gusty

Chris Cardillo

4/17/23

Table of contents

Preface

Orchestration, or the routine scheduling and exection of dependent tasks, is a core component of modern data work. Orchestration continues to reach more and more data workers - it was originally a focus for data engineers, but it now permeates the work of data analysts, analytics engineers, data scientists, and machine learning engineers. The easier it is for any class of data worker to orchestrate their code, the easier it is for any member of an organization to derive value from the outputs of that code.

Flavors of Orchestration Code

Orchestration with Python is a vast and opinionated landscape, but there are three clear flavors of orchestration to have emerged over time:

1. **Object-oriented** orchestration, where tasks are objects and dependencies between tasks are handled with methods. Airflow's classic style is a good example of object-oriented orchestration.

2. **Decorative** orchestration, where tasks are functions and decorators are used to configure the tasks. Airflow's taskflow API and Dagster's entire API are good examples of decorative orchestation.

3. **"File as a Task" (FaaT)** orchestration, where tasks are files. Tools like Mage, dbt, and Orchest exemplify FaaT orchestration.

What is gusty?

gusty is a FaaT framework for Airflow, the absolute standard for orchestrators today. While other orchestrators natively support FaaT, Airflow is a Top-Level Apache Project with sustained development, a gigantic ecosystem of provider packages, and is offered as a hosted service by major public clouds and other Airflow-focused companies. If you are reading this, you're probably already familiar with - or using - Airflow.

gusty exists to make FaaT orchestration fun and easy using Airflow, allowing for FaaT DAGs to be incorporated in existing Airflow projects without any need to change existing work or Airflow code.

1 gusty Basics

To familiarize ourselved with gusty, we'll start by making a simple DAG, called `hello_dag`.

1.1 gusty DAG Structure

A gusty DAG lives inside of your Airflow DAGs folder (by default `$AIRFLOW_HOME/dags`), and is comprised of a few core elements:

1. **Task Definition Files** - Each file hold specifications for a given task. In the example below, `hi.py`, `hey.sql`, and `hello.yml` are our Task Definition Files. These Task Definition Files are all stored inside our `hello_dag` folder.

2. **METADATA.yml** - This optional file contains any argument that could be passed to [Airflow's DAG object](), as well as some optional gusty-specifc argument. In the example below, `METADATA.yml` is stored inside of our `hello_dag` folder, alongside the Task Definition Files.

3. **DAG File** - The file that turns a gusty DAG folder into an Airflow DAG. It's more or less like any other Airflow DAG file, and it will contain gusty's `create_dag` function. In the example below, `hello_dag.py` is our DAG Generation File. The DAG Generation File does *not* need to be named identically to the DAG folder.

```
$AIRFLOW_HOME/dags/

    hello_dag/
        METADATA.yml
        hi.py
        hey.sql
        hello.yml

    hello_dag.py
```

In the event you wanted to create a second gusty DAG, you can just repeat this pattern. For example, if we wanted to add `goodbye_dag`:

```
$AIRFLOW_HOME/dags/

    goodbye_dag/
        METADATA.yml
        bye.py
        later.sql
        goodbye.yml
|
    hello_dag/
        METADATA.yml
        hi.py
        hey.sql
        hello.yml

    goodbye_dag.py
    hello_dag.py
```

1.2 Task Definition Files

The three primary file types used for Task Definition Files are Python, SQL, and YAML. gusty supports other file types, but these three are the most commonly used. The general pattern for Task Definition files is that they contain:

- **Frontmatter** - YAML which carries the specification and parameterization for the task. This can include which Airflow (or custom) operator to use, any keyword arguments to be passed to that operator, and any task dependencies the given task may have.

- **Body** - The primary contents of the task. For example, the Body of a SQL file is the SQL statement which will be executed; the body of a Python file can be the `python_callable` that will be ran by the operator. For YAML files, there is no Body because the whole Task Definition File is YAML.

gusty will pass any argument that can be passed to the `operator` specified (as well as any BaseOperator arguments) to the operator. The specified `operator` should be a full path to that operator.

The file name of each Task Definition File will become the name of the Airflow task.

Let's explore these different file types by looking at the contents of these Task Definition Files in `hello_dag`.

YAML Files with `hello.yml`

Here are the contents of our `hello.yml` file:

```
operator: airflow.operators.bash.BashOperator
bash_command: echo hello
```

Because the entire file is YAML, there is no separation of Frontmatter and Body.

The resulting task would contain a `BashOperator` with the task id `hello`.

SQL Files with `hey.sql`

Here are the contents of our `hey.sql` file:

```
---
operator: airflow.providers.sqlite.operators.sqlite.SqliteOperator
---

SELECT 'hey'
```

The resulting task would contain a `SqliteOperator` with the task id `hey`.

The Frontmatter of our SQL file is encased in a set of triple dashes (`---`). The Body of the file is everything below the second set of triple dashes. For SQL files, the Body of the file is passed to the `sql` argument of the underlying operator. In this case, `SELECT 'hey'` would be passed to the `sql` argument.

Python Files with `hi.py`

Here are the contents of our `hi.py` file:

```
# ---
# python_callable: say_hi
# ---

def say_hi():
  phrase = "hi"
  print(phrase)
  return phrase
```

The resulting task would contain a `PythonOperator` with the task id `hi`.

The Frontmatter of our Python file is also encased in a set of triple dashes (`---`), but you will also note that the entirety of the Frontmatter, including the triple dashes, are prefixed by comment hashes (`#`).

By default, gusty will specify specify Airflow's PythonOperator as the `operator`, if no `operator` argument is provided. As with any Task Definition File, you can specify whatever `operator` is available to you in your Airflow environment, so you could just as easily add `operator: airflow.operators.python.PythonVirtualenvOperator` to this Frontmatter to use the `PythonVirtualenvOperator` instead of the `PythonOperator`.

When a `python_callable` is specified in the Frontmatter of a Python file, gusty will search the Body of the Python file for a function with the name specified in the Frontmatter's `python_callable` argument. For the best results with Python files, it's recommended that you put all of the Body contents in a named function, as illustrated above.