AI VIET NAM – AIO COURSE 2024

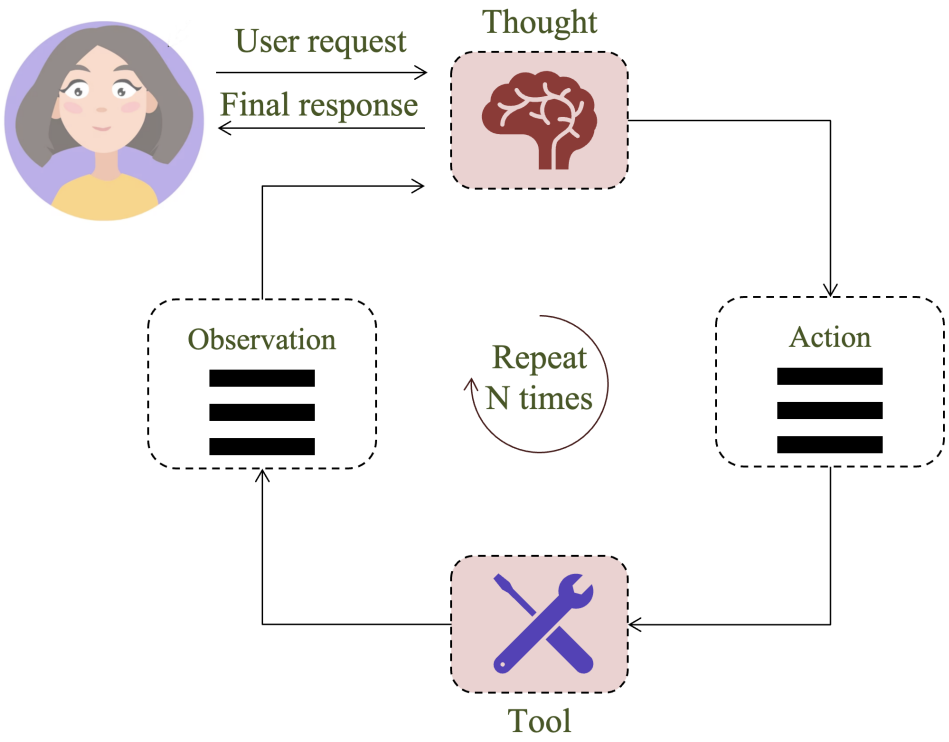
Reasoning Agents

ReAct Agent & Plan-and-Execute Agent using LangGraph

Quoc-Thai Nguyen, Quang-Hien Ho và Quang-Vinh Dinh

Ngày 11 tháng 5 năm 2025

Phần 1. Giới thiệu



Hình 1: ReAct Agent using LangGraph.

Mô hình ngôn ngữ lớn (LLM) đã tạo ra một bước đột phá trong khả năng xử lý ngôn ngữ tự nhiên. Tuy nhiên, để ứng dụng hiệu quả vào các tác vụ thực tế đòi hỏi khả năng tư duy nhiều bước, hành động có mục tiêu và phản ứng linh hoạt, chúng ta cần triển khai AI agents – những cấu trúc có khả năng lập kế hoạch, hành động, quan sát và suy luận. Hiện nay có nhiều thiết kế cho phép các Agent có khả năng tương tác và suy luận thông minh để hoàn thành mục tiêu, điển hình như ReAct, Reflexion, Plan-and-Execute hoặc Multi-Agents.

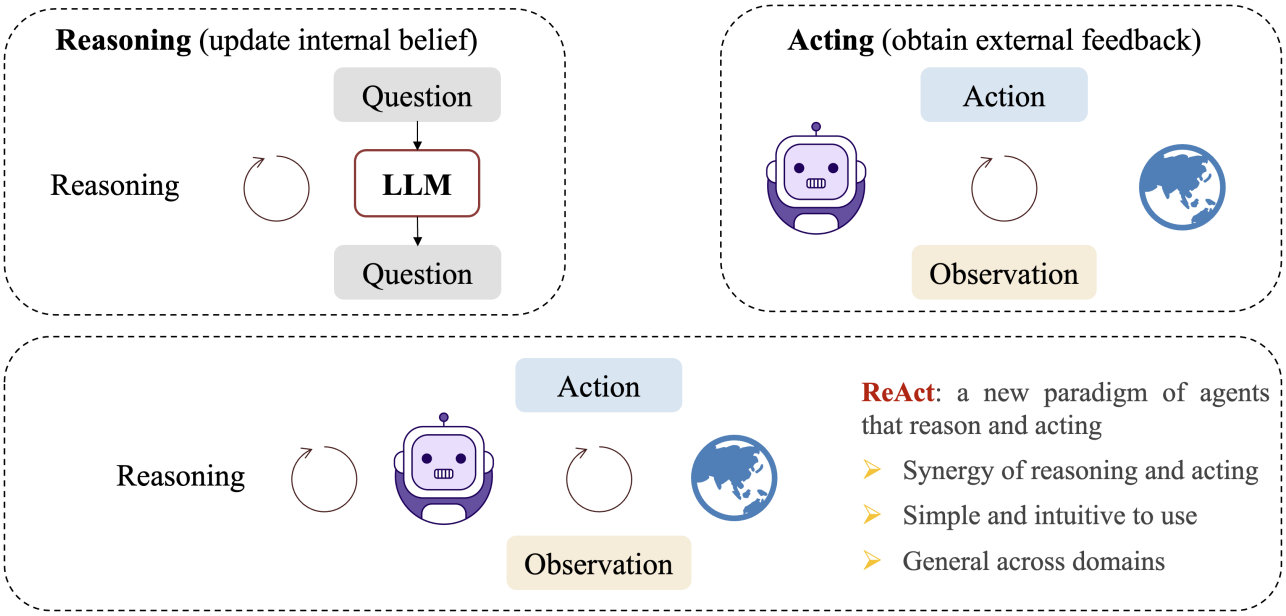
Trong phần nội dung này, chúng ta sẽ tìm hiểu hai kiến trúc reasoning agents nổi bật hiện nay là:

- **ReAct**: Reasoning and Acting – kết hợp suy nghĩ và hành động qua nhiều vòng lặp.
- **Plan-and-Execute**: tách bạch hoàn toàn giữa giai đoạn lên kế hoạch và thực thi.

Sau đó sẽ triển khai các agents thông qua thư viện LangGraph.

Phần 2. ReAct Agent

ReAct (Reasoning + Acting) là một phương pháp kết hợp giữa suy luận và hành động, cho phép các mô hình ngôn ngữ lớn (LLMs) vừa lý luận về các nhiệm vụ vừa thực hiện hành động liên quan đến nhiệm vụ đó. ReAct lần đầu tiên được giới thiệu là phương pháp tận dụng khả năng Reasoning của LLMs để giải quyết những tác vụ phức tạp trong bài báo: ReAct: Synergizing Reasoning and Acting in Language Models



Hình 2: ReAct là sự kết hợp ưu điểm của khả năng Reasoning + Acting LLMs.

Các mô hình chỉ có reasoning (suy luận) thường dựa hoàn toàn vào khả năng ngôn ngữ của LLM để tạo ra lời giải. Mặc dù chúng có thể mô phỏng tư duy logic, nhưng không thể tương tác với thế giới bên ngoài, như tìm kiếm thông tin mới, gọi công cụ, hay xác minh giả định. Điều này khiến chúng dễ bị giới hạn trong những gì đã được học trong quá trình huấn luyện, và dễ sinh ra câu trả lời "có vẻ hợp lý nhưng sai".

Ngược lại, các mô hình chỉ có acting (hành động) – ví dụ như các hệ thống gán tool cho prompt một cách trực tiếp – có thể thực thi tác vụ nhanh chóng, nhưng lại thiếu năng lực suy luận để quyết định khi nào nên gọi tool nào, hoặc nên dừng lại khi đã đủ thông tin. Chúng có thể bị "ngây thơ", thực hiện hành động không cần thiết, hoặc không biết xử lý khi tool trả về kết quả không mong muốn.

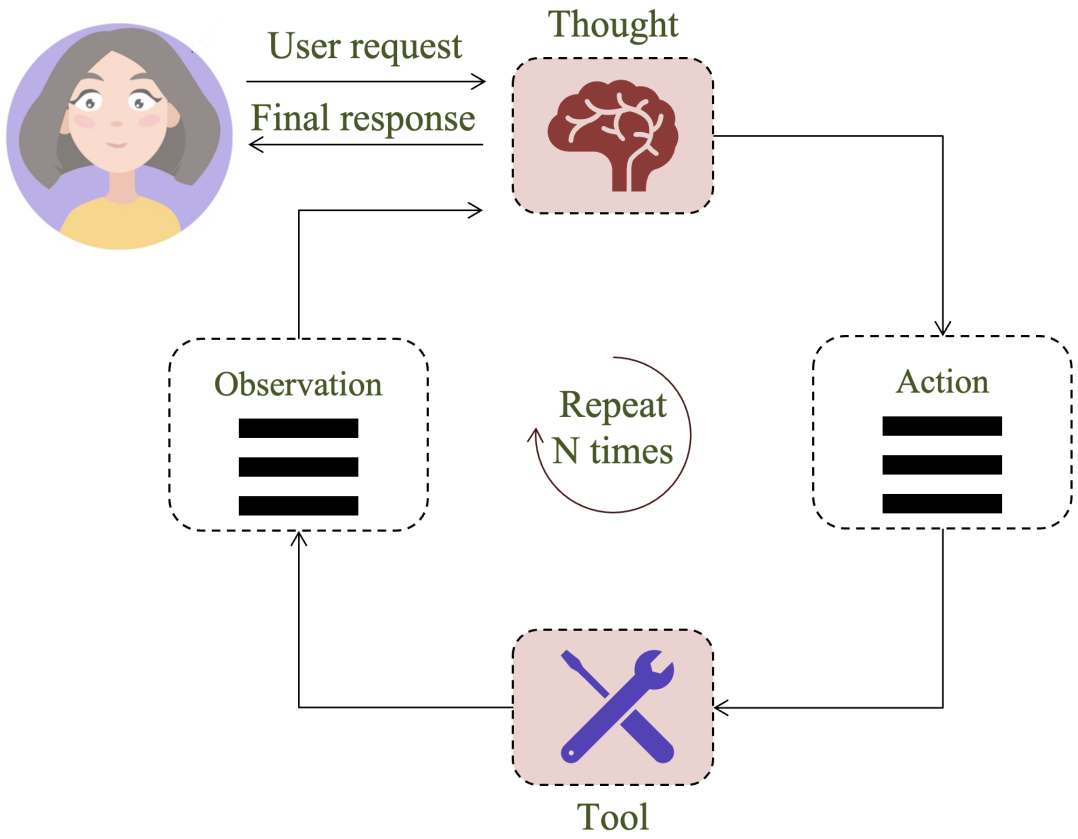
ReAct kết hợp cả hai thế giới: Nó cho phép mô hình "nghĩ thành tiếng" trước khi ra quyết định hành động, và sau khi hành động xong, quan sát kết quả để điều chỉnh suy luận tiếp theo. Điều này tạo ra một vòng lặp Thought → Action → Observation, giúp mô hình:

- Hiểu rõ về bối cảnh trước khi hành động
- Tự sửa sai nếu kết quả quan sát không như mong đợi
- Chọn lựa hành động một cách có lý do, thay vì theo rule cố định
- Kết hợp nhiều công cụ linh hoạt, ví dụ: tra cứu → tính toán → lập kế hoạch → trả lời

Nói cách khác, ReAct giống như một con người đang giải một bài toán: suy nghĩ, thử hành động, nhìn kết quả, rồi tiếp tục suy nghĩ – chứ không đơn thuần là "trả lời ngay" hay "làm ngay".

Phương pháp này hoạt động dựa trên việc kết hợp:

- Reasoning (Suy luận): Mô hình suy nghĩ về vấn đề, phân tích tình huống và lập kế hoạch cho hành động tiếp theo.
- Acting (Hành động): Mô hình thực hiện các hành động dựa trên suy luận của mình, thu thập thông tin mới và tiếp tục quá trình.

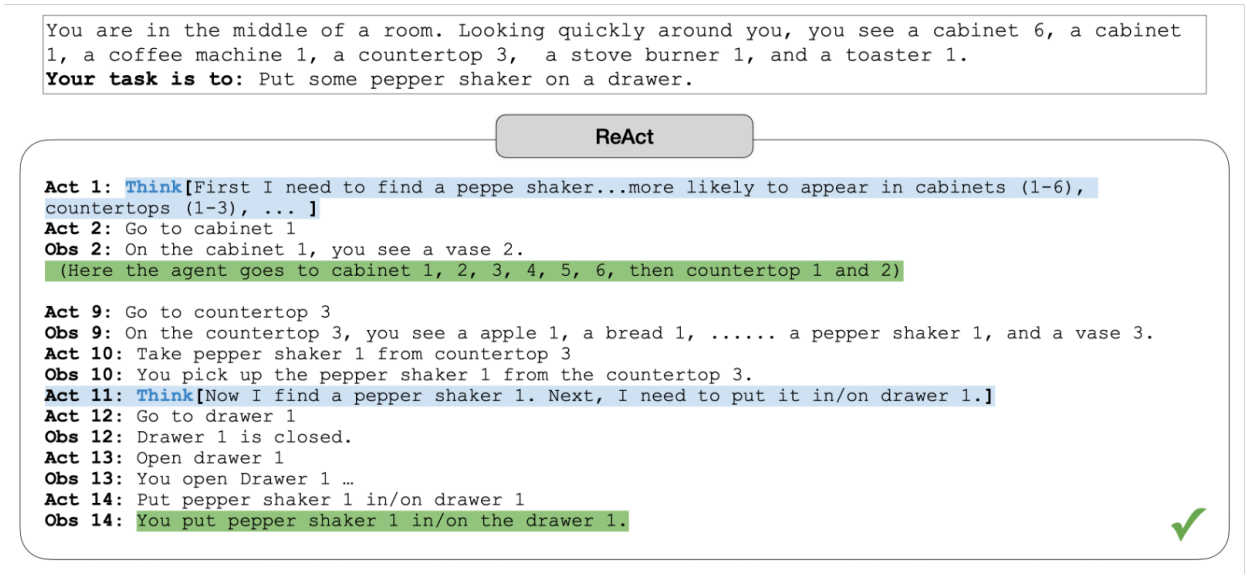


Hình 3: ReAct Agent sử dụng thư viện LangGraph.

ReAct thực hiện một quy trình lặp đi lặp lại gồm các bước sau:

- Suy luận (Thought): Mô hình suy nghĩ về tình huống hiện tại, phân tích các thông tin đã có và cân nhắc các bước tiếp theo. Mục tiêu để hoàn thiện tác vụ đặt ra và hiểu rõ khả năng hoàn thiện các tác vụ cho mỗi chu trình.
- Hành động (Action): Dựa trên suy luận, mô hình thực hiện một hành động cụ thể (ví dụ: tìm kiếm thông tin, gọi API, thực hiện tính toán). Vì vậy, việc định nghĩa và xác định các công cụ hữu ích cho quá trình thực hiện sẽ tối ưu tính hiệu quả khi xử lý các tác vụ khác nhau của Agent.
- Quan sát (Observation): Sau khi thực hiện hành động, mô hình quan sát kết quả thu được. Dựa trên kết quả hành động và yêu cầu cho hành động để gửi thông điệp xem xét tính hiệu quả của mỗi chu trình thực hiện.
- Tiếp tục chu trình: Quay lại bước suy luận với thông tin và quan sát mới, tiếp tục quá trình cho đến khi đạt được mục tiêu. Số lượng chu trình thực hiện có thể được ước lượng cho trước để đảm bảo Agent có thể dừng chu trình suy luận trong trường hợp chưa tìm được câu trả lời phù hợp.

Ví dụ về quá trình suy luận của ReAct thực hiện để giải quyết tác vụ phức tạp được mô tả như hình sau:



Hình 4: Ví dụ về quá trình suy luận của ReAct.

Trong phần này, chúng ta sẽ sử dụng thư viện LangGraph để xây dựng ReAct Agent

1. Cài đặt thư viện

Cài đặt một số thư viện và chuẩn bị các Key cho quá trình sử dụng LLMs cũng như lưu trữ thông tin thông qua công cụ LangSmith. Sau đó xây dựng lớp AgentState để thực thi gồm các thuộc tính như sau:

- messages: lưu trữ các thông tin trong quá trình thực hiện
- steps: để xác định số chu trình đã thực hiện

```
# Install libs
!pip install -U langchain==0.3.24 langchain-openai==0.3.14 langgraph==0.3.33
    langchain-tavily==0.1.6

import os
os.environ["LANGCHAIN_API_KEY"] = ### Your-key
os.environ["LANGCHAIN_PROJECT"] = "Reasoning-Agent"
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["TAVILY_API_KEY"] = ### Your-key
os.environ["OPENAI_API_KEY"] = ### Your-key

from typing import Annotated,Sequence, TypedDict

from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages

class AgentState(TypedDict):
    """The state of the agent."""
    messages: Annotated[Sequence[BaseMessage], add_messages]
    number_of_steps: int
```

2. Định nghĩa các công cụ

Trong phần này, chúng ta xây dựng 2 công cụ để thực hiện hành động là tìm kiếm và thực hiện phép tính nhân. Bên cạnh đó, chúng ta sử dụng mô hình chatGPT để thực hiện.

```
from langchain_tavily import TavilySearch
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain_core.messages import SystemMessage

@tool
def triple(num: float) -> float:
    """
    :param num: a number to triple
    :return: the number tripled ->  multiplied by 3
    """
    return 3 * float(num)


tools = [TavilySearch(max_results=1), triple]

system_prompt = SystemMessage(
"""
You are a reasoning agent. Always think step-by-step.

Use the following format:

Thought: what you are thinking
Action: the action to take, e.g. ‘search‘, ‘calculate‘
Action Input: the input to the action
Observation: the result of the action

(Repeat Thought/Action/Observation if needed)

Final Answer: your answer to the user

Question: {input}
"""
)

model = ChatOpenAI(model="gpt-4.1-nano")
```

3. Xây dựng các hàm thực hiện các tác vụ

Sau khi đã có các nodes tương ứng là mô hình và công cụ, chúng ta xây dựng các hàm để thực hiện lời gọi đến mô hình ngôn ngữ hoặc các công cụ tương ứng. Gồm 3 hàm như sau:

- tool_node: thực hiện lời gọi đến các công cụ và mô hình sẽ quyết định nên gọi đến công cụ nào cho một lần thực hiện hành động
- call_model: thực hiện lời gọi đến mô hình chatGPT thông qua API
- should_continue: định nghĩa hàm kiểm tra điều kiện để tiếp tục thực hiện hành động gọi tool hay là không để trả về kết quả.

```
import json
from langchain_core.messages import ToolMessage
from langchain_core.runnables import RunnableConfig

tools_by_name = {tool.name: tool for tool in tools}

```

```python

# Define our tool node
def tool_node(state: AgentState):
    outputs = []
    for tool_call in state["messages"][-1].tool_calls:
        tool_result = tools_by_name[tool_call["name"]].invoke(tool_call["args"])
        outputs.append(
            ToolMessage(
                content=json.dumps(tool_result),
                name=tool_call["name"],
                tool_call_id=tool_call["id"],
            )
        )
    return {"messages": outputs}


# Define the node that calls the model
def call_model(
    state: AgentState,
    config: RunnableConfig,
):
    response = model.invoke([system_prompt] + state["messages"], config)
    # We return a list, because this will get added to the existing list
    return {"messages": [response]}


# Define the conditional edge that determines whether to continue or not
def should_continue(state: AgentState):
    messages = state["messages"]
    last_message = messages[-1]
    # If there is no function call, then we finish
    if not last_message.tool_calls:
        return "end"
    # Otherwise if there is, we continue
    else:
        return "continue"
```

4. Xây dựng đồ thị

Cuối cùng để hoàn thiện, chúng ta xây dựng đồ thị cho ReAct Agent với thông tin như sau:

- Các nodes: agent (llms) và tool
- Vị trí bắt đầu khi người dùng gửi yêu cầu: agent
- Cạnh giữa agent đến tool: để kiểm tra nếu còn tool cần thực hiện sẽ tiếp tục nếu không thì sẽ kết thúc
- Cạnh giữa tool đến agent: sẽ trả về kết quả thực hiện hành động cho agent.

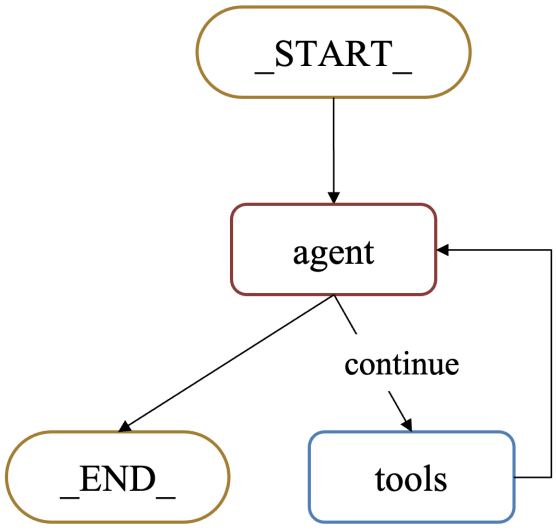
```python
from langgraph.graph import StateGraph, END

# Define a new graph
workflow = StateGraph(AgentState)

# Define the two nodes we will cycle between
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)

# Set the entrypoint as agent. This means that this node is the first one called
workflow.set_entry_point("agent")
```

```python

# We now add a conditional edge
workflow.add_conditional_edges(
    # First, we define the start node. We use `agent`.
    # This means these are the edges taken after the `agent` node is called.
    "agent",
    # Next, we pass in the function that will determine which node is called next.
    should_continue,
    {
        "continue": "tools", # If `tools`, then we call the tool node.
        "end": END, # Otherwise we finish.
    },
)

# We now add a normal edge from `tools` to `agent`.
# This means that after `tools` is called, `agent` node is called next.
workflow.add_edge("tools", "agent")

# Now we can compile and visualize our graph
graph = workflow.compile()
```

Để kiểm tra đồ thị vừa xây dựng, chúng ta thực thi lệnh sau:

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

try:
    display(Image(graph.get_graph().draw_mermaid_png()))
except Exception:
    # This requires some extra dependencies and is optional
    pass
```

Kết quả đồ thị như hình sau:



Hình 5: ReAct Graph.

5. Kiểm tra

Sau đó, chúng ta thực hiện đoạn code sau và kiểm tra kết quả.

```python
inputs = {"messages": [("user", "what is the weather in vietnam? Then Triple it "
    )]}

for state in graph.stream(inputs, stream_mode="values"):
    last_message = state["messages"][-1]
    last_message.pretty_print()
```

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

what is the weather in vietnam? Then Triple it
================================== Ai Message ==================================
Tool Calls:
  tavily_search (call_gN7sBfV4GynshBpXRt04V6xO)
 Call ID: call_gN7sBfV4GynshBpXRt04V6xO
  Args:
    query: current weather in Vietnam
    search_depth: basic
================================= Tool Message =================================
Name: tavily_search

{"query": "current weather in Vietnam", "follow_up_questions": null, "answer": null, "images
================================== Ai Message ==================================
Tool Calls:
  triple (call_NiT3Cptpz9kI2VBZ7putuI58)
 Call ID: call_NiT3Cptpz9kI2VBZ7putuI58
  Args:
    num: 28
================================= Tool Message =================================
Name: triple

84.0
================================== Ai Message ==================================
...
Action Input: 28
Observation: The result of tripling is 84.0.

Final Answer: The current temperature in Vietnam is 28.2°C, and when tripled, it is 84.0°C.
```

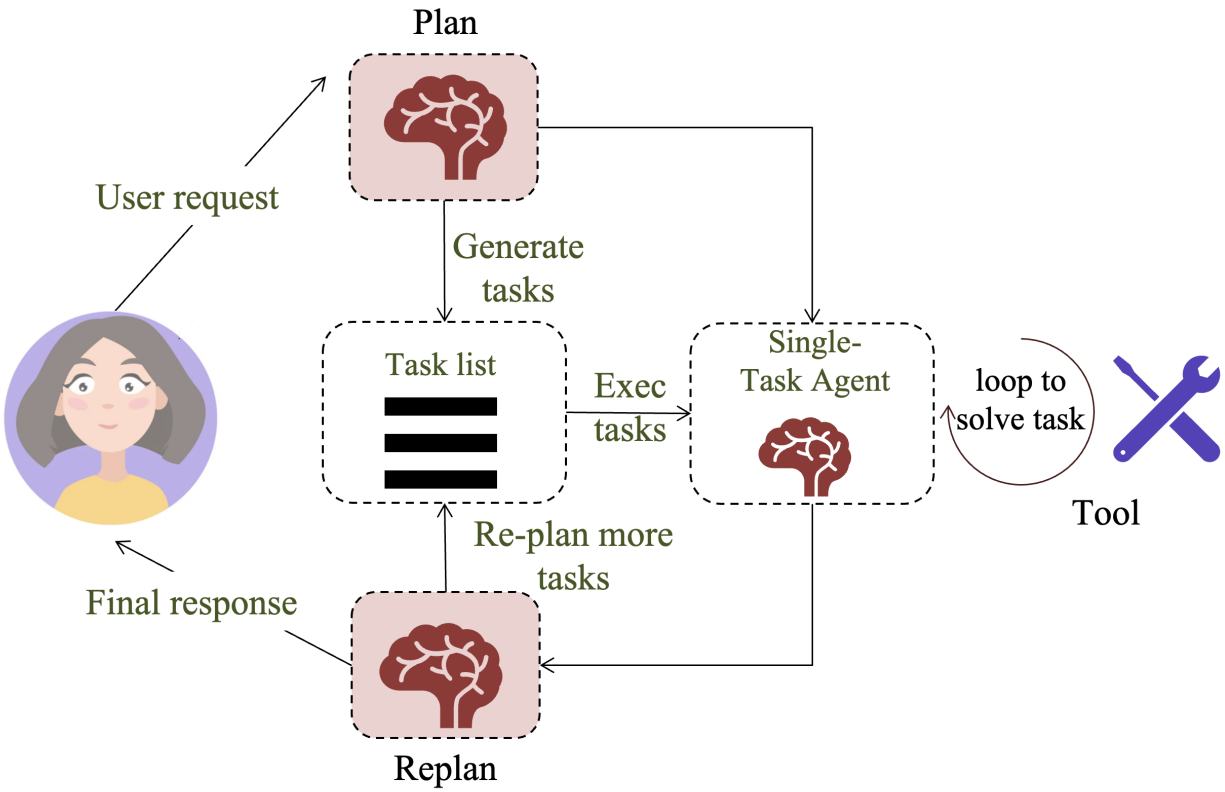
Hình 6: Test ReAct Agent.

Phần 3. Plan-and-Execute Agent

Plan-and-Execute (Lập kế hoạch và Thực thi) là một phương pháp trong đó agent đầu tiên phát triển một kế hoạch chi tiết cho toàn bộ nhiệm vụ, sau đó thực hiện kế hoạch đó theo từng bước.

Phương pháp Plan-and-Execute chia quá trình giải quyết vấn đề thành hai giai đoạn riêng biệt:

- Planning (Lập kế hoạch): Mô hình phát triển một kế hoạch tổng thể, chi tiết về các bước cần thực hiện để đạt được mục tiêu.
- Execution (Thực thi): Mô hình thực hiện từng bước một trong kế hoạch đã đề ra, theo đúng thứ tự đã định trước.



Hình 7: Plan-and-Execute Agent.

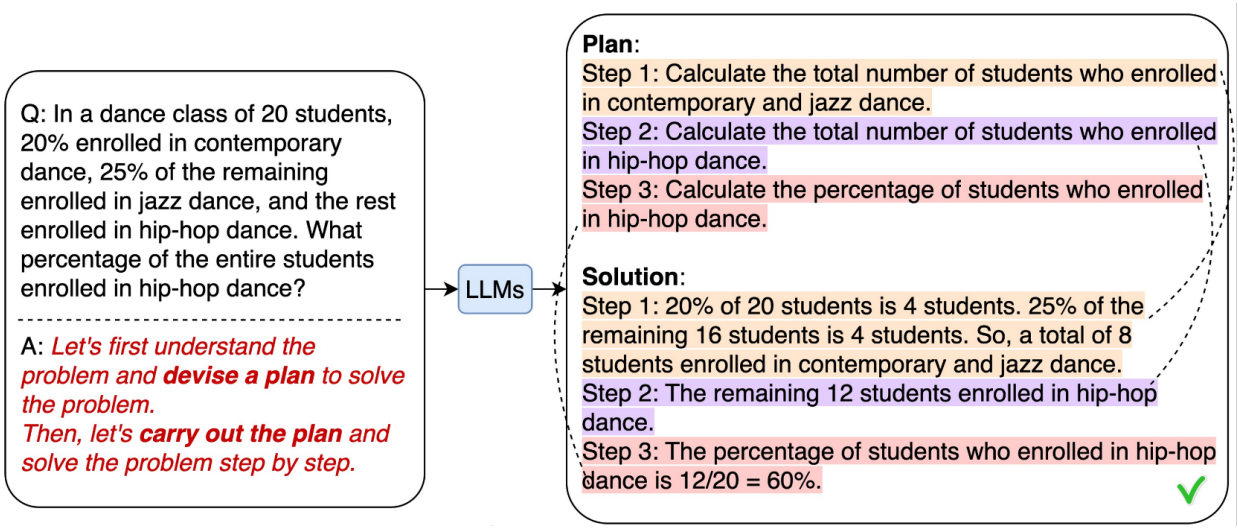
Giai đoạn lập kế hoạch:

- Phân tích vấn đề: Mô hình phân tích kỹ lưỡng vấn đề, xác định mục tiêu và các điều kiện ràng buộc.
- Phân chia thành các bước nhỏ: Vấn đề được chia thành các bước nhỏ, có thể quản lý được.
- Sắp xếp thứ tự các bước: Các bước được sắp xếp theo một trình tự logic, xác định rõ điều kiện tiên quyết.
- Xác định tài nguyên cần thiết: Mô hình xác định các công cụ, thông tin hoặc tài nguyên cần thiết cho mỗi bước.

Giai đoạn thực thi::

- Thực hiện từng bước: Mô hình thực hiện các bước theo kế hoạch đã đề ra.
- Thu thập kết quả: Sau mỗi bước, kết quả được ghi lại và đánh giá.
- Kiểm tra tiến độ: Tiến độ được theo dõi so với kế hoạch ban đầu.
- Hoàn thành mục tiêu: Quá trình tiếp tục cho đến khi tất cả các bước đã hoàn thành.

Ví dụ minh hoạ cho agent này như sau:



Hình 8: Plan-and-Execute Agent Example.

Trong phần này, chúng ta sẽ sử dụng thư viện LangGraph để xây dựng ReAct Agent

1. Cài đặt thư viện

Cài đặt một số thư viện và chuẩn bị các Key cho quá trình sử dụng LLMs cũng như lưu trữ thông tin thông qua công cụ LangSmith. Sau đó xây dựng lớp AgentState để thực thi gồm các thuộc tính như sau:

- input: Dữ liệu đầu vào
- plan: danh sách các bước cần thực hiện
- past_steps: danh sách các bước đã thực hiện
- response: phản hồi

```
# Install libs
!pip install -U langchain==0.3.24 langchain-openai==0.3.14 langgraph==0.3.33
    langchain-tavily==0.1.6

import os
os.environ["LANGCHAIN_API_KEY"] = ### Your-key
os.environ["LANGCHAIN_PROJECT"] = "Reasoning-Agent"
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["TAVILY_API_KEY"] = ### Your-key
os.environ["OPENAI_API_KEY"] = ### Your-key

import operator
```

```python
from typing import Annotated, List, Tuple
from typing_extensions import TypedDict


class PlanExecute(TypedDict):
    input: str
    plan: List[str]
    past_steps: Annotated[List[Tuple], operator.add]
    response: str
```

2. Xây dựng các công cụ

Trong phần này, chúng ta xây dựng 2 công cụ để thực hiện hành động là tìm kiếm. Bên cạnh đó, chúng ta sử dụng mô hình chatGPT để thực hiện.

```python
from langchain_tavily import TavilySearch

tools = [TavilySearch(max_results=1)]
```

3. Xây dựng mô hình lên kế hoạch

Mô hình lên kế hoạch là mô hình ngôn ngữ lớn, ở đây chúng ta sử dụng GPT và prompt để yêu cầu mô hình đề xuất các bước cần thực thi.

```python
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field

class Plan(BaseModel):
    """Plan to follow in future"""

    steps: List[str] = Field(
        description="different steps to follow, should be in sorted order"
    )

planner_prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """For the given objective, come up with a simple step by step plan. \
This plan should involve individual tasks, that if executed correctly will yield the correct answer. Do not add any superfluous steps. \
The result of the final step should be the final answer. Make sure that each step has all the information needed - do not skip steps.""",
        ),
        ("placeholder", "{messages}"),
    ]
)
planner = planner_prompt | ChatOpenAI(
    model="gpt-4o", temperature=0
).with_structured_output(Plan)

planner.invoke(
    {
        "messages": [
            ("user", "What is the weather in Vietnam?")
        ]
    }
)
```

4. Xây dựng mô hình thực thi

Mô hình thực thi cũng sẽ là mô hình GPT, trong đó sẽ có các thông tin đẩy vào prompt là plan và past_steps để đánh các bước thực thi và kết quả.

```
from typing import Union

class Response(BaseModel):
    """Response to user."""

    response: str

class Act(BaseModel):
    """Action to perform."""

    action: Union[Response, Plan] = Field(
        description="Action to perform. If you want to respond to user, use Response. "
        "If you need to further use tools to get the answer, use Plan."
    )

replanner_prompt = ChatPromptTemplate.from_template(
    """For the given objective, come up with a simple step by step plan. \
This plan should involve individual tasks, that if executed correctly will yield the correct answer. Do not add any superfluous steps. \
The result of the final step should be the final answer. Make sure that each step has all the information needed - do not skip steps.

Your objective was this:
{input}

Your original plan was this:
{plan}

You have currently done the follow steps:
{past_steps}

Update your plan accordingly. If no more steps are needed and you can return to the user, then respond with that. Otherwise, fill out the plan. Only add steps to the plan that still NEED to be done. Do not return previously done steps as part of the plan."""
)


replanner = replanner_prompt | ChatOpenAI(
    model="gpt-4o", temperature=0
).with_structured_output(Act)
```

5. Xây dựng chu trình thực hiện

Trong phần này chúng ta định nghĩa lời gọi hàm các các công cụ để tạo sự liên kết giữa các thành phần. Bên cạnh đó định nghĩa hàm should_end để xem xét số lượng các bước thực hiện đã hết chưa. Nếu chưa tiếp tục bước tiếp theo nếu đã hết trả về kết quả.

```
from typing import Literal
from langgraph.graph import END

async def execute_step(state: PlanExecute):
    plan = state["plan"]
    plan_str = "\n".join(f"{i+1}. {step}" for i, step in enumerate(plan))
    task = plan[0]
```

```
    task_formatted = f"""For the following plan:
        {plan_str}\n\nYou are tasked with executing step {1}, {task}."""
    agent_response = await agent_executor.ainvoke(
        {"messages": [("user", task_formatted)]}
    )
    return {
        "past_steps": [(task, agent_response["messages"][-1].content)],
    }


async def plan_step(state: PlanExecute):
    plan = await planner.ainvoke({"messages": [("user", state["input"])]})
    return {"plan": plan.steps}


async def replan_step(state: PlanExecute):
    output = await replanner.ainvoke(state)
    if isinstance(output.action, Response):
        return {"response": output.action.response}
    else:
        return {"plan": output.action.steps}


def should_end(state: PlanExecute):
    if "response" in state and state["response"]:
        return END
    else:
        return "agent"
```

6. Xây dựng đồ thị

Xây dựng đồ thị với các thông tin như sau:

- Các nodes: agent, planner và replan
- Vị trí bắt đầu: planner
- Các cạnh tương ứng giữa planner và agent, agent và replan. Trong đó cạnh có điều kiện là từ replan đến agent để kết thúc chu trình thực hiện.

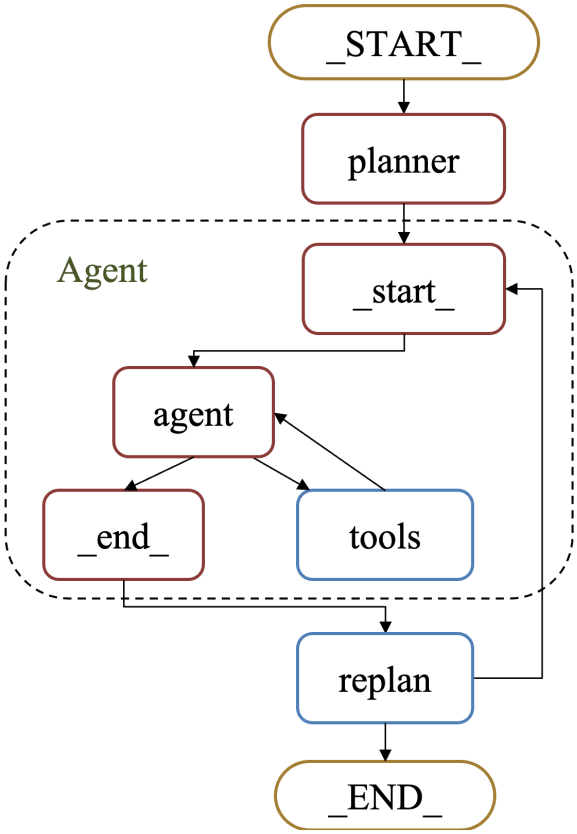
```
from langgraph.graph import StateGraph, START

workflow = StateGraph(PlanExecute)

# Add the plan node
workflow.add_node("planner", plan_step)

# Add the execution step
workflow.add_node("agent", execute_step)

# Add a replan node
workflow.add_node("replan", replan_step)

workflow.add_edge(START, "planner")

# From plan we go to agent
workflow.add_edge("planner", "agent")

# From agent, we replan
workflow.add_edge("agent", "replan")

```

```python
workflow.add_conditional_edges(
    "replan",
    # Next, we pass in the function that will determine which node is called next.
    should_end,
    ["agent", END],
)

# Finally, we compile it!
# This compiles it into a LangChain Runnable,
# meaning you can use it as you would any other runnable
app = workflow.compile()
```

Sau khi xây dựng đồ thị, để kiểm tra và hiển thị đồ thị chúng ta chạy lệnh sau:

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

display(Image(app.get_graph(xray=True).draw_mermaid_png()))
```

Kết quả thu được:



Hình 9: Plan-and-Execute Graph.

7. Thử nghiệm mô hình

Để kiểm tra khả năng suy luận của agent, chúng ta chạy đoạn code sau:

```python
config = {"recursion_limit": 50}

inputs = {"input": "Solve for all real solutions x to the equation: sqrt(x -2) = 3"}
```

```
4
5 async for event in app.astream(inputs, config=config):
6     for k, v in event.items():
7         if k != "__end__":
8             print(v)
```

Kết quả như hình sau:

Input

Solve for all real solutions x to the equation: sqrt(x -2) = 3

Response

All steps have been completed successfully, and the solution \(x = 11 \) has been verified as correct. No further steps are needed. The final answer is:

x = 11

Past Steps

- 0
  - 0 Start with the equation: sqrt(x - 2) = 3.
  - 1 Step 1 has already been specified in the plan. The equation give...
- 1
  - 0 Square both sides of the equation to eliminate the square root: ...
  - 1 To execute step 1, we start with the equation: \[\sqrt{x - 2} ...
- 2
  - 0 Add 2 to both sides to solve for x: x - 2 + 2 = 9 + 2.
  - 1 In step 1, we start with the equation \(x - 2 = 9 \). To solve...
- 3
  - 0 Verify the solution by substituting x = 11 back into the origina...
  - 1 To verify the solution by substituting \(x = 11 \) back into th...

Plan

- 0 Verify the solution by substituting x = 11 back into the origina...
- 1 Simplify the left side: sqrt(9) = 3.
- 2 Since both sides are equal, x = 11 is a valid solution.

Hình 10: Plan-and-Execute Result.

Phần 4. Câu hỏi trắc nghiệm

Câu hỏi 1 Reasoning Agents được định nghĩa như thế nào?
a) Hệ thống AI chỉ thực hiện các hành động theo lệnh
b) Hệ thống AI với khả năng suy luận có cấu trúc để giải quyết vấn đề phức tạp
c) Hệ thống chỉ gọi đến các công cụ bên ngoài khi cần thiết
d) Hệ thống đơn thuần phân tích dữ liệu lớn mà không lập kế hoạch

Câu hỏi 2 Phương pháp ReAct trong Reasoning Agents là viết tắt của:
a) Reactive Action
b) Reasoning Activity
c) Reasoning + Acting
d) Refined Action

Câu hỏi 3 Trình tự đúng của các bước trong phương pháp ReAct là gì?
a) Hành động → Suy luận → Quan sát → Tiếp tục chu trình
b) Suy luận → Hành động → Quan sát → Tiếp tục chu trình
c) Quan sát → Suy luận → Hành động → Tiếp tục chu trình
d) Suy luận → Quan sát → Hành động → Tiếp tục chu trình

Câu hỏi 4 Sau khi ReAct-agent gửi một Action để gọi công cụ bên ngoài (ví dụ: Wikipedia-API), bước kế tiếp điển hình là gì?
a) Kết thúc phiên làm việc
b) Thêm kết quả công cụ vào bối cảnh rồi tiếp tục Reasoning
c) Huấn luyện lại trọng số mô hình
d) Bỏ qua kết quả công cụ và suy luận tiếp

Câu hỏi 5 Mục đích chính của việc xen kẽ Reasoning và Acting trong ReAct là gì?
a) Tăng kích thước prompt để khai thác tối đa context của LLM
b) Tách logic và dữ liệu để thuận tiện deploy micro-service
c) Hạn chế "hallucination" bằng cách lấy dữ liệu thực qua mỗi bước hành động
d) Giảm lượng token sinh ra nhằm tiết kiệm chi phí

Câu hỏi 6 Đâu là ưu điểm chính của phương pháp Plan-and-Execute so với ReAct?
a) Khả năng thích ứng cao hơn với các tình huống không lường trước
b) Xử lý vấn đề nhanh hơn trong mọi trường hợp
c) Có tầm nhìn tổng thể về toàn bộ nhiệm vụ từ đầu
d) Tiêu tốn ít tài nguyên xử lý hơn

Câu hỏi 7 Đâu là khác biệt quan trọng nhất về cách tiếp cận giữa ReAct và Plan-and-Execute?
a) ReAct lặp đi lặp lại và linh hoạt, Plan-and-Execute tuần tự và có cấu trúc
b) ReAct dùng cho nhiệm vụ đơn giản, Plan-and-Execute cho nhiệm vụ phức tạp
c) ReAct chỉ phân tích, Plan-and-Execute chỉ hành động
d) ReAct tốn nhiều tài nguyên hơn Plan-and-Execute

Câu hỏi 8 Công cụ nào sau đây được sử dụng để tìm kiếm thông tin?
a) Tavily
b) Wikipedia

c) LangGraph
d) LangSmith

Câu hỏi 9 Trong LangGraph, "edges" (cạnh) có chức năng gì?
a) Kết nối node với dữ liệu đầu vào
b) Tạo điều kiện để chuyển trạng thái dựa vào kết quả node trước đó
c) Lưu trữ biến tạm
d) Tăng tốc độ xử lý mô hình LLM

Câu hỏi 10 LangGraph hỗ trợ loại agent nào sau đây?
a) ReAct
b) Plan-and-Execute
c) Tool-augmented reasoning agents
d) Tất cả các loại trên

Phần 4. Phụ lục

1. **Code**: Thư mục chứa mã nguồn notebook CODE
2. **Code**: Tài liệu code trên folder CODE
3. Tài liệu tham khảo:
   - ReAct: Synergizing Reasoning and Acting in Language Models
   - Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models
   - LangGraph
4. **Rubric**:

Phần	Kiến Thức	Đánh Giá
1	- Hiểu rõ đặc trưng reasoning agents - ReAct Agent và các bước thực hiện. Minh hoạ về khả năng xử lý các tác vụ phức tạp.	- Thực thi ReAct sử dụng langgraph
2.	- Tối ưu agent theo hướng plan-and-execute agent - Các thành phần quan trọng và xây dựng các agent trong plan-and-execute	- Thực thi plan-and-execute agent sử dụng langgraph. Kết hợp ReAct vào tối ưu hệ thống reasoning agents.

- Hết -