ChatGPT

7-Day Comprehensive Plan to Learn Apache Airflow

This 7-day plan assumes you have intermediate Python skills and some familiarity with data engineering concepts. Airflow is an open-source platform that lets you **programmatically author, schedule and monitor workflows** [1]. Workflows are defined as **Directed Acyclic Graphs (DAGs)** of tasks, enabling dynamic, scalable orchestration using Python code [1]. Before Airflow, many teams relied on cron jobs and ad-hoc scripts to automate pipelines; as data volumes grew, these manual solutions became fragile, with tangled dependencies, cascading failures and little visibility [2] [3]. Airflow addresses these pain points by providing centralized scheduling, dependency management, monitoring and retry logic [4] [5].

Day 0 – What Is Airflow and Why It Matters

Goals: Understand what Airflow is, why data orchestration matters, and how Airflow improves on cron-based automation. Explore a simple example contrasting manual scripts with an Airflow DAG.

🔍 Before Airflow: Manual Scripts and Cron Jobs

Many data engineers start by writing a few scripts to pull data from APIs, clean it and load it into a warehouse. They schedule these scripts with cron or run them manually. This works until data volumes grow and dependencies multiply [2]. Jobs run out of order, failures cascade and quick fixes turn into fragile automation [2]. Cron lacks monitoring, dependency management and scalability [6]. There is no easy way to see if a job succeeded or failed, to manage task dependencies or to retry on failure [7].

How Airflow Solves These Problems

Airflow is an open-source platform that lets you **author, schedule and monitor workflows** using Python [1]. Workflows are defined as DAGs—graphs of tasks with explicit dependencies—ensuring tasks run in a specific order [8]. Airflow provides built-in **monitoring**, **retry and failure handling**, and a **web UI** to view DAGs and logs [5]. It supports **dynamic workflows**, allowing you to generate tasks at runtime [9], and is highly **extensible**, with a modular architecture and a large ecosystem of operators, sensors and hooks [10]. Airflow scales horizontally by distributing tasks across workers [11] and offers a user-friendly UI for monitoring and managing workflows [12].

Example: Cron vs Airflow

Imagine you need to run a three-step ETL process every day: **collect data**, **transform it**, and **upload results**. Without Airflow, you might create three separate Python scripts and schedule them with cron. Managing dependencies and monitoring success manually quickly becomes error-prone. In Airflow, you can express this workflow as a DAG:

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

@dag(
    dag_id="daily_etl_example",
    schedule="@daily",
    start_date=datetime(2025, 12, 1),
    catchup=False,
)
def etl_pipeline():
    @task
    def collect_data():
        # fetch raw data
        return "raw_data.csv"

    @task
    def transform(file):
        # clean and transform the data
        return "clean_data.csv"

    @task
    def upload(file):
        # upload to cloud storage
        print(f"Uploading {file}")

    raw = collect_data()
    clean = transform(raw)
    upload(clean)

etl_pipeline = etl_pipeline()
```

When you run `airflow tasks run` or trigger this DAG via the UI, Airflow ensures that `collect_data` runs before `transform`, which runs before `upload`. If any step fails, Airflow can automatically retry or alert you. You can monitor the pipeline's status in the Airflow UI.

Deliverables for Day 0

- Summarize why cron-based automation breaks down for complex pipelines [2] [3].
- Explain how Airflow's DAGs, scheduling, and monitoring address these limitations [5] [1].
- Implement a simple three-task Airflow DAG using the TaskFlow API.

Day 1 – Installation and First DAG

Goals: Set up a local Airflow environment, explore the Airflow web UI, and create your first DAG.

📚 Reading & Concepts

Airflow's core architecture consists of a web server for the UI, a scheduler that triggers tasks once their dependencies are met [13], an executor that runs tasks on workers, and a metadata database to store run history [14]. Airflow is **dynamic** and **extensible**; DAGs are Python files that can leverage loops, conditionals and variables [1] [9].

🛠 Setup

1. **Install Airflow.** Use pip and specify extras for your desired database and provider packages. Example:

   ```
   pip install "apache-airflow[postgres,aws]==2.7.*" --constraint "https://raw.githubusercontent.com/apache/airflow/constraints-2.7.1/constraints-3.10.txt"
   ```

   Alternatively, install via Docker or the Astro CLI (`curl -sSL https://install.astronomer.io | sudo bash` then `astro dev init`) [15].
2. **Initialize the Airflow database.** Set the `AIRFLOW_HOME` environment variable (defaults to `~/airflow`) and run:

   ```
   airflow db init
   ```

3. **Create a user.** Add an admin user for the web UI:

   ```
   airflow users create --username admin --firstname Admin --lastname User --role Admin --email admin@example.com --password admin
   ```

4. **Start Airflow.** In separate terminals, run the web server and scheduler:

   ```
   airflow webserver --port 8080
   airflow scheduler
   ```

   Access the UI at `http://localhost:8080` and log in.

Practice

1. **Create your first DAG.** In `$AIRFLOW_HOME/dags/`, create `hello_airflow.py`:

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

   default_args = {
   ```

```
    "start_date": datetime(2025, 12, 1),
    "retries": 1,
}

with DAG(
    dag_id="hello_airflow",
    default_args=default_args,
    schedule_interval="@daily",
    catchup=False,
    tags=["example"],
) as dag:
    say_hello = BashOperator(
        task_id="say_hello",
        bash_command="echo 'Hello, Airflow!'"
    )
```

2. **Trigger the DAG.** Enable and run the DAG from the UI. Observe the task state (queued, running, success) and review logs.
3. **Explore the UI.** Navigate the Graph view, Tree view and Logs to understand how Airflow represents DAGs and tasks.

Deliverables

- A running local Airflow installation with web server and scheduler.
- A simple DAG that prints a message via `BashOperator`.
- Familiarity with the Airflow UI and log viewing.

Day 2 – DAGs, Tasks and Scheduling

Goals: Learn how to define DAGs and tasks using both the classic `DAG` context manager and the TaskFlow API. Explore scheduling, default arguments and dependency management.

📚 Reading & Concepts

In Airflow, a **DAG** is the blueprint of a workflow—it defines which tasks exist and in what order they run [16]. **Tasks** are the individual units of work [17]. Using the TaskFlow API, you can convert Python functions into tasks using the `@task` decorator [18], and Airflow automatically constructs the DAG dependencies when tasks are called in sequence [19].

🛠 Practice

1. **Use the TaskFlow API.** Create `taskflow_etl.py` in your DAGs directory:

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

```python
@dag(
    dag_id="taskflow_etl",
    schedule="0 6 * * *",  # run daily at 6am
    start_date=datetime(2025, 12, 1),
    catchup=False,
    default_args={"retries": 2, "retry_delay": 300},
)
def pipeline():
    @task
    def extract():
        return ["red", "green", "blue"]

    @task
    def transform(colors):
        return [c.upper() for c in colors]

    @task
    def load(colors):
        print("Loaded", colors)

    load(transform(extract()))

pipeline_dag = pipeline()
```

Deploy and run the DAG. Notice how `extract`, `transform` and `load` tasks are automatically ordered.

2. **Use the classic `DAG` context manager.** Create `classic_dag.py`:

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

def greet():
    print("Hello from classic DAG!")

with DAG(
    dag_id="classic_dag",
    start_date=datetime(2025, 12, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
    greet_task = PythonOperator(
        task_id="greet",
        python_callable=greet
    )
```

3. **Explore scheduling options.** Change the `schedule_interval` to cron expressions (e.g., `"0 */4 * * *"` for every 4 hours) or presets like `@hourly`, `@weekly`. Use `catchup=True` to backfill past runs if needed.

Deliverables

- DAGs defined using both TaskFlow API and classic `DAG` context manager.
- Understanding of scheduling and default arguments (retries, retry delay, start dates, catchup).
- Experience running DAGs and examining task dependencies in Graph view.

Day 3 – Operators, Hooks and Sensors

Goals: Explore Airflow's built-in operators, sensors and hooks to interact with external systems and wait for conditions. Build a DAG that uses multiple operator types.

📚 Reading & Concepts

Airflow provides an extensible set of **operators** to perform actions (e.g., BashOperator, PythonOperator, EmailOperator, PostgresOperator), **hooks** to connect to external systems and reuse authentication logic, and **sensors** to wait for a condition (like a file arrival or a database row). Airflow is highly extensible—you can create custom operators, sensors and hooks for bespoke tasks [10].

🛠 Practice

1. **Work with built-in operators.** Create `operators_example.py`:

```
from datetime import datetime
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
from airflow.operators.email import EmailOperator

def process_file():
    print("Processing file...")

with DAG(
    dag_id="operators_example",
    start_date=datetime(2025, 12, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    download = BashOperator(
        task_id="download_file",
        bash_command="curl -o /tmp/data.csv https://example.com/data.csv"
    )
```

```
    process = PythonOperator(
        task_id="process_file",
        python_callable=process_file
    )
    notify = EmailOperator(
        task_id="send_email",
        to="data-team@example.com",
        subject="Data pipeline complete",
        html_content="<p>The data pipeline succeeded!</p>"
    )
    download >> process >> notify
```

2. **Use sensors to wait for external events.** Create `sensor_example.py`:

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.sensors.filesystem import FileSensor
from airflow.operators.python import PythonOperator

def process():
    print("File is ready, processing now")

with DAG(
    dag_id="sensor_example",
    start_date=datetime(2025, 12, 1),
    schedule_interval="@hourly",
    catchup=False,
) as dag:
    wait_for_file = FileSensor(
        task_id="wait_for_file",
        filepath="/tmp/inbound/input.csv",
        poke_interval=60,  # check every minute
        timeout=3600,
    )
    process_file = PythonOperator(
        task_id="process",
        python_callable=process
    )
    wait_for_file >> process_file
```

3. **Connect to a database using hooks.** In `postgres_example.py`, use the `PostgresHook` to query a table:

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

```
from airflow.providers.postgres.hooks.postgres import PostgresHook

def query_table():
    hook = PostgresHook(postgres_conn_id="my_postgres")
    records = hook.get_records("SELECT COUNT(*) FROM users")
    print(f"User count: {records[0][0]}")

with DAG(
    dag_id="postgres_example",
    start_date=datetime(2025, 12, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    run_query = PythonOperator(
        task_id="query_table",
        python_callable=query_table
    )
```

Configure the connection `my_postgres` in the Airflow UI under **Admin → Connections**.

Deliverables

- DAG using multiple operators (bash, python, email) with dependencies.
- DAG using a FileSensor or other sensor to wait for external events.
- DAG demonstrating a hook (e.g., querying a Postgres table). Understand how to set up connections in Airflow.

Day 4 – Branching, XComs and Dynamic Workflows

Goals: Learn to build advanced DAG patterns using branching, cross-communication (XComs) and dynamic task mapping.

📚 Reading & Concepts

Airflow supports **branching** with operators like `BranchPythonOperator` to choose among different paths. **XComs** (cross-communications) allow tasks to exchange data. **Dynamic task mapping** creates tasks at runtime based on input data, enabling parallel processing of variable lists. These advanced patterns help create flexible and scalable workflows.

🛠 Practice

1. **Branching with `BranchPythonOperator`.** Create `branching_example.py`:

```
from datetime import datetime
from airflow import DAG
```

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

def decide_path():
    from random import choice
    return "path_a" if choice([True, False]) else "path_b"

def do_a():
    print("Executed path A")

def do_b():
    print("Executed path B")

def join():
    print("Both paths complete")

with DAG(
    dag_id="branch_example",
    start_date=datetime(2025, 12, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    branch = BranchPythonOperator(
        task_id="branch",
        python_callable=decide_path
    )
    task_a = PythonOperator(task_id="path_a", python_callable=do_a)
    task_b = PythonOperator(task_id="path_b", python_callable=do_b)
    join_task = PythonOperator(task_id="join", python_callable=join)
    branch >> [task_a, task_b] >> join_task
```

2. **Passing data via XCom.** Using the TaskFlow API automatically pushes return values to XCom. Retrieve XComs explicitly:

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

@dag(dag_id="xcom_example", start_date=datetime(2025, 12, 1),
schedule=None, catchup=False)
def pipeline():
    @task
    def generate_numbers():
        return [1, 2, 3]

    @task
    def sum_numbers(numbers):
        return sum(numbers)
```

```
    @task
    def log_result(total):
        print(f"Total is {total}")

    numbers = generate_numbers()
    total = sum_numbers(numbers)
    log_result(total)

dag = pipeline()
```

3. **Dynamic task mapping.** Process a list of files in parallel:

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

@dag(dag_id="dynamic_map_example", start_date=datetime(2025, 12, 1),
schedule=None, catchup=False)
def pipeline():
    @task
    def get_files():
        return ["file1.csv", "file2.csv", "file3.csv"]

    @task
    def process(file):
        print(f"Processing {file}")

    process.expand(file=get_files())

dag = pipeline()
```

   Airflow will spawn a task instance for each file, running them in parallel.

Deliverables

- DAG using a branching operator to run different task paths.
- DAG demonstrating XCom usage to pass data between tasks.
- DAG using dynamic task mapping to process a variable number of items.

Day 5 – ETL Pipelines and External Integrations

Goals: Build a realistic ETL pipeline using Airflow's operators and hooks to extract data from a source, transform it and load it into a target. Learn to manage credentials via Airflow connections and variables.

📚 Reading & Concepts

Airflow is widely used for ETL (Extract, Transform, Load) pipelines [20]. Its ability to handle complex dependencies and conditional execution makes it ideal for sequencing extraction, transformation and loading tasks [21]. Airflow's extensible hooks and operators enable integration with databases, cloud storage, APIs and data warehouses [10].

🛠 Practice

1. **Set up connections.** In the Airflow UI, navigate to **Admin → Connections** and add connections for your data sources (e.g., `postgres_default`, `aws_default`). Store credentials securely using environment variables or a secrets backend.
2. **Extract from a database.** Create `etl_pipeline.py`:

```
from datetime import datetime
from airflow import DAG
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
import pandas as pd

@task
def extract_orders():
    hook = PostgresHook(postgres_conn_id="postgres_default")
    df = hook.get_pandas_df("SELECT order_id, customer_id, amount FROM
orders WHERE order_date >= CURRENT_DATE - INTERVAL '1 day'")
    return df.to_json(orient="split")  # serialize for XCom

@task
def transform(json_data):
    df = pd.read_json(json_data, orient="split")
    df["amount"] = df["amount"].astype(float)
    df["amount_usd"] = df["amount"] * 1.0  # currency conversion example
    return df.to_csv(index=False)

@task
def load(csv_data):
    s3 = S3Hook(aws_conn_id="aws_default")
    s3_key = f"etl/output/
orders_{datetime.utcnow().strftime('%Y%m%d%H%M%S')}.csv"
    s3.load_string(csv_data, key=s3_key, bucket_name="my-data-bucket")
    return s3_key

with DAG(
    dag_id="etl_pipeline",
    start_date=datetime(2025, 12, 1),
    schedule_interval="0 2 * * *",  # daily at 2 AM
```

```
    catchup=False,
) as dag:
    key = load(transform(extract_orders()))
```

This DAG extracts orders from a Postgres database, transforms them with pandas, and uploads the result to S3.

3. **Use provider operators.** Replace custom code with provider operators like `PostgresOperator` for SQL queries or `S3ToRedshiftOperator` for loading into Amazon Redshift. Review Airflow's provider package documentation for details.

Deliverables

- A working ETL DAG that extracts from a database, transforms data and loads it into cloud storage.
- Experience configuring Airflow connections and variables for credentials.
- Awareness of provider-specific operators for common ETL tasks.

Day 6 – Monitoring, Alerts and Best Practices

Goals: Learn to monitor Airflow workflows, configure alerts and apply best practices for reliability and maintainability.

📚 Reading & Concepts

Airflow provides a **web UI** that displays DAGs, task states and logs [12]. Good practices include using retries and exponential backoff [5], enabling email or Slack notifications, writing idempotent tasks, and keeping business logic out of the DAG file. Airflow supports **dynamic workflows** and built-in idempotency and backfills, allowing you to rerun historical data safely [22].

🛠 Practice

1. **Configure email notifications.** Set SMTP credentials in `airflow.cfg` or via environment variables (`smtp_host`, `smtp_user`, `smtp_password`). In your DAG's default arguments, set `email`, `email_on_failure=True`, `email_on_retry=False`.
2. **Add retry logic.** In DAG default args, set `retries` and `retry_delay`. Use `max_active_runs` and `concurrency` to control parallelism.
3. **Use on_failure_callback.** Define a Python callback that sends a Slack message or logs a custom metric when a task fails.

```
from airflow.utils.email import send_email

def notify_slack(context):
    dag_id = context['dag'].dag_id
    task_id = context['task'].task_id
    msg = f"Task {task_id} in DAG {dag_id} failed."
    send_email(to="alerts@example.com", subject="Airflow Task Failure",
```

```
    html_content=msg)

default_args = {
    "retries": 3,
    "retry_delay": 300,
    "on_failure_callback": notify_slack,
}
```

4. **Monitor via the UI and CLI.** Use the Airflow UI to view logs and Gantt charts. Use `airflow tasks list <dag_id>` to list tasks, `airflow tasks test <dag_id> <task_id> <execution_date>` to test tasks locally, and `airflow dags backfill` to run historical data.
5. **Adopt best practices.** Keep tasks idempotent, use small, single-purpose tasks, externalize heavy processing to Spark or warehouses, version your DAGs in Git, and set sensible `start_date` and `catchup` values to avoid accidental backfills.

Deliverables

- DAG with email notifications and retry logic configured.
- Custom on-failure callback integrated into a DAG.
- Familiarity with monitoring tasks via UI and CLI.
- Summary of best practices for reliable Airflow pipelines.

Day 7 – Scaling, Deployment and Extensibility

Goals: Understand how to deploy Airflow in production, scale it across multiple workers and extend it with custom operators and sensors.

📚 Reading & Concepts

Airflow supports multiple **executors**—the component that runs tasks—including the LocalExecutor (runs tasks on the same machine), CeleryExecutor (distributes tasks across a Celery cluster) and KubernetesExecutor (schedules tasks on Kubernetes pods). The system's **extensibility** allows you to write custom operators and integrate with any external system [10] [11]. Airflow's UI and configuration-as-code approach make workflows shareable and maintainable [12].

🛠 Practice

1. **Run Airflow with LocalExecutor.** In `airflow.cfg`, set `executor = LocalExecutor` and restart your scheduler and webserver. Observe that tasks run in parallel on the local machine.
2. **Try the CeleryExecutor or KubernetesExecutor.** Follow the official deployment guides to configure a message broker (e.g., RabbitMQ or Redis) for Celery or a Kubernetes cluster for the Kubernetes executor. Experiment with scaling worker nodes and observing throughput.
3. **Containerize your Airflow deployment.** Use Docker Compose or the Astro CLI to run Airflow in containers. Example `docker-compose.yaml` is provided in the Airflow repository; `astro dev start` sets up a local Airflow environment with Docker [15].

4. **Write a custom operator.** Create a Python class that inherits from `BaseOperator` and implements the `execute()` method. For instance, write a `HelloOperator` that prints a greeting:

```
from airflow.models import BaseOperator
from airflow.utils.decorators import apply_defaults

class HelloOperator(BaseOperator):
    @apply_defaults
    def __init__(self, name: str, **kwargs):
        super().__init__(**kwargs)
        self.name = name

    def execute(self, context):
        print(f"Hello {self.name} from a custom operator!")
```

   Use the operator in a DAG:

```
from datetime import datetime
from airflow import DAG

with DAG(
    dag_id="custom_operator_example",
    start_date=datetime(2025, 12, 1),
    schedule_interval=None,
    catchup=False,
) as dag:
    hello = HelloOperator(task_id="hello", name="Airflow")
```

5. **Plan for production.** Consider using managed Airflow services (Astronomer, AWS MWAA, Google Cloud Composer) for production deployments. Set up CI/CD pipelines to test and deploy DAGs automatically.

Deliverables

- Experience with different executors and an understanding of when to use each.
- Containerized Airflow environment or local Astro CLI setup for development.
- Custom operator implemented and used in a DAG.
- Deployment strategy for a production Airflow installation.

Additional Resources

- **Dataquest Introduction to Apache Airflow:** This tutorial explains why manual scripts and cron jobs fail at scale and introduces Airflow as a scalable orchestration tool [2] [3].

- **Dremio Blog – Orchestration of Dremio with Airflow and CRON Jobs:** Discusses the limitations of CRON (no monitoring, dependency management, limited scalability) and highlights Airflow's advantages such as monitoring, retry handling, scalability and dynamic scheduling [23] [24].
- **Komodor Guide to Airflow:** Summarizes Airflow's design principles—dynamic, extensible, scalable and elegant [25] —and enumerates key use cases like ETL pipelines, machine learning workflows, data analytics and DevOps tasks [26].
- **Astronomer Get Started with Airflow:** Provides step-by-step instructions for starting an Airflow project using the Astro CLI and writing your first DAG [15].

By following this plan, dedicating several hours per day to practice, and exploring the official documentation and community resources, you will build a solid understanding of Airflow's core concepts—DAGs, tasks, scheduling, sensors, operators, monitoring, scaling and deployment—and be ready to orchestrate reliable data workflows in production.

---

[1] [8] [9] [10] [11] [12] [20] [21] [25] [26] Apache Airflow: Use Cases, Architecture, and 6 Tips for Success
https://komodor.com/learn/apache-airflow-use-cases-architecture-and-6-tips-for-success/

[2] [3] [14] [16] [17] [18] [19] [22] Introduction to Apache Airflow – Dataquest
https://www.dataquest.io/blog/introduction-to-apache-airflow/

[4] [5] [6] [7] [15] [23] [24] Orchestration of Dremio with Airflow and CRON Jobs | Dremio
https://www.dremio.com/blog/orchestrating-dremio-with-airflow/

[13] Scheduler — Airflow 3.1.5 Documentation
https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/scheduler.html