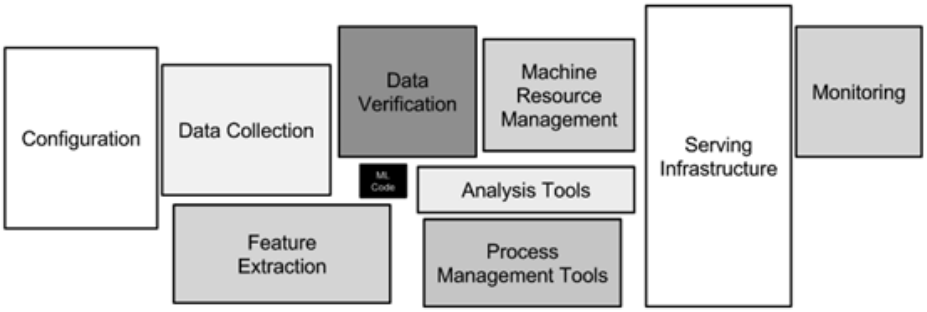
MODEL SERVING AND PACKAGING

AIVN Build Beta

1 Machine Learning System



Hình 1: Machine Learning System

Một hệ thống Machine Learning bao gồm nhiều thành phần quan trọng bao quanh mô hình Machine Learning cốt lõi. Các thành phần như Thu thập dữ liệu, Trích xuất đặc trưng được nằm trong khâu chuẩn bị cho một mô hình Machine Learning. Các phần Quản lý tài nguyên hay Theo dõi các tiến trình nằm trong bước kiểm tra và đánh giá chất lượng mô hình Machine Learning trước khi được đóng gói và triển khai thực tế. Tất cả các thành phần này làm việc cùng nhau để tạo nên một hệ thống Machine Learning đáng tin cậy và có khả năng mở rộng.
Triển khai mô hình Machine Learning là một trong những bước quan trọng để đưa mô hình vào sử dụng trong thực tế. Quá trình này bao gồm việc triển khai một mô hình Machine Learning đã được huấn luyện lên một cơ sở hạ tầng có thể xử lý được các yêu cầu dự đoán. Các công nghệ phổ biến như API RESTful hoặc gRPC được sử dụng thường xuyên để có thể đảm nhiệm khả năng tương tác của mô hình với các bên. Ngoài ra, Docker và Kubernetes cũng được sử dụng để đóng gói và quản lý các dịch vụ trong môi trường sản phẩm.
Để triển khai tốt một mô hình Machine Learning ngoài các quy tắc để có một quá trình inference tốt thì còn cần các yếu tố khác như Tính đáng tin cậy, Khả năng mở rộng, Tính bảo mật và Khả năng giám sát.

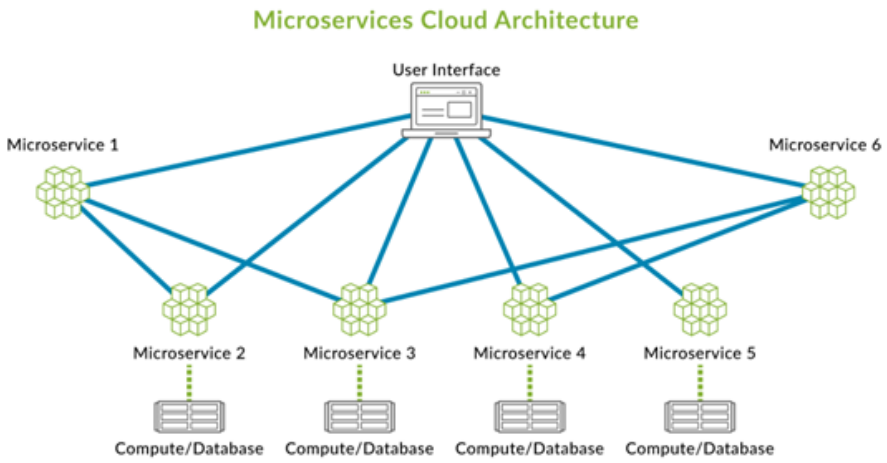
- **Tính đáng tin cậy** được thể hiện bởi thời gian hoạt động lâu dài, đảm bảo rằng dịch vụ luôn sẵn sàng khi cần thiết. Ngoài ra, khả năng tự phục hồi sau sự cố cũng là một đặc điểm quan trọng, cho phép hệ thống tự động hoạt động mà không cần can thiệp thủ công, giảm thiểu thời gian ngừng hoạt động và đảm bảo tính liên tục của dịch vụ.

- **Khả năng mở rộng** là tính năng cho phép hệ thống Machine Learning đáp ứng nhu cầu tăng trưởng và biến động của khối lượng công việc cần thực hiện. Điều này bao gồm khả năng thực hiện quá trình inference trên nhiều máy chủ, tận dụng sức mạnh tính toán phân tán để xử lý khối lượng dữ liệu lớn và phức tạp. Hơn nữa, hệ thống có thể tự động điều chỉnh cụm máy chủ theo lượng tải hiện tại, đảm bảo hiệu suất tối ưu và sử dụng tài nguyên hiệu quả trong mọi điều kiện hoạt động.

- **Tính bảo mật** là một khía cạnh không thể thiếu trong bất kỳ hệ thống nào. Các hệ thống cần được triển khai các cơ chế xác thực để đảm bảo rằng chỉ những người dùng được ủy quyền mới có thể truy cập vào hệ thống và dữ liệu. Bên cạnh đó việc mã hóa dữ liệu trong quá trình truyền

tải là cần thiết để bảo vệ thông tin nhạy cảm khỏi các mối đe dọa bảo mật tiềm ẩn, đảm bảo tính bảo mật và toàn vẹn của dữ liệu trong suốt quá trình xử lý và truyền tải. /item **Khả năng giám sát** là yếu tố cuối cùng nhưng cũng không kém phần quan trọng trong việc triển khai bất kỳ một hệ thống nào. Điều này bao gồm khả năng theo dõi liên tục hoạt động của hệ thống, ghi nhận các chỉ số vận hành quan trọng như thời gian phản hồi, tỷ lệ sử dụng tài nguyên và độ chính xác của mô hình. Hơn nữa, hệ thống cần có khả năng phát hiện và ghi lại các lỗi một cách hiệu quả, cho phép các nhà phát triển và quản trị viên nhanh chóng xác định và khắc phục sự cố, đảm bảo hiệu suất và độ tin cậy liên tục của hệ thống.

2 Serving Machine Learning model

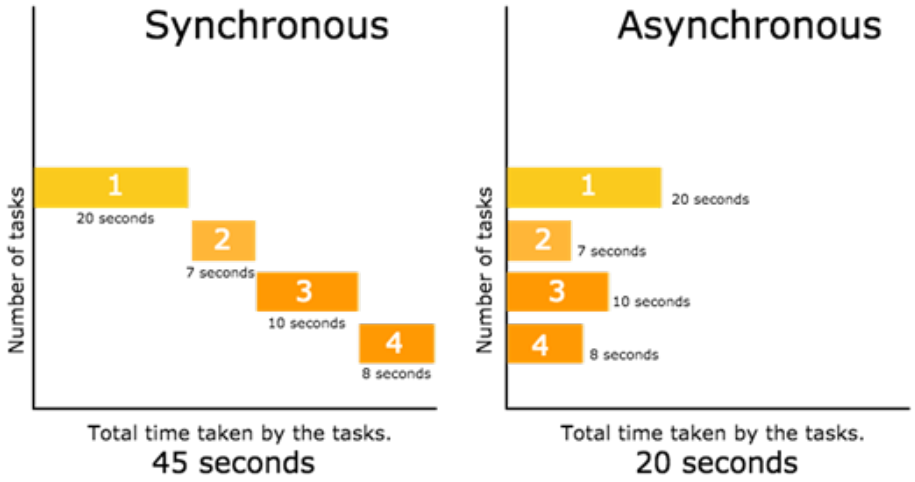


Hình 2: Serving Machine Learning model

Microservice là một kiến trúc trong đó các thành phần của ứng dụng được chia thành nhiều dịch vụ nhỏ, độc lập và có khả năng triển khai riêng biệt. Mỗi microservice tập trung vào một chức năng cụ thể và có thể được phát triển, cập nhật, và mở rộng một cách độc lập mà không ảnh hưởng đến toàn bộ hệ thống. Các microservice thường giao tiếp với nhau thông qua API như REST hoặc gRPC, và có thể được viết bằng nhiều ngôn ngữ lập trình khác nhau. Kiến trúc này mang lại nhiều lợi ích như tăng tính module, cải thiện khả năng mở rộng và linh hoạt trong việc triển khai. Tuy nhiên, nó cũng đặt ra các thách thức về quản lý, giám sát và đảm bảo tính nhất quán dữ liệu giữa các dịch vụ.

2.1 Phương thức giao tiếp Sync và Async

Trong kiến trúc microservice, giao tiếp đồng bộ (Sync) và bất đồng bộ (Async) là hai phương thức quan trọng để các dịch vụ tương tác với nhau. Mỗi phương thức đều có những ưu điểm và hạn chế riêng, và việc lựa chọn phương thức phù hợp đóng vai trò quan trọng trong việc đảm bảo hiệu suất, tính mở rộng và độ tin cậy của hệ thống.
Với giao tiếp đồng bộ, một dịch vụ sẽ gửi yêu cầu đến dịch vụ khác và chờ đợi phản hồi trước khi tiếp tục xử lý. Điều này được thực hiện thông qua các giao thức API phổ biến. Phương thức giao tiếp đồng bộ này dễ hiểu và dễ triển khai, giúp đảm bảo rằng yêu cầu được xử lý theo thứ tự và ngay lập tức. Tuy nhiên, nhược điểm lớn nhất của phương thức này là sự phụ thuộc giữa các dịch vụ, khi một dịch vụ bị chậm hoặc không khả dụng, nó có thể gây ra sự trì hoãn hoặc làm giảm hiệu suất tổng thể của hệ thống.



Hình 3: Phương thức giao tiếp Sync và Async

Trong giao tiếp bất đồng bộ, dịch vụ yêu cầu mà không cần chờ phản hồi ngay lập tức, giúp giảm bớt sự phụ thuộc giữa các dịch vụ. Thay vào đó, các yêu cầu thường được đưa vào hàng đợi hoặc được xử lý ở background, cho phép dịch vụ tiếp tục xử lý công việc khác. Giao tiếp bất đồng bộ giúp hệ thống chịu lỗi tốt hơn và dễ mở rộng hơn, nhưng lại đòi hỏi sự phức tạp trong việc quản lý trạng thái và đồng bộ dữ liệu. Dưới đây là bảng tổng hợp các điểm chính của hai phương thức giao tiếp này:

Tiêu chí	**Giao tiếp đồng bộ (Sync)**	**Giao tiếp bất đồng bộ (Async)**
Phương thức xử lý	Dịch vụ gửi yêu cầu và chờ đợi phản hồi ngay lập tức.	Dịch vụ gửi yêu cầu và không cần chờ phản hồi ngay, xử lý sau qua hàng đợi.
Ưu điểm	- Dễ triển khai, dễ hiểu. - Đảm bảo phản hồi tức thời.	- Tăng tính linh hoạt, khả năng mở rộng tốt hơn. - Hệ thống chịu lỗi cao.
Nhược điểm	- Phụ thuộc vào thời gian phản hồi của dịch vụ khác. - Hiệu suất giảm nếu dịch vụ bị chậm hoặc không khả dụng.	- Phức tạp trong việc quản lý trạng thái và đồng bộ dữ liệu. - Không phản hồi tức thời.
Thích hợp cho	Các trường hợp cần kết quả ngay lập tức, bắt buộc xử lý dữ liệu theo trình tự.	Các trường hợp cần xử lý nhiều yêu cầu không yêu cầu phản hồi tức thì.
Khả năng chịu lỗi	Thấp, do phụ thuộc vào dịch vụ khác.	Cao, không phụ thuộc vào phản hồi tức thời của dịch vụ khác.
Khả năng mở rộng	Giới hạn bởi tốc độ phản hồi của dịch vụ liên quan.	Cao, có thể xử lý nhiều yêu cầu cùng lúc mà không làm chậm hệ thống.

Bảng 1: So sánh giữa Giao tiếp đồng bộ và Giao tiếp bất đồng bộ

Để so sánh hiệu suất giữa giao tiếp đồng bộ (Sync) và bất đồng bộ (Async) trong Python, ta có thể sử dụng thư viện requests cho giao tiếp đồng bộ và aiohttp cho giao tiếp bất đồng bộ. Dưới đây là các

ví dụ đơn giản để gửi một số lượng lớn yêu cầu HTTP và so sánh thời gian thực hiện giữa hai phương thức.

Ví dụ 1: Giao tiếp đồng bộ với **requests**

```python
import requests
import time

NUM_REQUESTS = 10
URL = "https://jsonplaceholder.typicode.com/posts"

def fetch_sync(url):
    response = requests.get(url)
    return response.status_code

start_time = time.time()
for _ in range(NUM_REQUESTS):
    fetch_sync(URL)
end_time = time.time()
print(f"Sync requests took {end_time - start_time:.2f} seconds")

# Sync requests took 2.15 seconds
```

Ví dụ 2: Giao tiếp bất đồng bộ với **aiohttp**

```python
import aiohttp
import asyncio
import time

NUM_REQUESTS = 10
URL = "https://jsonplaceholder.typicode.com/posts"

async def fetch_async(session, url):
    async with session.get(url) as response:
        return response.status

async def main():
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_async(session, URL) for _ in range(NUM_REQUESTS)]
        await asyncio.gather(*tasks)

start_time = time.time()
asyncio.run(main())
end_time = time.time()
print(f"Async requests took {end_time - start_time:.2f} seconds")

# Async requests took 0.16 seconds
```

2.2 Giao thức giao tiếp REST và gRPC

REST và gRPC là hai giao thức giao tiếp phổ biến trong kiến trúc microservice. Chúng cho phép các dịch vụ có thể kết nối và giao tiếp với nhau. Mỗi giao thức có những đặc điểm riêng về cách thức giao tiếp, hiệu suất và trường hợp sử dụng.

REST là giao thức mà ở đó dữ liệu được truyền thông qua các định dạng như JSON hoặc XML. Giao thức này sử dụng HTTP để truyền tải dữ liệu cùng với các phương thức như GET, POST, PUT, DELETE để thao tác dữ liệu.

gRPC là một giao thức được phát triển bởi Google, sử dụng giao thức HTTP 2.0 để truyền tải và Protobuf để mô tả cấu trúc dữ liệu dùng để giao tiếp giữa client và server. Nhờ Protobuf mà dữ liệu

giao tiếp giữa các dịch vụ trở nên nhỏ hơn và do đó giúp việc giao tiếp giữa các dịch vụ trở nên nhanh hơn. Tuy nhiên, gRPC lại không thân thiện khi giao tiếp với các ứng dụng web.

Tiêu chí	**REST**	**gRPC**
Giao thức sử dụng	HTTP/1.1 hoặc HTTP/2	HTTP/2
Định dạng dữ liệu	JSON hoặc XML	Protobuf
Tính dễ sử dụng	Đơn giản, phổ biến và dễ dàng tích hợp với nhiều hệ thống	Khó sử dụng hơn, cài đặt phức tạp hơn
Phương thức giao tiếp	Chỉ hỗ trợ Unary, tức là nhận yêu cầu từ client và gửi về phản hồi tương ứng	Hỗ trợ đa dạng: Unary, Server Streaming, Client Streaming, Bidirectional Streaming
Môi trường sử dụng	Ứng dụng được hầu hết tất cả môi trường có hỗ trợ JSON, đặc biệt là web	Không thân thiện với môi trường web. Phù hợp với giao tiếp các dịch vụ nội bộ trong microservice

Bảng 2: So sánh giữa REST và gRPC

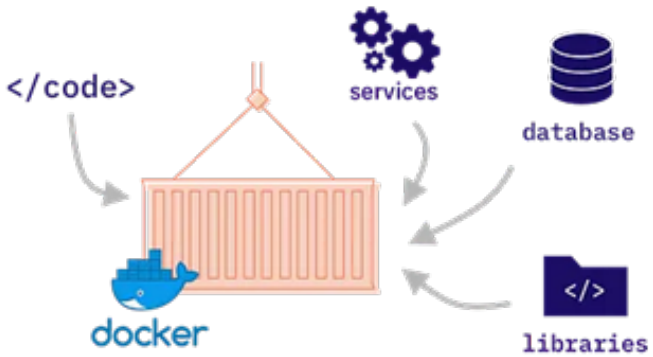
2.3 Giới thiệu về FastAPI

FastAPI là một framework Python phổ biến dùng để xây dựng các API. So với các thư viện tương tự như Flask hay Django thì FastAPI được đánh giá cao về tốc độ và dễ tiếp cận.
So với Flask, FastAPI có các tính năng hỗ trợ trực tiếp cho các phép xử lý bất đồng bộ, giúp API có khả năng xử lý nhiều yêu cầu cùng lúc mà không cần phải chờ các tác vụ hoàn thành tuần tự. Ngoài ra FastAPI còn được tích hợp sẵn với OpenAPI và JSON Schema, giúp tự động tạo tài liệu API mà không cần bất kỳ cấu hình phức tạp nào khác.
Ta có ví dụ sau đây để giới thiệu cơ bản cách sử dụng FastAPI.

```python
from fastapi import FastAPI

# Tạo ứng dụng FastAPI
app = FastAPI()

# Tạo một endpoint GET
@app.get("/")
def read_root():
    return {"message": "Welcome to FastAPI"}

# Tạo một endpoint GET có tham số
@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}

# Khởi chạy ứng dụng với Uvicorn nếu chạy trực tiếp tệp này
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

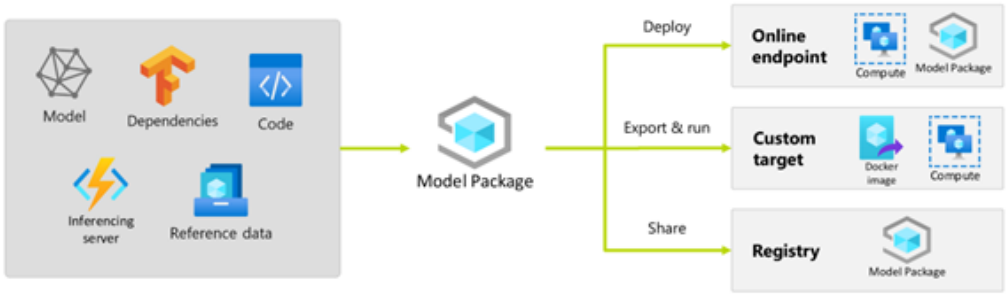
Sau khi chạy đoạn code này, ta có thể truy cập vào địa chỉ http://localhost:8000 và sẽ thấy JSON trả về là `{"message": "Welcome to FastAPI"}`. Nếu truy cập vào http://localhost:8000/items/1?q=test, API sẽ trả về `{"item_id": 1, "q": "test"}`.
Ngoài ra, FastAPI còn tự động tạo tài liệu cho API. Để xem được tài liệu, ta cần truy cập vào địa chỉ http://localhost:8000/docs.

3 Packaging the ML code



Hình 4: Packaging the ML code

Trong phát triển phần mềm, việc đóng gói phần mềm đóng vai trò vô cùng quan trọng vì nó giúp chuẩn hóa quy trình phân phối và triển khai phần mềm. Đóng gói đảm bảo rằng tất cả các thành phần của phần mềm, bao gồm mã nguồn, thư viện, tài nguyên và cấu hình, đều được gói gọn trong một cấu trúc có tổ chức, dễ dàng quản lý và tái sử dụng. Điều này giúp giảm thiểu rủi ro khi triển khai trên các môi trường khác nhau, đảm bảo rằng phần mềm hoạt động nhất quán bất kể hệ điều hành hay các phiên bản thư viện khác nhau. Đặc biệt, đóng gói giúp hỗ trợ việc bảo trì, nâng cấp và sửa lỗi dễ dàng hơn, đồng thời cải thiện tính khả chuyển và khả năng mở rộng của phần mềm. Công cụ được sử dụng phổ biến trong đóng gói phần mềm là Docker.



Trong phát triển và triển khai các mô hình Machine Learning, việc đóng gói không chỉ tập trung đóng gói mã nguồn và các thư viện như trong phát triển phần mềm truyền thống, mà còn bao gồm việc quản lý và đóng gói mô hình hoặc dữ liệu. Sự khác biệt chính giữa đóng gói trong Machine Learning và phần mềm truyền thống nằm ở tính động và phức tạp của mô hình và dữ liệu. Mô hình Machine Learning có thể cần được huấn luyện lại khi dữ liệu mới xuất hiện, và việc theo dõi các phiên bản của mô hình, dữ liệu huấn luyện cũng như các tham số là rất quan trọng. Đóng gói Machine Learning không chỉ đảm bảo mã nguồn và môi trường nhất quán mà còn cần quản lý phiên bản của mô hình. Điều này đòi hỏi sự kết hợp của nhiều công cụ khác nhau như MLflow hoặc DVC để quản lý toàn bộ quy trình, từ huấn luyện đến triển khai, đảm bảo mô hình hoạt động ổn định và có thể tái tạo khi triển khai trong môi trường thực tế.

4 Giới thiệu về Docker

Docker là một nền tảng cho phép xây dựng, đóng gói và triển khai ứng dụng một cách nhất quán và hiệu quả trên các môi trường khác nhau. Thay vì lo ngại về sự khác biệt giữa các hệ điều hành hoặc cấu hình môi trường, Docker gói gọn phần mềm thành một đơn vị là container trong đó chứa tất cả các thành phần để triển khai phần mềm (thư viện, công cụ và cấu hình). Mỗi container hoạt ododnjg giống như một môi trường ảo hóa, đảm bảo rằng ứng dụng se chạy một cách nhất quán dù được triển khai ở bất cứ đâu. Docker giúp giảm thiểu các vấn đề về "phù hợp môi trường"và tăng cường tính khả chuyển, hiệu suất, và hiệu quả trong quá trình phát triển, kiểm thử, và triển khai phần mềm.
Để cài đặt docker ta tham khảo hướng dẫn tại https://docs.docker.com/compose/install/.
Ta có một ví dụ đóng gói đơn giản sau đây để mô tả các thành phần chính và tính tiện dụng của Docker.
Ví dụ: Xây dựng một dịch vụ mà khi ta gửi yêu cầu sẽ trả ra cho ta một đoạn JSON có nội dung là ***{"message": "Hello, Docker with FastAPI!"}***. Cây thư mục của ví dụ như sau:

```
fastapi-docker-app/     # Thư mục gốc của dự án
|
|-- app.py              # Mã nguồn của ứng dụng FastAPI
|-- requirements.txt    # Danh sách các thư viện phụ thuộc
|-- Dockerfile          # Định nghĩa cách đóng gói ứng dụng trong Docker
```

Nội dung file **app.py**:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Hello, Docker with FastAPI!"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Nội dung file **requỉrements.txt**:

```
fastapi
uvicorn
```

Ta có file tên là **Dockerfile** được sử dụng để định nghĩa cách đóng gói ứng dụng FastAPI vào container.
Nội dung file **Dockerfile**:

```
# Sử dụng Python 3.9 làm hình ảnh nền
FROM python:3.9-slim
# Đặt thư mục làm việc trong container
WORKDIR /app
# Sao chép file requirements.txt vào container
COPY requirements.txt requirements.txt
# Cài đặt các thư viện cần thiết
RUN pip install -r requirements.txt
# Sao chép toàn bộ mã nguồn vào container
COPY . .
# Mở cổng 8000 (cổng mặc định của FastAPI)
EXPOSE 8000
# Câu lệnh để chạy ứng dụng
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Để đóng gói ứng dụng này ta sử dụng câu lệnh **docker build**

```
docker build -t fastapi-docker-app .
```

Giải thích:

- `docker build`: Tạo docker image từ file Dockerfile.
- `-t fastapi-docker-app`: Gán tên cho image là `fastapi-docker-app`.
- Dấu `"."`: Đặt build context cho Dockerfile ở thư mục hiện tại.

Tiếp theo, ta chạy ứng dụng thông qua image đã được đóng gói trước đó bằng câu lệnh docker run.

```
docker run -p 8000:8000 fastapi-docker-app
```

Giải thích:

- `docker run`: Khởi chạy một container từ image đã tạo.
- `-p 8000:8000`: Mở cổng 8000 trên máy chủ và ánh xạ với cổng 8000 của container, để truy cập ứng dụng qua địa chỉ http://localhost:8000.
- `fastapi-docker-app`: Tên của Docker image đã tạo.

Ta sử dụng câu lệnh `docker ps` để kiểm tra xem container có đang chạy hay không. Tiếp theo, tại trình duyệt web ta truy cập vào địa chỉ http://localhost:8000 để kiểm tra xem ứng dụng có chạy đúng như mong đợi không.

5 Giới thiệu về DVC

DVC hay Data Version Control là công cụ giúp quản lý phiên bản cho dữ liệu hoặc các pipeline. Nó có chức năng tương tự như git, nhưng dành cho dữ liệu, mô hình và pipeline trong các dự án Machine Learning. DVC giúp ta kiểm soát và theo dõi các phiên bản của dữ liệu đầu vào, mô hình, và kết quả trong suốt quá trình phát triển, từ lúc huấn luyện mô hình đến lúc triển khai.
Các tính năng chính của DVC:

1. **Quản lý phiên bản dữ liệu và mô hình**: DVC cho phép theo dõi sự thay đổi của dữ liệu, mô hình và giúp lưu lại từng phiên bản cụ thể để dễ dàng tái sử dụng và tái hiện thí nghiệm.
2. **Tích hợp với Git**: DVC hoạt động song song với Git để quản lý mã nguồn mà không cần sử dụng tới Git LFS.
3. **Quản lý pipeline**: DVC giúp quản lý toàn bộ quy trình phát triển mô hình, từ tải dữ liệu, xử lý dữ liệu, huấn luyện mô hình đến triển khai. Điều này đảm bảo rằng mỗi bước trong pipeline được kiểm soát và có thể tái tạo.
4. **Lưu trữ đám mây**: DVC hỗ trợ lưu trữ dữ liệu trên các hệ thống đám mây như AWS S3, Google Drive, Azure Blob Storage, hoặc thậm chí là hệ thống file cục bộ. Điều này giúp quản lý dữ liệu lớn một cách dễ dàng mà không phải lưu trữ tất cả trong hệ thống máy cục bộ.
5. **Tái hiện thí nghiệm**: DVC cho phép dễ dàng tái hiện lại các thí nghiệm học máy dựa trên dữ liệu và mô hình đã được quản lý phiên bản.

Ta thể hiện tính năng lưu trữ đám mây và quản lý phiên bản của DVC thông qua ví dụ dưới đây.
Ví dụ: Ta có một file trọng số mô hình `model.pth` và mong muốn file mô hình này có thể được quản lý phiên bản.
Trước tiên ta cài đặt DVC thông qua lệnh

```
pip install dvc
```

Sau đó, ta khởi tạo DVC trong một dự án mà ở đó đã được theo dõi bằng git(nếu chưa thì cần khởi tạo git, lưu ý: bạn phải cài đặt git trước).

```
# git init
dvc init
touch .gitignore
git add .dvc .gitignore
git commit -m "Initialize DVC"
```

Tiếp theo, ta sử dụng DVC để theo dõi file model.pth

```
dvc add model.pth
git add model.pth.dvc .gitignore
git commit -m "Add model.pth to DVC"
```

Lệnh **dvc add** sẽ tạo ra một file **.dvc** (trong trường hợp này là **model.pth.dvc**). File này sẽ được theo dõi bởi Git, trong khi file mô hình thực tế sẽ được lưu trữ ở một nơi khác và được quản lý bằng DVC. Nơi lưu trữ có thể là các dịch vụ lưu trữ đám mây như Amazon S3, Google Drive, Azure Blob Storage. Trong ví dụ này, ta sẽ lấy ví dụ lưu trữ bằng S3. Để thiết lập, ta cần thêm remote vào DVC như sau:

```
dvc remote add -d mys3 s3://<remote-url>
```

Sau đó, ta cần bổ sung thêm các credential như ACCESS_KEY hoặc SECRET_KEY để có quyền truy cập tới S3.

```
dvc push -r mys3
```

Như vậy là ta đã thành công lưu trữ và quản lý phiên bản file mô hình model.pth thông qua DVC. Nhờ thiết lập này, ta có thể dễ dàng truy cập cả source code để chạy inference mô hình và phiên bản model tương ứng ở các máy khác nhau, giúp tăng khả năng phối hợp với cả nhóm cũng như dễ dàng quản lý phiên bản mô hình.