Best Practices in Apache Airflow :-

Airflow is 100% code, Basics Python knowledge is required to get started writing DAGs.

However, writing DAGs that are efficient, secure, and scalable requires some Airflow-specific skill.

1. Keep Tasks Atomic
2. Use Template Fields, Variables, and Macros
3. Incremental Record Filtering
4. Avoid code that makes request to external systems
5. Use a Consistent Method for Task Dependencies
6. Use an ELT (Extract load transform) Framework
7. Use Intermediary Data Storage
8. Use a Consistent File Structure
9. Use DAG Name and Start Date Properly
10. Retries range
11. Since Airflow was developed at Airbnb where all systems run on UTC, make sure to run all systems on UTC time.
12. Make sure daily tasks run when the day is over, not at the start of the day.

Keep Tasks Atomic:-

This means each task should be responsible for one operation that can be re-run independently of the others.

Its mean a success in part of the task means a success of the entire task.

For example, in an ETL pipeline you would ideally want your Extract, Transform, and Load operations covered by three separate tasks.
Atomizing these tasks allows you to rerun each operation in the pipeline independently.

Use Template Fields, Variables, and Macros:-

By using template fields in Airflow, you can pull values into DAGs using environment variables and jinja template.

Compared to using Python functions, using template fields helps keep your DAGs idempotent and ensures you aren't executing functions on every Scheduler heartbeat (see "Avoid Top Level Code in Your DAG File" for more about Scheduler optimization).

following example defines variables based on datetime Python functions:

```
# Variables used by tasks
# Bad example - Define today's and yesterday's date using datetime module
today = datetime.today()
yesterday = datetime.today() - timedelta(1)
```

If this code is in a DAG file, these functions will be executed on every Scheduler heartbeat, which may not be performant.

Even more importantly, this doesn't produce an idempotent DAG: If you needed to rerun a previously failed DAG Run for a past date, you wouldn't be able to because datetime.today() is relative to the current date, not the DAG execution date.

A better way of implementing this is by using an Airflow variable:

```
# Variables used by tasks
# Good example - Define yesterday's date with an Airflow variable
yesterday = {{ yesterday_ds_nodash }}
```

Best Practices in Apache Airflow Part2:-

Incremental Record Filtering:-

It is ideal to break out your pipelines into incremental extracts and loads wherever possible.

For example, if you have a DAG that runs hourly, each DAG Run should process only records from that hour, rather than the whole dataset.

When the results in each DAG Run represent only a small subset of your total dataset, a failure in one subset of the data won't prevent the rest of your DAG Runs from completing successfully.

And if your DAGs are idempotent, you can rerun a DAG for only the data that failed rather than reprocessing the entire dataset.

There are multiple ways you can achieve incremental pipelines. The two best and most common methods are described below.

Last Modified Date:-

Using a "last modified" date is the gold standard for incremental loads. Ideally, each record in your source system has a column containing the last time the record was modified.

With this design, a DAG Run looks for records that were updated within specific dates from this column.

For example, with a DAG that runs hourly, each DAG Run will be responsible for loading any records that fall between the start and end of its hour. If any of those runs fail, it will not impact other Runs.

Sequence IDs:-
When a last modified date is not available, a sequence or incrementing ID can be used for incremental loads.

This logic works best when the source records are only being appended to and never updated.

While we recommend implementing a "last modified" date system in your records if possible, basing your incremental logic off of a sequence ID can be a sound way to filter pipeline records without a last modified date.

Avoid code that makes request to external systems:-

Code that makes requests to external systems, like an API or a database, or makes function calls outside of your tasks can cause performance issues.

Additionally, including code that isn't part of your DAG or operator instantiations in your DAG file makes the DAG harder to read, maintain, and update.

Because Airflow executes all code in the DAGS_Folder on every scheduler heartbeat,

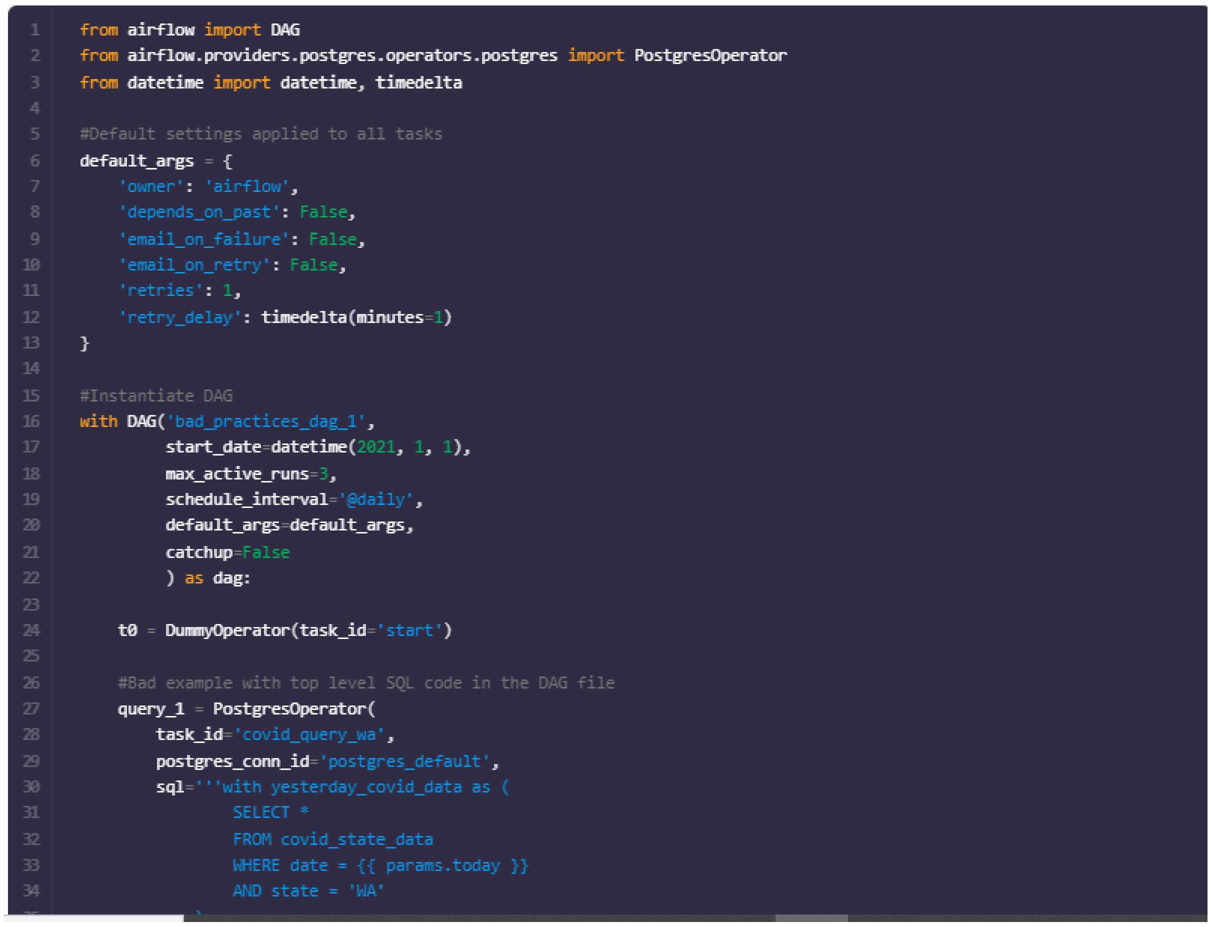
Treat your DAG file like a config file and leave all of the heavy lifting to the hooks and operators that you instantiate within the file.

If your DAGs need to access additional code such as a SQL script or a Python function, keep that code in a separate file that can be read into a DAG Run.

In first screenshot DAG below a PostgresOperator executes a SQL query that was dropped directly into the DAG file, we need to avoid that.

Keeping the query in the DAG file like this makes the DAG harder to read and maintain.

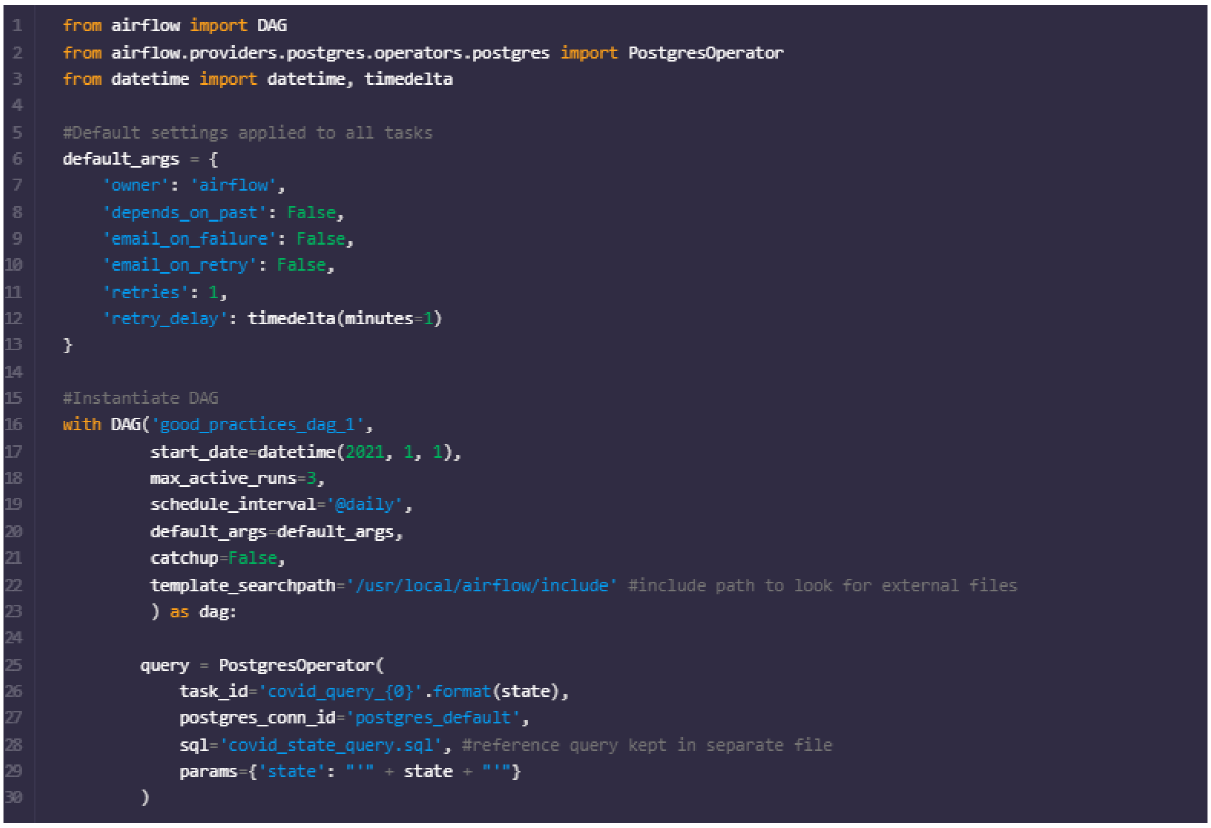
Screenshot1:-



```python
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
from datetime import datetime, timedelta

#Default settings applied to all tasks
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

#Instantiate DAG
with DAG('bad_practices_dag_1',
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval='@daily',
         default_args=default_args,
         catchup=False
         ) as dag:

    t0 = DummyOperator(task_id='start')

    #Bad example with top level SQL code in the DAG file
    query_1 = PostgresOperator(
        task_id='covid_query_wa',
        postgres_conn_id='postgres_default',
        sql='''with yesterday_covid_data as (
                SELECT *
                FROM covid_state_data
                WHERE date = {{ params.today }}
                AND state = 'WA'
```

Instead, in the DAG below we call in a file named `covid_state_query.sql` into our PostgresOperator instantiation, which embodies the best practice and run code like below screenshot2.

Screenshot 2:-



```python
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
from datetime import datetime, timedelta

#Default settings applied to all tasks
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

#Instantiate DAG
with DAG('good_practices_dag_1',
         start_date=datetime(2021, 1, 1),
         max_active_runs=3,
         schedule_interval='@daily',
         default_args=default_args,
         catchup=False,
         template_searchpath='/usr/local/airflow/include' #include path to look for external files
         ) as dag:

        query = PostgresOperator(
            task_id='covid_query_{0}'.format(state),
            postgres_conn_id='postgres_default',
            sql='covid_state_query.sql', #reference query kept in separate file
            params={'state': "'" + state + "'"}
        )
```

Best Practices in Apache Airflow Part3:-

Use a Consistent Method for Task Dependencies:-

Same work can be done by multiple ways, So it's better Which method you use should be same across all the DAG programs,.

otherwise it will create confusion, but for readability it's best practice to choose one method and stick with it

For example
task dependencies can be set multiple ways
You can use set_upstream() and set_downstream() functions, or you can
use << and >> operators.

For example, instead of mixing methods like this

task_1.set_downstream(task_2)
task_3.set_upstream(task_2)
task_3 >> task_4

Try to be consistent with something like this
task_1 >> task_2 >> [task_3, task_4]

Don't Use Airflow as a Processing Framework(Try to avoid processing work in airflow)

Airflow was not designed to be a processing framework

Since DAGs are written in Python, it can be tempting to make use of data processing libraries like Pandas

Use apache spark for processing

Use airflow for processing where the data are limited in size

Use an ELT (Extract load transform) Framework:-

Whenever possible, look to implement an ELT (extract, load, transform) data pipeline pattern with your DAGs

This means that you should look to offload as much of the transformation logic to the source systems or the destinations systems as possible, which avoids using Airflow as a processing framework

Use Intermediary Data Storage:-

Because it requires less code and fewer pieces, it can be tempting to write your DAGs to move data directly from your source to destination

However, this means you can't individually rerun the extract or load portions of the pipeline

By putting an intermediary storage layer such as S3 or SQL Staging tables in between your source and destination, you can separate the testing and rerunning of the extract and load

This is also useful in situations where you no longer have access to the source system

Use a Consistent File Structure:-

Having a consistent file structure for Airflow projects keeps things organized and easy to adopt

```
├── dags/ # Where your DAGs go
│   ├── example-dag.py # An example dag that comes with the initialized project
```

```
├── Dockerfile # For Docker image and runtime overrides
├── include/ # For any other files you'd like to include
├── plugins/ # For any custom or community Airflow plugins
├── packages.txt # For OS-level packages
└── requirements.txt # For any Python packages
```

Use DAG Name and Start Date Properly:-

You should always use a static start_date with your DAGs. A dynamic start_date is misleading, and can cause failures when clearing out failed task instances and missing DAG runs

Additionally if you change the start_date of your DAG you should also change the DAG name

Changing start_date of a DAG creates a new entry in Airflow's database which could confuse the scheduler because there will be two DAGs with the same name but different schedules

Retries range:-

A good range is ~2–4 retries

- Since Airflow was developed at Airbnb where all systems run on UTC, make sure to run all systems on UTC time ...

  Webserver

  Metadata DB

  Scheduler

  Workers

- Make sure daily tasks run when the day is over, not at the start of the day.