10Alytics
data & strategy

APACHE AIRFLOW

by Dr Ughele E.N

Orchestrating & Scheduling Workflows

Learning objectives

By the end of this session, you should:

- understand what data pipelines really are;
- know what the differences between directed acyclic graphs and cyclic graphs are;
- be able to represent data pipelines in a graphical format;
- understand the core architectural components of Airflow; and
- know the process of using Python to define a DAG.

Airflow Introduction

Alternatives

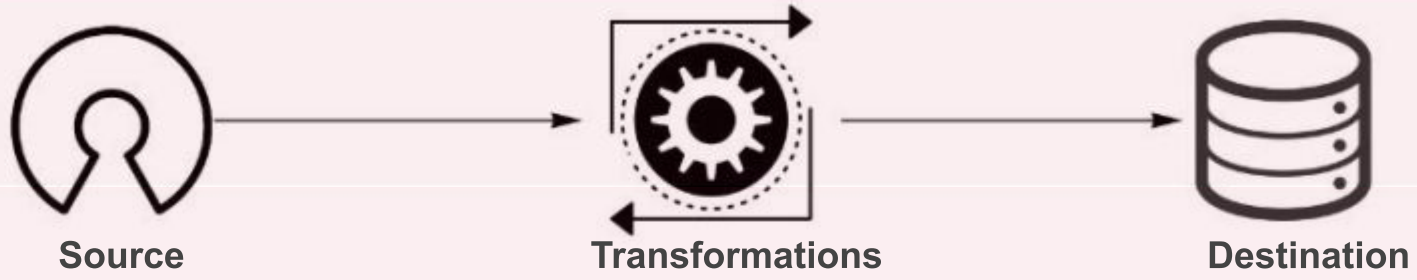
Apache Airflow

Apache Airflow is an open-source platform to programmatically author, schedule, and monitor **workflows.**

PREFECT

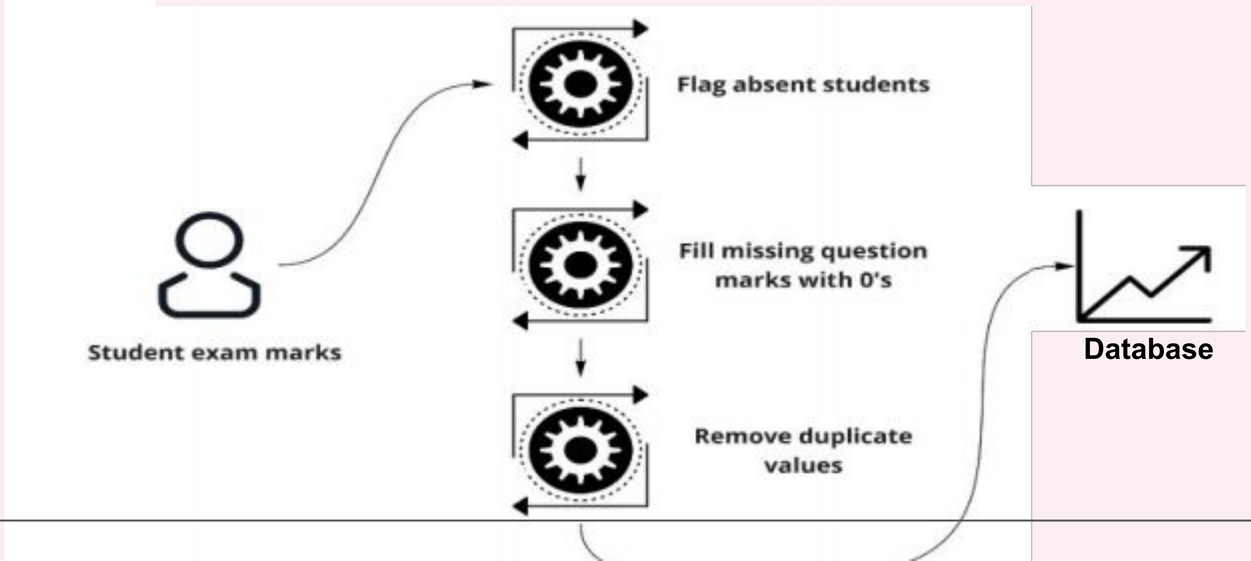
APACHE nifi

Luigi



A workflow is a sequence of tasks

In airflow, this sequence is defined by **DAG**



The Objective:

Design a data pipeline for processing student and making it ready for further analysis and insights by data consumers.

Metrics:

Class Average
Best Performing Student
Lowest and Highest Scoring Questions

Data Transformations:

Data Transformations:

Absent Students

Erroneous Data Entries

Wrong formatting

Process Overview

1

Extract data from the

dataset

2

Apply data cleaning

transformations

3

Aggregate data & make it ready for Data consumers

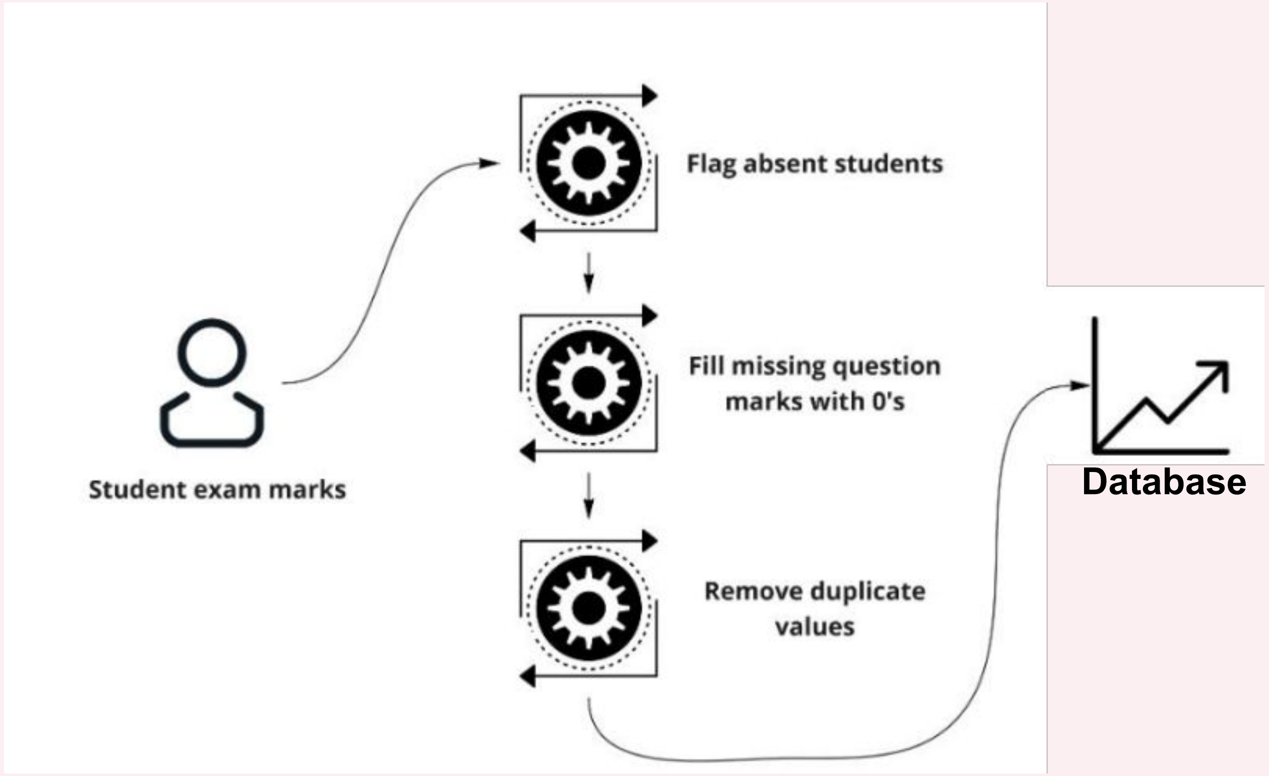
BENEFITS:

Clear Insight into students performance

Enables quick decisions-making for educators and administrators.

Identifies areas for improvement and enhances overall academic performance

Directed Acyclic Graph(DAG) is a graph structure consisting of nodes (tasks) and edges (dependencies) where each edge indicates that one task must be executed before another. **Directed** means the edges have a specific direction, and **Acyclic** means there are no cycles or loops in the graph.



Benefits of DAGs

DAGs offer a clear visualization of the pipeline's structure and data flow, making it easier to understand complex data processes

Each node in the graph typically represent an individual task or operation, allowing for modular design and easier maintenance

VISUAL CLARITY

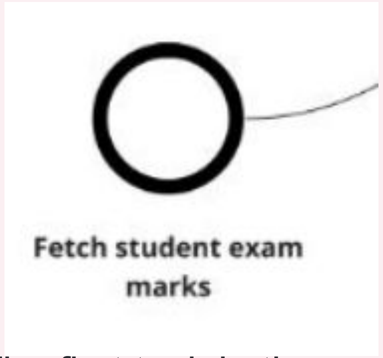
ERROR HANDLIN G

MODULARIT Y

PARALLELIS M

Visualizing the pipeline as a graph allows better error tracking and management, enabling faster debugging and troubleshooting

DAGs enable identifying parallel processing opportunities, optimizing performance, and reducing latency



The first task in the pipeline is to fetch

the student marks data.

A task represents a unit of work or astep in the workflow. It can be anything from running a Python script, executing a SQL query, transferring files , sending emails, etc. etc. Each task is a component

that performs a specific action.

Send data to database

The last task in the pipeline is to push the

transformed data to the database.

Operator:

An operator is a class that defines what the task actually does.

Airflow provides a variety of built-in operators, such as:

1. Bash Operator for executing Bash commands,
2. Python Operator for executing Python functions,
3. SQL operators for running SQL queries, and more

Operators encapsulate the logic of a task.

The execution algorithm of a DAG

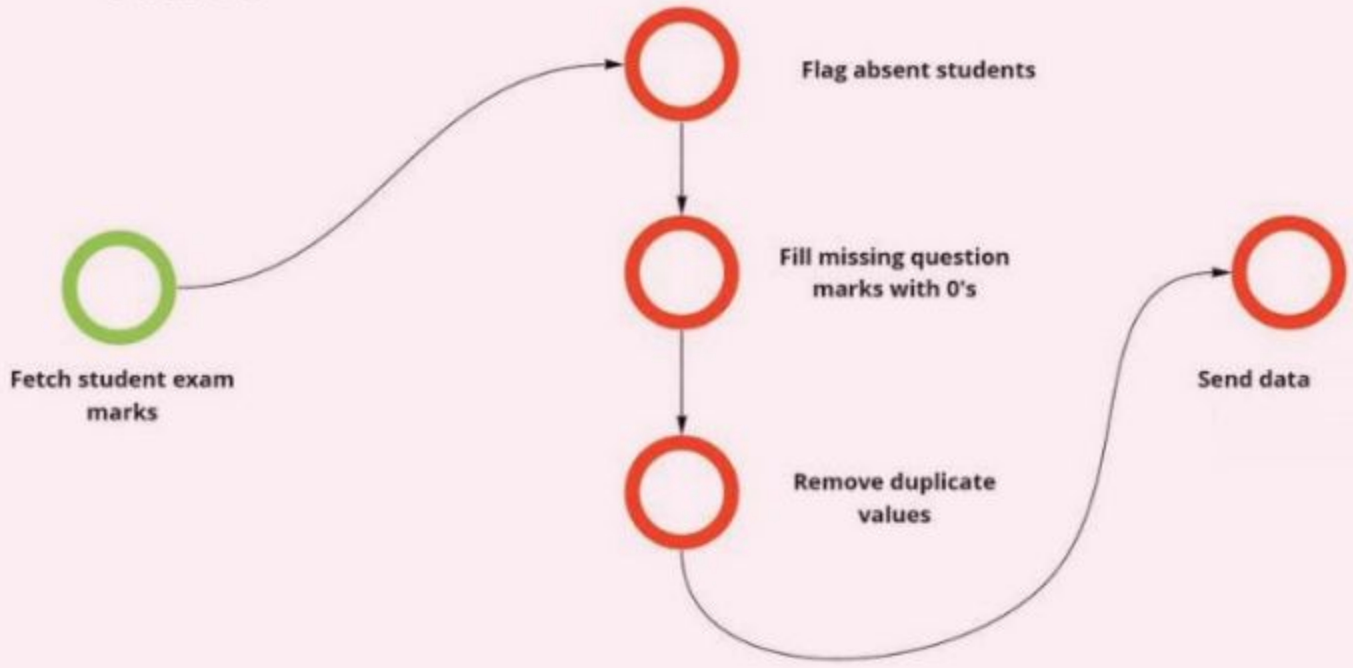
1. For each open (unexecuted) node in the DAG:
   - Identify the upstream task(s) upon which the current task is dependent.
   - Determine if all upstream tasks identified have been successfully executed.
   - If all upstream tasks identified have not been successfully executed, move to the previous task and repeat steps a) and b).
   - If all dependent tasks have been successfully executed, add the current task to a queue of tasks to be executed.

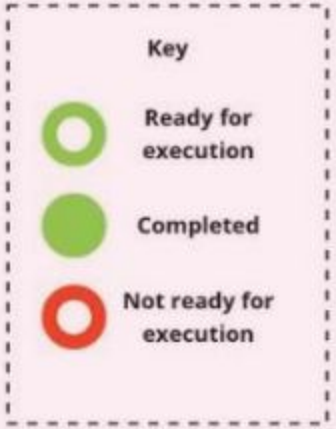
2. Sequentially execute the tasks in the queue:
   - Assign a *"completed"* status to the tasks that have been successfully executed.

3. Repeat steps 1) and 2) for the remaining tasks in the DAG

The execution algorithm of a DAG

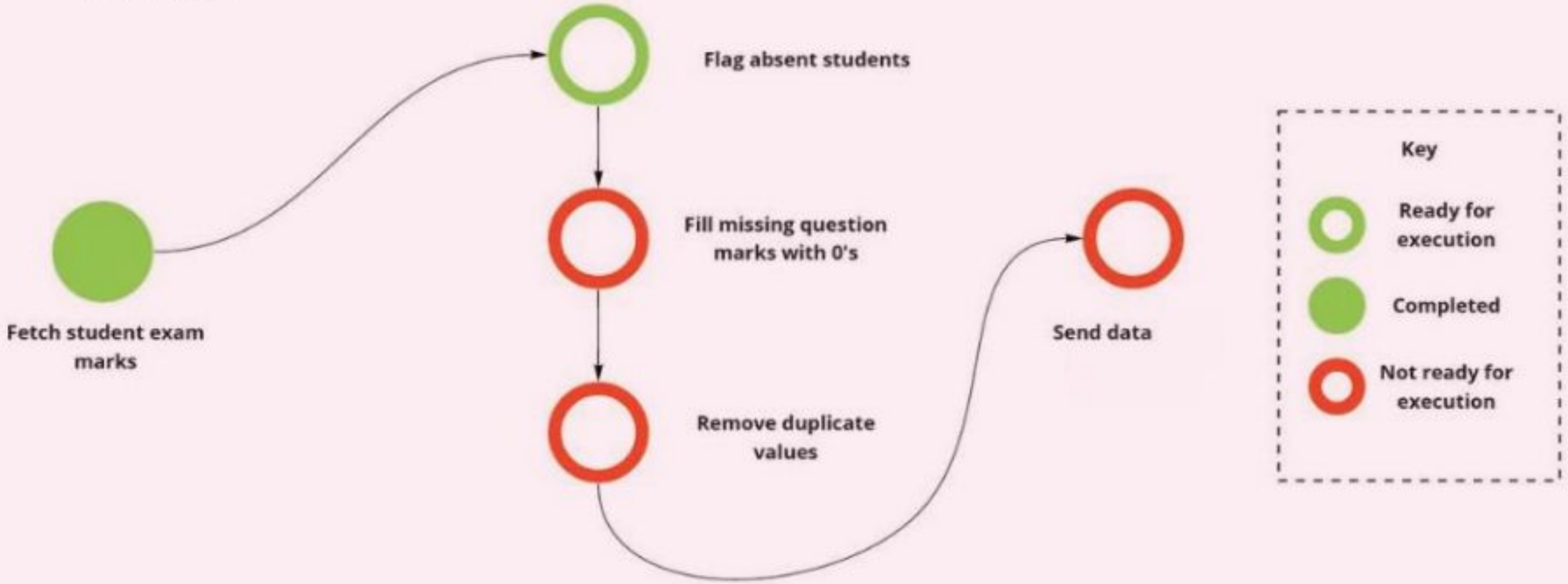
STEP: 1





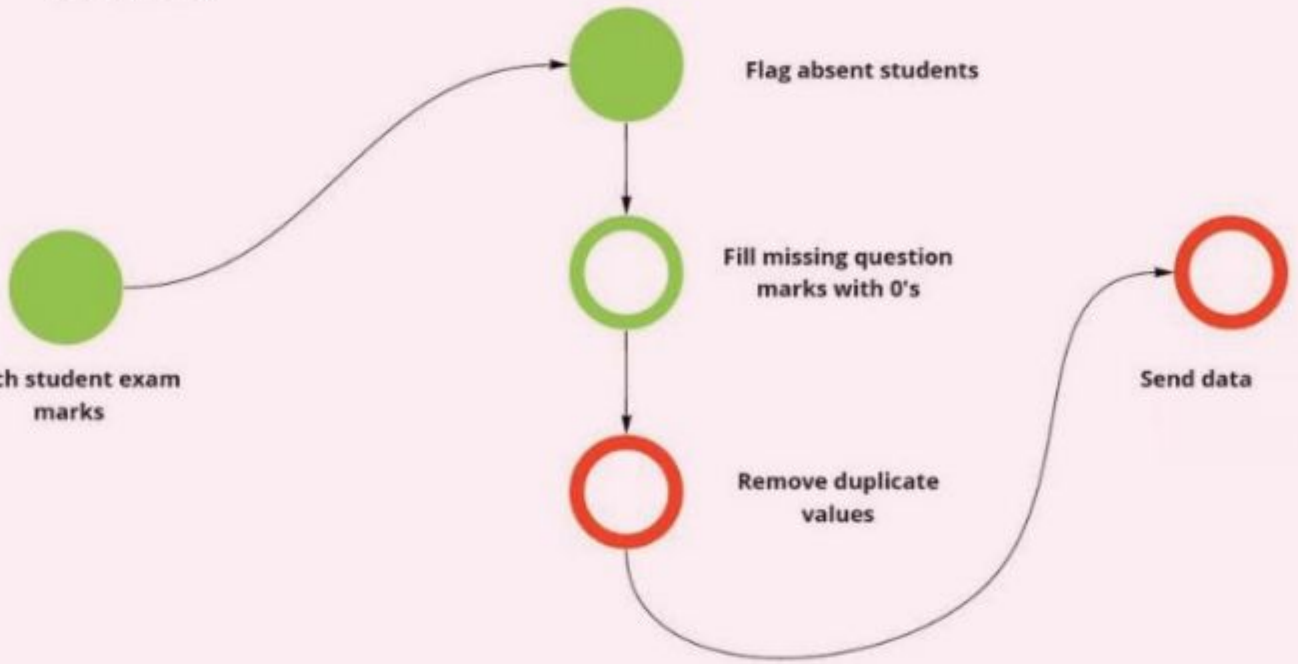
The execution algorithm of a DAG

STEP: 2



The execution algorithm of a DAG

STEP: 3

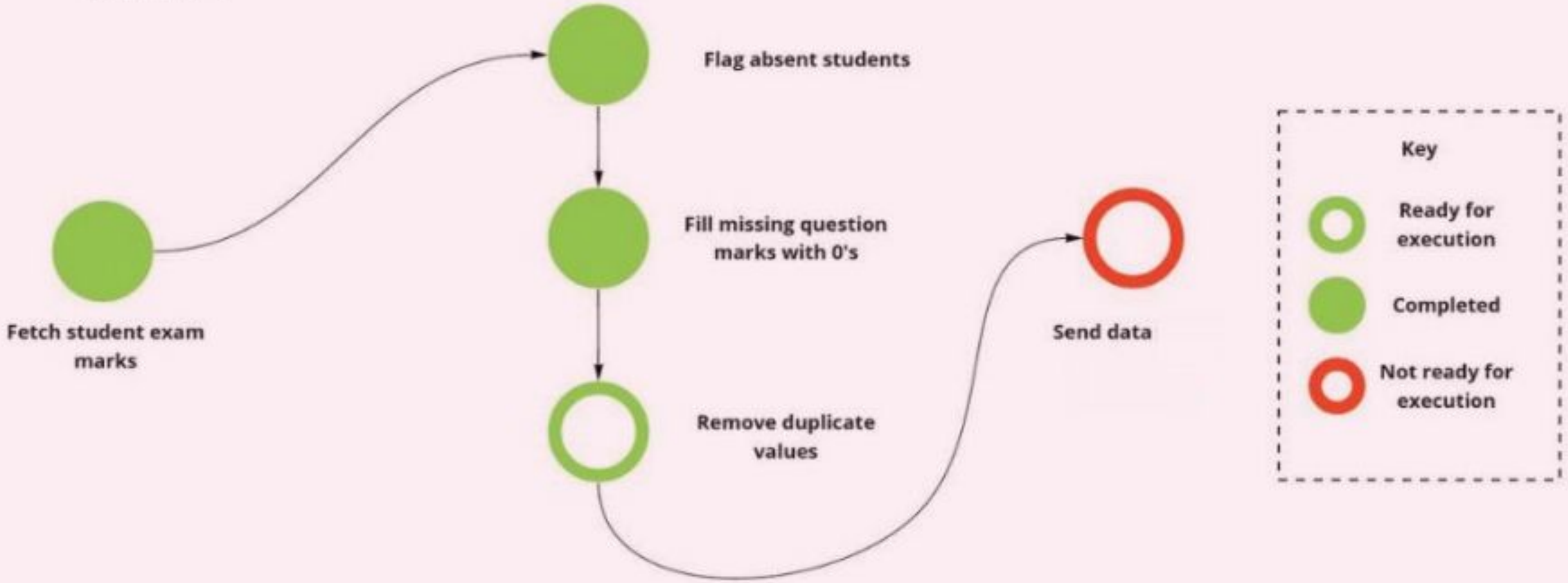


Key

○	Ready for execution
●	Completed
○	Not ready for execution

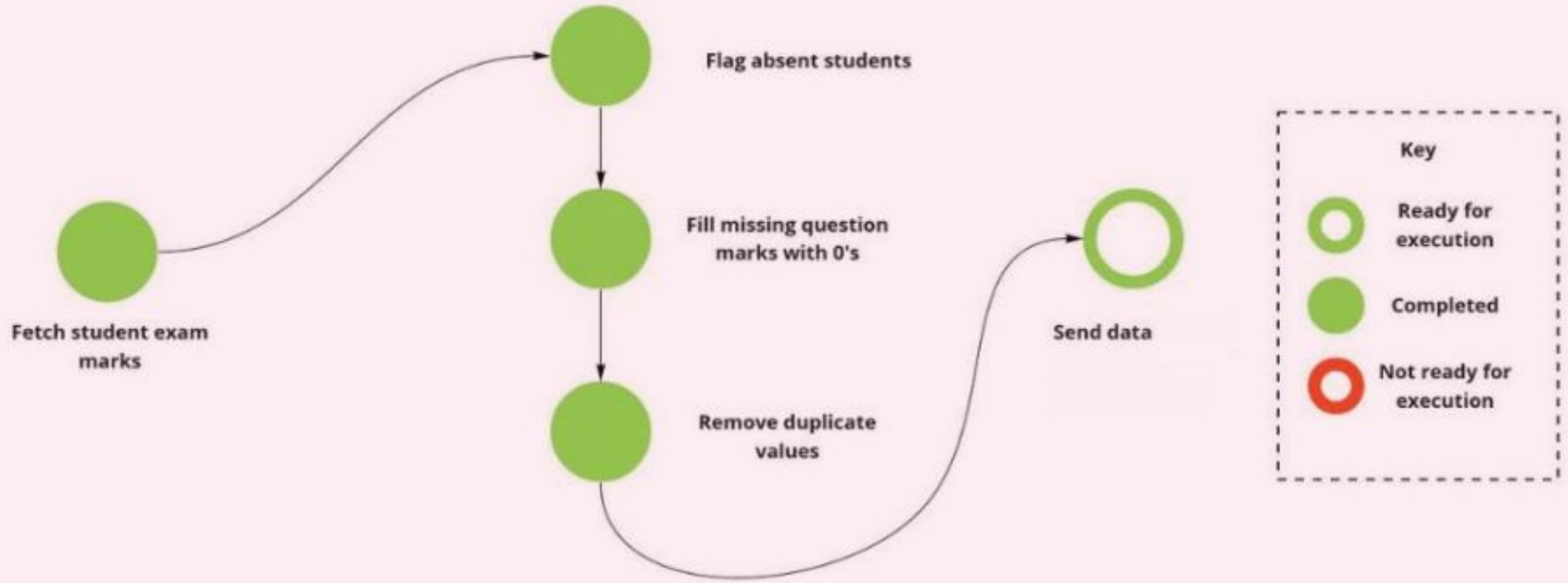
The execution algorithm of a DAG

STEP: 4



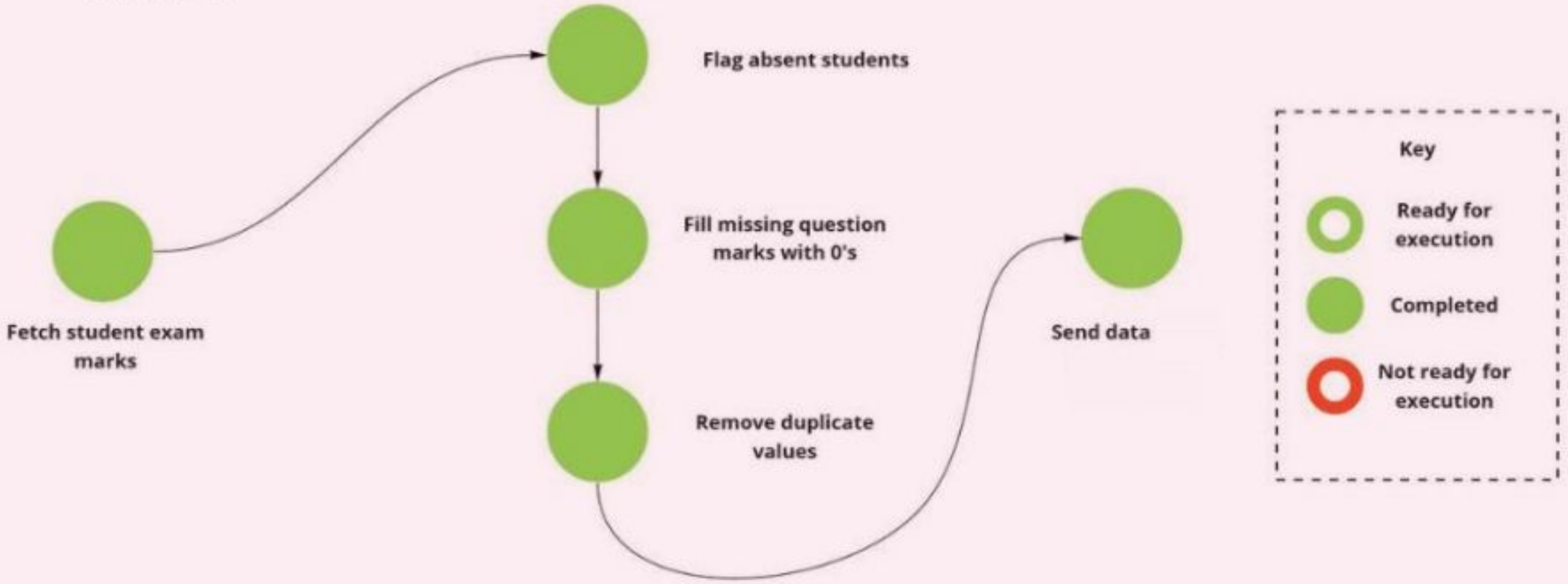
The execution algorithm of a DAG

STEP: 5

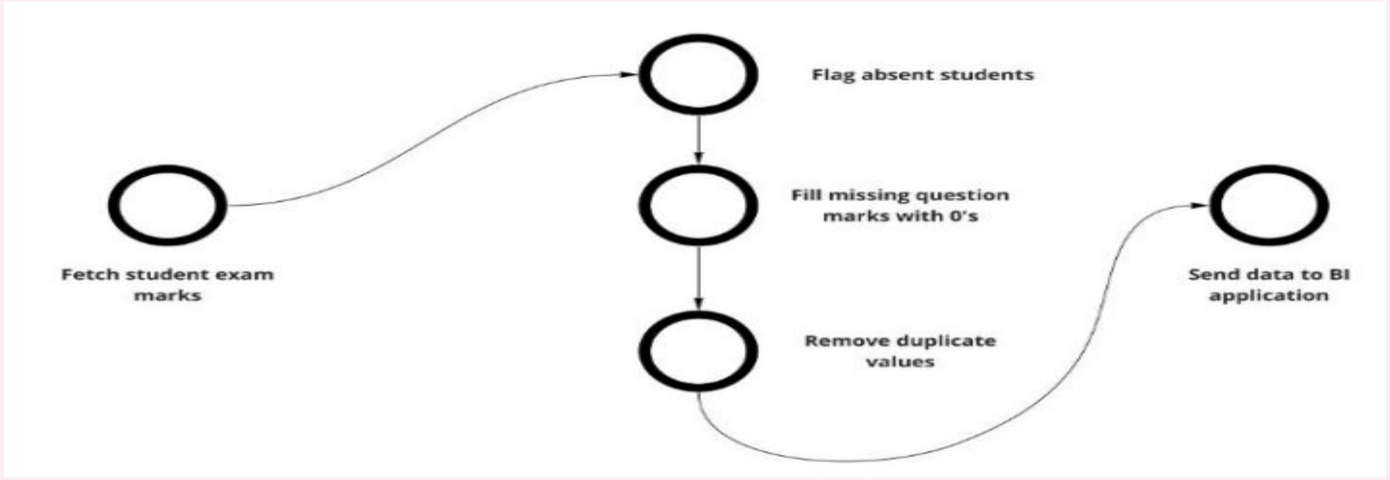


The execution algorithm of a DAG

STEP: 6



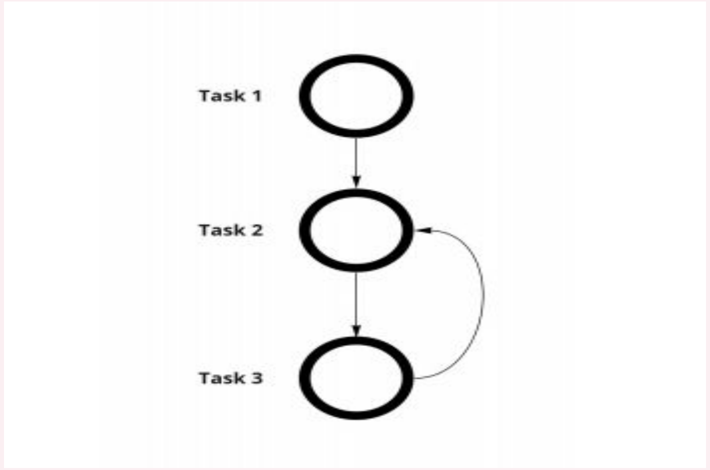
Representing our example data pipeline as a graph



- The first task in the pipeline is to fetch the student marks data.
- Similarly, the last task in the pipeline is to push the transformed data to the BI application.
- The direction of the edges indicates the direction of *task dependency* within the pipeline. For example, the arrow pointing from the *"Flag absent students"* node to the *"Fill missing question marks with zeroes“* node indicates that the flagging operation executes before the filling one.

This type of graph, where there is directional dependence, is referred to as a **directed graph**. When such a graph contains no loops or cycles (no node has a dependence on nodes further down the chain of execution), it is known as a **directed acyclic graph** (DAG). To illustrate this point, we can observe in our example graph how task dependence flows from the “Fetch student exam marks" node all the way to the "Send data to BI application" node, without ever referring to a previous task. As such, our example pipeline graph is a DAG.

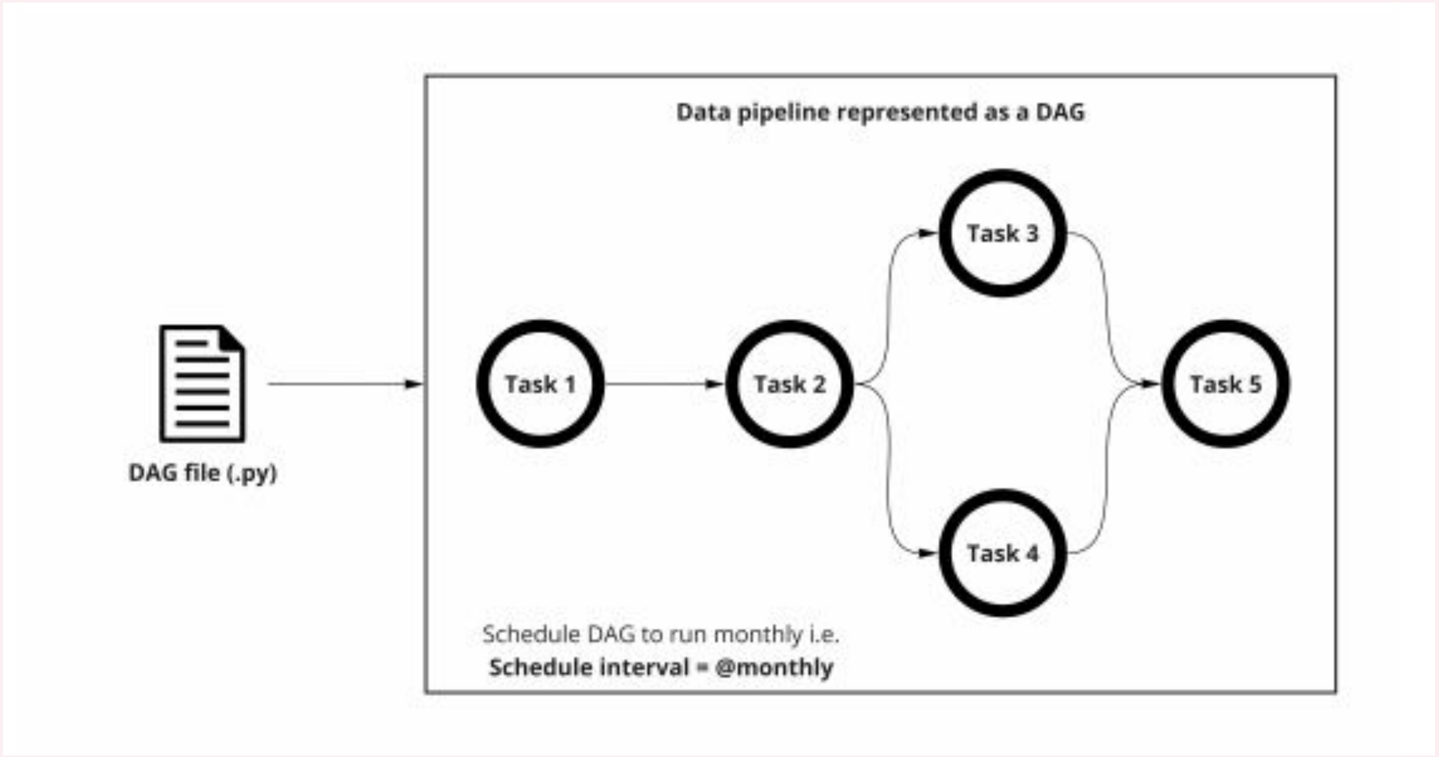
A Simple Cyclic Graph Representing A Malformed Pipeline



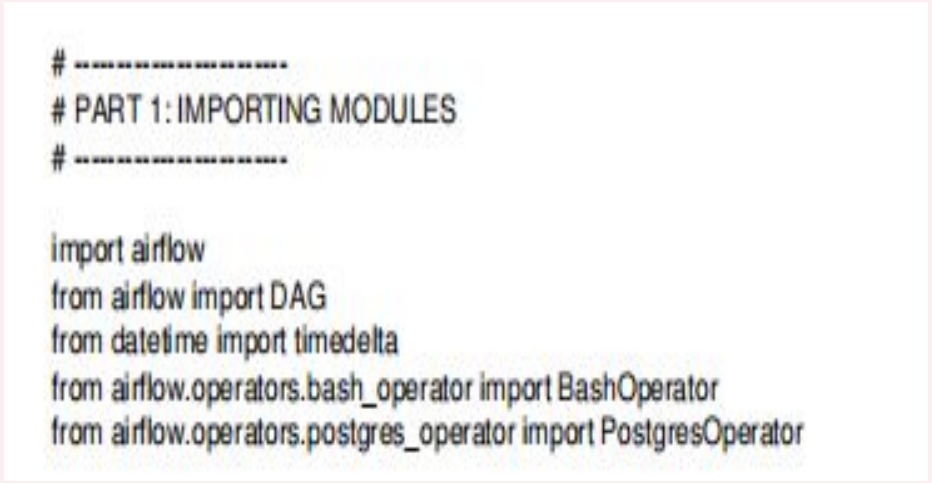
Task 2's dependence on Task 3 means that it can never successfully execute, leaving the pipeline in a deadlock.

Let's imagine for a brief moment that there was no requirement for a pipeline, represented by a graph, to be acyclic. How would this affect the pipeline's execution? Figure 4 can help us reason this out. Here we can see directed connections from Task 1 to Task 2, and then to Task 3. However, we also see that Task 3 links back to its previous task. Following this chain of dependence, once Task 1 completes, we expect Task 2 to begin. However, Task 2 can't begin as it awaits output from Task 3, which will never occur as Task 3 needs the output from Task 2. Such a scenario leads to a flaw in our simple example's execution logic, and hence, a pipeline with a *cyclic* dependence will never execute, as it will remain in a *deadlock state.*

Defining a DAG as a python script


Data pipeline represented as a DAG
DAG file (.py)
Task 1
Task 2
Task 3
Task 4
Task 5
Schedule DAG to run monthly i.e.
Schedule interval = @monthly

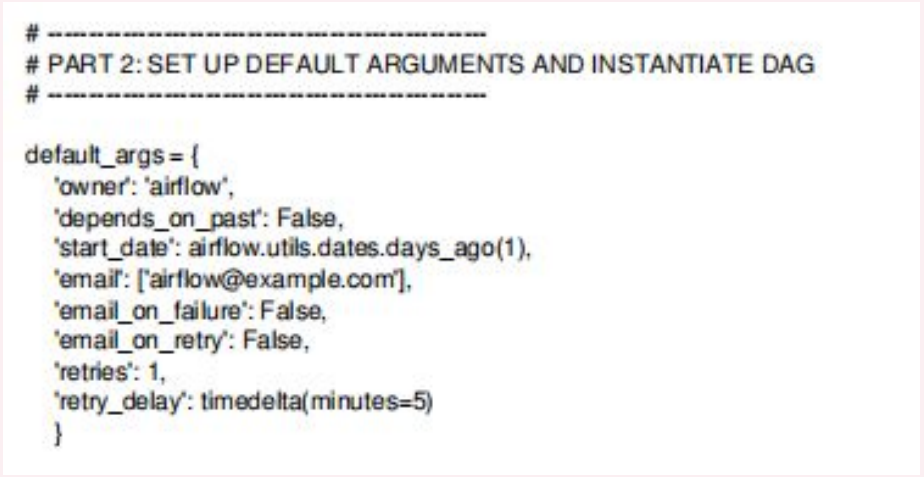
Defining a DAG as a python script



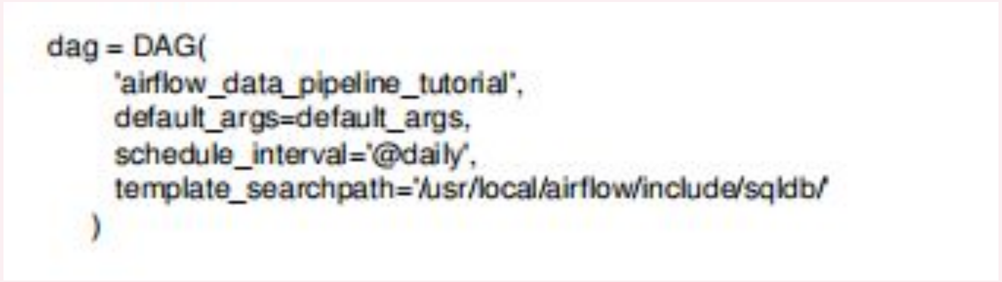
```
# -------------------
# PART 1: IMPORTING MODULES
# -------------------

import airflow
from airflow import DAG
from datetime import timedelta
from airflow.operators.bash_operator import BashOperator
from airflow.operators.postgres_operator import PostgresOperator
```

Part 1: Importing modules

Defining a DAG as a python script



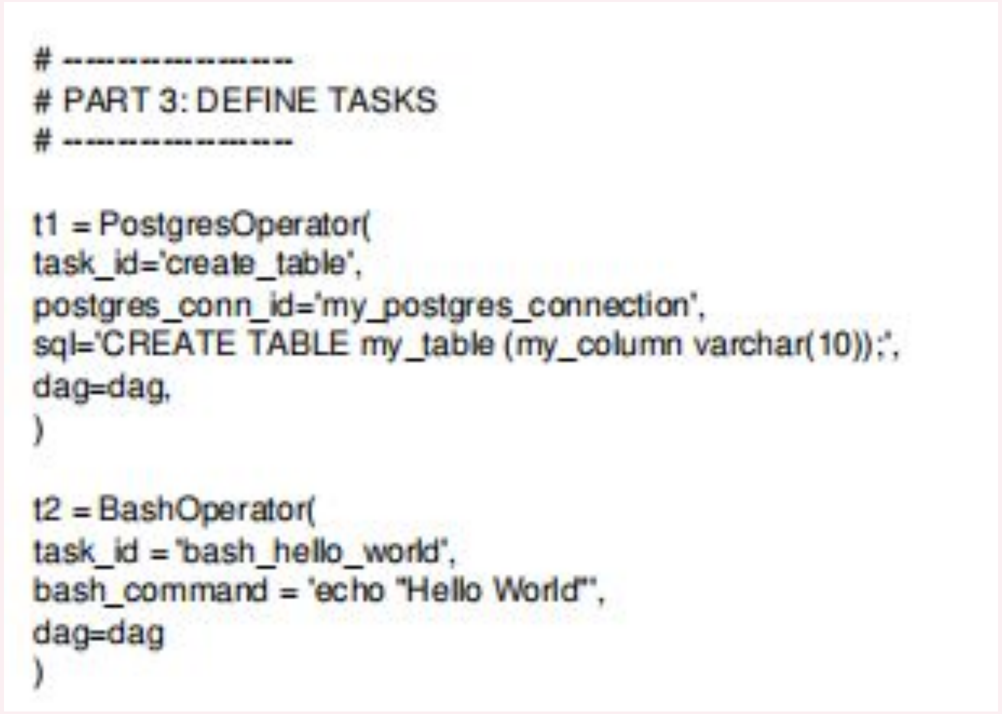
```
# ---------------------------------------------------------
# PART 2: SET UP DEFAULT ARGUMENTS AND INSTANTIATE DAG
# ---------------------------------------------------------

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': airflow.utils.dates.days_ago(1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
    }
```



```
  dag = DAG(
          'airflow_data_pipeline_tutorial',
          default_args=default_args,
          schedule_interval='@daily',
          template_searchpath='/usr/local/airflow/include/sqldb/'
      )
```

Part 2: Default arguments and DAG instantiation

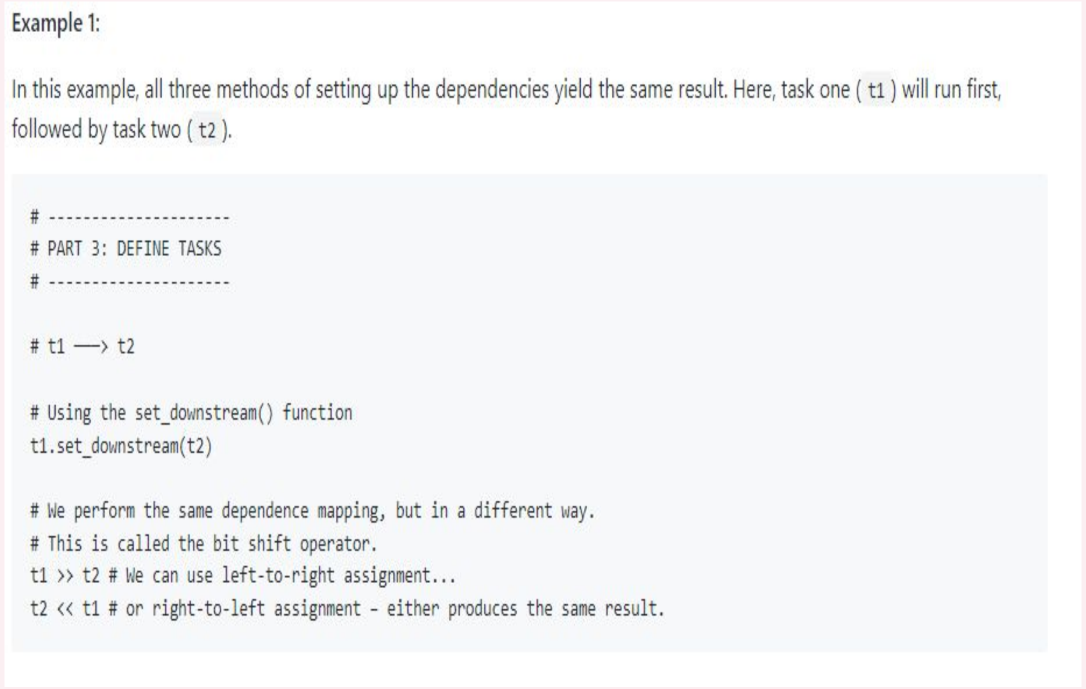
Defining a DAG as a python script



```
# ----------------
# PART 3: DEFINE TASKS
# ----------------

t1 = PostgresOperator(
task_id='create_table',
postgres_conn_id='my_postgres_connection',
sql='CREATE TABLE my_table (my_column varchar(10));',
dag=dag,
)

t2 = BashOperator(
task_id = 'bash_hello_world',
bash_command = 'echo "Hello World"',
dag=dag
)
```

Part 3: Task definitions

Defining a DAG as a python script

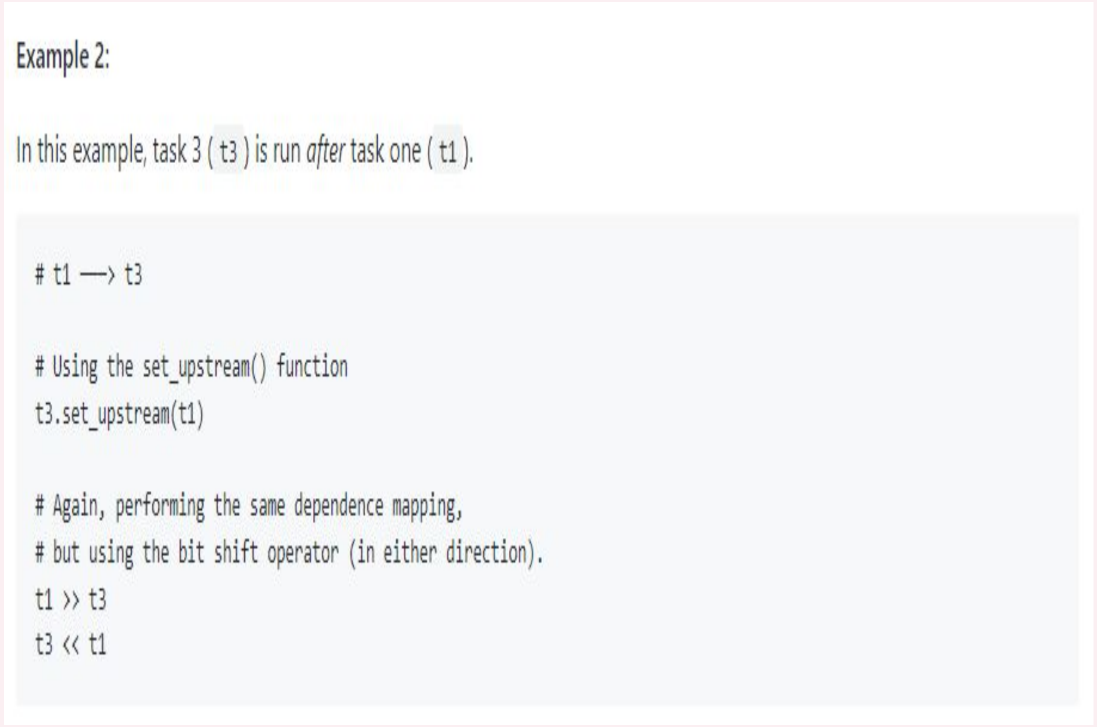

Example 1:

In this example, all three methods of setting up the dependencies yield the same result. Here, task one (t1) will run first, followed by task two (t2).

```
# --------------------
# PART 3: DEFINE TASKS
# --------------------

# t1 ——> t2

# Using the set_downstream() function
t1.set_downstream(t2)

# We perform the same dependence mapping, but in a different way.
# This is called the bit shift operator.
t1 >> t2 # We can use left-to-right assignment...
t2 << t1 # or right-to-left assignment - either produces the same result.
```

Part 4: Setting up task dependencies

Defining a DAG as a python script

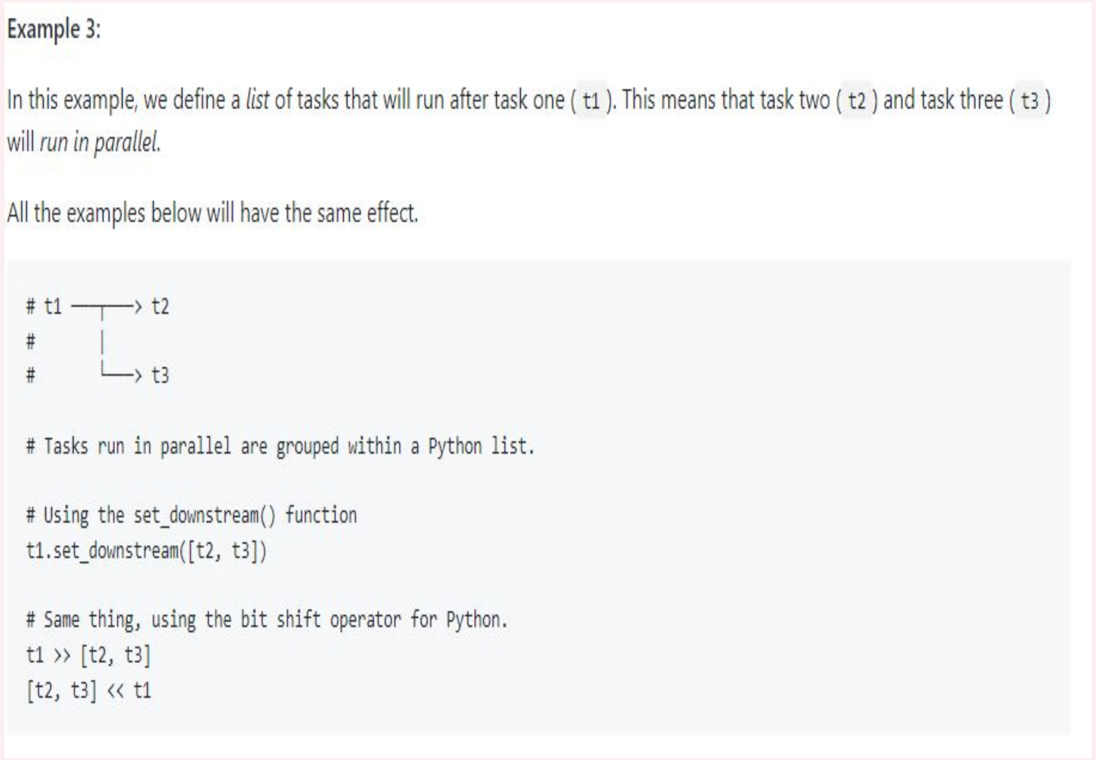

Example 2:

In this example, task 3 (t3) is run *after* task one (t1).

```
# t1 —> t3

# Using the set_upstream() function
t3.set_upstream(t1)

# Again, performing the same dependence mapping,
# but using the bit shift operator (in either direction).
t1 >> t3
t3 << t1
```

Part 4: Setting up task dependencies

Defining a DAG as a python script



Example 3:

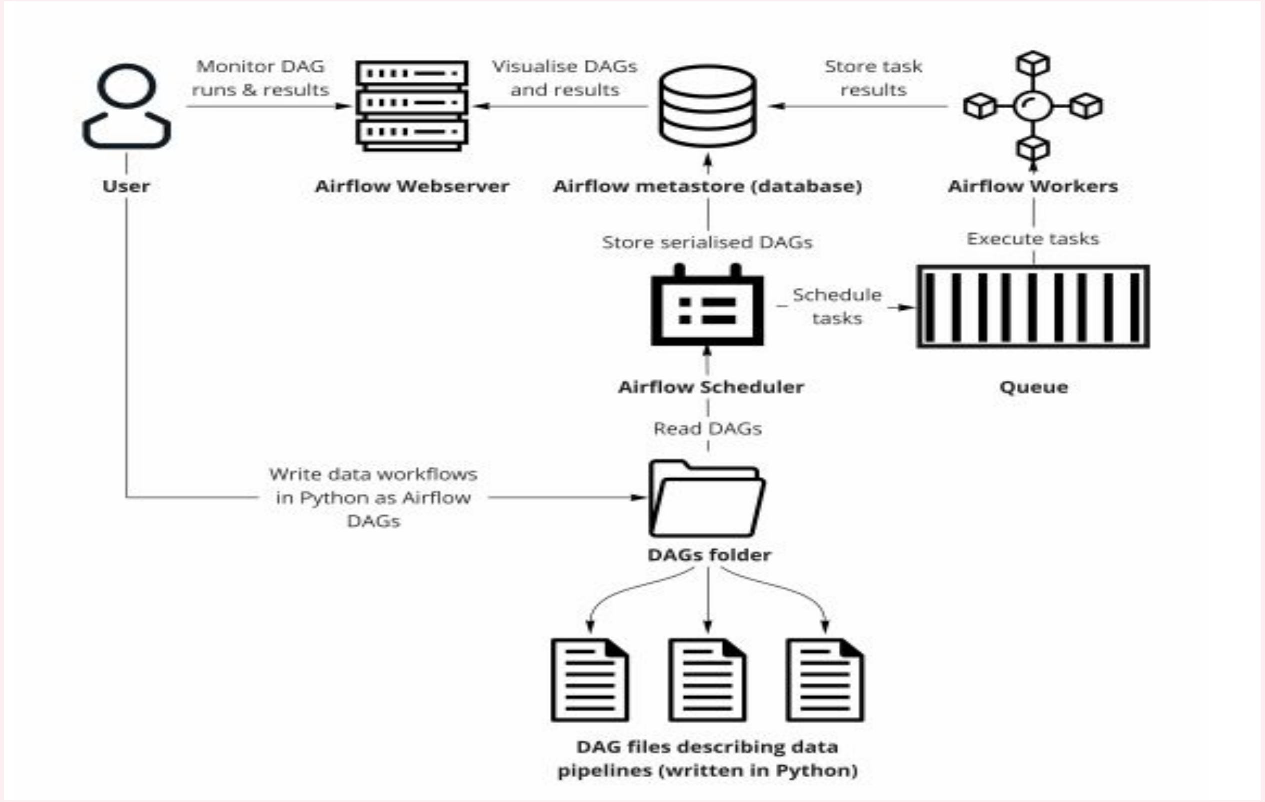
In this example, we define a *list* of tasks that will run after task one (`t1`). This means that task two (`t2`) and task three (`t3`) will *run in parallel*.

All the examples below will have the same effect.

```
# t1 ──┬──> t2
#      |
#      └──> t3

# Tasks run in parallel are grouped within a Python list.

# Using the set_downstream() function
t1.set_downstream([t2, t3])

# Same thing, using the bit shift operator for Python.
t1 >> [t2, t3]
[t2, t3] << t1
```

Part 4: Setting up task dependencies

A Typical Airflow Session


Monitor DAG runs & results
Visualise DAGs and results
Store task results
User
Airflow Webserver
Airflow metastore (database)
Airflow Workers
Store serialised DAGs
Execute tasks
Schedule tasks
Airflow Scheduler
Queue
Read DAGs
Write data workflows in Python as Airflow DAGs
DAGs folder
DAG files describing data pipelines (written in Python)

10Alytics
data & strategy

CASE STUDY FOR ZIPCO FOODS

Executive Summary for Zipco Foods

Zipco Foods is a vibrant and growing business that specializes in the sales of pizzas and cakes. As a key player in the fast-casual dining industry, Zipco Foods operates numerous outlets across the country, serving a wide variety of pizzas and cakes that cater to local tastes and preferences. With a strong commitment to customer satisfaction and quality service, Zipco Foods aims to leverage advanced data engineering solutions to enhance operational efficiency, improve product offerings, and ultimately boost profitability.

Business Problem Statement

Zipco Foods generates a significant amount of sales data daily, which is currently underutilized due to inefficient data handling and analysis processes. The primary challenge is the disparate nature of data collection and storage, with critical sales and inventory information scattered across multiple CSV files without a unified system for aggregation and analysis. This fragmentation leads to operational inefficiencies, including delays in data access, difficulty in obtaining real-time insights, and challenges in maintaining data integrity and accuracy.

Objectives & Benefits

Objectives:

- Implement a streamlined ETL (Extract, Transform, Load) pipeline to automate data processing and ensure data consistency.
- Design a database schema that supports efficient data retrieval and scalability while adhering to 2NF/3NF normalization standards.
- Develop a system for real-time data analytics to aid in decision-making processes.
- Ensure robust data governance and compliance through effective version control and data orchestration.

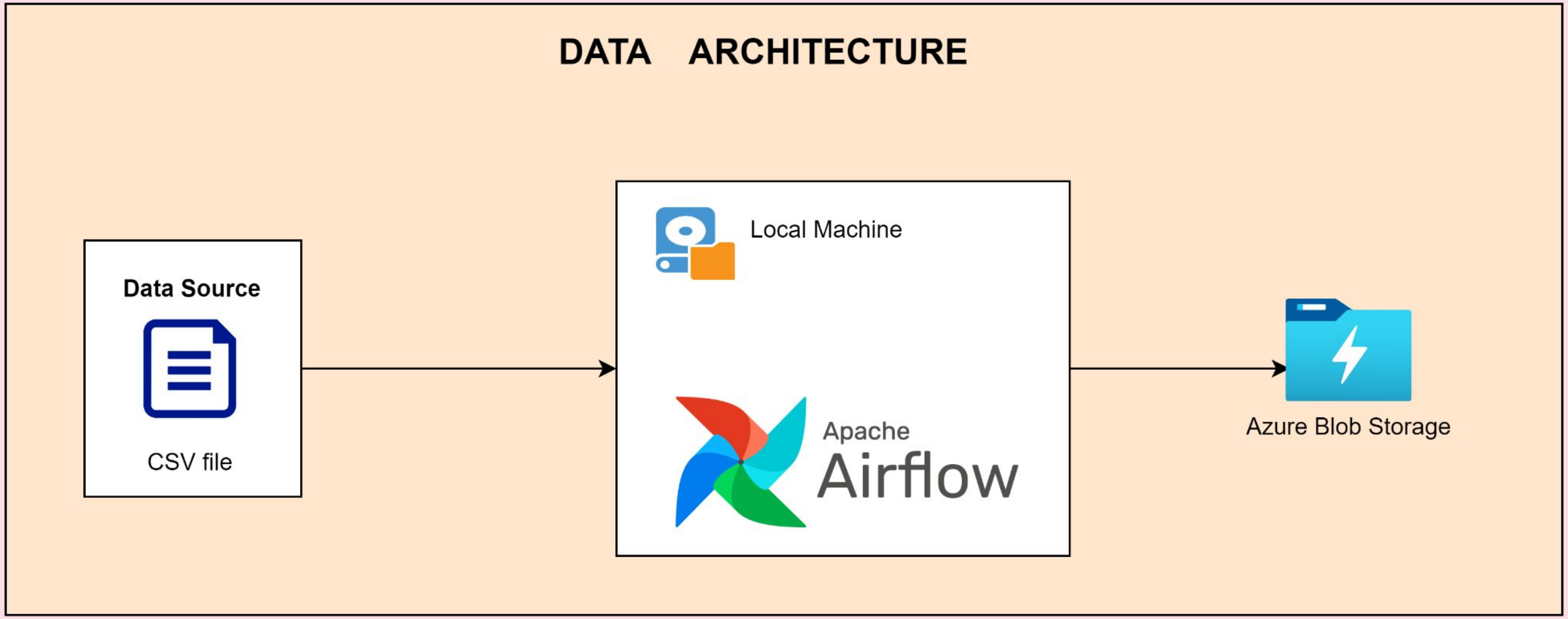
Benefits:

- Enhanced decision-making capabilities through real-time, accurate data analytics.
- Improved operational efficiency and reduced manual labor by automating data processes.
- Greater scalability and flexibility in data management, accommodating future business growth.
- Strengthened data integrity and reliability, ensuring high-quality information for strategic planning.

Tech Stack

- Python: Utilized for scripting the ETL processes, data cleaning, transformation, and analysis tasks due to its powerful libraries like pandas and NumPy.
- SQL: Employed for querying, updating, and managing the database stored in Azure, ensuring efficient data manipulation and retrieval.
- Azure Blob Storage: Chosen for its scalability and reliability, serving as the centralized data repository for storing processed data.
- Github: Used for version control, allowing for collaborative development and maintenance of the ETL scripts and other project documents.
- Apache Airflow: Orchestrates the ETL processes, scheduling jobs efficiently and monitoring the workflow of data through various stages of the pipeline.

Data Architecture



Data Source

- This is the link to the data source ⇒ LINK

Project Scope

- Data Extraction
    - Extract data from various CSV files into a Pandas DataFrame. This step involves reading large datasets efficiently, handling different data formats, and managing incomplete or corrupt data files.

- Data Transformation
    - Clean the extracted data to remove inconsistencies, duplicates, and handle missing values.
    - Transform the data to fit into a designed schema that adheres to the principles of 2NF and 3NF. This involves decomposing tables to reduce redundancy, ensuring referential integrity, and optimizing the schema for query performance.

Project Scope

- Data Loading
  - Load the cleaned and transformed data into Azure Blob Storage, which serves as the centralized repository for all analytical data.
  - Implement version control using GitHub to maintain revisions of the data transformation scripts and other configurations.
  - Use Apache Airflow to orchestrate the entire ETL process. Define DAGs (Directed Acyclic Graphs) to manage the workflow of tasks including dependencies and sequence of operations, ensuring that the data flows smoothly from extraction through to loading, with logging and error handling to manage failures or retries effectively.
  - This case study outlines the strategic approach for utilizing advanced data engineering techniques to drive operational improvements and business growth for Zipco Foods.