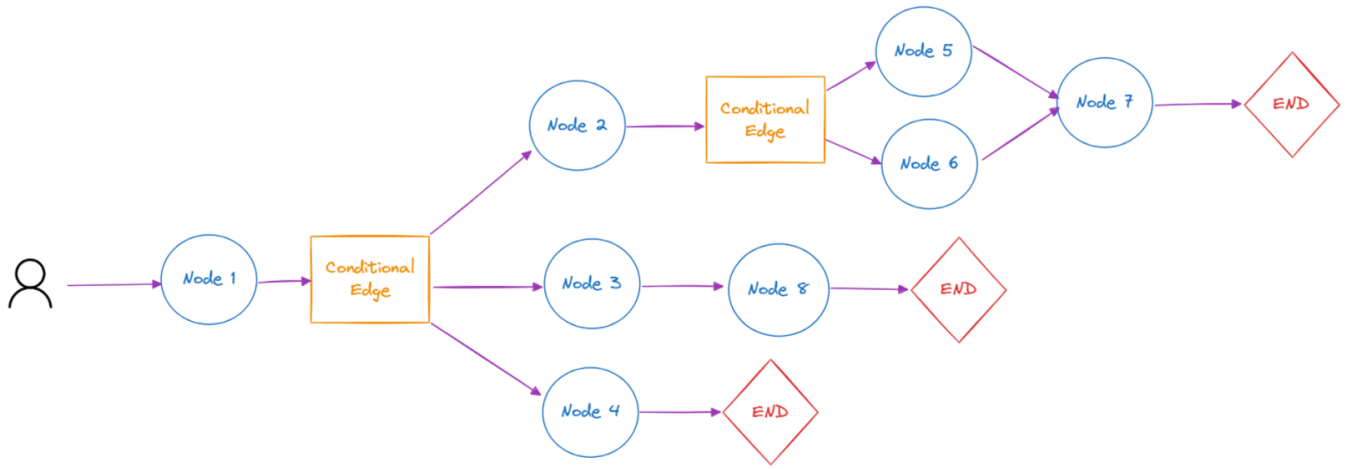
Giới thiệu về LangGraph

Dưới đây là một bài viết giới thiệu về **LangGraph**, được trình bày theo kiểu bài viết chia sẻ kiến thức công nghệ, phù hợp cho tài liệu kỹ thuật.

Giới thiệu về LangGraph

LangGraph Tạo LLM Workflow dễ dàng như vẽ đồ thị

Trong thời đại của trí tuệ nhân tạo và mô hình ngôn ngữ lớn (LLM), việc xây dựng các workflow phức tạp để xử lý ngôn ngữ tự nhiên đang trở nên ngày càng quan trọng. Tuy nhiên, quản lý các chuỗi tác vụ, rẽ nhánh logic, và vòng lặp trong quá trình tương tác với LLM có thể nhanh chóng trở nên phức tạp. Đây chính là lý do **LangGraph** ra đời.



LangGraph là gì?

LangGraph là một thư viện mã nguồn mở được phát triển dựa trên **LangChain**, cho phép bạn xây dựng các **đồ thị trạng thái (stateful graphs)** cho các ứng dụng sử dụng LLM. Thay vì chỉ chạy một chuỗi cố định các bước, LangGraph cho phép bạn xây dựng các quy trình linh hoạt hơn, bao gồm:

- Rẽ nhánh điều kiện (conditional branching)
- Vòng lặp (looping)
- Giao tiếp hai chiều giữa người dùng và LLM
- Quản lý trạng thái qua nhiều bước xử lý

LangGraph mang lại cách tiếp cận **kết hợp giữa Lập trình khai báo(**declarative**) và Lập trình mệnh lệnh(**imperative**)**, giúp bạn dễ dàng hình dung và kiểm soát luồng dữ liệu và trạng thái trong quá trình xử lý.

Vì sao nên sử dụng LangGraph?

Dễ hiểu, dễ hình dung

Bạn xây dựng workflow như một **đồ thị gồm các nút (nodes)**, mỗi nút thực hiện một tác vụ cụ thể, như gọi API, xử lý dữ liệu, tương tác với LLM, hay rẽ nhánh theo điều kiện.

Tái sử dụng và mở rộng tốt

Mỗi nút trong LangGraph là một function, nên bạn dễ dàng viết lại, mở rộng hoặc chia sẻ logic giữa các ứng dụng khác nhau.

Tích hợp chặt chẽ với LangChain

LangGraph kế thừa sức mạnh từ LangChain nên bạn có thể dễ dàng kết hợp với các **PromptTemplate**, **Agents**, **Memory**, hay **Tools** trong hệ sinh thái LangChain.

Cách hoạt động của LangGraph

Cấu trúc cơ bản của một LangGraph bao gồm:

- **State**: Biến lưu trữ dữ liệu cần thiết trong quá trình chạy đồ thị.
- **Node**: Hàm xử lý dữ liệu (có thể là gọi LLM, tính toán, rẽ nhánh...).
- **Edges**: Kết nối các node theo logic định sẵn.

Ví dụ đơn giản về đồ thị gồm 3 node:

```
from langgraph.graph import StateGraph, END

# Định nghĩa trạng thái
class MyState(TypedDict):
    input: str
    output: str

# Định nghĩa node
def process_input(state):
    result = state["input"].upper()
    return {"output": result}

# Tạo đồ thị
```

```
builder = StateGraph(MyState)
builder.add_node("process", process_input)
builder.set_entry_point("process")
builder.set_finish_point("process", END)
graph = builder.compile()

# Chạy đồ thị
result = graph.invoke({"input": "hello"})
print(result)
```

Ứng dụng thực tế

LangGraph rất phù hợp cho các bài toán phức tạp với LLM như:

- Chatbot nhiều bước với trạng thái duy trì qua từng lượt chat
- Tóm tắt tài liệu theo yêu cầu với logic tùy chỉnh
- Agent phức tạp với hành vi có điều kiện
- Hệ thống tư vấn, hỏi đáp với nhiều nhánh logic

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Vì sao LangGraph ra đời và Graph có ý nghĩa gì

Khi các ứng dụng xây dựng trên mô hình ngôn ngữ lớn (LLM) ngày càng trở nên phức tạp – từ chatbot đơn giản đến hệ thống tự động hóa tác vụ – thì nhu cầu về **cách điều phối luồng suy nghĩ, trạng thái, và hành động của LLMs** cũng tăng theo.

Đó là lý do vì sao **LangGraph** ra đời.

LangGraph là một **framework kết hợp LLMs với cấu trúc đồ thị trạng thái (stateful graphs)**, giúp tạo ra các ứng dụng có khả năng:

- Gọi nhiều công cụ khác nhau
- Lặp lại hành động nếu cần thiết
- Chuyển hướng luồng xử lý dựa trên kết quả
- Giữ được trạng thái trong suốt quá trình suy luận

I. Tại sao lại là LangGraph?

1. Giới hạn của mô hình agent truyền thống (LangChain Agents)

Các agents trong LangChain rất linh hoạt, nhưng đôi khi:

- Thiếu khả năng kiểm soát luồng suy luận
- Khó kiểm tra và debug khi có nhiều bước
- Thiếu khả năng giữ trạng thái rõ ràng (statefulness)
- Gặp khó khăn với các vòng lặp (loop), nhánh (branch), hoặc điều kiện lặp lại

Kết quả là khi xây dựng các ứng dụng phức tạp như AutoGPT, trợ lý đa tác vụ, hệ thống hỏi đáp nhiều bước... developer phải viết rất nhiều code xử lý luồng hoặc hack workaround.

2. LangGraph: Giải pháp điều phối LLM theo kiểu đồ thị trạng thái

LangGraph đưa ra một kiến trúc mới, nơi **quá trình suy nghĩ và hành động của LLM được mô hình hóa như một đồ thị (graph)**.

Mỗi node trong đồ thị:

- Có thể là một hành động cụ thể (gọi API, suy nghĩ, hỏi người dùng…)
- Có thể gọi LLM để đưa ra quyết định chuyển hướng (routing)

Lợi ích chính:

Tính năng	Ý nghĩa
Có trạng thái	Lưu giữ thông tin giữa các bước
Có vòng lặp	Hỗ trợ dễ dàng quá trình phản xạ (reflection), retry, sửa lỗi
Modular	Mỗi node độc lập, dễ tái sử dụng
Dễ debug	Có thể quan sát từng node và hướng đi trong quá trình xử lý
Tùy chọn luồng	Có thể dùng LLM để quyết định đi hướng nào trong graph

II. Tại sao lại dùng từ “Graph”?

1. Graph = Đồ thị trạng thái có hướng

LangGraph sử dụng **lý thuyết đồ thị (directed graph)** như một cách để **mô hình hóa quy trình tư duy** của một agent.

Trong graph:

- **Node** là một bước trong quy trình (ví dụ: "Phân tích yêu cầu", "Tìm kiếm trên web", "Tóm tắt dữ liệu", "Phản hồi")
- **Edge** là hướng đi từ một bước đến bước kế tiếp, có thể **phụ thuộc vào điều kiện** (ví dụ: nếu LLM nói "Chưa đủ dữ liệu", hãy quay lại bước tìm kiếm)

Graph này **giống như flowchart thông minh**, nhưng được điều khiển bởi LLM.

2. Graph giúp dễ hình dung quy trình phức tạp

Dưới dạng graph, developer có thể:

- Nhìn thấy toàn bộ hành trình của agent
- Điều chỉnh từng bước mà không ảnh hưởng đến toàn hệ thống
- Thêm logic phức tạp (loop, điều kiện, xử lý lỗi) một cách trực quan

Ví dụ:

```
[User Input]
   ↓
[LLM hiểu yêu cầu]
   ↓
[Chọn hành động] ──→ [Dùng công cụ] ──→ [Tóm tắt kết quả]
   ↑                          ↓
 [Thiếu dữ liệu?] ←──────[Kiểm tra đủ chưa?]
```

III. Khi nào nên dùng LangGraph?

LangGraph **cực kỳ phù hợp** khi bạn xây dựng:

Multi-step reasoning (suy luận nhiều bước)
Agents có khả năng reflection / retry / repair
Workflow có trạng thái (stateful flows)
Hệ thống cần kiểm soát logic chặt chẽ
Quy trình có loop, branching hoặc fallback logic

Ví dụ ứng dụng:

- Trợ lý hỏi đáp tài liệu nhiều bước (hay bị "không đủ dữ liệu")
- Tác nhân viết báo cáo (chia nhỏ tác vụ và lặp lại từng phần)
- QA system với logic xác minh độ chính xác
- Tác nhân phân tích dữ liệu và tự sửa lỗi truy vấn

❓ Câu hỏi	✅ Trả lời
Vì sao LangGraph?	Vì các ứng dụng LLM ngày càng phức tạp, cần khả năng điều phối, giữ trạng thái, và xử lý logic phức tạp.
Vì sao "Graph"?	Vì LangGraph mô hình hóa tư duy của agent dưới dạng **đồ thị trạng thái có hướng** — giúp kiểm soát, mở rộng, debug dễ dàng hơn.

Lập trình khai báo(declarative) và Lập trình mệnh lệnh(imperative)

Lập trình khai báo(declarative) và Lập trình mệnh lệnh(imperative) là hai **phong cách lập trình** khác nhau — chúng phản ánh **cách bạn mô tả công việc mà máy tính cần thực hiện**

I. Lập trình khai báo(declarative)

- **Bạn nói cho máy tính biết *bạn muốn đạt được gì***.
- Tập trung vào **cái gì** là kết quả cuối cùng.
- Máy tính sẽ tự lo cách thực hiện.

```
# Declarative (giống kiểu functional programming)
result = [x * 2 for x in data if x > 10]
```

Hoặc ví dụ quen thuộc hơn:

```
SELECT * FROM users WHERE age > 18;
```

Bạn không nói “lặp qua từng dòng, kiểm tra tuổi”, mà chỉ nói “hãy lấy dữ liệu theo điều kiện”.

II. Lập trình mệnh lệnh(imperative)

- **Bạn nói cho máy tính biết *từng bước phải làm gì***.
- Tập trung vào **cách** để đạt được mục tiêu.
- Ví dụ: viết một vòng lặp `for`, gọi hàm này rồi hàm kia, kiểm tra điều kiện...

```
# Imperative
result = []
for x in data:
    if x > 10:
```

```
        result.append(x * 2)
```

Bạn phải quản lý từng bước cụ thể.

III. Trong LangGraph thì sao?

LangGraph **kết hợp cả hai phong cách**:

- Bạn **khai báo các node và mối quan hệ giữa chúng** (kiểu **Lập trình khai báo(declarative)** – giống vẽ sơ đồ luồng).
- Nhưng bên trong mỗi node, bạn viết **các bước cụ thể xử lý dữ liệu** (kiểu **Lập trình mệnh lệnh(imperative)** – như viết hàm bình thường).

Ví dụ:

```
builder.add_node("process", process_input)  # declarative
def process_input(state):                   # imperative
    return {"output": state["input"].upper()}
```

Tóm lại:

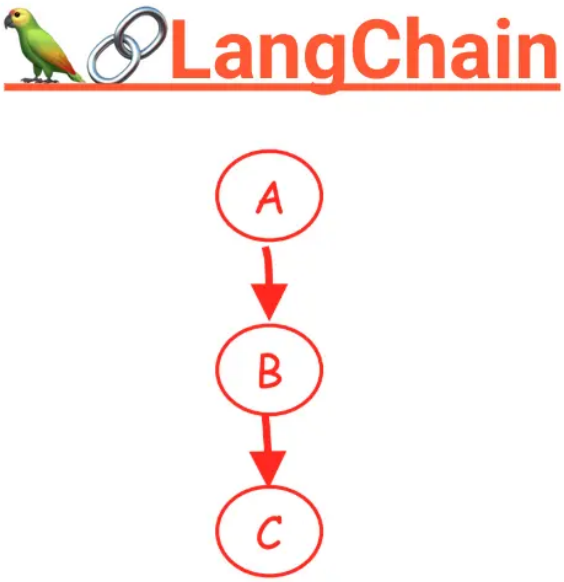
LangGraph giúp bạn tận dụng **sức mạnh kết hợp**: vừa dễ hình dung (declarative), vừa dễ điều khiển (imperative).

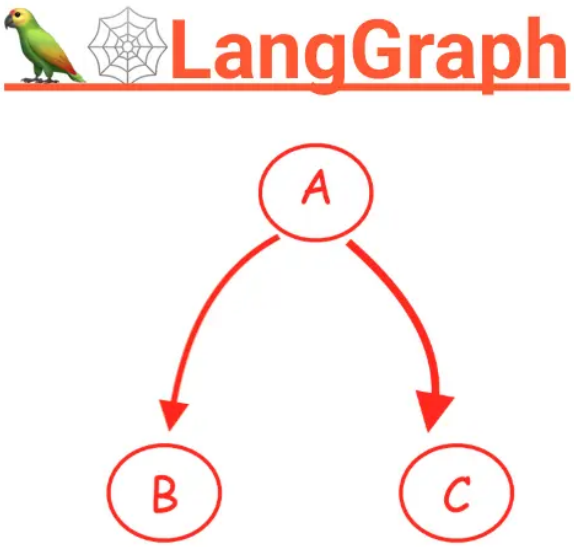
Phong cách	Bạn mô tả...	Ưu điểm	Nhược điểm
Imperative	CÁCH máy tính phải làm	Kiểm soát chi tiết	Dễ rối với quy trình phức tạp
Declarative	CÁI GÌ bạn muốn đạt được	Gọn, dễ hiểu	Khó kiểm soát từng bước nhỏ

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

So sánh LangGraph và LangChain





Với sự phát triển nhanh chóng của các mô hình ngôn ngữ lớn như GPT, Claude, Mistral..., các thư viện như **LangChain** và **LangGraph** ra đời để giúp lập trình viên dễ dàng xây dựng ứng dụng sử dụng LLM. Nhưng giữa hai cái tên quen thuộc này, bạn nên chọn cái nào? Hoặc khi nào nên dùng cả hai?

Bài viết này sẽ giúp bạn hiểu rõ **sự khác biệt, điểm mạnh, và cách sử dụng** của **LangChain** và **LangGraph**.

I. Tổng quan

Đặc điểm	LangChain	LangGraph
Mục tiêu	Tạo **pipeline** (chuỗi) LLM logic	Tạo **workflow** (đồ thị trạng thái) phức tạp
Cấu trúc	Tuyến tính (linear chain / agent)	Đồ thị trạng thái có vòng lặp, rẽ nhánh
Mô hình dữ liệu	Stateless (không lưu trạng thái)	Stateful (quản lý trạng thái qua nhiều bước)
Tích hợp LLM	Rất mạnh, nhiều tools tích hợp	Kế thừa từ LangChain
Độ linh hoạt	Cao với chuỗi đơn giản hoặc agents	Rất cao với workflow phức tạp

Đặc điểm	LangChain	LangGraph
Sử dụng chính	Chatbot, Q&A, Retrieval, Agents	Bot đa bước, phân nhánh logic, memory flows

II. Kiến trúc hoạt động

LangChain

LangChain cung cấp một tập hợp các **abstractions** như:

- `LLMChain`: Gọi một LLM với prompt
- `Chain`: Kết hợp nhiều bước xử lý nối tiếp nhau
- `Agent`: Cho phép mô hình quyết định hành động tiếp theo
- `Memory`: Lưu trạng thái cuộc hội thoại tạm thời

Mỗi chain là **tuyến tính** — dữ liệu đi từ đầu đến cuối qua từng bước.

Ưu điểm: Dễ dùng, dễ hiểu, setup nhanh.
Nhược điểm: Hạn chế khi bạn muốn rẽ nhánh, lặp lại, hoặc xử lý logic phức tạp.

LangGraph

LangGraph đưa kiến trúc lên tầm cao hơn: bạn không chỉ nối các bước, bạn **vẽ ra một đồ thị trạng thái**. Trong đó:

- Mỗi node là một bước xử lý
- Bạn có thể rẽ nhánh tùy vào điều kiện
- Có thể lặp lại một node hoặc quay lại node trước
- **Trạng thái được lưu và truyền qua toàn bộ đồ thị**

LangGraph kế thừa toàn bộ các công cụ từ LangChain như `LLMChain`, `PromptTemplate`, `Tool`, `Memory`, `Agent`,... nhưng nâng cấp khả năng tổ chức luồng xử lý lên **mức workflow engine**.

Ưu điểm: Xử lý logic phức tạp, hỗ trợ đa chiều, có vòng lặp.
Nhược điểm: Khó hơn để bắt đầu, yêu cầu hiểu về state và luồng.

III. Khi nào dùng LangChain?

Dùng **LangChain** khi bạn:

- Chỉ cần chạy một chuỗi các bước (pipeline)
- Viết một chatbot đơn giản
- Làm Q&A với Retriever
- Gọi LLM để xử lý văn bản đầu vào và trả lại đầu ra ngay lập tức

- Không cần quản lý nhiều trạng thái phức tạp

Ví dụ: Lấy nội dung từ một file, tóm tắt nó bằng GPT, rồi gửi email.

IV. Khi nào dùng LangGraph?

Dùng **LangGraph** khi bạn:

- Cần xây dựng các **hệ thống xử lý nhiều bước có rẽ nhánh**
- Muốn xây chatbot đa luồng hội thoại (conversation flows)
- Có logic phức tạp: vòng lặp, kiểm tra điều kiện, lùi lại bước trước
- Quản lý trạng thái giữa nhiều bước, hoặc muốn lưu vào database
- Xây dựng agent với logic phức tạp hơn `ReAct`

Ví dụ:
Hệ thống kiểm tra đơn hàng:

1. Người dùng gửi mã đơn →
2. Hệ thống kiểm tra trạng thái đơn →
3. Nếu đơn đang giao: gửi thông báo;
4. Nếu đơn bị huỷ: hỏi lý do → ghi log → gửi xác nhận.

Bạn khó làm điều này gọn gàng với LangChain, nhưng **LangGraph làm rất tốt**.

V. Có thể dùng kết hợp không?

Hoàn toàn có thể!
LangGraph thực tế **được xây dựng trên nền LangChain**, nên bạn có thể dùng `LLMChain`, `RetrievalChain`, `Agents`, `PromptTemplate` bên trong các node của LangGraph.

Ví dụ: Một node trong đồ thị LangGraph có thể gọi một LangChain Agent để xử lý một phần công việc.

VI. So sánh ngắn gọn qua ví dụ

LangChain (linear):

```
chain = SimpleSequentialChain(chains=[chain1, chain2])
result = chain.run("Hello world")
```

LangGraph (graph):

```
builder = StateGraph()
builder.add_node("step1", step1)
builder.add_node("step2", step2)
builder.add_edge("step1", "step2")
builder.set_entry_point("step1")
graph = builder.compile()
graph.invoke({"input": "Hello world"})
```

VII. Kết luận

Nếu bạn cần...	Hãy chọn...
Xây dựng nhanh một chuỗi xử lý	◫ LangChain
Quản lý nhiều trạng thái phức tạp	◫ LangGraph
Làm chatbot đơn giản	◫ LangChain
Làm agent với hành vi linh hoạt	◫ LangGraph
Kết hợp các công cụ và logic rẽ nhánh	◫ LangGraph

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Tác nhân điều phối(Agentic) trong ứng dụng mô hình ngôn ngữ lớn (LLMs)

Với sự phát triển mạnh mẽ của các mô hình ngôn ngữ lớn (LLMs), chúng ta có thể xây dựng nhiều loại ứng dụng khác nhau. Trong đó, hai hướng tiếp cận nổi bật là:

- **Non-Agentic LLM Apps** – ứng dụng không có tác nhân điều phối
- **Agentic LLM Apps** – ứng dụng có "tác nhân" (agent) tự điều phối hành động

Vậy đâu là sự khác biệt, điểm mạnh – điểm yếu, và bạn nên chọn loại nào cho dự án của mình?

I. Định nghĩa

1. Non-Agentic LLM App là gì?

Đây là các ứng dụng **chạy theo logic tuyến tính**, được xác định rõ ràng. Mỗi bước được lập trình cụ thể, LLM chỉ làm từng nhiệm vụ một cách tường minh.

Đặc điểm:

- Không có khả năng tự ra quyết định.
- Luồng xử lý cố định.
- Phù hợp với các tác vụ đơn giản, lặp lại.

Ví dụ:

- Tóm tắt văn bản.
- Dịch ngôn ngữ.
- Chatbot hỗ trợ theo kịch bản cố định.
- Lấy dữ liệu từ cơ sở kiến thức rồi trả lời.

```
summary = llm("Tóm tắt đoạn văn sau...")
```

2. Agentic LLM App là gì?

Là ứng dụng mà trong đó **LLM hoạt động như một "agent" (tác nhân)** — có khả năng:

- **Ra quyết định**
- **Lập kế hoạch**
- **Chọn công cụ phù hợp để thực thi nhiệm vụ**
- **Tự kiểm tra lại kết quả** và tiếp tục cho đến khi hoàn thành

Đặc điểm:

- Có khả năng tư duy nhiều bước.
- Có thể gọi nhiều công cụ khác nhau.
- Tự động hóa quy trình phức tạp.

Ví dụ:

- Một trợ lý AI có thể:
  - Tìm kiếm trên Google,
  - Lấy dữ liệu từ API,
  - Phân tích dữ liệu,
  - Gửi email báo cáo → mà bạn chỉ cần ra lệnh: **“Tóm tắt số liệu bán hàng tuần này và gửi qua email cho sếp.”**

II. So sánh chi tiết

Đặc điểm	Non-Agentic	Agentic
Mục tiêu	Xử lý 1 nhiệm vụ đơn giản hoặc theo kịch bản	Tự động hóa nhiều bước, xử lý tác vụ mở
Luồng xử lý	Tuyến tính, xác định trước	Rẽ nhánh, điều kiện, có thể điều phối vòng lặp
Khả năng lập kế hoạch	Không	**Có (có thể lên kế hoạch và thực thi từng bước)**
Khả năng chọn công cụ phù hợp	Không	**Có thể chọn tool/action phù hợp theo ngữ cảnh**
Tính linh hoạt	Trung bình – cao	Rất cao
Dễ debug, kiểm soát	**Rất dễ**	Phức tạp hơn do có hành vi tự động
Hiệu suất	Nhanh hơn, ít token hơn	Chậm hơn, tốn token hơn do phải lập kế hoạch
Khả năng sử dụng thực tế	Rất tốt với quy trình ổn định	Tốt với quy trình phức tạp hoặc cần tự thích nghi

3. Minh họa đơn giản

Tác vụ: “Hãy tìm thông tin về Elon Musk và tóm tắt”

Non-Agentic:

- Gửi prompt: “Tóm tắt thông tin về Elon Musk”
- GPT trả lời bằng những gì đã học.

Agentic:

- Agent tự chọn hành động:
  1. Gọi công cụ `WebSearch`
  2. Lấy 3 kết quả đầu
  3. Tóm tắt chúng
  4. Kiểm tra nếu có thông tin mới → bổ sung → xuất bản kết quả

4. Công nghệ hỗ trợ

Tính năng	Non-Agentic	Agentic
Thư viện phổ biến	LangChain, Transformers	LangChain Agents, LangGraph, AutoGPT
Công cụ gọi ngoài (Tools)	Có, nhưng gọi tường minh	LLM tự chọn công cụ thông minh
Bộ nhớ (Memory)	Đơn giản hoặc không có	Quan trọng để theo dõi trạng thái
Loop / Retry logic	Phải lập trình thủ công	Agent có thể tự thực hiện

5. Khi nào nên dùng loại nào?

Trường hợp	Nên chọn
Xử lý một bước (dịch, tóm tắt, phân loại)	□ Non-Agentic
Luồng cố định, logic rõ ràng	□ Non-Agentic
Tác vụ phức tạp, cần suy luận nhiều bước	□ Agentic
Cần tương tác nhiều hệ thống / API / Tool	□ Agentic
Trợ lý AI hoặc hệ thống hỗ trợ quyết định	□ Agentic
Ưu tiên tốc độ, đơn giản, tiết kiệm tài nguyên	□ Non-Agentic

6. Tổng kết

Tiêu chí	Non-Agentic App	Agentic App
Đơn giản	✅	❌ (phức tạp hơn)
Linh hoạt	❌	✅
Tự chủ	❌	✅
Dễ kiểm soát	✅	❌
Tác vụ phức tạp	❌	✅
Sử dụng tool ngoài	Có nhưng hạn chế	Mạnh mẽ, linh hoạt

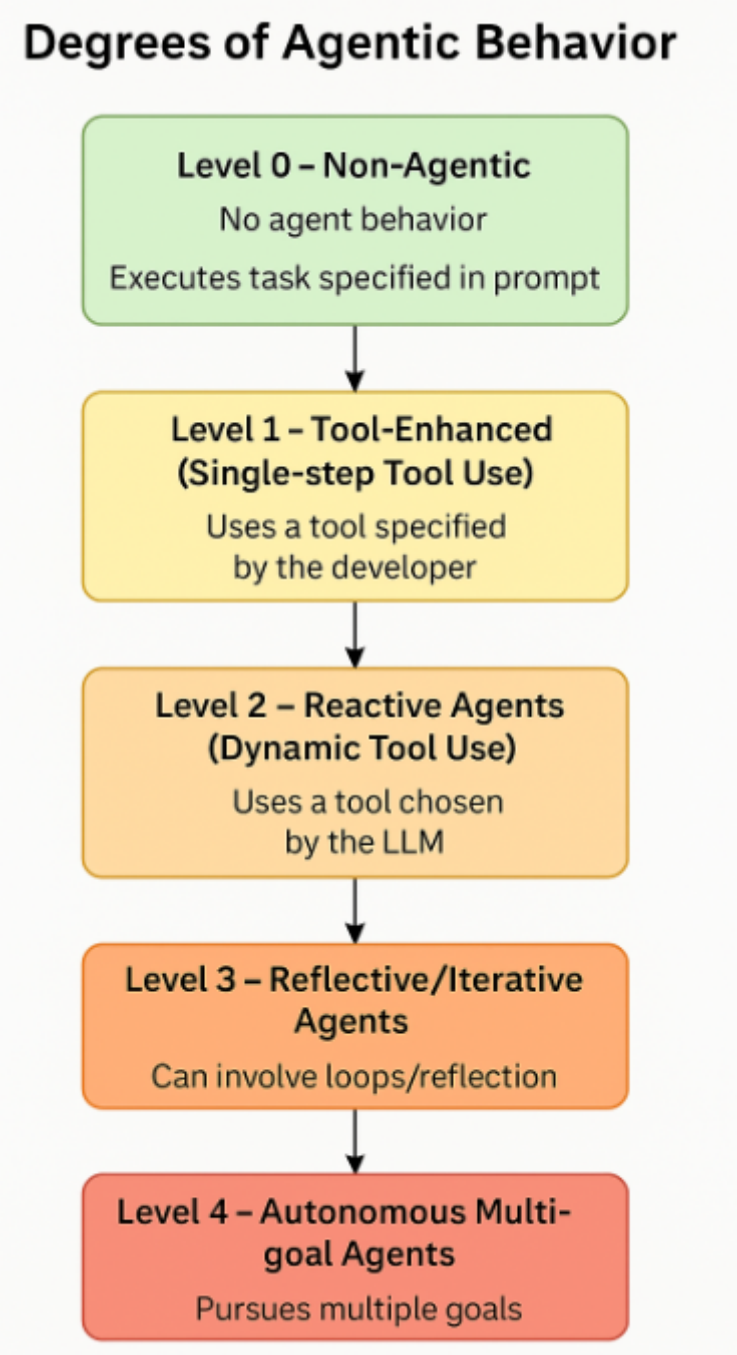
Không có lựa chọn "tốt hơn tuyệt đối" — bạn cần chọn dựa trên **mức độ phức tạp của bài toán**, khả năng chấp nhận rủi ro, và yêu cầu logic xử lý.

- Nếu bạn đang **xây chatbot trả lời kiến thức công ty**, hãy bắt đầu với **Non-Agentic**.
- Nếu bạn muốn một **trợ lý AI làm việc như con người**, thì **Agentic** là tương lai.

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Cấp độ hành vi của tác nhân(Agentic) trong ứng dụng LLM



LLMs như GPT, Claude, hay Gemini, ngày càng nhiều ứng dụng được xây dựng theo kiến trúc **agentic** – tức là cho phép mô hình **"suy nghĩ"**, **lập kế hoạch**, và **tự ra quyết định**. Nhưng không phải ứng dụng nào cũng cần mức độ agentic giống nhau.

Vì vậy, khái niệm Cấp độ hành vi của tác nhân(**Degrees of Agentic Behavior)** ra đời – mô tả các **cấp độ từ đơn giản đến phức tạp** mà một hệ thống có thể thể hiện hành vi như một tác nhân (agent).

Agentic Behavior là gì?

"Agentic behavior" đề cập đến **khả năng hành động một cách có mục đích** của mô hình – không chỉ trả lời từng lệnh một, mà còn:

- Lập kế hoạch hành động
- Chọn công cụ phù hợp
- Gọi nhiều hành động kế tiếp
- Đánh giá lại kết quả
- Điều chỉnh chiến lược

Ví dụ, thay vì chỉ trả lời “không biết”, một agentic app có thể:

> “Tôi không có thông tin trong cơ sở dữ liệu. Tôi sẽ tìm kiếm trên web, tóm tắt, rồi trình bày kết quả.”

5 Cấp độ Hành vi Agentic (Degrees of Agentic Behavior)

Chúng ta có thể chia hành vi agentic thành **năm cấp độ**, từ đơn giản (non-agentic) đến phức tạp (autonomous agents):

Level 0 – Không có hành vi tác nhân(Non-Agentic)

□□ Chỉ thực hiện một hành động theo prompt đã lập trình sẵn.
□□ Không có logic điều phối, không lập kế hoạch.

Ví dụ:

```
response = llm("Hãy tóm tắt đoạn văn sau...")
```

Dùng cho:

- Tóm tắt, dịch, phân loại
- Chatbot trả lời trực tiếp

Level 1 – Tăng cường bằng công cụ - Sử dụng công cụ một bước(Tool-Enhanced - Single-step Tool Use)

□□ Ứng dụng có thể sử dụng **một công cụ bên ngoài**, nhưng người lập trình phải chỉ định rõ **khi nào dùng tool nào**.

Ví dụ:

```
if "search" in user_input:
    result = web_search_tool(query)
else:
    result = llm(user_input)
```

Dùng cho:

- Tìm kiếm trên web theo trigger
- Trích xuất dữ liệu từ cơ sở kiến thức

Level 2 – Các tác nhân phản ứng - Sử dụng công cụ động(Reactive Agents - Dynamic Tool Use)

□□ LLM **tự quyết định khi nào cần dùng công cụ nào**
□□ Các công cụ (tools) được mô tả bằng prompt hoặc API
□□ LLM chọn hành động tốt nhất dựa trên ngữ cảnh hiện tại

Ví dụ: LLM tự chọn `Calculator` để tính toán thay vì chỉ đoán kết quả.

Dùng LangChain Agents, OpenAI Function Calling, hoặc LangGraph Nodes.

Dùng cho:

- Trợ lý cá nhân AI có khả năng tìm kiếm, phân tích, tóm tắt
- Chatbot hiểu ngữ cảnh và chọn hành động phù hợp

Level 3 – Các tác nhân tự phản hồi hoặc thực hiện vòng lặp (Reflective / Iterative)

□□ Agent có khả năng **tự phản hồi (self-reflect)** và thực hiện **vòng lặp xử lý** đến khi đạt được mục tiêu
□□ Có thể lập kế hoạch, thực hiện hành động, kiểm tra lại, và sửa lỗi

Ví dụ:

```
if "search" in user_input:
    result = web_search_tool(query)
else:
    result = llm(user_input)
```

Dùng **Looping Graphs** trong LangGraph hoặc Agent Executors có khả năng hồi quy.

Dùng cho:

- Tác vụ đòi hỏi tư duy nhiều bước (multi-step reasoning)
- Tự kiểm tra và cải thiện kết quả

Level 4 – Các tác nhân tự chủ đa mục tiêu (Autonomous Multi-goal Agents)

□□ Agent có khả năng:

- Nhận mục tiêu tổng quát ("Viết báo cáo thị trường tháng này")
- **Tự chia nhỏ thành các sub-goals**
- **Tự lên kế hoạch**
- **Tự chạy, điều chỉnh, giám sát tiến trình**

Ví dụ: AutoGPT, BabyAGI, OpenDevin

Có thể chạy **nhiều vòng lặp**, tự lưu trạng thái và chuyển hướng kế hoạch khi có lỗi.

Dùng cho:

- Quản lý dự án
- Trợ lý lập trình hoặc viết báo cáo tự động
- Tự động hóa tác vụ phức tạp không xác định trước logic

So sánh nhanh

Cấp độ	Tên gọi	Mức độ tự chủ	Có lập kế hoạch	Có hành vi phản xạ	Dùng cho tác vụ phức tạp
0	Không có hành vi tác nhân (Non-agentic)	□	□	□	□
1	Tăng cường bằng công cụ (Tool-enhanced)	●	□	□	□
2	Các tác nhân phản ứng (Reactive agents)	□	□	□	□ (mức vừa)
3	Các tác nhân tự phản hồi hoặc thực hiện vòng lặp (Reflective/Iterative agents)	□□	□	□	□□
4	Các tác nhân tự chủ đa mục tiêu (Autonomous agents)	□□□	□□	□□	□□□

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

LangGraph Studio, giới thiệu, cài đặt và cách dùng

Giới thiệu LangGraph Studio – Trình quan sát & phát triển Agentic Graphs

I. LangGraph Studio là gì?

LangGraph Studio là một công cụ **giao diện đồ họa (GUI)** giúp bạn:

- Quan sát trực quan đồ thị agent (agent graph)
- Gỡ lỗi (debug) các bước thực thi của LLM agent
- Theo dõi trạng thái, dữ liệu, và hướng đi qua từng node
- Phát triển và kiểm thử agent nhanh chóng

Nó giống như một **"developer console" dành riêng cho LangGraph**, giúp bạn hiểu và kiểm soát quá trình suy luận và hành động của mô hình AI một cách rõ ràng, trực quan.

II. Tại sao nên dùng LangGraph Studio?

Vấn đề khi không có Studio	LangGraph Studio giúp
Debug khó vì agent ẩn trong LLM	Hiển thị chi tiết từng bước, trạng thái
Khó hiểu LLM đang đi hướng nào	Hiển thị flow trực quan
Phát triển agent lặp đi lặp lại	Test từng phần trực tiếp
Không biết LLM nghĩ gì	Hiển thị prompt, output, action, tool dùng

III. Cài đặt LangGraph Studio

LangGraph Studio được tích hợp sẵn trong thư viện `langgraph`. Để cài đặt và sử dụng, bạn chỉ cần:

1. Cài đặt thư viện:

```
pip install langgraph[dev]
```

Hoặc nếu bạn đã có `langgraph`:

```
pip install -U "langgraph[dev]"
```

`dev` extras sẽ cài thêm các gói phục vụ cho Studio như `fastapi`, `uvicorn`, `jupyter`, v.v.

IV. Cách dùng LangGraph Studio

LangGraph Studio hiện hỗ trợ **hai cách chính để sử dụng**:

---

1. Chạy dưới dạng local app (FastAPI server)

Bạn có thể thêm dòng sau vào project của bạn để chạy Studio như một app mini:

```
from langgraph.studio import start_trace_server


start_trace_server()
```

Mặc định server chạy tại `http://localhost:4000`

Khi bạn chạy graph của mình, Studio sẽ tự động ghi lại luồng xử lý, trạng thái, inputs/outputs, và hiển thị trên giao diện web.

2 . Tích hợp trong Jupyter Notebook

Nếu bạn đang làm việc trong Jupyter (hoặc Google Colab), bạn có thể dùng:

```
from langgraph.studio.jupyter import trace_graph


with trace_graph(graph) as session:
    result = session.invoke(input_data)
```

Tính năng này giúp bạn:

- Thấy flow đồ thị ngay trong notebook
- Click để xem chi tiết từng node, prompt, tool call, response
- So sánh nhiều lần chạy khác nhau (useful for debugging)

```
from langgraph.graph import StateGraph
from langgraph.studio import start_trace_server

# Define some simple node functions
def say_hello(state):
    print("Hello!")
    return state

def ask_name(state):
    state["name"] = "Alice"
    return state

# Build graph
builder = StateGraph(dict)
builder.add_node("hello", say_hello)
builder.add_node("ask", ask_name)
builder.set_entry_point("hello")
builder.add_edge("hello", "ask")

graph = builder.compile()

# Start studio server
start_trace_server()

# Run
graph.invoke({})
```

Khi chạy đoạn code này, bạn có thể truy cập `http://localhost:4000` để thấy luồng `hello → ask` cùng với trạng thái và log chi tiết.

VI. Các tính năng nổi bật

Tính năng	Mô tả
Flow Graph View	Hiển thị đồ thị agent và luồng thực tế đã đi qua
Prompt Viewer	Hiển thị đầy đủ prompt mà LLM nhận được
Input/Output Log	Xem tất cả input và output của từng bước

Tính năng	Mô tả
Multi-run Comparison	So sánh nhiều phiên bản chạy graph khác nhau
Token usage & Cost (tương lai)	Theo dõi chi phí sử dụng OpenAI, Anthropic,...
Node debug	Hiển thị error và traceback nếu có lỗi ở một node

VII. Khi nào nên dùng LangGraph Studio?

- Khi bạn xây dựng agent có nhiều bước
- Khi bạn xử lý luồng có điều kiện, vòng lặp, branch
- Khi cần kiểm soát tool call / prompt / phản hồi
- Khi muốn debug graph state dễ dàng

VIII. Tổng kết

Câu hỏi	Trả lời
LangGraph Studio là gì?	Là công cụ GUI để quan sát và phát triển LangGraph Agents một cách trực quan và có thể debug.
Làm gì được với nó?	Theo dõi prompt, response, trạng thái, tool call... từng bước trong graph
Cài đặt ra sao?	`pip install langgraph[dev]`
Chạy ở đâu?	Trong local (FastAPI) hoặc trong Jupyter Notebook

Tác giả: Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft