Big Data Lab

Lab 6: Data Orchestration with Apache Airflow

Dr. Innocent Nyalala

IIT Madras Zanzibar
Department of Data Science and AI

Monday, 27 April 2026



Session Plan & Learning Objectives

Theory (60 minutes)

- Airflow **purpose**: orchestration vs compute
- Core model: **DAG** → **tasks** → **runs**
- Scheduling: `start_date`, `schedule_interval`, `catchup`, `ds`
- Building blocks: operators/sensors + Connections/Variables + logs

Practical (60 minutes)

- Verify services + Airflow UI
- Create Connections (MinIO + Spark)
- Build a minimal DAG, then add dependencies + retries
- Add a sensor + SparkSubmitOperator; debug in the UI

Theory: mental model + vocabulary → Practical: build & run a DAG



Today's Map

1. Why orchestration exists
2. Airflow basics: DAGs, tasks, scheduling, and architecture
3. How to configure Airflow (Connections/Variables)
4. Step-by-step lab: build and run a DAG, then integrate Spark



Key Definitions: Airflow, Workflow, Orchestration

What is Apache Airflow?
Airflow is an **open-source workflow orchestrator** used to **author, schedule, and monitor** pipelines as code (**DAGs**).

What is a workflow?
A workflow is a **sequence of tasks** (steps) that transform data or run jobs, often with **dependencies** (e,g A must finish before B).

What is orchestration?
Orchestration is the **coordination of the workflow execution**: scheduling, dependency handling, retries, backfills, logging, and alerting (not the compute itself).



What is a Cron Job (and Why It Fails for Pipelines)?

What is a cron job?
A cron job is a **time-based scheduled command** run by the Unix **cron** service (e.g., run a script every day at 01:00).

Why cron breaks down in real data pipelines:

- **No dependency model:** it schedules time, not *"run B after A succeeds"*
- **Weak failure handling:** retries, timeouts, and "resume where you left off" are manual
- **No backfills/catchup:** rerunning historical dates is painful and error-prone
- **Low observability:** limited UI, metadata, lineage, and centralized logs
- **Scaling is ad-hoc:** parallelism, resource limits, and concurrency control are not built-in

Cron is great for small, independent tasks. Orchestrators exist for multi-step, dependency-heavy workflows.



Why Do We Need Workflow Orchestration?

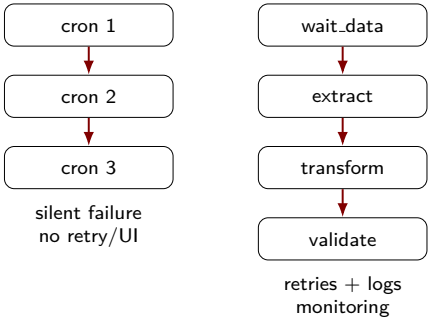
Production pipelines must:

- Run on a schedule
- Respect dependencies
- Retry on transient failures
- Support backfills (historical dates)
- Be observable: UI + logs + alerts

Example contexts (East Africa):

- Mobile-money ETL (M-Pesa): nightly aggregation
- Disease surveillance dashboards: daily ingestion + QA

Cron vs orchestration (picture)



Apache Airflow — Overview

- Open-source platform to **author, schedule, and monitor** workflows
- Workflows are defined as **Python code/script** (not YAML, not XML)
- Originally created at **Airbnb** (2014), donated to Apache (2016)
- Current stable release: **Airflow 2.x** (we use 2.8+)
- Workflows are represented as **DAGs** — Directed Acyclic Graphs

Key design principles

- **Dynamic:** pipelines can be generated programmatically in Python
- **Extensible:** custom operators, hooks, and plugins
- **Elegant:** Jinja templating for parameterised tasks
- **Scalable:** modular architecture (Local/Celery/Kubernetes executors)



Airflow Orchestrator (Not a Compute Engine)

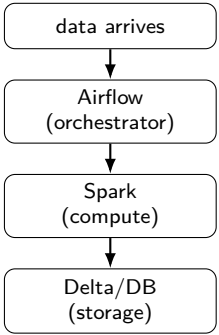
One sentence: Airflow **coordinates** your pipeline steps.

What it does

- runs tasks in the right **order**
- triggers tasks on a **schedule**
- retries on failure; keeps **logs**
- shows everything in a **UI**

What it does not do

- It does not crunch big data itself (Spark does)
- It does not store your tables (Delta/DB does)



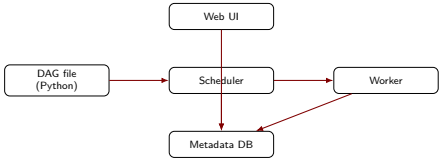
How Airflow Works (One DAG Run)

Two clocks (important)

- **Logical date** ds: which data interval you process
- **Wall-clock time**: when the scheduler actually runs it

Core objects

- Scheduler creates a **DAG run**
- A DAG run contains **task instances**
- States are stored in the **metadata DB**



Configuring Airflow: 3 Things You Must Get Right

1) Connections (credentials/hosts)

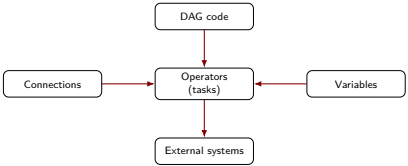
- Spark master, MinIO/S3, databases, APIs
- Operators reference them by `conn_id`

2) Variables (small runtime config)

- bucket names, feature flags, API keys (small values)

3) DAG parameters (scheduling behaviour)

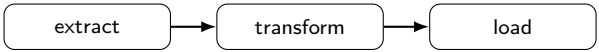
- `start_date`, `schedule_interval`, `catchup`, retries



Lab checklist: `http://localhost:8085` (admin/admin) create `minio_s3` and `spark_default`.

Airflow Glossary: See the Objects (Not Just Definitions)

DAG → tasks → one run



DAG run for `ds`
creates task instances

Quick mapping

- **DAG:** pipeline definition (graph)
- **Task:** a node (operator instance)
- **Task instance:** one task for one `ds`

Building blocks (what you code with)

- **Operator:** does work (Bash/Python/Spark)
- **Sensor:** waits (file/partition/API)
- **Hook:** connects (S3/DB/HTTP)

UI-managed config

- **Connection** (`conn_id`)
- **Variable**
- **XCom** (small messages)

Template Variables You'll Actually Use

Most common

- {{ ds }} = YYYY-MM-DD
- {{ ds_nodash }} = YYYYMMDD
- {{ params.x }} = custom run parameters

Mini example

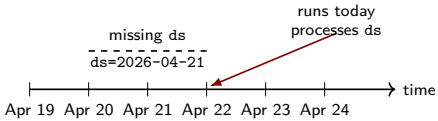
```
echo "processing date={{ ds }}"
aws s3 ls s3://warehouse/raw/date={{ ds
    }}/
```



Scheduling: Logical Dates (ds) vs Real Time

Key idea

- ds labels the **data interval** you're processing.
- Airflow can run later, but it still processes that interval.

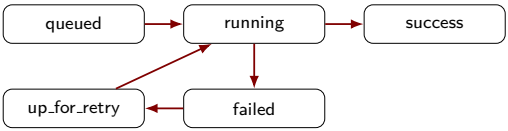


Defaults for this course

- catchup=False
- choose a start_date slightly in the past
- max_active_runs=1 while learning

Task States + Retries: Read the UI Like a Debugger

State flow (typical)



Other states

- **skipped** (branching)
- **upstream_failed** (dependency failed)

Retry intuition

- Transient (network/timeout) ⇒ retry is useful
- Deterministic (bug) ⇒ fix code, then **clear** and re-run

Fast debugging

1. open task instance → **Log**
2. read last 30 lines
3. classify: config vs code vs data

Air.png

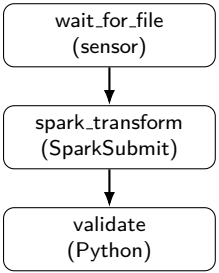


Worked Example: A DAG You Can Explain in 60 Seconds

Goal: daily pipeline

1. wait for a file
2. run a Spark job
3. validate output

Key teaching point: Airflow coordinates; Spark computes.



Worked Example (1/2): Define the DAG + Defaults

```python
from airflow import DAG
from datetime import datetime, timedelta

default_args = {
  "retries": 2,
  "retry_delay": timedelta(minutes=5),
}

with DAG(
  dag_id="lab06_min_etl",
  start_date=datetime(2026, 4, 20),
  schedule_interval="0 2 * * *",
  catchup=False,
  default_args=default_args,
  tags=["lab06"],
) as dag:
  pass
```



Worked Example (2/2): Add Tasks + Dependencies

```python
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from airflow.operators.python import PythonOperator

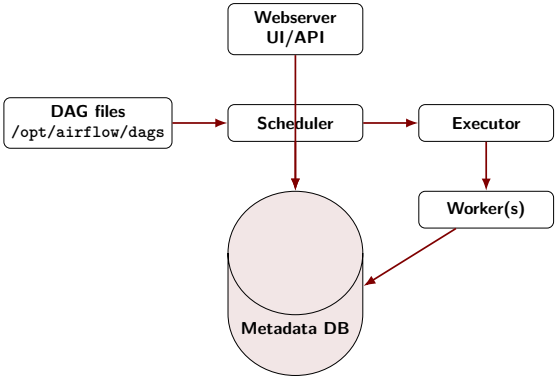
wait = S3KeySensor(
  task_id="wait_for_file",
  bucket_name="warehouse",
  bucket_key="raw/transactions/{{ ds }}/data.csv",
  aws_conn_id="minio_s3",
  mode="reschedule",
)

spark_job = SparkSubmitOperator(
  task_id="spark_transform",
  application="/opt/airflow/spark_jobs/transform_transactions.py",
  conn_id="spark_default",
  application_args=["--date", "{{ ds }}"],
)

def validate(**context):
  print("validate output for", context["ds"])  # replace with real checks
```



Airflow Architecture — Components (Picture + Roles)



What each component does

- **Webserver:** UI + REST API (trigger runs, view logs)
- **Scheduler:** decides what **should** run, and when
- **Executor:** chooses how to run tasks (Local/Celery/K8s)
- **Workers:** execute your task code (Python, bash, Spark submit)
- **Metadata DB:** stores DAG runs, task states, Variables, XCom

Executors — LocalExecutor vs CeleryExecutor

LocalExecutor (our lab)

- Runs tasks in **subprocesses** on the scheduler machine
- Simple: one machine, no extra queue
- Good for: development and small workloads
- Parallelism is limited by available CPU cores

KubernetesExecutor

- Each task runs in its own **Pod**
- Strong isolation; auto-scales; good for multi-tenant clusters

CeleryExecutor (production)

- Tasks are queued via a broker (Redis / RabbitMQ)
- Workers run on **separate machines**
- Horizontally scalable
- Requires: Celery + broker (optionally Flower UI)

Lab setting

- `AIRFLOW__CORE__EXECUTOR=LocalExecut`
- Airflow UI: `http://localhost:8085`
- Login: `admin / admin`



Defining a DAG — Classic Style

```python
from datetime import datetime, timedelta
from airflow import DAG

default_args = {
    "owner": "innocent",
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
    "email_on_failure": True,
    "email": ["innocent@iitmz.ac.in"],
    "sla": timedelta(hours=2),
}

with DAG(
    dag_id="east_africa_etl",
    default_args=default_args,
    description="Daily ETL for mobile money data",
    schedule_interval="0 2 * * *",   # 02:00 EAT every day
    start_date=datetime(2026, 4, 1),
    catchup=False,
    max_active_runs=1,
    tags=["etl", "finance", "east-africa"],
) as dag:
```



Key DAG Parameters

Parameter	Type	Meaning
`dag_id`	str	Unique identifier in UI
`schedule_interval`	str/cron	When the DAG runs
`start_date`	datetime	Earliest logical date
`catchup`	bool	Re-run missed intervals?
`max_active_runs`	int	Concurrent DAG runs
`tags`	list	UI grouping labels
`default_args`	dict	Defaults applied to all tasks
`params`	dict	Jinja-accessible run params

Cron quick reference

- `0 2 * * *` = daily at 02:00
- `0 * * * *` = hourly
- `@daily` = midnight daily
- `0 8 * * 1` = every Monday at 08:00
- `None` = manual trigger only



Catchup & Backfill

Catchup (`catchup=True`):

- Airflow automatically schedules all *missed* DAG runs between `start_date` and now
- Dangerous with large historical windows
- **Always set `catchup=False`** unless you intentionally want it

Backfill (CLI command):

```
airflow dags backfill -s 2026-03-01 -e
2026-03-31 east_africa_etl
```

- Fills a specific date range on demand
- Tasks must be **idempotent** for safe backfilling

Idempotency rule

- Running a task **N times** for the same logical date must produce the **same result** as running it once.
- Use `INSERT OVERWRITE`, not `INSERT`
- Partition output by `ds` (execution date)
- Prefer atomic writes: write to temp, then rename



PythonOperator

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

def extract_transactions(**context):
    execution_date = context["ds"]          # "2026-04-20"
    print(f"Extracting data for {execution_date}")
    # ... business logic ...
    return {"rows": 15423}                  # pushed to XCom automatically

extract = PythonOperator(
    task_id="extract_transactions",
    python_callable=extract_transactions,
    provide_context=True,                   # Airflow 1.x style; default in 2.x
    op_kwargs={"bucket": "warehouse"},      # extra keyword args
    dag=dag,
)
```

- `python_callable`: any Python function
- `op_kwargs`: passed as keyword arguments to the callable
- Return value is automatically pushed to XCom under key `"return_value"`



BashOperator

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

run_validation = BashOperator(
    task_id="validate_schema",
    bash_command="""
        echo "Running schema check for {{ ds }}"
        python /opt/airflow/scripts/validate.py \
            --date {{ ds }} \
            --bucket {{ params.bucket }}
    """,
    params={"bucket": "warehouse"},
    env={"PYTHONPATH": "/opt/airflow"},
    dag=dag,
)
```

- `bash_command`: a Jinja-templated shell string
- `{{ ds }}`: execution date string `YYYY-MM-DD`
- `env`: additional environment variables for the subprocess
- Trailing space after script path is **required** when using `bash_command` with a script file



SparkSubmitOperator

```python
from airflow.providers.apache.spark.operators.spark_submit \
    import SparkSubmitOperator

run_spark = SparkSubmitOperator(
    task_id="aggregate_transactions",
    application="/opt/airflow/spark_jobs/agg_transactions.py",
    conn_id="spark_default",                # configured in Airflow UI
    application_args=["--date", "{{ ds }}"],
    conf={
        "spark.hadoop.fs.s3a.endpoint":         "http://minio:9000",
        "spark.hadoop.fs.s3a.access.key":       "minioadmin",
        "spark.hadoop.fs.s3a.secret.key":       "minioadmin",
        "spark.hadoop.fs.s3a.path.style.access": "true",
    },
    num_executors=2,
    executor_memory="1g",
    driver_memory="512m",
    dag=dag,
)
```

BranchPythonOperator — Conditional Logic

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

def choose_path(**context):
    ti = context["ti"]
    result = ti.xcom_pull(task_ids="extract_transactions")
    if result["rows"] > 10000:
        return "heavy_processing"
    return "light_processing"

branch = BranchPythonOperator(
    task_id="route_by_volume",
    python_callable=choose_path,
    dag=dag,
)

heavy = DummyOperator(task_id="heavy_processing", dag=dag)
light = DummyOperator(task_id="light_processing", dag=dag)
join  = DummyOperator(task_id="join", trigger_rule="none_failed_min_one_success", dag=dag)

branch >> [heavy, light]
heavy  >> join
```



S3KeySensor — Wait for Data Arrival

```python
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor

wait_for_file = S3KeySensor(
    task_id="wait_for_upload",
    bucket_name="warehouse",
    bucket_key="raw/transactions/{{ ds }}/data.csv",
    aws_conn_id="minio_s3",              # configured in Airflow Connections
    poke_interval=30,                    # check every 30 seconds
    timeout=60 * 30,                     # fail after 30 minutes
    mode="poke",                         # or "reschedule" to free worker slot
    dag=dag,
)
```

Sensor modes

- **poke:** worker slot is occupied while waiting (simple)
- **reschedule:** releases the slot between checks (better for long waits)
- **deferrable:** async; requires Triggerer component (Airflow 2.2+)



TaskFlow API — @dag and @task Decorators

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

@dag(
    dag_id="mobile_money_taskflow",
    schedule_interval="@daily",
    start_date=datetime(2026, 4, 1),
    catchup=False,
    tags=["taskflow", "finance"],
)
def mobile_money_pipeline():

    @task
    def extract() -> dict:
        return {"rows": 9801, "source": "safaricom"}

    @task
    def transform(raw: dict) -> dict:
        raw["processed"] = True
        return raw

    @task
```



TaskFlow API vs Classic Operators

Aspect	Classic	TaskFlow
XCom passing	`xcom_push / xcom_pull`	return values (automatic)
Dependency syntax	`t1 >> t2` explicit	inferred from function calls
Code style	Class instantiation	function decorators
External operators	Full support	Wrap in `@task` or mix
Type hints	Not used	Can use; XCom-typed (2.5+)
Readability	More verbose	Pythonic, concise

Recommendation

- Use **TaskFlow** for pure Python tasks.
- Use **classic operators** (SparkSubmitOperator, S3KeySensor, etc.) for external system integrations.
- You can **mix both** styles in a single DAG.



XComs — Cross-Task Communication

```python
# --- PUSH (inside a Python task) ---
def extract(**context):
    ti = context["ti"]  # task instance handle (gives access to XCom)
    ti.xcom_push(key="row_count", value=15423)  # store a small value in metadata DB
    return {"status": "ok"}  # return value is also stored as XCom key="return_value"

# --- PULL (inside another task) ---
def transform(**context):
    ti = context["ti"]
    count = ti.xcom_pull(task_ids="extract_transactions", key="row_count")  # by key
    rv    = ti.xcom_pull(task_ids="extract_transactions")  # default key="return_value"
    print(f"Processing {count} rows, status={rv['status']}")
```

XCom limitations

- XComs are stored in the **metadata database**.
- Keep values **small**: strings, dicts, counts.
- Never push DataFrames or large files; pass **paths** (MinIO/S3) instead.



Airflow Variables & Connections (Concept + Code)

Variables (key/value config in the metadata DB)

```
from airflow.models import Variable

# Read runtime config from the Airflow metadata
    DB (UI-managed)
bucket = Variable.get("warehouse_bucket",
    default_var="warehouse")
api_key = Variable.get("api_key")  # secrets
    belong here (not in DAG code)
```

Connections (credentials/hosts in the metadata DB)

```
from airflow.hooks.base import BaseHook

# Fetch credentials/host by conn_id
    (configured in Admin -> Connections)
conn =
    BaseHook.get_connection("minio_s3")
print(conn.host, conn.login)  #
    host/user (avoid printing passwords
    in real DAGs)
```

Rule: do not hard-code secrets in DAG files.



Configure Connections in the UI (Lab Setup)

Go to: `Admin` → `Connections`

minio_s3 (Amazon S3)

- Host: `http://minio:9000`
- Login/Password: `minioadmin` / `minioadmin`
- Extra: `{"endpoint_url": "http://minio:9000"}`

spark_default (Spark)

- Host: `spark://spark-master`
- Port: `7077`

Check: operators reference these via `conn_id`.



Jinja Templating in Airflow

```
# Template variables available in templated_fields:
# {{ ds }}               -> "2026-04-20"   (execution date YYYY-MM-DD)
# {{ ds_nodash }}        -> "20260420"
# {{ ts }}               -> ISO 8601 timestamp
# {{ execution_date }}   -> pendulum datetime object
# {{ next_ds }}          -> next scheduled execution date
# {{ prev_ds }}          -> previous execution date
# {{ dag.dag_id }}       -> "east_africa_etl"
# {{ task.task_id }}     -> "extract_transactions"
# {{ params.bucket }}    -> from params= dict
# {{ var.value.bucket }} -> from Airflow Variable

load_cmd = BashOperator(
    task_id="load_to_minio",
    bash_command="""
        mc cp /tmp/output_{{ ds_nodash }}.parquet \
           minio/warehouse/transactions/date={{ ds }}/
    """,
    dag=dag,
)
```



Retries, SLA, and Email Alerts

```python
from datetime import timedelta

# Task defaults: applied to every task unless overridden
default_args = {
    "owner": "innocent",
    "retries": 3,  # retry transient failures (network/timeouts)
    "retry_delay": timedelta(minutes=10),
    "retry_exponential_backoff": True,      # 10, 20, 40... minutes between retries
    "max_retry_delay": timedelta(hours=1), # cap the backoff
    "email_on_failure": True,
    "email_on_retry": False,
    "email": ["innocent@iitmz.ac.in", "team@iitmz.ac.in"],
    "sla": timedelta(hours=3),              # alert if task exceeds this duration
}

# Per-task SLA override (stricter for critical tasks):
critical_task = PythonOperator(
    task_id="critical_load",
    python_callable=load_fn,
    sla=timedelta(hours=1),
    dag=dag,
)
```



Airflow UI — Key Views

DAG list view

- Toggle DAG on/off (paused)
- See last run status
- Quick trigger button
- Tag-based filtering

Grid view (Airflow 2.3+)

- All DAG runs as columns
- Task instance matrix
- Click a cell for logs

Graph view

- Visual DAG topology
- Task status colours
- Hover for task details

Gantt chart

- Task timing and overlap
- Identify bottlenecks
- Duration vs queue time

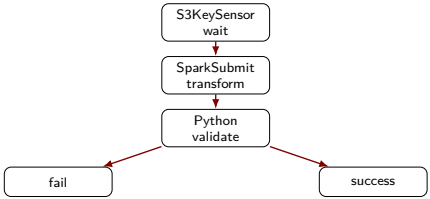
Access logs: Grid View → click task cell → **Log** tab



ETL Pipeline Design — Full Architecture

Pipeline idea (end-to-end)

1. Wait for input data to arrive (sensor)
2. Run a Spark transformation job (SparkSubmitOperator)
3. Validate output (row-count / schema checks)
4. Branch: notify success vs notify failure



Storage model

- Raw files land in MinIO: `s3a://warehouse/raw/...`
- Curated outputs land in Delta Lake: `s3a://warehouse/warehouse/...`

Full ETL DAG — Part 1: Setup & Extract

```python
from airflow.decorators import dag, task
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
from airflow.operators.dummy import DummyOperator
from datetime import datetime, timedelta

# Spark + MinIO (S3) config passed to SparkSubmitOperator
SPARK_CONF = {
    "spark.hadoop.fs.s3a.endpoint":          "http://minio:9000",
    "spark.hadoop.fs.s3a.access.key":        "minioadmin",
    "spark.hadoop.fs.s3a.secret.key":        "minioadmin",
    "spark.hadoop.fs.s3a.path.style.access": "true",
}

# Define the DAG (schedule + start date + retry defaults)
@dag(
    dag_id="east_africa_etl",
    schedule_interval="0 2 * * *",          # daily at 02:00
    start_date=datetime(2026, 4, 1),
    catchup=False,
    default_args={"retries": 2, "retry_delay": timedelta(minutes=5)},
)
```



Full ETL DAG — Part 2: Spark & Validate

```python
# Task 2: submit the Spark job (Airflow orchestrates; Spark computes)
spark_job = SparkSubmitOperator(
    task_id="spark_transform",
    application="/opt/airflow/spark_jobs/transform_transactions.py",
    conn_id="spark_default",                  # Connection in Airflow UI
    application_args=["--date", "{{ ds }}"],  # pass logical date to Spark
    conf=SPARK_CONF,
    num_executors=2,
    executor_memory="1g",
)

# Task 3: validate outputs (keep validation deterministic + lightweight)
@task
def validate_output(ds=None):
    from pyspark.sql import SparkSession
    spark = SparkSession.builder.master("spark://spark-master:7077").getOrCreate()
    df = spark.read.parquet(f"s3a://warehouse/transactions/date={ds}/")
    count = df.count()
    if count == 0:
        raise ValueError("Output table is empty!")
    return {"rows": count, "date": ds}       # small dict returned via XCom
```



Best Practices for Production DAGs

Do:

- ✓ Make every task **idempotent**
- ✓ Use **atomic writes** (write to temp, rename)
- ✓ Partition output by {{ ds }}
- ✓ Set `catchup=False` by default
- ✓ Keep DAG files **lightweight** (no heavy imports at top level)
- ✓ Use **Variables/Connections** for config
- ✓ Add **tags** for discoverability
- ✓ Set **SLAs** and email alerts

Don't:

- × No `print(df.show())` in DAG file body
- × No database connections at module level
- × No `SELECT *` without date filter (unbounded)
- × Don't store large blobs in XCom
- × Don't rely on local filesystem for state
- × Don't hard-code credentials in DAG files
- × Don't use mutable global state



Hands-On Roadmap (What You Will Build)

Target: one DAG that waits for data, runs a Spark job, validates output, then branches to success/fail.

You will do it in 7 steps:

1. Start/verify Docker services
2. Log in to Airflow UI and locate your DAGs
3. Create required **Connections** (MinIO/S3 and Spark)
4. Run a **minimal hello-world DAG**
5. Add retries + params (Jinja templates)
6. Add a sensor (wait for file)
7. Add SparkSubmitOperator + validation and observe everything in the UI



Step 1: Verify the Lab Services (Docker)

Goal: confirm Airflow, Spark, and MinIO containers are running.

```
# show running services
docker compose ps | egrep "airflow|spark|minio"

# if needed, start the core Airflow services
docker compose up -d airflow-webserver airflow-scheduler

# optional: restart if the UI is stuck
docker compose restart airflow-webserver airflow-scheduler
```

Expected:

- Airflow UI at `http://localhost:8085`
- Spark master UI at `http://localhost:8080`
- MinIO console depends on your compose file (often `:9001`)



Step 2: Airflow UI (Where to Click)

Login: `admin / admin`

UI flow you will use constantly:

1. **DAGs list** → search your DAG name
2. Click DAG → **Graph** or **Grid**
3. Click a task → **Log** (read the last lines first)
4. For failures: **Clear** task instance → re-run after fixing code

Important: keep DAG *unpaused* (toggle on) to allow scheduling.



Step 3: Create Connections (Mandatory)

Go to: `Admin` → `Connections`

Create these two connections:

- **minio_s3** (Type: Amazon S3)
  - Host: `http://minio:9000`
  - Login/Password: `minioadmin` / `minioadmin`
  - Extra: {`"endpoint_url": "http://minio:9000"`}
- **spark_default** (Type: Spark)
  - Host: `spark://spark-master`
  - Port: 7077

Why this matters: operators (S3 sensor, Spark submit) look up credentials via `conn_id`.



Step 4: Minimal DAG (Hello World)

Goal: prove Airflow can parse your DAG and run one task.
Create file: /opt/airflow/dags/lab06_hello.py

```python
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

# A minimal DAG: 1 task, manual trigger
with DAG(
    dag_id="lab06_hello",               # name shown in the UI
    start_date=datetime(2026, 4, 20),   # required even when schedule=None
    schedule_interval=None,             # manual trigger only
    catchup=False,
    tags=["lab06"],
) as dag:

    # One task; prints the logical date (ds)
    hello = BashOperator(
        task_id="hello",
        bash_command='echo "Hello Airflow! ds={{ ds }}"',
    )
```



Step 5: Dependencies + Retries + Parameters

Goal: two tasks with dependency and retry behaviour.

```
from datetime import timedelta

# Task defaults: retry twice with a 2-minute delay
default_args = {
    "retries": 2,
    "retry_delay": timedelta(minutes=2),
}

with DAG(
    dag_id="lab06_retry_demo",
    start_date=datetime(2026, 4, 20),
    schedule_interval=None,   # manual trigger
    catchup=False,
    default_args=default_args,
    tags=["lab06"],
) as dag:

    step1 = BashOperator(
        task_id="step1",
        bash_command='echo "Step 1 for {{ ds }}"',  # ds is the logical date
    )
```



Step 6: Add a Sensor (Wait for a File)

Goal: do not start the pipeline until input data exists.

```python
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor

# Sensor task: polls S3/MinIO until the file for this ds exists
wait_for_file = S3KeySensor(
    task_id="wait_for_file",
    bucket_name="warehouse",
    bucket_key="raw/transactions/{{ ds }}/data.csv",  # templated path by date
    aws_conn_id="minio_s3",                           # Connection in Airflow UI
    poke_interval=30,                                  # check every 30s
    timeout=60 * 20,                                   # fail after 20 min
    mode="reschedule",                                 # free worker slot between pokes
)
```

Rule: sensors should usually use `mode="reschedule"` so they do not block a worker slot.



Step 7: Trigger Spark + Validate Output (End-to-End)

Goal: wait → spark job → validate → branch.
What to look at in the UI:

- Graph view: are dependencies correct?
- Logs: does SparkSubmitOperator reach the Spark master?
- On failure: confirm retries and read the error in logs

You already have the full example in this deck:

- **ETL diagram** (the intended DAG)
- **Full ETL DAG code Part 1 and Part 2**



Today's Lab Tasks

Lab 6 — Apache Airflow & pipeline orchestration

1. **Environment check:** verify all Docker services, Airflow UI login, and SparkSession.
2. **Build a custom DAG:** domain of your choice; $\geq$5 tasks with dependencies and scheduling.
3. **Error handling:** retries, SLA, BranchPythonOperator, and XCom between tasks.
4. **Spark integration:** SparkSubmitOperator triggering a real PySpark job.
5. **Reflection:** idempotency and TaskFlow API discussion (4–6 sentences).



References & Further Reading

- Apache Airflow Documentation: `airflow.apache.org/docs/`
- Airflow Best Practices Guide: `/best-practices.html`
- TaskFlow API Tutorial: `/tutorial/taskflow.html`
- Harenslak & de Ruiter, *Data Pipelines with Apache Airflow*, Manning 2021
- Lam, *Learning Apache Airflow*, Packt 2023
- Delta Lake Documentation: `docs.delta.io`

Questions?

`innocent@iitmz.ac.in` | Room 105, Door F1

