How to Use Airflow without Headaches

Simon Hawe Nov 14, 2019 · 7 min read ★

Photo by Sebastian Herrmann on Unsplash

Data pipelines and/or batch jobs that process and move data on a scheduled basis are well known to all us data folks. The de-facto standard tool to orchestrate all that is Apache Airflow. It is a platform to programmatically author, schedule, and monitor workflows. A workflow is a sequence of tasks represented as a Direct Acyclic Graph (DAG). As an example, think of an extract, transform, load (ETL) job as a

system like Git, which is super handy.

All in all, Airflow is an awesome tool and I love it. But, I initially used it the wrong way, and probably others also do. This misusage leads to headaches, especially when it comes to workflow deployments. Why? In short, we use it for both **orchestrating** workflows *and* **running** tasks on the same Airflow instance. In this article, I gonna tell you why this is an issue. For sure, I will also show you how you can easily fix that. I hope this leads to reducing your Aspirin consumption in the future as it did for me:)

Airflow the Bad Way

I start this article with a short story about myself and Airflow.

When you create a workflow, you need to implement and combine various tasks. In Airflow, you implement a task using Operators. Airflow offers a set of operators out of the box, like a BashOperator and PythonOperator just to mention a few. Obviously, I heavily used the PythonOperator for my tasks as I am a Data Scientist and Python lover. This started very well, but after a while, I thought

> *"Hm how am I gonna deploy my workflow to our production instance? How are my packages and other dependencies installed there?"*

One way is adding a requirements.txt file for each workflow, which gets installed on all Airflow workers on deployment. We tried that but my tasks required a different Pandas version than a colleague's task. Due to this **package dependency issue**, the workflows cannot run on the same Python instance. And not only that, I used the Dataclasses package which requires Python 3.7 but on the production instance, we only had Python 3.6 installed. That's not good.

So I went on and googled a little bit to find another solution suggested by Airflow which is named Packaged-DAGs. This says: "Package all your DAGs and external dependencies into a zip file and deploy that." You can follow this link for more details. To me, this does not sound like a great solution. It also doesn't solve the Python **version issue**.

Another possibility, offered as an Operator, is wrapping your task inside a PythonVirtualEnvOperator. I have to say, I haven't tried that out, as it still doesn't solve the Python version issue. But, I guess it might be rather **slow** if you have to create a

Damm, three tries and still no satisfying solution for deploying tasks written in Python.

Last, I asked myself how can I write a task using other languages than Python? Can that step become **language agnostic**? Can even someone implement a task who **doesn't know the specifics of Airflow**? And how do I deploy and integrate such a task into Airflow? Do I have to compile it and install the result with all dependencies on each worker to finally invoke it through the BashOperator? That sounds like a **painful deployment and development experience**. Furthermore, this might again lead to clashing dependencies. Altogether, this does not sound very satisfying.

> But I want to use Airflow!

So, can we fix it?

Taken from https://knowyourmeme.com/photos/292809-obama-hope-posters

Airflow the Good Way

offers a **DockerOperator** (and also one for Kubernetes) out of the box. This allows us to invoke isolated docker containers as tasks from Airflow.

That was quick :)

But now, a bit more in detail. In the following, I show you how the process from developing Docker-based tasks and DAGs to deploying them looks like end-to-end. For each step, I highlight the respective issues solved.

Task Development and Deployment

1. Develop and test your task in any language and version of the language you want. *This allows you to test and develop tasks in isolation from Airflow specifics and other tasks. It reduces the entry barrier for new developers as they can choose the language they are most comfortable with.*

2. Package the artifacts together with all dependencies into a Docker image. *This solves issues with dependencies and version clashes. It heavily contributes to reducing your headaches.*

3. Expose an Entrypoint from your container to invoke and parameterize a task using the DockerOperator. *This enables you to use your images from Airflow.*

images to the machines executing the tasks. *This facilitates deploying tasks. Again a headache reduction step. Furthermore, it allows you to have multiple versions of the same task. Last, having a central registry enables you to share and reuse tasks across your organization.*

DAG Development and Deployment

1. Build your DAG using the DockerOperator as the only operator. Use it to invoke various tasks available from your Docker registry. *For me, this made my DAG definitions small, clean, and readable. I also did not have to learn any specific Airflow operators other than the DockerOperator.*

2. Put your DAG into a version control system. *With this, deploying your DAG is just a git push and pull away. Again, this should be automated and be part of your CI/CD pipeline.*

I hope that sound reasonable and you get the many advantages it offers. All you have to do is learn Docker if you don't know it yet and you are ready to go.

Now, we know **what** to do. In the next paragraph, let's go through an example to see **how** to do it.

Example

Let's quickly step through an example DAG where I only use the DockerOperator. With this DAG, I simulate an ETL job. Note that the DAG definition file does not reveal that it is just a *simulated* ETL job. It could be a super complex one or just a dummy one. The actual complexity is taken away from the DAG definition and moved to the respective task implementations. The resulting DAG definition file is concise and readable. You can find all the code in my Github repository. You not only find the DAG definition there but also how to build and run a corresponding Airflow instance using Docker. You also find the simulated ETL task implementation together with the Dockerfile. But now, let's get concrete.

The Code

First, let's import the necessary modules and define the default arguments for our DAG.

```
from airflow.operators.docker_operator import DockerOperator
d_args = {
    "start_date": datetime(2019, 11, 14),
    "owner": "shawe",
    "depends_on_past": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}
```

Nothing fancy here, but you see that I have only imported DAG, which is always required, and the DockerOperator. We need nothing else at this stage from Airflow.

Before we build the DAG, some more words about the DockerOperator and what you gonna see. All my tasks use the same Docker image named **etl-dummy** tagged with **latest**. The image offers a CLI named **etl**. This CLI has 3 sub-CLIs namely **extract**, **transform**, and **load**. The sub CLIs have different arguments. To run for example the transform task, you need to call

```
etl --out-dir /some/path transform --upper
```

To invoke a Docker image from the DockerOperator, you just have to specify the image name as name:tag, and the command you want to invoke. Note that my image is stored in my local Docker registry. With this, my etl dag definition looks like

```
si = "@hourly"
with DAG("etl", default_args=d_args, schedule_interval=si) as dag:
    def etl_operator(task_id: str, sub_cli_cmd: str):
        out_dir = "/usr/local/share"
        cmd = f"'etl --out-dir {out_dir} {sub_cli_cmd}'"
        return DockerOperator(
            command=cmd,
            task_id=task_id,
            image=f"etl-dummy:latest",
            volumes=[f"/usr/local/share:{out_dir}"],
        )

    extract = etl_operator("e-step", "extract --url http://etl.de")
    transform = etl_operator("t-step", "transform --lower")
    load = etl_operator("l-step", "load --db 9000://fake-db")
    # Combine the tasks to a DAG
    extract >> transform >> load
```

Final Notes and Tipps

If you host your images in a remotely accessible Docker registry, you have to pass the image name as *registry-name/image-name:image-tag*. Furthermore, you have to provide a *docker_conn_id* to enable Airflow to access the registry. This docker_conn_id references a secret managed by Airflow.

Another feature you can add is storing the image name and tag as variables in Airflow. If you want to update your DAG, all you have to do is push another image to your registry with a new tag, and change the value of the variable in Airflow.

Finally, I want to repeat that you can find all the code including Airflow on Docker and the example Docker image in my **Github repository.**

Wrap Up

I hope this article was useful for you, and if you had headaches in the past, I hope they will go away in the future. Thank you for following this post. As always, feel free to contact me for questions, comments, or suggestions.