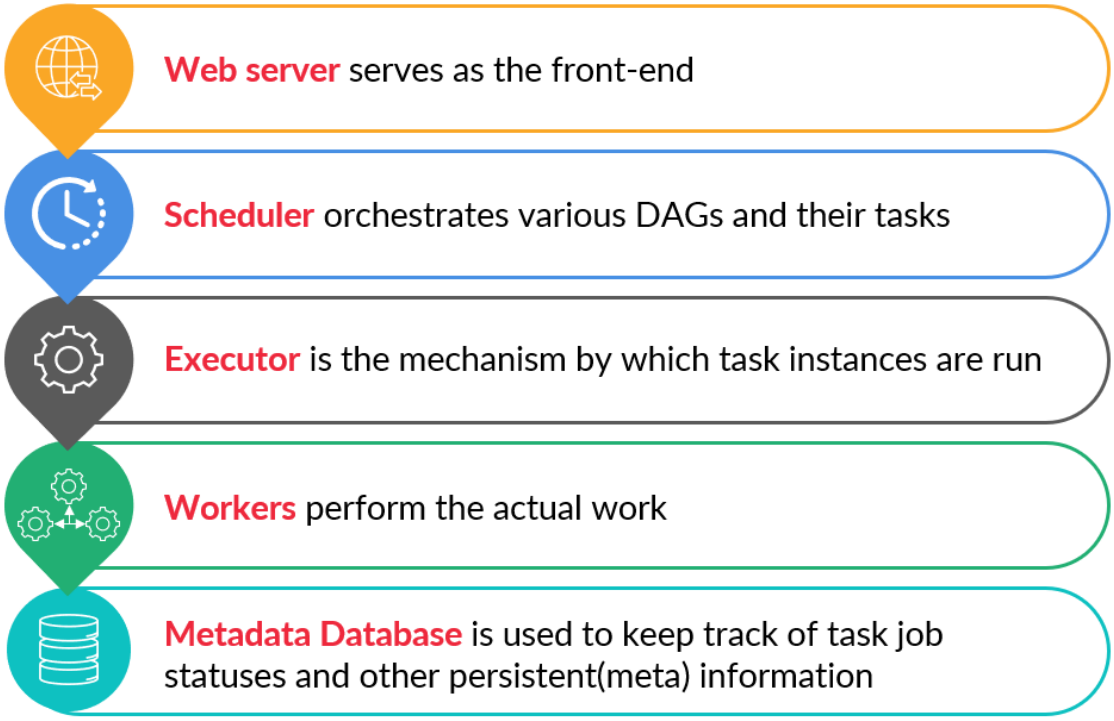
Finally, below are the components that make up a DAG:

- Task - It defines a unit of work within a DAG.
- Task dependencies - They define the order in which the tasks in a DAG are executed.

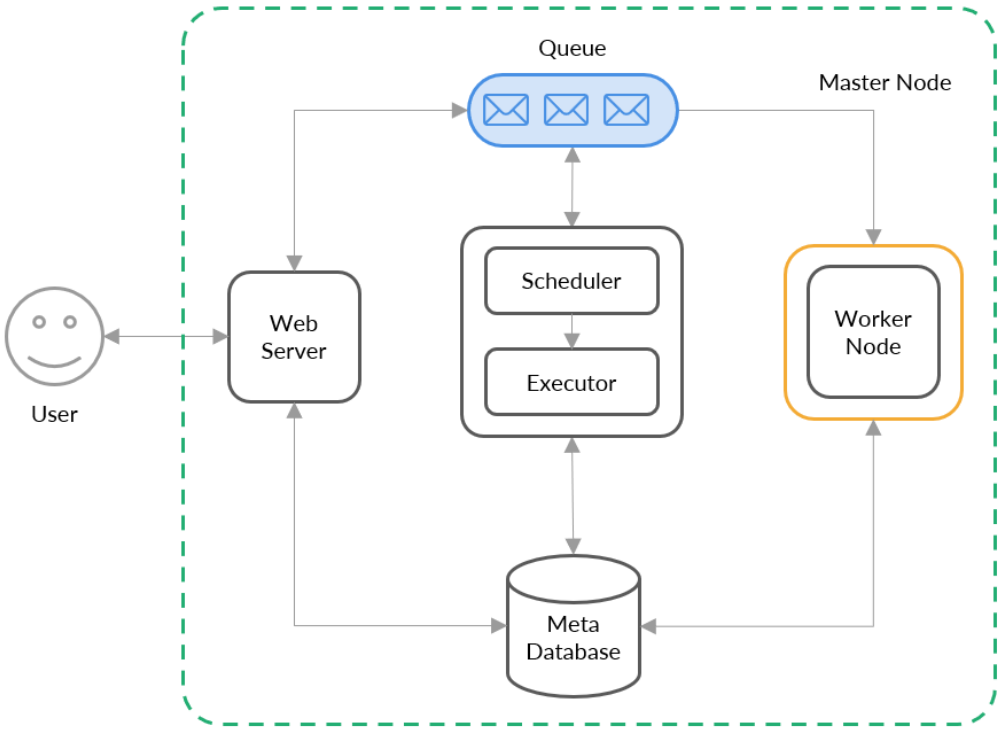
‘

Airflow Architecture

The diagram below shows the various components that make up Airflow's architecture:



The following graphic illustrates a single-node architecture:



A single-node architecture is quite simple and needs only one node, but it is impractical in a production scenario. Thus, Airflow also allows a distributed architecture with the use of different executors.

Some of the executors provided by Airflow are listed below:

Sequential Executor:

- It is the default executor and only runs one task instance at a time.
- It is suitable for testing and debugging DAGs before they are implemented in an industrial environment.

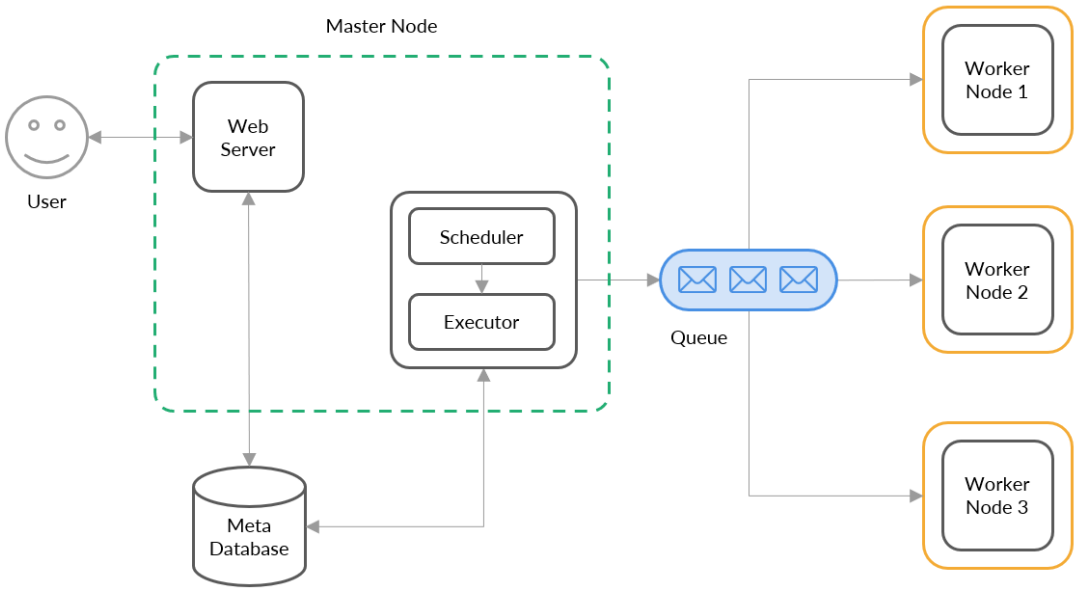
Local Executor:

- It is like a sequential executor with unlimited parallelism.
- It runs tasks by spawning processes in a controlled fashion in different modes.

Celery Executor:

- It is used in scalable environments.
- It needs RabbitMQ and Redis for configuration.
- Each worker is in a different node. So, the number of workers can be easily scaled by adding more nodes.
- It is recommended in a production scenario.

Below, you can see the multi-node/distributed architecture used in the industry:



Airflow Configurations

You can refer to the table below to learn about some important configurations in the airflow.config file.

Config Name	Description	Sample Values
dags_folder	• The folder where your airflow pipelines live • This path must be absolute	/home/ubuntu/airflow/dags
executor	• The type of executor that airflow should use. • Choices include SequentialExecutor, LocalExecutor, CeleryExecutor	CeleryExecutor
sql_alchemy_conn	• The SqlAlchemy connection string to the metadata database	postgresql+psycopg2://airflow:airflow@localhost/airflow
parallelism	• The amount of parallelism as a setting to the executor • This defines the max number of task instances that should run simultaneously for the airflow installation	32
dag_concurrency	• The number of task instances allowed to run concurrently by the scheduler	16
max_active_runs_per_dag	• The maximum number of active DAG runs per DAG	16
base_url	• The base URL of your website as airflow cannot guess what domain or cname you are using.	http://localhost:8080
broker_url	• The Celery broker URL. Celery supports RabbitMQ, Redis and experimentally a sqlalchemy database.	redis://127.0.0.1:6379/1

Airflow Operators

Operators describe a single task in a workflow, which can be a Shell script, a Hive query, a Python function, and so on. Different operators have different attributes, but certain arguments are common to all operators, for example, the **task_id (name of the task)** and **dag (name of the dag object the task belongs to)** arguments.

Some important operators are listed below:

- The **BashOperator** is used for running any Shell command or script with your DAG.
- The **PythonOperator** is used to execute Python callables.
- The **SqoopOperator** is usually used to transfer data between RDBMS and HDFS. Essentially, you can perform any Sqoop using this operator.
- The **HiveOperator** is used to connect to Hive using hive_conn_id and execute Hive queries.
- The **SparkSubmitOperator** launches Spark using the spark-submit CLI on the Airflow machine.
- The **EmailOperator** is used for alerting task events, such as task completion, task failure, task retry and task fail on retry. It is also used to send attachments containing error messages or logs along with emails.
- The **BaseOperator** is the abstract base class for all the operators; thus, the data members and methods declared in this class are available to all the operators.

Operators are like blueprints based on which tasks are built on. These tasks are the basic units of work in a DAG and the relationships between these tasks are defined by the task dependencies.

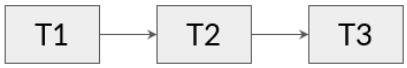
Listed below is a guide to defining task dependencies:

If task T1 should finish before T2 starts, you can do this using:

- T1.set_downstream(T2)
- T1 >> T2
- T2.set_upstream(T1)
- T2 << T1

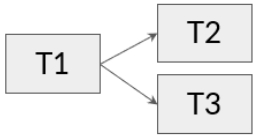
You can also define a chain of dependencies at once:

- T1 >> T2 >> T3 instead of T1 >> T2 and T2 >> T3



You can set multiple dependencies at once:

- T1 >> [T2, T3]



Best Practices

Some tips and best practices for Airflow are listed here:

- Do not use Airflow to debug your application code.
- The start date and the actual execution of a DAG differ by one schedule interval. Consider this while creating/scheduling your DAG.
- Never store any config file in the local file system. If possible, use XCOM to communicate small messages.
- By default, DAGs are turned off as a cautionary measure. Ensure that you turn them ON.
- Sensors are mostly idle processes. So, do not use them unnecessarily.
- Use static start_date for the DAGs.

- Ensure that DAGs are as independent as possible.
- Make DAGs idempotent (running them multiple times is the same as running them once).
- Renaming a DAG will introduce a new DAG. You can use this to back up your DAGs.
- Run different components on different machines.

Disclaimer: *All content and material on the upGrad website is copyrighted material, belonging to either upGrad or its bona fide contributors, and is purely for the dissemination of education. You are permitted to access, print, and download extracts from this site purely for your own education only and on the following basis:*

- *You can download this document from the website for self-use only.*
- *Any copies of this document, in part or full, saved to disc or to any other storage medium, may be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.*
- *Any further dissemination, distribution, reproduction, and copying of the content of the document herein, or the uploading thereof on other websites, or use of the content for any other commercial/unauthorized purposes in any way that could infringe the intellectual property rights of upGrad or its contributors is strictly prohibited.*
- *No graphics, images, or photographs from any accompanying text in this document will be used separately for unauthorized purposes.*
- *No material in this document will be modified, adapted, or altered in any way.*
- *No part of this document or upGrad content may be reproduced or stored on any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.*
- *Any rights not expressly granted in these terms are reserved.*