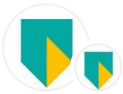
Apache Airflow Tutorial, Part 2: Complete Guide for a Basic Production Installation Using LocalExecutor



ABN AMRO · Follow
Published in ABN AMRO Developer Blog
9 min read · Nov 19, 2020

Listen Share

By Rafael Pierre

In his first *Apache Airflow* tutorial, Rafael Pierre wrote about how to install, setup, and run *Apache Airflow*. In the second part of the tutorial, Rafael gives a complete guide for a Basic Production Installation, using LocalExecutor. To learn more, read the tutorial below.

Source: Unsplash

Recap

In the first post of the series, we learned a bit about *Apache Airflow.* How it not only can help us build Data Engineering & ETL pipelines, but also other types of relevant workflows within advanced analytics, such as MLOps workloads.

We briefly skimmed through some of its building blocks: Sensors, Operators, Hooks, and Executors. These components provide the basic foundation for working with *Apache Airflow*. In the previous tutorial, we worked with the SequentialExecutor. Which is the simplest possible *Airflow* setup. It only has support for running just one task at a time. Therefore, it is used mainly for simple demonstrations. That obviously is not enough for production scenarios, in which we might want to have many tasks and workflows executed in parallel.

As discussed, *Apache Airflow* ships with support for multiple types of Executors. Each of them is more suited to a particular type of scenario.

- **LocalExecutor**
  LocalExecutor unleashes more possibilities by allowing multiple parallel tasks and/or DAGs. This is achieved by leveraging a full-fledged RDBMS for the job metadata database. Which can be built on top of a PostgreSQL or MySQL

database. While you can definitely run some light production workloads using LocalExecutor, for scaling up you would have to resort with vertical scaling by beefing up the hardware resources of your environment.

- **CeleryExecutor**
  CeleryExecutor addresses this limitation. It unlocks the possibility of scaling the setup horizontally, by allowing you to create a processing cluster with multiple machines. Each machine spins up one or more Celery Workers, in order to split *Airflow*'s workload. Communication between workers and the Scheduler is performed through a message broker interface (can be implemented using Redis or RabbitMQ). Having said that, you must define the number of workers beforehand, and set it up in *Airflow*'s configuration. Which can make the setup a bit static.

- **DaskExecutor**
  DaskExecutor works in a similar way as CeleryExecutor. However, instead of Celery, it leverages the Dask framework for achieving distributed computing capabilities.

- **KubernetesExecutor**
  More recently, KubernetesExecutor has become available as an option to scale an *Apache Airflow* setup. It allows you to deploy *Airflow* to a Kubernetes cluster. The capability of spinning Kubernetes Pods up and down is really out of the box. Use such a setup if you would like to add more scale, in a more flexible and dynamic manner. However, that comes at the expense of managing a Kubernetes cluster.

An Apache Airflow MVP

When starting up a data team or capability, evaluating cost versus benefit and complexity versus added value is a critical, time consuming, daunting task. Agile organizations and startups usually work with prototypes to tackle such a scenario — sometimes working products are better at answering questions than people.

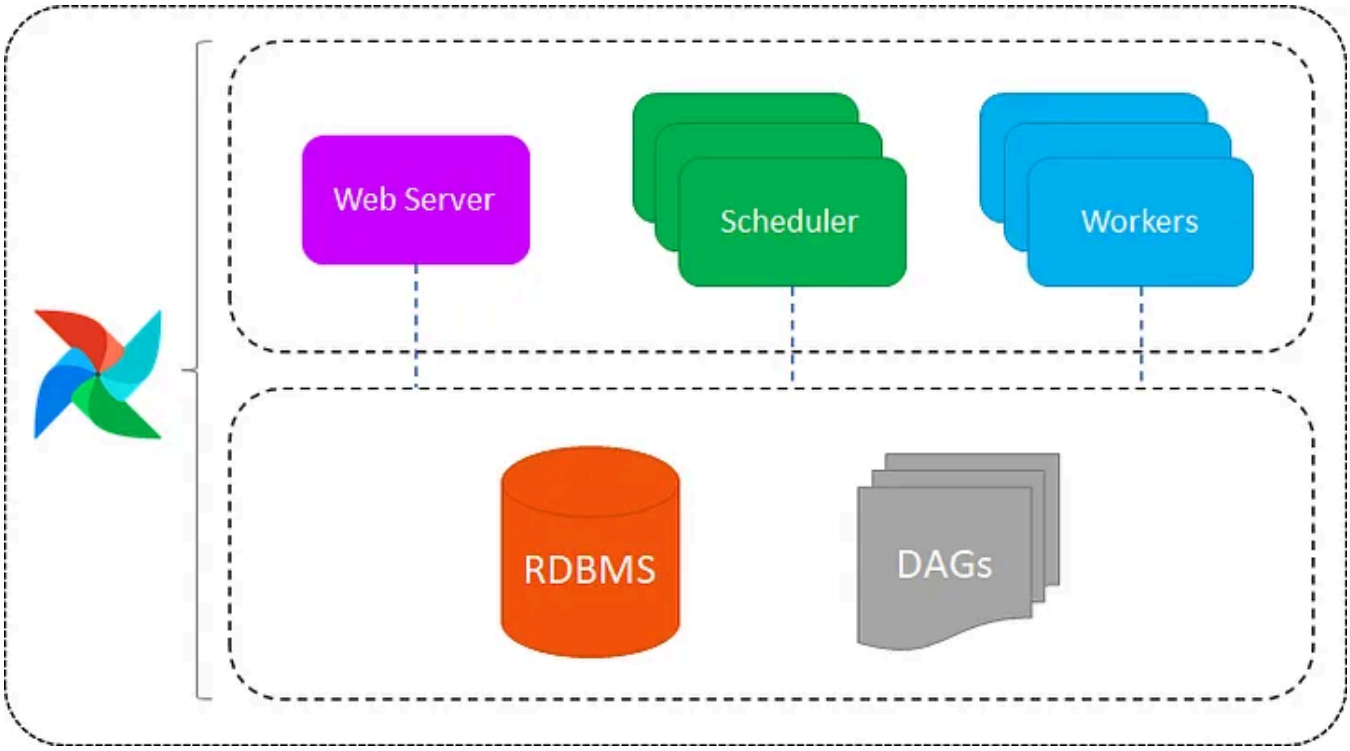
Inspired by this philosophy, we will create a basic hypothetical setup for an *Apache Airflow* production environment. We will have a walkthrough on how to deploy such an environment using the LocalExecutor, one of the possible *Apache Airflow* task mechanisms.

For a production prototype, choosing LocalExecutor is justified by the following reasons:

- Provides parallel processing capabilities
- Only one computing node — less maintenance & operations overhead
- No need for Message Brokers

LocalExecutor

You might be asking — how is that possible? Well, as the name indicates, when we use the LocalExecutor we are basically running all *Airflow* components from the same physical environment. When we look at the *Apache Airflow* architecture, this is what we are talking about:



Main Airflow Components for a LocalExecutor Setup. Source: Author

We have multiple OS processes running the Web Server, Scheduler, and Workers. We can think of LocalExecutor in abstract terms as the layer that makes the interface between the Scheduler and the Workers. Its function is basically spinning up Workers, in order to execute the tasks from Airflow DAGs, while monitoring its status and completion.

Getting The Wheels Turning

Now, we had a conceptual introduction about LocalExecutor. Without further ado, let's set up our environment. Our work will revolve around the following:

1. **Postgresql** Installation and Configuration
2. **Apache Airflow** Installation
3. **Apache Airflow** Configuration
4. **Testing**
5. Setting up *Airflow* to run as a **Service**

These steps were tested with Ubuntu 18.04 LTS, but they should work with any Debian based Linux distro. Here we assume that you already have Python 3.6+ configured. If that's not the case, please refer to this post.

Note: you could also use a managed instance of PostgreSql, such as Azure Database for PostgreSql or Amazon RDS for PostgreSql, for instance. This is in fact recommended for a production setup because that would remove maintenance and backup burden.

1. Postgresql Installation and Configuration

To install PostgreSql we can simply run the following in our prompt:

```
sudo apt-get install postgresql postgresql-contrib
```

In a few seconds, PostgreSql should be installed.

Next, we need to set it up. First step is creating a psql object:

```
sudo -u postgres psql
```

We proceed to set up the required user, database, and permissions:

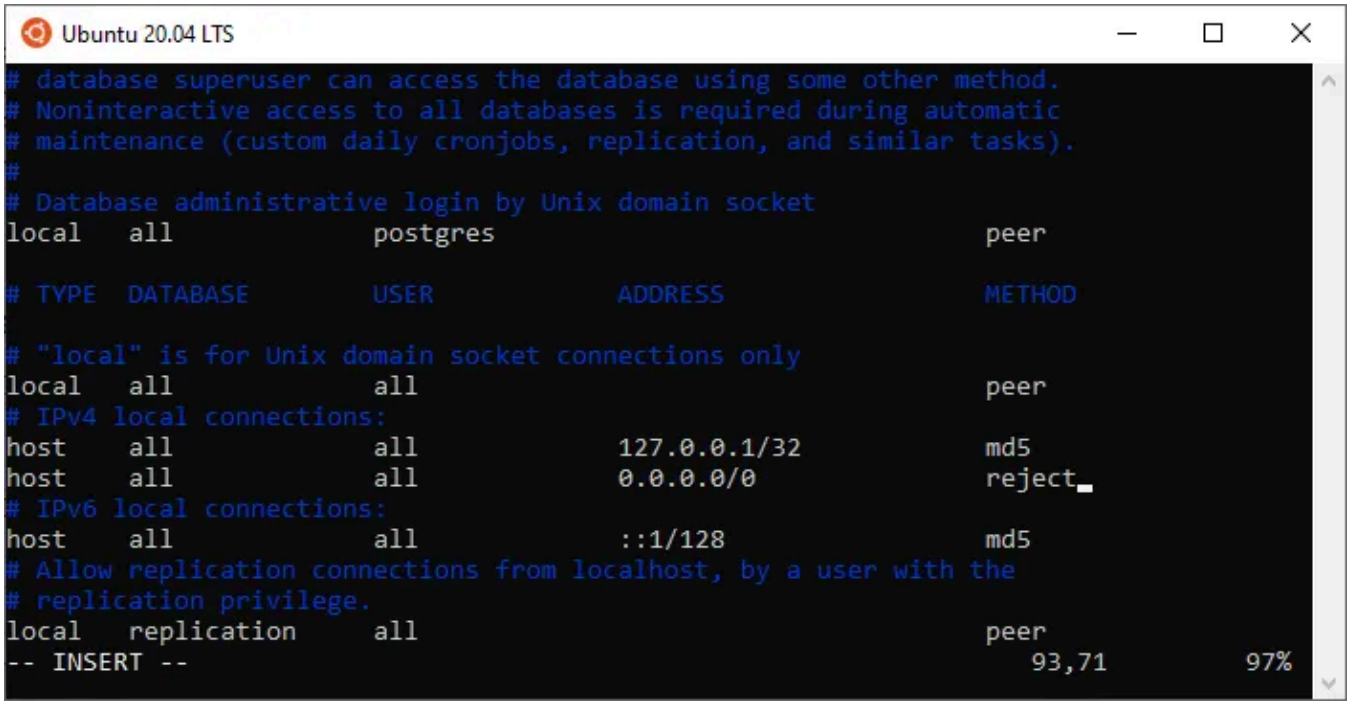
```
postgres=# CREATE USER airflow PASSWORD 'airflow'; #you might wanna
change this
CREATE ROLE
postgres=# CREATE DATABASE airflow;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO
airflow;
GRANT
```

Finally, we need to install libpq-dev for enabling us to implement a PostgreSql client:

```
sudo apt install libpq-dev
```

Optional Step 1: you can make your setup more secure by restricting the connections to your database only to the local machine. To do this, you need to change the IP addresses in the pg_hba.conf file:

```
sudo vim /etc/postgresql/12/main/pg_hba.conf
```



PostgreSql Configurations (pg_hba.conf)

Optional Step 2: you might want to configure PostgreSql to start automatically whenever you boot. To do this:

```
sudo update-rc.d postgresql enable
```

2. Apache Airflow Installation

We will install *Airflow* and its dependencies using pip:

```
pip install apache-airflow['postgresql']
pip install psycopg2
```

By now you should have *Airflow* installed. By default, *Airflow* gets installed to *~/.local/bin*. Remember to run the following command:

```
export PATH=$PATH:/home/your_user/.local/bin/
```

This is required so the system knows where to locate *Airflow*'s binary.

Note: for this example, we are not using virtualenv or Pipenv, but you can feel free to use it. Just make sure that environment dependencies are properly mapped when you setup *Airflow* to run as a service :)

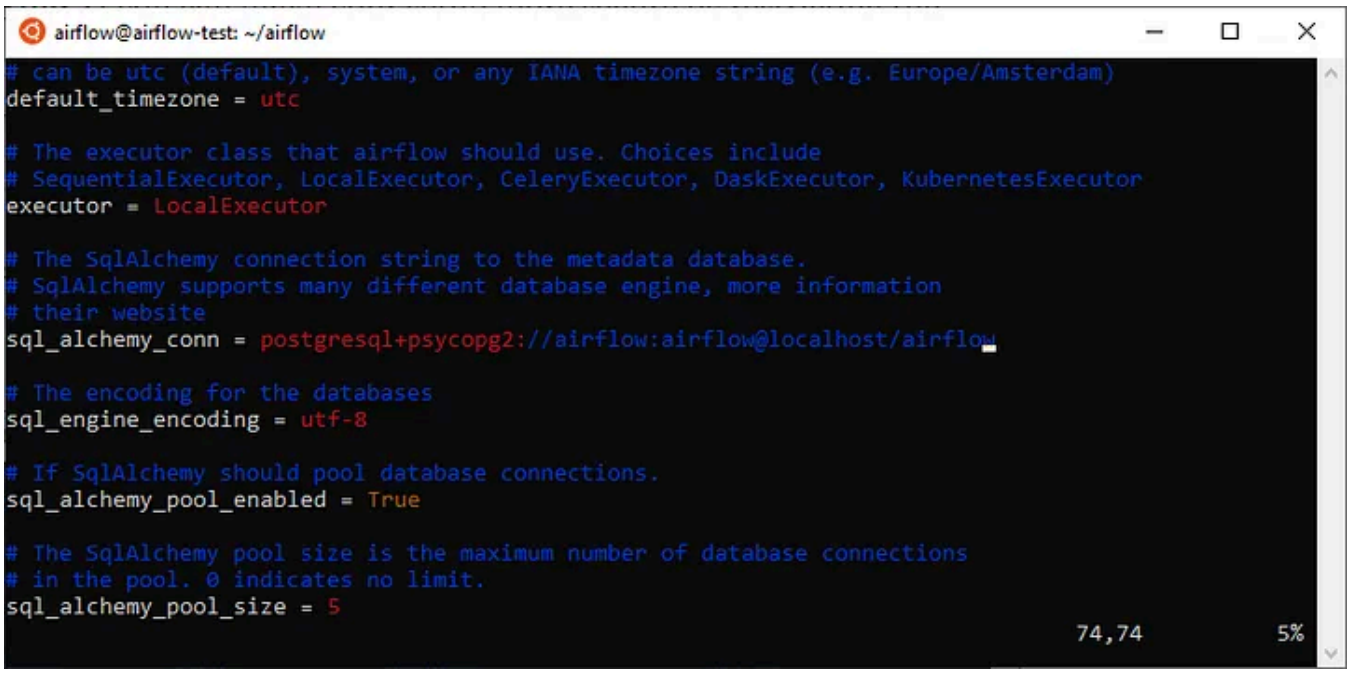
3. Apache Airflow Configuration

Now we need to configure *Airflow* to use LocalExecutor and to use our PostgreSql database.

Go to *Airflow*'s installation directory and edit airflow.cfg.

```
vim airflow.cfg
```

Make sure that the executor parameter is set to LocalExecutor and SqlAlchemy connection string is set accordingly:



Airflow configuration for LocalExecutor

Finally, we need to initialize our database:

```
airflow initdb
```

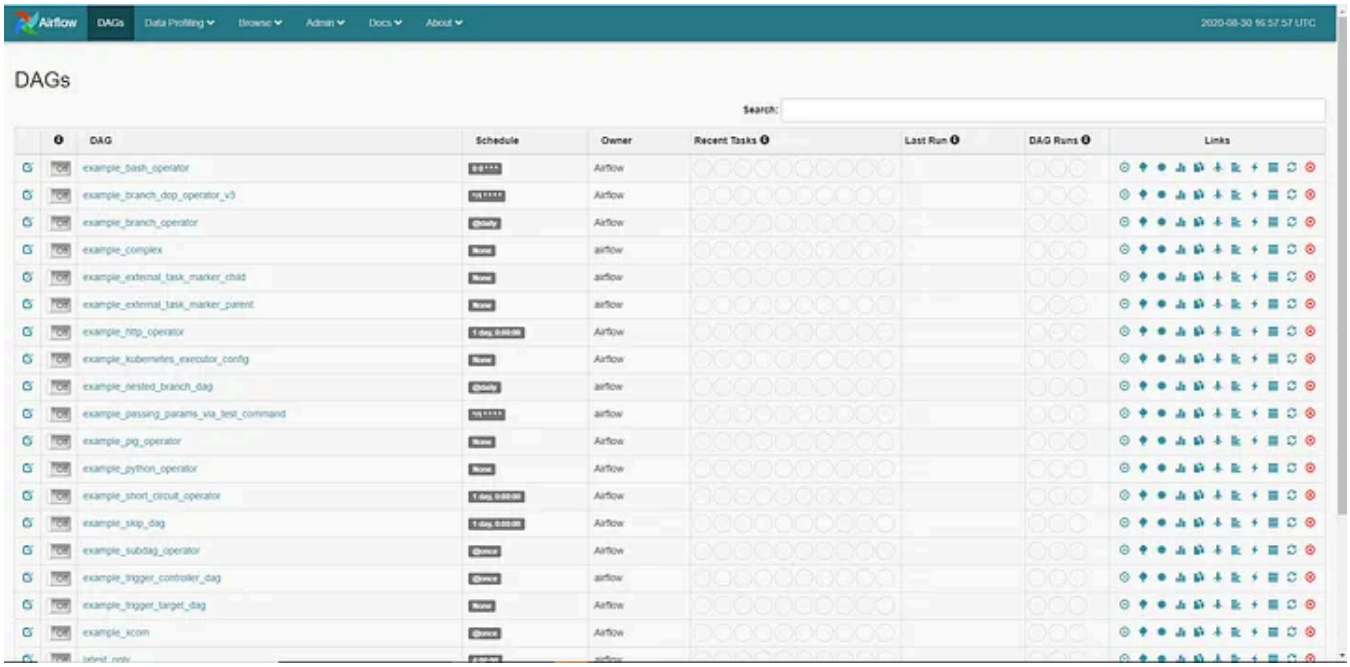
Make sure that no error messages were displayed as part of initdb's output.

4. Testing

It is time to check if *Airflow* is properly working. To do that, we spin up the Scheduler and the Webserver:

```
airflow scheduler
airflow webserver
```

Once you fire up your browser and point to your machine's IP, you should see a fresh *Airflow* installation:



5. Setting up Airflow to Run as a Service

Our last step is to configure the daemon for the Scheduler and the Webserver services. This is required so that we ensure that *Airflow* gets automatically restarted in case there is a failure, or after our machine is rebooted.

As an initial step, we need to configure Gunicorn. Since by default it is not installed globally, we need to create a symbolic link for it.

```
sudo ln -fs $(which gunicorn) /bin/gunicorn
```

Next, we create service files for Webserver and Scheduler:

```
sudo touch /etc/systemd/system/airflow-webserver.service
sudo touch /etc/systemd/system/airflow-scheduler.service
```

Our airflow-webserver.service must look like the following:

```
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# “License”); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.[Unit]
Description=Airflow webserver daemon
After=network.target postgresql.service mysql.service
Wants=postgresql.service mysql.service
[Service]
EnvironmentFile=/etc/environment
User=airflow
Group=airflow
Type=simple
ExecStart= /home/airflow/.local/bin/airflow webserver
Restart=on-failure
RestartSec=5s
PrivateTmp=true
[Install]
```



```
#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# “License”); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
```

```
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.[Unit]
Description=Airflow scheduler daemon
After=network.target postgresql.service mysql.service
Wants=postgresql.service mysql.service
[Service]
EnvironmentFile=/etc/environment
User=airflow
Group=airflow
Type=simple
ExecStart=/home/airflow/.local/bin/airflow scheduler
Restart=always
RestartSec=5s
[Install]
WantedBy=multi-user.target
```

Note: depending on the directory where you installed *Airflow*, your ExecStart variable might need to be changed.

Now we just need to reload our system daemon, enable and start our services:

```
sudo systemctl daemon-reload
sudo systemctl enable airflow-scheduler.service
sudo systemctl start airflow-scheduler.service
sudo systemctl enable airflow-webserver.service
sudo systemctl start airflow-webserver.service
```

Our services should have been started successfully. To confirm that:

```
$ sudo systemctl status airflow-webserver.service
$ sudo systemctl status airflow-scheduler.service
```

You should see some output stating that both services are active and enabled. For example, for the Webserver, you should see something similar to this:

```
airflow@airflow-test: ~
● airflow-webserver.service - Airflow webserver daemon
   Loaded: loaded (/etc/systemd/system/airflow-webserver.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2020-08-30 17:13:38 UTC; 13min ago
 Main PID: 26638 (/usr/bin/python)
    Tasks: 10 (limit: 9479)
   CGroup: /system.slice/airflow-webserver.service
           ├─26638 /usr/bin/python3 /home/airflow/.local/bin/airflow webserver
           ├─26679 gunicorn: master [airflow-webserver]
           ├─30570 [ready] gunicorn: worker [airflow-webserver]
           ├─30944 [ready] gunicorn: worker [airflow-webserver]
           ├─31313 [ready] gunicorn: worker [airflow-webserver]
           └─31699 [ready] gunicorn: worker [airflow-webserver]

Aug 30 17:26:59 airflow-test airflow[26638]: [2020-08-30 17:26:59 +0000] [26679] [INFO] Handling signal: ttou
Aug 30 17:26:59 airflow-test airflow[26638]: [2020-08-30 17:26:59 +0000] [29799] [INFO] Worker exiting (pid: 29799)
Aug 30 17:27:28 airflow-test airflow[26638]: [2020-08-30 17:27:28 +0000] [26679] [INFO] Handling signal: ttin
Aug 30 17:27:28 airflow-test airflow[26638]: [2020-08-30 17:27:28 +0000] [31699] [INFO] Booting worker with pid: 31699
Aug 30 17:27:29 airflow-test airflow[26638]: [2020-08-30 17:27:29,224] {__init__.py:50} INFO - Using executor LocalExecuto
Aug 30 17:27:29 airflow-test airflow[26638]: [2020-08-30 17:27:29,226] {dagbag.py:417} INFO - Filling up the DagBag from /
Aug 30 17:27:29 airflow-test airflow[26638]: /home/airflow/.local/lib/python3.6/site-packages/airflow/models/dag.py:1342:
Aug 30 17:27:29 airflow-test airflow[26638]:   category=PendingDeprecationWarning)
lines 1-21/23 90%
```

That's it.

Now you have a basic Production setup for *Apache Airflow* using the LocalExecutor, which allows you to run DAGs containing parallel tasks and/or run multiple DAGs at the same time. This is definitely a must-have for any kind of serious use case — which I also plan on showcasing on a future post.

Of course, there are many possible improvements here:

- The most obvious one would be to automate these steps by creating a CICD pipeline with an Ansible runbook, for instance
- Using more secure PostgreSql credentials for *Airflow* and storing them in a more secure manner. We could store them as a secret variable within a CICD pipeline and set them up as environment variables, instead of storing in airflow.cfg
- Restricting permissions for *Airflow* users in both OS and PostgreSql

But for now, we will leave these steps for a future article. Click here to read part one of the tutorial.

Rafael Pierre

Tech Apache Airflow Tutorial Developer Abn Amro

Follow

Written by ABN AMRO

790 Followers · Editor for ABN AMRO Developer Blog

Build the future of banking! Use our APIs to automate, innovate, and connect to millions of customers. Go to: https://developer.abnamro.com/

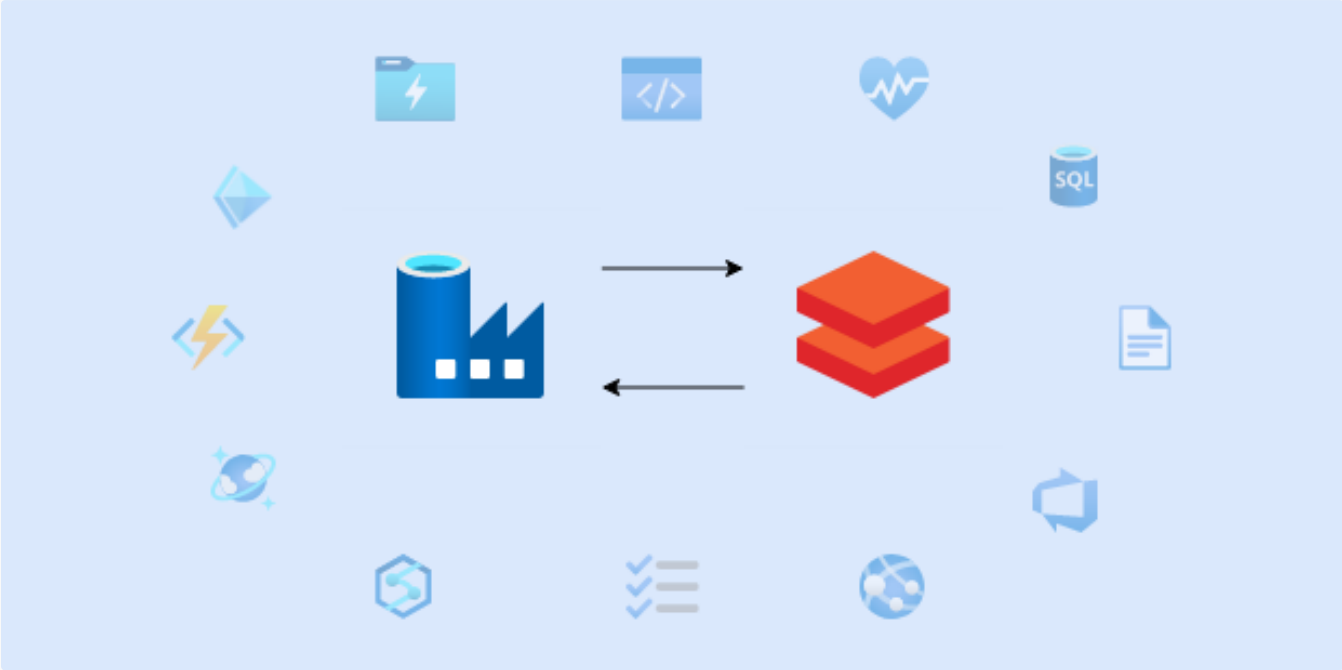
More from ABN AMRO and ABN AMRO Developer Blog

ABN AMRO in ABN AMRO Developer Blog

Azure DevOps Pipeline Tutorial — Part 1: CI pipeline fundamentals

By Penny Xuran Qian

Oct 12, 2021 259 4



ABN AMRO in ABN AMRO Developer Blog

How to pass parameters between Data Factory and Databricks

Tech

Jun 21, 2022 15

ABN AMRO in ABN AMRO Developer Blog

Customer Lifetime Value in Banking

Tech

Sep 12 13

ABN AMRO in ABN AMRO Developer Blog

Building a scalable metadata-driven data ingestion framework

Tech

Oct 25, 2022 76

See all from ABN AMRO

See all from ABN AMRO Developer Blog

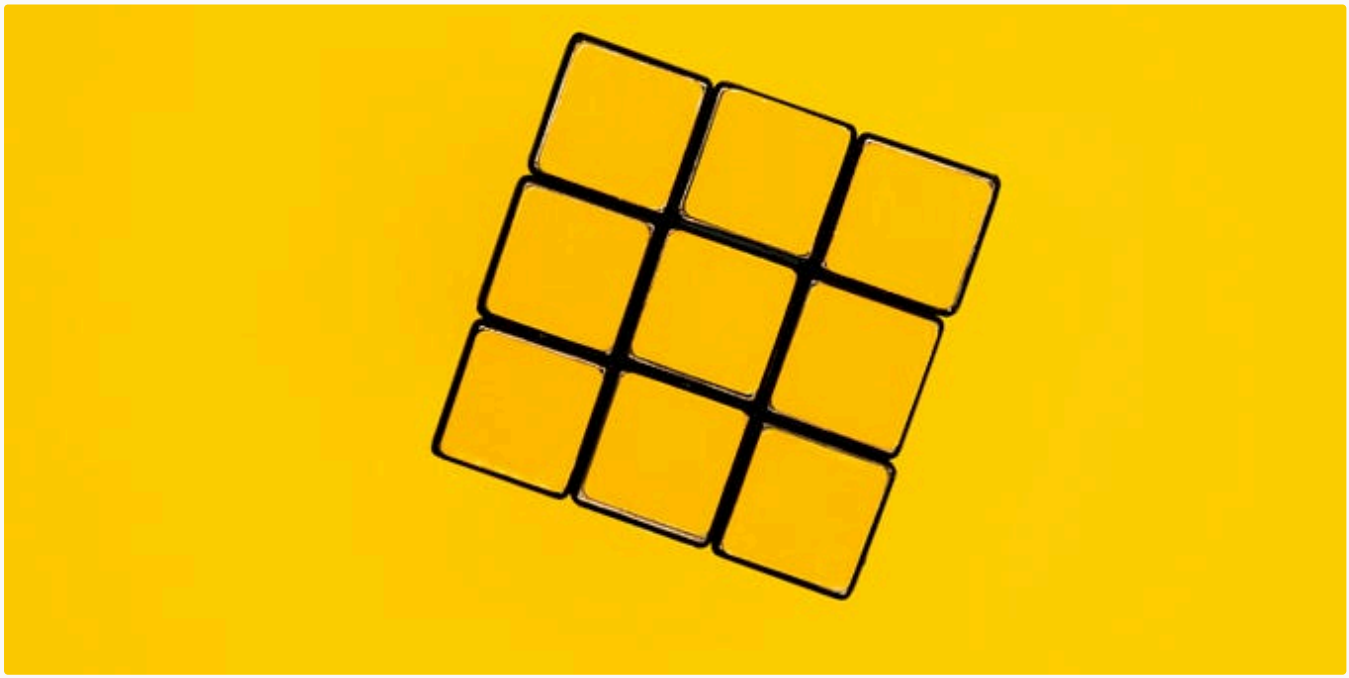
Recommended from Medium

Arpita Mishra

From Basics to Advanced: Navigating Apache Hive for Big Data Professionals

Apache Hive is a data warehousing and SQL-like query language for Hadoop. Developed by Facebook, it is now a part of the Apache Software...

Jun 23 11



Giorgos Myrianthous in Towards Data Science

Mastering Airflow Variables

The way you retrieve variables from Airflow can impact the performance of your DAGs

Jan 27 385 4

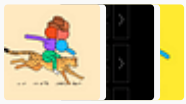
Lists

Apple's Vision Pro

7 stories · 75 saves

Business 101

25 stories · 1194 saves



Figma 101

7 stories · 738 saves

Stories to Help You Grow as a Software Developer

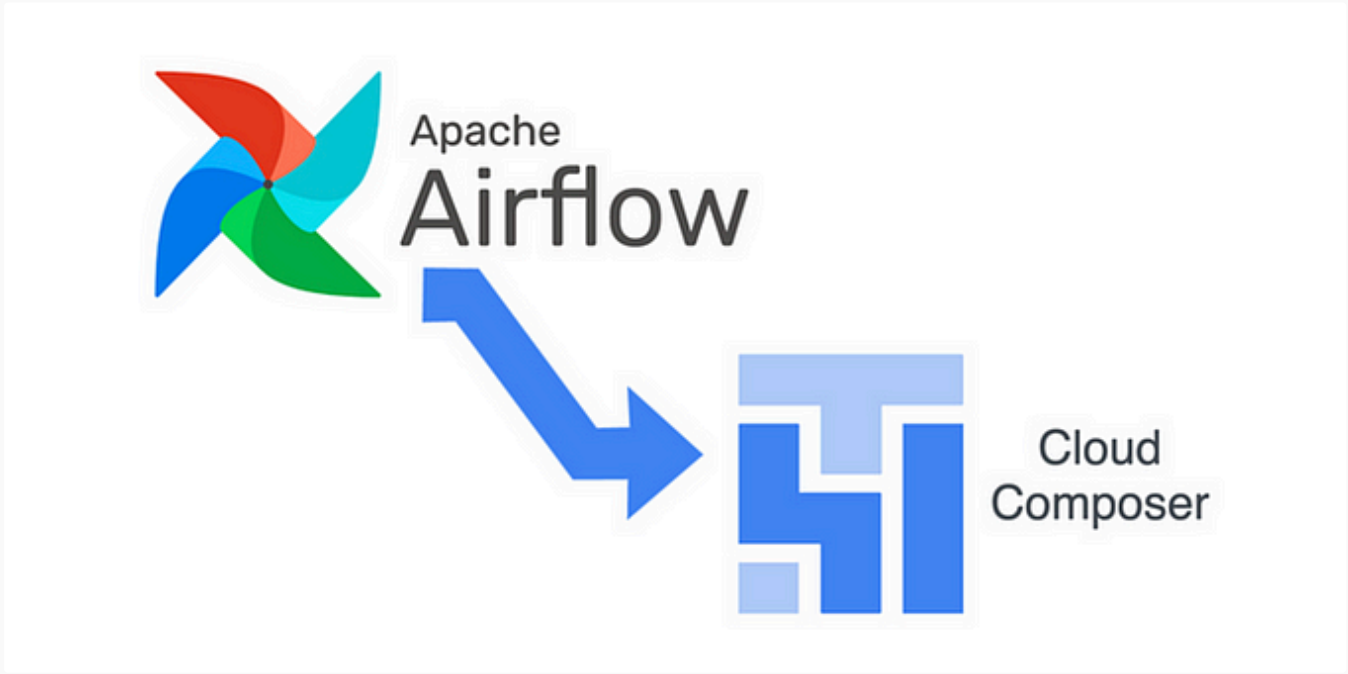
19 stories · 1398 saves

Mayurkumar Surani

Ace Your Data Engineering Interview: 20 Questions and Answers to Land Your Dream Job

So, you're gearing up for a data engineering interview? Congratulations! It's an exciting field with tons of opportunity. But let's be...

Sep 29

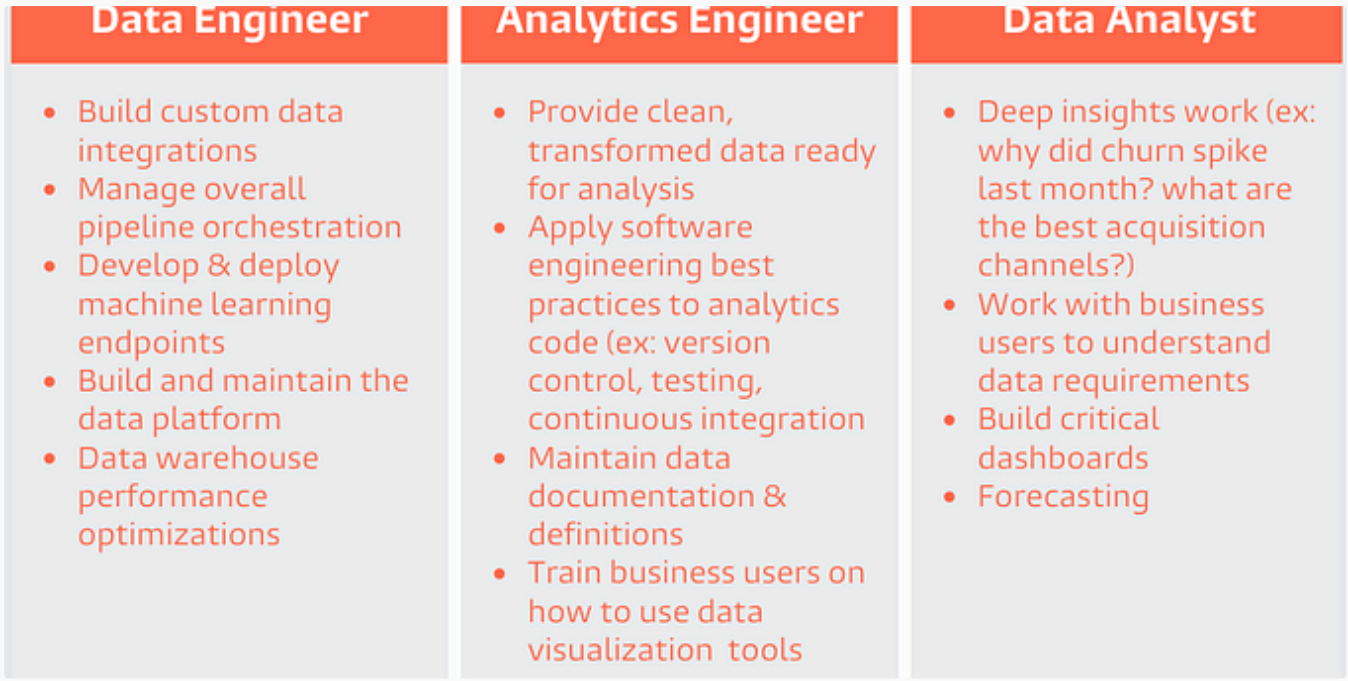


Praveen Bhushan

Optimizing Cloud Composer Dags: Deferrable Operators

Are your Airflow DAGs maxing out your resources? 🤯

Jul 29 55



Srikanta Ghosh

Analytics Engineering Interview Questions- 2024 (Part-1)

Data Modelling and Data Warehousing Questions

Aug 31 26 4

Giorgos Myrianthous in Making Plum

dbt + Airflow = ❤

Building dbt-airflow: A Python package that integrates dbt and Airflow

Apr 16 593 9

See more recommendations