# Tạo ứng dụng cơ bản LangGraph AI Agents.

Tài liệu này sẽ hướng dẫn bạn cách cài đặt dự án và hiểu rõ các thành phần cốt lõi trong một đồ thị (graph).
Từ đó, bạn có thể dễ dàng tùy chỉnh, mở rộng hoặc tích hợp thêm các công cụ để phù hợp với nhu cầu thực tế.
Biên soạn bởi:
Đỗ Ngọc Tú
CEO VHTSoft

- LangGraph: Framework Xây Dựng Ứng Dụng LLM Dựa Trên Agent + Graph
- 5 Khái Niệm Trụ Cột
- Giới thiệu Poetry
- Cơ bản về GraphBuilder
- Xây dựng một ứng dụng đơn giản (graph) quyết định xem người dùng nên uống cà phê hay trà
- Ứng dụng LangGraph cơ bản với chatbot và công cụ
- Định tuyến(Router) trong LangGraph
- So sánh Router và Node trong LangGraph
- Hiểu rõ hơn về add_conditional_edges
- Bài Thực Hành Router Trong LangGraph
- Giới thiệu về ReAct Architecture
- Thực hành – Node LLM quyết định bước tiếp theo
- Kết nối các công cụ (tools) bằng bind_tools

# LangGraph: Framework Xây Dựng Ứng Dụng LLM Dựa Trên Agent + Graph

**LangGraph** là một framework mã nguồn mở được xây dựng trên nền tảng của **LangChain**, cho phép bạn phát triển các ứng dụng sử dụng **LLM (Large Language Models)** theo mô hình **agentic** – nơi AI không chỉ trả lời câu hỏi, mà còn **suy luận, quyết định, gọi tool, và ghi nhớ trạng thái** trong nhiều bước.

Điểm đặc biệt của LangGraph là nó tổ chức toàn bộ logic ứng dụng dưới dạng một **đồ thị trạng thái (stateful graph)** – mỗi "node" là một hành động, một agent, hoặc một bước xử lý.

## Tại sao cần LangGraph?

Các ứng dụng AI ngày nay không chỉ dừng ở hỏi-đáp đơn lẻ. Chúng cần:

- Gọi nhiều công cụ (tool use)
- Ghi nhớ trạng thái để suy luận liên tục
- Lặp lại hành động hoặc rẽ nhánh theo kết quả
- Kiểm soát logic phức tạp có điều kiện

LangGraph giúp bạn làm tất cả điều đó **một cách có cấu trúc, dễ theo dõi và mở rộng.**

## LangGraph hoạt động như thế nào?

LangGraph xây dựng ứng dụng theo mô hình **Finite-State Machine (FSM)**:

- **Mỗi node** là một bước xử lý (ví dụ: gọi LLM, tìm kiếm, phân tích, v.v.)
- **State** là dữ liệu được lưu và truyền qua từng bước
- **Edge** quyết định luồng đi tiếp theo dựa trên kết quả hoặc trạng thái

Bạn có thể tạo các ứng dụng như:

- Chatbot thông minh
- Multi-tool agent (Google + Calculator + Document Retriever…)
- Workflow phân tích dữ liệu
- Trợ lý quy trình nghiệp vụ (ERP, CRM…)

# Cú pháp cơ bản

```
from langgraph.graph import StateGraph

def greet(state): print("Hello!"); return state
def ask_name(state): state["name"] = "Alice"; return state

builder = StateGraph(dict)
builder.add_node("greet", greet)
builder.add_node("ask", ask_name)
builder.set_entry_point("greet")
builder.add_edge("greet", "ask")

graph = builder.compile()
graph.invoke({})
```

# 5 Khái Niệm Trụ Cột

LangGraph là một framework mạnh mẽ để xây dựng các ứng dụng LLM có logic phức tạp theo mô hình **graph-based agent**. Để làm chủ LangGraph, bạn cần hiểu rõ 5 thành phần chính: **State**, **Schema**, **Node**, **Edge**, và **Compile**.

## 1.  State = Memory (Trạng thái là bộ nhớ)

Trong LangGraph, **State** là nơi lưu trữ toàn bộ thông tin mà ứng dụng của bạn cần ghi nhớ trong suốt quá trình chạy.

- Nó có thể chứa input từ người dùng, kết quả từ LLM, thông tin truy xuất từ tool, v.v.
- Mỗi node trong graph có thể **đọc và ghi vào state**.
- State giúp ứng dụng **suy nghĩ nhiều bước**, ghi nhớ dữ kiện, và sử dụng lại ở bước sau.

**Ví dụ:**

```
{"question": "Chủ tịch nước Việt Nam là ai?", "llm_response": "..." }
```

State giống như **RAM** của agent – càng tổ chức tốt, agent càng thông minh và hiệu quả.

## 2. Schema = Data Format (Định dạng dữ liệu)

LangGraph yêu cầu bạn **định nghĩa Schema cho State** để đảm bảo dữ liệu được quản lý nhất quán, an toàn và rõ ràng.

- Schema là một **class Python** (thường dùng `pydantic.BaseModel` hoặc `TypedDict`)
- Nó xác định rõ những gì tồn tại trong state, loại dữ liệu, và bắt buộc hay không.

**Ví dụ:**

```
from typing import TypedDict


class MyState(TypedDict):
    question: str
    answer: str
```

**Schema = bản thiết kế dữ liệu, giúp bạn tránh lỗi, dễ debug và mở rộng.**

# 3. Node = Step (Một bước xử lý)

**Node** là một bước trong đồ thị – nơi một phần công việc được thực hiện.

- Có thể là một LLM call, một hành động với tool, một quyết định logic, v.v.
- Mỗi node nhận `state`, xử lý và trả lại `state` đã được cập nhật.

**Ví dụ node đơn giản:**

```
def call_llm(state):
    response = llm.invoke(state["question"])
    state["answer"] = response
    return state
```

# 4.  Edge = Kết nối giữa các bước(Operation Between Steps)

**Edge** là mối liên kết giữa các node – giúp bạn định nghĩa luồng đi của chương trình.

- Một node có thể có **nhiều edge** đi tới các bước khác nhau (ví dụ dựa vào kết quả).
- Có thể định nghĩa **logic rẽ nhánh (branching)**, **lặp lại (loop)** hoặc **kết thúc** tại đây.

**Ví dụ:**

```
graph.add_edge("ask_user", "call_llm")
graph.add_conditional_edges("call_llm", {
    "need_more_info": "search_tool",
    "done": END
})
```

**Edge quyết định luồng suy nghĩ và logic quyết định của agent.**

# 5. Compile = Đóng gói đồ thị để sử dụng(Pack)

Sau khi bạn đã định nghĩa đầy đủ các **node, edge và schema**, bước cuối cùng là **compile** – để tạo ra một graph sẵn sàng chạy.

- `compile()` sẽ kiểm tra sự nhất quán, ràng buộc, và đóng gói toàn bộ graph.
- Sau đó, bạn có thể dùng `.invoke()` để chạy agent một cách mượt mà.

**Ví dụ:**

```
builder = StateGraph(MyState)
# add node, edge...
graph = builder.compile()
graph.invoke({"question": "..."})
```

Compile = **chuyển bản thiết kế sang sản phẩm hoàn chỉnh có thể vận hành.**

### Tóm lại

| Thành phần | Vai trò | Ví dụ dễ hiểu |
|---|---|---|
| **State** | Bộ nhớ dùng để truyền dữ liệu giữa các bước | “Tôi nhớ câu hỏi người dùng” |
| **Schema** | Định nghĩa cấu trúc và loại dữ liệu trong state | “State phải có ‘question’ dạng chuỗi” |
| **Node** | Một bước xử lý như LLM call, tool call, v.v. | “Gọi GPT để trả lời” |
| **Edge** | Điều khiển luồng di chuyển giữa các node | “Nếu xong thì kết thúc, nếu thiếu thì tìm tiếp” |
| **Compile** | Đóng gói toàn bộ graph để vận hành | “Biến các bước rời rạc thành một chương trình” |

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Giới thiệu Poetry

**Poetry** – một công cụ hiện đại để quản lý gói (package) và môi trường (environment) trong Python.

## Poetry là gì?

**Poetry** là một công cụ giúp bạn:

- Quản lý dependencies (thư viện phụ thuộc).
- Quản lý và build package Python.
- Tạo môi trường ảo tự động.
- Publish package lên PyPI dễ dàng.

Poetry thay thế việc bạn phải dùng `pip`, `venv`, `requirements.txt`, `setup.py`, `pyproject.toml`,... riêng lẻ — **tất cả tích hợp trong một công cụ**.

## Tại sao nên dùng Poetry?

| Truyền thống (pip + venv + requirements) | Poetry |
|---|---|
| Quản lý môi trường và dependency riêng | Tất cả-in-one |
| Cần nhiều file config khác nhau | Dùng duy nhất `pyproject.toml` |
| Khó pin version chính xác | Poetry lock chính xác |
| Không tự động tạo môi trường | Poetry tự tạo virtualenv |
| Không có tính năng publish | Poetry có lệnh `poetry publish` |

## Khi nào nên dùng Poetry?

- Khi bạn muốn quản lý dependency dễ dàng và sạch sẽ.
- Khi bạn muốn tạo project có thể publish lên PyPI.
- Khi bạn làm việc nhóm hoặc CI/CD cần môi trường chính xác.
- Khi bạn phát triển package, lib hoặc ứng dụng production.

## Cài đặt Poetry

```
curl -sSL https://install.python-poetry.org | python3 -
```

```
Sau đó thêm vào shell config nếu cần:
export PATH="$HOME/.local/bin:$PATH"


poetry --version
```

# Khởi tạo project với Poetry

```
poetry new myproject
cd myproject
```

Cấu trúc thư mục sẽ được tạo sẵn:

```
myproject/
├── myproject/
│   └── __init__.py
├── tests/
├── pyproject.toml
└── README.rst
```

# Thêm dependency

```
poetry add requests
```

Poetry sẽ:

- Cài thư viện vào virtualenv riêng.
- Ghi lại vào `pyproject.toml` và `poetry.lock`.

# Muốn thêm dev-dependency?

```
poetry add --dev black
```

**Tạo môi trường và chạy shell**

```
poetry shell
```

Giống như `source venv/bin/activate` — bạn sẽ vào môi trường ảo của Poetry.

Hoặc chạy lệnh trong môi trường mà không cần shell:

```
poetry run python script.py
```

# pyproject.toml là gì?

Đây là file cấu hình trung tâm của project Python hiện đại.
Poetry dùng file này để:

- Khai báo tên project, version, mô tả...
- Danh sách dependencies.
- Cấu hình build.

Ví dụ:

```
[tool.poetry]
name = "VHTSoft"
version = "0.1.0"
description = "Dự án tuyệt vời"
authors = ["Bạn <admin@vhtsoft.com>"]

[tool.poetry.dependencies]
python = "^3.10"
requests = "^2.31.0"

[tool.poetry.dev-dependencies]
black = "^23.0.0"
```

# Build và Publish

```
poetry build
poetry publish --username __token__ --password pypi-***
```

Bạn có thể đăng lên PyPI nếu muốn chia sẻ package.

# Lệnh phổ biến

| Lệnh | Chức năng |
| --- | --- |
| `poetry new myproject` | Tạo mới một project |

| Lệnh | Chức năng |
| --- | --- |
| `poetry init` | Tạo `pyproject.toml` trong thư mục hiện tại |
| `poetry add <package>` | Thêm dependency |
| `poetry install` | Cài dependencies từ `pyproject.toml` |
| `poetry update` | Cập nhật toàn bộ thư viện |
| `poetry lock` | Tạo/cập nhật `poetry.lock` |
| `poetry run <cmd>` | Chạy lệnh trong môi trường ảo |
| `poetry shell` | Mở môi trường ảo |
| `poetry build` | Đóng gói project |
| `poetry publish` | Đăng lên PyPI |

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Cơ bản về GraphBuilder

Trong LangGraph, bạn sẽ sử dụng một **GraphBuilder** để định nghĩa các bước (nodes), luồng xử lý (edges), điều kiện phân nhánh (routers), và điểm bắt đầu/kết thúc của một ứng dụng dựa trên LLM.

```
from langgraph.graph import StateGraph


builder = StateGraph(StateType)
```

Ở đây `StateType` là schema mô tả state (thường là TypedDict hoặc pydantic model).

Ví dụ

```
from typing import TypedDict, Literal


class DrinkState(TypedDict):
    preference: Literal["coffee", "tea", "unknown"]
```

## 1. builder.add_node(name, node_callable)

Thêm một bước xử lý (node) vào graph.

**Cú pháp:**

```
builder.add_node("tên_node", hàm_xử_lý)
```

Ví dụ

```
def ask_name(state):
    state["name"] = input("Tên bạn là gì? ")
    return state


builder.add_node("ask_name", ask_name)
```

## 2. builder.add_edge(from_node, to_node)

Tạo liên kết (đường đi) giữa hai node.

**Cú pháp:**

```
builder.add_edge("node_nguồn", "node_đích")
```

**Ví dụ:**

```
builder.add_edge("ask_name", "ask_preference")
```

## 3. builder.add_conditional_edges(from_router_node, conditions_dict)

**from_router_node: là một hàm xử lý**

Tạo đường đi có điều kiện (routing) từ một node tới nhiều node tùy thuộc vào kết quả của node đó.

**Cú pháp:**

```
builder.add_conditional_edges("router_node", {
    "kết_quả_1": "node_1",
    "kết_quả_2": "node_2",
    ...
})
```

**Ví dụ:**

```
builder.add_conditional_edges("router", {
    "coffee_node": "coffee_node",
    "tea_node": "tea_node",
    "fallback_node": "fallback_node"
})
```

=> Node `"router"` sẽ trả về một chuỗi (ví dụ `"coffee_node"`), và flow sẽ đi đến node tương ứng.

## 4. builder.set_entry_point(node_name)

Xác định điểm bắt đầu của graph.

Ví dụ

```
builder.set_entry_point("start")
```

## 5. builder.set_finish_point(node_name)

Đánh dấu node kết thúc, tức là sau node này thì flow dừng lại.

Có thể có nhiều node kết thúc!

**Ví dụ**

```
builder.set_finish_point("coffee_node")
builder.set_finish_point("tea_node")
```

# 6. builder.compile()

**Biên dịch graph** của bạn thành một `CompiledGraph` có thể chạy được.

**Ví dụ**

```
app = builder.compile()
```

Sau đó

```
app.invoke({"preference": "unknown"}
```

# 7. Tổng kết

| Hàm | Mô tả |
|---|---|
| `add_node(name, node_fn)` | Thêm một bước xử lý đơn vào graph |
| `add_edge(from_node, to_node)` | Tạo liên kết tuyến tính giữa 2 node |
| `add_conditional_edges(name, mapping)` | Tạo node phân nhánh có điều kiện |
| `set_entry_point(name)` | Xác định điểm bắt đầu của flow |
| `set_finish_point(name)` | Xác định điểm kết thúc |
| `add_branch(name, sub_graph)` | Thêm một graph phụ (dạng phân nhánh) vào một node |
| `add_recursion(name, recursive_graph)` | Cho phép gọi đệ quy một graph con |
| `compile()` | Biên dịch toàn bộ graph thành một ứng dụng chạy được |
| `add_generator_node(name, generator_node)` | Thêm một node sử dụng generator (yield từng bước) |
| `add_parallel_nodes(nodes: dict)` | Chạy nhiều node song song trong một bước |
| `add_conditional_edges_with_default(name, conditions, default)` | Giống `add_conditional_edges` nhưng có nhánh mặc định |
| `add_stateful_node(name, node_fn)` | Dùng khi node cần giữ trạng thái riêng biệt |
| `add_async_node(name, async_node)` | Thêm node xử lý bất đồng bộ (`async def`) |

**Tác giả: Đỗ Ngọc Tú**

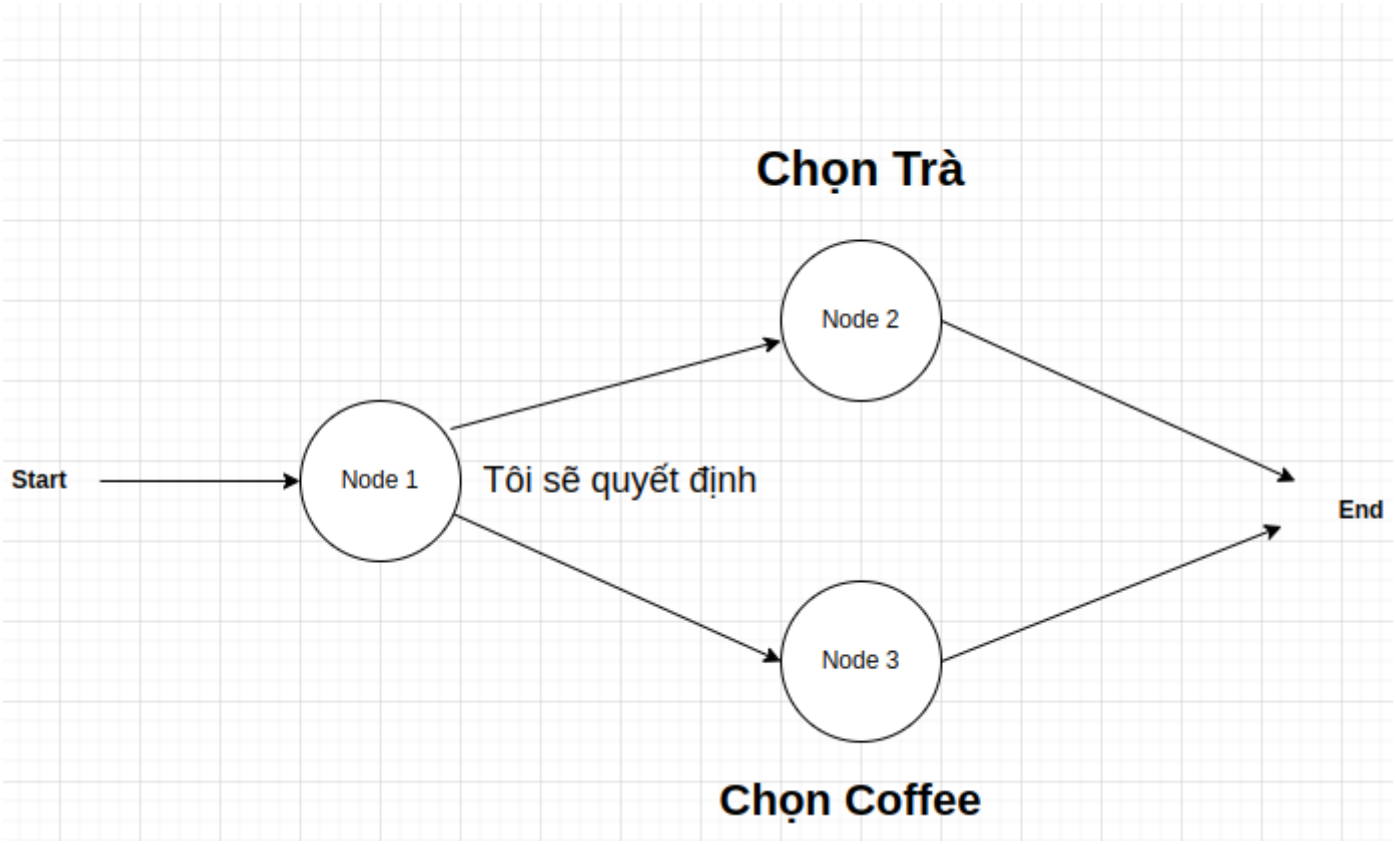
**Công Ty Phần Mềm VHTSoft**

# Xây dựng một ứng dụng đơn giản (graph) quyết định xem người dùng nên uống cà phê hay trà

Mục tiêu của bài tập này là làm quen với các thành phần và bước chính của ứng dụng LangGraph cơ bản. **State**, **Schema**, **Node**, **Edge**, và **Compile**.

I. Cài đặt

- Cài đặt Peotry tại đây
- tại terminal:
  - cd project_name
  - pyenv local 3.11.4
  - poetry install
  - poetry shell
  - jupyter lab
- Tạo file .env
- ```
  OPENAI_API_KEY=your_openai_api_key
  LANGCHAIN_TRACING_V2=true
  LANGCHAIN_ENDPOINT=https://api.smith.langchain.com # Bạn sẽ cần đăng ký tài khoản tại smith.langchain.com để lấy API key.
  LANGCHAIN_API_KEY=your_langchain_api_key
  LANGCHAIN_PROJECT=your_project_name
  ```
- Kết nối với tệp .env nằm trong cùng thư mục vào notebook
- ```
  #pip install python-dotenv
  ```
- ```
  import os
  from dotenv import load_dotenv, find_dotenv
  _ = load_dotenv(find_dotenv())
  ```
  ```

```
openai_api_key = os.environ["OPENAI_API_KEY"]
```

- Cài đặt LangChain(Nếu bạn đang sử dụng poetry shell được tải sẵn, bạn không cần phải cài đặt gói sau vì nó đã được tải sẵn cho bạn:)
- ```
  #!pip install langchain
  ```
- Kết nối với **LLM**
- ```
  #!pip install langchain-openai
  from langchain_openai import ChatOpenAI

  chatModel35 = ChatOpenAI(model="gpt-3.5-turbo-0125")
  chatModel4o = ChatOpenAI(model="gpt-4o")
  ```

Đây là hình ảnh đồ họa của ứng dụng chúng ta sẽ xây dựng




# 2. Xác định lược đồ trạng thái(state schema)

- Điều đầu tiên chúng ta sẽ làm là xác định State của ứng dụng.
- State schema cấu hình những gì,  sẽ có trong state và với định dạng nào.

- State thường sẽ là TypedDict hoặc mô hình Pydantic.

Đối với ví dụ này, lớp State sẽ chỉ có một khóa, graph_state, với định dạng chuỗi

```
from typing_extensions import TypedDict


class State(TypedDict):
    graph_state: str
```

**TypedDict là gì?**
- TypedDict cho phép bạn định nghĩa các từ điển có cấu trúc cụ thể. Nó cho phép bạn chỉ định các khóa chính xác và các loại giá trị liên quan của chúng, giúp mã của bạn rõ ràng hơn và an toàn hơn về mặt kiểu.

Ví dụ, thay vì một từ điển đơn giản như

```
{"name": "Ngoc Tu", "age": 40}
```

bạn có thể định nghĩa một TypedDict để thực thi rằng tên phải luôn là một chuỗi và tuổi phải luôn là một số nguyên.

Ở đây, một cấu trúc giống như từ điển mới có tên là State được định nghĩa.
State kế thừa từ TypedDict, nghĩa là nó hoạt động giống như một từ điển, nhưng có các quy tắc cụ thể về các khóa và giá trị mà nó phải có.

```
state: State = {"graph_state": "active"}  # Đúng
```

```
state: State = {"graph_state": 123}  # Sai, Điều này sẽ gây ra lỗi kiểm tra kiểu vì giá trị 123 không phải là chuỗi.
```

# 3. Xác định các Nodes của ứng dụng

Các Nodes của ứng dụng này được định nghĩa là các hàm python.
Đối số đầu tiên của hàm Node là state. Mỗi Node có thể truy cập khóa graph_state, với state['graph_state'].
Trong bài tập này, mỗi Node sẽ trả về một giá trị mới của khóa trạng thái graph_state. Giá trị mới do mỗi Node trả về sẽ ghi đè lên giá trị **state** trước đó.
Nếu luồng đồ thị là Node 1 đến Node 2, trạng thái cuối cùng sẽ là "**Chọn Trà**".
Nếu luồng đồ thị là nút 1 đến nút 3, trạng thái cuối cùng sẽ là "**Chọn Coffee**".

```
def node_1(state):
    print("---Node 1---")
    return {"graph_state": state['graph_state'] +" Tôi sẽ quyết định"}
```

```
def node_2(state):
    print("---Node 2---")
    return {"graph_state": state['graph_state'] +" Chọn Trà"}

def node_3(state):
    print("---Node 3---")
    return {"graph_state": state['graph_state'] +" Chọn Coffee"}
```

# 4. Xác định các Edges(cạnh) kết nối các Node

Cạnh thông thường đi từ Node này sang Node khác.

## 4.1 Edge không điều kiện giữa các node

```
builder = GraphBuilder()


# Thêm các node
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Tạo edge không điều kiện
builder.add_edge("node_1", "node_2")  # Sau node_1 thì luôn đi đến node_2
builder.add_edge("node_2", "node_3")  # Sau node_2 thì luôn đi đến node_3

# Đặt node bắt đầu
builder.set_entry_point("node_1")

# Tạo app và chạy thử
app = builder.compile()

initial_state = {"graph_state": ""}
final_state = app.invoke(initial_state)
print("Kết quả:", final_state["graph_state"])
```

- `add_edge("node_1", "node_2")`: Khi node_1 chạy xong, **luôn luôn** chuyển sang node_2.
- `add_edge("node_2", "node_3")`: Khi node_2 chạy xong, **luôn luôn** chuyển sang node_3.
- Không cần bất kỳ điều kiện nào – đó là lý do gọi là **unconditional edge**.

```
---Node 1---
---Node 2---
```

```
---Node 3---
Kết quả: Tôi sẽ quyết định. Chọn Trà. Chọn Coffee.
```

## 4.2 Edge Có điều kiện giữa các node

Cạnh có điều kiện được sử dụng khi bạn muốn tùy chọn định tuyến giữa các Node. Chúng là các hàm chọn nút tiếp theo dựa trên một số logic.
Trong ví dụ trên, để sao chép kết quả quyết định của người dùng, chúng ta sẽ xác định một hàm để mô phỏng xác suất **Chọn Trà** là 60%.

```
import random
from typing import Literal

def decide_dring(state) -> Literal["node_2", "node_3"]:
    """
    Quyết định nút tiếp theo dựa trên xác suất chia 60/40.
    """
    # Mô phỏng xác suất 60% cho "node_2"
    if random.random() < 0.6:  # 60% chance
        return "node_2"

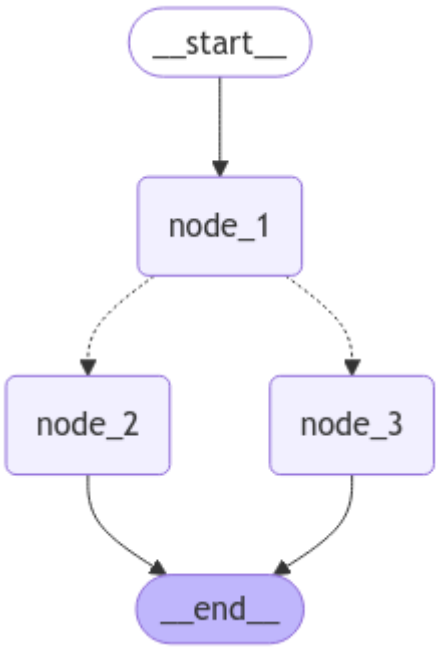
    # Còn lại 40% cơ hội
    return "node_3"
```

```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END

# Build graph
builder = StateGraph(State)

builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Add the logic of the graph
builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_dring) #decide_dring trả về tên `node 2` hoặc `node 3`
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)

# Compile the graph
```

```
graph = builder.compile()


# Visualize the graph
display(Image(graph.get_graph().draw_mermaid_png()))
```

Đầu tiên, chúng ta khởi tạo StateGraph với lớp State mà chúng ta đã định nghĩa ở trên.
Sau đó, chúng ta thêm các nút và cạnh.
START Node là một nút đặc biệt gửi dữ liệu đầu vào của người dùng đến **graph**, để chỉ ra nơi bắt đầu đồ thị của chúng ta.

```
builder.add_conditional_edges("node_1", decide_dring)
```

Mỗi lần hàm được gọi, nó đưa ra quyết định dựa trên cơ hội ngẫu nhiên:
60% cơ hội: Trả về "node_2".
40% cơ hội: Trả về "node_3".
Tại sao sử dụng điều này?
Để ra quyết định: Nó mô phỏng một lựa chọn xác suất, có thể hữu ích trong các ứng dụng như mô phỏng, trò chơi hoặc học máy.
Kiểu trả về theo nghĩa đen đảm bảo rằng hàm sẽ luôn chỉ trả về "node_2" hoặc "node_3", giúp dễ dự đoán và gỡ lỗi hơn.

END Node là một nút đặc biệt biểu thị một nút đầu cuối.
Chúng ta biên dịch **graph** của mình để thực hiện một vài kiểm tra cơ bản trên cấu trúc **graph**.




# 5. Chạy ứng dụng

Graph đã biên dịch triển khai giao thức runnable, một cách chuẩn để thực thi các thành phần LangChain. Do đó, chúng ta có thể sử dụng invoke làm một trong những phương pháp chuẩn để chạy ứng dụng này.

Đầu vào ban đầu của chúng ta là từ điển {"graph_state": "Xin chào, đây là VHTSoft."}, từ điển này đặt giá trị ban đầu cho từ điển trạng thái.

Khi invoke được gọi:

Biểu đồ bắt đầu thực thi từ nút START.

Nó tiến triển qua các nút đã xác định (node_1, node_2, node_3) theo thứ tự.

Cạnh có điều kiện sẽ đi qua từ nút 1 đến nút 2 hoặc 3 bằng quy tắc quyết định 60/40.

Mỗi hàm nút nhận trạng thái hiện tại và trả về một giá trị mới, giá trị này sẽ ghi đè trạng thái biểu đồ.

Việc thực thi tiếp tục cho đến khi đạt đến nút END.

```
graph.invoke({"graph_state" : "Hi, Tôi là VHTSoft."})
```

```
{'graph_state': 'Hi, Tôi là VHTSoft. Tôi sẽ quyết định Chọn Trà'}
```

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Ứng dụng LangGraph cơ bản với chatbot và công cụ

## Mục tiêu của ứng dụng.


Sử dụng LangGraph quyết định phản hồi bằng chatbot hoặc sử dụng công cụ

Chúng ta sẽ xây dựng một ứng dụng cơ bản thực hiện 4 thao tác
Sử dụng chat messages làm **State**

Sử dụng chat model làm **Node**

Liên kết một công cụ với chat model

Thực hiện lệnh gọi công cụ trong **Node**

**I. Cài đặt**

```
* cd project_name
* pyenv local 3.11.4
* poetry install
* poetry shell
* jupyter lab
```

tạo file .env

```
OPENAI_API_KEY=your_openai_api_key
* LANGCHAIN_TRACING_V2=true
* LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
* LANGCHAIN_API_KEY=your_langchain_api_key
* LANGCHAIN_PROJECT=your_project_name
```

Bạn sẽ cần đăng ký tài khoản tại smith.langchain.com để lấy API key.

Kết nối với file .env nằm trong cùng thư mục của notebook

```
#pip install python-dotenv
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
openai_api_key = os.environ["OPENAI_API_KEY"]
```

Cài  LangChain

```
#!pip install langchain-openai


from langchain_openai import ChatOpenAI


chatModel35 = ChatOpenAI(model="gpt-3.5-turbo-0125")
chatModel4o = ChatOpenAI(model="gpt-4o")
```

# II. Xác định State schema

```
from typing_extensions import TypedDict
from langchain_core.messages import AnyMessage
from typing import Annotated
from langgraph.graph.message import add_messages

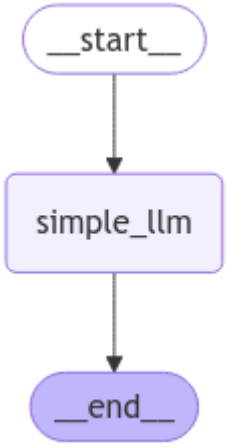

class MessagesState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
```

```
from typing_extensions import TypedDict
```

- `TypedDict` cho phép bạn định nghĩa một kiểu dictionary có cấu trúc rõ ràng, giống như `dataclass`, nhưng dành cho dict.
- `typing_extensions` được dùng để hỗ trợ các tính năng mới khi bạn đang dùng phiên bản Python cũ hơn

```
from langchain_core.messages import AnyMessage
```

- `AnyMessage` là một kiểu tin nhắn (message) tổng quát trong LangChain.
- Có thể là tin nhắn từ người dùng (`HumanMessage`), AI (`AIMessage`), hệ thống (`SystemMessage`)

```
from typing import Annotated
```

- `Annotated` là một cách để **gắn thêm metadata** vào kiểu dữ liệu.
- Không thay đổi kiểu dữ liệu, nhưng cho phép các công cụ/phần mềm xử lý kiểu đó biết thêm thông tin phụ.

Metadata là gì? Xem tại đây

Annotated là gì? Xem tại đây

```
from langgraph.graph.message import add_messages
```

- `add_messages` là một **annotation handler** từ thư viện `langgraph`.
- Nó được dùng để **bổ sung logic** cho cách mà LangGraph xử lý trường `messages`

```
class MessagesState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
```

đây là định nghĩa một State schema

# III. Xác định Node duy nhất của ứng dụng

```
def simple_llm(state: MessagesState):
    return {"messages": [chatModel4o.invoke(state["messages"])]}
```

Đây là một **LangGraph Node Function**. Nó:

1. Nhận vào `state` (dạng dictionary có cấu trúc `MessagesState`)
2. Lấy danh sách tin nhắn từ `state["messages"]`
3. Gửi toàn bộ danh sách đó vào mô hình ngôn ngữ `chatModel4o`
4. Nhận lại một phản hồi từ LLM
5. Trả về phản hồi đó trong một dict mới dưới dạng `{"messages": [<message>]}`

Ví dụ minh họa

Giả sử:

```
state = {
    "messages": [
        HumanMessage(content="Chào bạn"),
    ]
}
```

Gọi

```
simple_llm(state)
```

Trả về

```
{
   "messages": [
      AIMessage(content="Chào bạn! Tôi có thể giúp gì?")
   ]
}
```

Tóm lại

| Thành phần | Vai trò |
| --- | --- |
| `state: MessagesState` | Trạng thái hiện tại (danh sách message) |
| `chatModel4o.invoke(...)` | Gửi prompt đến LLM |
| `return {"messages": [...]}` | Trả về message mới để nối thêm vào state |

**Vì trong ví dụ này chúng ta chỉ có một nút nên chúng ta không cần phải xác định các cạnh.**

# IV. Biên dịch ứng dụng

```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END

# Build graph
builder = StateGraph(MessagesState)

builder.add_node("simple_llm", simple_llm)

# Add the logic of the graph
builder.add_edge(START, "simple_llm")
builder.add_edge("simple_llm", END)

# Compile the graph
graph = builder.compile()

# Visualize the graph
display(Image(graph.get_graph().draw_mermaid_png()))
```

```
from IPython.display import Image, display
```

- Dùng trong Jupyter Notebook để hiển thị hình ảnh (ở đây là sơ đồ luồng graph).
- `display(...)` dùng để hiển thị hình ảnh.
- `Image(...)` nhận vào dữ liệu ảnh hoặc URL.

```
from langgraph.graph import StateGraph, START, END
```

- `StateGraph`: class để **xây dựng graph gồm các bước (nodes)** xử lý state.
- `START` và `END`: các node mặc định dùng để đánh dấu điểm **bắt đầu** và **kết thúc**.

Xây dựng Graph:

```
builder = StateGraph(MessagesState)
```

Khởi tạo 1 builder để tạo ra graph, với `MessagesState` là định nghĩa schema cho **state**.

```
builder.add_node("simple_llm", simple_llm)
```

- Thêm một node có tên `"simple_llm"` và hàm xử lý tương ứng là `simple_llm`.

**Kết nối các bước xử lý:**

```
builder.add_edge(START, "simple_llm")
builder.add_edge("simple_llm", END)
```

`START → simple_llm → END`: bạn đang định nghĩa **luồng chạy của graph** theo thứ tự:

1. **Bắt đầu**
2. Chạy `simple_llm`
3. Kết thúc

**Biên dịch Graph:**

```
graph = builder.compile()
```

**Hiển thị sơ đồ:**

```
display(Image(graph.get_graph().draw_mermaid_png()))
```




V. Chạy ứng dụng

```
from pprint import pprint
from langchain_core.messages import AIMessage, HumanMessage

messages = graph.invoke({"messages": HumanMessage(content="Where is the Golden Gate Bridge?")})

for m in messages['messages']:
    m.pretty_print()
```

- **Gọi Graph để xử lý một prompt** (giống như một chatbot).
- `graph.invoke(...)` là cách chạy luồng xử lý bạn đã định nghĩa bằng `StateGraph`.
- Bạn truyền vào một `dict` có key `"messages"` và giá trị là một **HumanMessage duy nhất**.

Trong thực tế, `"messages"` thường là **list[BaseMessage]**, nhưng LangGraph tự động chuyển đổi nếu bạn truyền một message duy nhất (nó sẽ biến nó thành `[HumanMessage(...)]`)

---### ```python
for m in messages['messages']:
    m.pretty_print()

# Kết quả in ra (ví dụ)

Giả sử AI trả lời như sau, bạn sẽ thấy:

```
Human: Where is the Golden Gate Bridge?
AI: The Golden Gate Bridge is located in San Francisco, California, USA.
```

# Diễn giải toàn bộ quy trình

1. Bạn tạo một **HumanMessage** hỏi AI một câu.

2. Gửi message đó vào graph (đã xây bằng LangGraph).
3. Graph chạy các node, ví dụ `simple_llm`, và thêm `AIMessage` phản hồi vào state.
4. Trả về danh sách messages mới (có cả user + AI).
5. In ra từng message bằng `.pretty_print()`.

## V. Bây giờ chúng ta hãy thêm một công cụ vào ChatModel4o

```
def multiply(a: int, b: int) -> int:
    """Multiply a and b.

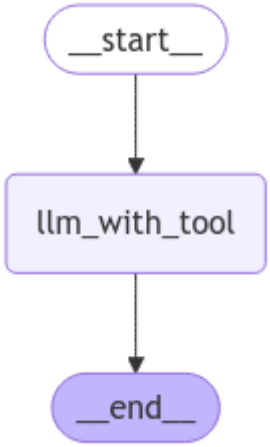
    Args:
        a: first int
        b: second int
    """
    return a * b

chatModel4o_with_tools = chatModel4o.bind_tools([multiply])
```

Định nghĩa một hàm nhân 2 số nguyên `a` và `b`.

`chatModel4o.bind_tools([multiply])`

- Gắn hàm `multiply` vào mô hình `chatModel4o` như một **tool** (công cụ).
- Khi mô hình nhận câu hỏi phù hợp (ví dụ: "*What is 6 times 7?*"), nó có thể **gọi hàm `multiply`** để xử lý thay vì tự trả lời.

# VI. Bây giờ chúng ta hãy tạo một ứng dụng LangGraph thứ hai có thể quyết định xem ứng dụng đó có sử dụng chatbot LLM hay Công cụ Multiply để trả lời câu hỏi của người dùng hay không

```
# Node
def llm_with_tool(state: MessagesState):
    return {"messages": [chatModel4o_with_tools.invoke(state["messages"])]}
```

```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END

# Build graph
builder = StateGraph(MessagesState)
```

```
builder.add_node("llm_with_tool", llm_with_tool)

# Add the logic of the graph
builder.add_edge(START, "llm_with_tool")
builder.add_edge("llm_with_tool", END)

# Compile the graph
graph = builder.compile()

# Visualize the graph
display(Image(graph.get_graph().draw_mermaid_png()))
```




```
from pprint import pprint
from langchain_core.messages import AIMessage, HumanMessage

# The following two lines are the most frequent way to
# run and print a LangGraph chatbot-like app results.
messages = graph.invoke({"messages": HumanMessage(content="Where is the Eiffel Tower?")})

for m in messages['messages']:
    m.pretty_print()
```

================================ Human Message
================================ Where is the Eiffel Tower?
================================== Ai Message
================================== The Eiffel Tower is located in Paris, France. It is situated on the Champ de Mars, near the Seine River.

```
from pprint import pprint
from langchain_core.messages import AIMessage, HumanMessage
```

```
# The following two lines are the most frequent way to
# run and print a LangGraph chatbot-like app results.
messages = graph.invoke({"messages": HumanMessage(content="Multiply 4 and 5")})

for m in messages['messages']:
    m.pretty_print()
```

```
================================ Human Message
================================ Multiply 4 and 5
================================== Ai Message
================================== Tool Calls: multiply
(call_QRyPmp5TdcIlfEOyBrJ9E7V5) Call ID: call_QRyPmp5TdcIlfEOyBrJ9E7V5 Args: a: 4 b: 5
```

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Định tuyến(Router) trong LangGraph

## Router trong LangGraph là gì?

Trong LangGraph, `Router` là một **Node đặc biệt** (tức một bước – `step`) cho phép bạn **tạo ra các nhánh (branches)** dựa trên dữ liệu hoặc trạng thái hiện tại.

Nói đơn giản:

> **Router giống như ngã ba đường** — dựa trên state hiện tại, bạn chọn hướng đi tiếp theo (chuyển tới node A, B hoặc C).

## Khi nào bạn cần dùng Router?

Khi bạn có logic như:

- Nếu người dùng chọn "Coffee" → đi tới node xử lý coffee.
- Nếu người dùng chọn "Tea" → đi tới node xử lý trà.
- Nếu không rõ → hỏi lại hoặc kết thúc.

Bạn không muốn xử lý tất cả trong một node, mà **tách nhánh theo logic**, thì `Router` là giải pháp.

## Cách dùng Router trong LangGraph

```
def my_router(state):
    if state["preference"] == "coffee":
        return "coffee_node"
    elif state["preference"] == "tea":
        return "tea_node"
    else:
        return "unknown_preference"
```

Đây là một **Router node**, tức là nó nhận một `state` (trạng thái hiện tại), và **trả về tên node kế tiếp** dựa trên logic.

> `state["preference"]` là giá trị đầu vào mà bạn đưa vào để quyết định nhánh.

- Nếu thích **coffee**, thì đi tới node `coffee_node`.
- Nếu thích **tea**, thì đi tới node `tea_node`.
- Nếu không rõ, thì về `unknown_preference` (mặc dù node này **chưa được thêm** trong ví dụ – mình sẽ nói thêm ở dưới).

**Đăng ký Router node trong graph**

**Ví dụ sau, Router không thực thi hành động chính mà chọn node tiếp theo dựa trên logic phân nhánh.**

```
builder = StateGraph(StateSchema)
builder.add_node("start", start_node)
builder.add_node("router", my_router)
builder.add_node("coffee_node", coffee_node)
builder.add_node("tea_node", tea_node)

# Router branching
builder.add_edge("start", "router")
builder.add_conditional_edges(
   "router",
   {
      "coffee_node": "coffee_node",
      "tea_node": "tea_node"
   }
)
```

- Node router sử dụng hàm my_router, hàm này sẽ trả về các giá trị là tea_node, coffee_node hoặc unknown_preference, LangGraph dùng kết quả đó để **tra trong từ điển** :
- ```
  {
     "coffee_node": "coffee_node",
     "tea_node": "tea_node"
  }
  ```

- Nếu kết quả là `"coffee_node"` → đi đến node `"coffee_node"`
- Nếu kết quả là `"tea_node"` → đi đến node `"tea_node"`

# Lợi ích của Router
  ```

- **Tách logic phức tạp thành các bước nhỏ**.
- **Tạo flow rõ ràng và dễ debug**.
- **Xây dựng app agent có khả năng phân nhánh tùy theo trạng thái**.
- Dễ áp dụng trong các hệ thống có điều kiện (if/else).

# Gợi ý dùng thực tế

| Use case | Router logic |
|---|---|
| Trợ lý cá nhân | Phân nhánh theo yêu cầu người dùng: tra cứu lịch, gửi mail, hoặc dịch văn bản |
| Ứng dụng học tập | Phân hướng học viên đến bài kiểm tra phù hợp trình độ |
| Ứng dụng đặt hàng | Nếu sản phẩm còn → đặt hàng; nếu không → đề xuất khác |

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# So sánh Router và Node trong LangGraph

Dưới đây là **bài so sánh chi tiết giữa Router và Node** trong LangGraph, giúp bạn dễ phân biệt và áp dụng chính xác trong từng tình huống.

| Tiêu chí | Node (Nút xử lý) | Router (Bộ định tuyến) |
|---|---|---|
| **Chức năng chính** | Xử lý logic cụ thể và trả về dữ liệu mới | Định tuyến (chọn nhánh tiếp theo dựa trên trạng thái) |
| **Trả về** | Một dictionary chứa các giá trị mới cho state | Một chuỗi (string) tên của node tiếp theo |
| **Điều hướng tiếp theo** | Sử dụng `add_edge()` hoặc `add_conditional_edges()` | Sử dụng `add_conditional_edges()` để ánh xạ chuỗi → node |
| **Tính logic** | Chứa logic chính như gọi LLM, xử lý dữ liệu... | Chứa logic quyết định hướng đi tiếp theo |
| **Dữ liệu trả về** | Ví dụ: `{ "graph_state": "Tôi chọn trà" }` | Ví dụ: `"coffee_node"` hoặc `"tea_node"` |
| **Dùng khi nào?** | Khi cần thực hiện một hành động/ghi nhớ dữ liệu | Khi cần rẽ nhánh theo điều kiện hoặc phân luồng logic |

## Ví dụ minh họa

### Node bình thường trả về giá trị mới cho state:

```
def coffee_node(state):
    print("☕ Tôi chọn coffee")
    return {"result": "Bạn sẽ có một ly coffee"}
```

### Router trả về tên của node tiếp theo:

```
def my_router(state):
    if state["preference"] == "coffee":
        return "coffee_node"
    elif state["preference"] == "tea":
        return "tea_node"
    else:
        return "unknown_node"
```

# Khi nào dùng Router thay vì Node?

- Khi bạn cần **rẽ nhánh logic** theo điều kiện → dùng `router`.
- Khi bạn cần **thực hiện hành động cụ thể** (gọi LLM, tính toán, ghi log,...) → dùng `node`.

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Hiểu rõ hơn về add_conditional_edges

Chúng ta lấy 2 ví dụ để so sánh

```
builder.add_conditional_edges("node_1", decide_dring)
```

và

```
builder.add_conditional_edges("router", {
    "coffee_node": "coffee_node",
    "tea_node": "tea_node",
    "fallback_node": "fallback_node"
})
```

Hai dòng `builder.add_conditional_edges(...)` trên **đều dùng để thêm điều kiện rẽ nhánh**, nhưng **cách dùng và mục đích của chúng có sự khác biệt rõ ràng**.

| **Tiêu chí** | **`builder.add_conditional_edges("node_1", decide_dring)`** | **`builder.add_conditional_edges("router", { "coffee_node": ..., ... })`** |
|---|---|---|
| **Kiểu truyền đối số** | Truyền **một hàm** điều kiện (condition function) | Truyền **một dict** ánh xạ giá trị trả về từ router đến các node tương ứng |
| **Ai quyết định rẽ nhánh?** | Hàm `decide_dring(state)` sẽ quyết định đi đến node nào | Hàm `router(state)` sẽ trả về một string → dict ánh xạ string đó đến node tương ứng |
| **Giá trị trả về** của hàm | Trả về tên node dưới dạng string | Trả về tên node dưới dạng string, **phải trùng với key trong dict** |
| **Node được gắn vào** | Gắn vào một node bình thường (`node_1`) | Gắn vào một router node (`router`) |
| **Sử dụng khi** | Khi bạn muốn **node đang xử lý** tự quyết định rẽ nhánh tiếp theo | Khi bạn tách riêng logic định tuyến ra một **router riêng biệt** |
| **Ví dụ hàm điều kiện** | `def decide_dring(state): return "coffee_node" if ... else "tea_node"` | `def router(state): return "coffee_node" / "tea_node" / "fallback_node"` |

| Tiêu chí | `builder.add_conditional_edges("node_1", decide_dring)` | `builder.add_conditional_edges("router", { "coffee_node": ..., ... })` |
|---|---|---|
| **Thường dùng khi** | Bạn muốn `node_1` vừa xử lý vừa quyết định hướng đi tiếp | Bạn muốn tách riêng xử lý (node) và định tuyến (router) một cách rõ ràng |

| Dạng | Ý nghĩa |
|---|---|
| `add_conditional_edges("node_1", decide_dring)` | Node xử lý xong rồi chọn tiếp theo đi đâu. |
| `add_conditional_edges("router", { ... })` | Router chỉ quyết định hướng đi, còn xử lý nằm ở node khác. |

# Khi nào dùng cái nào?

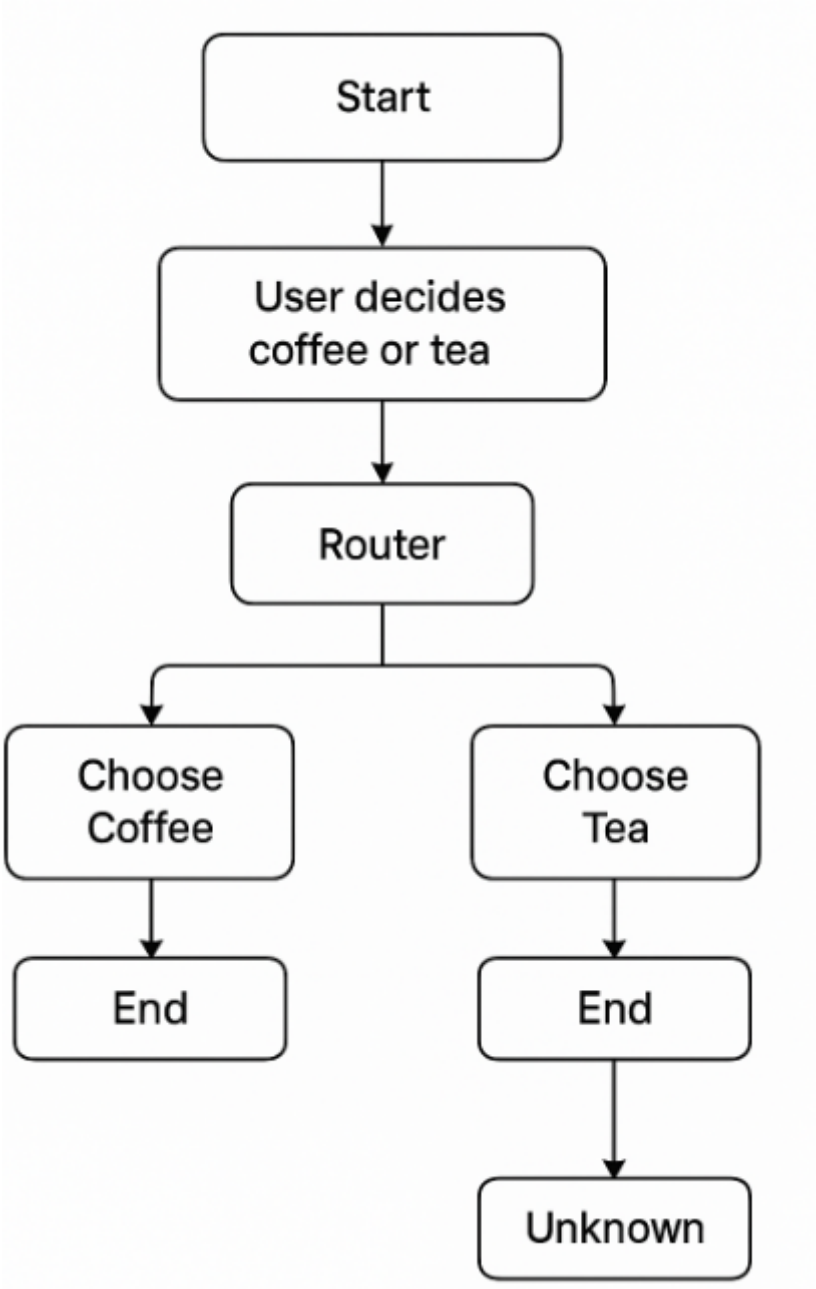
| Nếu bạn muốn... | Hãy dùng... |
|---|---|
| Một node tự quyết định rẽ nhánh tiếp theo | `add_conditional_edges("node_1", decide_dring)` |
| Một router chuyên dùng để rẽ nhánh | `add_conditional_edges("router", { ... })` |

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Bài Thực Hành Router Trong LangGraph

> Mục tiêu: Tạo một LangGraph gồm các bước đơn giản, giúp người dùng quyết định sẽ uống Coffee hay Trà.



I. Cài đặt

```
* cd project_name
* pyenv local 3.11.4
* poetry install
* poetry shell
* jupyter lab
```

Tạo .env file

```
OPENAI_API_KEY=your_openai_api_key
LANGCHAIN_TRACING_V2=true
LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
LANGCHAIN_API_KEY=your_langchain_api_key
LANGCHAIN_PROJECT=your_project_name
```

Kết nối file .env

```
#pip install python-dotenv
```

```
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
openai_api_key = os.environ["OPENAI_API_KEY"]
```

**II. Kết nối LangChain**

```
#!pip install langgraph langchain openai
```

# III. Định nghĩa `State Schema` (State = Memory)

```
from typing import TypedDict, Literal


class DrinkState(TypedDict):
    preference: Literal["coffee", "tea", "unknown"]
```

## `TypedDict`

- Đây là một **tính năng của Python (từ `typing` module)** cho phép bạn khai báo **dictionary có cấu trúc rõ ràng**, giống như một `class`.

- Mục đích là để **giúp IDE và trình kiểm tra kiểu như MyPy bắt lỗi sớm** nếu bạn dùng sai key hoặc giá trị.

### `DrinkState`

- Là tên của lớp, đại diện cho **trạng thái (state)** của ứng dụng LangGraph của bạn.

### `preference: Literal["coffee", "tea", "unknown"]`

- Trường `preference` chỉ chấp nhận **một trong ba giá trị cụ thể**: `"coffee"`, `"tea"` hoặc `"unknown"`.
- Đây là cách giới hạn giá trị hợp lệ, giúp bạn tránh lỗi như `"cofee"` hay `"t"` do gõ sai.

## IV. Định nghĩa các `Nodes` (mỗi Node là một Step)

```
# Node: Bắt đầu
def start_node(state: DrinkState) -> DrinkState:
    print("□□ Bạn thích uống gì hôm nay?")
    choice = input("Nhập 'coffee' hoặc 'tea': ").strip().lower()
    if choice not in ["coffee", "tea"]:
        choice = "unknown"
    return {"preference": choice}

# Node: Nếu chọn coffee
def coffee_node(state: DrinkState) -> DrinkState:
    print("☕ Bạn đã chọn Coffee! Tuyệt vời!")
    return state

# Node: Nếu chọn tea
def tea_node(state: DrinkState) -> DrinkState:
    print("□□ Bạn đã chọn Trà! Thanh tao ghê!")
    return state

# Node: Nếu nhập sai
def fallback_node(state: DrinkState) -> DrinkState:
    print("Không rõ bạn chọn gì. Vui lòng thử lại.")
    return state
```

## V. Định nghĩa Router

```
def drink_router(state: DrinkState) -> str:
    if state["preference"] == "coffee":
```

```
        return "coffee_node"
    elif state["preference"] == "tea":
        return "tea_node"
    else:
        return "fallback_node"
```

# VI. Xây dựng Graph

```
from langgraph.graph import StateGraph

builder = StateGraph(DrinkState)

# Thêm các nodes
builder.add_node("start", start_node)
builder.add_node("coffee_node", coffee_node)
builder.add_node("tea_node", tea_node)
builder.add_node("fallback_node", fallback_node)
builder.add_node("router", drink_router)

# Tạo flow giữa các bước
builder.set_entry_point("start")
builder.add_edge("start", "router")

builder.add_conditional_edges("router", {
    "coffee_node": "coffee_node",
    "tea_node": "tea_node",
    "fallback_node": "fallback_node"
})

# Gắn điểm kết thúc cho mỗi nhánh
builder.set_finish_point("coffee_node")
builder.set_finish_point("tea_node")
builder.set_finish_point("fallback_node")

# Compile graph
app = builder.compile()
```

# VII. Chạy thử

```
# Trạng thái khởi đầu rỗng
initial_state: DrinkState = {"preference": "unknown"}
final_state = app.invoke(initial_state)
```

# Kết quả

Tùy người dùng nhập vào `coffee` hay `tea`, graph sẽ tự động điều hướng tới đúng bước xử lý, và in kết quả tương ứng.

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

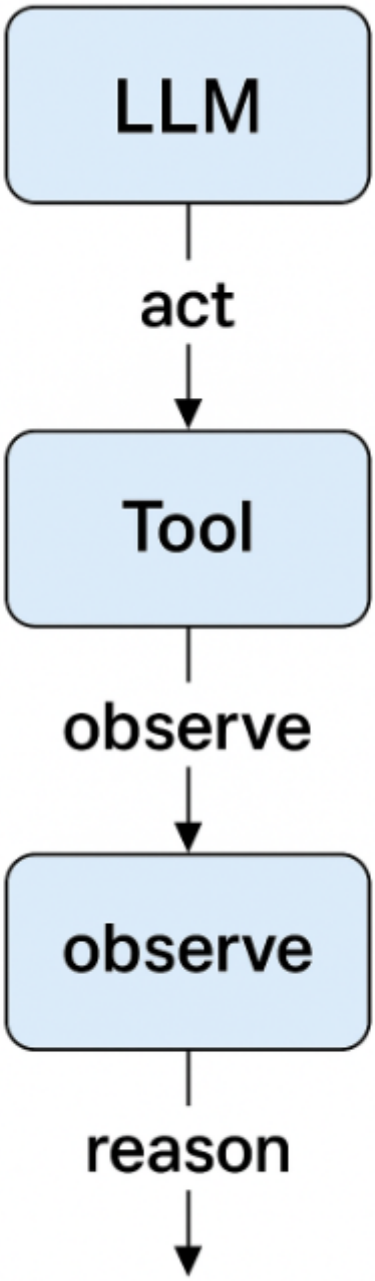
# Giới thiệu về ReAct Architecture

Kiến trúc **ReAct** là một mô hình rất quan trọng trong việc xây dựng các Agent dựa trên LLM. Tên “ReAct” là viết tắt của **Reasoning and Acting** – tức là kết hợp giữa *lập luận* và *hành động*. Đây là cách mà một Agent thông minh có thể thực hiện nhiều bước để hoàn thành một nhiệm vụ, thay vì chỉ trả lời một lần rồi kết thúc.

## Giới thiệu về ReAct Architecture

ReAct cho phép các LLM (Large Language Models) hoạt động theo chu trình:

1. **act (hành động)**
   Mô hình quyết định gọi một công cụ (tool) để thực hiện một hành động cụ thể. Ví dụ: tìm kiếm thông tin, tính toán, tra dữ liệu từ API, v.v.
2. **observe (quan sát)**
   Công cụ thực hiện hành động và trả về kết quả. Kết quả này sẽ được đưa trở lại LLM như một phần của cuộc hội thoại (hoặc “context”).
3. **reason (lập luận)**
   LLM tiếp nhận kết quả, phân tích và quyết định bước tiếp theo: có thể gọi thêm một công cụ khác, tiếp tục hỏi người dùng, hoặc kết thúc bằng một câu trả lời.




**ReAct** là cách mà **AI (LLM)** suy nghĩ và hành động như một con người:

> “Nghĩ – Làm – Quan sát – Nghĩ tiếp – Làm tiếp…”

## Câu chuyện đơn giản

Bạn nói với AI:

> "Tính giúp tôi 3 + 5"

AI làm gì?

| Bước | Hành động | Giải thích dễ hiểu |
|---|---|---|
| **Act** | "Tôi sẽ gọi công cụ máy tính để tính toán" | (Gọi hàm `calculator_tool`) |
| **Tool** | Máy tính nhận yêu cầu, tính ra kết quả `8` | |
| **Observe** | AI nhìn vào kết quả: `8` | (Cập nhật kết quả vào bộ nhớ) |
| **Reason** | AI tự hỏi: “Người dùng muốn tính tiếp không?” | Nếu có, lặp lại từ đầu |

# Ví dụ thực tế

Giả sử bạn hỏi:
**"Hôm nay thời tiết ở TP Hồ Chí Minh như thế nào và tôi có nên mang ô không?"**

Agent thực hiện:

- **act**: Gọi một công cụ thời tiết để lấy dữ liệu thời tiết tại **TP Hồ Chí Minh**.
- **observe**: Nhận được kết quả: "*Trời có khả năng mưa vào chiều tối.*"
- **reason**: LLM suy luận: "*Trời có thể mưa → nên mang ô → trả lời người dùng.*"

Trả lời cuối cùng:
"*Hôm nay **TP Hồ Chí Minh** có khả năng mưa vào chiều tối, bạn nên mang theo ô.*"

# Ứng dụng của ReAct

- Xây dựng **multi-step agents**: Giải quyết các bài toán phức tạp cần nhiều hành động trung gian.
- Cho phép Agent có thể **tự khám phá**, điều chỉnh kế hoạch dựa trên kết quả trung gian.
- Hữu ích trong các hệ thống như **LangGraph**, **LangChain**, nơi bạn cần điều phối nhiều bước và công cụ.

**Ví dụ cơ bản**

Tạo một agent hỏi người dùng một phép tính, gọi một công cụ để tính, rồi tiếp tục hỏi đến khi người dùng muốn dừng.

Kiến trúc Agent như sau

```
START
  ↓
LLM (hỏi phép tính)
  ↓
TOOL (tính kết quả)
  ↓
```

```
LLM (xem kết quả, hỏi tiếp hay kết thúc?)
  → Nếu tiếp → quay lại TOOL
  → Nếu kết thúc → END
```

### 1. Định nghĩa `State`

```
from typing import TypedDict, List, Optional

class CalcState(TypedDict):
    history: List[str]
    next_action: Optional[str]
    question: Optional[str]
    answer: Optional[str]
```

### 2. Tạo Tool Node

```
def calculator_tool(state: CalcState) -> CalcState:
    question = state["question"]
    try:
        answer = str(eval(question))  # WARNING: chỉ dùng với ví dụ đơn giản!
    except:
        answer = "Không thể tính được."

    state["answer"] = answer
    state["history"].append(f"Q: {question}\nA: {answer}")
    return state
```

### 3. Tạo LLM Node

```
def llm_node(state: CalcState) -> CalcState:
    print("□□ LLM: Tôi đang nghĩ...")
    print(f"Câu hỏi: {state.get('question')}")
    print(f"Kết quả: {state.get('answer')}")

    # Quyết định bước tiếp theo
    choice = input("Bạn muốn tiếp tục (yes/no)? ")
    if choice.lower() == "no":
        state["next_action"] = "end"
    else:
        new_q = input("Nhập phép tính mới: ")
        state["question"] = new_q
```

```
        state["next_action"] = "calculate"
    return state
```

### 4. Router

```
def calc_router(state: CalcState) -> str:
    return state["next_action"]
```

### 5. Kết nối các bước bằng LangGraph

```
from langgraph.graph import StateGraph

builder = StateGraph(CalcState)

builder.add_node("llm_node", llm_node)
builder.add_node("tool_node", calculator_tool)
builder.add_node("router", calc_router)

builder.set_entry_point("llm_node")
builder.add_edge("llm_node", "router")

builder.add_conditional_edges("router", {
    "calculate": "tool_node",
    "end": "__end__"
})

builder.add_edge("tool_node", "llm_node")
graph = builder.compile()
```

- Dòng 5,6,7 Chúng ta thêm 3 nodes: llm_node, calculator_tool, calc_router
- Dòng 9, llm_node là được đặt là node đầu tiên
- Dòng 10, chúng ta nối node có id là  llm_node với node có id là `router`**(router là một node đặc biệt, node này không trả về một state mà trả về một chuỗi)**
- Dòng 12, router thực hiện rẽ nhánh dựa trên add_conditional_edges
  - "router" là id của node calc_router, nếu calc_router trả về "calculate" thì thực hiên tool_node, nếu calc_router trả về "end" thì thực hiện "__end__"
- Dòng 17, thực hiện nối "tool_node" với "llm_node" và tiếp tục

### 6. Chạy thử

```
state = {
    "history": [],
```

```
    "next_action": None,
    "question": "2 + 3",
    "answer": None,
}

graph.invoke(state)
```

**Output mẫu**

```
LLM: Tôi đang nghĩ...
Câu hỏi: 2 + 3
Kết quả: None
Nhập phép tính mới: 2 + 3
Q: 2 + 3
A: 5
Bạn muốn tiếp tục (yes/no)? yes
Nhập phép tính mới: 10 * 2
...
```

# Tóm tắt kiến trúc ReAct

| Pha | Vai trò | Ý nghĩa |
|---|---|---|
| `Act` | **LLM gọi công cụ** | LLM yêu cầu tính một biểu thức toán học |
| `Observe` | **Công cụ trả kết quả** | Công cụ `calculator_tool` xử lý phép tính và trả về kết quả |
| `Reason` | **LLM quyết định bước tiếp theo** | Dựa trên kết quả, LLM quyết định tiếp tục hay dừng |
| `Tool` | **Công cụ được LLM gọi** | Ở đây là hàm `calculator_tool` dùng `eval()` để tính toán |

**Phân tích chi tiết ví dụ trên**

1. **Act – LLM gọi công cụ**

Trong hàm `llm_node`, nếu người dùng muốn tiếp tục:

```
state["next_action"] = "calculate"
```

LLM quyết định hành động tiếp theo là **gọi công cụ tính toán**.

2. **Observe – Công cụ trả kết quả**

```
def calculator_tool(state: CalcState) -> CalcState:
    ...
    state["answer"] = answer
```

Công cụ thực hiện tính toán và **cập nhật lại state** (trạng thái) với kết quả.
LLM sẽ **quan sát lại kết quả** ở bước tiếp theo.

3. **Reason – LLM quyết định tiếp theo**

Sau khi đã có kết quả trong `state["answer"]`, LLM đánh giá và hỏi:

```
choice = input("Bạn muốn tiếp tục (yes/no)? ")
```

Nếu **yes** thì act lần nữa (gọi lại công cụ).
Nếu **no** thì kết thúc (gọi `"end"`).

4. **Tool – Công cụ độc lập**

```
def calculator_tool(state: CalcState) -> CalcState:
    ...
```

Đây chính là một công cụ (Tool) – thực hiện công việc cụ thể do LLM yêu cầu.
Nó không thông minh, chỉ đơn thuần xử lý dữ liệu đầu vào.

**Tổng kết dễ hiểu**

| Bước | Thành phần | Vai trò |
|---|---|---|
| 1 | `llm_node` | Act: Quyết định gọi tool |
| 2 | `calculator_tool` | Observe: Trả kết quả về |
| 3 | `llm_node` | Reason: Xem kết quả và quyết định hành động tiếp theo |

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Thực hành – Node LLM quyết định bước tiếp theo

Chúng ta sẽ xây dựng lại ứng dụng trước đó, nhưng lần này sẽ bổ sung hành vi mang tính "agent" điển hình. Theo cách gọi của nhóm LangGraph, đây sẽ là **Agent đầu tiên** của chúng ta.

**Những điểm chính:**

- **Hành vi agent điển hình**: node công cụ (tool node) sẽ phản hồi lại cho LLM, và sau đó LLM sẽ quyết định bước tiếp theo.
- Node bao gồm chatbot, công cụ (tool), và thông điệp hệ thống (system message).
- Các thao tác gọi công cụ theo trình tự (sequential tool-calling operations).

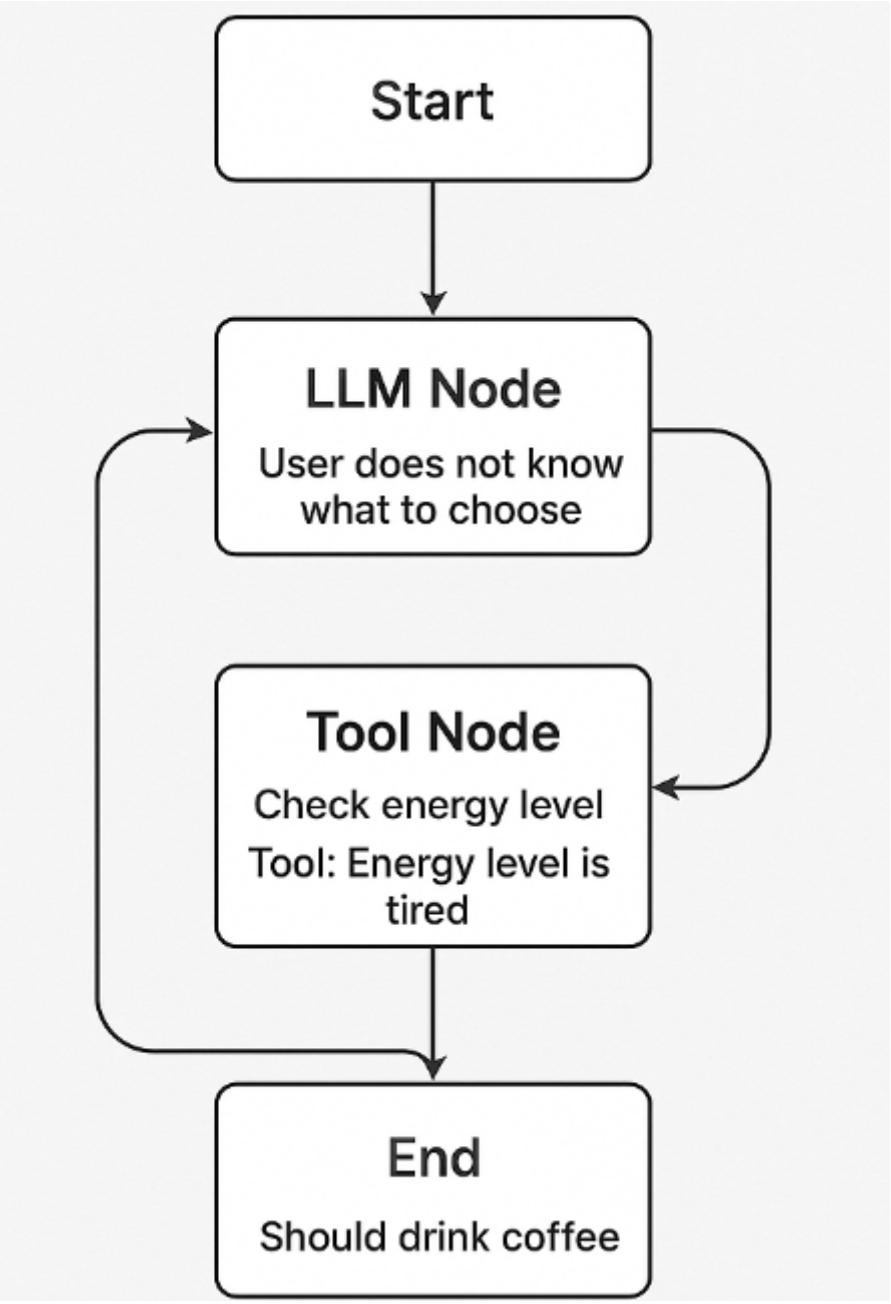
# Agent Cơ Bản

- **Mục tiêu**: Chúng ta sẽ xây dựng một ứng dụng tương tự như trong bài thực hành trước. **Điểm khác biệt chính** là trong ứng dụng lần này, **node công cụ (tools node) sẽ phản hồi trở lại cho assistant (LLM) thay vì đi thẳng đến node KẾT THÚC (END Node)**.
- Điều này thể hiện kiến trúc được gọi là **ReAct**: mô hình LLM sẽ tiếp tục sử dụng các công cụ cho đến khi nhận được phản hồi thỏa đáng.
  - **act** – LLM gọi một công cụ.
  - **observe** – công cụ trả kết quả lại cho LLM.
  - **reason** – LLM quyết định bước tiếp theo (gọi công cụ khác hoặc trả lời trực tiếp).

# Thực hành

Tạo một agent sử dụng kiến trúc ReAct với quy trình:

1. Người dùng không biết chọn gì → LLM hỏi ý kiến.
2. LLM gọi một tool để kiểm tra trạng thái năng lượng người dùng.
3. Tool trả lời về mức năng lượng.
4. Dựa vào đó, LLM đưa ra lời khuyên nên uống trà hay cà phê.




1. **Cài đặt và chuẩn bị**

```
pip install langgraph openai
```

2. **Định nghĩa State**

```
from typing import TypedDict, Literal, Optional

class DrinkState(TypedDict):
    preference: Literal["coffee", "tea", "unknown"]
    energy_level: Optional[str]
    messages: list[dict]
```

3. **Tạo công cụ kiểm tra năng lượng**

```
def check_energy_level(state: DrinkState) -> DrinkState:
    # Giả sử công cụ xác định người dùng đang "mệt"
    energy = "tired"
    state["energy_level"] = energy
    state["messages"].append({"role": "tool", "content": f"Energy level is {energy}"})
    return state
```

### 4. Tạo node LLM để đưa ra quyết định

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(temperature=0)

def llm_node(state: DrinkState) -> DrinkState:
    messages = state["messages"]
    response = llm.invoke(messages)
    state["messages"].append({"role": "assistant", "content": response.content})
    if "coffee" in response.content.lower():
        state["preference"] = "coffee"
    elif "tea" in response.content.lower():
        state["preference"] = "tea"
    return state
```

### 5. Tạo Graph

```
from langgraph.graph import StateGraph

builder = StateGraph(DrinkState)

builder.add_node("llm", llm_node)
builder.add_node("check_energy", check_energy_level)

builder.set_entry_point("llm")

# LLM gọi tool rồi quay lại chính nó
builder.add_edge("llm", "check_energy")
builder.add_edge("check_energy", "llm")
```

```
# Thiết lập điểm kết thúc
builder.set_finish_point("llm")

graph = builder.compile()
```

**6. Chạy thử**

```
initial_state = {
    "preference": "unknown",
    "energy_level": None,
    "messages": [{"role": "user", "content": "Tôi không biết nên uống gì hôm nay"}],
}

final_state = graph.invoke(initial_state)
print("Gợi ý của Agent:", final_state["preference"])
```

# Giải thích luồng hoạt động

1. Người dùng không đưa ra lựa chọn cụ thể.
2. LLM phân tích và quyết định cần kiểm tra mức năng lượng.
3. Tool `check_energy` trả về trạng thái "mệt".
4. LLM đưa ra khuyến nghị uống cà phê để tăng tỉnh táo.
5. Kết thúc.

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Kết nối các công cụ (tools) bằng bind_tools

`bind_tools` là một tính năng trong LangChain (thường dùng với các mô hình như OpenAI's GPT) để **kết nối các công cụ (tools)** với LLM. Nó cho phép mô hình gọi những tool đó khi cần, như tính toán, truy xuất thông tin, tra cứu, v.v.

`ind_tools` giúp bạn nói với LLM rằng:

> "Ê, ngoài việc trả lời bằng văn bản, mày có thể **gọi** mấy tool này nếu thấy cần."

Cú pháp cơ bản

```
llm_with_tools = chat_model.bind_tools(
    tools=[tool_1, tool_2],
    parallel_tool_calls=False  # hoặc True nếu muốn gọi song song nhiều tool
)
```

- `tools`: danh sách các tool bạn định sử dụng (được định nghĩa trước).
- `parallel_tool_calls`: nếu `True`, mô hình có thể gọi nhiều tool một lúc (song song). Nếu `False`, gọi từng cái một theo thứ tự.

Ví dụ đơn giản

Bước 1: Định nghĩa các tool

```
from langchain.tools import tool

@tool
def add(a: int, b: int) -> int:
    """Cộng hai số"""
    return a + b

@tool
def multiply(a: int, b: int) -> int:
    """Nhân hai số"""
```

```
    return a * b
```

Bước 2: Bind tools vào mô hình

```
from langchain.chat_models import ChatOpenAI

chat_model = ChatOpenAI(model="gpt-4")

llm_with_tools = chat_model.bind_tools(
    tools=[add, multiply],
    parallel_tool_calls=False  # toán học thường nên làm tuần tự
)
```

Bước 3: Gọi mô hình và để nó tự chọn tool

```
response = llm_with_tools.invoke("Hãy cộng 3 và 5 giúp tôi")
print(response.content)
```

Nếu LLM hiểu rằng cần thực hiện phép cộng, nó sẽ tự động gọi tool `add(3, 5)` và trả lại kết quả là `8`.

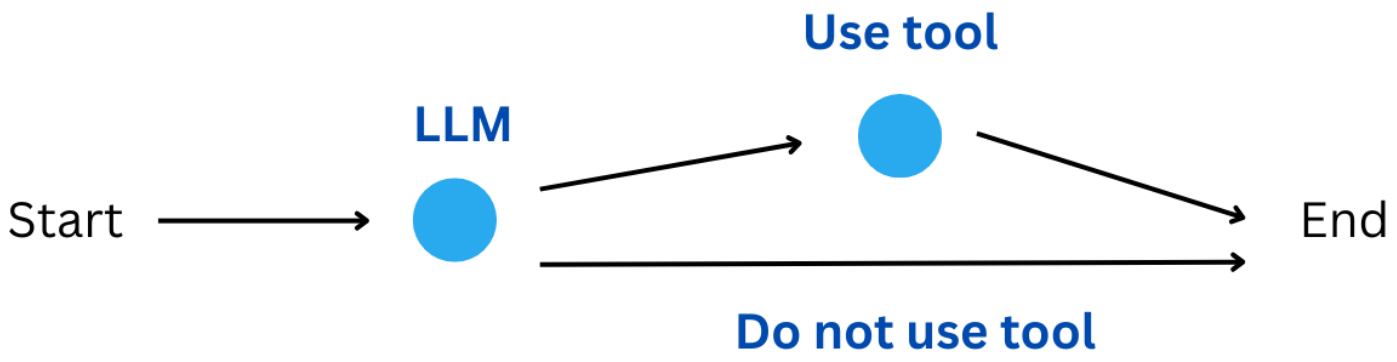
`bind_tools` hữu ích vì

- **Modular hóa**: LLM không cần biết cách hoạt động nội bộ của tool, chỉ cần mô tả chức năng.
- **Mở rộng dễ dàng**: Bạn có thể thêm bao nhiêu tool tùy thích.
- **Tự động lựa chọn**: LLM tự chọn tool phù hợp dựa trên yêu cầu người dùng.

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Xây dựng một ứng dụng quyết định xem có nên trò chuyện bằng LLM hay sử dụng công cụ




1. Cài đặt

```
* cd project_name
* pyenv local 3.11.4
* poetry install
* poetry shell
* jupyter lab
```

2. Tạo .env file

```
* OPENAI_API_KEY=your_openai_api_key
* LANGCHAIN_TRACING_V2=true
* LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
* LANGCHAIN_API_KEY=your_langchain_api_key
* LANGCHAIN_PROJECT=your_project_name
```

3. Kết nối file .env

```
#pip install python-dotenv
```

```
import os
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
openai_api_key = os.environ["OPENAI_API_KEY"]
```

## 4. Cài đặt LangChain

```
#!pip install langchain
#!pip install langchain-openai
```

```
from langchain_openai import ChatOpenAI

chatModel35 = ChatOpenAI(model="gpt-3.5-turbo-0125")
chatModel4o = ChatOpenAI(model="gpt-4o")
```

## 5. LLM với các công cụ

```
from langchain.tools import tool

@tool
def multiply(a: int, b: int) -> int:
    """Multiply a and b.

    Args:
        a: first int
        b: second int
    """
    return a * b

@tool
def add(a: int, b: int) -> int:
    """Adds a and b.

    Args:
        a: first int
        b: second int
    """
    return a + b
```

```
@tool
def divide(a: int, b: int) -> float:
    """Divide a and b.

    Args:
        a: first int
        b: second int
    """
    return a / b

tools = [add, multiply, divide]

llm_with_tools = chatModel4o.bind_tools(tools, parallel_tool_calls=False)
```

# Define the State schema

```
from langgraph.graph import MessagesState

class MessagesState(MessagesState):
    # Add any keys needed beyond messages, which is pre-built
    pass
```

# Define the first Node

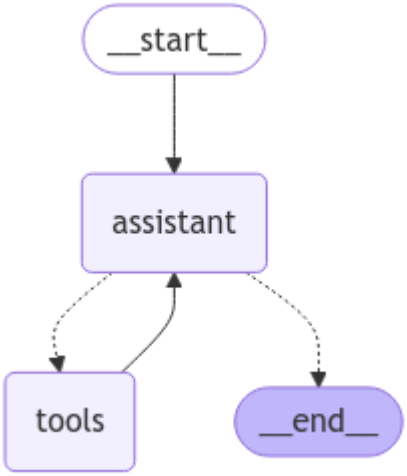
```
from langchain_core.messages import HumanMessage, SystemMessage

# System message, system prompt
sys_msg = SystemMessage(content="You are a helpful assistant tasked with performing arithmetic on a set of inputs.")

# Node
def assistant(state: MessagesState):
   return {"messages": [llm_with_tools.invoke([sys_msg] + state["messages"])]}
```

* Các nút của đồ thị được định nghĩa là các hàm python.
* Đối số đầu tiên của hàm Node là trạng thái. Do đó, trong bài tập này, mỗi nút có thể truy cập khóa `messages`, với `state['messages']`.
* Trong ví dụ này, chúng ta sẽ bắt đầu bằng một nút tương tự như nút chúng ta đã tạo trong bài tập trước, **nhưng lần này bằng SystemMessage**:

# Kết hợp các nút và cạnh để xây dựng đồ thị

```
from langgraph.graph import START, StateGraph
from langgraph.prebuilt import tools_condition
from langgraph.prebuilt import ToolNode
from IPython.display import Image, display

# Build graph
builder = StateGraph(MessagesState)

builder.add_node("assistant", assistant)
builder.add_node("tools", ToolNode(tools))

# Add the logic of the graph
builder.add_edge(START, "assistant")

builder.add_conditional_edges(
    "assistant",
    # If the latest message (result) from assistant is a tool call -> tools_condition routes to tools
    # If the latest message (result) from assistant is a not a tool call -> tools_condition routes to END
    tools_condition,
)

# PAY ATTENTION HERE: from the tools node, back to the assistant
builder.add_edge("tools", "assistant")

# PAY ATTENTION: No END edge.

# Compile the graph
react_graph = builder.compile()

# Visualize the graph
display(Image(react_graph.get_graph(xray=True).draw_mermaid_png()))
```

* act - LLM gọi một công cụ.
* observer - công cụ chuyển đầu ra trở lại LLM.
* reason - LLM quyết định phải làm gì tiếp theo (gọi một công cụ khác hoặc phản hồi trực tiếp).




## Chạy ứng dụng

```
messages = [HumanMessage(content="What was the relationship between Marilyn and JFK?")]
messages = react_graph.invoke({"messages": messages})

for m in messages['messages']:
    m.pretty_print()
```

```
================================ Human Message
=================================

What was the relationship between Marilyn and JFK?
================================== Ai Message
==================================

Marilyn Monroe and President John F. Kennedy (JFK) are often rumored to have had a romantic relationship, though concrete details about the nature and extent of their relationship remain speculative and are part of popular lore. The most famous public connection between them was Marilyn Monroe's sultry performance of "Happy Birthday, Mr. President" at a Democratic Party fundraiser and early birthday celebration for Kennedy in May 1962. However, both contemporaneous accounts and historical research have not conclusively proven a long-term affair, and much of the speculation is based on anecdotal evidence and hearsay.
```

```
messages = [HumanMessage(content="Add 3 and 4. Multiply the output by 2. Divide the output by 5")]
messages = react_graph.invoke({"messages": messages})

for m in messages['messages']:
    m.pretty_print()
```

```
================================ Human Message
================================

Add 3 and 4. Multiply the output by 2. Divide the output by 5
================================== Ai Message
==================================
Tool Calls:
  add (call_UR7Cp0pbVpRyof8Dyq4kZUHm)
 Call ID: call_UR7Cp0pbVpRyof8Dyq4kZUHm
  Args:
    a: 3
    b: 4
================================= Tool Message
=================================
Name: add

7
================================== Ai Message
==================================
Tool Calls:
  multiply (call_OiwYtGUW13AY8CB3qY3Y3Vlt)
 Call ID: call_OiwYtGUW13AY8CB3qY3Y3Vlt
  Args:
    a: 7
    b: 2
================================= Tool Message
=================================
Name: multiply

14
...
2.8
================================== Ai Message
==================================

The final result after adding 3 and 4, multiplying the result by 2, and then dividing by 5 is 2.8.
```

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**

# Tóm tắt chương

**Tạm dừng một chút để cùng nhau ôn lại những gì chúng ta đã học được nhé!**

Chúng ta bắt đầu chương này với cảm giác khá lo lắng – có lẽ bạn cũng từng nghĩ: "Cái LangGraph này nghe thật rối rắm và phức tạp", đúng không? Những khái niệm nghe có vẻ như dành riêng cho những người có siêu năng lực, phải có bằng tiến sĩ hay ít nhất là thạc sĩ mới hiểu nổi cách dùng nó!

Nhưng rồi khi đi đến cuối chương, ta nhận ra rằng **ai cũng có thể xây dựng ứng dụng với LangGraph**. Đằng sau những thuật ngữ nghe phức tạp đó thật ra lại là những khái niệm rất đơn giản. Và chúng ta đã thấy rằng **bất kỳ ai cũng có thể tạo ra những agent thông minh, thậm chí là những ứng dụng có thể thay đổi cả thế giới**.

# Vậy cụ thể, ta đã làm gì?

- Đầu tiên, chúng ta học các **khái niệm cơ bản** của một ứng dụng LangGraph: state là gì, schema của state là gì, node là gì, edge là gì, quá trình compile là gì...
- Sau đó, ta xây dựng một app đơn giản dự đoán người dùng sẽ chọn Trà hay Coffee.
- Ta học được quy trình 5 bước để xây dựng một ứng dụng LangGraph:
  1. Định nghĩa state.
  2. Thêm node và edge.
  3. Biên dịch (compile).
  4. Hiển thị sơ đồ (visualize).
  5. Chạy ứng dụng.

Đơn giản đúng không nào?

- Tiếp theo, ta xây dựng một **chatbot** sử dụng LangGraph, học về khái niệm **reducer** – một từ nghe "kêu" nhưng thực chất chỉ là cách để cập nhật lại state theo cách riêng của mình.
- Ta sử dụng các **module có sẵn** như `message state`, giúp giảm lượng code cần viết.
- Sau đó, ta nâng cấp chatbot bằng cách **thêm tool** và **xây dựng hành vi agent**, nơi mà ứng dụng quyết định dùng chatbot hay gọi tool để trả lời người dùng.
- Ta học cách dùng `tool node` và `tool condition`, tạo router để định tuyến theo điều kiện đầu vào.

Và cuối cùng, **ta đã tạo ra agent đầu tiên của mình** bằng cách cho phép tool phản hồi trở lại LLM – chính là mô hình **ReAct**: act (gọi tool), observe (quan sát kết quả), reason (suy luận xem làm gì tiếp theo).

---

# Sắp tới là gì?

Trong chương tiếp theo, chúng ta sẽ khám phá một chủ đề cực kỳ quan trọng trong thế giới LangGraph: **Memory (Bộ nhớ)**.

Dù trước giờ ta đã nói về state và một chút về memory, nhưng bạn sẽ thấy rằng, trong LangGraph, **memory không đơn giản chỉ là lưu trữ thông tin – nó là cốt lõi để tạo ra những agent thông minh và có tính phản hồi cao**.

Hãy chuẩn bị tinh thần để bước vào một chương siêu hấp dẫn nhé!

**Tác giả: Đỗ Ngọc Tú**

**Công Ty Phần Mềm VHTSoft**