Contents

[Intent – 2](#_Toc196140992)

[A Short Tale of a Data Engineer – 2](#_Toc196140993)

[Benefits of Configurations – 2](#_Toc196140994)

[The Framework – 2](#_Toc196140995)

[Components of Framework – 3](#_Toc196140996)

[Appendix – 4](#_Toc196140997)

Create Airflow DAGs Dynamically Using YAML Configurations

# Intent –

The intent of this article help audience understand – how to simplify and scale workflow management by dynamically creating Airflow DAGs using YAML configurations. This approach enhances maintainability, reduces code duplication, and allows non-developers to define workflows easily, enabling teams to manage complex data pipelines with greater flexibility and clarity.

# A Short Tale of a Data Engineer –

Once upon a time, a data engineer started writing ETL DAGs for a never-ending project. ETL was simple, getting data from different sources and applying transformations (maybe reusable maybe not). So, they kept writing multiple DAGs creating replica py files as the sources and transformations changed. Also, while they were using **PythonOperator** (mostly) they decided to shift **KubernetesPodOperator** for some workloads. This means that they should apply the similar code changes to all the replica DAGs. Additionally, after migrating to the Airflow 2.0 **KubernetesPodOperator** gotmajor changes which means applying same change to replica py files.

I don’t want you to go to your job and start generalizing and abstracting things like crazy. Over-engineering and over-generalization are a problem too. You should strive for balance.

# Benefits of Configurations –

1. DAG writing is a repetitive process. With configs one can avoid writing DAGs and avoid duplicating code.
2. A big help for those who don’t have the know-how. When that happens, data engineers become the ones who write DAGs for other people. Which is not necessarily bad, but we can do better.

# The Framework –

Imagine that writing a DAG would be as simple as writing a simple configuration file. The configuration file would do everything. One must specify the things they want to do and **something else** will handle the rest.

That’s what I am trying to achieve when I write this framework. This framework allows for defining data pipelines using a more declarative approach. Rather than the imperative approach enforced by DAG writing. I believe that declarative data pipelines are the future of data pipelines.

To do so, I combined Dynamic DAGs, OOP, and software design patterns. Some of the benefits I obtained by building and using this framework are:

1. It helped in standardizing the way we work with DAGs. So, everyone on the team is on the same page.
2. Almost all DAGs are generated from the same lines of code. So, if we want to introduce a change, we do it in one file and that applies to the rest of the project. We don’t have to do the same change in 10 or 20 files.
3. We used design patterns to obtain a level of code decoupling that makes us feel safe. Introducing new functionalities is easier now. Just as deprecating them.
4. Since writing a DAG is as easy as writing a configuration file, almost anyone can do it. In this way, we empower Data Analysts and Data scientists to write their own DAGs without having to get dirty with the Airflow programming style.

# Components of Framework –

The framework comprises 5 components:

Config files, a DAG factory, a Task creator, Task strategies, and an Operator Factory.

1. Config Files –

Config Files are how DAGs are generated. You basically specify what you want in your DAG and that’s it. Take for instance a simple DAG with 5 tasks. Two of those 2 tasks are EmptyOperators making the DAG look pretty with a start task and an end task. The other 3 tasks are just PythonOperators doing whatever you want — data extraction, data transformation, etc. This is how it looks like on the Airflow UI.



Sample Configuration YAML

A screen shot of a computer program

AI-generated content may be incorrect.

An interface takes the configuration file and passes it to the other components of the framework. Such components take the file a build the DAG as requested by you.

1. DAG Factory –

The DAG Factory is just an implementation of the **Factory Method pattern**. The responsibility of the DAG factory is not only to create DAGs. But, to make sure the tasks of the DAGs are properly created according to the dependencies declared in the configuration file. To achieve that, it uses the Task Creator.

1. Task Creator & Task Strategy –

The Task Creator and the Task Strategies are an implementation of the Strategy Pattern. Basically, you have an interface (strategy) that declares what should be implemented by the child classes. Then, a context class interacts with those child classes through the interface. It is an interesting pattern since you replace inheritance with composition. **Furthermore, it follows the Open-Closed Principle. So, you can add new strategies without having to change the context.**

1. Operator Factory –

Finally, we have the Operator Factory. This is another implementation of the FactoryMethod pattern. I personally love Operator Factory. It helps you make changes once in Factory method which will be applied to operators in case the version upgrade changes any operator definition.

# Appendix –

GitHub - [airflow/dynamic\_dags at main · AbhishekBtra/airflow](https://github.com/AbhishekBtra/airflow/tree/main/dynamic_dags)