**Airflow:**

**Regarding the connection between PyCharm, Docker, and Airflow:**

1. Development Environment (PyCharm): You develop your DAGs and other Python scripts in PyCharm. PyCharm provides a comfortable and feature-rich environment for writing and debugging code.
2. Docker: You use Docker to run your Airflow environment. Docker ensures that all the necessary dependencies and configurations for Airflow are present and consistent across different environments.
3. Localhost:8080: Airflow's web interface runs on port 8080 by default. When you start Airflow using Docker, it exposes port 8080 from the Docker container to your local machine's port 8080. So, when you access localhost:8080 in your browser, you're actually accessing the Airflow web interface running inside the Docker container.

Airflow:[ Workflow Management Tool]

Open source tool to programmatically author,schedule and monitor workflows

We can easily visualize our datapipeline dependencies ,process,logs,code,trigger,tasks,success status

Goal : is to Manage work flow

Airflow is used for :

Scheduling and Orchestration of Data pipelines and workflows

Orchestration – refers to - sequencing ,co-ordination,scheduling ,managing of complex data pipelines from diverse resources.

These pipelines deliver datasets that are ready for consumption either by BI Applications and MI Models that support Big data Applications.

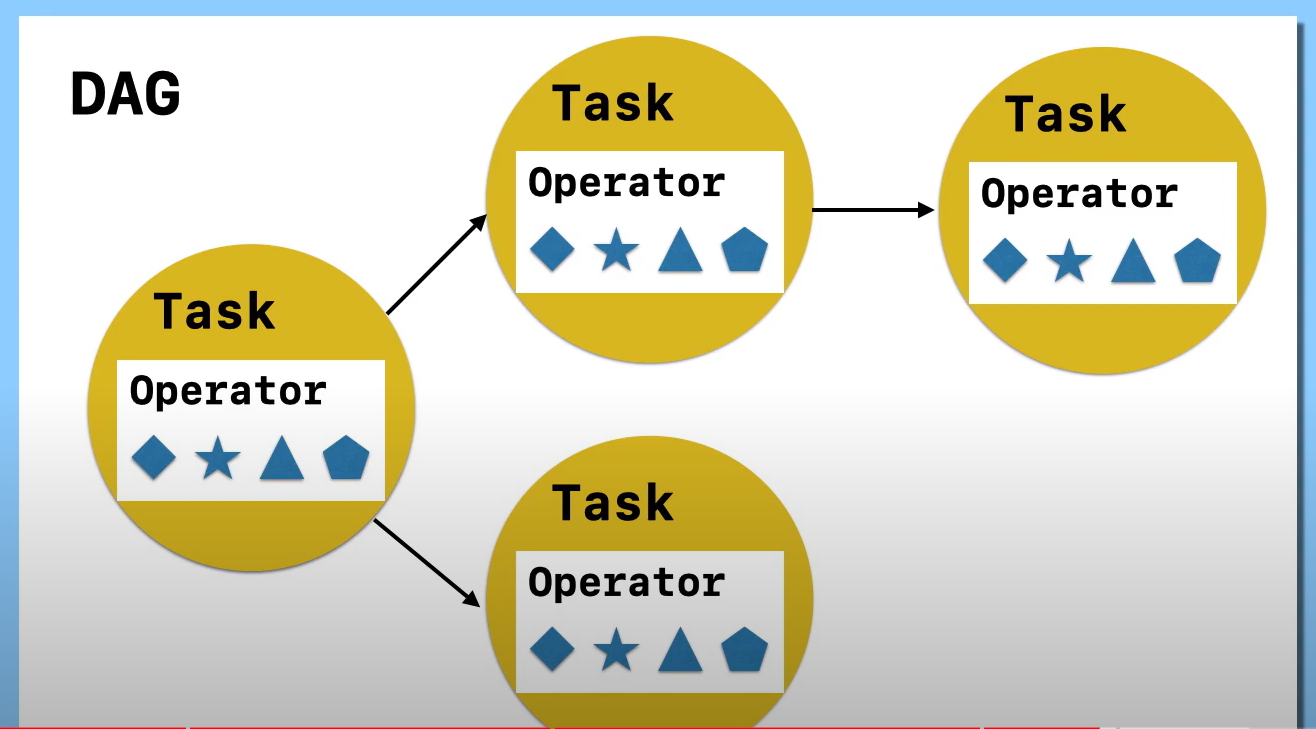
Workflows [Represented as DAG in Airflow]

* sequence of tasks
* Series of tasks that need to be executed in the specific order
* Collection of different tasks is DAG

**D**irected- means task moves in one direction

**A**cyclic– means there are no loops task do not move in circles

**G**raph – visual representation of different tasks



Tasks :

* Basic unit of execution in Airflow
* Represented as node in DAG
* Written in Python

Operator:

* To create a task , we need operator
* It is like a function provided by the airflow

Types of operator:

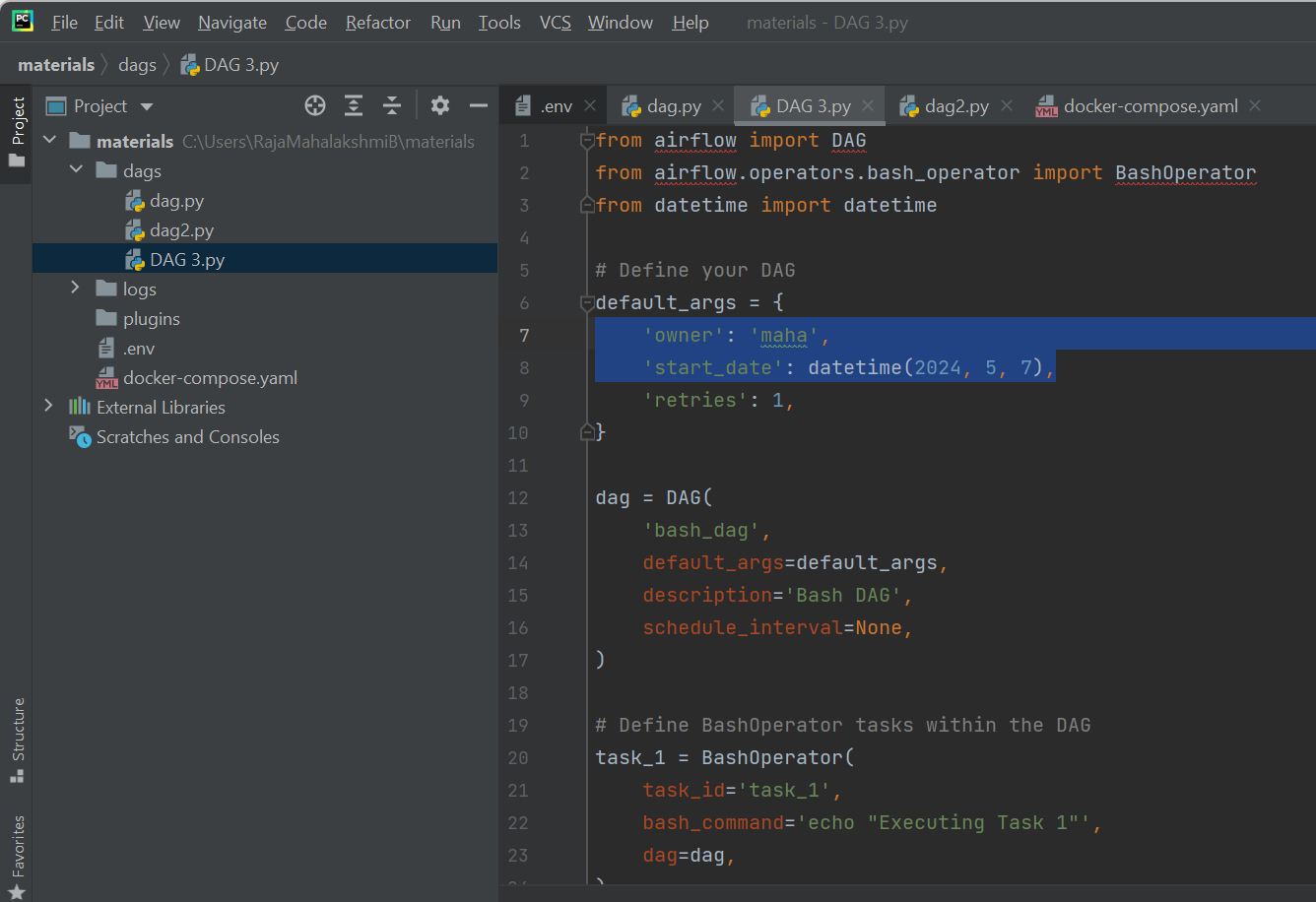
1.Bash operator **–** executes bash command

2.Python operator –calls an arbitrary python function

3.email operator –sends an email

4.customised operator

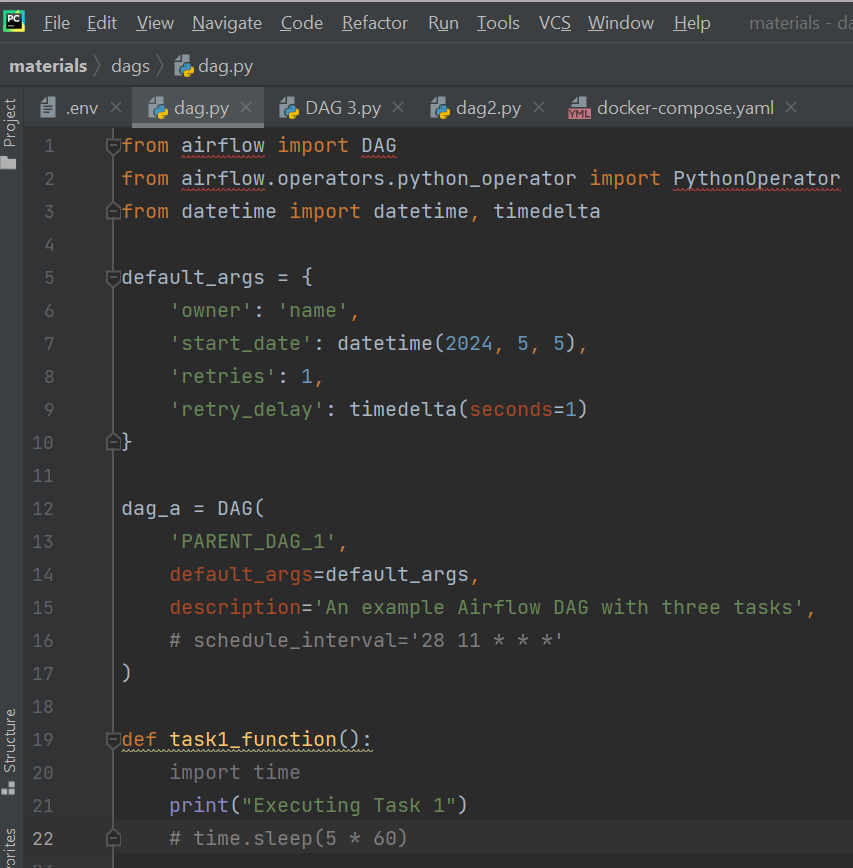
Bash Operator:

****

**A computer screen shot of a black screen

Description automatically generated**

**2.Python operator:**

****

**A screenshot of a computer program

Description automatically generated**

**A computer screen shot of a program code

Description automatically generated**

Executors:

* Used to run all these DAG
* Basically determine how your tasks will run

Types of executor:

1.Sequential:

If you want to run the task sequentially you can use this executor

2.Local:

If you want to run the task in parallel in a single machine you can use this executor

3.Celery:

If you want to distribute your tasks across Multiple machines, then you can use the celery executor

Task life cycle:

A screenshot of a computer

Description automatically generated

**Scheduled**: This is the initial state where the scheduler determines that a task instance needs to be run. This can happen based on various factors such as the schedule of the DAG (Directed Acyclic Graph) containing the task, or when all its upstream tasks have finished successfully.

**Queued**: Once the task is determined to be scheduled, the scheduler sends it to the executor to be run. The executor maintains a queue of tasks waiting to be executed by worker processes.

**Running**: When a worker process becomes available, it picks up a task from the queue and starts running it. The task is now in the ‘running’ state.

**Success**: If the task finishes running successfully, without errors, it transitions to the ‘success’ state.

**Failed**: If the task encounters an error during execution, it transitions to the ‘failed’ state. Airflow can then be configured to retry the failed task a certain number of times before giving up completely.

**Upstream\_failed**: If a task has upstream tasks (meaning tasks that need to be completed before it can run), and one of those upstream tasks fails, the task transitions to the ‘upstream\_failed’ state.

**Skipped**: A task can be skipped for various reasons, such as when it’s part of a branch in a DAG where a different branch was chosen, or when the trigger\_rule is set to one\_success and an upstream task has already succeeded.

**Up\_for\_retry**: If a task fails but has retries remaining according to the configuration, it transitions to the ‘up\_for\_retry’ state. The scheduler will then schedule the task for another attempt at a later time.

**Shutdown**: If a running task is externally terminated by the user or another process, it transitions to the ‘shutdown’ state.

Air flow architecture:

**A diagram of a computer

AI-generated content may be incorrect.**

Apache Airflow is a tool used to orchestrate workflows (i.e., automate and schedule tasks). Its architecture consists of different components working together to manage and execute workflows efficiently.

Key Components of Airflow

1. User

* The user interacts with Airflow through a web interface to create, monitor, and manage workflows.

2. Master Node (Central Control)

* The Master Node is responsible for managing the entire workflow execution. It consists of:
  + Web Server: Provides a UI for users to view and control workflows.
  + Scheduler: Determines when and where tasks should be executed.
  + Executor: Runs the tasks and ensures they are completed.
  + Metadata Database: Stores details about workflows, execution history, and configurations.

3. Queue (Task Buffer)

* The queue acts as a holding area where tasks are placed before being picked up by workers.
* The scheduler puts tasks into the queue, waiting for an available worker to execute them.

4. Worker Nodes (Task Executors)

* Workers are responsible for executing tasks assigned to them.
* Each worker picks up tasks from the queue and runs them independently.
* Multiple workers allow Airflow to run tasks in parallel, making execution faster.

How it Works

1. The user creates and schedules a workflow (DAG) using the Airflow UI.
2. The scheduler places tasks into a queue based on their dependencies and schedule.
3. Available workers pick up tasks from the queue and execute them.
4. After execution, the metadata database stores the task status and logs.
5. The user can monitor everything through the web UI.
6. the tasks and update their status in the database.

**Sensors:**

Special type of operator that are designed to wait for something to occur

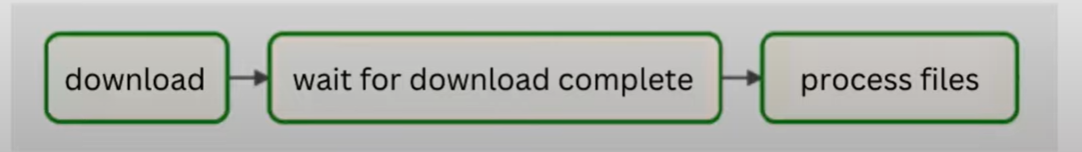
Ex: wait for,

* + - Specified time,
    - a file to land,
    - an external event to happen

And then ,

* Succeed so their downstream tasks can run
* Fail after max retries and time out

[soft fail : skip task ]



Because, the sensor has to keep on checking for the condition of the given state if its met or not

**So, this check operation is performed in two different modes**

**1.Poke**: In this case the sensor takes up the full workout slot for its entire runtime

**2.Reschedule** : In this case the sensor takes up the worker slot only when it is checking and sleeps for set duration between checks

**When to use these 2 modes:**

* Use **poke mode** when you need continuous monitoring or frequent checks for an external condition, especially when the time interval between checks is short.
* Use **reschedule mode** when you want periodic checks with built-in retry logic, or when the condition may not be immediately met but is expected to be met within a reasonable timeframe.

**Types of Sensors:**

**1. SQL Sensor:**

* **What it does:** Checks if a certain condition in a SQL script is met.
* **Example:** You might use it to wait until a specific database record is updated or a query returns a certain result.
* **Use case:** Waiting for a database operation to complete before proceeding with downstream tasks.

**2. S3 Key Sensor:**

* **What it does:** Checks if a file exists in an Amazon S3 bucket.
* **Example:** You could use it to wait until a file is uploaded to a specific S3 location.
* **Use case:** Waiting for a file to be available for processing in an S3 bucket.

**3. DateTime Sensor:**

* **What it does:** Waits until a specific date and time before proceeding.
* **Example:** You might want to wait until midnight on a certain date before executing the next tasks.
* **Use case:** Scheduling tasks to run at specific times.

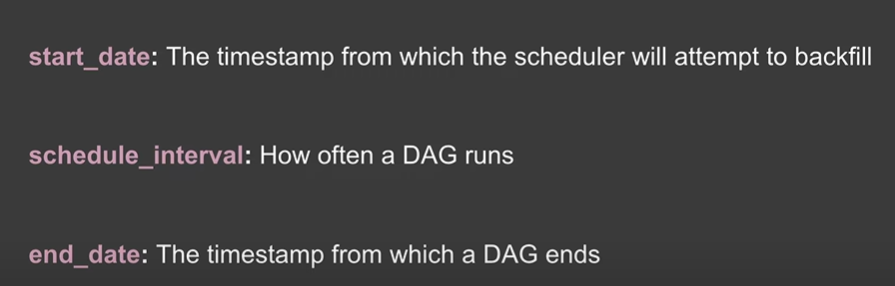
**4. Batch Sensor:**

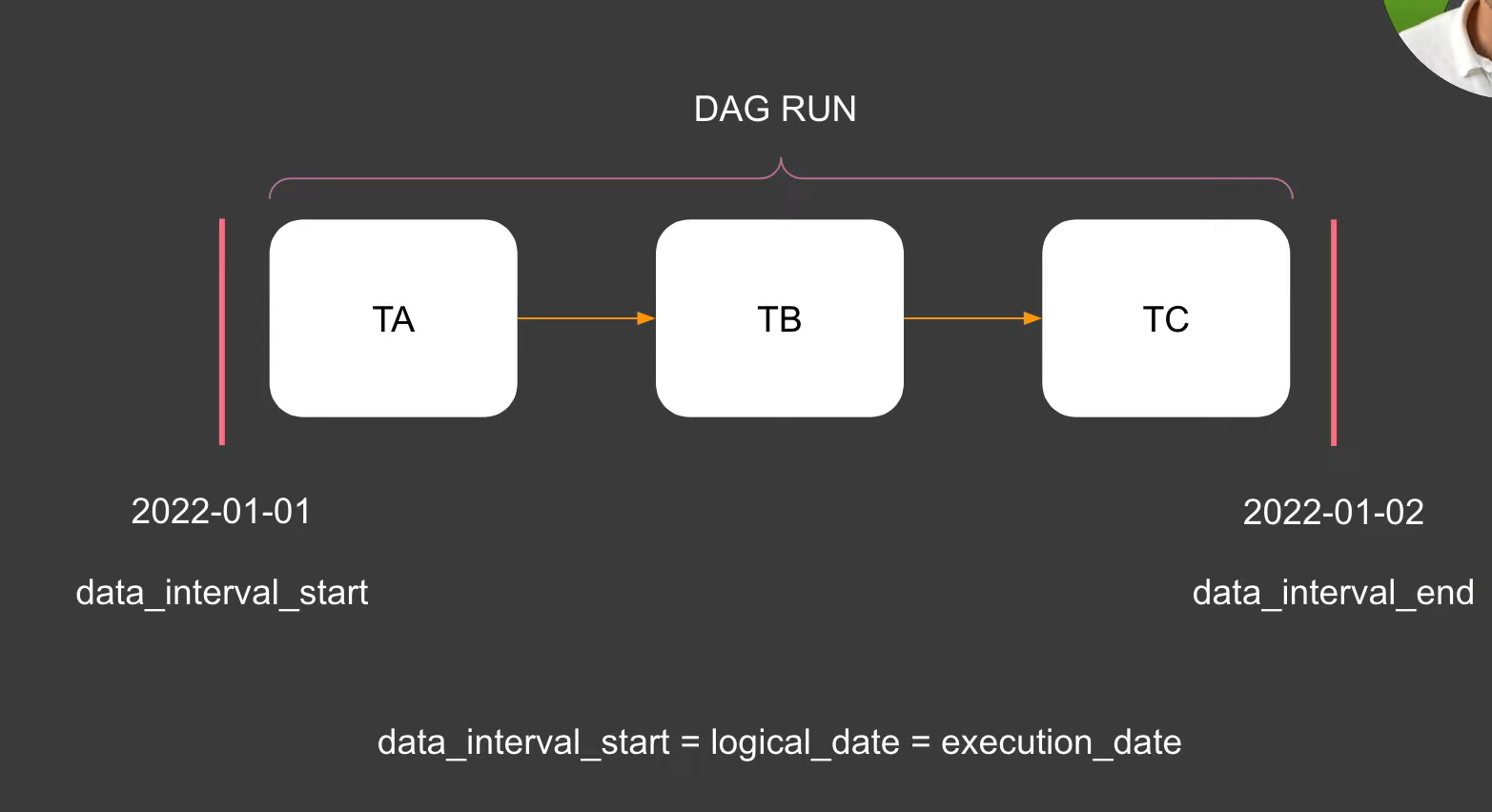
* **What it does:** Allows you to set criteria for a batch script to be considered successful.
* **Example:** You could wait until a batch job finishes processing a certain number of records.
* **Use case:** Waiting for batch processing tasks to complete before continuing.

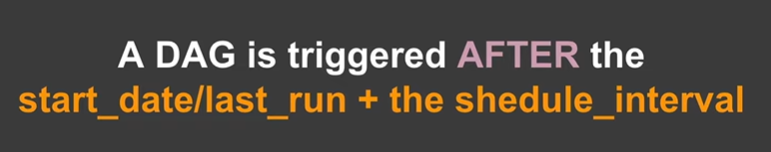
**5. External Task Sensor:**

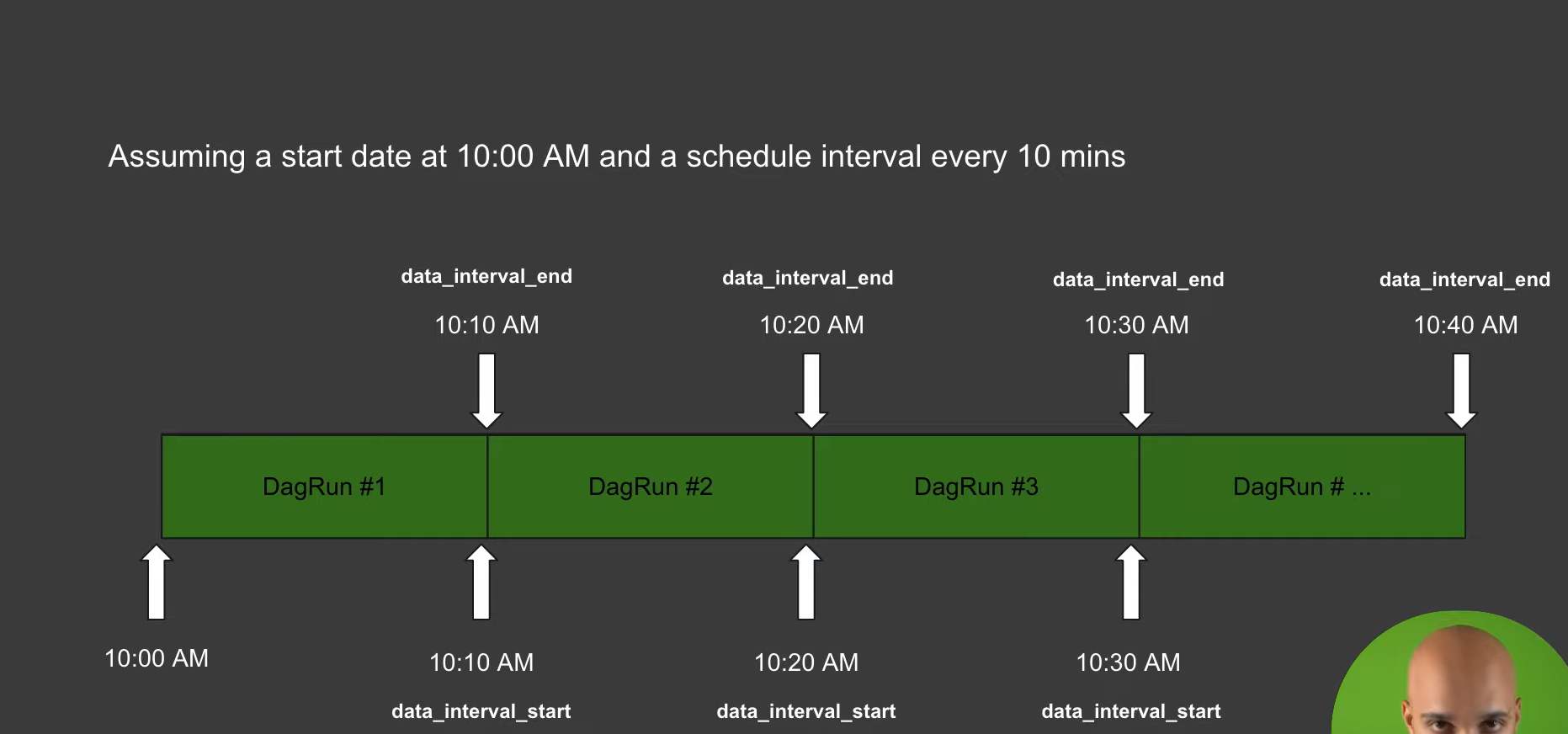
* **What it does:** Waits for a specific task from another DAG to complete.
* **Example:** You might use it to wait until a data extraction task from another workflow finishes.
* **Use case:** Coordinating tasks between different workflows.

**Air flow DAG Scheduling:**

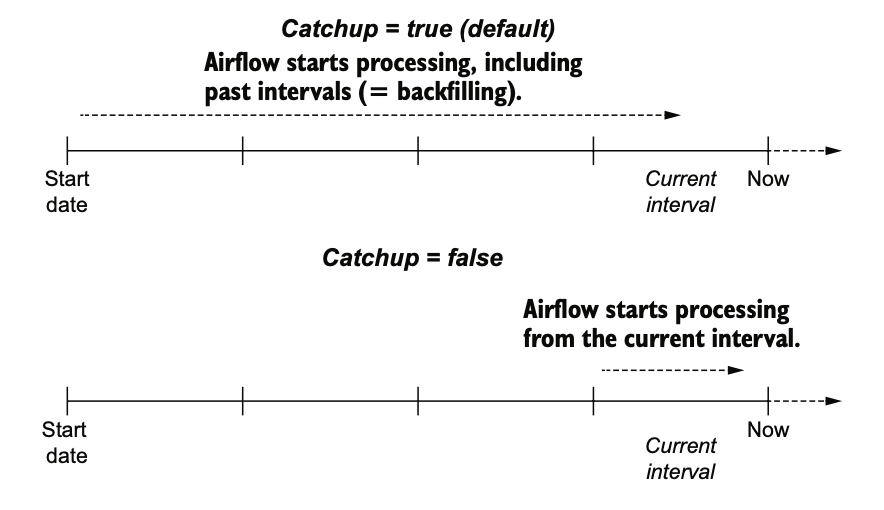








**Catch-Up:**



* Airflow's catch-up feature allows DAGs (Directed Acyclic Graphs) to execute tasks for periods when the DAG was inactive.
* **Purpose:** It ensures that historical data is processed, even if the DAG is created or modified after the data's timestamp.
* **Usage :**Enabling catch-up ensures that all eligible DAG runs from the DAG's start date to the current date are processed.
  + It's particularly useful when implementing new DAGs or modifying existing ones, ensuring no data is missed during the transition.
* **Configuration:**
  + Catch-up behavior can be controlled at both the DAG level and the individual task level.
  + It can be enabled or disabled in the DAG definition using the **catchup** parameter.
* **Example:** If a DAG is created today but should have started processing data from a week ago, enabling catch-up ensures that the DAG processes data for the entire week.

**Backfill:**

* Backfilling in Airflow involves triggering DAG runs for specific time ranges in the past to ensure that historical data is processed.
* **Purpose:** It fills gaps in data processing caused by DAG modifications, downtime, or other issues.
* **Usage:**
  + Backfilling is typically performed when catch-up is enabled and the DAG's schedule interval covers the time range to be backfilled.
  + It can be triggered manually using Airflow's CLI commands or programmatically through the Airflow API.
* **Example:** 1.If a DAG is scheduled to run daily and encounters a failure for a week due to maintenance, backfilling can be used to reprocess the data for the failed days once the issue is resolved.

2.Imagine you have a DAG that processes data every day. Today is May 7, 2024, but you also need the data processed for May 2, 2024. In this case, you can use the Airflow backfill command to run the DAG specifically for May 2nd.

Convert Local Time to UTC:

import pendulum

local\_time = pendulum.datetime(2024, 2, 25, 12, 0, tz="Asia/Kolkata") # IST Time

utc\_time = local\_time.in\_tz("UTC")

print(utc\_time) # Output: 2024-02-25T06:30:00+00:00 (Converted to UTC)

Convert UTC to Local time:

import pendulum

utc\_time = pendulum.datetime(2024, 2, 25, 10, 0, tz="UTC")

local\_time = utc\_time.in\_tz("Asia/Kolkata")

print("UTC Time:", utc\_time) # 2024-02-25T10:00:00+00:00

print("Local Time (IST):", local\_time) # 2024-02-25T15:30:00+05:30