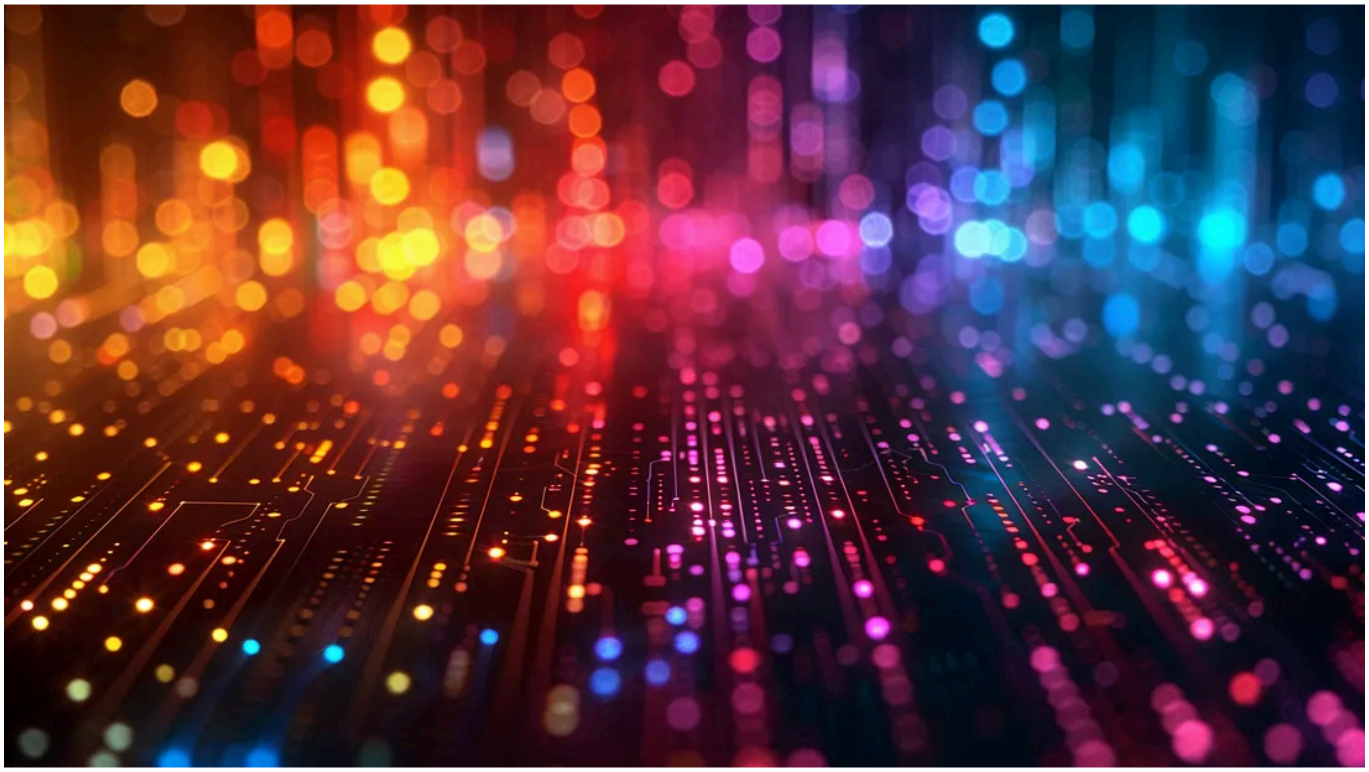
How we orchestrate 2000+ DBT models in Apache Airflow

Alexandre Magno Lima Martins · Follow
13 min read · 1 day ago



In recent years, DBT (Data Build Tool) has established itself as the go-to data transformation workflow, connecting to a variety of processing engines with templating. Along with that, it provides good support for documentation, testing, and community-made packages to extend the native features. It has surely made the **T in ELT** a lot easier and more enjoyable.

Despite taking care of lineage between models, DBT Core does not come with a solution for **where** and **when** it should be executed in a production environment. In other words, it doesn't come with orchestration out of the box.

In this text, you will find how we used Airflow to orchestrate our DBT Core project, creating an **intuitive pipeline that empowered data analysts and even product owners** to create and maintain their own data models. With just SQL and the basics of Git, different people in the business can see their models turned into **Airflow DAGs within a few minutes**, ready to be executed in a distributed and scalable environment, **with alerting, data quality tests, and access control built-in**. And the most important thing: without needing to know what an Airflow DAG is — besides interacting with it in the UI 😄

Let's break it down into key areas:

1. *Mono vs Multi DAG approach*
2. *Project structure and DAGs layout*
3. *The DAG generation pipeline*
4. *How and why we created our DBTOperator*
5. *Conclusion and the road ahead*

Mono vs Multi DAG approach

An intuitive way of approaching this problem is to **model your entire DBT project as "One Big DAG"**. This makes it easier to connect tasks given the DBT lineage and provides a nice lineage view of your entire DBT project in Airflow.

However, the Mono DAG approach comes with some drawbacks that were crucial for us at the time we started this project:

- As the schedule is set at the DAG level, this means that your entire project is going to **run on the same schedule**. This is problematic if you have different SLAs for models across your project.
- This big DAG can be hard to navigate. If you have 2000+ models in your project, finding your way through this gigantic DAG can be challenging, especially for analysts and business people not very used to Airflow.
- It is not efficient for **access control**. As we have different teams owning different parts of the DBT project, we need to leverage this segregation into Airflow as well: only your team should be able to manually trigger your models or decide to perform a full refresh, for example. **Having just one big DAG means having one access control layer for the entire project**.
- It can be hard to split notifications in the case of a model failure. Again, we wanted to notify only the relevant teams in the case of a model failure.

A very important note: we started our project way before DBT released native support for multiple projects. Despite not being fully available in DBT Core, the DBT mesh can be a way to split your project and make the experience of having one DAG per project less of a struggle.

Splitting the DBT project into multiple DAGs

To overcome the mentioned issues, we decided to split the project into different DAGs according to grouping rules that made sense for our organization. By doing this, we can have **different SLAs for different parts of the project**, **access control at the DAG level**, and different targets for **alerting/notification in callback functions**. Also, a team can easily filter just their DAGs and have a better experience browsing through their models in Airflow.

However, natural questions arise, such as: how do we decide which models to group in which DAGs? And how do we connect dependent DAGs?

These important questions guided us in developing the solution we have today. It is important to mention here that being able to see the full DBT lineage in Airflow is not that important for us. We use Datahub for data exploration, which provides an extremely nice lineage view. Therefore, we decided to use Airflow for **managing model execution** in the most efficient way possible, and not as a data discovery tool.

Project Structure and DAGs Layout

When thinking about the questions mentioned earlier, we came up with the concept of **model groups**. A model group is a set of data transformations that are deeply related to each other — tables from the same data mart that need to refresh together and only make sense as a unit, for example. Additionally, they are owned and maintained by a single team. A model group is meant to achieve a goal in the business: perform intermediate transformations and prepare a group of tables, create a data mart, calculate KPIs, etc.

So, we decided to have **one DAG per model group**, as the models are closely related to each other and also need to be scheduled together.

The minimalistic project structure presented below can help in understanding the layout.

Minimalistic DBT project structure.

Let’s break it down:

- **dbt_project.yml:** that’s your regular `dbt_project` file at the root of the project. Nothing special here.
- **deployment.yml:** in this file, you register model groups to be deployed — i.e. for a model group to be converted into a DAG. You also specify the run

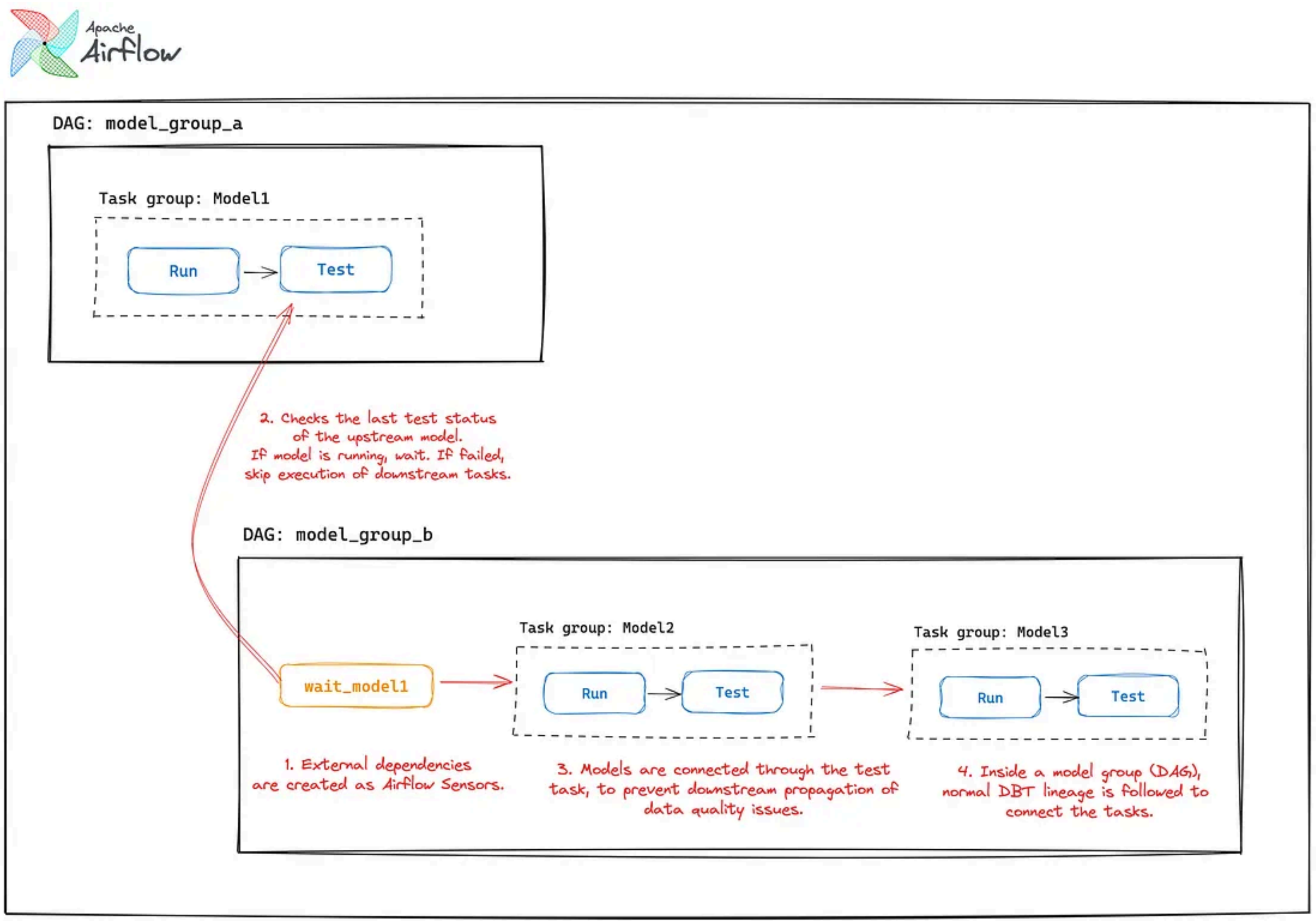
schedule, tags, owners, etc. It would look like this:

```
# deployment.yml
---
model_groups:
  - name: model_group_a # this is the name of the folder.
    schedule: 0 0 * * * # this is the DAG schedule.
    owner: Team_A # This is the owner of the DAG in Airflow (role).
    tags: [tag1, tag2] # Tags for the Airflow DAG
    description: This prepares tables for further transformation. # DAG descript

  - name: model_group_b
    schedule: 0 2 * * *
    owner: Team_A
    tags: [tag1, tag2]
    description: Creates a data mart by joining multiple tables.
```

- **model_group_a** and **model_group_b**: folders that contain the SQL models (DBT Python models also work in the same way). For this example, consider that the `model2.sql` references `model1.sql` in `model_group_a` as a dependency. A `model_group` is nothing but a folder in the DBT project, containing models. You can place as many models as you want inside it, and it also allows for the generation of DAGs for sub-folders.

In Airflow, this structure would look like the following.


Apache Airflow
DAG: model_group_a
Task group: Model1
Run
Test
2. Checks the last test status of the upstream model. If model is running, wait. If failed, skip execution of downstream tasks.
DAG: model_group_b
wait_model1
Task group: Model2
Run
Test
Task group: Model3
Run
Test
1. External dependencies are created as Airflow Sensors.
3. Models are connected through the test task, to prevent downstream propagation of data quality issues.
4. Inside a model group (DAG), normal DBT lineage is followed to connect the tasks.

Schematic Airflow DAGs layout for the DBT project.

By having this structure, we ensure some important points:

- **Dependent DAGs are connected by sensors:** This allows each model group to execute on a different schedule while also preventing failures from propagating downstream. If a sensor check fails, the downstream models are skipped. An important note here is that **we had to fork the native Airflow external task sensor**. This is because we wanted to check the last status of a given upstream model execution, **independent** of its execution date. The native sensor only allows for poking at specific execution dates.

- **Inside the same DAG, the lineage of execution is based on the `dbt-test` task:** This prevents data quality errors from propagating downstream, avoiding a snowball effect where data quality issues get worse down the line.
- **Each DAG (model group) has an owner:** This means that manual actions on that DAG (triggering a full refresh run, clearing tasks, etc.) can only be taken by the appropriate team members.
- **The number of DAGs and their size are flexible, following the DBT project layout:** As all DAGs are generated dynamically based on model groups, they can be as granular or as large as you want them to be. The number of models contained in a model group in the DBT project governs the DAG layout.

The DAG generation pipeline

Now, we will explore what happens from the moment a team member creates a PR in the DBT repository. In a nutshell, the deployment pipeline for the DBT project looks like the following.

Two very important elements were introduced as CI steps to every pull request:

1. **Check of organization governance requirements**: Every model needs to have an owner and appropriate tags, descriptions, etc. This is extremely important because it populates Datahub, making our data catalog rich and meaningful.
2. **Run updated models in a staging environment**: This ensures that the changes being introduced will result in successful runs for both the model being updated and its downstream dependencies. Our staging area for DBT CI run contains representative samples of the production models, minimizing the cost of running CI tests. Properly testing DBT models in CI is a topic of its own and would require a separate post.

After the PR is merged, our DAG generation process kicks in. It works by parsing the DBT `manifest.json` file to get the full graph. Then, according to the rules of model groups defined in `deployment.yaml`, different DAGs are created.

An important concept here is the differentiation between “inner” and “outer” models when parsing the DBT manifest. The inner models are those contained in the relevant model group, whereas outer models are their dependencies outside of the given model group. By using this differentiation, we can allocate appropriate sensors using our `ExternalLatestTaskSensor`, a fork of the native Airflow external task sensor. We modified the metadata database query to fetch the latest state of the upstream task (ordered by execution date), so that the sensor checks the latest `dbt-test` result for the upstream task.

How we determine “inner” and “outer” models when parsing a model group.

Therefore, every model group is also connected by sensors, which allows them to run on individual schedules. Another option considered by us was using the `TriggerDagRunOperator`, but this would only allow us to set the schedule at the upstream-most models.

The actual generation of the DAGs itself is done by templating using Jinja, as we are after all just creating a bunch of Python files 😃. All we need to do is determine which models to include in a particular DAG, their “inner” lineage, and “outer” model dependencies (sensors).

Finally, when the generation is completed, the DAGs and the DBT project artifacts are pushed to Airflow’s artifacts bucket, where another process (running in Airflow) will pick them up. If you want to know how that works on Airflow's side, please refer to my Airflow post.

How and why we created our DBTOperator

If we're running DBT on Airflow, we can either use the `BashOperator` to execute the `dbt` commands, or we can create a `DBTOperator` to handle that. The latter option has numerous advantages over the former, and I'll explain why you should consider creating your own `DBTOperator`.

We began our `DBTOperator` journey using an open-source implementation by the airflow-dbt project. That served us well for a couple of months, but we realized that it would be best if we created our own Operator.

We wanted to use the DBT programmatic invocations instead of subprocess commands, which offer a better way to handle run results and also adhere to best practices. The code became cleaner and more readable after using the Python entry point for the `dbt cli`.

Most importantly, **we wanted to address the evident limitations** in our DBT orchestration solution, especially when handling manual interventions for bug fixes. Some of these limitations are listed below.

Schema changes for incremental models

None of the native DBT `on_schema_change` options solved our problem because in almost all cases, we have to backfill information when a column gets added, for example. So, the only option in the case of a schema change is to trigger a full refresh. We ended up having a considerable number of models failing due to expected source schema changes, and at that time, the only way for us to "trigger" a full refresh was to delete the table on Snowflake 😅.

Of course, this is not ideal. So, one of the first things we implemented in our custom `DBTOperator` was the ability to parse the `dbt-run` execution logs after a failed run. If, by parsing the logs, we detect that the failure was due to a schema change, **we automatically re-trigger that model** passing the `--full-refresh` flag. This simple feature saved us hours in daily maintenance of DBT models.

Initial processing of big models or full refreshes

Sometimes, when doing the initial processing of a very large model or when triggering a manual `full-refresh` for various reasons, we ended up overloading our Snowflake DBT Warehouse. To avoid that, we created a feature in the `DBTOperator` that dynamically changes the warehouse used to execute that model and sets its size (small, medium, large, etc.).

By doing this, we can have all the regular small incremental models running on the same DBT Warehouse, while simultaneously having a huge execution happening on an isolated warehouse with dedicated resources. This prevents the build-up of Snowflake query execution queues.

Additionally, we allow data analysts to trigger full refreshes from the Airflow interface itself, without the need to drop tables. All the `DBTOperator` does is pass the `--full-refresh` flag to the `dbt run` command when appropriate.

Manual trigger of individual models in a model group

Sometimes, our data analysts needed to trigger the run of just one or two models in the model group DAG. Occasionally, these runs had to be full refreshes of the specified models.

To accommodate this, we created an option available as an Airflow DAG parameter, allowing the analyst to trigger only specific models in a DAG. All other models not selected for that specific `DagRun` would be skipped. **This approach prevents wasting resources** by running all models in a DAG when only one or two need to be executed.

While using Airflow's clear task option is also a solution, it falls short in cases where users need to run full refreshes or change the warehouse executing that model. You can't customize runs with parameters by just clearing a task in Airflow. This custom option ensures that analysts can specify their needs more precisely, improving efficiency and flexibility in model execution.

Trigger of downstream dependencies after a model fix

In our Airflow-DBT structure, we have many DAGs according to model groups, and some models have a long dependency chain consisting of 4–5 DAGs. When one model execution (`run` or `test`) in the first DAG of that chain fails, all the downstream models in different DAGs will be skipped to prevent error propagation. How do we ensure that we also re-run all the downstream DAGs after the first model is fixed?

Previously, we used to do this manually 😞. Data analysts had to keep track of downstream DAGs that needed to be re-triggered after a model fix was applied. This process was time-consuming and prone to errors.

To solve this, we created a **Trigger Downstream option** in the Airflow DAG, powered by custom logic in the `DBTOperator` . By setting this value on DAG execution, if all models succeed, the DAG is automatically made aware of all downstream dependencies and triggers the `DagRun` for them. This eliminates the need for a manual process to trigger DAGs after bug fixes.

This process was straightforward to implement using the `dbt ls` command, which lists models in the dependency graph. We then mapped them to the DAGs they are part of and used the Airflow `trigger_dag()` function to automatically trigger downstream executions.

More importantly, this process continues automatically in a “chain reaction”: the triggered DAG receives an argument flag to also trigger its downstream dependencies once finished, continuing the process until the last DAG in the chain is cleared.

DAG Parameters as an interface to DBT executions

After the implementation of the above-mentioned solutions in the `DBTOperator`, we also created DAG parameters to expose some configurations to the users, allowing them to customize manual runs.

DAG Parameters as an interface to DBT

This made a big difference in the day-to-day life of data analysts and analytics engineers who now can fully customize manual runs when needed.

All those parameters are dynamically added to the template when generating DAGs automatically, as explained before. So it uses the available models in that model group to populate the `Models` dropdown, for example.

Also, this "parameter injection" method in the DAG generation pipeline is quite extensible, allowing us to create more parameters in the future when there is a need for it.

Conclusion and the road ahead

I hope this post can bring a different perspective on DBT orchestration in Airflow. Although this implementation continues to serve our needs even after two years, it is far from perfect or ideal and can be improved.

A similar open-source implementation that you can find is the brilliant Astronomer Cosmos project. Interesting features here include the usage of task groups to couple `run` and `test` for every model (as we also did 😍) and the extremely easy and clean way of declaring a `DbtDag` given project configuration.

It is also possible to split your project into more than one DAG, as the constructor can accept `dbt select` arguments. So, you can pass tags and split your project according to different tags, for example. However, it is not clear to me how or if they deal with possible interactions (model references) across DAGs. If you are starting your DBT orchestration journey, you definitely should check it out as it provides a very clean abstraction.

As of now, what lies ahead of us and is heavily impacting how we interact with DBT is the implementation of data contracts. By using a contract as the bridge between a source system and the tables in the data lake, we can automate the provisioning of connectors (data extractors) and also **DBT models to perform initial (basic) transformations that don't require business knowledge,** like casting types, standardizing column names and unnesting of complex fields. Consequently, some of the model groups described earlier are being created in an entirely automated fashion based on data contracts.

I'm fully open to discussing the DBT-Airflow implementation further and very keen to hear how the community is solving this problem. So, if you have a similar implementation, please let me know how you are doing it 😆. It's only through a connected community that we build truly amazing solutions that deliver real value.

Data Engineering Dbt Airflow Orchestration

Written by Alexandre Magno Lima Martins

Follow

151 Followers

https://www.linkedin.com/in/alex-magno/

More from Alexandre Magno Lima Martins

Alexandre Magno Lima Martins in Apache Airflow

What we learned after running Airflow on Kubernetes for 2 years

Apache Airflow is one of the most important components in our Data Platform, used by different teams inside the business. It...

13 min read · Feb 7, 2024

756 20

See all from Alexandre Magno Lima Martins

Recommended from Medium

Blosher Brar

How dbt fits into ETL /ELT

Karl Sorensen in Digitalis.io Blog

In 2006 mathematician Clive Humby is coined the phrase “data is the new oil.” Indeed, entir...

6 min read · May 18, 2024

9

Apache Airflow Branching (and gotcha!)

Apache Airflow is a powerful, open-source tool that allows you to define workflows for...

5 min read · Feb 21, 2024

1

Lists

Natural Language Processing

1472 stories · 987 saves

Staff Picks

649 stories · 996 saves

Abel Bekele

Data warehouse tech stack with PostgreSQL, DBT, Airflow and...

The objective of this initiative is to support the city traffic department by employing swarm...

7 min read · Dec 24, 2023

59

Suneet Sahadevan

Read CSV or Excel files into Snowflake using Python DBT...

I searched a lot of forums for a python model in DBT which can read csv or excel files into ...

1 min read · Apr 19, 2024

2

Leo Godin in Data Engineer Things

Automate Dbt Date Logic with Python — Part 2

Simplifying Our Models and Tests From Part 1 Using Meta Config

12 min read · May 14, 2024

19

Louis ADAM

Anatomy of a Data Platform — How to choose your data architecture

I've seen many data architectures during my assignments in different companies....

8 min read · Apr 4, 2024

131 1

See more recommendations