Apache Airflow

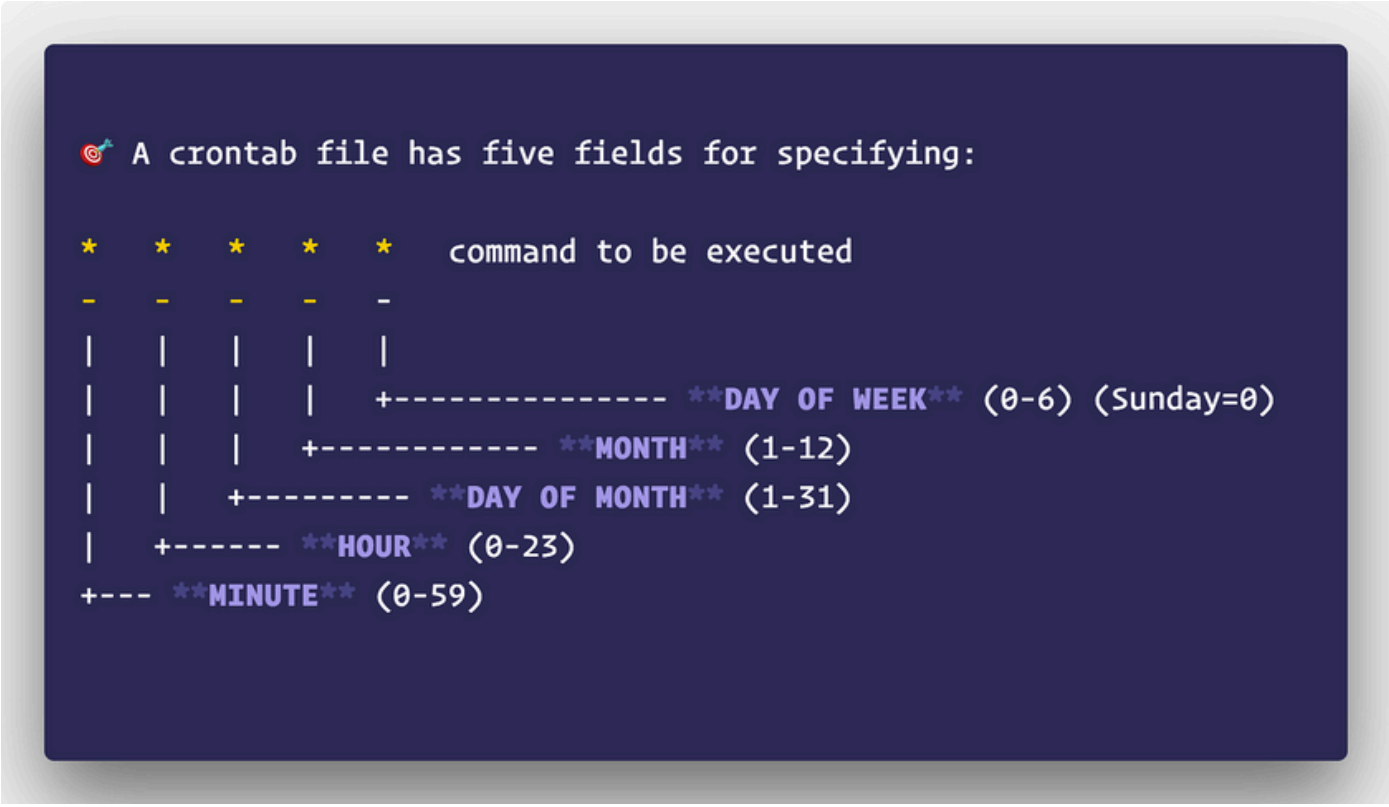
Apache Airflow Understanding Cron

In today’s data-driven world, managing complex workflows and orchestrating data pipelines is key to business success. As organizations handle increasing volumes of data, the need for automation and efficient data management tools becomes more important. Apache Airflow has emerged as a vital tool in the realm of data engineering for exactly this purpose. Even if you have zero knowledge about Airflow, follow along. I will try to explain everything about it from scratch, and by the end, you will have enough knowledge to build your own cool projects with Apache Airflow. Let’s get started.

Understanding Cron: The Backbone of Task Scheduling

In the world of automation, cron is one of the most fundamental and powerful tools used to schedule recurring tasks. From system maintenance to data processing pipelines, cron jobs ensure that tasks run on time without manual intervention. If you're working with Apache Airflow, you'll encounter cron expressions often as they play a significant role in defining when your workflows (or DAGs) should execute. In this blog, we'll dive into the basics of cron, how it works, and why it's essential for automation, particularly in Airflow.

What is cron?



It is not as complicated as it sounds. Cron is a time-based job scheduler in Unix-like operating systems. It automates the execution of scripts, commands, or programs at scheduled times or intervals. Cron is perfect for tasks like system backups, database updates, or even monitoring log files. Essentially, cron allows you to focus on more important tasks by automating routine processes.

It works through a background service called the cron daemon. This daemon checks for any scheduled tasks by reading from the crontab, a file where all cron jobs are listed, and executes them at the right time.

```
* * * * * command-to-be-executed
```

Each * in this expression represents a time value, which can be adjusted to schedule jobs with precision:

Minute: (0–59) — the minute of the hour

Hour: (0–23) — the hour of the day

Day of the month: (1–31) — the day of the month

Month: (1–12) — the month of the year

Day of the week: (0–7) — the day of the week (0 or 7 is Sunday)

Examples of cron jobs

1. Run a task every day at midnight

0 0 * * * /example/mytask.sh

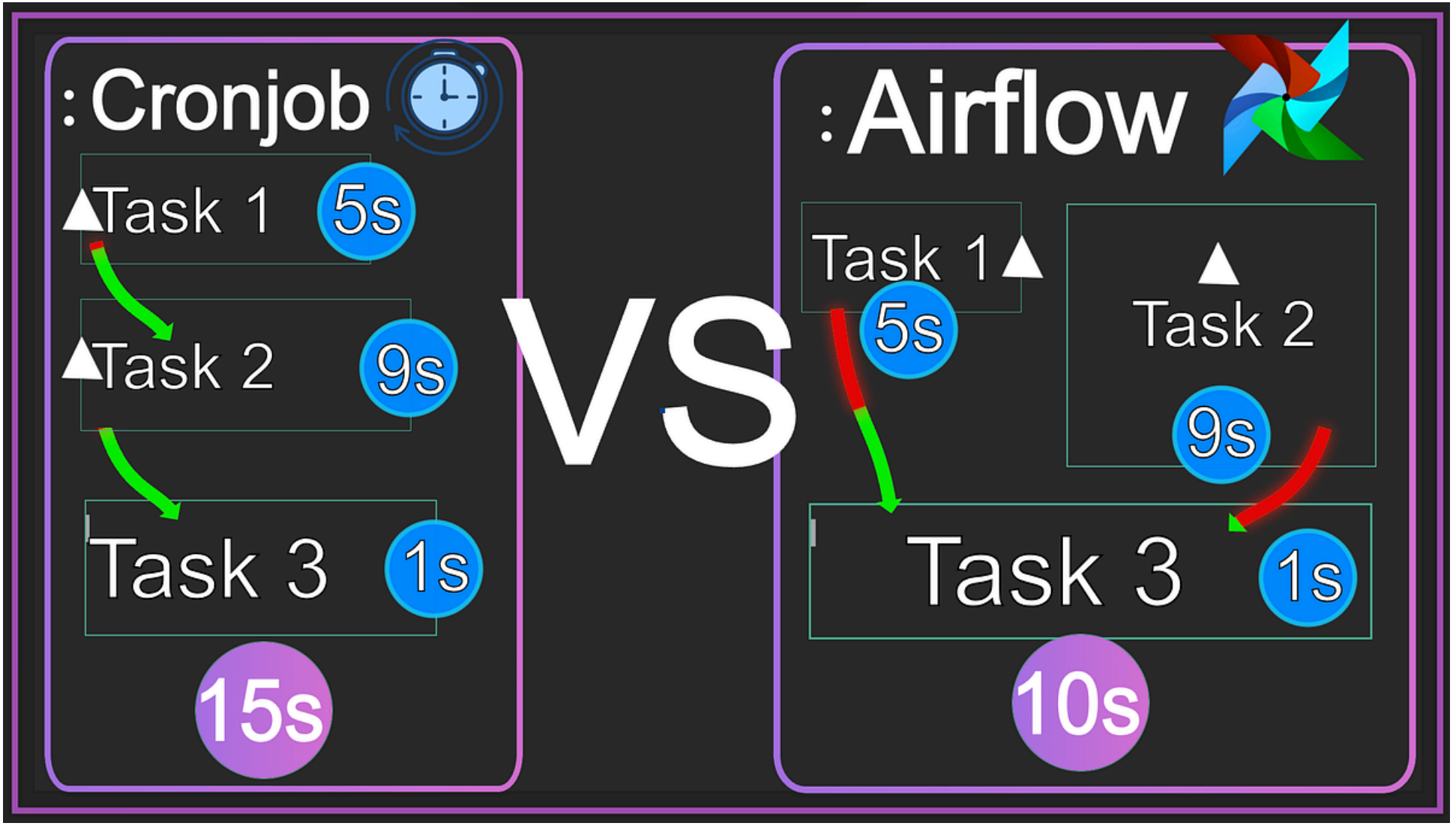
2. Run a task every Tuesday at 10 AM

0 10 * * 2 /example/mytask.sh

3.Run a task every 15 mins

/15 * * * * /example/mytask.sh

Cronjob vs Airflow



Cron in Airflow

While cron is powerful on its own, it becomes even more useful when paired with workflow automation tools like Apache Airflow. Airflow uses cron expressions to define the schedules of workflows, also known as DAGs (Directed Acyclic Graphs). A typical cron expression in an Airflow DAG would look something like this:

```
dag = DAG( ‘example_dag’,
default_args=default_args,
schedule_interval=’0 6 * * *’,
# Runs every day at 6:00 AM)
```

This simple cron expression tells Airflow to run the example_dag workflow at 6:00 AM every day. By using cron in Airflow, you gain fine control over when workflows are executed and can easily manage complex scheduling needs.

Why Cron Matters in Automation

In a world where time is money, automation is essential for efficiency, and cron is one of the oldest and most reliable tools for scheduling tasks. Whether you're using it directly in a Unix system or within a platform like Airflow, cron simplifies the execution of regular processes, ensuring that they are always performed on time.

For those using Airflow, understanding cron is crucial for defining task schedules and building efficient workflows. While Airflow handles the orchestration of tasks, cron ensures that these workflows are triggered at precisely the right moments. Feel free to explore more about cron expressions. I will see you in the next blog.

> It works through a background service called the cron daemon. This daemon checks for any scheduled tasks by reading from the crontab, a file where all cron jobs are listed, and executes...

Introduction to Apache Airflow

We’ll explore Apache Airflow, a powerful open-source tool used for automating, scheduling, and managing workflows. Airflow has gained widespread adoption in the data engineering community for orchestrating complex workflows, especially when dealing with large-scale data processing.

Before we start, I would like to say that there may be parts that you do not completely understand right now. But don’t worry, we will explore each topic in detail in the coming blogs. And if you are really curious, explore it on the internet. Just don’t give up. Let’s get into it.

What is Apache Airflow?

Apache Airflow is a workflow orchestration platform that helps you programmatically author, schedule, and monitor workflows. This might sound a bit too fancy, but in simple words, think of Airflow as a tool that helps you create and schedule tasks. This task can be anything based on your requirements. You might think that we can create and automate tasks with simple python scripts. You are not wrong. But there is a lot more Airflow brings to the table, and we will learn everything about it.

These workflows are typically defined as Directed Acyclic Graphs (DAGs), where each node in the graph represents a task, and edges define dependencies between these tasks.

At its core, Airflow is designed to help you automate and manage ETL (Extract, Transform, Load) processes, data pipelines, and any kind of batch processing.

Key Features of Apache Airflow:

- Scalability: Airflow can handle thousands of tasks in a distributed environment.
- Extensibility: You can integrate with external systems and write custom plugins.
- Dynamic Workflow Creation: DAGs in Airflow are defined using Python, allowing flexibility and dynamic generation of workflows.
- Monitoring & Logging: Airflow provides a web-based UI for monitoring workflow execution and viewing logs.

Why Use Apache Airflow?

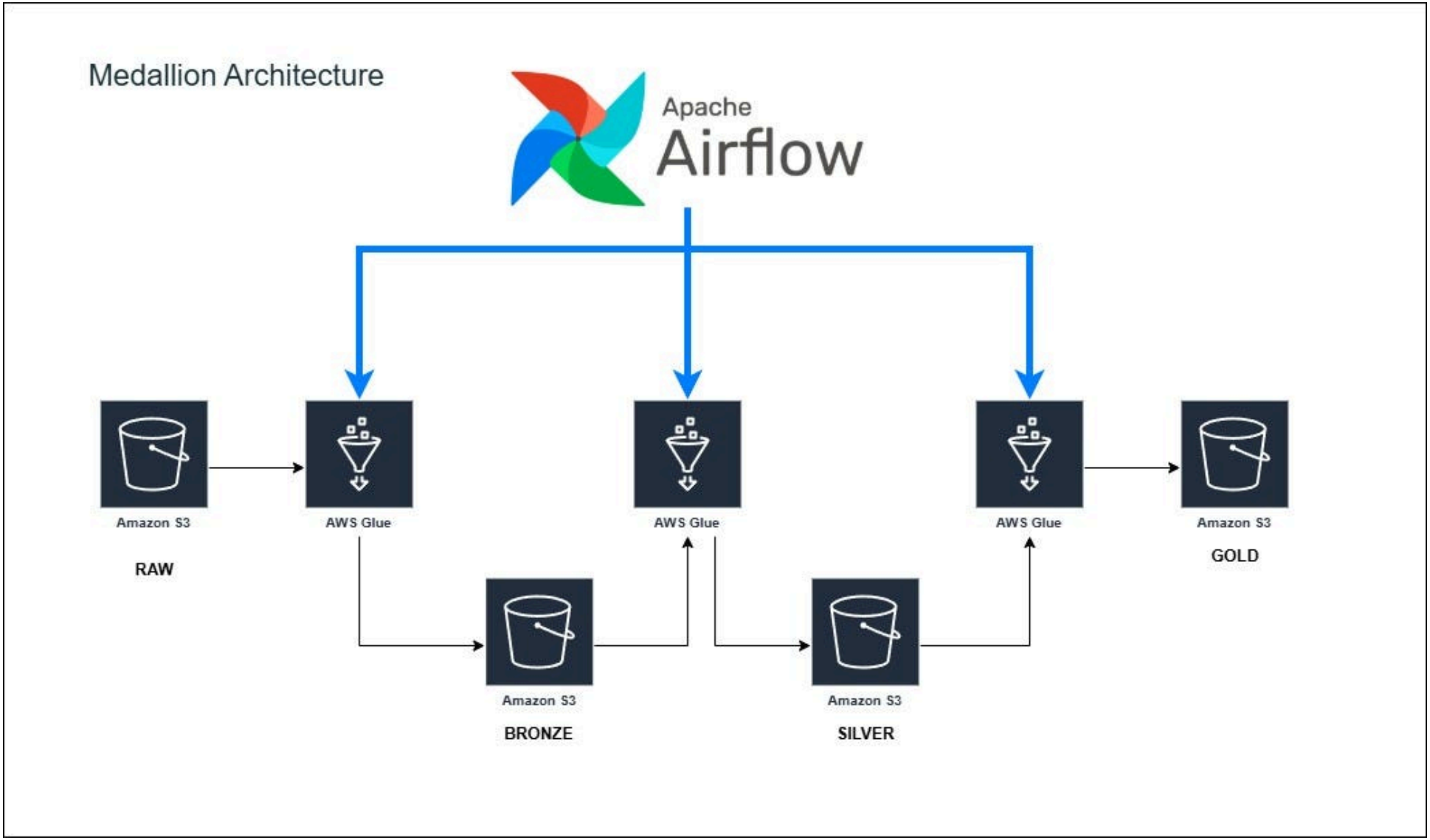
1. Automation: Automates complex workflows by scheduling tasks and ensuring that they run in the correct order based on defined dependencies.

2. Visibility: With Airflow's UI, you can easily monitor the status of each task, check for failures, and even trigger tasks manually.

3. Reusability: You can reuse workflows across different environments or teams, making it highly scalable for large organizations.

4. Community Support: As an open-source project, Airflow has a large community that continually contributes plugins, operators, and improvements.

Airflow in the Data Engineering Ecosystem

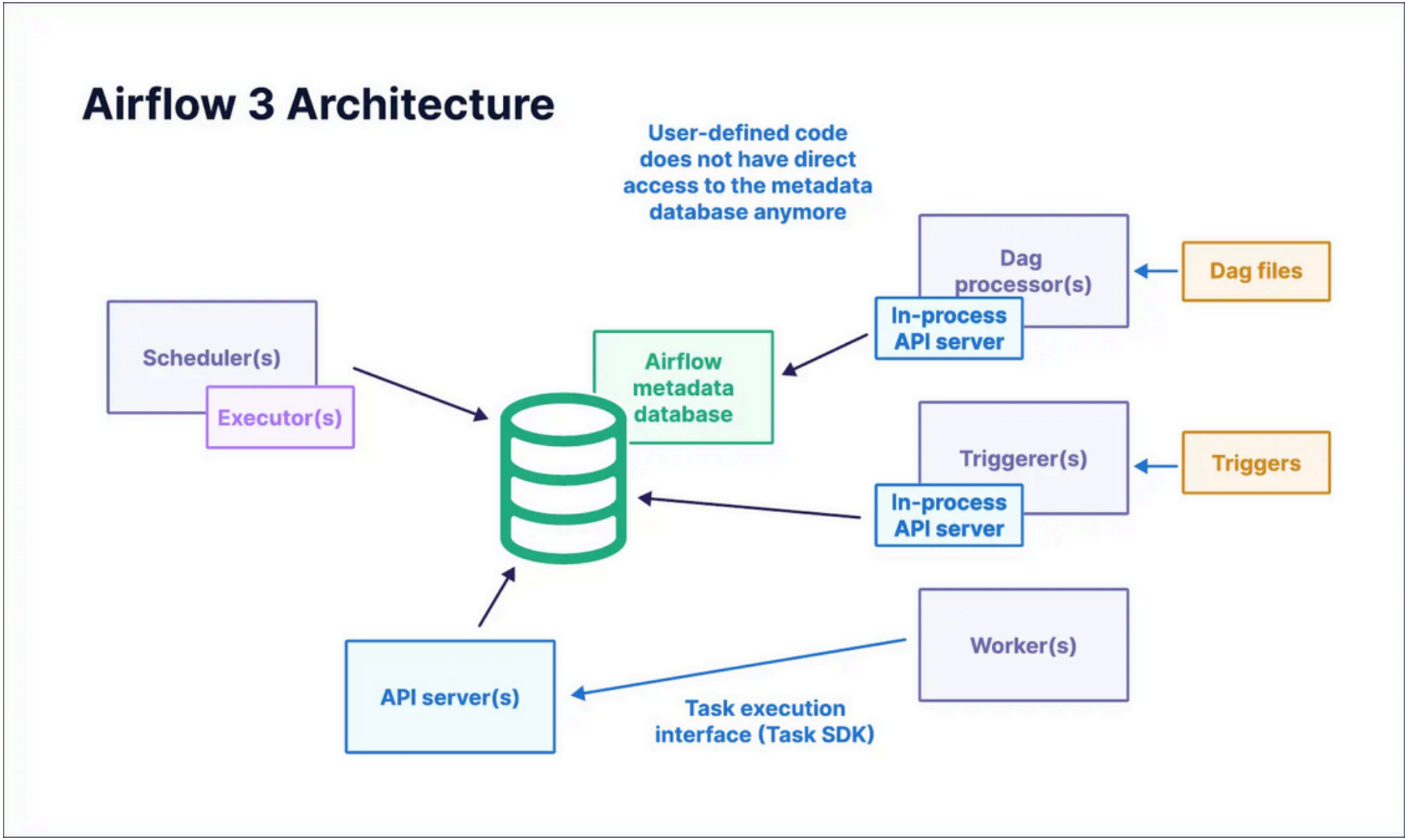


Airflow plays a crucial role in data engineering, especially for orchestrating and automating data pipelines. In a typical data pipeline, data is extracted from a source, transformed into a usable format, and loaded into a target system (like a database or a data warehouse). Airflow is the glue that ensures all these steps happen in the right order.

Some common use cases in data engineering include:

- Scheduling and orchestrating ETL pipelines.
- Managing machine learning workflows.
- Running periodic data quality checks.
- Integrating with cloud services (e.g., AWS, GCP, Azure) to move and process data.

Components of Apache Airflow



Apache Airflow is built around several core components that work together to orchestrate workflows. We will explore more about these individually in the coming blogs, so don't worry about them too much right now.

• **DAG (Directed Acyclic Graph):** Represents the workflow or pipeline, where each node is a task, and edges define the dependencies between tasks. A DAG defines how and when a task should run.

• **Task:** A unit of work in a DAG. It can be anything from running a Python script, executing a SQL query, or even sending an email.

· **Scheduler:** Responsible for scheduling tasks based on the defined DAGs and triggering task execution at the right time.

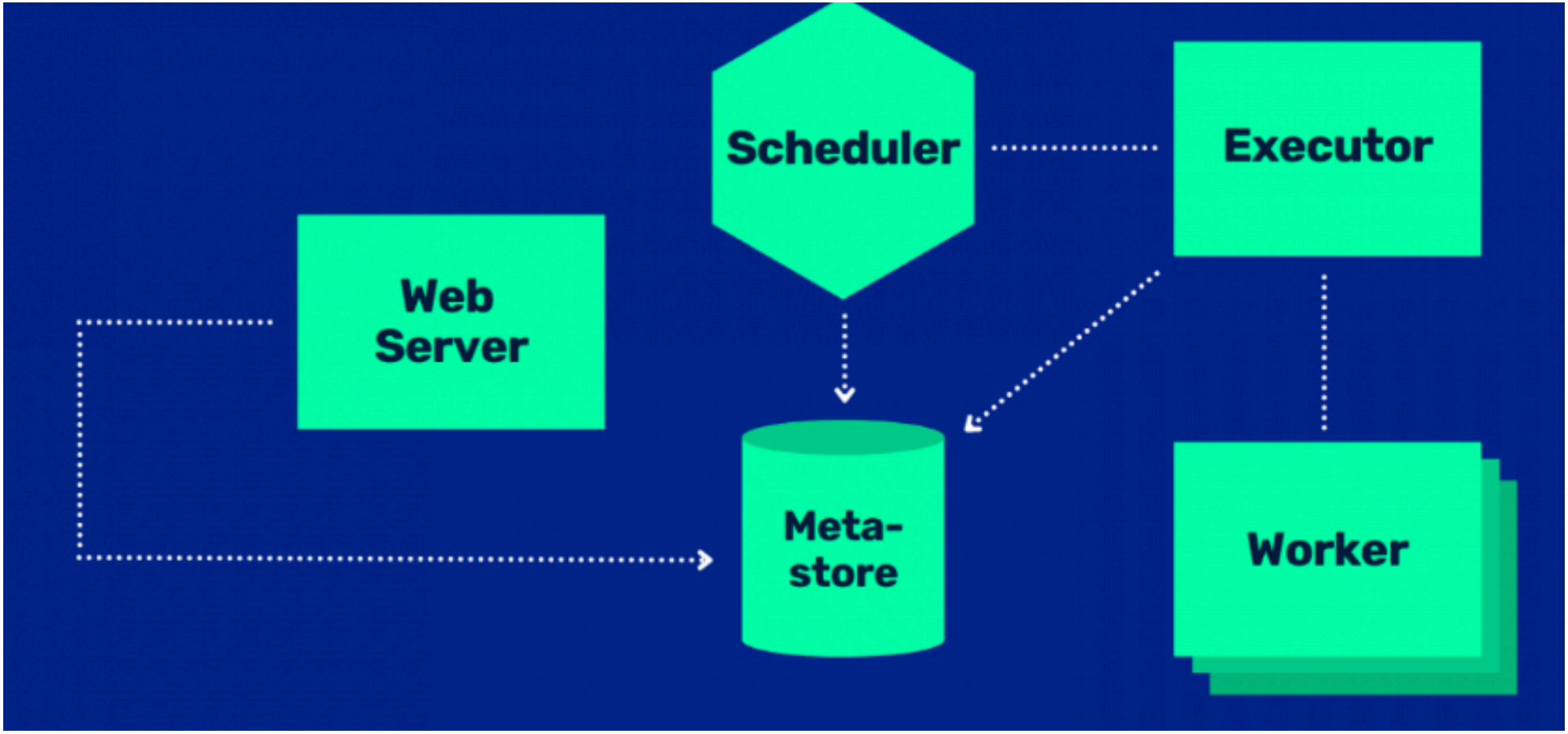
• **Executor:** Defines how tasks are executed (locally, on a remote machine, or distributed across a cluster).

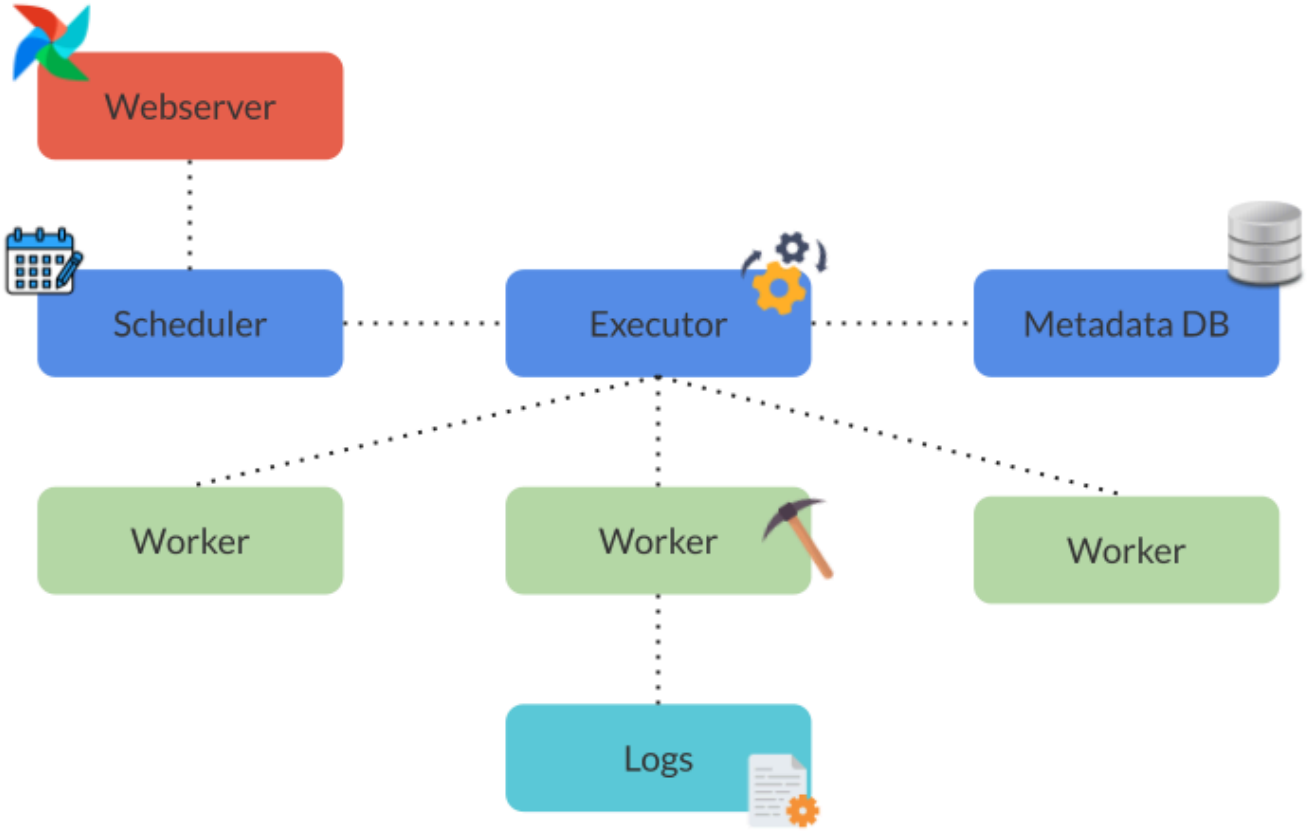
· **Web UI:** A user-friendly interface that allows you to monitor DAGs, trigger tasks manually, and check task logs.

Apache Airflow is a versatile and powerful tool for automating and orchestrating workflows, especially in the data engineering field. By allowing you to define complex workflows with clear dependencies, it simplifies the management of data pipelines and ensures that everything runs smoothly. I hope you now have a good understanding of what Apache Airflow is, and why it is crucial in the realm of data engineering. I will see you in the next blog. Until then, stay healthy and keep learning!

The Core Components of Apache Airflow

We explored what Apache Airflow is and how it fits into the world of workflow orchestration. We'll take a closer look at the core components that make up Airflow's architecture. Understanding these components is essential for mastering Airflow and building efficient, scalable workflows.





Airflow Architecture

1. DAG (Directed Acyclic Graph)

At the heart of Airflow lies the DAG, which stands for Directed Acyclic Graph. A DAG represents the workflow you want to schedule and run. Each DAG is composed of tasks, which define the work to be done. The directed and acyclic nature of the graph means that the tasks flow in a specific direction (e.g., from one task to another), and there are no cycles (a task cannot depend on itself).

DAGs are defined in Python, and they are designed to be dynamic, meaning they can be programmatically generated, modified, or extended. This flexibility is one of Airflow's strongest features.

2. Tasks and Task Instances

Each DAG consists of tasks, which are the individual units of work. A task in Airflow is defined by an operator, which determines the nature of the work to be performed. For example, a task might execute a Python function, run a bash command, or send an email.

Each time a DAG runs, each task within it creates a task instance. A task instance represents the execution of that task at a specific point in time and includes its state, such as whether it succeeded, failed, or was skipped.

3. Operators

Operators define the specific action that each task performs. Airflow comes with a rich set of built-in operators that you can use to execute a variety of tasks. Some of the common operators include:

- PythonOperator: Executes Python functions.
- BashOperator: Runs bash commands or scripts.
- EmailOperator: Sends emails.
- DummyOperator: A no-op operator that is useful for defining task dependencies.
- MySqlOperator: Executes SQL commands on a MySQL database.
- PostgresOperator: Executes SQL commands on a PostgreSQL database.

In addition to these built-in operators, you can create custom operators to fit specific use cases, making Airflow highly extensible.

4. Scheduler

The scheduler is a crucial component of Airflow, responsible for determining when each task should be executed. Based on the schedule you define for your DAG (e.g., run every day at midnight), the scheduler triggers the tasks at the right time.

The scheduler constantly monitors your DAGs and ensures that tasks are executed in the correct order, respecting task dependencies and handling retries in case of failures.

5. Executor

The executor is the component that actually runs your tasks. The executor in Apache Airflow doesn't directly execute tasks but instead defines how tasks will be run and manages task execution. Its primary role is to determine where and how the tasks are dispatched for execution, based on the type of executor being used. Airflow offers several types of executors depending on your use case:

LocalExecutor: Runs tasks on the same machine where the Airflow scheduler is running. This is ideal for smaller setups or development environments.

CeleryExecutor: Distributes tasks across a Celery cluster, allowing for scalable task execution across multiple workers.

KubernetesExecutor: Runs each task in its own Kubernetes pod, providing isolation and scalability in containerized environments.

Each executor has its own strengths and weaknesses, so choosing the right one depends on the scale and complexity of your workflows.

6. Web Server

The web server is a crucial component that provides Airflow's user interface. Through this web-based interface, you can easily monitor your workflows, view logs, manage DAGs, and perform task-level actions such as retries, manual triggers, and more.

This web UI provides real-time insights into DAG execution and serves as a control center for orchestrating and managing workflows.

7. Queue

Airflow's task execution often involves a queue mechanism, particularly when using distributed executors like CeleryExecutor or KubernetesExecutor. When tasks are scheduled, they are placed into a queue, from which workers (which we'll cover below) pick up tasks to execute.

This queue allows for efficient task distribution, ensuring that tasks are picked up as resources become available, and enables parallel execution across multiple machines.

8. Worker

A worker is the component responsible for actually executing tasks. In a distributed setup (e.g., CeleryExecutor), workers run on separate machines and are responsible for pulling tasks from the queue and executing them. Workers ensure that the system can scale by distributing tasks across multiple nodes.

In smaller setups with a LocalExecutor, the worker might be running on the same machine as the scheduler, but in larger production environments, the workers are typically distributed to increase performance and scalability.

9. Triggerer

Introduced in later versions of Airflow, the triggerer is designed to improve the efficiency of long-running tasks, particularly those that rely on sensors (e.g., tasks waiting for external events like the presence of a file). Instead of using traditional task polling, the triggerer handles tasks asynchronously, improving resource usage and scalability.

This component allows Airflow to scale more effectively when workflows have tasks that involve waiting for external events.

10. Metadata Database

Airflow relies on a metadata database to store information about your DAGs, tasks, task instances, and more. This database keeps track of the state of your workflows and stores historical execution data, allowing Airflow to manage retries, dependencies, and other workflow states.

The metadata database can be configured to use popular relational databases like PostgreSQL or MySQL, ensuring that Airflow has a robust and reliable storage layer for tracking and monitoring workflow execution.

11. Logs

Airflow automatically generates logs for every task execution. These logs are incredibly useful for debugging and monitoring the health of your tasks. You can view the logs through the web interface or directly on the server where Airflow is running. Each task instance logs detailed information about its execution, including input parameters, output, and any errors encountered.

Logs can be stored locally or sent to external systems like AWS S3, Google Cloud Storage, or Elasticsearch for centralized monitoring and retention.

12. Plugins

Airflow allows you to extend its functionality using plugins. Plugins are a powerful way to customize Airflow's behavior by adding new operators, sensors, hooks, and even modifying the web interface. This extensibility ensures that Airflow can adapt to specific business needs and integrate seamlessly with other systems in your data architecture.

Conclusion

Understanding the core components of Apache Airflow is essential for building scalable, reliable workflows. From defining DAGs and tasks to scheduling and executing them, Airflow's architecture is designed to handle everything from simple automation to complex data pipelines. We will talk more about each of these components in the coming blogs. Until then, stay healthy and keep learning!

Operators in Apache Airflow

We'll dive deep into one of the core concepts of Airflow: Operators. Operators are the building blocks of Airflow DAGs (Directed Acyclic Graphs), and they define what tasks should be executed in your workflows. I hope you guys now have a basic understanding of Airflow. Now we are diving deep into the core components of Airflow and it might get a bit overwhelming from hereon. But I have got you covered, just give this a good read. I have tried to make your life easy by keeping this extremely beginner friendly, so let's get started.

What are Operators?

In Apache Airflow, an Operator is a predefined template that encapsulates the logic of a specific task in a workflow. Each task in a DAG is associated with an operator, which defines what the task will do. Operators don't actually execute the code but describe how a task should be executed.

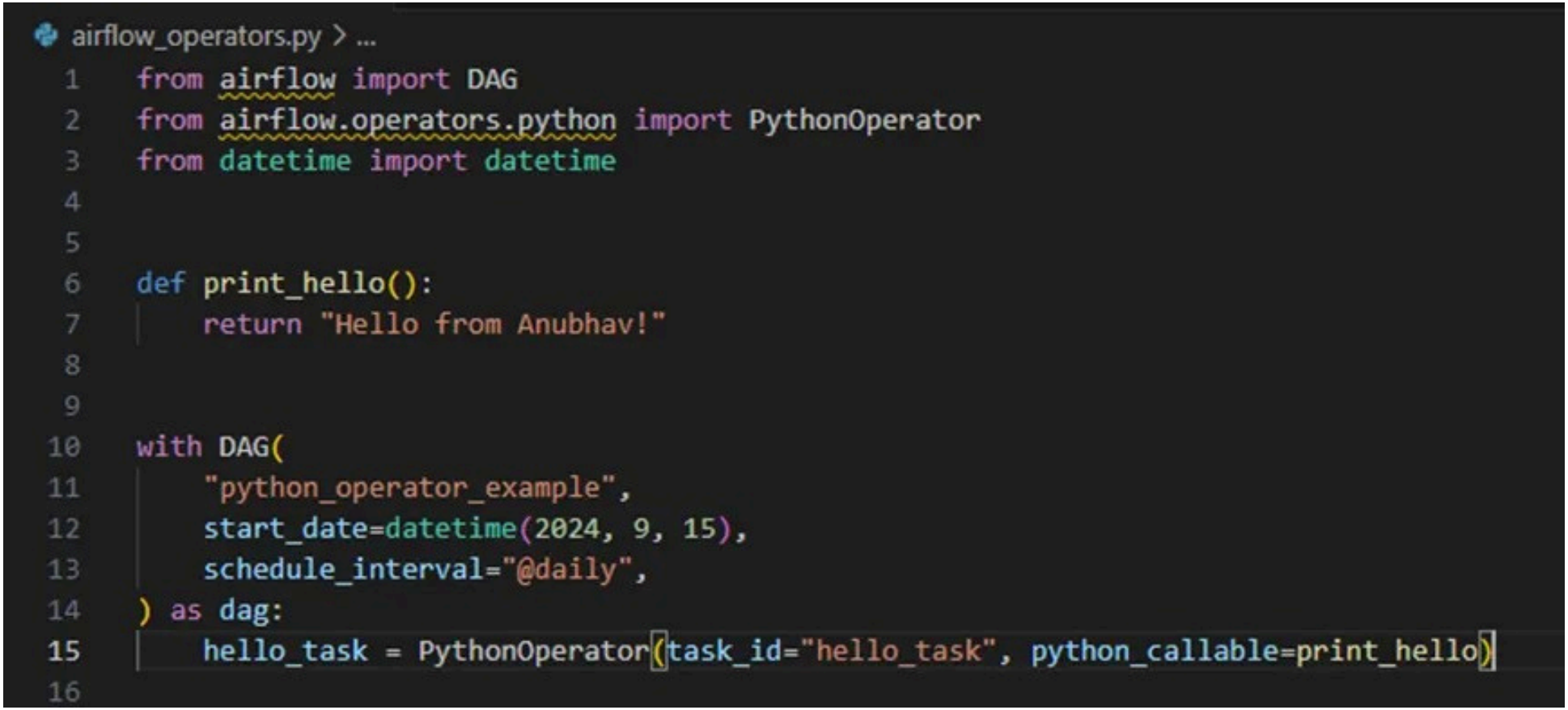
Broadly speaking, Operators can be divided into the following categories:

- Action Operators: Perform an action, such as executing a Python function, running a shell command, or making an HTTP request.
- Transfer Operators: Move data from one system to another, like transferring files between databases or from an FTP server to cloud storage.
- Sensor Operators: Wait for a certain condition to be met before proceeding, like waiting for a file to land in a directory or for an API to respond.

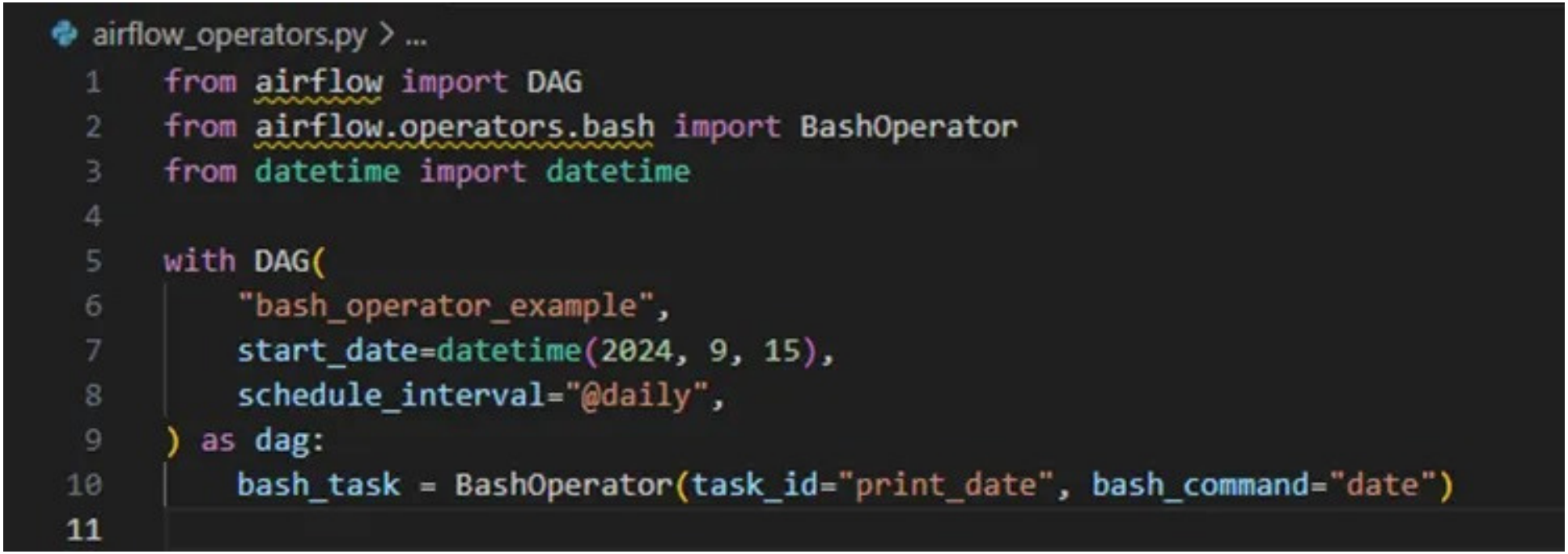
Action Operators

Let’s begin by exploring Action Operators in detail, as these are the ones used to execute tasks such as running a function, triggering a command, or sending a message.

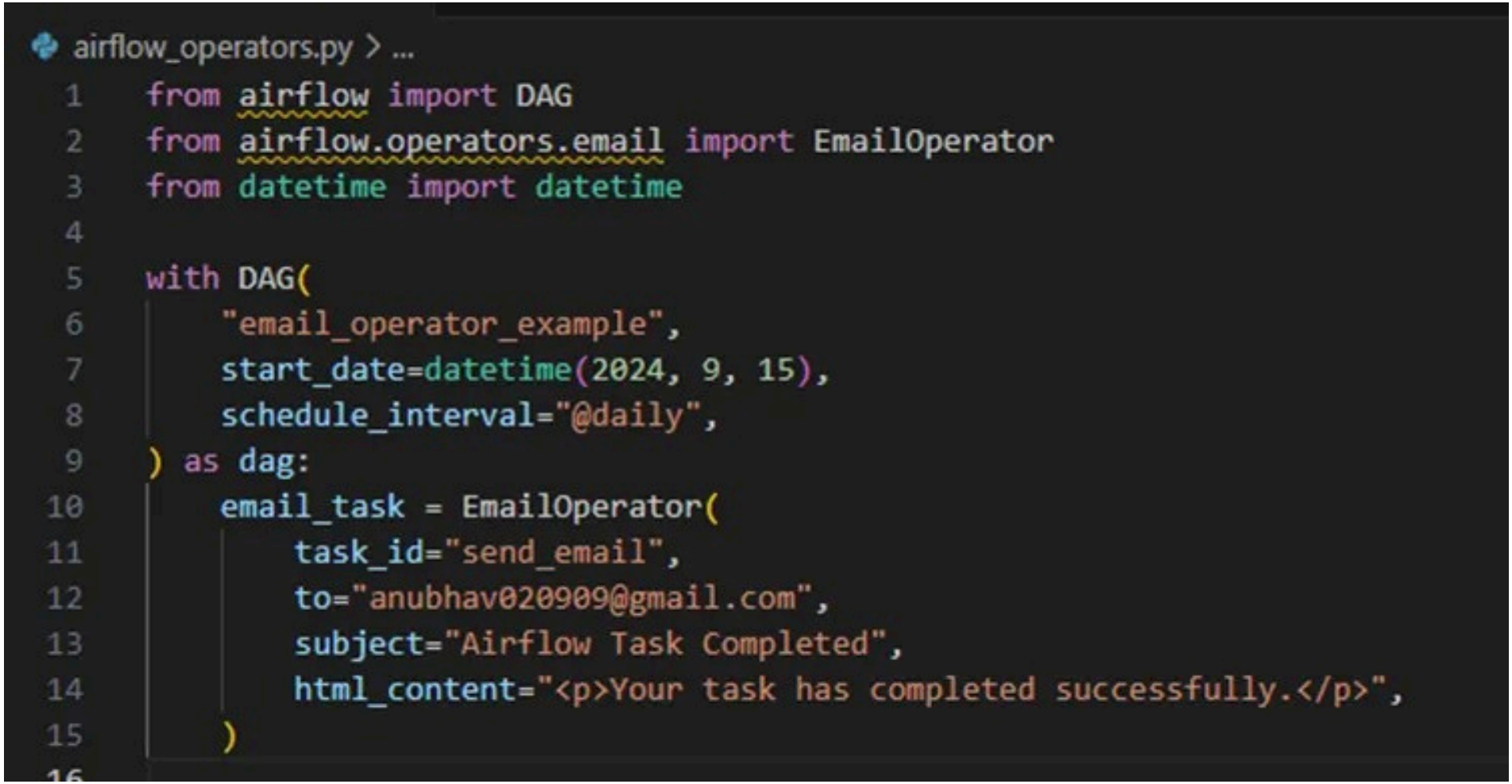
1. **PythonOperator** — The PythonOperator is one of the most popular and versatile operators in Airflow. It allows you to execute a Python function in your DAG. It's useful for any kind of custom logic, such as data transformations, API calls, or calculations.



2. **BashOperator** — The BashOperator allows you to execute a bash shell command. It is useful for running shell scripts, launching programs, or interacting with the underlying system.



3. **EmailOperator** — The EmailOperator is used to send an email as part of your DAG's workflow. It's especially useful for sending notifications, alerts, or reports.



4. **HttpOperator** — The HttpOperator is used to send HTTP requests. It's often used to interact with REST APIs or trigger external systems from within an Airflow DAG.

airflow_operators.py > ...

```python
from airflow import DAG
from airflow.providers.http.operators.http import SimpleHttpOperator
from datetime import datetime

with DAG(
    "http_operator_example",
    start_date=datetime(2024, 9, 15),
    schedule_interval="@daily",
) as dag:
    http_task = SimpleHttpOperator(
        task_id="get_weather_data",
        method="GET",
        http_conn_id="weather_api",
        endpoint="/data/2.5/weather?q=Seattle&APPID={{ var.value.api_key }}",
    )

```

Transfer Operators

Transfer Operators are used to move data from one system to another. Airflow provides built-in operators to facilitate data movement between databases, cloud storage, or FTP servers.

1. **S3ToRedshiftOperator** — Transfers data from an Amazon S3 bucket to an Amazon Redshift database.
2. **MySqlToS3Operator** — Transfers data from a MySQL database to an Amazon S3 bucket.

Sensor Operators

Sensor Operators are designed to wait for a certain condition to be met. They are used to pause a task until some external condition is satisfied, like waiting for a file to be uploaded or an API to return a valid response.

1. **FileSensor** — Waits for a file to appear in a specified directory.

airflow_operators.py > ...

```python
from airflow import DAG
from airflow.sensors.filesystem import FileSensor
from datetime import datetime

with DAG(
    "file_sensor_example", start_date=datetime(2024, 9, 15), schedule_interval="@daily"
) as dag:
    file_task = FileSensor(
        task_id="wait_for_file",
        fs_conn_id="file_system",
        filepath="/path/to/file.csv",
        poke_interval=60,
    )

```

2. **HttpSensor** — Waits for a successful response from an HTTP endpoint.

Custom Operators

While Airflow provides a rich set of built-in operators, you can also create custom operators to handle more specific tasks. Custom operators are useful when you need to integrate with a non-standard system or service. You can create custom operators by inheriting from Airflow's BaseOperator class.

Conclusion

Operators are essential to defining tasks in Apache Airflow. Whether it’s running Python code, interacting with external systems via APIs, moving data between databases, or waiting for conditions to be met, operators provide the flexibility to orchestrate a wide range of workflows. Now you might be getting a clearer understanding of Airflow, but there is a lot explore still. Don’t worry, we will get there. I’ll see you in the next blog. Until then, stay healthy and keep learning!

How does Apache Airflow work?

We’ll go deeper into the key components of Apache Airflow, including the Scheduler, Executor, Web Server, Worker, Meta Database, and DAG Directory. These components are essential to understanding how Airflow orchestrates workflows efficiently.

1. Scheduler: The Workflow Orchestrator

> The Scheduler is the heart of Airflow’s architecture, responsible for orchestrating tasks and workflows. Its main job is to continuously monitor your DAGs (Directed Acyclic Graphs), check task dependencies, and queue tasks for execution when they are due.

Key Functions of the Scheduler:

- Dependency Management: Ensures that tasks run in the correct order by checking dependencies defined in the DAG.
- Task Queuing: Schedules tasks based on their timing or external triggers, placing them in a queue for the Executor to handle.
- Monitoring DAGs: Watches the DAG directory for any updates or new DAGs, ensuring the workflow definitions are always up-to-date.

The Scheduler does not execute the tasks but is crucial in determining which tasks are ready to run.

2. Executor: The Task Distributor

The Executor is responsible for determining how tasks will be executed. It interacts with the Workers by distributing the tasks across them.

Different Executor types include:

- **LocalExecutor**: Executes tasks locally on the same machine running Airflow, useful for smaller setups.
- **CeleryExecutor**: Distributes tasks to Workers across different machines, making it ideal for scaling in production environments.
- **KubernetesExecutor**: Spins up a separate pod for each task in a Kubernetes cluster, offering high scalability and isolation.

The Executor doesn't execute tasks but defines the strategy for distributing them to Workers.

3. Web Server: The User Interface

The Web Server is the user-facing component of Airflow, providing an interface to interact with your workflows. This UI is where users can monitor DAGs, trigger manual task executions, view task logs, and track overall workflow health.

Key features of the Web Server:

- DAG Visualization: Offers a visual representation of your workflows, helping you understand task dependencies and statuses.
- Task Monitoring: Allows you to check the status of each task (success, failure, running, etc.) and view logs for troubleshooting.
- Manual Triggering: You can manually trigger DAGs or individual tasks through the interface, which is useful for debugging or running ad-hoc jobs.

The Web Server serves as a crucial tool for monitoring and managing workflows in real time.

4. Worker: The Task Executor

Workers are processes that execute the tasks assigned by the Executor. Each Worker runs on a separate machine or process and can execute multiple tasks in parallel.

Key responsibilities of Workers:

- Task Execution: Runs the Python code or other scripts defined in the DAG s .
- Task Reporting: After executing a task, Workers send the status back to the Scheduler and store logs for debugging.
- Parallel Processing: Workers can process multiple tasks simultaneously depending on how they are configured, allowing workflows to scale efficiently.

Workers are responsible for doing the actual work in Airflow by running the tasks and reporting their results.

5. Meta Database: The Data Storage Engine

The Meta Database (often PostgreSQL or MySQL) stores all metadata related to the workflows. This includes DAG configurations, task statuses, scheduling information, and logs.

Key roles of the Meta Database:

- Task Status Tracking: Stores the current state of each task (queued, running, success, or failure) and updates it as the task progresses.
- DAG Storage: Maintains the configuration and versioning of all DAGs in the system.
- Logs and Audits: Stores task logs and historical records for auditing and troubleshooting purposes.

The Meta Database is critical to keeping Airflow in sync with task statuses and DAG configurations.

6. DAG Directory: Where Workflows Live

The DAG Directory is a folder where all your DAG files (Python scripts defining workflows) reside. Airflow scans this directory at regular intervals to identify any new or updated DAGs.

Key points about the DAG Directory:

- Workflow Definitions: All DAGs are defined here as Python files, specifying tasks, schedules, and dependencies.
- DAG Parsing: The Scheduler regularly scans this directory to update its view of all workflows, ensuring that any changes are immediately reflected.
- Dynamic Workflows: You can programmatically generate or update DAGs based on external data, making the DAG Directory a flexible place to define complex workflows.

The DAG Directory acts as the storage point for your workflow definitions, enabling Airflow to manage and execute them.

How These Components Work Together

1. DAGs are defined and stored in the DAG Directory.
2. The Scheduler scans the DAGs and places tasks in the execution queue.
3. The Executor determines how tasks should be distributed across Workers.
4. Workers execute the tasks and send status updates to the Meta Database.
5. The Web Server allows users to monitor workflows, check task statuses, and trigger actions.
6. Logs and historical data are stored in the Meta Database for future reference.

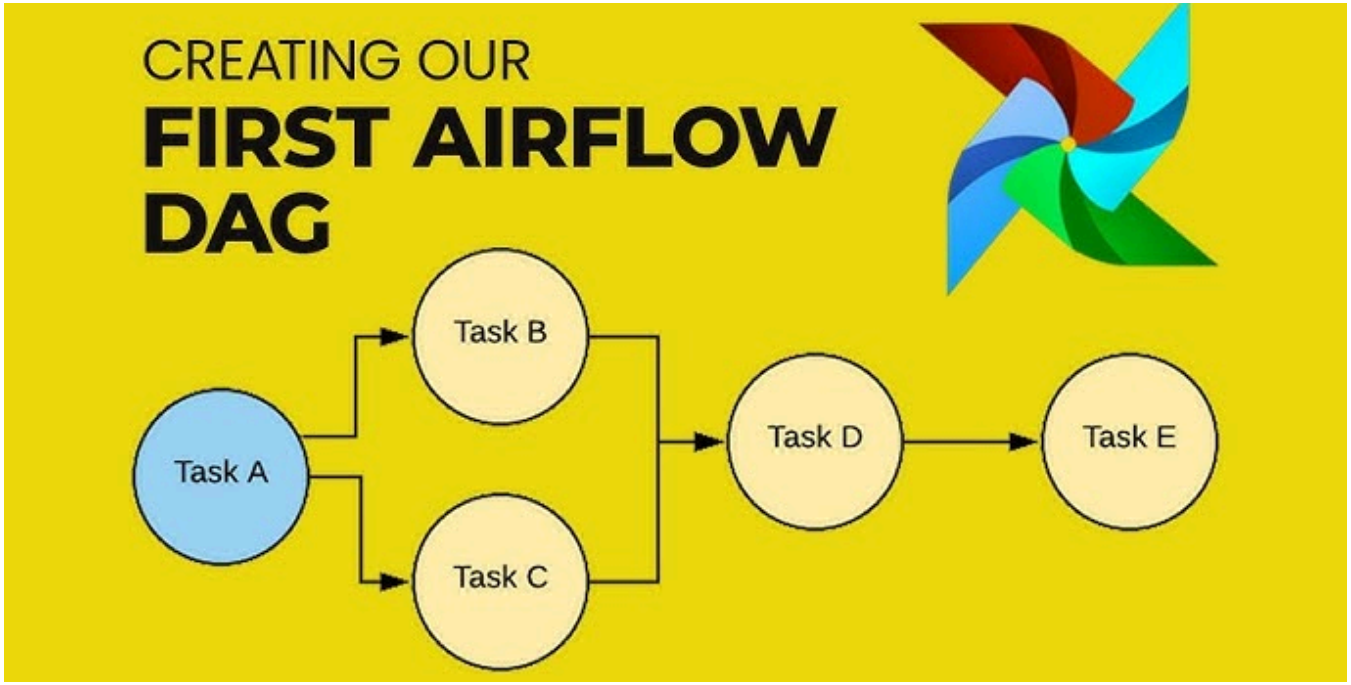
The combination of the Scheduler, Executor, Web Server, Worker, Meta Database, and DAG Directory provides a robust framework for orchestrating and monitoring workflows in Apache Airflow. Understanding how these components interact is essential for any data engineer looking to leverage Airflow for managing complex workflows in a scalable way. I hope you now have a good understanding of how Airflow works, and you can easily explain this in interviews. I will see you in the next blog. Until then, stay healthy and keep learning!

How to Build and Schedule Your First DAG

We’ll be diving into the practical aspects of Apache Airflow by building and scheduling your first Directed Acyclic Graph (DAG). A DAG is a collection of tasks organized in a way that reflects their dependencies and relationships. It’s the core concept of Airflow, representing your workflow as a series of tasks.

Why Building a DAG is Important

Creating a DAG is the first step in orchestrating and automating complex workflows. By defining tasks and their dependencies, you ensure your data pipelines run in the correct order and are easy to manage, monitor, and troubleshoot.



Step 1: Prerequisites

Before we start building our first DAG, make sure you have the following set up:

1. Apache Airflow Installed: You should have Airflow installed and configured. If not, check out the official installation guide.

2. Basic Python Knowledge: A basic understanding of Python is needed.
3. Text Editor: Use your preferred text editor or an IDE like VS Code, PyCharm, etc

Step 2: Setting Up the DAG File

DAGs are defined in Python files and stored in the dags folder of your Airflow home directory. Let’s create a simple DAG file named first_dag.py:

airflow_operators.py > ...

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime, timedelta

# Define default arguments
default_args = {
    "owner": "airflow",
    "depends_on_past": False,
    "start_date": datetime(2023, 9, 19),
    "email": ["anubhav020909@gmail.com"],
    "email_on_failure": False,
    "email_on_retry": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

# Initialize the DAG
dag = DAG(
    "first_dag",
    default_args=default_args,
    description="Anubhav DAG",
    schedule_interval=timedelta(days=1),
)


# Define a simple Python function
def hello_world():
    print("Hello, world!")


# Create a PythonOperator to execute the function
task = PythonOperator(
    task_id="print_hello_world",
    python_callable=hello_world,
    dag=dag,
)

```

Step 3: Understanding the DAG Structure

1. Importing Libraries:

- We import DAG from Airflow and PythonOperator to create tasks.
- We also import datetime and timedelta to handle time-based scheduling.

2. Defining Default Arguments:

- default_args contains default settings for your tasks, such as start date, owner, retry policy, etc.

3. Initializing the DAG:

- We create an instance of the DAG class, setting its name (first_dag), description, and schedule interval (timedelta(days=1), which means it will run daily).

4. Creating a Task:

- We define a simple Python function hello_world() that prints a message.

- We use PythonOperator to create a task called print_hello_world that runs the hello_world function.

Step 4: Scheduling the DAG

The schedule_interval parameter defines how often your DAG should run. Some common options are:

- @daily: Runs the DAG once a day.
- @hourly: Runs the DAG every hour.
- @weekly: Runs the DAG once a week.
- Cron expressions like 0 6 * * * for more complex schedules.

Step 5: Placing the DAG File

Place your first_dag.py file in the dags folder of your Airflow home directory. By default, it’s located at ~/airflow/dags. Airflow will automatically detect and register your DAG.

Step 6: Testing and Monitoring the DAG

1. Activate the DAG:

    - Go to the Airflow UI (usually accessible at http://localhost:8080) and find your DAG under the DAGs tab.
    - Toggle the switch to activate the DAG.

2. Trigger the DAG:

    - Click on the DAG name and then click on the “Trigger DAG” button. This will manually trigger the DAG.

3. Monitoring:

    - You can monitor the status of the DAG run and individual tasks in the Airflow UI.
    - Check the logs for detailed information on each task’s execution.

Step 7: Troubleshooting

If you encounter any issues, check the following:

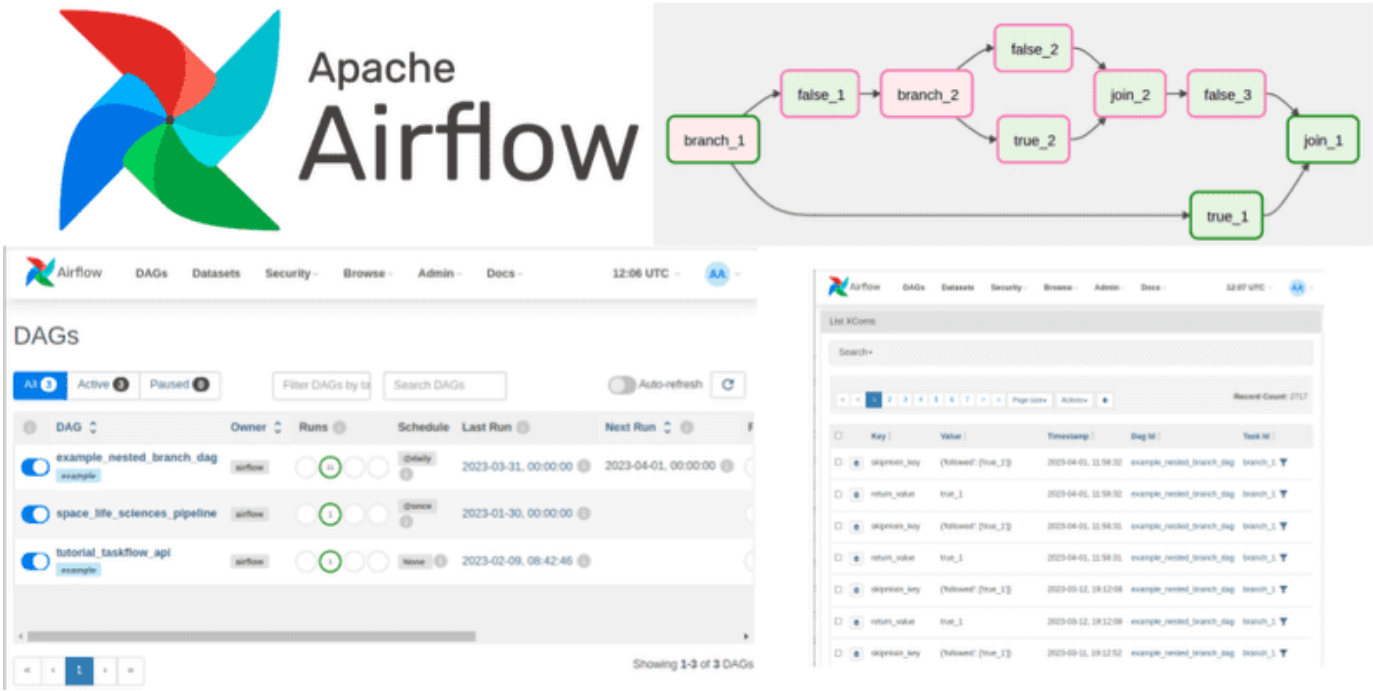
- **Logs**: Logs are your best friend for debugging issues. Check the logs in the UI under the "Task Instances" tab.
- **Code Errors:** Ensure your code is error-free and follows proper syntax.
- **Dependencies**: Ensure all dependencies are installed and correctly configured.

Congratulations! You've just built and scheduled your first DAG in Apache Airflow. This foundational knowledge will help you understand how workflows are managed in Airflow. As you progress, you can explore more complex DAGs, task dependencies, and advanced scheduling. See you in the next blog. Until then, stay healthy and keep learning!

Advanced DAG Concepts in Apache Airflow

We've covered the basics of building and scheduling a DAG. Today, we'll delve into some advanced DAG concepts that will help you create more dynamic, flexible, and powerful workflows. Understanding these concepts will allow you to manage complex data pipelines and handle real-world scenarios more efficiently.

Why Advanced DAG Concepts Matter



As your workflows grow in complexity, you'll need more sophisticated tools to handle various use cases. Advanced DAG features like branching, dynamic task generation, and error handling are crucial for creating robust, production-ready pipelines. These features will allow you to:

- **Optimize Task Execution:** Control how and when tasks are executed based on certain conditions.
- **Enhance Flexibility:** Dynamically generate tasks and adapt to changing inputs or configurations.
- **Improve Error Handling:** Gracefully handle errors and ensure data consistency and reliability.

1. Task Dependencies and Branching

In simple DAGs, tasks are executed in a linear sequence, but real-world workflows often require conditional branching. Airflow provides several ways to manage task dependencies beyond simple linear execution:

Using the BranchPythonOperator
The BranchPythonOperator allows you to conditionally execute certain tasks based on the output of a Python function. It helps you control the flow of your DAG.

```python
from airflow import DAG
from airflow.operators.python import BranchPythonOperator
from airflow.operators.dummy import DummyOperator
from datetime import datetime


def choose_branch(**kwargs):
    # You can add your logic here, for now, we are just selecting branch_a
    return "branch_a"


default_args = {"owner": "airflow", "start_date": datetime(2023, 9, 15)}

with DAG(
    "branching_example", default_args=default_args, schedule_interval="@daily"
) as dag:

    start = DummyOperator(task_id="start")

    branching = BranchPythonOperator(task_id="branching", python_callable=choose_branch)

    branch_a = DummyOperator(task_id="branch_a")
    branch_b = DummyOperator(task_id="branch_b")

    join = DummyOperator(task_id="join", trigger_rule="none_failed_or_skipped")

    start >> branching >> [branch_a, branch_b] >> join
```

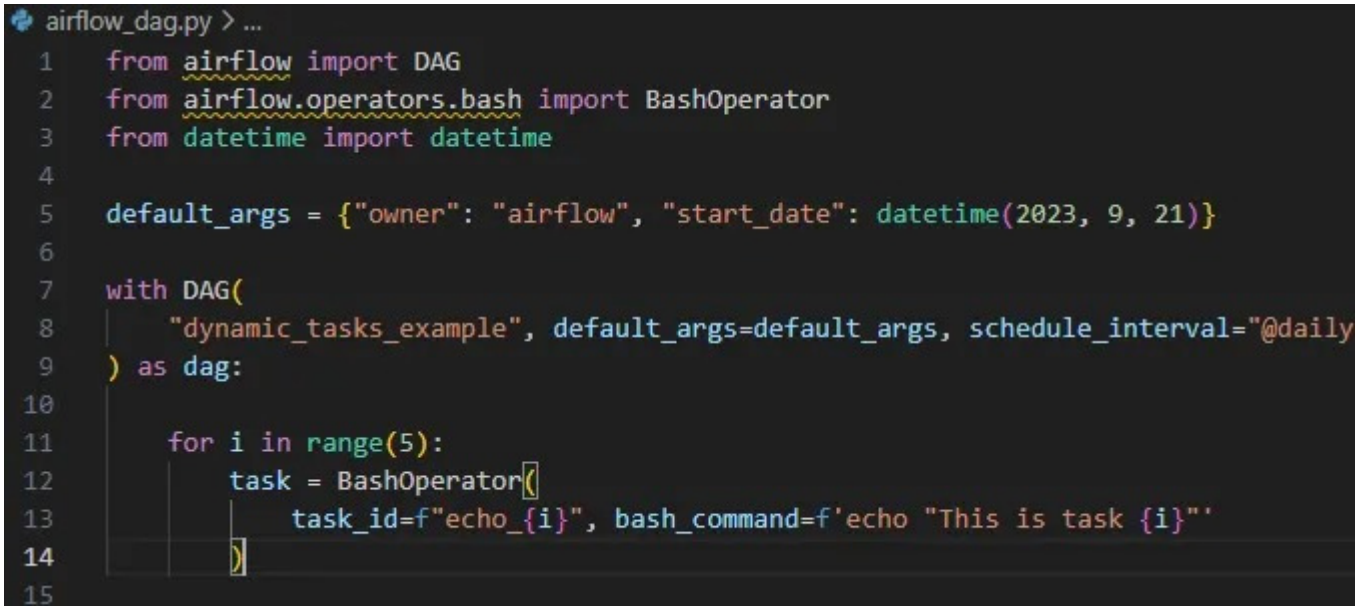
In this example:

- The **BranchPythonOperator** decides whether to follow branch_a or branch_b.
- The **join** task waits for either branch_a or branch_b to complete.

2. Dynamic Task Generation

Dynamic task generation allows you to create multiple tasks programmatically within a DAG. This is particularly useful when you have a variable number of tasks to execute based on some external input or configuration.

Using a for loop to generate tasks



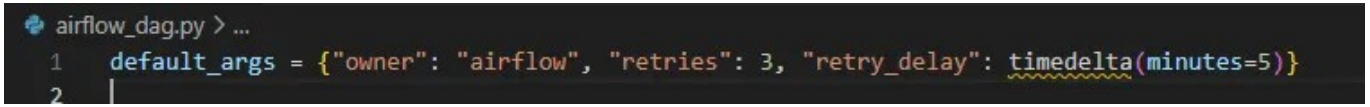
In this example:

- Five tasks (`echo_0` to `echo_4`) are dynamically created in the DAG.
- This is useful when you don’t know the exact number of tasks in advance or they depend on some external data.

3. Error Handling and Task Retries

In production workflows, failures are inevitable. Handling these failures gracefully is crucial to maintaining a reliable data pipeline.

Setting Retry Policies
You can configure how many times a task should be retried and the delay between retries using the retries and retry_delay parameters in the default_args .

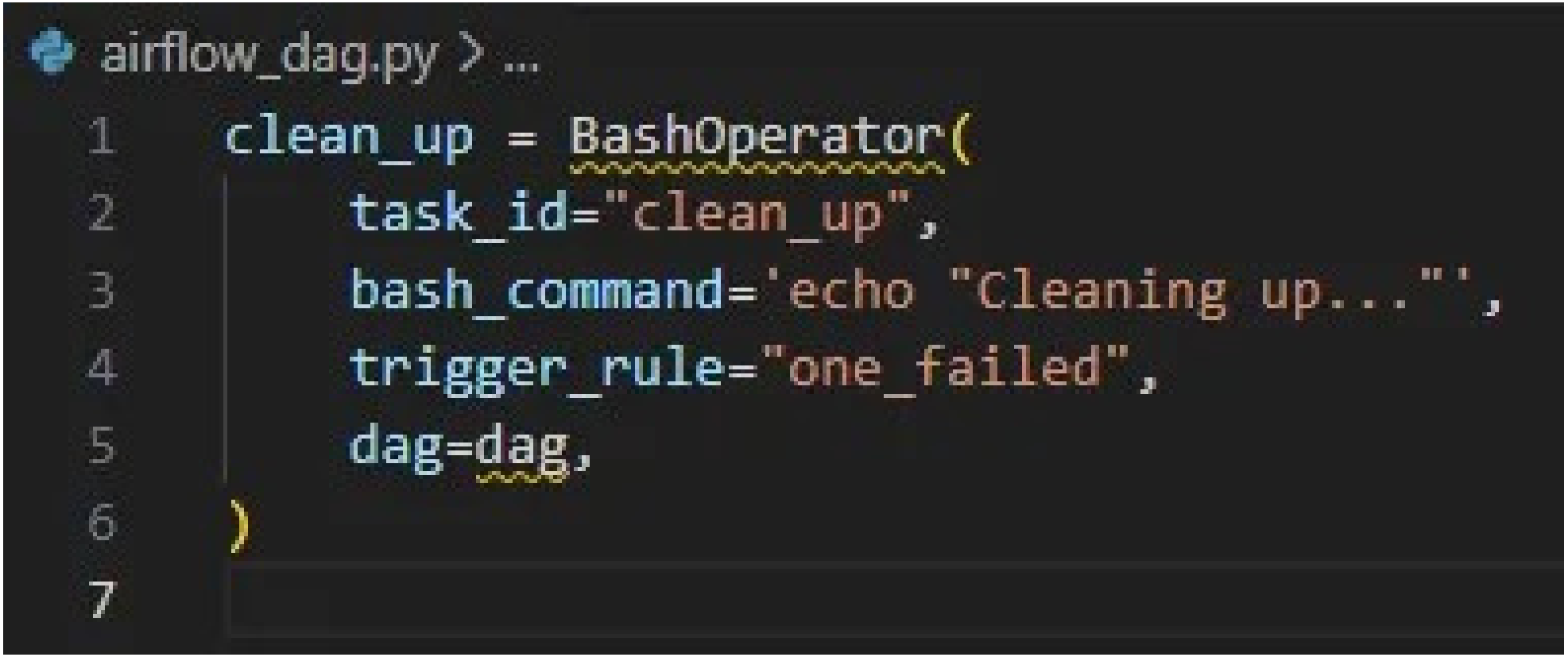


Using the **`trigger_rule`** Parameter

The trigger_rule parameter controls how a task is triggered based on the outcome of its upstream tasks. Common values include:

all_success: The task is triggered only if all upstream tasks succeeded(default behavior).
all_failed: The task is triggered if all upstream tasks failed.
one_failed: The task is triggered if any upstream task failed.
none_failed_or_skipped: The task is triggered if no upstream tasks failed or were skipped.

This is useful for creating fallback or cleanup tasks that should run even if certain tasks fail.



In this example, the **clean_up** task will be triggered if any upstream task fails.

4. Using the XCom Feature for Task Communication

In some cases, you might want tasks to share information. Airflow provides a feature called XCom (Cross-Communication) for passing small amounts of data between tasks.

Pushing and Pulling XComs

- **Push Data:** Use the xcom_push method to send data from one task.
- **Pull Data:** Use the xcom_pull method to retrieve data in another task.

airflow_dag.py > ...

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime


def push_data(**kwargs):
    kwargs["ti"].xcom_push(key="sample_data", value=42)


def pull_data(**kwargs):
    value = kwargs["ti"].xcom_pull(key="sample_data", task_ids="push_task")
    print(f"The value is: {value}")


with DAG("xcom_example", default_args=default_args, schedule_interval="@daily") as dag:

    push_task = PythonOperator(
        task_id="push_task", python_callable=push_data, provide_context=True
    )

    pull_task = PythonOperator(
        task_id="pull_task", python_callable=pull_data, provide_context=True
    )

    push_task >> pull_task

```

In this example:

push_data pushes a value (42) to XCom.

pull_data pulls the value from XCom and prints it.

5. Advanced Scheduling with Sensors

Sensors are a type of operator that wait for a certain condition to be met before executing downstream tasks. This is useful for integrating external systems or waiting for data availability.

Example: FileSensor

airflow_dag.py > ...

```python
from airflow.operators.sensors import FileSensor

file_sensor_task = FileSensor(
    task_id="file_sensor_task",
    filepath="/path/to/file",
    poke_interval=10,
    timeout=60 * 5,
    dag=dag,
)
```

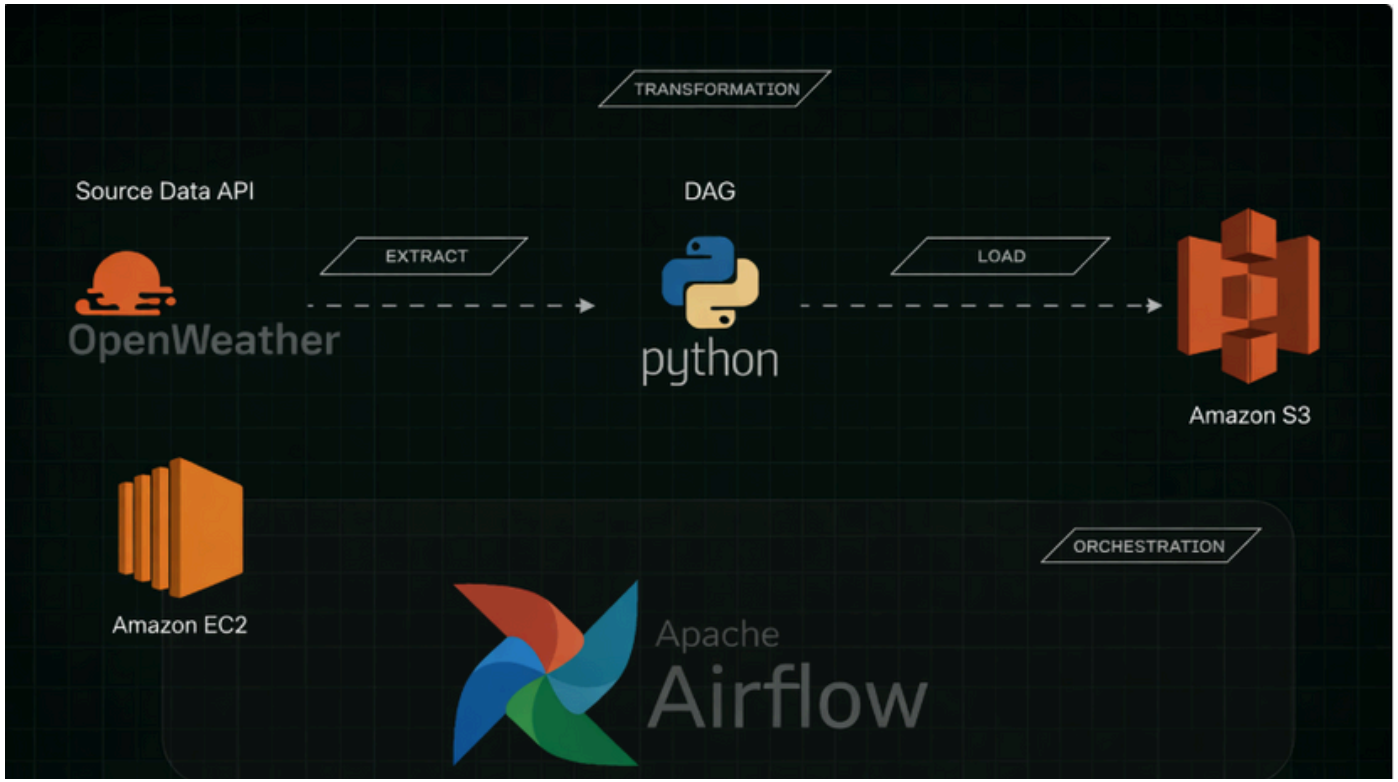
In this example:

- The FileSensor waits for a file to be available at /path/to/file.
- It checks every 10 seconds (poke_interval) and times out after 5 minutes (timeout).

This is all we have for today. I know I discussed a few advanced topics here, but once you start building your own DAGs, you will get comfortable with all of these concepts. I will see you in the next blog, until then, stay healthy and keep learning!

End-to-end Data Engineering Project

We’ll walk through building an end-to-end data pipeline using Apache Airflow on an AWS EC2 instance. This project automates the process of fetching weather data for the city of Seattle from the OpenWeather API and stores the data in an Amazon S3 bucket. If you’re looking to learn about integrating external APIs, managing cloud infrastructure, and building data pipelines, this project is a fantastic hands- on exercise.



Workflow for our Project

🔍 Project Overview:

We'll be setting up Apache Airflow on an Ubuntu instance hosted on AWS EC2, creating a custom DAG (Directed Acyclic Graph) to orchestrate tasks that:

1. Fetch weather data for Seattle from the OpenWeather API.
2. Store the data in an S3 bucket for further analysis and use.

📅 Steps Breakdown:

1. Setting up AWS EC2 Instance

- Launch an Ubuntu EC2 instance and configure security groups to allow access to port 8080 (for Airflow) and port 22 (for SSH).
- SSH into your instance and install Docker and Docker Compose to set up the Apache Airflow environment.

2. Setting up Airflow on EC2

Use Docker Compose to configure Apache Airflow on the instance. Here's a sample of the docker-compose.yml file:

docker-compose.yml

```yaml
version: '3'
services:
  airflow:
    image: apache/airflow:latest
    environment:
      - LOAD_EX=y
    ports:
      - "8080:8080"
    volumes:
      - ./dags:/opt/airflow/dags

```

- Run Airflow with **docker-compose up**, and once it’s running, access the Airflow UI through the browser.

3. Creating the Custom DAG

The DAG will consist of the following steps:

- Check API Availability: Ensure the OpenWeather API is up and running.
- Fetch Weather Data: Retrieve weather data from the API.
- Store Data in S3: Save the fetched data in an S3 bucket for further analysis.

Here’s the DAG code I have used for the project:

```
from datetime import timedelta, datetime
import json
from airflow import DAG
from airflow.providers.http.sensors.http import HttpSensor
from airflow.providers.http.operators.http import SimpleHttpOperator
from airflow.operators.python import PythonOperator
import pandas as pd
from airflow.hooks.base import BaseHook


def get_api_key():
        conn = BaseHook.get_connection("weather_api")
        extras = conn.extra_dejson
        return extras["api_key"]


api_key = get_api_key()


def kelvin_to_fahrenheit(temp_in_kelvin):
        temp_in_fahrenheit = (temp_in_kelvin - 273.15) * (9 / 5) + 32
        return temp_in_fahrenheit


def get_aws_credentials():
        conn = BaseHook.get_connection("aws_credentials")
        aws_credentials = {
                "key": conn.login,
                "secret": conn.password,
                "token": conn.extra_dejson.get("aws_session_token"),
        }
        return aws_credentials
```

```
def transform_load_data(task_instance):
    data = task_instance.xcom_pull(task_ids="extract_data_from_api") city = data["name"]
    weather_description = data["weather"][0]["description"]
    temp_fahrenheit = kelvin_to_fahrenheit(data["main"]["temp"])
    feels_like_fahrenheit = kelvin_to_fahrenheit(data["main"]["feels_like"])
    min_temp_fahrenheit = kelvin_to_fahrenheit(data["main"]["temp_min"])
    max_temp_fahrenheit = kelvin_to_fahrenheit(data["main"]["temp_max"])
    pressure = data["main"]["pressure"]
    humidity = data["main"]["humidity"] wind_speed = data["wind"]["speed"]
    time_of_record=datetime.utcfromtimestamp(data["dt"] + data["timezone"])
    sunrise_time=datetime.utcfromtimestamp(data["sys"]["sunrise"] + data["time
    sunset_time=datetime.utcfromtimestamp(data["sys"]["sunset"] + data["timezo

    transformed_data = {
        "City": city,
        "Description": weather_description,
        "Temperature (F)": temp_fahrenheit,
        "Feels Like (F)": feels_like_fahrenheit,
        "Minimum Temp (F)": min_temp_fahrenheit,
        "Maximum Temp (F)": max_temp_fahrenheit,
        "Pressure": pressure,
        "Humidity": humidity,
        "Wind Speed": wind_speed,
        "Time of Record": time_of_record,
        "Sunrise(LocalTime)": sunrise_time,
        "Sunset(LocalTime)": sunset_time,
    }

    transformed_data_list = [transformed_data]
    df_data = pd.DataFrame(transformed_data_list)

    # Retrieve AWS credentials
    aws_credentials = get_aws_credentials()
    now = datetime.now()
```

```
        dt_string = now.strftime("%d%m%Y%H%M%S")
        file_name = f"current_weather_data_seattle_{dt_string}.csv"

        #Savedatato S3
        df_data.to_csv(
            f"s3://weatherapibucket-anubhav/{file_name}",
            index=False,
            storage_options={
                "key": aws_credentials["key"],
                "secret": aws_credentials["secret"],
                "token": aws_credentials["token"],
            },
        )


default_args = {
        "owner": "airflow",
        "depends_on_past": False,
        "start_date": datetime(2024, 9, 15),
        "email": ["anubhav020909@gmail.com"],
        "email_on_failure": False,
        "email_on_retry": False,
        "retries": 3,
        "retry_delay": timedelta(minutes=5),
}

with DAG(
        "weather_dag", default_args=default_args, schedule_interval="@daily", catchu
) as dag:

        is_api_working = HttpSensor(
            task_id="is_api_working",
            http_conn_id="weather_api",
            endpoint=f"/data/2.5/weather?q=Seattle&APPID={api_key}",
        )
```

```
extract_data_from_api = SimpleHttpOperator(
        task_id="extract_data_from_api",
        http_conn_id="weather_api",
        endpoint=f"/data/2.5/weather?q=Seattle&APPID={api_key}",
        method="GET",
        response_filter=lambda r: json.loads(r.text),
        log_response=True,
)

transform_load_data = PythonOperator(
        task_id="transform_load_data", python_callable=transform_load_data
)

is_api_working >> extract_data_from_api >> transform_load_data
```

4. Configuring Connections in Airflow

- Define your API key in Airflow using Variables (api_key) to avoid hardcoding sensitive information.
- Set up the S3 connection in Airflow's Admin > Connections for seamless integration with AWS.

5. Running the DAG

- Once the DAG is ready, trigger it manually from the Airflow UI or let it run as per the daily schedule (@daily). The DAG will first check if the API is available and then fetch and store weather data in your designated S3 bucket.

6. Monitoring and Logging

- You can monitor the DAG's execution in the Airflow UI to ensure that all tasks are running as expected.
- Logs from each task will be available for debugging and review.

💡 Why This Project?

This project provides valuable insights into orchestrating workflows in real- world scenarios. It teaches you how to:

Set up Airflow on a cloud instance.

- Build a simple but effective data pipeline that pulls data from an API and stores it in a cloud storage service (Amazon S3).
- Understand the value of automating workflows in the context of data engineering.

Whether you're an aspiring data engineer or simply looking to add cloud experience to your portfolio, this project gives you hands-on experience with crucial tools like Airflow, APIs, and AWS. I hope you learnt a lot from this blog. I'll see you in the next blog. Until then, stay healthy and keep learning!

Airflow Hooks and Sensors — Monitoring and Interacting with External Systems

Apache Airflow is a powerful orchestration tool not only because it allows you to manage complex workflows, but also because it integrates easily with external systems through Hooks and Sensors. These features enable Airflow to interact with APIs, databases, file systems, and other external resources, making it a versatile solution for automating a variety of tasks in data engineering and beyond.

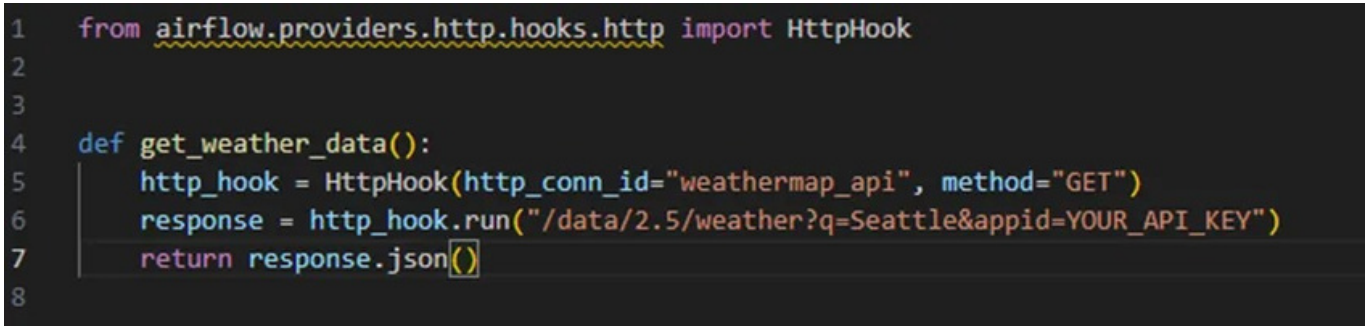
What are Hooks?

Hooks in Airflow are interfaces to external systems that provide methods for connecting, interacting, and executing commands. Whether you're working with a database, an API, or a cloud service, Hooks abstract the connection details and provide a simple interface for operations.

Some common hooks include:

- HttpHook: Connects to HTTP APIs.
- PostgresHook: Interacts with Postgres databases.
- S3Hook: Interacts with Amazon S3 buckets.
- MySqlHook: Connects to MySQL databases.
- GoogleCloudStorageHook: Connects to Google Cloud Storage.

Example of using a Hook in your DAG:



In this example, HttpHook is used to make an API call to the OpenWeather API and return the response data.

What are Sensors?

Sensors in Airflow are special types of tasks that “wait” for a condition to be met before allowing the workflow to continue. They continuously poll an external system until the desired condition is satisfied. They are typically used when waiting for external events such as file availability, API status, or database readiness.

Some common sensors include:

- HttpSensor: Waits for a specific HTTP response.
- FileSensor: Waits for a file to be present at a specified path.
- S3KeySensor: Waits for a file or key to be present in an S3 bucket.
- TimeSensor: Waits for a specified time to be reached.

Example of using a Sensor in your DAG:

```python
from airflow.providers.http.sensors.http import HttpSensor

is_api_working = HttpSensor(
    task_id="is_api_working",
    http_conn_id="weathermap_api",
    endpoint="/data/2.5/weather?q=Seattle&appid=YOUR_API_KEY",
    response_check=lambda response: "weather" in response.text,
    poke_interval=5,
    timeout=20,
)

```

In this case, **HttpSensor** checks if the weather API is working by validating the response contains weather data.

How Hooks and Sensors Fit into Your Workflow Hooks and Sensors allow Airflow to interact with external systems efficiently. Hooks provide a way to fetch or send data, while Sensors ensure that tasks don’t proceed until external conditions are met. These features are invaluable in data engineering workflows where you often need to:

- Wait for external files to arrive (e.g., from an FTP server or cloud storage).
- Query an API and wait for a valid response.
- Ensure that a database or resource is available before proceeding.

Best Practices with Hooks and Sensors

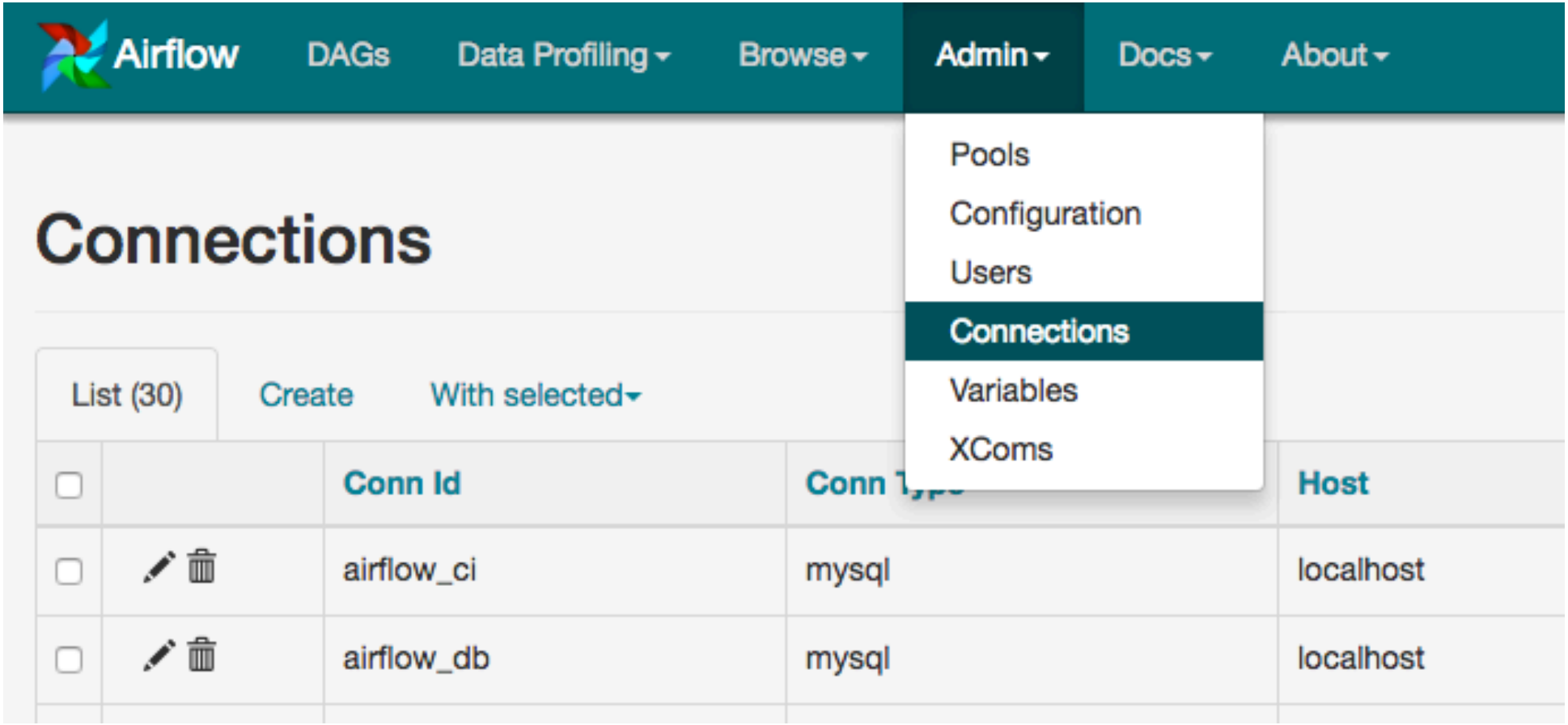
1. **Limit Sensor Polling:** Avoid frequent polling by using appropriate poke_interval and timeout values. Sensors can be resource-intensive, so it's important to set reasonable intervals.

2. **Use Soft Sensors for Long Waiting Times:** If a sensor needs to wait for a long time, consider using time-based sensors like TimeDeltaSensor or external sensors that are less resource-intensive.

3. **Leverage Hooks for External API and Database Integration:** Hooks make it easy to integrate APIs and databases into your workflows. Instead of hardcoding connection details, use Airflow's connections and hooks to manage them in a more maintainable way.

Airflow’s Hooks and Sensors provide a robust way to interact with external systems and monitor conditions. By using them effectively, you can build sophisticated workflows that involve a wide range of external services, databases, and APIs. These features ensure that your data pipelines can automate not only internal processes but also extend their reach into the larger ecosystem of external data sources. I hope you liked this blog, I will see you in the next one. Until then, stay healthy and keep learning!

Managing Airflow Connections and Variables for External Integrations



We will dive into one of the essential aspects of building data pipelines in Apache Airflow: Managing Connections and Variables. These features allow Airflow to communicate with external services, securely store credentials, and pass dynamic configurations across workflows.

What are Airflow Connections and Variables?

Connections: Airflow needs to interact with various systems such as databases, APIs, and cloud storage services. To securely manage credentials and connection information (e.g., host, port, username, password), Airflow uses Connections. These can be managed directly from the Airflow UI under the Admin tab.

Variables: These are dynamic values that you might want to reference throughout your DAGs. For instance, API keys, file paths, or configuration flags. Variables can be added through the UI, CLI, or environment files, making them highly flexible.

Why Use Connections and Variables?

- Separation of Code and Credentials: Hardcoding credentials such as API keys in your DAGs is a risky practice. By using Connections, you keep your secrets out of the codebase and manage them securely within Airflow's environment.
- Dynamic Configuration: With Variables, you can pass different configuration parameters into your DAGs based on environments (e.g., dev, prod), which improves flexibility and maintainability.

How to Create Connections and Variables in Airflow

1. Setting up a Connection

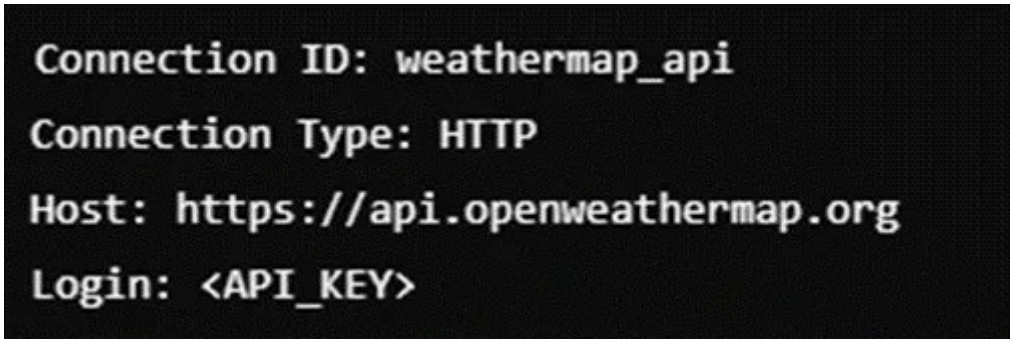
Here's how you can set up a connection for an external service, such as the OpenWeather API or AWS, which you may need in your data pipelines:

- Go to the Airflow UI.
- Navigate to the Admin > Connections tab.
- Click the + button to add a new connection.

You'll need to fill in details like:

- Connection ID: A unique name to reference this connection in your DAGs (e.g., weathermap_api or aws_s3).
- Connection Type: Choose from the dropdown (HTTP, Amazon S3, MySQL, Postgres, etc.).
- Host, Login, Password: These fields store your host URLs, access keys, secrets, etc. (e.g., API endpoint for weather data).

Here's an example for an HTTP connection to the OpenWeather API:



2. Creating Variables

To pass sensitive or dynamic data (e.g., API keys, parameters), create a variable:

- Navigate to Admin > Variables.
- Click the + button to add a variable.
- Define Key (e.g., api_key) and Value (e.g., the actual API key 123456).

Now, you can reference the variable in your DAGs using {{ var.value.api_key }}.

Using Connections and Variables in DAGs

Example: Fetching Weather Data with a Secure Connection

Let’s revisit our weather data project from Day 8 and show how to use Connections and Variables in the code:

```python
from airflow import DAG
from airflow.providers.http.sensors.http import HttpSensor
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta
import requests
import json

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 9, 15),
    'retries': 3,
    'retry_delay': timedelta(minutes=5)
}

def fetch_weather_data():
    api_key = "{{ var.value.api_key }}"
    response = requests.get(f"https://api.openweathermap.org/data/2.5/weather?q=Seattle&appid={api_key}")
    return response.json()

def store_in_s3(**context):
    data = fetch_weather_data()
    s3 = S3Hook(aws_conn_id='aws_default')
    s3_key = f'weather_data/Seattle_{context["execution_date"].strftime("%Y%m%d")}.json'
    s3.load_string(json.dumps(data), s3_key, bucket_name='your-s3-bucket', replace=True)

with DAG(dag_id='weather_data_pipeline_v2', default_args=default_args, schedule_interval='@daily') as dag:

    check_api = HttpSensor(
        task_id='check_weather_api',
        http_conn_id='weathermap_api',  # Connection to OpenWeather API
        endpoint='/data/2.5/weather?q=Seattle',
        poke_interval=5,
        timeout=20
    )

    store_weather_data = PythonOperator(
        task_id='store_weather_data',
        python_callable=store_in_s3
    )

    check_api >> store_weather_data
```

How it works:

- The **HttpSensor** task references the connection holds the **weathermap_api** , which API base URL and credentials.
- The **fetch_weather_data** function uses the Airflow variable **api_key** to dynamically pull in the API key without hardcoding it.
- S3Hook securely interacts with the S3 bucket using the **aws_default** connection.

Best Practices for Using Connections and Variables

- **Centralize Credentials:** Always use Connections and Variables for external credentials. Avoid hardcoding them into DAGs.
- **Environment-Specific Variables:** Utilize environment-based variables (dev, staging, prod) for different environments to maintain flexibility and security.
- **Audit Changes:** Keep track of who adds or modifies credentials in Connections for security purposes.

you should have a clear understanding of how to manage external connections and pass dynamic variables to your DAGs. These are crucial features when orchestrating complex workflows in production environments. With this knowledge, you can now securely manage integrations and avoid potential security risks while building scalable and maintainable Airflow pipelines. I'll see you in the blog. Until then, stay healthy and keep learning!

Troubleshooting and Debugging DAGs

We'll dive into an important topic for every Apache Airflow user: Troubleshooting and Debugging DAGs. No data pipeline is perfect from the start, and encountering errors is part of the process. Knowing how to identify and fix these issues is critical for keeping workflows running smoothly.

Why Troubleshooting is Essential

Airflow's flexibility and complexity can sometimes lead to errors in the DAGs. These issues could arise from:

- Task failures: A Python function or external system might fail.
- Missing dependencies: Files, credentials, or environment variables might not be correctly set up.
- Configuration errors: Improper configurations or misconfigurations of Airflow settings can cause failures.

🛠️ Common Errors and How to Fix Them

1. Task Failures

- Symptom: You notice that a task has failed in the UI with an error message.
- Fix: Look at the detailed logs in the UI. For example, if the error is related to an API call, check if the API response is valid or if the credentials are set up correctly.

2. Broken DAGs

- Symptom: Your DAG does not show up in the UI, or it shows up as broken.
- Fix: This typically happens due to syntax errors or missing imports in the DAG file. Check your DAG code for missing dependencies, indentation issues, or typos.

3. Scheduler Not Picking Up DAGs

- Symptom: The scheduler is not scheduling your DAG runs, or they are being skipped.
- Fix: Ensure that catchup=False is set if you don’t want to backfill old DAG runs.
- Also, check the DAG’s start_date and schedule_interval to ensure they are correctly set for your workflow.

4. Missing Connections or Variables

- Symptom: The DAG fails because it cannot find the required connection or variable.
- Fix: Double-check in the Admin > Connections and Admin > Variables sections of the Airflow UI. Make sure you’ve added the correct connection (e.g., API keys, S3 credentials) and variables needed for your DAG.

Debugging Best Practices

1. Use Logs

Logs are your best friend when it comes to debugging in Airflow. For each task, you can inspect detailed logs to see what went wrong. Logs provide tracebacks for Python tasks, details on API calls, and more.

How to check logs:

- Navigate to the Airflow UI.
- Click on the DAG run.
- Click on the task that failed.
- In the task instance view, you'll find the logs button to inspect what went wrong.

2. Test Tasks

Individually When developing or fixing your DAG, it's a good practice to test tasks individually before running the whole DAG. You can do this using the Airflow CLI:

```
airflow tasks test <dag_id> <task_id> <execution_date>
```

3. Check Environment Setup

Many issues can stem from misconfigurations in the environment, especially when deploying Airflow in production or across different environments. Double-check:

- **Environment variables:** Are you missing any required variables?
- **Permissions**: Does Airflow have the necessary permissions for interacting with APIs, databases, or cloud services?
- **AWS credentials:** If using AWS services, ensure IAM roles and credentials are correctly set.

4. Use airflow dags backfill for Debugging

Airflow provides a backfill feature, which allows you to run historical DAGs to troubleshoot and test workflows. Use the following command to backfill a DAG:

```
airflow dags backfill -s <start_date> -e <end_date> <dag_id>
```

5. Enable Task Retries

Airflow has built-in retry mechanisms that can help automatically recover from transient errors (e.g., network issues, temporary API outages). Always configure retries with a reasonable retry_delay:

```
default_args = {"retries": 3, "retry_delay": timedelta(minutes=5)}
```

By mastering these troubleshooting techniques, you'll be able to quickly identify and resolve errors, ensuring that your Airflow workflows run smoothly in production. This is all for this blog, I will see you in the next one. Until then, stay healthy and keep learning!

Follow

Save

THANK YOU
FOR DOWNLOADING

If you find this document helpful, I'd appreciate it if you could like, share, and follow me for more updates and insights!

Like

Share

Ganesh R

Senior Azure Data Engineer

@rg_data_talks

@rganesh0203