Module 6 - Tuần 1 - Apache Airflow trong thực hành MLOps

Lập lịch và điều phối pipeline cho dự án AI

Time-Series Team

Ngày 21 tháng 11 năm 2025

Bài blog này được viết dựa trên nội dung buổi TA về Apache Airflow trong khoá All-in-One AI Vietnam, với mục tiêu diễn giải lại chi tiết hơn dưới dạng câu chuyện để người đọc có thể dùng như một tài liệu tự học. Thay vì chỉ nhìn slide rời rạc, ta sẽ lần lượt đi từ bối cảnh thực tế, lý do cần Airflow, tới kiến trúc, cách định nghĩa workflow, rồi kết thúc bằng một ví dụ DAG cụ thể.

1 Vì sao chúng ta cần Airflow?

Khi mới học máy học, ta thường làm việc trong một notebook duy nhất: đọc dữ liệu từ một file CSV, tiền xử lý, huấn luyện một mô hình, vẽ vài biểu đồ rồi dừng lại. Cách làm này rất phù hợp cho các thử nghiệm ban đầu. Tuy nhiên, nếu đặt trong một hệ thống thực tế, mọi thứ phức tạp hơn nhiều.

Hãy hình dung một dự án dự báo nhu cầu sản phẩm cho một chuỗi bán lẻ. Dữ liệu không chỉ nằm trong một file CSV duy nhất, mà liên tục được đổ về từ các nguồn khác nhau: đơn hàng mới, trạng thái kho, khuyến mãi, ngày nghỉ lễ. Mỗi đêm, ta cần chạy một loạt bước: trích xuất dữ liệu từ cơ sở dữ liệu giao dịch, làm sạch và tổng hợp lại theo ngày, huấn luyện hoặc cập nhật lại mô hình dự báo, đánh giá xem mô hình mới có tốt hơn bản đang chạy hay không, nếu tốt thì triển khai, nếu không thì giữ nguyên mô hình cũ. Tất cả những việc đó lại phải lặp lại đều đặn hàng ngày, hàng tuần, trong suốt vòng đời của hệ thống.

Nếu ta chỉ viết một script Python thật dài rồi dùng `cron` để chạy, việc quản lý sẽ nhanh chóng trở nên rối rắm: khi một bước nào đó lỗi, ta khó nhìn thấy ngay nó dừng ở đâu; khi muốn chạy lại một bước giữa chừng, ta phải sửa script; khi muốn thêm bước gửi email khi lỗi, lại phải chèn thêm code. Đó là lúc các nền tảng orchestration như Apache Airflow xuất hiện. Thay vì nghĩ tới "một script dài", Airflow buộc ta phải nghĩ theo *workflow*: một tập các tác vụ nhỏ, có quan hệ phụ thuộc rõ ràng, được lập lịch và giám sát một cách có hệ thống.

2 DAG: cách Airflow nhìn một workflow

Airflow biểu diễn workflow dưới dạng *Directed Acyclic Graph* (DAG). "Directed" nghĩa là mỗi cạnh có hướng, cho biết task nào phải chạy trước, task nào chạy sau. "Acyclic" nghĩa là đồ thị không được phép có vòng lặp: ta không thể để A phụ thuộc vào B trong khi B lại phụ thuộc vào A thông qua một chuỗi các task khác.

Suy nghĩ theo DAG giúp ta thiết kế pipeline rõ ràng hơn. Ví dụ, trong một pipeline huấn luyện mô hình đơn giản, ta có thể xác định bốn bước: tải dữ liệu, xử lý dữ liệu, huấn luyện mô hình, và lưu mô hình. Trong Airflow, bốn bước này sẽ trở thành bốn task. Ta khai báo rằng task xử lý dữ liệu chỉ bắt đầu sau khi tải dữ liệu xong, huấn luyện mô hình chỉ bắt đầu sau khi xử lý dữ liệu xong, và cuối cùng lưu mô hình chỉ diễn ra khi huấn luyện thành công. Khi vẽ lên UI của Airflow, ta sẽ thấy một DAG tuyến tính gồm bốn node nối nhau, rất dễ kiểm tra.

Một điểm hay nữa của DAG là nó cho phép ta dễ dàng song song hoá những phần không phụ thuộc lẫn nhau. Chẳng hạn, nếu sau khi xử lý dữ liệu, ta muốn vừa huấn luyện mô hình A vừa huấn luyện mô hình B, ta chỉ cần cho hai task này cùng phụ thuộc vào bước xử lý dữ liệu và không phụ thuộc vào nhau. Scheduler của Airflow sẽ tự phân bổ tài nguyên phù hợp để chạy hai task này song song nếu có thể.

3 Kiến trúc Airflow: scheduler, webserver và metadata

Trong kiến trúc cơ bản mà slide trình bày, Airflow có ba thành phần chính. *Scheduler* là bộ não của hệ thống: nó đọc các file DAG, hiểu cấu trúc workflow, tính toán thời điểm cần tạo ra các "task instance" (các lần chạy cụ thể), và yêu cầu executor thực thi chúng. *Webserver* là giao diện để con người tương tác: chúng ta vào UI để bật/tắt DAG, xem lịch chạy, xem log, xem biểu đồ Gantt hoặc tree view cho từng workflow. Cuối cùng, mọi thông tin về lịch sử chạy được ghi vào một cơ sở dữ liệu gọi là *metadata database*: DAG nào đã chạy, run nào thành công, run nào thất bại, mỗi task mất bao nhiêu thời gian.

Việc tách ba thành phần này mang lại nhiều lợi ích. Ta có thể mở rộng webserver và worker (nơi thực thi task) độc lập. Trong một môi trường lớn, executor có thể chạy trên cluster Kubernetes, mỗi task được thực thi trong một container riêng. Tuy nhiên, với người học, điều quan trọng là hiểu rằng Airflow không thay thế code ML của ta; nó chỉ là "nhạc trưởng" điều phối các đoạn code đó theo đúng thứ tự và lịch trình.

4 Task, Operator và cách truyền dữ liệu giữa các bước

Trong Airflow, đơn vị công việc nhỏ nhất là *task*. Mỗi task được sinh ra từ một *operator*. Nếu ta muốn chạy một câu lệnh bash, ta dùng `BashOperator`. Nếu muốn gọi một hàm Python, ta dùng `PythonOperator`. Nếu muốn kích hoạt một container Docker, có `DockerOperator`. Nhờ đó, dù bản chất công việc bên trong khá đa dạng, bề mặt cấu hình để đưa nó vào DAG vẫn thống nhất.

Một câu hỏi thường gặp là: dữ liệu đi từ task này sang task khác như thế nào? Airflow cung cấp cơ chế XCom để các task có thể đẩy và kéo những mẩu dữ liệu nhỏ. Ví dụ, một task có thể tính ra đường dẫn tới file dữ liệu và đẩy giá trị đó vào XCom; task phía sau chỉ việc kéo giá trị từ XCom ra để tiếp tục xử lý. Tuy nhiên, vì XCom được lưu trong metadata database, nó không phù hợp cho dữ liệu lớn như bảng hàng triệu dòng hay file ảnh. Trong thực tế, ta thường kết hợp Airflow với một hệ thống lưu trữ ngoài như S3, GCS, hoặc MinIO: mỗi task sẽ đọc/ghi dữ liệu lớn từ đó, còn Airflow chỉ lưu các "điểm nối" (như đường dẫn) bằng XCom khi cần thiết.

5 Một ví dụ DAG đơn giản: chạy hằng ngày

Để hình dung rõ hơn, ta thử xây dựng một DAG rất nhỏ cho bài toán giả lập: mỗi ngày, lúc 1 giờ sáng, ta muốn tải dữ liệu mới, làm sạch, và huấn luyện lại mô hình. Trong Airflow, đoạn code có thể trông như sau.

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

def fetch_data():
    # Fake: download data from S3 and save to /tmp/data.csv
    print("Downloading data from S3 ...")

def clean_data():
    # Fake: read /tmp/data.csv, clean, save to /tmp/clean.csv
    print("Cleaning data ...")

def train_model():
    # Fake: read /tmp/clean.csv, train model, save model.pkl
    print("Training model ...")

default_args = {
    "owner": "data_team",
```

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

with DAG(
    dag_id="daily_sales_forecast",
    default_args=default_args,
    schedule_interval="0 1 * * *",  # run at 1AM every day
    start_date=datetime(2025, 11, 6),
    catchup=False,
) as dag:

    t_fetch = PythonOperator(
        task_id="fetch_data",
        python_callable=fetch_data,
    )

    t_clean = PythonOperator(
        task_id="clean_data",
        python_callable=clean_data,
    )

    t_train = PythonOperator(
        task_id="train_model",
        python_callable=train_model,
    )

    t_fetch >> t_clean >> t_train
```

Code Listing 1: Vi du DAG don gian cho he thong du bao ban hang

Trong ví dụ này, ba hàm `fetch_data`, `clean_data` và `train_model` chỉ là những đoạn code giả lập, in ra log để minh hoạ. Điều quan trọng là ta đã biến ý tưởng "mỗi ngày hãy tải dữ liệu, làm sạch, rồi huấn luyện lại" thành một DAG cụ thể với ba task có thứ tự rõ ràng. Trong UI của Airflow, ta sẽ nhìn thấy DAG `daily_sales_forecast` với ba node nối theo thứ tự, và mỗi lần chạy hệ thống sẽ hiển thị trạng thái thành công/thất bại của từng task cùng với log chi tiết.

6 Case study từ slide: pipeline với Arxiv, MinIO và MongoDB

Trong phần sau của slide, giảng viên đưa ra một ví dụ phức tạp hơn nhiều, nhưng rất đáng để hình dung như một "mini project MLOps". Thay vì chỉ in ra log, pipeline thực sự đi qua đầy đủ các bước: gọi API của Arxiv để tải danh sách bài báo, lưu dữ liệu thô vào MinIO, làm sạch dữ liệu, lưu dữ liệu sạch vào MongoDB, sau đó dùng dữ liệu này để huấn luyện và đánh giá một mô hình phân loại.

Ở DAG đầu tiên, dữ liệu được thu thập bằng cách gọi API Arxiv theo một từ khoá đã định. Kết quả trả về dưới dạng JSON được ghi vào MinIO với cấu trúc thư mục phụ thuộc vào *run_id* của Airflow. Mỗi lần DAG chạy tạo ra một thư mục mới, ví dụ `2025-11-08T01:00:00Z/scrape_arxiv_papers.json`, giúp việc truy vết sau này trở nên dễ dàng. Task tiếp theo đọc file JSON này, loại bỏ những trường không cần, chuẩn hoá text, và lưu lại một phiên bản "sạch hơn". Ở bước cuối, dữ liệu sạch được đưa vào MongoDB để dùng lâu dài.

DAG thứ hai tập trung vào huấn luyện mô hình. Nó bắt đầu bằng việc tạo các thư mục cần thiết trong MinIO để lưu lại mô hình và các artifact. Sau đó, DAG tải dữ liệu từ MongoDB hoặc MinIO, kiểm tra sơ bộ chất lượng (ví dụ đảm bảo không thiếu cột quan trọng, số lượng sample không quá ít). Khi mọi thứ ổn, các task tiền xử lý văn bản được kích hoạt: token hoá, chuyển chữ thường, loại bỏ ký tự không cần thiết,... Tiếp theo, dữ liệu được chia thành train, validation và test. Một task huấn luyện

mô hình được gọi, task khác đánh giá mô hình trên tập test, và cuối cùng kết quả (bao gồm score, ma trận nhầm lẫn, mô hình đã train) được lưu lại, một lần nữa gắn với *run_id* hiện tại.

Điểm hay của case study này là nó cho ta thấy sự kết hợp giữa nhiều thành phần: Airflow làm orchestration, MinIO đóng vai trò đối tượng lưu trữ giống S3, MongoDB là nơi lưu các document dữ liệu, và bản thân code ML chạy trong các task Python hoặc Docker. Dù sơ đồ trong slide trông có vẻ phức tạp, nhưng khi nhìn dưới lăng kính DAG, mọi thứ chỉ là tập hợp các task có quan hệ phụ thuộc: "nếu chưa scrape được dữ liệu thì không thể clean", "nếu chưa có dữ liệu sạch thì không thể train model", "nếu train chưa xong thì chưa thể evaluate".

7 Kết lời

Từ những slide giới thiệu ban đầu về Airflow tới case study Arxiv phức tạp, thông điệp chính vẫn nhất quán: khi dự án AI vượt khỏi phạm vi một notebook, ta cần một công cụ đóng vai trò "nhạc trưởng" cho toàn bộ pipeline. Airflow không làm thay ta công việc machine learning, nhưng nó cung cấp ngôn ngữ để mô tả workflow (DAG), cơ chế để lập lịch và retry, giao diện để theo dõi và gỡ lỗi, cùng khả năng tích hợp với rất nhiều hệ thống xung quanh.

Sau khi đọc xong blog này, một bước tiếp theo hợp lý là dựng một môi trường Airflow nhỏ (ví dụ bằng Docker Compose), rồi thử chuyển một mini-project của riêng bạn thành DAG: từ việc tải dữ liệu, xử lý, huấn luyện, cho tới đánh giá và lưu lại mô hình. Khi đã quen với cách suy nghĩ "theo DAG", bạn sẽ thấy việc mở rộng lên hệ thống lớn hơn trở nên tự nhiên hơn rất nhiều.