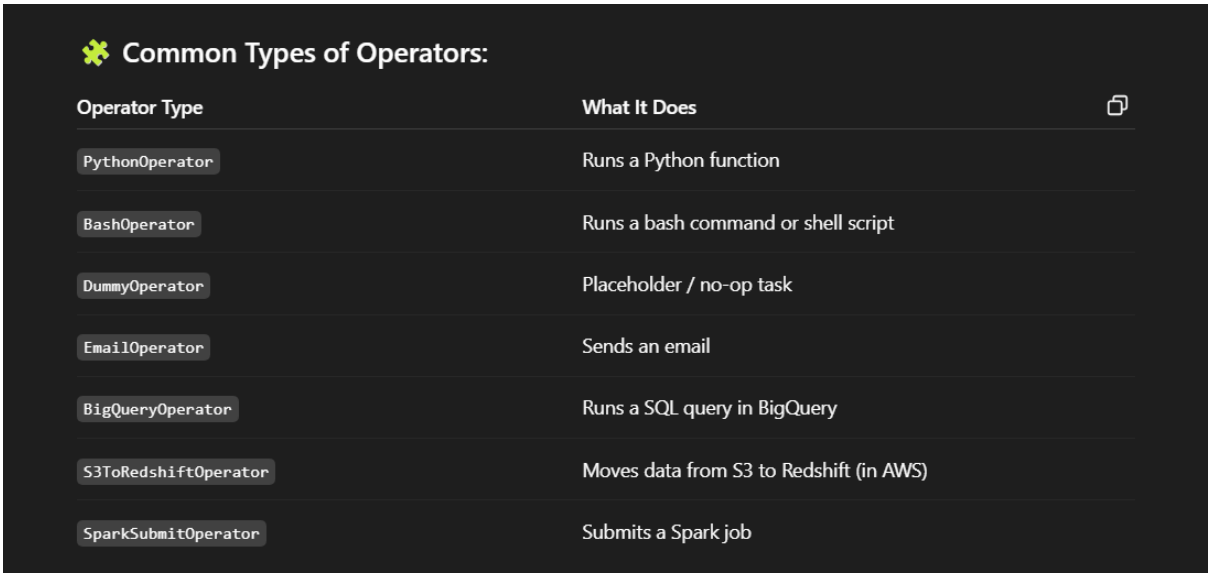
Airflow- Orchestration tool : [Github Notes]

Airflow is an orchestration tool, not a processing engine.
It defines *when*, *how*, and *in what order* tasks should run — using **operators** to trigger the actual work — while handling retries, dependencies, logging, and monitoring.

Actual work is done by Operators which are defined in the dag block.



🧩 Common Types of Operators:

Operator Type	What It Does
`PythonOperator`	Runs a Python function
`BashOperator`	Runs a bash command or shell script
`DummyOperator`	Placeholder / no-op task
`EmailOperator`	Sends an email
`BigQueryOperator`	Runs a SQL query in BigQuery
`S3ToRedshiftOperator`	Moves data from S3 to Redshift (in AWS)
`SparkSubmitOperator`	Submits a Spark job

Airflow handles:

- **Scheduling** (when to run tasks),
- **Dependencies** (order of tasks),
- **Retries**, **Logging**, and **Monitoring**.

It acts like a **conductor**, ensuring all tasks run at the right time, in the right order, and under the right conditions.

The definition of orchestration is a set of configurations to automate tasks, jobs, and their dependencies. If we are talking about database orchestration, we talk about how to automate the table creation process.

Now, what is Airflow? Airflow is an open source workflow management tool. What is unique in Airflow is that we use a Python script to manage our workflows.

When talking about workflow management tools, there are three main components here, as follows:

• Handling task dependencies

• Scheduler

• System integration

If you have ever heard of or used tools such as Control-M, Informatica, Talend, or many other ETL tools, Airflow has the same positioning as these tools. The difference is Airflow is not a user interface (UI)-based drag and drop tool. Airflow is designed for you to write the workflow using code

Every single workflow in Airflow is called a directed acyclic graph (DAG). A DAG is a collection of tasks that are chained together with their dependencies.

The DAG code looks like this in Python:

```
args = {
    'owner': ' packt-developer  ,
}
with DAG(
    dag_id='hello_world_airflow',
    default_args=args,
    schedule_interval='0 5 * * *',
    start_date=days_ago(1),
) as dag:
```

This is a very common DAG declaration in Airflow. Do remember that a DAG has three important pieces of information: the DAG ID, the time you want to start the DAG, and the interval—in other words, how you want to schedule the DAG. As a note, the DAG ID needs to be unique for the entire Airflow environment.

For `schedule_interval`, it follows the `CronJob` scheduling format. If you are not familiar with the `CronJob` format, it uses five numerical values that represent the following: minute, hour, day(month), month, and day(week).

An asterisk (`*`) means *every*. For example, `* * * * *` means the DAG will run every minute, while `0 1 * * *` means the DAG will run every day at 1:00 A.M.

After learning about DAG, we will learn about **tasks** and **operators**. A DAG consists of one or many tasks. A task is declared using operators. As in our example code, we will use two `BashOperator` instances, and both operators will print words.

The first task will print `Hello` and the second task will print `World`, like this:

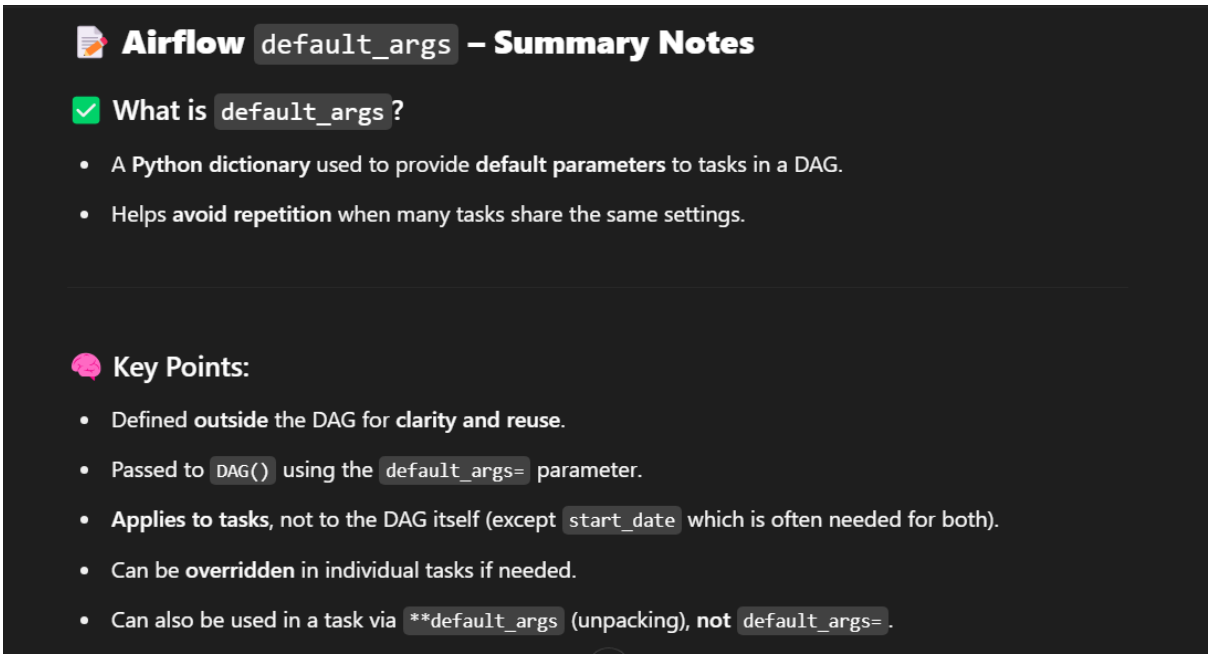
```
    print_hello = BashOperator(
        task_id='print_hello',
        bash_command='echo Hello',
    )

    print_world= BashOperator(
        task_id='print_world',
        bash_command='echo World',
    )
```

`BashOperator` is one of many operators that are available for Airflow. `BashOperator` is a simple operator for you to run Linux commands.

Now it's time to Write some Code:

Question 1 : What is default_arg in a Dag Code?



❌ Misuse to Avoid:

- Don't use `default_args=...` inside `PythonOperator` or other tasks — use `**default_args` instead.

✅ Example:

python Copy Edit

```python
default_args = {
    'owner': 'prashant',
    'start_date': datetime(2023, 1, 1),
    'retries': 2,
    'retry_delay': timedelta(minutes=5)
}

with DAG('my_dag', default_args=default_args, schedule_interval='@daily') as dag:
    task = PythonOperator(task_id='hello', python_callable=print_hello)
```

❗ Important Note:

You **cannot** do:

python Copy Edit

```python
PythonOperator(task_id='t1', python_callable=..., default_args=default_args)
```

❌

That will raise an error — `default_args` is not a valid parameter of `PythonOperator`.

If you pass `default_args` to the **DAG**, tasks automatically **inherit** them:

python Copy Edit

```python
with DAG('example_dag', default_args=default_args) as dag:
    PythonOperator(task_id='task1', python_callable=do_something)
```

But you can still **override or use different args** per task by passing them individually.

Start Date and Schedule interval :

1. What does the schedule look like?

- schedule_interval = '0 5 * * *': This means the DAG will run every day at 5:00 AM.
- start_date = days_ago(1): If today is May 5, days_ago(1) means the start date is May 4, 2024.

---

2. When is the first execution_date?

- First run: The first run of the DAG will be for the interval of May 4, from 5:00 AM May 4 to 5:00 AM May 5.
- But, it will run after the interval ends, which means the DAG will actually run at 5:00 AM on May 5.

---

3. Summary of Timing

- Start Date: May 4 (since days_ago(1) makes it May 4)
- First execution_date: May 4 (this is the interval Airflow will process)
- Actual first run time: May 5 at 5:00 AM (Airflow will run it after the interval ends)

Important note:

The DAG won't run exactly at the start date but after the scheduled interval ends.

We have three options for declaring variables, as follows:

1. Environment variables
2. Airflow variables
3. DAG variables

DAG variables are variables that we already used, the variables that live in the DAG script, applicable only to the DAG.

The higher-level variables are Airflow variables. You can call Airflow variables from all of your DAG.

Understanding Airflow backfilling, rerun, and catchup

In a data pipeline, we often need to handle data from the past. This is a very common scenario in data engineering. Most of the time, applications as the data sources are created before a data lake or data warehouse, so we need to load data from the past. There are three main terms related to this.

This is a very Important topic which we need to understand.

The first one is **backfilling**. Backfilling happens when you need to load data from the past.

For illustration, imagine in a real-life scenario you already run a data pipeline for *7 days* starting from `2021-01-01`, without any issue. For some reason, your end user asked you to also load data from `2020-12-01`. This is a backfilling scenario. In this case, you need to load the historical data without changing or disturbing your running data pipeline, as illustrated in the following diagram:

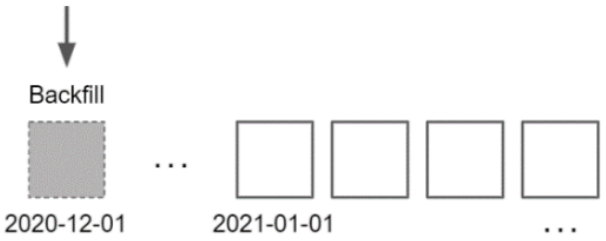


Figure 4.16 – Backfill illustration

In Cloud Composer, you can run backfilling using the `gcloud` command, like this:

```
gcloud composer environments run \ ${your_composer_environment_
name} \
--location [your composer environment region] \
backfill -- -s [your backfill start date] \
-e [your backfill end date] [your dag id]
```

No — in **backfilling**, you can **choose to run for just one specific date**. It does **not** have to run for all previous dates unless you tell it to.

If you run the preceding command, Airflow will trigger DAG Runs for the given date. Remember our section about using Airflow macros for getting the `execution_date` variable? The `execution_date` variable will return the backfill date.

The second one is a **rerun**. A rerun happens when you need to reload data from the past. The difference between a rerun and a backfill is that a rerun works for a DAG or tasks that have run before. The scenario of using a rerun is when a DAG or tasks have failed, so you need to rerun them, as illustrated in the following diagram:

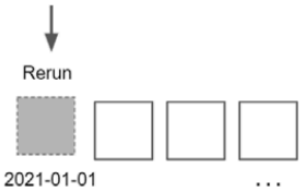


Figure 4.17 – Rerun illustration

You can trigger a rerun from the Airflow web UI. In the web UI, if you click **Clear** in either the DAG or task indicator (in the DAG **Tree View**), the DAG or task will retry, and that's what we call a rerun. A second option is using the `gcloud` command, like this:

```
gcloud composer environments run \
[your composer environment name] \
  --location [your composer environment region] \
clear -- [your dag id] -t [your tasks id or regex] -s \
[your start date] -d [your end date]
```

The command will trigger a rerun of the specific DAG and tasks.

The third one is a **catchup**. A catchup happens when you deploy a DAG for the first time. A catchup is a process when Airflow automatically triggers multiple DAG Runs to load all expected date data. It's similar to a backfill, but the trigger happens automatically as intended, as illustrated in the following diagram:

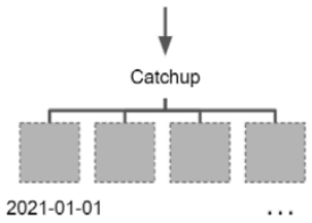


Figure 4.18 – Catchup illustration

For example, today is `2021-01-07`.

You declare the `start_date` value of a DAG is `2021-01-01`.

When you deploy the first time, there will be seven DAG Runs automatically running: `2021-01-01` to `2021-01-07`.

The preceding scenario is by default how Airflow works for your DAG, but you can disable that behavior. You can set the catchup to `False` in your DAG declaration—for example, like this:

```
with DAG(
    dag_id='example_dag',
    start_date=datetime(2021, 1, 1),
    catchup=False
) as dag:
```

If you set the `catchup` parameter to `False` in your DAG, the catchup won't run.

These three Airflow features are very important and highly relevant to real-world scenarios. Whether you are in the development phase or in production mode, you will use these features.

task idempotency

A good task in a DAG is a task that can run and produce the same result every time it runs, and there is a term for this: task idempotency.

If you implement the WRITE TRUNCATE method, your task will be idempotent, which means that every time there is a rerun for any date, the data in BigQuery will always be correct without duplication. The BigQuery table won't have repeated data since the records will be reloaded on each DAG Run. But is this a best practice? The answer is: not yet. This is not yet the best practice. Using this approach, you will notice that the BigQuery table will be rewritten over and over again for the whole table. If you have a table of 1 terabyte (TB) in size and the DAG already runs for 1 year, Airflow will rewrite 1 TB of data 365 times. And we don't want that—we need the BigQuery partition tables to improve this approach.

Introducing BigQuery partitioning

In this section, before continuing our Cloud Composer exercise, let's take a little step back to BigQuery. There is one essential feature in BigQuery called a BigQuery partitioned table. A BigQuery partitioned table will logically divide the data in the BigQuery table by partitioning it into segments using a key.

There are three partition key options, outlined as follows:

- **Time-unit columns**: Based on a column containing `TIMESTAMP`, `DATE`, or `DATETIME` value
- **Ingestion time**: Based on the timestamp when BigQuery ingests data to the table
- **Integer range**: Based on a column containing the integer value

The most common scenario is using either a time-unit column or ingestion time, and even though you can partition up to an hourly granular level, the most common scenario is still partitioning at a daily level. This feature will benefit mainly cost and performance optimization, but other than those two factors, using BigQuery partitioned tables can help our load jobs.

Take a look at the next example.

With this, every time there is a rerun on a particular date, the table partition will be rewritten, but again, only on a specific partition date, as illustrated in the following diagram:

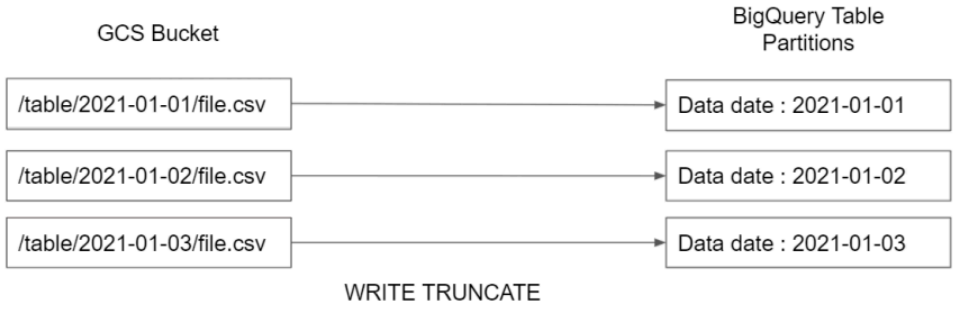


Figure 4.23 – WRITE_TRUNCATE method with BigQuery partitioned table illustration

With this, your DAG and all tasks are idempotent. Imagine that you want to retry any tasks on any expected day—in that case, you do not need to worry. The tasks are safe to rerun and there will be no data duplication, thanks to the `WRITE_TRUNCATE` method. And also, for the `partitioned_by` date, the table will only rewrite the records on the rerun execution day.