# Airflow

Sunday, November 19, 2023 7:12 PM

## # Chapter 1:

→ Airflow is not a data processing tool in itself but orchestrates the different components responsible for processing your data in pipelines.

→ In Airflow's DAG, tasks (ie. data processing) is represented as nodes and dependencies betw tasks are edges. Lack of edges between nodes means the tasks are independent.

→ The idea to have DAG and not one monolithic script is because if one step fails, will need to run the entire script again.

- *The Airflow scheduler*—Parses DAGs, checks their schedule interval, and (if the DAGs' schedule has passed) starts scheduling the DAGs' tasks for execution by passing them to the Airflow workers.
- *The Airflow workers*—Pick up tasks that are scheduled for execution and execute them. As such, the workers are responsible for actually "doing the work."
- *The Airflow webserver*—Visualizes the DAGs parsed by the scheduler and provides

the main interface for users to monitor DAG runs and their results.



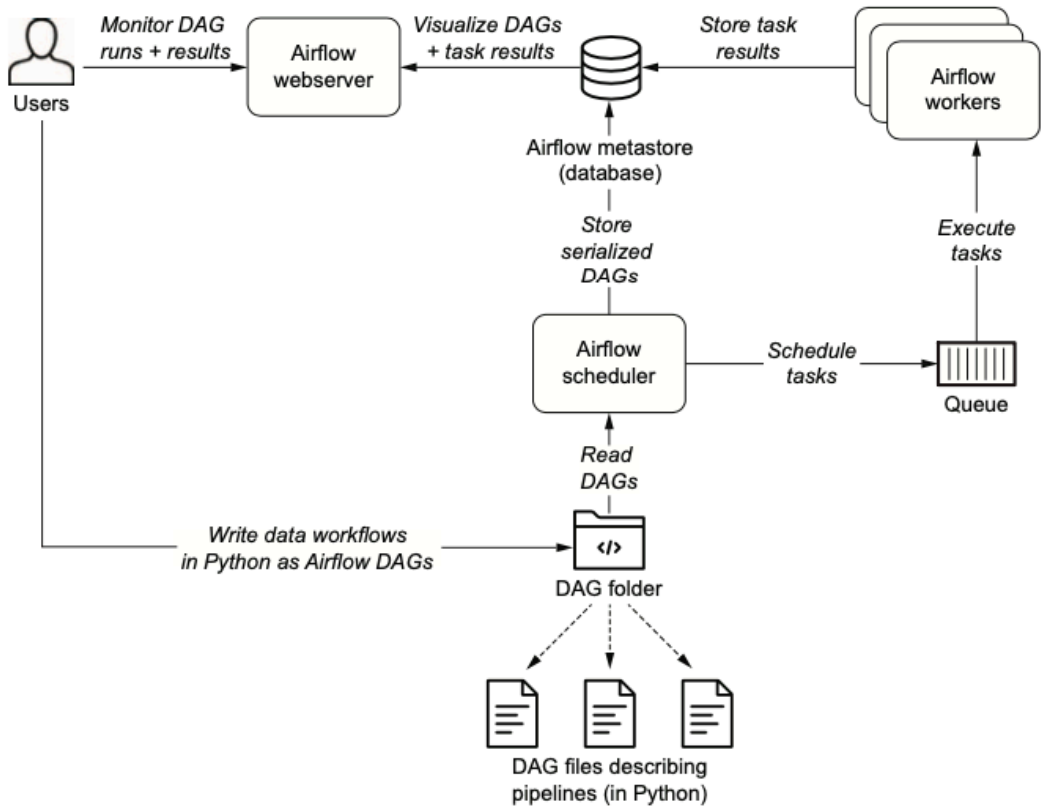



**Figure 1.8 Overview of the main components involved in Airflow (e.g., the Airflow webserver, scheduler, and workers)**

⇒ When not to use airflow:

(1) Handling streaming data

(2) Implementing highly dynamic pipelines, where tasks are added/removed b/w every pipeline run. Airflow can

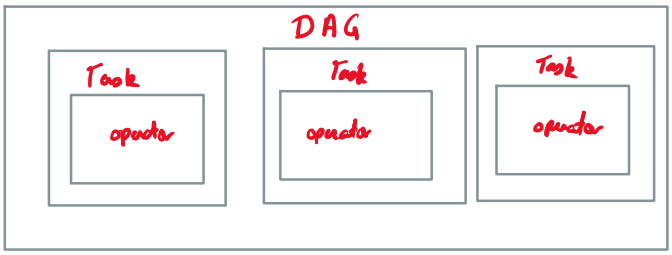
implement dynamic behaviour, but the web-interface will only show tasks that are still defined the most recent version of the DAG.

(3) Teams with no Python experience

# Chapter - 2

## ⇒ Tasks vs Operators

→ In airflow documentation, tasks and operators are used interchangeably. But the difference is; operators provides the implementation of a piece of work. Eg. PythonOperator, EmailOperator etc. Task manages the execution of an operator; they can be thought as wrappers around operator that ensures operators are executed correctly.



# Chapter 3: Scheduling

⇒ If `schedule_interval` is set to none, the DAG is only run if triggered by API or from UI manually.

⇒ Scheduling @daily will run every day at midnight

⇒ Running in regular interval:

(i) `schedule_interval = @daily` → macro based (@daily, @monthly etc)

(ii) Cron based `schedule_interval = 0 0 * * *`

(iii) Frequency based: using time delta to run every 3rd day.

`schedule_interval = dt.timedelta(days=3)`

## 3.3 Processing data incrementally.

⇒ Fetching events incrementally and partitioning data

→ To incrementally load events you can parameterize (for eg) API calls by passing the

execution time to API call so only the data during that interval is fetched. You can use jinja template. Eg:

**Listing 3.6 Using templating for specifying dates (dags/06_templated_query.py)**

```
fetch_events = BashOperator(
    task_id="fetch_events",
    bash_command=(
        "mkdir -p /data && "
        "curl -o /data/events.json "
        "http://localhost:5000/events?"
        "start_date={{execution_date.strftime('%Y-%m-%d')}}"          (1)
        "&end_date={{next_execution_date.strftime('%Y-%m-%d')}}"      (2)
    ),
    dag=dag,
)
```

(1) Formatted execution_date inserted with Jinja templating
(2) next_execution_date holds the execution date of the next interval.

→ To partition data, you can use template dict

**Listing 3.10 Calculating statistics per execution interval (dags/08_templated_path.py)**

```
def _calculate_stats(**context):                                      (1)
    """Calculates event statistics."""
    input_path = context["templates_dict"]["input_path"]              (2)
    output_path = context["templates_dict"]["output_path"]

    Path(output_path).parent.mkdir(exist_ok=True)

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)


calculate_stats = PythonOperator(
    task_id="calculate_stats",
    python_callable=_calculate_stats,
    templates_dict={
        "input_path": "/data/events/{{ds}}.json",                     (3)
        "output_path": "/data/stats/{{ds}}.csv",
    },
    dag=dag,
)
```

(1) Receive all context variables in this dict.
(2) Retrieve the templated values from the templates_dict object.
(3) Pass the values that we want to be templated.

# 3.4 Understanding Airflow execution dates

→ Airflow follows an interval-based approach. This means that

airflow executes at the end of every interval, so we exactly know the start & end date for each interval. For example, if we have @daily interval, then the task for 2023-01-03 will be executed shortly after 2023-01-04 00:00:00. This is because at that point we know, we will no longer be receiving any new data for 2023-01-03. Hence the execution_date for this run will also be 2023-01-03, even though the task was technically run on 2023-01-04 00:00:00. This is different from CRON based scheduling.

## #3.5 Backfilling events

→ You can use `catchup` to backfill events from `start_date` to current time.

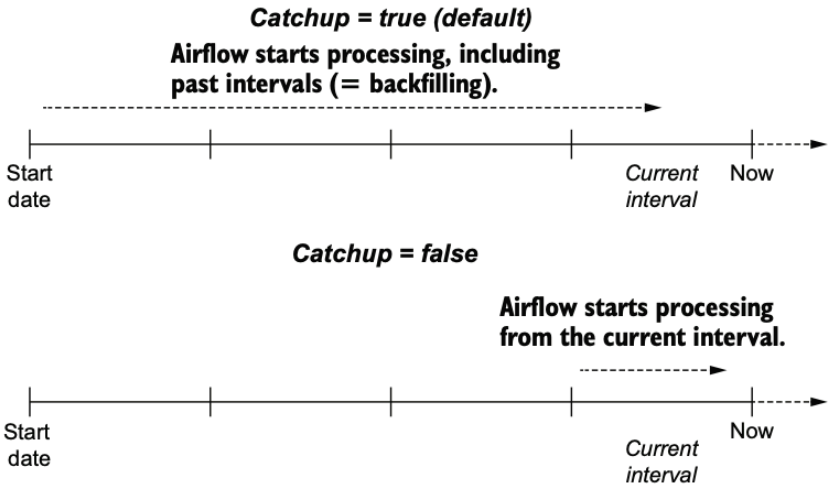



**Figure 3.8 Backfilling in Airflow. By default, Airflow will run tasks for all past intervals up to the current time. This behavior can be disabled by setting the `catchup` parameter of a DAG to false, in which case Airflow will only start executing tasks from the current interval.**

## #3.6 Best practices for designing tasks.

→ A proper airflow task must follow 2 important properties.

(1) Atomicity.

→ A task must be designed such that they either succeed & produce some proper result or fail in a manner that doesn't affect the state of system.

→ This also coincides with the idea that one operation = one specific task.

→ However if there are 2 tasks that are tightly coupled, eg first authenticating API and then fetching data from it, it is better to keep these two in one operation

(2) Idempotency.

→ Means that calling the same task multiple times with the same inputs should have no additional effects.

→ In general the tasks that write data can be made idempotent by checking existing results and making sure previous results are overwritten by the task.

# Chapter 4: Templating

## # Task Context in PythonOperator

-> You have context passed inside each python callable function in the form of **kwargs. This dict provides the context within that callable python function.

**Listing 4.7 Renaming kwargs to context for expressing intent to store task context**

```
def _print_context(**context):      ❶
   print(context)


print_context = PythonOperator(
    task_id="print_context",
    python_callable=_print_context,
    dag=dag,
)
```

❶ Naming this argument context indicates we expect Airflow task context.

The context variable is a dict of all context variables, which allows us to give our task different behavior for the interval it runs in, for example, to print the start and end datetime of the current interval:

**Listing 4.8 Printing start and end date of interval**

```
def _print_context(**context):
   start = context["execution_date"]        ❶
   end = context["next_execution_date"]
   print(f"Start: {start}, end: {end}")


print_context = PythonOperator(
   task_id="print_context", python_callable=_print_context, dag=dag
)

# Prints e.g.:
# Start: 2019-07-13T14:00:00+00:00, end: 2019-07-13T15:00:00+00:00
```

❶ Extract the execution_date from the context.

Or you can also provide named argument

```
_get_data(conf=..., dag=..., dag_run=..., execution_date=..., ...)

def _get_data(execution_date, **context):
    year, month, day, hour, *_ = execution_date.timetuple()
    # ...
```

# Providing variables to PythonOperator.

→ Can use `op_args` to pass positional args.

```
def _get_data(output_path, **context):
    year, month, day, hour, *_ = context["execution_date"].timetuple()
    url = (
        "https://dumps.wikimedia.org/other/pageviews/"
        f"{year}/{year}-{month:0>2}/pageviews-{year}{month:0>2}{day:0>2}-{hour:0>2}0000.gz"
    )
    request.urlretrieve(url, output_path)
```

**output_path now configurable via argument**

**Figure 4.10** The `output_path` is now configurable via an argument.

The value for `output_path` can be provided in two ways. The first is via an argument: `op_args`.

**Listing 4.11 Providing user-defined variables to the `PythonOperator` callable**

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_args=["/tmp/wikipageviews.gz"],      (1)
    dag=dag,
)
```

(1) Provide additional variables to the callable with op_args.

Or you can use `op_kwargs`

**Listing 4.12 Providing user-defined kwargs to callable**

```
get_data = PythonOperator(
    task_id="get_data",
    python_callable=_get_data,
    op_kwargs={"output_path": "/tmp/wikipageviews.gz"},      (1)
    dag=dag,
)
```

(1) A dict given to op_kwargs will be passed as keyword arguments to the callable

A dict given to op_kwargs will be passed as keyword arguments to the callable.

# Inspecting template arguments

→ The rendered values of template (i.e. the actual passed values) are only visible in the UI after the task has started executing.

→ But airflow CLI can render all templated attributes as if the task was run for a desired datetime. Using the CLI doesn't register anything in the metastore so it is lightweight.

**Listing 4.14 Rendering templated values for any given execution date**

```
# airflow tasks render stocksense get_data 2019-07-19T00:00:00
# ----------------------------------------------------------
# property: templates_dict
```



```
# ----------------------------------------------------------
None

# ----------------------------------------------------------
# property: op_args
# ----------------------------------------------------------
[]

# ----------------------------------------------------------
# property: op_kwargs
# ----------------------------------------------------------
{'year': '2019', 'month': '7', 'day': '19', 'hour': '0', 'output_path':
    '/tmp/wikipageviews.gz'}
```

# Passing data between tasks

Airflow's tasks run independently of each other & possibly on different machines. So they can't share objects in memory. Hence data has to be persisted.

(1) By using Airflow's metastore to write and read results but tasks. Called XCom

(2) By writing results to and from a persistent location (eg disk / db) but tasks

# Chapter 5: Dependencies btw tasks

⇒ Basic dependencies

→ Airflow doesn't execute downstream tasks if the upstream tasks fail.

→ Linear Dependencies

```
task_1 >> task_2
```

→ Fan-in/out dependencies

```
[task_1, task_2] >> task_3

task_1 >> [task_2, task_3]
```

→ Create a dummy start task

```
start = DummyOperator(task_id="start")
start >> [task_1, task_2]
```

# Branching

⇒ Branching within tasks

**Listing 5.7 Branching within the cleaning task (dags/02_branch_task.py)**

```
def _clean_sales(**context):
    if context["execution_date"] < ERP_CHANGE_DATE:
        _clean_sales_old(**context)
    else
        _clean_sales_new(**context)

...

clean_sales_data = PythonOperator(
    task_id="clean_sales",
    python_callable=_clean_sales,
)
```

## ⇒ Branching within DAG

→ Suppose branching spans across multiple set of tasks,

→ there is no way to visually see which DAG branch the code is going through. Only way to see is via the logs.

→ The 'BranchPythonOperator' returns the appropriate task_id based on the condition.

**Listing 5.11 Adding the branching condition function (dags/03_branch_dag.py)**

```
def _pick_erp_system(**context):
    if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

pick_erp_system = BranchPythonOperator(
    task_id="pick_erp_system",
    python_callable=_pick_erp_system,
)

pick_erp_system >> [fetch_sales_old, fetch_sales_new]
```

⇒ One problem using BranchPythonOperator is that, since only one task is executed, the downstream will be skipped since the downstream expects both the parents to be complete. Because the downstream of these tasks will look like

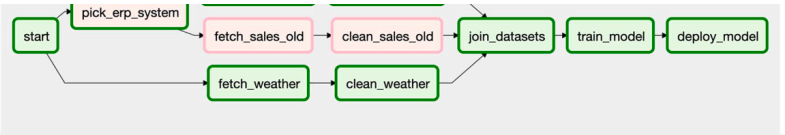
[fetch_sales_old, fetch_sales_new] >> downstream_task

⇒ To overcome this, change the trigger rule to none_failed

**Listing 5.14 Fixing the trigger rule of the join_datasets task (dags/03_branch_dag.py)**

```
join_datasets = PythonOperator(
    ...,
    trigger_rule="none_failed",
)
```

This way, `join_datasets` will start executing as soon as all of its parents have finished executing without any failures, allowing it to continue its execution after the branch (figure 5.9).




One drawback of this approach is that we now have three edges going into the `join_datasets` task. This doesn't really reflect the nature of our flow, in which we essentially want to fetch sales/weather data (choosing between the two ERP systems first) and then feed these two data sources into `join_datasets`. For this reason, many people choose to make the branch condition more explicit by adding a dummy task that joins the different branches before continuing with the DAG (figure 5.10).

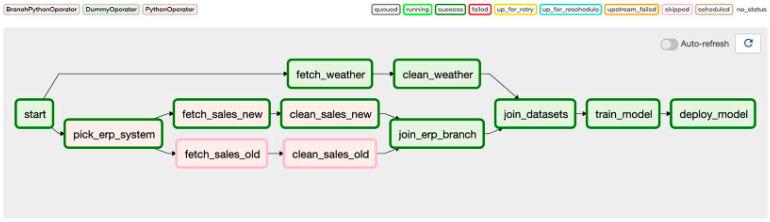



**Figure 5.10 To make the branching structure more clear, you can add an extra join task after the branch, which ties the lineages of the branch together before continuing with the rest of the DAG. This extra task has the added advantage that you don't have to change any trigger rules for other tasks in the DAG, as you can set the required trigger rule on the join task. (Note that this means you no longer need to set the trigger rule for the `join_datasets` task.)**

To add such a dummy task to our DAG, we can use the built-in `DummyOperator` provided by Airflow.

**Listing 5.15 Adding a dummy join task for clarity (dags/04_branch_dag_join.py)**

```
from airflow.operators.dummy import DummyOperator

join_branch = DummyOperator(
    task_id="join_erp_branch",
    trigger_rule="none_failed"
)

[clean_sales_old, clean_sales_new] >> join_branch
join_branch >> join_datasets
```

This change also means that we no longer need to change the trigger rule for the `join_datasets` task, making our branch more self-contained than the original.

# Conditional Tasks.

→ We can make certain tasks run only if certain datasets are available or only if the DAG is executed for the most recent execution date.

⇒ Conditioning within task

```
def _deploy(**context):
    if context["execution_date"] == ...:
        deploy_model()

deploy = PythonOperator(
    task_id="deploy_model",
```

→ not the best option because it is not explicit when seen on UI and it hides [illegible] logic from

```
    python_callable=_deploy,
)
```

business logic from the task

## ⇒ Making tasks Conditional

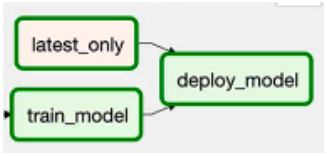
→ Make a task conditional by adding another task that that tests the said condition and ensures any downstream tasks are skipped if condition fails.

```
def _latest_only(**context):
    ...

latest_only = PythonOperator(
    task_id="latest_only",
    python_callable=_latest_only,
    dag=dag,
)

latest_only >> deploy_model
```

→ This makes the implementation much more explicit




→ Raise an exception in the _latest_only function. This is Airflow's way of saying to skip the downstream tasks.

## ⇒ Using built-in Operators

→ Use LatestOnlyOperator

```
from airflow.operators.latest_only import LatestOnlyOperator

latest_only = LatestOnlyOperator(
    task_id="latest_only",
    dag=dag,
)

train_model >> latest_only >> deploy_model
```

# Trigger Rules.

In essence, when Airflow is executing a DAG, it continuously checks each of your tasks to see whether it can be executed. As soon as a task is deemed ready for execution, it is picked up by the scheduler and scheduled to be executed. As a result, the task is executed as soon as Airflow has an execution slot available.

Trigger rules are essentially the conditions that Airflow applies to tasks to determine whether they are ready to execute, as a function of their dependencies (= preceding tasks in the DAG).

→ When an upstream tasks fails, it sets the state of the downstream task as upstream_fail, which indicates that downstream task can't be executed with default trigger rule.

**Table 5.1 An overview of the different trigger rules supported by Airflow**

| Trigger rule | Behavior | Example use case |
|---|---|---|
| `all_success` (default) | Triggers when all parent tasks have been completed successfully | The default trigger rule for a normal workflow |
| `all_failed` | Triggers when all parent tasks have failed (or have failed as a result of a failure in their parents) | Trigger error handling code in situations where you expected at least one success among a group of tasks |
| `all_done` | Triggers when all parents are done with their execution, regardless of their resulting state | Execute cleanup code that you want to execute when all tasks have finished (e.g., shutting down a machine or stopping a cluster) |
| `one_failed` | Triggers as soon as at least one parent has failed; does not wait for other parent tasks to finish executing | Quickly trigger some error handling code, such as notifications or rollbacks |
| `one_success` | Triggers as soon as one parent succeeds; does not wait for other parent tasks to finish executing | Quickly trigger downstream computations/notifications as soon as one result becomes available |
| `none_failed` | Triggers if no parents have failed but have either completed successfully or been skipped | Join conditional branches in Airflow DAGs, as shown in section 5.2 |
| `none_skipped` | Triggers if no parents have been skipped but have either completed successfully or failed | Trigger a task if all upstream tasks were executed, ignoring their result(s) |
| `dummy` | Triggers regardless of the state of any upstream tasks | Testing |

# Sharing data between tasks

## ⇒ Using XComs

→ XComs allow to exchange messages between tasks.
→ You can push values to XComs explicitly using the xcom_push

```
def _train_model(**context):
    model_id = str(uuid.uuid4())
    context["task_instance"].xcom_push(key="model_id", value=model_id)
```

→ Or do it implicitly by simply letting your python function return the xcom value.

```
def _train_model(**context):
    model_id = str(uuid.uuid4())
    return model_id
```

→ To use the value of the xcom variable use the xcom_pull.

```
def _deploy_model(**context):
    model_id = context["task_instance"].xcom_pull(
        task_ids="train_model", key="model_id"
    )
    print(f"Deploying model {model_id}")
```

This tells Airflow to fetch the XCom value with key `model_id` from the `train_model` task. Note that `xcom_pull` also allows you to define the `dag_id` and execution date when fetching XCom values. By default, these parameters are set to the current DAG and execution date so that `xcom_pull` only fetches values published by the current DAG run.

→ XCom creates an implicit dependency between tasks which is not accounted for when scheduling the tasks and it is also not visible in the DAG.
→ Also XCom variables HAVE to be serialized & have size limit depending on the db.

## # Taskflow API

⇒ Limitations with existing approach

→ We first need to define functions and then wrap in the PythonOperator to create task.
→ Explicitly push and pull xcoms.

⇒ Task API

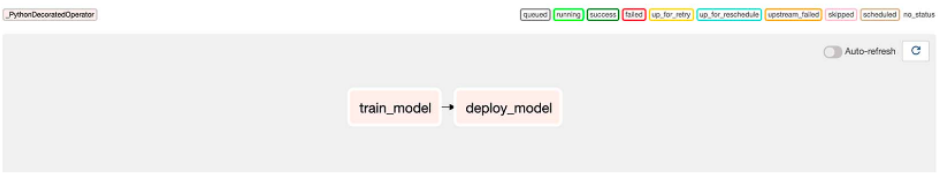
→ The task API makes it simpler to declare DAGs and let use of variables more explicit w.r.t. DAG structure and more implicit for programmers.

```
with DAG(...) as dag:
    ...
    @task
    def train_model():
        model_id = str(uuid.uuid4())
        return model_id

@task
def deploy_model(model_id):
    print(f"Deploying model {model_id}")

model_id = train_model()
deploy_model(model_id)
```

This code should result in a DAG with two tasks (`train_model` and `deploy_model`) and a dependency between the two tasks (figure 5.18).




In essence, when we call the decorated `train_model` function, it creates a new operator instance for the `train_model` task (shown as the

_PythonDecoratedOperator in figure 5.18). From the return statement in the `train_model` function, Airflow recognizes that we are returning a value that will automatically be registered as an XCom returned from the task. For the `deploy_model` task, we also call the decorated function to create an operator instance, but now also pass along the `model_id` out- put from the `train_model` task. In doing so, we're effectively telling Airflow that the `model_id` output from `train_model` should be passed as an argument to the deco- rated `deploy_model` function. This way, Airflow will both realize there is a depen- dency between the two tasks and take care of passing the XCom values between the two tasks for us.

⇒ Limitations:
- → Supports only python operators
- → However you can mix-merge the 2 styles if using non-python operators.
- → The size of data sets passed b/w tasks will be limited because Task API uses XCom internally.

# Chapter 6

Tuesday, November 21, 2023 10:49 AM

## # 6.1 Polling Conditions with Sensors

→ To trigger task on some conditions, use sensors. Sensors continuously poll for certain condition and succeed if so. If false, sensor will wait until either the condition is true or a timeout is eventually reached. This in Airflow language called poking. i.e. running a sensor & checking sensor conditions.

→ If using sensor, start time of DAG must be before the expected condition.

⇒ Polling custom condition

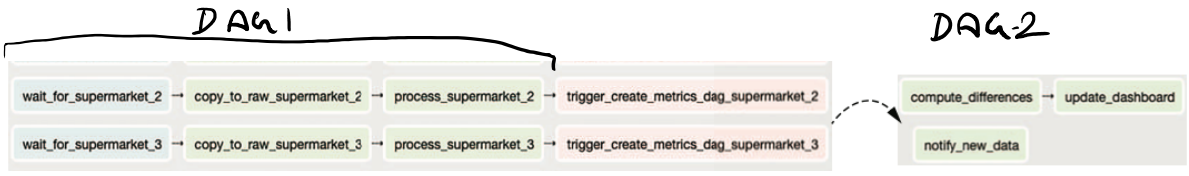
→ PythonSensor supports a python callable and expects a boolean return value.

# 6.2 Triggering other DAGs

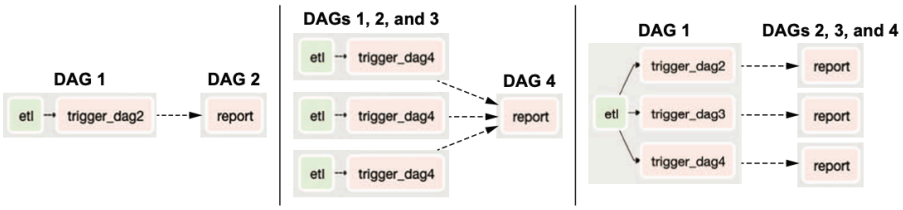
→ TriggeredDagRunOperator allows triggering other DAGs which can help decouple workflows

```
dag1 = DAG(
    dag_id="ingest_supermarket_data",
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval="0 16 * * *",
)
# ...
trigger_create_metrics_dag = TriggerDagRunOperator(
    task_id=f"trigger_create_metrics_dag_supermarket_{supermarket_id}",
    trigger_dag_id="create_metrics",  (1)
    dag=dag1,
)
```

```
dag2 = DAG(
    dag_id="create_metrics",  (1)
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval=None,  (2)
)
```



→ You can also have multiple DAGs triggered.




↓

this will trigger the report if single of the upstream etl dag is triggered. But to trigger report if all of the upstream DAGs are completed then a workaround is to use External Task Sensor

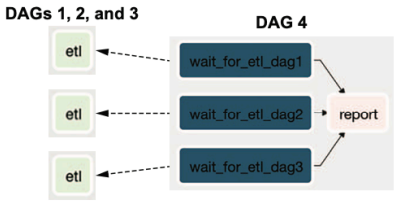



**Figure 6.19 Instead of pushing execution with the `TriggerDagRunOperator`, in some situations such as ensuring completed state for DAGs 1, 2, and 3, we must pull execution toward DAG 4 with the `ExternalTaskSensor`.**

The way the `ExternalTaskSensor` works is by pointing it to a task in another DAG to check its state (figure 6.20).

→ Starting workflows with REST/CLI

→ Can use REST API / CLI to trigger DAGs.