

Apache Airflow Reference

From first DAG to production-grade pipeline orchestration.

Last reviewed 29 Sep 2026 · Lab-tested with Apache Airflow 3.3.2 ·

Prerequisites: [Python for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/python-reference/) · [Docker](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/)

Related: [dbt](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/) · [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Kubernetes](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/kubernetes-for-de/) · [Testing and CI/CD](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/testing-cicd/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 05 — Airflow Orchestration](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/05-airflow-orchestration) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/05-airflow-orchestration)

Overview

Challenge: A typical daily pipeline consists of dependent steps that must run in order: 1. Download the previous day's sales data from object storage 2. Validate that the file is not empty 3. Load it into a staging table in the warehouse 4. Run three SQL transformations in parallel 5. Send a notification when everything completes

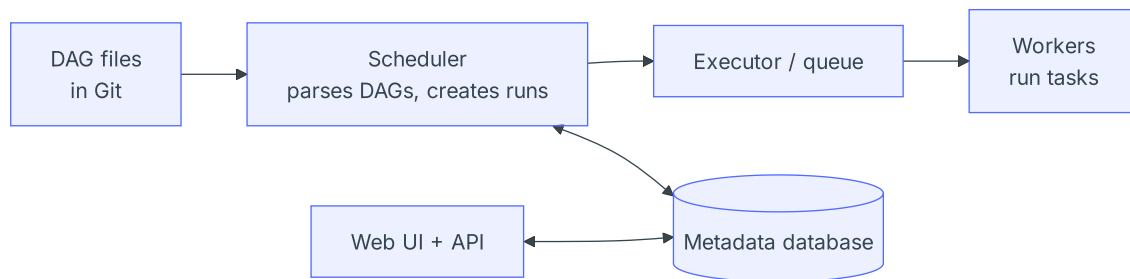
Cron jobs and shell scripts can run these steps, but they do not answer the operational questions: what happens downstream when step 2 fails, how to rerun only the failed step, and where to find the history of past runs and failures.

Solution: Airflow is a workflow orchestrator that understands dependencies. A pipeline is defined in Python as a **DAG** (Directed Acyclic Graph) — tasks connected by dependency edges. Airflow then: - Schedules the DAG to run on a timetable (daily, hourly, every 15 minutes) - Runs tasks in the right order, in parallel where it can - Retries failures automatically - Provides a UI with the status and history of every task in every run - Allows individual tasks to be rerun without repeating the whole pipeline

"DAG" describes the structure: tasks form a graph, dependencies have a direction, and there are no cycles — no task can depend, directly or indirectly, on itself.

When to use Airflow: - Multi-step pipelines where step B depends on step A finishing first
- Daily/hourly batch jobs (ETL, data loads, report generation) - Workflows that need human-readable monitoring, retries, and alerting - Anything more complex than a single cron job

When it is unnecessary: a single, independent script that runs occasionally — a cron job is sufficient.



On this page

Basics - [What is Airflow?](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#what-is-airflow) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#what-is-airflow) - [Core Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#core-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#core-concepts) - [Your First DAG](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#your-first-dag) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#your-first-dag) - [Task Dependencies](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#task-dependencies) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#task-dependencies) - [Common Operators](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#common-operators) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#common-operators)

Intermediate - [Scheduling](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#scheduling) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#scheduling) - [XComs — Passing Data Between Tasks](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#xcoms--passing-data-between-tasks) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#xcoms--passing-data-between-tasks) - [Variables & Connections](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#variables--connections) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#variables--connections) - [Sensors](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#sensors) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#sensors) - [Branching](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#branching) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#branching) - [TaskFlow API](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#taskflow-api) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#taskflow-api)

Advanced - [Dynamic DAGs](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#dynamic-dags) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#dynamic-dags) - [Task Groups](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#task-groups) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#task-groups) - [Hooks](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#hooks) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#hooks) - [Custom Operators](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#custom-operators) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#custom-operators) -

[Backfilling & Catchup](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#backfilling--catchup) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#backfilling--catchup) - [Best Practices](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#best-practices) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#best-practices)

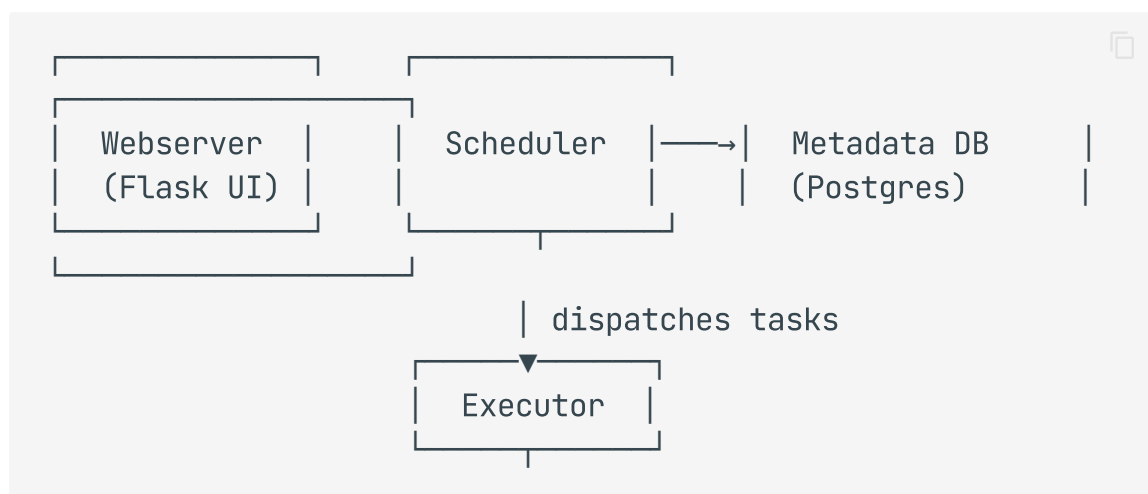
Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/#further-reading)

What is Airflow?

Airflow is a platform to programmatically author, schedule, and monitor workflows. Workflows are defined as **DAGs** (Directed Acyclic Graphs) written in Python — not config files, not drag-and-drop.

Architecture

```
Webserver      - UI at :8080; browse DAGs, view logs, trigger runs
Scheduler      - parses DAG files, schedules task instances, watches for failures
Executor       - actually runs tasks (LocalExecutor, CeleryExecutor, KubernetesExecutor)
Metadata DB    - Postgres/MySQL; stores DAG/task state, XComs, connections, variables
Workers        - processes that execute tasks (CeleryExecutor only)
```





Executors

Executor	Runs tasks	Use when
<code>SequentialExecutor</code>	One at a time, same process	Dev/testing only (removed in Airflow 3)
<code>LocalExecutor</code>	Parallel, same machine	Small to medium workloads (Airflow 3 default)
<code>CeleryExecutor</code>	Distributed across workers	Production, large scale
<code>KubernetesExecutor</code>	Each task in a K8s pod	Cloud-native, isolated dependencies

Airflow 2 vs Airflow 3

Airflow 3.0 (April 2025) is a major release. Examples in this guide use syntax that works on **2.4+ and 3.x**; the differences that matter most:

Area	Airflow 2.x	Airflow 3.x
Schedule argument	<code>schedule_interval=</code> (deprecated from 2.4)	<code>schedule=</code> only
catchup default	<code>True</code>	<code>False</code>
Cron schedules	<code>logical_date</code> = start of the data interval (run happens after it ends)	<code>logical_date</code> = the run time (<code>CronTriggerTimetable</code>); use <code>CronDataIntervalTimetable</code> for the old behavior
DAG authoring imports	<code>airflow.decorators</code> , <code>airflow.operators.*</code>	<code>airflow.sdk (dag, task, DAG)</code> and <code>airflow.providers.standard.*</code> ; old paths are deprecated
Web UI / API	Flask webserver	New React UI served by <code>airflow api-server</code>

Area	Airflow 2.x	Airflow 3.x
Task ↔ metadata DB	Tasks talk to the DB directly	Tasks go through the Task Execution API (no direct DB access)
Backfill	<code>airflow dags backfill</code> (client-side)	<code>airflow backfill create</code> (run by the scheduler, visible in the UI)
Removed	—	SubDAGs, SLAs (replaced by deadline alerts), <code>SequentialExecutor</code> , <code>execution_date</code>
Data-aware scheduling	Datasets	Assets (<code>@asset</code> , <code>schedule=[Asset(...)]</code>)

Core Concepts

Concept	Definition
DAG	A Python file defining a workflow — nodes are tasks, edges are dependencies
Task	A single unit of work in a DAG (runs one operator)
Operator	A template defining what a task does (BashOperator, PythonOperator, etc.)
Task Instance	A specific run of a task for a specific <code>logical_date</code>
DAG Run	One execution of a full DAG for a specific <code>logical_date</code>
Logical date	The data interval start time (formerly <code>execution_date</code>) — not when the job runs
Schedule	A cron expression or timedelta defining how often the DAG runs
Sensor	An operator that waits for a condition before proceeding
Hook	A client for an external system (database, S3, etc.) — used inside operators
XCom	Cross-communication — small values passed between tasks
Connection	Named credentials for external systems stored in the metadata DB
Variable	Key-value config stored in the metadata DB; accessible in DAGs

Concept	Definition
Pool	Limits concurrency for a group of tasks (e.g. max 5 DB tasks at once)

Your First DAG

```

from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator

# Default arguments applied to every task in this DAG
default_args = {
    "owner": "data-engineering",
    "depends_on_past": False,
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
    "email_on_failure": True,
    "email": ["de-alerts@example.com"],
}

# DAG definition
with DAG(
    dag_id="orders_daily_load",
    description="Load and transform daily orders",
    default_args=default_args,
    start_date=datetime(2024, 1, 1),
    schedule="0 2 * * *",          # 2am UTC daily
    catchup=False,                 # don't backfill
    missed runs
    tags=["orders", "daily"],
) as dag:

    def extract_orders(**context):
        logical_date = context["logical_date"]
        date_str = logical_date.strftime("%Y-%m-%d")
        print(f"Extracting orders for {date_str}")
        # ... extraction logic

    def transform_orders(**context):

```

```

        print("Transforming orders...")
        # ... transform logic

extract = PythonOperator(
    task_id="extract_orders",
    python_callable=extract_orders,
)

transform = PythonOperator(
    task_id="transform_orders",
    python_callable=transform_orders,
)

load = BashOperator(
    task_id="load_to_warehouse",
    bash_command="python /opt/pipelines/load_orders.py
--date {{ ds }}",
)

notify = BashOperator(
    task_id="notify_success",
    bash_command='echo "Pipeline completed for {{ ds
}}"',
)

# Dependencies
extract >> transform >> load >> notify

```

Jinja templating in operators

Airflow renders Jinja templates at runtime inside string arguments:

```

# Available template variables
{{ ds }}           # logical date as YYYY-MM-DD string
{{ ds_nodash }}    # logical date as YYYYMMDD
{{ logical_date }} # pendulum datetime object
{{ prev_ds }}      # previous run's logical date
{{ next_ds }}      # next run's logical date
{{ dag.dag_id }}   # DAG ID string
{{ task.task_id }} # task ID string
{{ run_id }}       # unique run identifier
{{ params.my_key }} # DAG/task params dict

```

```
# Example
BashOperator(
    task_id="export",
    bash_command="python export.py --date {{ ds }} --dag {{
dag.dag_id }}",
)
```

Task Dependencies

```
# Sequential
task_a >> task_b >> task_c

# Fan-out – task_a runs first, then b and c run in parallel
task_a >> [task_b, task_c]

# Fan-in – both b and c must complete before d
[task_b, task_c] >> task_d

# Full diamond
task_a >> [task_b, task_c] >> task_d

# Equivalent to >> and <<
task_b.set_upstream(task_a)    # same as task_a >> task_b
task_b.set_downstream(task_c)  # same as task_b >> task_c

# Cross-DAG dependency – trigger from another DAG
from airflow.sensors.external_task import
ExternalTaskSensor

wait_for_upstream = ExternalTaskSensor(
    task_id="wait_for_customers_dag",
    external_dag_id="customers_daily_load",
    external_task_id="load_complete",
    timeout=3600,    # wait up to 1 hour
)
wait_for_upstream >> my_task
```

Task states

State	Meaning
<code>success</code>	Task completed without error
<code>failed</code>	Task raised an exception
<code>running</code>	Currently executing
<code>up_for_retry</code>	Failed, will retry
<code>upstream_failed</code>	A dependency failed — this task skipped
<code>skipped</code>	Skipped by a BranchOperator
<code>queued</code>	Waiting for an executor slot

Common Operators

PythonOperator

```
from airflow.operators.python import PythonOperator

def my_function(param1, param2, **context):
    ds = context["ds"]    # logical date string
    print(f"Running for {ds}, param1={param1}")
    return "done"         # return value stored as XCom
                           # automatically

task = PythonOperator(
    task_id="run_python",
    python_callable=my_function,
    op_kwargs={"param1": "orders", "param2": 42},
)
```

BashOperator

```
from airflow.operators.bash import BashOperator
```

```
task = BashOperator(
    task_id="run_script",
    bash_command="python /opt/pipelines/extract.py --date
    {{ ds }}",
    env={"PYTHONPATH": "/opt/pipelines"},
)
```

SQLExecuteQueryOperator (Airflow 2.4+)

```
from airflow.providers.common.sql.operators.sql import
SQLExecuteQueryOperator

run_query = SQLExecuteQueryOperator(
    task_id="create_daily_summary",
    conn_id="warehouse_default",      # any SQL connection
    defined in Airflow
    sql="""
        INSERT INTO summary.daily_orders
        SELECT DATE('{{ ds }}') AS order_date,
               COUNT(*)        AS orders,
               SUM(amount)      AS revenue
        FROM   staging.orders
        WHERE  order_date = '{{ ds }}'
    """
)
```

S3 / Cloud Storage Operators

```
from airflow.providers.amazon.aws.operators.s3 import
S3CreateObjectOperator
from airflow.providers.amazon.aws.sensors.s3 import
S3KeySensor

# Wait for a file to land in S3
wait_for_file = S3KeySensor(
    task_id="wait_for_orders_file",
    bucket_name="my-data-bucket",
    bucket_key="raw/orders/{{ ds }}/orders.csv",
    aws_conn_id="aws_default",
    poke_interval=60,      # check every 60 seconds
)
```

```
        timeout=3600,  
    )
```

EmptyOperator (placeholder)

```
from airflow.operators.empty import EmptyOperator  
  
start = EmptyOperator(task_id="start")  
end    = EmptyOperator(task_id="end")  
start >> [task_a, task_b] >> end
```

Scheduling

```
# Cron expressions  
schedule="0 2 * * *"           # 2am daily  
schedule="0 * * * *"           # hourly  
schedule="0 2 * * 1"           # 2am every Monday  
schedule="0 2 1 * * *"         # 2am on the 1st of each  
month  
schedule="*/15 * * * *"        # every 15 minutes  
  
# Preset strings  
schedule="@daily"              # midnight daily  
schedule="@hourly"  
schedule="@weekly"  
schedule="@monthly"  
schedule="@once"               # run once only  
  
# timedelta  
from datetime import timedelta  
schedule=timedelta(hours=6)  
  
# Data interval – CRITICAL concept  
# A DAG with schedule "@daily" and start_date=2024-01-01:  
# Run 1: data interval 2024-01-01 → 2024-01-02, RUNS at  
# 2024-01-02 00:00 (after the interval ends)  
# Run 2: data interval 2024-01-02 → 2024-01-03, RUNS at  
# 2024-01-03 00:00  
#
```

```
# Airflow 2 (and interval timetables in 3): logical_date =
start of the data interval,
# so {{ ds }} is "yesterday" relative to when the run
starts.
# Airflow 3 cron schedules default to CronTriggerTimetable:
logical_date = run time.
# Portable choice: read {{ data_interval_start }} / {{
data_interval_end }} explicitly.
```

Catchup

```
with DAG(
    dag_id="my_dag",
    start_date=datetime(2024, 1, 1),
    schedule="@daily",
    catchup=True,    # True = create runs for all missed
                    # False = only run from now forward
    intervals since start_date
):
    ...

# Global default in airflow.cfg:
# catchup_by_default = False ← recommended (the default
in Airflow 3)
```

XComs — Passing Data Between Tasks

XComs (cross-communications) let tasks share small values. Stored in the metadata DB — not suitable for large data (use S3/HDFS for that).

```
# Push: return value is automatically pushed as XCom
def extract(**context):
    records_count = 1500
    return records_count    # stored under key
"return_value"

# Push explicitly
def extract(**context):
    context["ti"].xcom_push(key="record_count", value=1500)
```

```

        context["ti"].xcom_push(key="file_path",
value="s3://bucket/file.parquet")

# Pull in a downstream task
def load(**context):
    ti    = context["ti"]
    count = ti.xcom_pull(task_ids="extract",
key="return_value")
    path  = ti.xcom_pull(task_ids="extract",
key="file_path")
    print(f>Loading {count} records from {path}")

# Pull in Jinja template
BashOperator(
    task_id="notify",
    bash_command='echo "Loaded {{
ti.xcom_pull(task_ids=\'extract\') }} records"',
)

```

XComs are for small metadata, not data. Pass file paths, record counts, timestamps. Never put entire DataFrames in XComs.

Variables & Connections

Variables — key-value config

```

from airflow.models import Variable

# Get a variable (set in Admin > Variables in the UI)
bucket = Variable.get("s3_data_bucket")
config = Variable.get("pipeline_config",
deserialize_json=True) # parses JSON

# With a default (won't raise if missing)
env = Variable.get("environment", default_var="production")

# Set programmatically (usually done in UI or via CLI)
Variable.set("last_run_date", "2024-03-15")

```

Connections — external system credentials

```
from airflow.hooks.base import BaseHook

# Get connection details (set in Admin > Connections in the
UI)
conn = BaseHook.get_connection("warehouse_default")
conn.host, conn.login, conn.password, conn.schema

# Or use the hook directly (preferred)
from airflow.providers.postgres.hooks.postgres import
PostgresHook

hook = PostgresHook(postgres_conn_id="postgres_analytics")
df = hook.get_pandas_df("SELECT * FROM orders WHERE date
= %s", parameters=["2024-03-15"])
records = hook.get_records("SELECT COUNT(*) FROM orders")
hook.run("DELETE FROM staging.orders WHERE date = '2024-03-
15'")
```

Sensors

Sensors wait for a condition to be true before allowing downstream tasks to proceed.

```
from airflow.sensors.filesystem import FileSensor
from airflow.sensors.python import PythonSensor
from airflow.sensors.time_delta import TimeDeltaSensor

# Wait for a file on the local filesystem
wait_for_file = FileSensor(
    task_id="wait_for_orders",
    filepath="/data/raw/orders_{{ ds }}.csv",
    poke_interval=60,      # check every 60 seconds
    timeout=7200,          # fail after 2 hours
    mode="poke",           # "poke" = occupies a worker slot;
    "reschedule" = frees it
)

# Custom condition with PythonSensor
def check_api_ready(**context):
```

```

import requests
resp = requests.get("https://api.example.com/status")
return resp.json().get("status") == "ready"

wait_for_api = PythonSensor(
    task_id="wait_for_api",
    python_callable=check_api_ready,
    poke_interval=30,
    timeout=3600,
    mode="reschedule", # frees worker slot between checks
    - prefer for long waits
)

```

Mode	Behavior	Use when
<code>poke</code>	Occupies a worker slot while waiting	Short wait, few sensors
<code>reschedule</code>	Releases slot between checks	Long wait, many sensors

Branching

Route execution to different tasks based on runtime logic.

```

from airflow.operators.python import BranchPythonOperator
from airflow.operators.empty import EmptyOperator

def choose_branch(**context):
    ds = context["ds"]
    dow = context["logical_date"].day_of_week # 0=Monday,
6=Sunday
    if dow == 0: # Monday
        return "weekly_summary"
    return "daily_summary"

branch = BranchPythonOperator(
    task_id="branch_by_day",
    python_callable=choose_branch,
)

daily_task = EmptyOperator(task_id="daily_summary")

```

```

weekly_task = EmptyOperator(task_id="weekly_summary")

# Both branches merge here – trigger_rule needed because
# one branch is skipped
end = EmptyOperator(
    task_id="end",
    trigger_rule="none_failed_min_one_success",
)

branch >> [daily_task, weekly_task] >> end

```

Trigger rules

Rule	Task runs when...
<code>all_success</code> (default)	All upstream tasks succeeded
<code>all_failed</code>	All upstream tasks failed
<code>all_done</code>	All upstream tasks are done (any state)
<code>one_success</code>	At least one upstream succeeded
<code>one_failed</code>	At least one upstream failed
<code>none_failed</code>	No upstream task failed (skipped is OK)
<code>none_failed_min_one_success</code>	None failed AND at least one succeeded

TaskFlow API

Modern Airflow 2.x API — decorate Python functions, dependencies inferred from function calls. Much cleaner than classic operators.

```

from airflow.decorators import dag, task
from datetime import datetime

@dag(
    dag_id="orders_pipeline_taskflow",
    schedule="@daily",

```

```

        start_date=datetime(2024, 1, 1),
        catchup=False,
    )
    def orders_pipeline():

        @task
        def extract(ds=None):
            print(f"Extracting for {ds}")
            return {"record_count": 1500, "path":
f"s3://bucket/raw/{ds}/orders.parquet"}

        @task
        def transform(extracted: dict):
            count = extracted["record_count"]
            path = extracted["path"]
            print(f"Transforming {count} records from {path}")
            return {"cleaned_path":
f"s3://bucket/clean/{path.split('/')[1]}"}

        @task
        def load(transformed: dict):
            print(f"Loading from
{transformed['cleaned_path']}")
            return "success"

        @task
        def notify(status: str):
            print(f"Pipeline status: {status}")

        # Dependencies are inferred from function call chain –
no >> needed
        extracted = extract()
        transformed = transform(extracted)
        status = load(transformed)
        notify(status)

    # Instantiate the DAG
    orders_pipeline()

```

TaskFlow is the recommended approach for new DAGs. It automatically handles XCom push/pull, making data flow between tasks explicit and type-safe.

Dynamic DAGs

Generate tasks programmatically based on a list or config.

```
from airflow.decorators import dag, task
from datetime import datetime

TABLES = ["orders", "customers", "products", "inventory"]

@dag(schedule="@daily", start_date=datetime(2024, 1, 1),
catchup=False)
def dynamic_table_load():

    @task
    def extract(table: str, ds=None):
        print(f"Extracting {table} for {ds}")
        return table

    @task
    def load(table: str):
        print(f"Loading {table} to warehouse")

    @task
    def reconcile(tables: list):
        print(f"All tables loaded: {tables}")

    # Generate one extract+load pair per table
    loaded = []
    for table in TABLES:
        extracted =
extract.override(task_id=f"extract_{table}")(table)

loaded.append(load.override(task_id=f"load_{table}")
(extracted))

    reconcile(loaded)

dynamic_table_load()
```

Task Groups

Visually group related tasks in the UI without creating a SubDAG.

```
from airflow.utils.task_group import TaskGroup
from airflow.operators.python import PythonOperator

with DAG("grouped_pipeline", ...) as dag:

    with TaskGroup("extract", tooltip="Extract from sources") as extract_group:
        extract_orders =
        PythonOperator(task_id="orders", python_callable=...)
        extract_customers =
        PythonOperator(task_id="customers", python_callable=...)

    with TaskGroup("transform") as transform_group:
        transform_orders =
        PythonOperator(task_id="orders", python_callable=...)
        transform_customers =
        PythonOperator(task_id="customers", python_callable=...)

    with TaskGroup("load") as load_group:
        load_orders = PythonOperator(task_id="orders",
python_callable=...)
        load_customers =
        PythonOperator(task_id="customers", python_callable=...)

    extract_group >> transform_group >> load_group
```

Hooks

Hooks are the low-level clients for external systems. Operators use hooks internally. Use them directly when you need more control.

```
# PostgreSQL
from airflow.providers.postgres.hooks.postgres import
PostgresHook
pg = PostgresHook(postgres_conn_id="postgres_analytics")
pg.run("INSERT INTO log VALUES (%s, %s)", parameters=
```

```

["pipeline", "started"])
df = pg.get_pandas_df("SELECT * FROM orders WHERE date =
'2024-03-15'")

# S3
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
s3 = S3Hook(aws_conn_id="aws_default")
s3.load_file("/local/path/file.csv", "s3-key/file.csv",
bucket_name="my-bucket")
keys = s3.list_keys(bucket_name="my-bucket",
prefix="raw/orders/")

# Any SQL database or warehouse – resolve the right hook
from the connection type
from airflow.hooks.base import BaseHook
db =
BaseHook.get_connection("warehouse_default").get_hook() #
e.g. Snowflake, BigQuery, Redshift hook
db.run("CALL refresh_daily_aggregates()")
df = db.get_pandas_df("SELECT * FROM orders LIMIT 1000")

```

Custom Operators

Build your own operator when you have logic you'll reuse across many DAGs.

```

from airflow.hooks.base import BaseHook
from airflow.models.baseoperator import BaseOperator

class PartitionReloadOperator(BaseOperator):
    """Idempotently reload one date partition of a table
    from a staging table.

    Works with any SQL connection (Postgres, MySQL,
    Snowflake, BigQuery, Redshift, ...):
    get_hook() returns the provider-specific hook for the
    connection type.
    """

    # template_fields: Jinja renders these attributes
    before execute()

```

```

    template_fields = ("target_table", "staging_table",
"partition_value")

    def __init__(
        self,
        target_table: str,
        staging_table: str,
        partition_column: str,
        partition_value: str,
        conn_id: str = "warehouse_default",
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.target_table = target_table
        self.staging_table = staging_table
        self.partition_column = partition_column
        self.partition_value = partition_value
        self.conn_id = conn_id

    def execute(self, context):
        hook =
BaseHook.get_connection(self.conn_id).get_hook()
        statements = [
            f"DELETE FROM {self.target_table} WHERE
{self.partition_column} = '{self.partition_value}'",
            f"INSERT INTO {self.target_table} SELECT * FROM
{self.staging_table} "
            f"WHERE {self.partition_column} =
'{self.partition_value}'",
        ]
        self.log.info("Reloading %s for %s",
self.target_table, self.partition_value)
        hook.run(statements, autocommit=False) # both
statements in one transaction
        return f"Reloaded {self.target_table} for
{self.partition_value}"

# Use it in a DAG
reload_orders = PartitionReloadOperator(
    task_id="reload_orders",
    target_table="analytics.orders",
    staging_table="staging.orders",

```

```
    partition_column="order_date",
    partition_value="{{ ds }}",
)
```

Backfilling & Catchup

Catchup

When `catchup=True` and the `start_date` is in the past, Airflow creates runs for every missed interval between `start_date` and now.

```
# catchup=True with start_date 30 days ago and @daily
# schedule
# → creates 30 DAG runs immediately on first activation
# → order of execution is not guaranteed unless
# max_active_runs=1

with DAG(
    ...,
    catchup=True,
    max_active_runs=1,          # run intervals sequentially,
    not in parallel
    max_active_tasks=3,        # max tasks running at once
    within a DAG run
):
    ...
```

Manual backfill via CLI

```
# Airflow 3 – the scheduler runs the backfill; progress is
# visible in the UI
airflow backfill create \
    --dag-id orders_daily_load \
    --from-date 2024-01-01 \
    --to-date 2024-01-31 \
    --max-active-runs 2

# Airflow 3 – preview which runs would be created
airflow backfill create --dag-id orders_daily_load \
```

```

--from-date 2024-01-01 --to-date 2024-01-31 --dry-run

# Airflow 2
airflow dags backfill \
  --dag-id orders_daily_load \
  --start-date 2024-01-01 \
  --end-date 2024-01-31

```

Every task must be **idempotent** for backfilling to be safe. Running a task twice for the same date should produce the same result — not double the data.

Best Practices

DAG design

```

# Recommended: Set catchup=False unless you explicitly need
backfill
with DAG(..., catchup=False):
    ...

# Recommended: Use start_date in the past (a fixed date,
not datetime.now())
start_date=datetime(2024, 1, 1)    # good
start_date=datetime.now()          # bad — changes every
time DAG is parsed

# Recommended: Keep DAG files lightweight — no heavy
imports at module level
# Heavy imports inside callables, not at the top of the DAG
file
def extract(**context):
    import pandas as pd            # import here, not at top of
DAG file
    ...

# Recommended: Use default_args for shared task config
default_args = {
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
    "on_failure_callback": slack_alert,

```

```

}

# Recommended: Name task_ids clearly – they appear in logs
and UI
# Bad: task_id="task1"
# Good: task_id="extract_orders_from_postgres"

# Recommended: Keep tasks atomic – one task does one thing
# Avoid: one giant Python function that extracts,
transforms, and loads
# Prefer: separate extract, transform, load tasks

```

Performance

```

# Don't store data in XComs – pass file paths instead
@task
def extract(ds=None):
    path = f"s3://bucket/raw/{ds}/orders.parquet"
    # write data to S3
    return path    # XCom only holds the path, not the data

# Use pools to limit concurrency on shared resources
from airflow.models import Pool
# In Admin > Pools: create "warehouse_pool" with 10 slots

heavy_query = SQLExecuteQueryOperator(    # generic SQL
operator for any database connection
    task_id="heavy_query",
    conn_id="warehouse_default",
    pool="warehouse_pool",    # max 10 concurrent warehouse
queries
    pool_slots=2,            # this task uses 2 slots
    sql="CALL refresh_daily_aggregates()",
)

# Set reasonable timeouts
PythonOperator(
    task_id="extract",
    python_callable=extract,
    execution_timeout=timedelta(hours=1),
)

```

Alerts

```
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator

def slack_alert(context):
    dag_id = context["dag"].dag_id
    task_id = context["task"].task_id
    ds = context["ds"]
    msg = f":red_circle: *{dag_id}.{task_id}* failed for `{ds}`"

    SlackWebhookOperator(
        task_id="slack_fail",
        slack_webhook_conn_id="slack_alerts",
        message=msg,
    ).execute(context)

default_args = {
    "on_failure_callback": slack_alert,
}
```

Common Pitfalls

Pitfall	Symptom	Fix
Heavy work at the top level of a DAG file (DB queries, API calls, big imports)	Slow scheduler, DAG import timeouts, a query every 30 seconds on every parse	Top-level code only <i>defines</i> the DAG; do work inside tasks
<code>start_date=datetime.now()</code>	DAG never runs, or runs unpredictably	A fixed date in the past
Using <code>datetime.now()</code> inside tasks instead of the run's interval	Backfills and reruns process the wrong day	Use <code>data_interval_start</code> / <code>data_interval_end</code> (or <code>{{ ds }}</code>) from the context
Non-idempotent tasks (plain <code>INSERT</code>)	Retries and backfills create duplicates	Overwrite the partition, or <code>DELETE + INSERT</code> / <code>MERGE</code> for that interval

Pitfall	Symptom	Fix
Passing data through XCom	Metadata DB bloats; tasks slow down	Write data to S3 or a table; pass only the path or key
Airflow workers doing the heavy compute	Workers run out of memory; one pandas job starves the rest	Push work down to Spark, the warehouse, dbt, or a Kubernetes pod; Airflow orchestrates
Many long-waiting sensors in <code>poke</code> mode	Worker slots all taken by sensors doing nothing	<code>mode="reschedule"</code> or deferrable operators (triggerer)
<code>catchup=True</code> by accident with an old <code>start_date</code>	Hundreds of runs appear the moment the DAG is unpaused	<code>catchup=False</code> unless you really want the history; <code>max_active_runs</code> to throttle
No <code>execution_timeout</code>	A hung task blocks its pool slot forever	Set <code>execution_timeout</code> on every task (via <code>default_args</code>)
Secrets in Variables, DAG code, or Git	Leaked credentials	Connections with a secrets backend (Vault, AWS Secrets Manager, GCP Secret Manager)
One giant <code>PythonOperator</code> doing extract, transform, and load	Can't retry just the failed step; no visibility	Split into atomic tasks that can each be retried
Generating thousands of tasks with Python loops	Slow parsing; unreadable graph	Dynamic task mapping (<code>.expand()</code>) decided at runtime
Relying on implicit timezones	Runs at the wrong local time around DST changes	Timezone-aware <code>start_date</code> (<code>pendulum.datetime(..., tz="Europe/London")</code>)

Cheat Sheet

DAG skeleton (TaskFlow, works on 2.4+ and 3.x)

```
import pendulum
from datetime import timedelta
from airflow.decorators import dag, task           # Airflow
3: from airflow.sdk import dag, task
```

```

@dag(
    schedule="0 2 * * *",
    start_date=pendulum.datetime(2024, 1, 1, tz="UTC"),
    catchup=False,
    max_active_runs=1,
    default_args={"retries": 2, "retry_delay":
timedelta(minutes=5),
                  "execution_timeout": timedelta(hours=1)},
    tags=["orders"],
)
def orders_daily():
    @task
    def extract(data_interval_start=None,
data_interval_end=None) -> str:
        return
f"s3://bucket/raw/orders/{data_interval_start:%Y-%m-
%d}.parquet"

    @task
    def load(path: str) -> None:
        ...

    load(extract())

orders_daily()

```

Task	How
Map over a runtime list	<code>process.expand(table=get_tables())</code> · fixed args: <code>.partial(conn_id="x").expand(...)</code>
Run after another DAG's data	Producer task <code>outlets=[Asset("s3://.../orders")]</code> → consumer <code>schedule=[Asset("s3://.../orders")]</code>
Wait without holding a slot	<code>mode="reschedule"</code> or <code>deferrable=True</code> operators
Limit concurrency on a resource	<code>pool="warehouse_pool", pool_slots=2</code>
Join after a branch	<code>trigger_rule="none_failed_min_one_success"</code>

Task	How
Always run a cleanup task	<code>trigger_rule="all_done"</code>
Template variables	<code>{{ ds }}</code> · <code>{{ data_interval_start }}</code> · <code>{{ data_interval_end }}</code> · <code>{{ run_id }}</code> · <code>{{ params.x }}</code>
Test one task locally	<code>airflow tasks test <dag_id> <task_id> 2024-03-15</code>
Test a whole DAG in-process	<code>dag.test()</code> (in a <code>__main__</code> block)
Find import errors	<code>airflow dags list-import-errors</code>
Trigger with config	<code>airflow dags trigger <dag_id> --conf '{"table": "orders"}'</code>
Backfill	Airflow 3: <code>airflow backfill create --dag-id d --from-date ... --to-date ...</code> · Airflow 2: <code>airflow dags backfill -s ... -e ... d</code>
Clear failed tasks to rerun	<code>airflow tasks clear <dag_id> -s <start> -e <end> --only-failed</code>

Executor picker: local/dev → `LocalExecutor` · many workers, steady load → `CeleryExecutor` · per-task isolation and dependencies → `KubernetesExecutor` · managed → MWAA, Cloud Composer, Astronomer

Interview Questions

Q: What is the difference between a DAG's `schedule` and its `start_date`? A: `start_date` is when the DAG becomes eligible to run — Airflow won't schedule runs before this date. `schedule` (called `schedule_interval` before Airflow 2.4) defines the frequency (e.g., `"0 2 * * *"` = daily at 2am). With interval-based timetables (the Airflow 2 default), a daily DAG with `start_date=2024-01-01` first runs at `2024-01-02 00:00` to process the data interval that *starts* on `2024-01-01` — the run happens after the interval it represents, which trips up beginners. Airflow 3 changed cron schedules to trigger-based timetables where `logical_date` is the run time, so it's safest to read `data_interval_start` / `data_interval_end` explicitly.

Q: What is `catchup` and when would you set it to `False`? A: When `catchup=True` (the default in Airflow 2; Airflow 3 defaults to `False`), if your DAG was paused for 30 days and you re-enable it, Airflow will schedule 30 backfill runs to cover the missed intervals. Set

`catchup=False` when you only want the next upcoming run, not historical backfill. For event-driven or near-real-time pipelines where historical reruns don't make sense (e.g., "send daily email"), always set `catchup=False` to avoid an avalanche of runs on startup.

Q: What are XComs and what's the limitation you need to know? A: XComs (cross-communications) let tasks share small values: one task pushes a value, another pulls it with `ti.xcom_pull(task_ids="upstream_task")`. In the TaskFlow API, return values are automatically pushed as XComs. The critical limitation: XComs are stored in the Airflow metadata database (Postgres/MySQL). They're for small values like IDs, row counts, or status strings — not DataFrames or large payloads. Storing a 1GB file path is fine; storing the file contents will bloat your metadata DB and cause performance issues.

Q: What is the difference between `depends_on_past` and `wait_for_downstream`? A: `depends_on_past=True` means a task won't start its run for date D+1 until the same task's run for date D succeeded. Useful for incremental loads where each day builds on the previous. `wait_for_downstream=True` goes further: it waits until the entire downstream pipeline from the previous run has finished before starting. Use `depends_on_past` for sequential processing; use `wait_for_downstream` when you can't start the next batch until the previous batch's consumers have fully finished.

Q: How would you pass a file path between tasks — what's the right pattern? A: Don't pass the file contents through XComs — push the path or identifier instead. Task A downloads a file to S3 and pushes the S3 URI (`s3://bucket/path/file.parquet`) as an XCom. Task B pulls that URI and reads the file directly. This keeps XComs small and your tasks decoupled. For structured handoffs, consider writing the result to an intermediate table and passing only the table name or run ID downstream.

Q: What's the difference between `LocalExecutor`, `CeleryExecutor`, and `KubernetesExecutor`? A: `LocalExecutor` runs tasks as subprocesses on the same machine as the scheduler — simple, no extra infrastructure, good for small deployments. `CeleryExecutor` distributes tasks to a pool of separate worker machines via a message broker (Redis/RabbitMQ) — scalable, but requires maintaining workers and the broker. `KubernetesExecutor` launches each task instance in its own Kubernetes pod — best for cloud-native deployments, perfect isolation, no idle workers (pods spin up/down per task), but has pod startup overhead (~30s) that makes it poor for fast, short tasks.

Further Reading

- [Apache Airflow documentation](https://airflow.apache.org/docs/apache-airflow/stable/index.html) (https://airflow.apache.org/docs/apache-airflow/stable/index.html)
- [Upgrading to Airflow 3](https://airflow.apache.org/docs/apache-airflow/stable/installation/upgrading_to_airflow3.html) (https://airflow.apache.org/docs/apache-airflow/stable/installation/upgrading_to_airflow3.html)
- [Airflow best practices](https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html) (https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html) — top-level code, idempotency, testing

- [Dynamic task mapping](https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/dynamic-task-mapping.html) (https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/dynamic-task-mapping.html)
 - [Astronomer guides](https://www.astronomer.io/docs/learn) (https://www.astronomer.io/docs/learn) — practical, well-maintained tutorials
 - [Cosmos](https://astronomer.github.io/astronomer-cosmos/) (https://astronomer.github.io/astronomer-cosmos/) — run dbt projects as Airflow task groups
 - *Data Pipelines with Apache Airflow* — Bas Harenslak & Julian de Ruiter (Manning)
-

Previous: [DataOps](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/dataops-operations/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/dataops-operations/) • **Next:** [Dagster](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/dagster-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/dagster-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026