

English version

Cloud Computing workflows with Apache AirFlow

What is Apache Airflow used for?	2
What is a Cloud Computing workflow?	3
Why use AirFlow or a Cloud Computing Workflow tool?	5
Installation of Apache AirFlow	6
Planner Initialization	8
Initialization of the web service and order line for workflow management	8
Creating a Workflow in AirFlow	9
Operators or types of tasks in the workflow	12
Complete example of multi-operator workflow	13
References	18

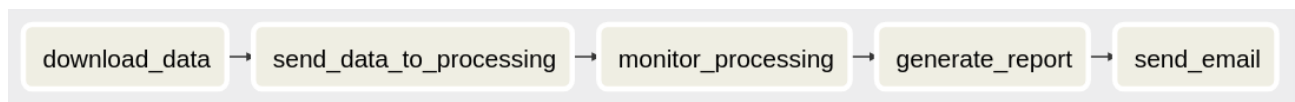
What is Apache Airflow?

Airflow (<https://airflow.apache.org/>) is a platform for programming and monitoring workflows. It is used to create workflows composed of tasks whose dependencies can be modeled using Directed Acyclic Graphics (DAG). Airflow integrates a "scheduler" that executes the tasks in a set of nodes/instances called "workers". It allows you to manage all the workflows from a Shell interface or from an administration web. This web allows the visualization of the data flows that are executed in the production, the control of the progress and the management of problems with data and the workflows on a large scale.

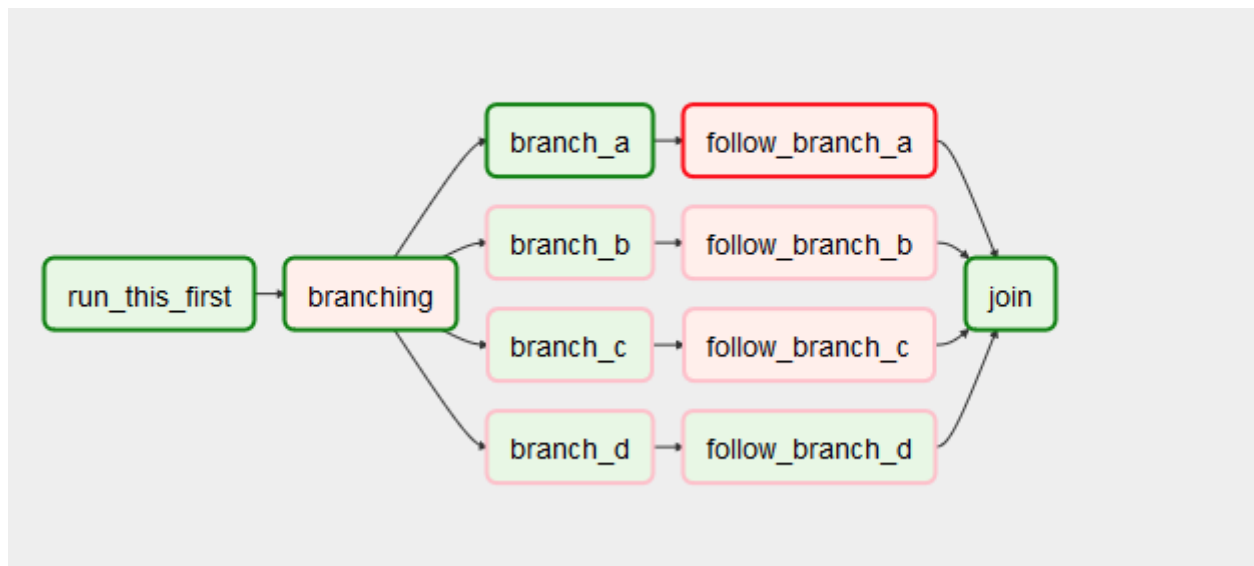
What is Apache Airflow used for?

If we reduce AirFlow to a minimum, basically AirFlow, it helps to automate "things" to perform tasks (of any kind). It's made in Python, but you can run a program regardless of the language. For example, the first stage of your workflow you have to run a C++ based program to perform the image analysis and then a Python based program to transfer that information to a cloud service to publish results. The possibilities are endless.

You can do simple things like the one shown in the image below (download data, send data, monitor processing, create a report, and then mail the results)



to other much more complete ones that require intensive distributed calculation as in the image below, where it is used in other software projects that are very relevant today, such as Spark and Tensorflow, to create execution models:



Some of the highlights of the Airflow are as follows:

Management of source failures:

Integrated management of errors in data access It is possible to define a different behavior for each case.

Reprocessing of historical jobs:

Ordering past jobs to be reprocessed directly from the GUI in a very simple way.

Parameters between jobs:

interchange of parameters between different jobs through a buffer. This allows us to assign states to the workflows.

Automatic retries:

Automatic retry management based on the configuration of each DAG.

CI and CD support in the workflows:

Easily integrating Airflow with our integration or continuous deployment systems either by using tools like Jenkins or by managing the DAGs in GitHub.

Many integrations:

Airflow can be integrated with a large number of third-party platforms and software through the operators. Many of them are community contributions: Hive, Presto, Druid, AWS, Google Cloud, Azure, Databricks, Jenkins, Kubernetes, Mongo, Oracle, SSH, etc. so we can transparently use those resources as tasks within the processing.

Data sensors:

are a particular type of operator that allows us to wait until a certain condition is met, for example, that a file is written in HDFS or S3.

Test work capabilities:

allows us to test and validate our tests before deployment.

Task triggers:

there are different ways in which we can launch our workflows to be executed automatically according to a schedule; we can also execute them by hand.

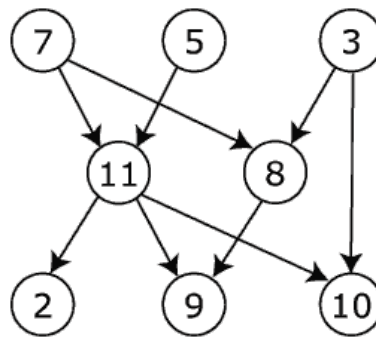
RT monitoring and alerts:

we can monitor the execution status of our workflows in real time from the graphical interface

What is a Cloud Computing workflow?

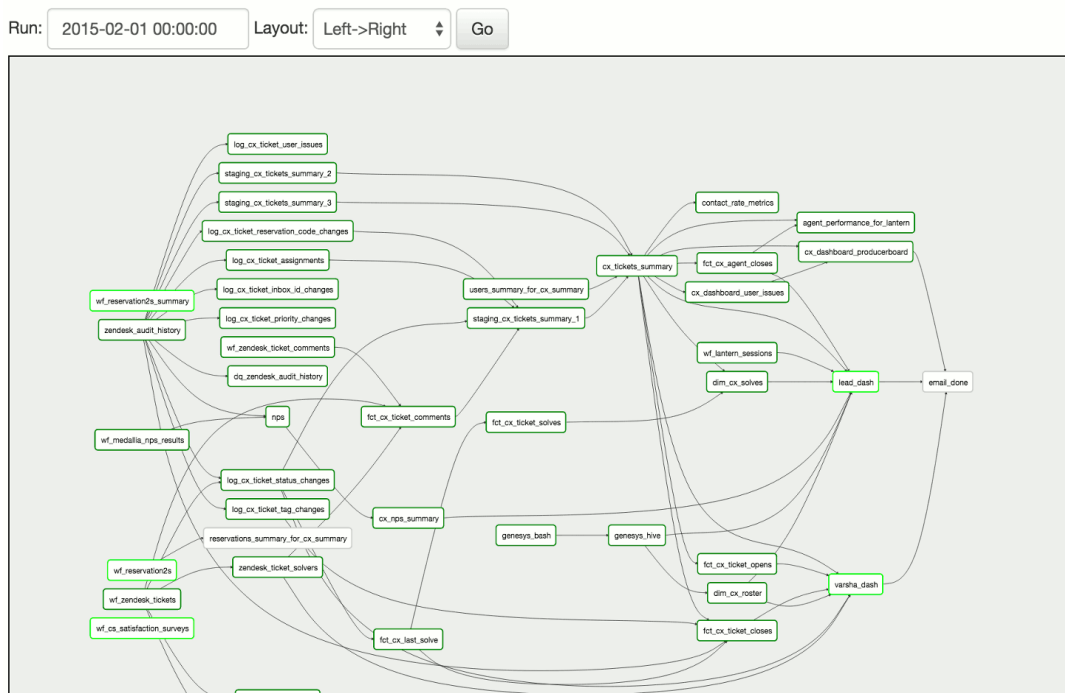
A workflow can be represented as a Directed Acyclic Graph (DAG). This is a directed network (the arcs have a defined direction) that does not have cycles. This means that for each v-verse, there is not one path that begins and another that ends in v.

In this DAG example, the tasks that start are (7,5,3) and could be done in parallel.



When we refer to a workflow in Cloud Computing, we mean tasks to be performed by means of cloud services whose interdependencies can be expressed by means of a DAG. This allows to collect all the possibilities that usually arise one or more sequential tasks, parallel or a mixture of both concrete will be to deploy a service in Cloud Computing following all key aspects of the concept of this paradigm. A philosophy of architectures of special interest is Cloud Native. The philosophy focuses on how applications are created and deployed, but not where, exploiting the advantages of the deployment model offered by Cloud Computing. A key aspect to understand the Cloud Native model is that both the creation, deployment, authoring, and processing of all components from the time we have the source code until we have the final product or service running in the Cloud, corresponds to a series of operations seen as a DAG.

In the image below you can see a DAG with several dozen operations corresponding to the entire life cycle of a Cloud Native service using AirFlow workflows, where it supports virtually any type of operation, task, or functionality in each of the nodes.



Why use AirFlow or a Cloud Computing Workflow tool?

The answer to this question is based on these four principles:

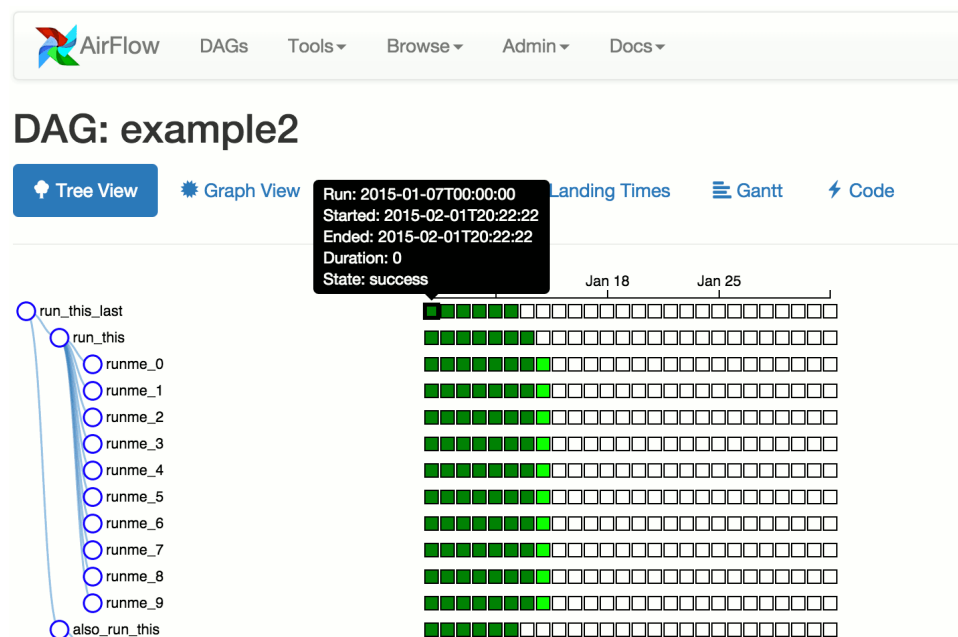
Dynamic: The data flows are configured as code (Python, Scala, R, Java, JS, ...), which allows the generation of dynamic flows. This allows you to write code that instantiates tasks dynamically.

Extensible: Allows you to easily define your own operators, executors and extend the libraries to fit the level of abstraction that suits your environment.

Elegant: Data flows are simple and very explicit.

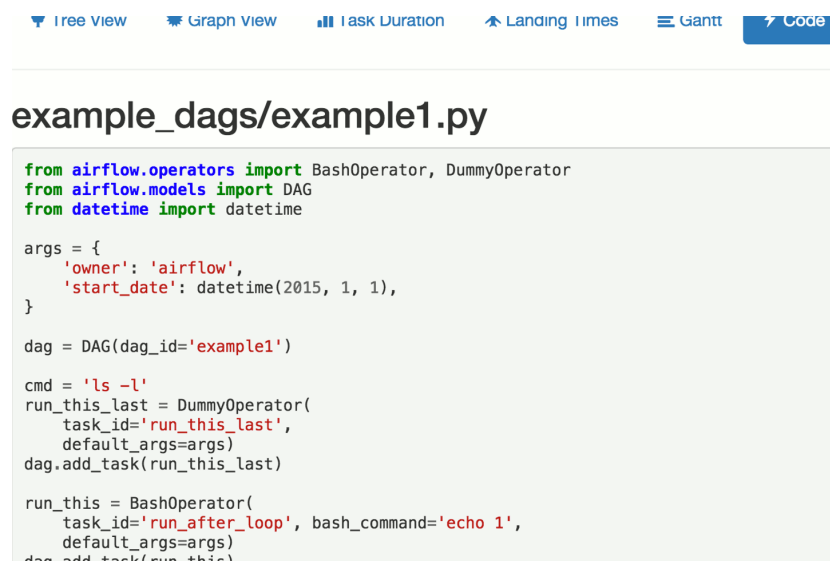
Scalable: Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. Airflow is ready to scale without limits.

In the following image you can see the status of the execution of a task DAG in as a workflow for each of the operations, in addition to the temporal sequencing of the same:



Another key element for using AirFlow in Cloud Computing is that it allows workflows to be coded as code, which gives a lot of versatility when it comes to programmatically creating these types of flows from different APIs.

In the following image you can see an example of how to implement a workflow in Cloud Computing, with different operators (or tasks), for Python, for example:



```
from airflow.operators import BashOperator, DummyOperator
from airflow.models import DAG
from datetime import datetime

args = {
    'owner': 'airflow',
    'start_date': datetime(2015, 1, 1),
}

dag = DAG(dag_id='example1')

cmd = 'ls -l'
run_this_last = DummyOperator(
    task_id='run_this_last',
    default_args=args)
dag.add_task(run_this_last)

run_this = BashOperator(
    task_id='run_after_loop', bash_command='echo 1',
    default_args=args)
dag.add_task(run_this)
```

Installation of Apache AirFlow

Installing AirFlow is simple, as it installs like any package in Python from your system's command line and is therefore independent of the platform (hardware or operating system) being used, whether it be Mac, Linux or Windows. The installation is done from a shell (or Linux terminal, or from PowerShell if you use Windows).

- To install Airflow, you need:
 - Python >= 3.6 and pip3 or pip installed with Python (try `python -v` to know your version).
 - In case you don't have Python in version 3.6, download CONDA to switch between version 2.7 and 3.X
https://salishsea-meopar-docs.readthedocs.io/en/latest/work_env/python3_conda_environment.html

Steps for quick installation (in Linux):

The steps for installing Airflow on a Linux computer are simple. From any of the Linux distributions, open a Shell/Terminal window:

Step one, install the software with PIP:

```
pip install apache-airflow
```

Or if you have pip3 use:

```
pip3 install apache-airflow
```

Second step, initialize the database:

```
airflow initdb
```

With these steps AirFlow is installed

Other more advanced installation options:

The easiest way to install the latest stable version of the Airflow is with pip:

```
pip install apache-airflow
```

You can also install Airflow with support for additional features such as gcp or postgres depending on the needs of your system:

```
pip install apache-airflow [postgres,gcp]
```

Or if you need support for Kubernetes, AWS, Azure, etc. we use the following:

```
pip install 'apache-airflow [aws]'
```

```
pip install 'apache-airflow [azure]'
```

```
pip install 'apache-airflow [kubernetes]'
```

It has packages/tasks for almost any Cloud environment we want to deploy, here are all the supported ones: <http://airflow.apache.org/docs/stable/installation.html#extra-packages>

We will install AirFlow with :

```
pip install apache-airflow
```

Once this is done, we need to initialize the database that controls the entire AirFlow system. To do this, by default we will use the one it has (it is SQLite), although it is possible to use other DBMS such as MySQL, PostgreSQL, etc. Therefore we use:

```
airflow initdb
```

At this moment we already have AirFlow working, now we just need to review some additional concepts.

Planner Initialization

The AirFlow Workflow Scheduler monitors all tasks and DAGs, and activates the instances of tasks whose dependencies have been fulfilled. It actually instantiates a sub-process, which monitors and stays in sync with a folder for all the DAG objects it may contain, and periodically (every minute or so or as configured) collects the results of the DAG analysis and inspects the active tasks to see if they can be activated.

The scheduler is designed to function as a persistent service in a production environment, so it should always be started whenever you want to run your workflows. To start it, all you need to do is run the AirFlow Scheduler. It will use the configuration specified in `airflow.cfg`:

```
airflow scheduler
```

To run this service and to be able to consult the operation messages it is preferable to execute the start command in a shell (console window) independent from another one from which you interact with AirFlow.

In the following sections we explain how to use AirFlow through different examples and from several points of view regarding the way of interaction in workflow management with the platform.

Initialization of the web service and order line for workflow management

AirFlow can be managed from the command line (shell) or from a web interface.

To start the management service via web we use

```
airflow webserver -p 8080
```

Once this is done, we open a browser and request the following URL (from the same computer where the service is running):

```
http://localhost:8080
```

and access the AirFlow management system.

All the managements that are made from the web environment can be made from the Shell orders interface (through orders that the airflow program interprets and executes) as we see here:

```
airflow list_dags
```

It shows the set of workflows available in the system.

```
airflow list_tasks <DAG_ID>
```

Displays the task set of a workflow given its DAG_ID

```
airflow task_state <DAG_ID> <TASK_ID> <execution_date>
```

Displays the status of a task in a workflow given its TASK_ID, DAG_ID and execution_date

```
airflow list_dag_runs <DAG_ID>
```

Shows the set of executions of a workflow given its DAG_ID

```
airflow pause <DAG_ID>
```

For a given workflow your DAG_ID

```
airflow unpause <DAG_ID>
```

Restart a workflow given your DAG_ID

In the following sections we will show detailed examples of how a workflow is built programmatically, through different examples. The student must carry out all the tasks indicated in the creation of workflows and verifying that they are correct when they are introduced in the planner.

Creating a Workflow in AirFlow

The creation of workflows with AirFlow is done by creating a file with Python source code where all the key aspects of a DAG in AirFlow are defined. The structure is as follows:

- AirFlow and DAG libraries.

- Workflow configuration elements.

- Creation of the DAG.

Creation/definition of operators or tasks to be performed in the workflow.
Specifying the dependencies (network creation) between tasks.

This is a complete example in Python (call it dag_practica2.py), you can download it from:

<https://github.com/manuparra/MaterialCC2020/blob/master/airflow/p2example1.py>

Here is the code:

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': days_ago(2),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
    # 'pool': 'backfill',
    # 'priority_weight': 10,
    # 'end_date': datetime(2016, 1, 1),
    # 'wait_for_downstream': False,
    # 'dag': dag,
    # 'sla': timedelta(hours=2),
    # 'execution_timeout': timedelta(seconds=300),
    # 'on_failure_callback': some_function,
    # 'on_success_callback': some_other_function,
    # 'on_retry_callback': another_function,
    # 'sla_miss_callback': yet_another_function,
    # 'trigger_rule': 'all_success'
}

#Inicialización del grafo DAG de tareas para el flujo de trabajo
dag = DAG(
    'practica2_ejemplo',
    default_args=default_args,
    description='Un grafo simple de tareas',
    schedule_interval=timedelta(days=1),
)

# Operadores o tareas
# t1, t2 y t3 son ejemplos de tareas que son operadores bash, es decir comandos
t1 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

t2 = BashOperator(
```

```
task_id='sleep',
depends_on_past=False,
bash_command='sleep 5',
retries=3,
dag=dag,
)

t3 = BashOperator(
    task_id='new_file',
    depends_on_past=False,
    bash_command='ps ax > /tmp/procs.txt',
    dag=dag,
)

#Dependencias
t1 >> [t2, t3]
```

Once this is done, copy it to a file with a .py extension and save it in \$HOME/airflow/dags/ which is the place where AirFlow consults the workflows that exist in the system.

Once this is done we wait for AirFlow to process the workflow and load it into the scheduler. This operation takes a couple of minutes and depends on the configuration, by default every 30 seconds AirFlow checks if there are new flows created. Therefore you can see the new flow created from the web <http://localhost:8080> or using the command line with: `airflow list_dags` once it is processed.

Explanation of the workflow:

This workflow consists of 3 very simple tasks T1, T2, and T3, as they are shell operations, i.e. shell commands.

T1 → prints the current date → `bash_command='date'`,

T2 → use the 'sleep 5' command to sleep the process → `bash_command='sleep 5'`

T3 → creates a file in /tmp/procs.txt with the content of 'ps' → `bash_command='ps ax > /tmp/procs.txt'`.

To model the tasks it is done in this way:

`T1 >> [T2, T3]` means that first T1 is executed and then [T2, T3] waits for T2 and T3 to finish before starting both.

Another way to model them would be:

`T1 >> T2 >> T3` where it would be a sequence of tasks starting in T1 and ending in T3

In this way you can model any DAG network structure you want with different combinations of that type.

Operators or types of tasks in the workflow

An operator represents a single task, ideally independent. The operators determine what is actually executed when the DAG is executed. They are the minimum units of execution. While DAGs describe how to execute a workflow, operators determine what is actually done with a task.

An operator describes a single task in a workflow. Operators are usually (but not always) atomic, which means that they are self-sufficient and do not need to share resources with other operators. The DAG will make sure that the operators operate in the correct order; apart from these dependencies, operators usually operate independently. In fact, they can operate on two completely different machines.

With AirFlow, the following operators are available for many common tasks, including

1. BashOperator - executes a bash command
2. PythonOperator - calls an arbitrary Python function
3. EmailOperator - send an email
4. SimpleHttpOperator - sends an HTTP request
5. MySQLOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - Execute a SQL command
6. Sensor - an Operator waiting (polling) for a certain time, file, a row in the database, key in S3, etc...

We will work with BashOperator and PythonOperator during the training.

Bash Operator -- BashOperator

Allows you to execute a shell command, so any command, macro, script or application can be launched with this operator. To define it inside a DAG file we will use

```
Tarea1 = BashOperator(  
    task_id='primera_tarea',  
    bash_command='echo 1',  
    dag=dag,  
)
```

Or more complex:

```
BuscaFich1G = BashOperator(  
    task_id='primera_tarea',  
    bash_command='echo 1',  
    dag=dag,  
)
```

```
task_id='buscar_ficheros',  
bash_command='find /path -mtime +180 -size +1G',  
dag=dag,  
)
```

Operator Python -- PythonOperator

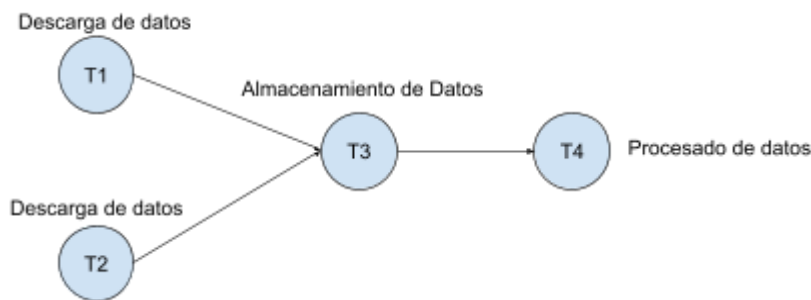
With the Python operator we can run any python service, code, function or library that is available or that we have developed or coded. Therefore, it is possible to access databases, process data with libraries such as Pandas, ScikitLearn, etc, or perform any operation or process from Python code.

```
def imprime(ds, **kwargs):  
    pprint(kwargs)  
    print(ds)  
    return 'Lo que devuelvas se imprime en los logs'  
  
tarea1 = PythonOperator(  
    task_id='imprime_contexto',  
    provide_context=True,  
    python_callable=imprime,  
    op_kwargs={'parameter1': 10},  
    dag=dag,  
)
```

Complete example of multi-operator workflow

For this example we are going to mix all the aspects we have seen so far, on one hand the operators and on the other the dependencies. Here is the complete code of the example:
<https://github.com/manuparra/MaterialCC2020/blob/master/airflow/p2example2.py>

This workflow will do the following:



T1 → BashOperator: CURL download data

T2 → BashOperator: CURL download data

T3 → BashOperator: MV move to the datastore

T4 → PythonOperator, mix both datasources

Here we define all the tasks:

T1:

```

DescargaDatos1 = BashOperator(
    task_id='descarga1',
    bash_command='curl -o /tmp/partel.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Tax
i-Trip-Data/biws-g3hs',
    dag=dag,
)

```

T2:

```

DescargaDatos2 = BashOperator(
    task_id='descarga2',
    bash_command='curl -o /tmp/parte2.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Tax
i-Trip-Data/biws-glhs',
    dag=dag,
)

```

T3:

```

MoverDataStore = BashOperator(
    task_id='mover_aDataStore',
    bash_command='cat /tmp/partel.csv <(tail +2
/tmp/parte2.csv) > /tmp/DataStore/output.csv',
)

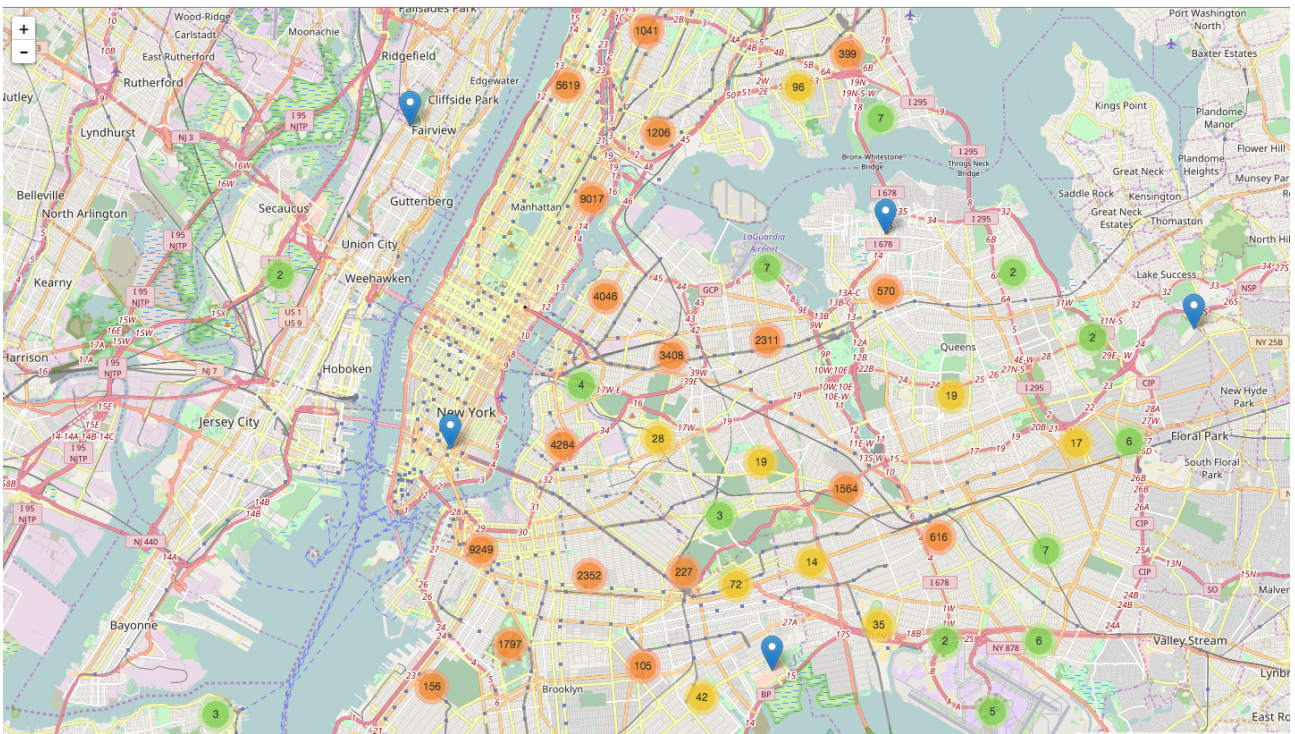
```

```
        dag=dag,  
    )
```

T4:

```
# Estas bibliotecas deberían ser instaladas previamente  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from datetime import datetime  
import os  
import folium  
from folium.plugins import MarkerCluster  
  
def generaMapa():  
    NY_COORDINATES = (40, -73)  
    gdata = pd.read_csv('green_tripdata.csv')  
    MAX_RECORDS = 1048576  
    map_nyctaxi = folium.Map(location=NY_COORDINATES, zoom_start=9)  
    marker_cluster = folium.MarkerCluster().add_to(map_nyctaxi)  
    for each in gdata[0:MAX_RECORDS].iterrows():  
        folium.Marker(  
            location =  
            [each[1]['Pickup_latitude'],each[1]['Pickup_longitude']],  
            popup='picked here').add_to(marker_cluster)  
    map_nyctaxi  
  
CreaVisualizacion = PythonOperator(  
    task_id='extrae_mapa',  
    provide_context=True,  
    python_callable=generaMapa,  
    dag=dag,  
)
```

This T4 operation (CreaVisualization) will generate a web service where the data is already processed and the following map will appear in the output of this final task:



Therefore with this workflow, we take data from different sources, then store them, merge them and perform a visualization of the data.

Once the tasks are defined, we have to build the network with the same operations of the previous drawing, so it is indicated:

```
[DescargaDatos1, DescargarDatos2] >> MoverDataStore >>
CrearVisualizacion
```

```
#          T1                      T2                      T3                      T4
```

So the code will be the next:

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import days_ago

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': days_ago(2),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
    # 'queue': 'bash_queue',
```



```
# 'pool': 'backfill',
# 'priority_weight': 10,
# 'end_date': datetime(2016, 1, 1),
# 'wait_for_downstream': False,
# 'dag': dag,
# 'sla': timedelta(hours=2),
# 'execution_timeout': timedelta(seconds=300),
# 'on_failure_callback': some_function,
# 'on_success_callback': some_other_function,
# 'on_retry_callback': another_function,
# 'sla_miss_callback': yet_another_function,
# 'trigger_rule': 'all_success'
}

#Inicialización del grafo DAG de tareas para el flujo de trabajo
dag = DAG(
    ' practica2_ejemplo_Mapas',
    default_args=default_args,
    description='Varias tareas mas elaboradas',
    schedule_interval=timedelta(days=1),
)

# Operadores o tareas
# t1, t2 t3 y t4 son ejemplos de tareas que son operadores bash, es decir comandos
DescargaDatos1 = BashOperator(
    task_id='descarga1',
    bash_command='curl -o /tmp/partel.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Taxi-Trip-Data/biws-g3hs',
    dag=dag,
)

DescargaDatos2 = BashOperator(
    task_id='descarga2',
    bash_command='curl -o /tmp/parte2.csv
https://data.cityofnewyork.us/Transportation/2017-Yellow-Taxi-Trip-Data/biws-glhs',
    dag=dag,
)

MoverDataStore = BashOperator(
    task_id='mover_aDataStore',
    bash_command='cat /tmp/partel.csv <(tail +2 /tmp/parte2.csv) > /tmp/DataStore/output.csv',
    dag=dag,
)

def generaMapa():
    NY_COORDINATES = (40, -73)
    gdata = pd.read_csv('green_tripdata.csv')
    MAX_RECORDS = 1048576
    map_nyctaxi = folium.Map(location=NY_COORDINATES, zoom_start=9)
    marker_cluster = folium.MarkerCluster().add_to(map_nyctaxi)
    for each in gdata[0:MAX_RECORDS].iterrows():
        folium.Marker(
            location = [each[1]['Pickup_latitude'],each[1]['Pickup_longitude']],
            popup='picked here').add_to(marker_cluster)
    map_nyctaxi

CreaVisualizacion = PythonOperator(
    task_id='extrae_mapa',
```

```
provide_context=True,  
python_callable=generaMapa,  
dag=dag,  
)
```

```
#Dependencias - Construcción del grafo DAG  
[DescargaDatos1, DescargarDatos2] >> MoverDataStore >> CrearVisualizacion
```

Finally we copy this file to the AirFlow workflow deployment directory for the planner to process and start the work developed.

References

Apache AirFlow:

<http://airflow.apache.org/>

How to start with AirFlow:

<http://airflow.apache.org/docs/stable/>

AirFlow Git:

<https://github.com/apache/airflow>

Data Management with AirFlow:

<https://towardsdatascience.com/why-apache-airflow-is-a-great-choice-for-managing-data-pipelines-48effc3e41>

AirFlow and AirBnB:

<https://airbnb.io/projects/airflow/>

AirFlow para Cloud Native:

<https://www.datacouncil.ai/talks/running-airflow-reliably-with-kubernetes>

Desarrollo de DAGs para flujos de trabajo con AirFlow:

<http://michal.karzynski.pl/blog/2017/03/19/developing-workflows-with-apache-airflow/>