

Module 10

Reasoning Agents

ReAct Agent & Plan-and-Execute Agent

[Code - Data](#)

[LLM-Agents Space](#)

Nguyen Quoc Thai
MSc in Computer Science
STA Ho Quang Hien

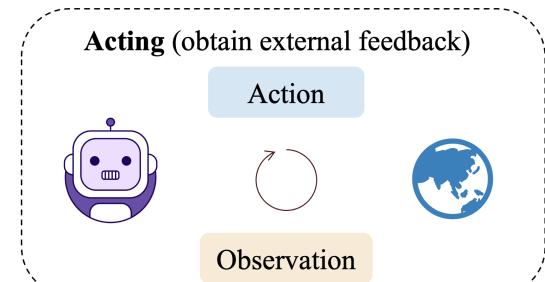
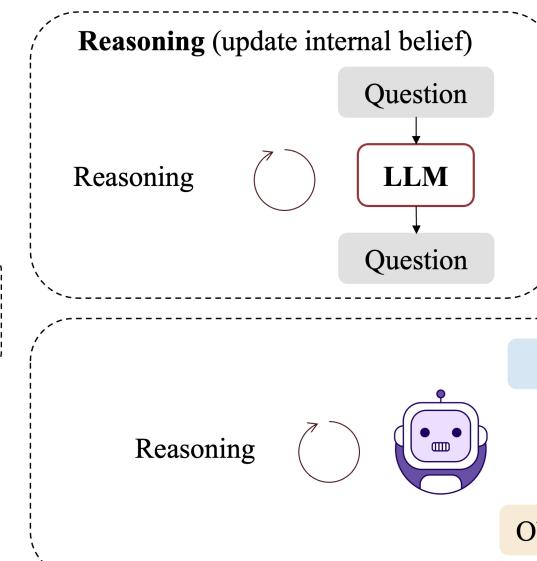
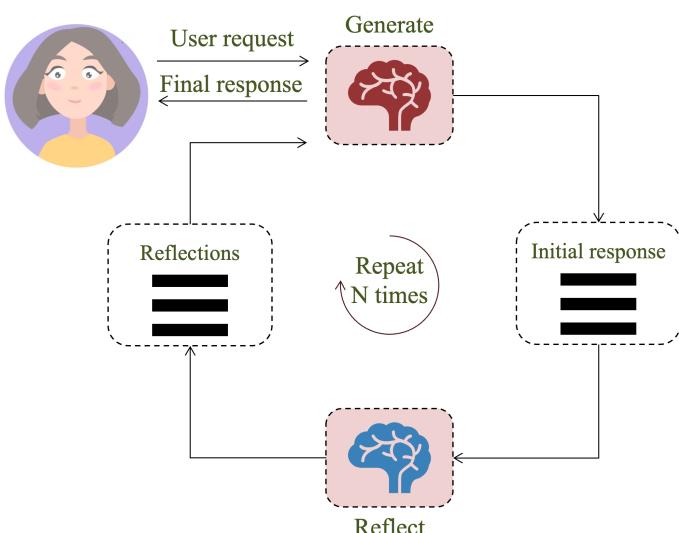
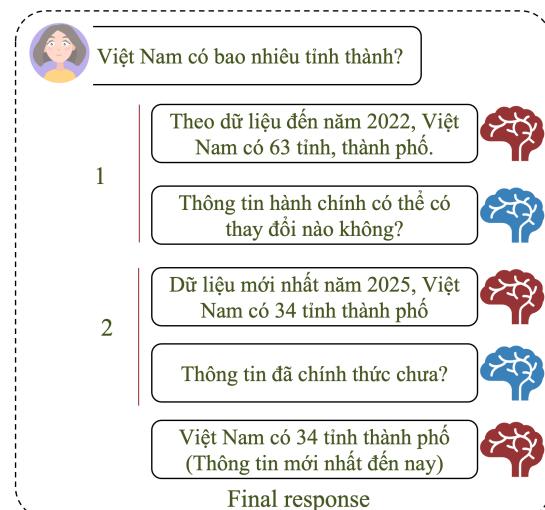
Objectives

LLM Agent

- ❖ AI Agent
- ❖ LLM Agent
- ❖ Reflection Agent

Reasoning Agents

- ❖ Reasoning Agent
- ❖ ReAct Agent
- ❖ Plan-and-Execute Agent

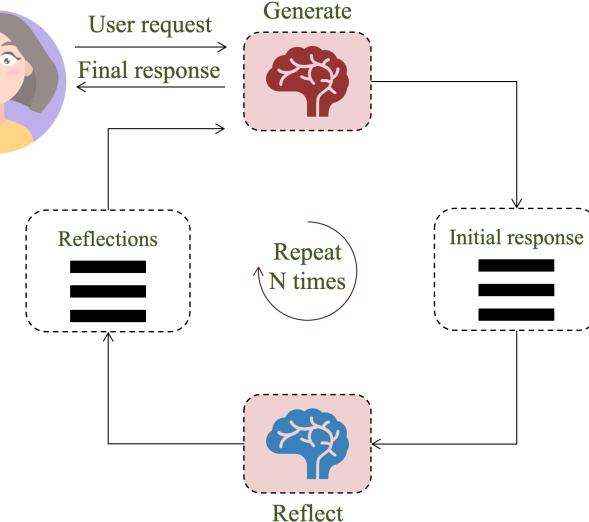
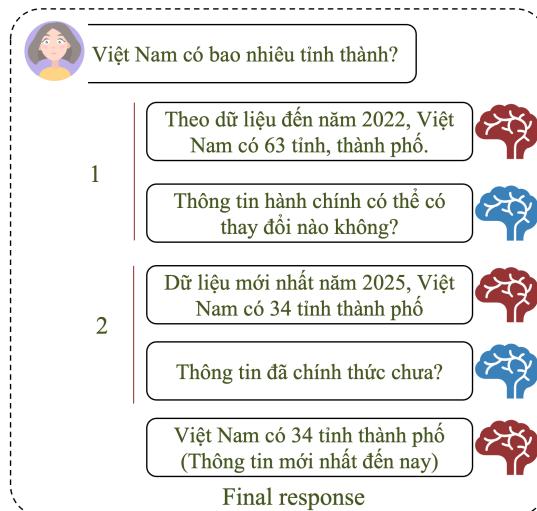


ReAct: a new paradigm of agents that reason and act
 ➤ Synergy of reasoning and acting
 ➤ Simple and intuitive to use
 ➤ General across domains

Outline

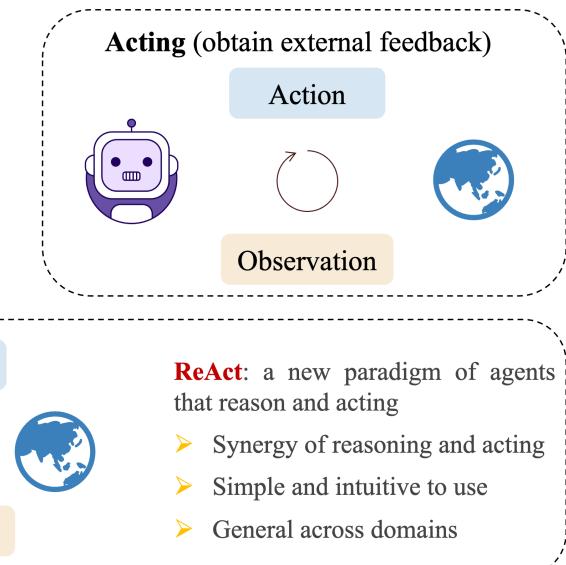
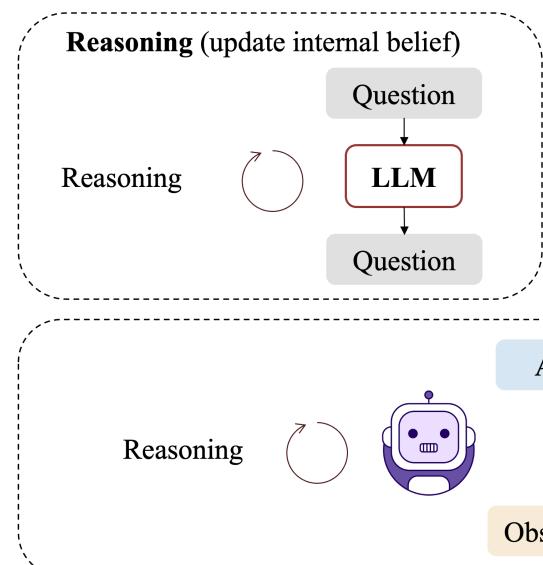
SECTION 1

LLM Agent



SECTION 2

Reasoning Agents



LLM Agent



Overview of AI Agent Series

Day 01: AI Agent Libraries

- High-level understanding of AI Agent.
- Investigate how to build AI Agent applications through popular libraries.

Day 02: LLMs Tools Calling

- First low-level understanding of AI Agent.
- Investigate the simplest and early form of AI Agent: Make LLMs using tools to solve tasks.

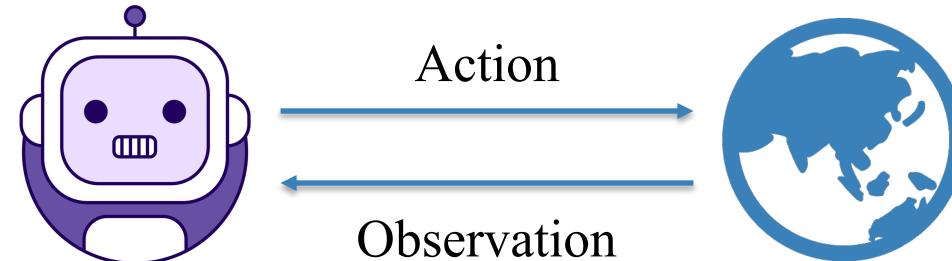
Day 03: ReAct AI Agent

- Second low-level understanding of AI Agent.
- Investigate the general baseline of current AI Agent: Thought-Action-Observe through ReAct AI Agent.

LLM Agent



What is “Agent”?



- An “intelligent” system that interacts with some “environment”
 - Physical environments: robot, autonomous car,...
 - Digital environments: Siri, AlphaGo,...
 - Humans as environments: chatbot
- Define “agent” by defining “intelligent” and “environment”
 - It changes over time!
 - Exercise question: how would you define “intelligent”?

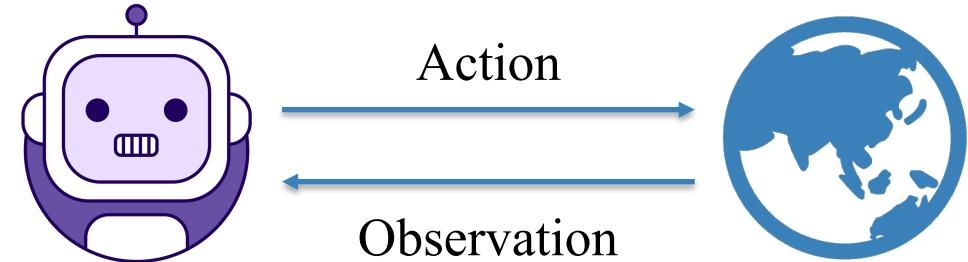
LLM Agent



What is “LLM Agent”?

➤ Level 1: Text agent

- Uses text action and observation
- Example: ELIZA, LSTM-DQN



```
Welcome to
      EEEEEE  LL      IIII    ZZZZZZ  AAAAAA
      EE     LL      II      ZZ  AA  AA
      EEEEEE  LL      II      ZZZ  AAAAAAAA
      EE     LL      II      ZZ  AA  AA
      EEEEEE  LLLLLL  IIII  ZZZZZZ  AA  AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU: Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU: They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU: Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU: He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU: It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

LLM Agent



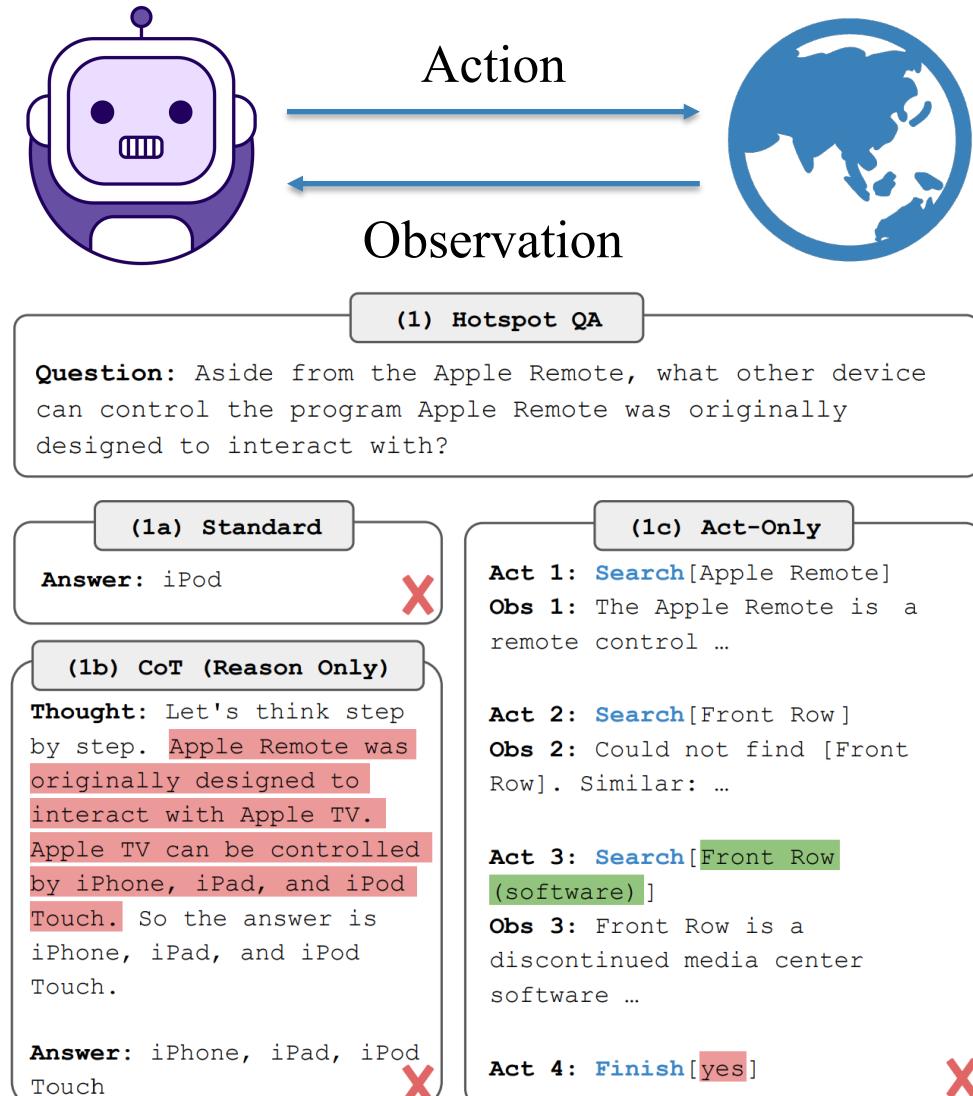
What is “LLM Agent”?

➤ Level 1: Text agent

- Uses text action and observation
- Example: ELIZA, LSTM-DQN

➤ Level 2: LLM agent

- Uses LLMs to act
- Example: SayCan, Language Planner



LLM Agent



What is “LLM Agent”?

➤ Level 1: Text agent

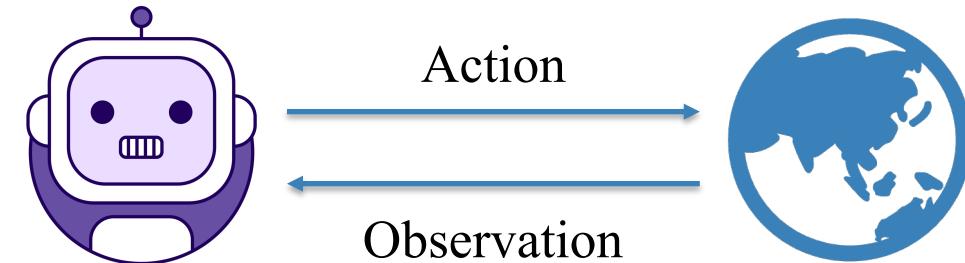
- Uses text action and observation
- Example: ELIZA, LSTM-DQN

➤ Level 2: LLM agent

- Uses LLMs to act
- Example: SayCan, Language Planner

➤ Level 3: Reasoning agent

- Uses LLMs to reason to act
- Example: ReAct, Plain-and-Execute Agent, AutoGPT, ...



(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.

Act 1: Search[Apple Remote]

Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple ... originally designed to control the Front Row media center program ...

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.

Act 2: Search[Front Row]

Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports', 'Front Row (software)', ...]

Thought 3: Front Row is not found. I need to search Front Row (software) .

Act 3: Search[Front Row (software)]

Obs 3: Front Row is a discontinued media center software ...

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.

Act 4: Finish[keyboard function keys]

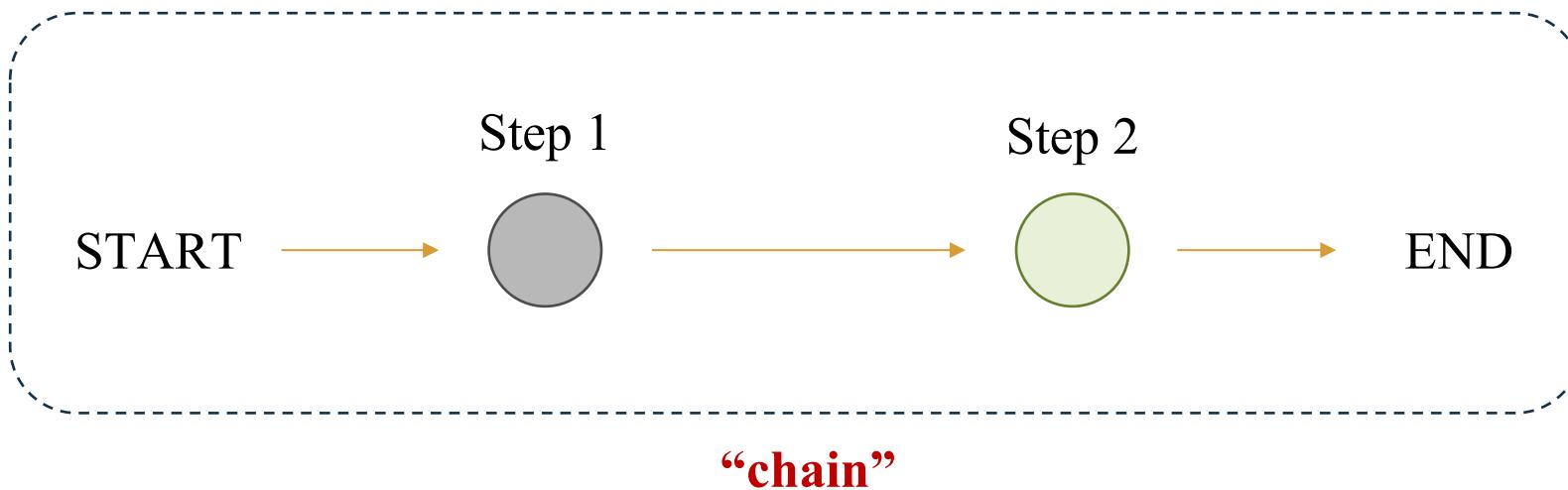


LLM Agent



Control Flow Concept

- Good AI Applications follow structured control flow, which can be visualized as chains of operations
- This enables consistency and robustness in execution

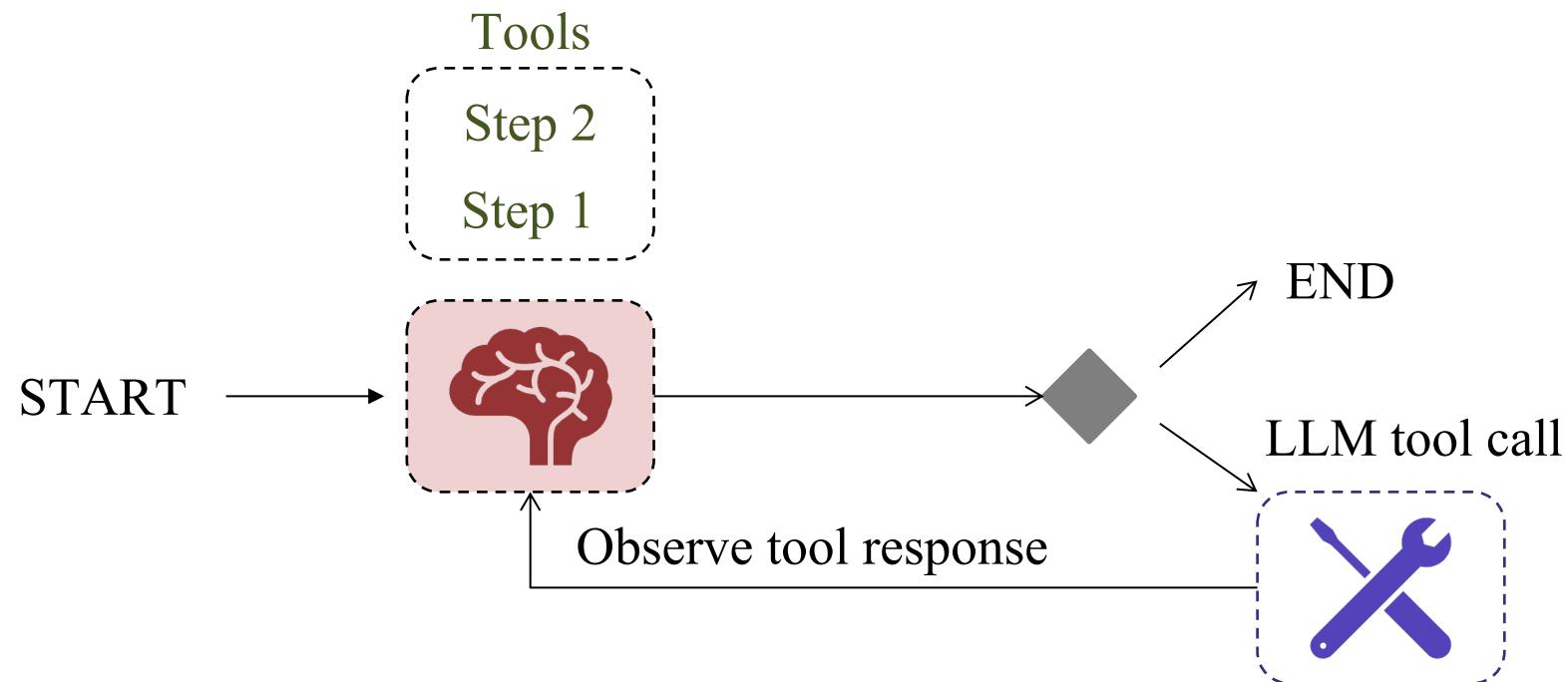


LLM Agent



Agent-Defined Flow (Tool Calling)

- Autonomous agents can define dynamic control flow using tools to execute steps, but this introduces challenges like ambiguity, non-determinism, and potential misuse.

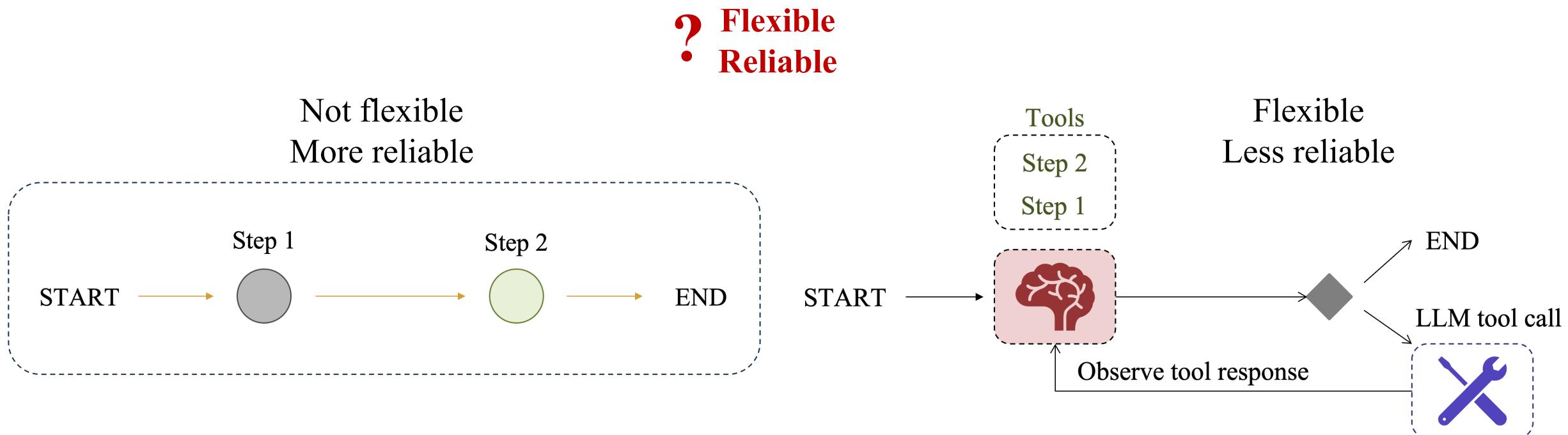


LLM Agent



Reliability – Flexible Issues

- Autonomous agents often face reliability problems due to unpredictable behaviors and tool mismanagement, as highlighted in user feedback forums.



LLM Agent



Introduction to LangGraph

```
graph TD; start([_start_]) --> assistant[assistant]; tools[tools] --> start; tools -.-> assistant; assistant -.-> end([_end_]);
```

Input

Messages Required

+ Message

Submit

! _start_
Hi

A assistant
Hello! How can I assist you today as the customer support assistant for Solar Panels Belgium?

! _start_
What are solar panels?

A assistant
Solar panels are devices that convert sunlight directly into electricity. They are made up of many individual solar cells that use the photovoltaic effect to generate an electric current when exposed to sunlight.

Some key things to know about solar panels:

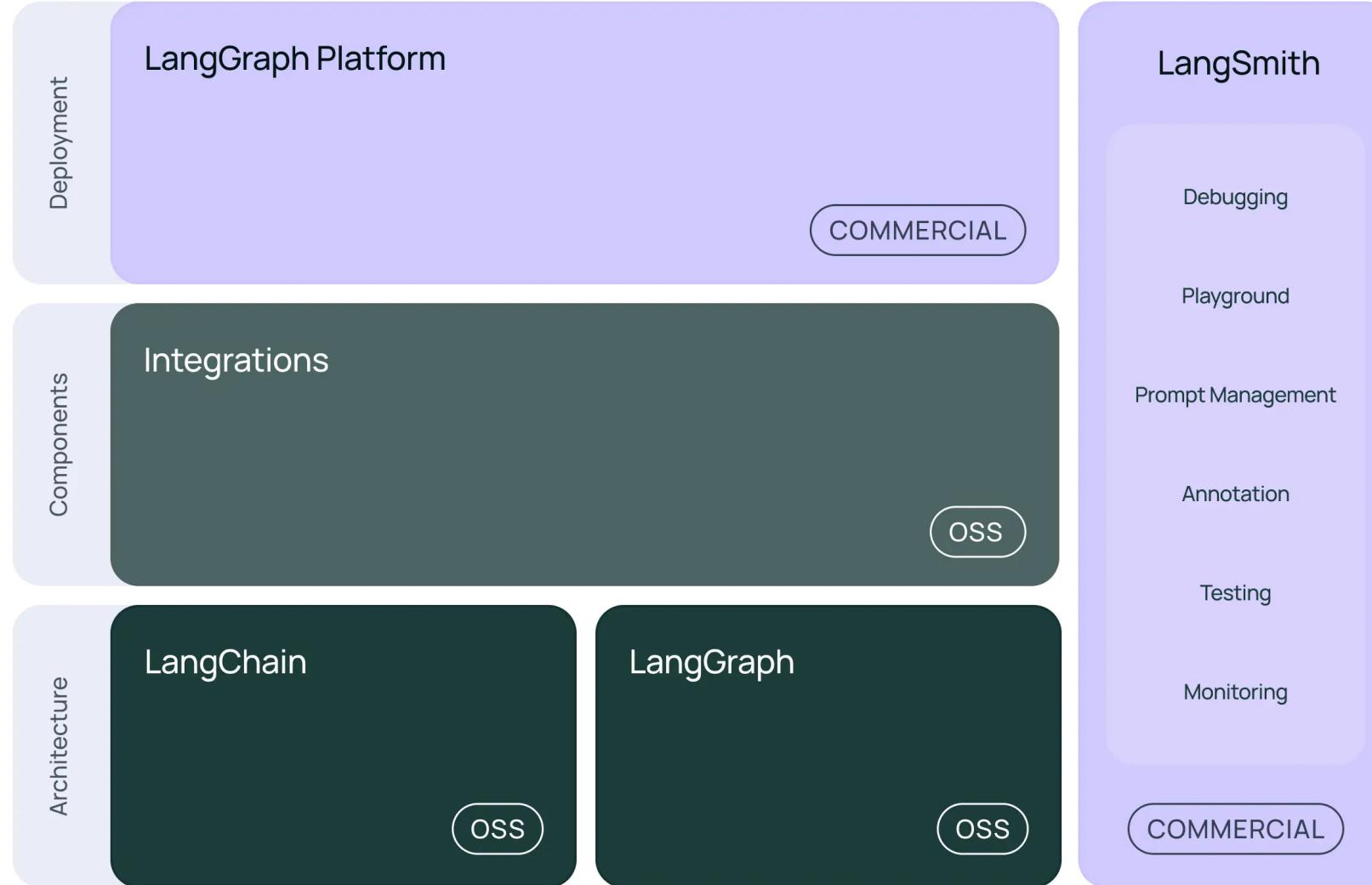
- They are made of semiconductor materials like silicon that can absorb photons from sunlight and dislodge electrons to create an electrical current.
- Solar panels produce direct current (DC) electricity which can be converted to alternating current (AC) to power homes and feed into the utility grid.
- They require virtually no fuel and have very low operating costs once installed, providing a renewable source of electricity.

OUTPUT >

LLM Agent



Introduction to LangGraph

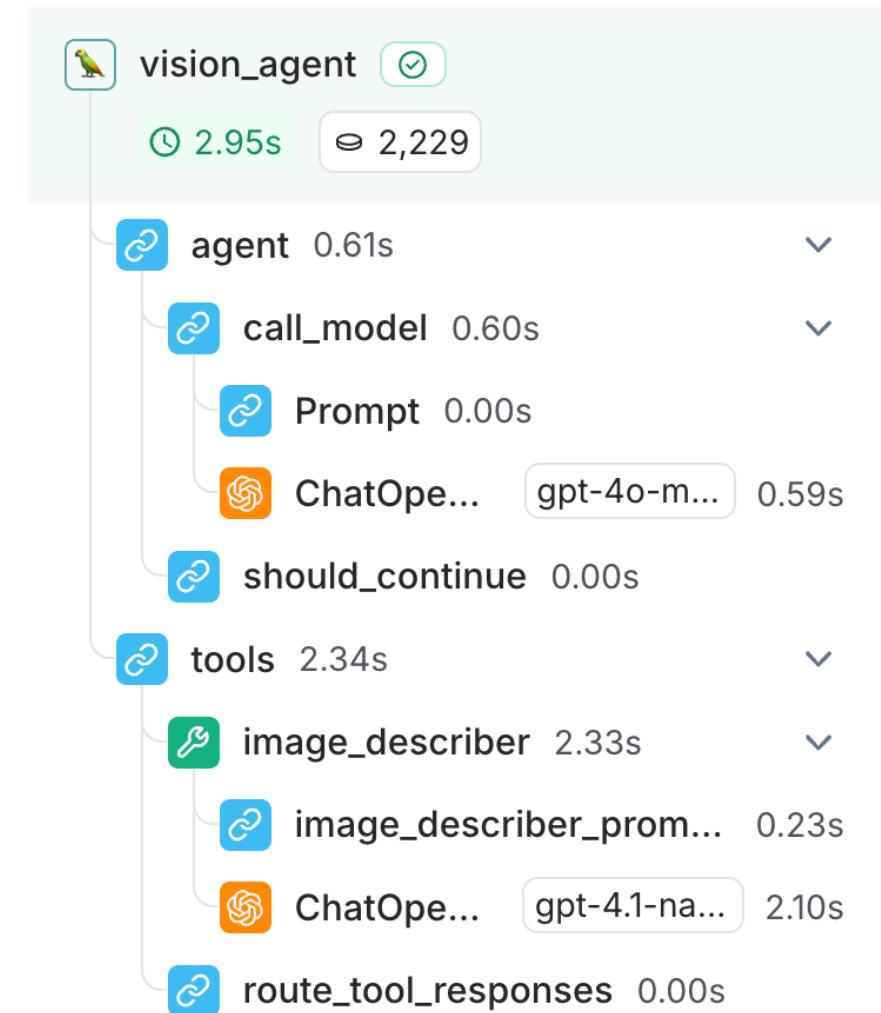


LLM Agent



Introduction to LangGraph

- **LangGraph applications balance agent control with agency.**
- Its core pillars support:
 - Controllability: to define both explicit and implicit workflows
 - Persistence: for thread-specific and cross-thread “Memory”
 - Interaction: between agents, humans and external systems
 - Streaming: to expose any event (or token) as it occurs
- Python & Jupyter notebooks
- Traceability and Prompt Management (LangSmith - Langfuse)



LLM Agent



Key Components of LangGraph

- **StateGraph**
- A class that represents the graph
- LangGraph maintains a persistent state across different nodes, which means your application can pause and resume tasks without losing context.

```
from langgraph.graph import StateGraph
from typing import TypedDict, List, Annotated
import Operator

class State(TypedDict):
    input: str
    all_actions: Annotated[List[str], operator.add]

graph = StateGraph(State)
```

LLM Agent



Key Components of LangGraph

- **Nodes:** represent the individual components of your workflow, such as LLMs, agents, or specific functions. Each node performs a distinct part of the overall task.

```
graph.add_node("model", model)
graph.add_node("tools", tool_executor)
```

- There is also a special <END> node that is used to represent the end of the graph.

```
from langgraph.graph import END
```

LLM Agent



Key Components of LangGraph

- **Edges:** define the connections between nodes, determining the flow of data and control.
- Normal Edges
- Conditional Edges

```
graph.add_edge("tools", "model")
```

```
graph.add_conditional_edge(  
    "model",           ←———— The upstream node  
    should_continue,   ←———— A function  
    {                 ←———— A mapping  
        "end": END,  
        "continue": "tools"  
    }  
)
```

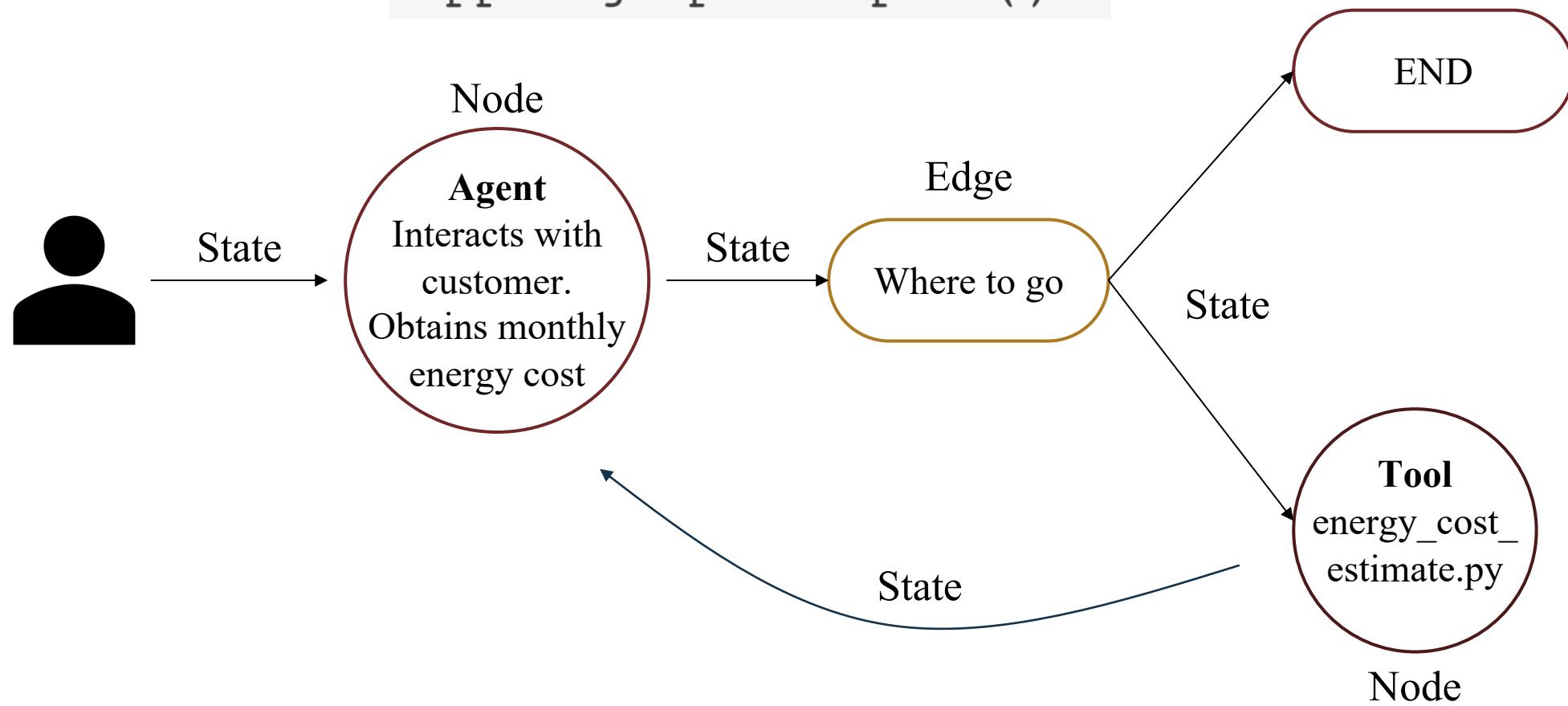
LLM Agent



Key Components of LangGraph

- **Compile:** This runnable exposes all the same method as LangChain runnables (invoke, stream, astream_log, etc)

```
app = graph.compile()
```



LLM Agent



Reflection Agent

- Reflection is a prompting strategy used to improve the quality and success rate of agents and similar AI systems.
- It involves prompting an LLM to reflect on and critique its past actions, sometimes incorporating additional external information such as tool observations.

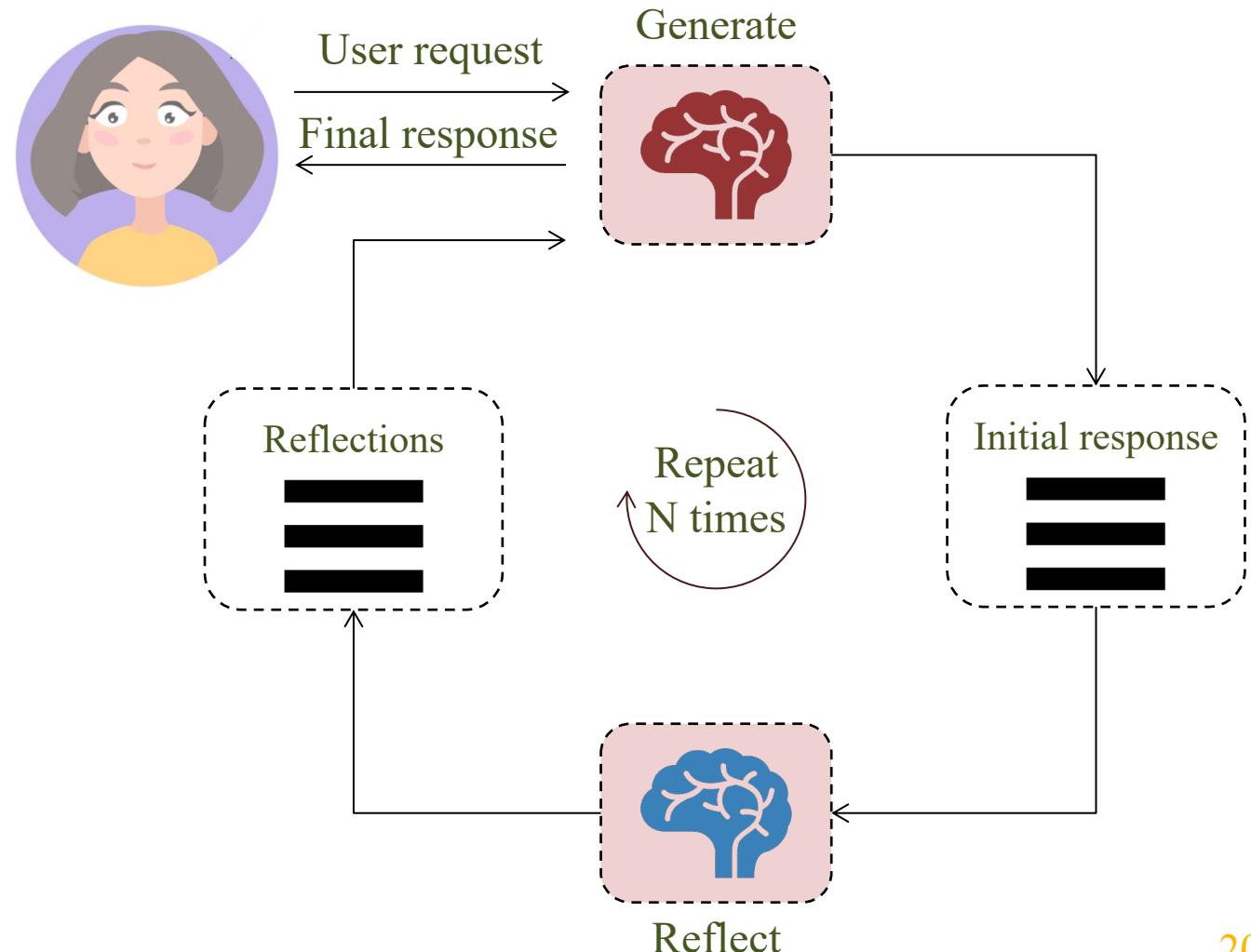


LLM Agent



Reflection Agent

- Compose LLM calls:
 - A generator
 - A reflector
- The generator tries to respond directly to the user's requests.
- The reflector is prompted to role play as a teacher and offer constructive criticism for the initial response.

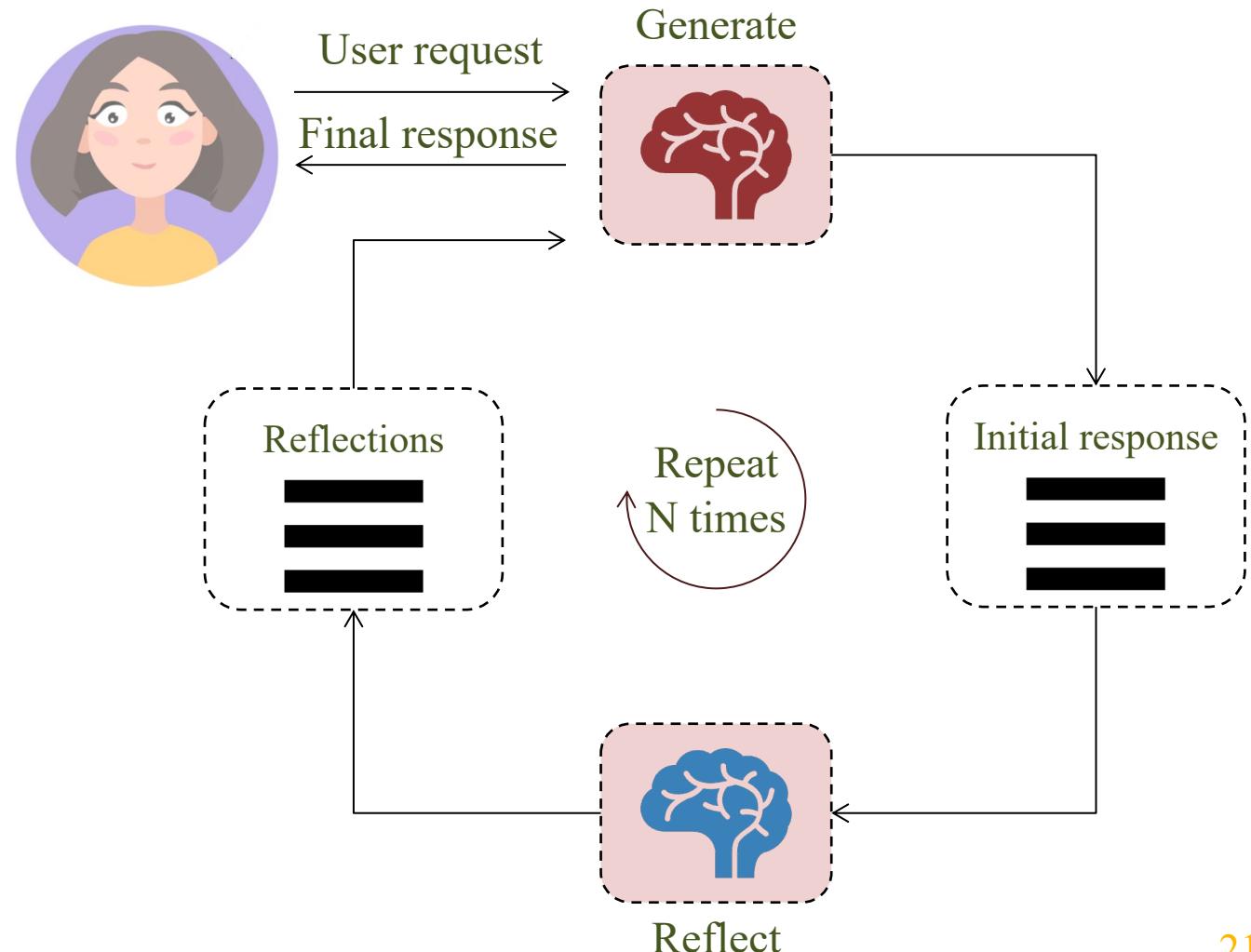


LLM Agent

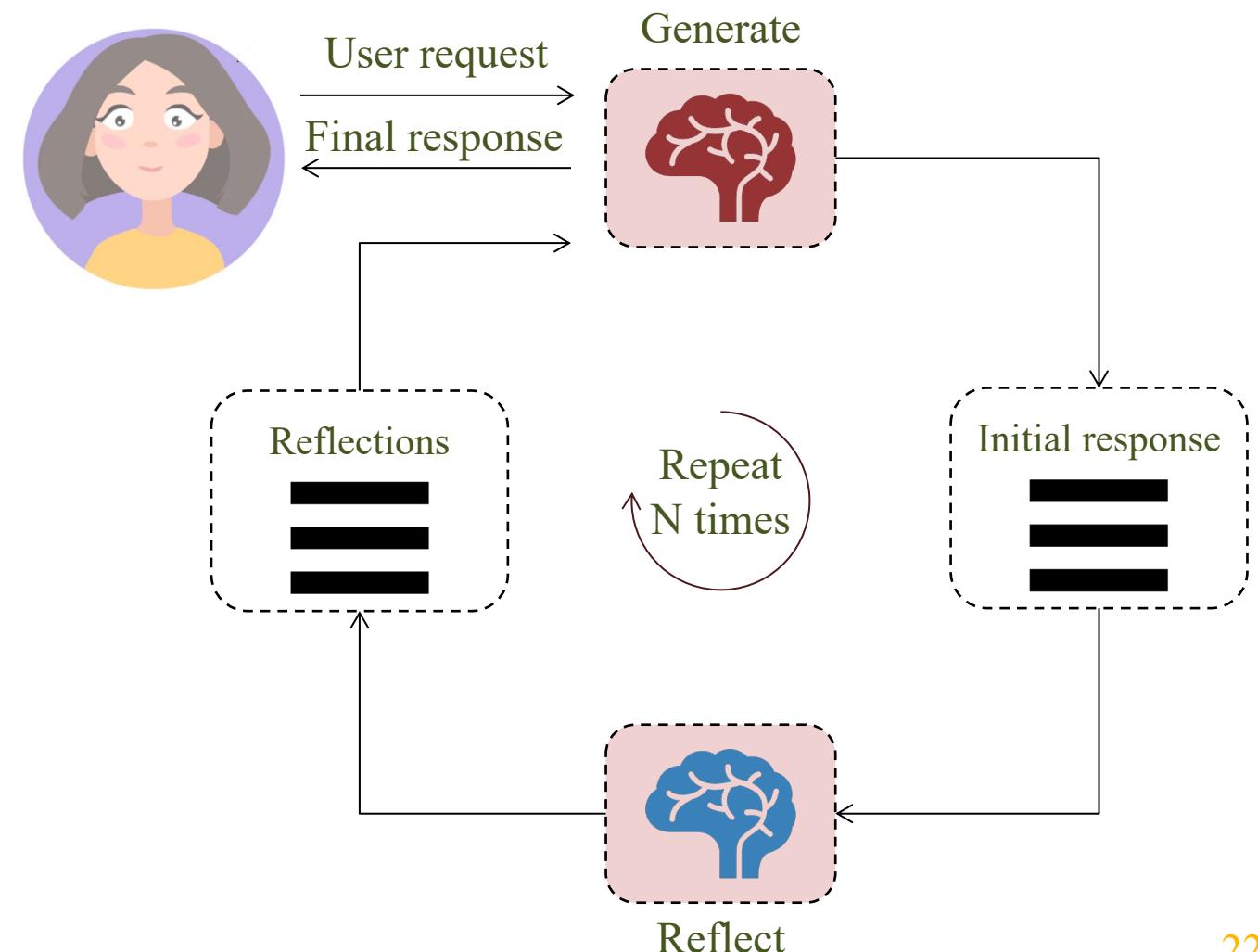
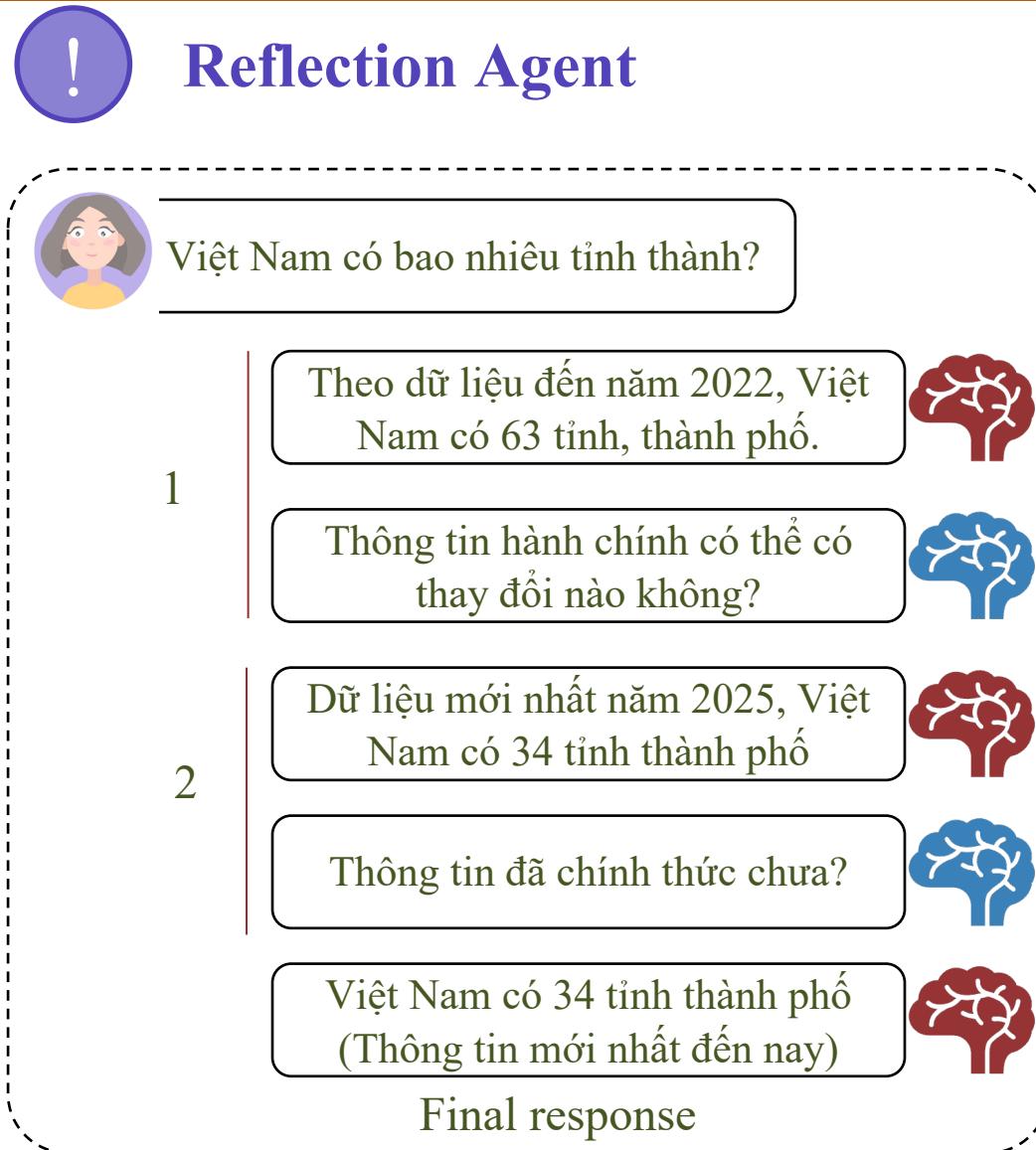


Reflection Agent

- Reflections:
 - Critique
 - Merits
 - Recs
- The loop proceeds a fixed number of times, and the final generated output is returned.



LLM Agent



LLM Agent



Reflection Agent

Generate



```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import AIMessage, HumanMessage
from langchain_openai import ChatOpenAI
```

```
generation_prompt = ChatPromptTemplate.from_messages(
```

```
[{"role": "system",  
 "content": "You are a twitter techie influencer assistant tasked with writing excellent twitter posts."  
 "content": " Generate the best twitter post possible for the user's request."  
 "content": " If the user provides critique, respond with a revised version of your previous attempts."},  
 MessagesPlaceholder(variable_name="messages")]
```

Repeat
N times

```
llm = ChatOpenAI(model="gpt-4.1-nano")
generate_chain = generation_prompt | llm
```



Reflect

LLM Agent

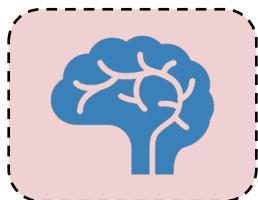


Reflection Agent

Generate



Repeat
N times



Reflect

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI

reflection_prompt = ChatPromptTemplate.from_messages([
    {
        "system": "You are a viral twitter influencer grading a tweet."
        "Generate critique and recommendations for the user's tweet."
        "Always provide detailed recommendations,"
        "including requests for length, virality, style, etc.",
    },
    MessagesPlaceholder(variable_name="messages"),
])

reflect_chain = reflection_prompt | llm
```

LLM Agent



Reflection Agent

Generate



```
from typing import List, Sequence
from langchain_core.messages import BaseMessage
from langgraph.graph import END, MessageGraph
```

```
REFLECT = "reflect"
GENERATE = "generate"
```



```
def generation_node(state: Sequence[BaseMessage]):
    return generate_chain.invoke({"messages": state})
```



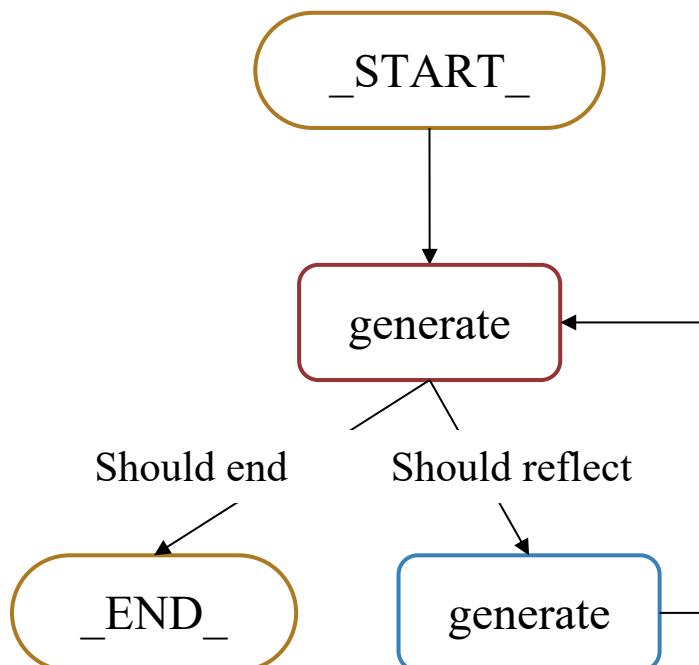
```
def reflection_node(messages: Sequence[BaseMessage]) -> List[BaseMessage]:
    res = reflect_chain.invoke({"messages": messages})
    return [HumanMessage(content=res.content)]
```

Reflect

LLM Agent



Reflection Agent



```
builder = MessageGraph()
builder.add_node(GENERATE, generation_node)
builder.add_node(REFLECT, reflection_node)
builder.set_entry_point(GENERATE)

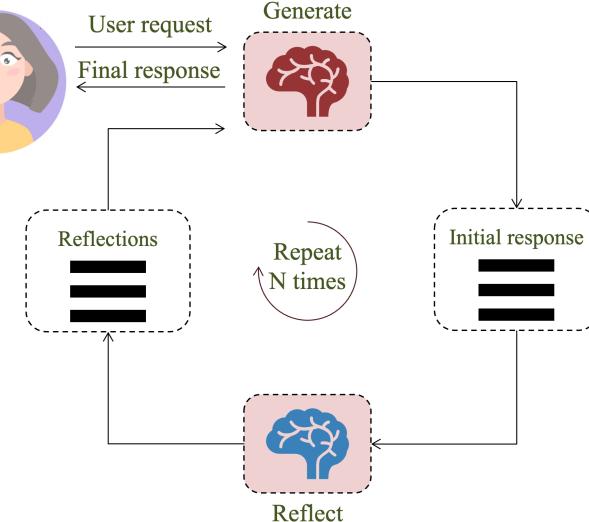
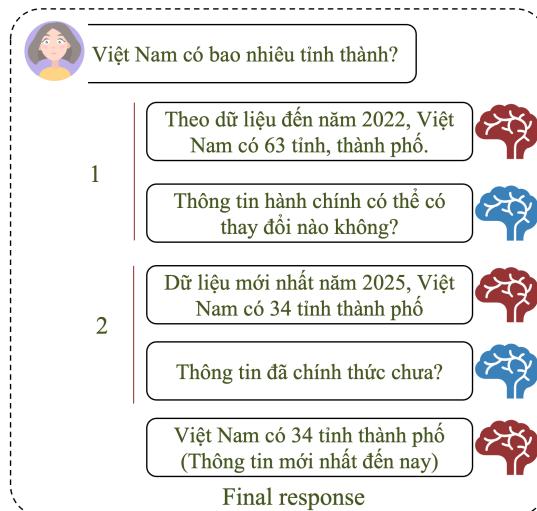
def should_continue(state: List[BaseMessage]):
    if len(state) > 6:
        return END
    return REFLECT

builder.add_conditional_edges(
    "generate", should_continue, {END:END, "reflect":"reflect"}
)
builder.add_edge(REFLECT, GENERATE)
graph = builder.compile()
graph.get_graph().draw_mermaid_png(output_file_path="graph.png")
graph.get_graph().print_ascii()
```

Outline

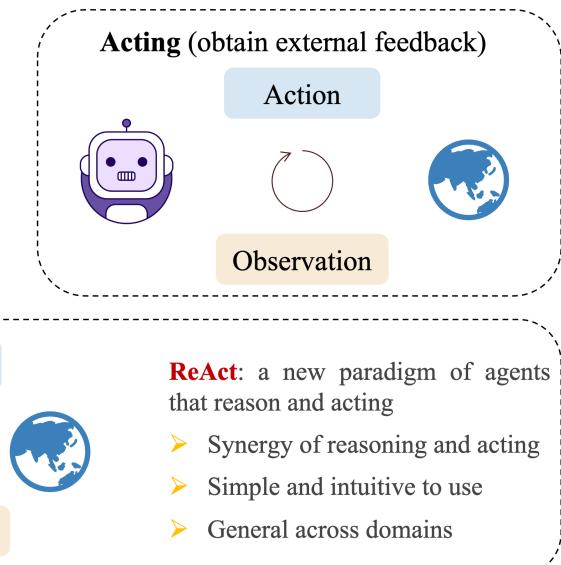
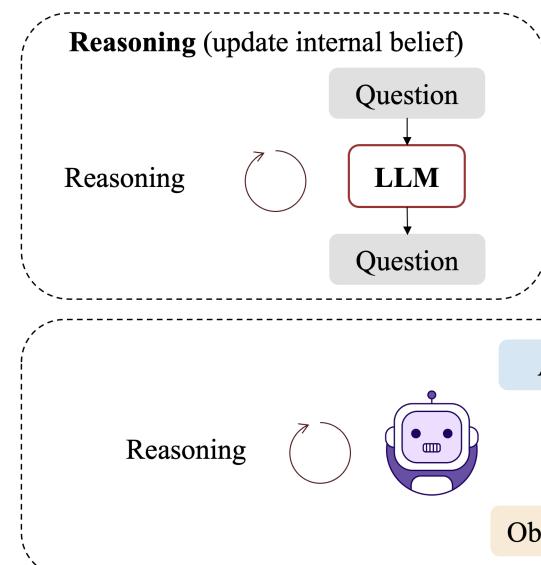
SECTION 1

LLM Agent



SECTION 2

Reasoning Agents



ReAct: a new paradigm of agents that reason and acting

- Synergy of reasoning and acting
- Simple and intuitive to use
- General across domains

Reasoning Agents



Question Answering

Q: what is $1 + 2$?



A: 3

Q: Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder for \$2 per egg. How much does she make every day?



Requires reasoning

Q: who is the latest UK PM?



Requires knowledge

Q: what is the prime factorization of 34324329?



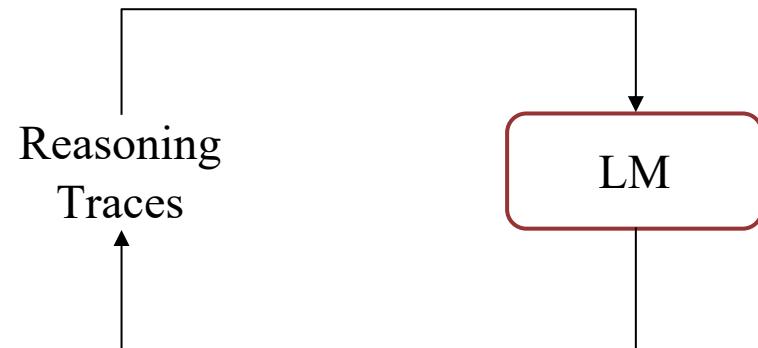
Requires computation

Reasoning Agents



Reasoning OR Acting

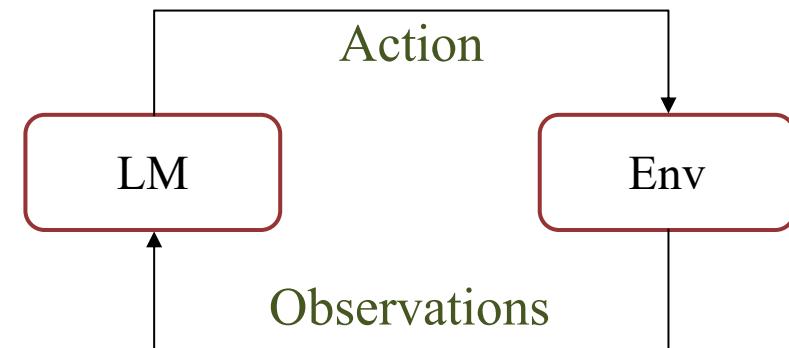
CoT



Flexible and general to augment
test-time compute

Lack of external knowledge and tools

RAG / Retrieval /
Code / Tool Use



Flexible and general to augment
knowledge, computation, feedback, etc.

Lack of reasoning

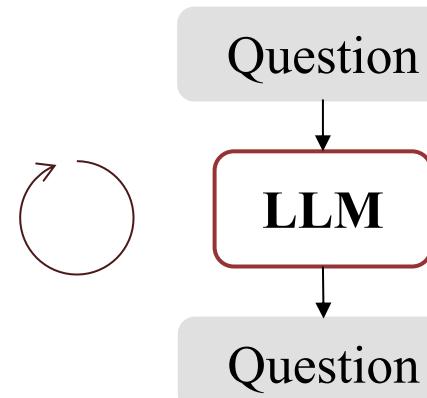
Retrieval / Search Engine / Calculator
/ Yolo / Python / ...

Reasoning Agents

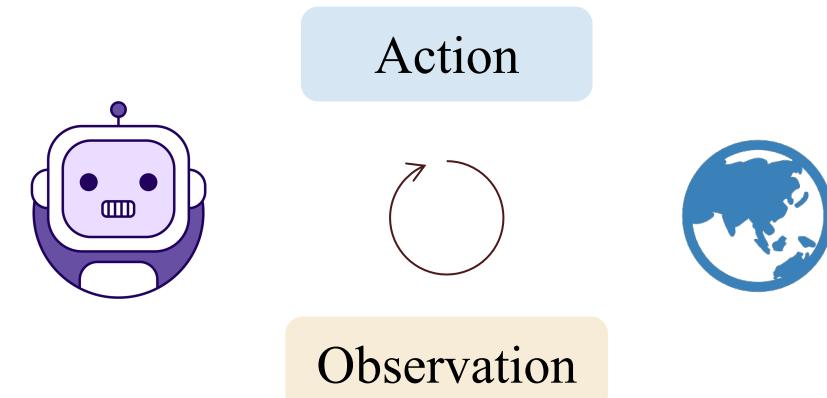


ReAct: Reasoning + Acting

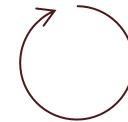
Reasoning (update internal belief)



Acting (obtain external feedback)



Reasoning

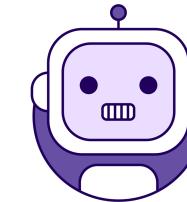


Question

LLM

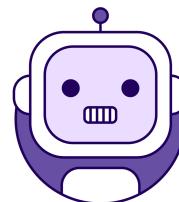
Question

Action



Observation

Reasoning



Action



Observation

ReAct: a new paradigm of agents that reason and acting

- Synergy of reasoning and acting
- Simple and intuitive to use
- General across domains

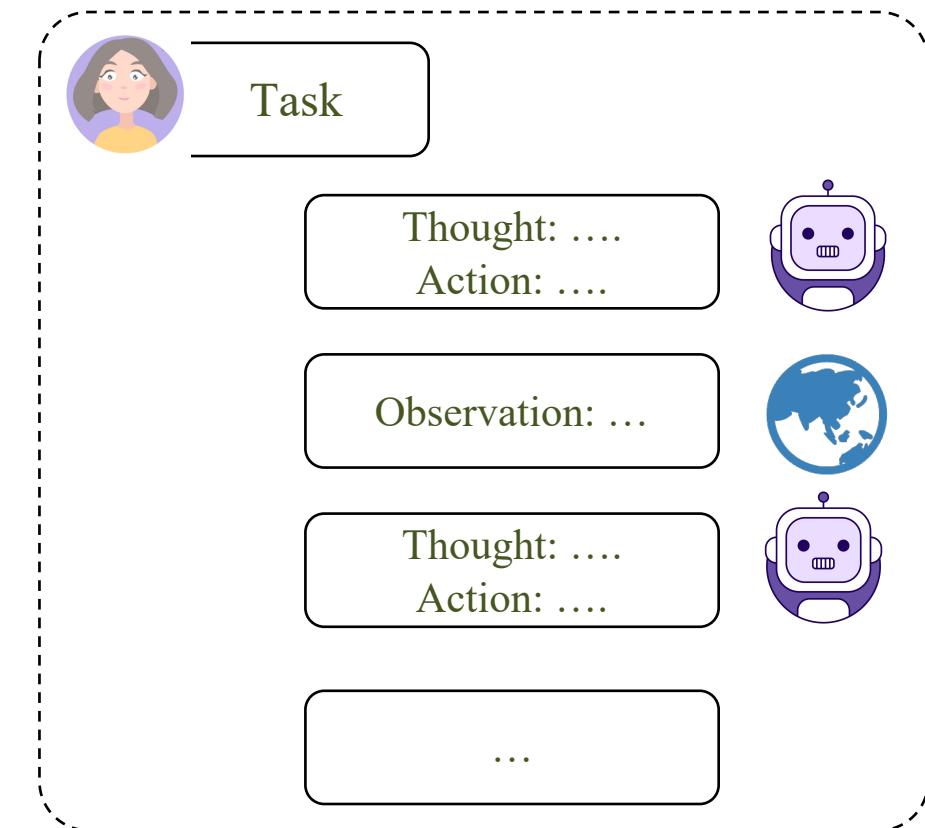
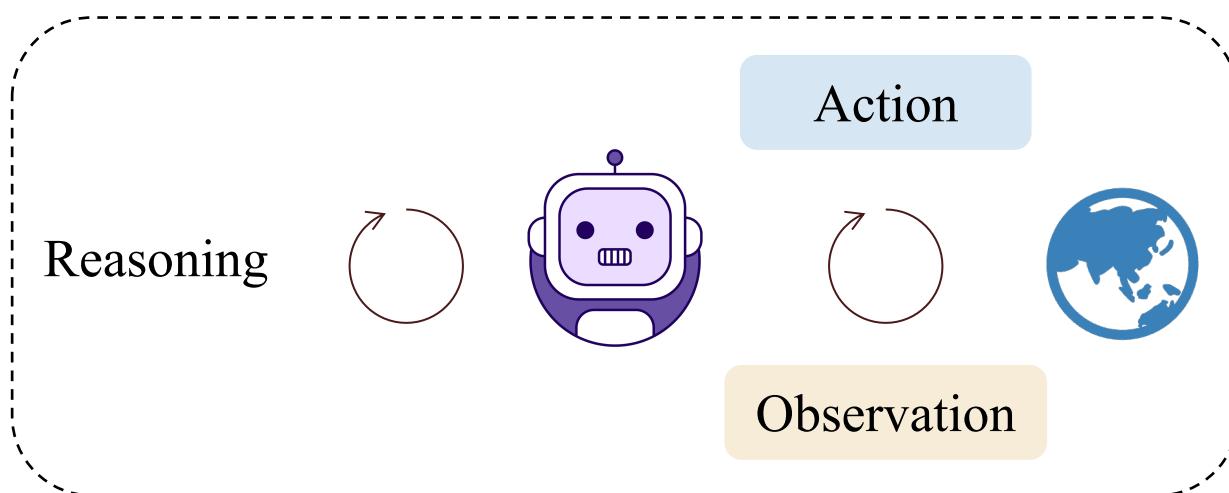
Reasoning Agents



ReAct is simple and intuitive to use

➤ ReAct supports

- One-shot prompting
- Few-shot prompting
- Fine-tuning



Reasoning Agents



A zero-shot ReAct prompt

➤ ReAct supports

- One-shot prompting
- Few-shot prompting
- Fine-tuning

You are an agent that answers questions by using two actions:

- search[query]: Google search the query. You can also use it to calculate math.
- finish[answer]: return the answer

Your generation should have the following format:

Thought: to analyze the observation and inform the action.

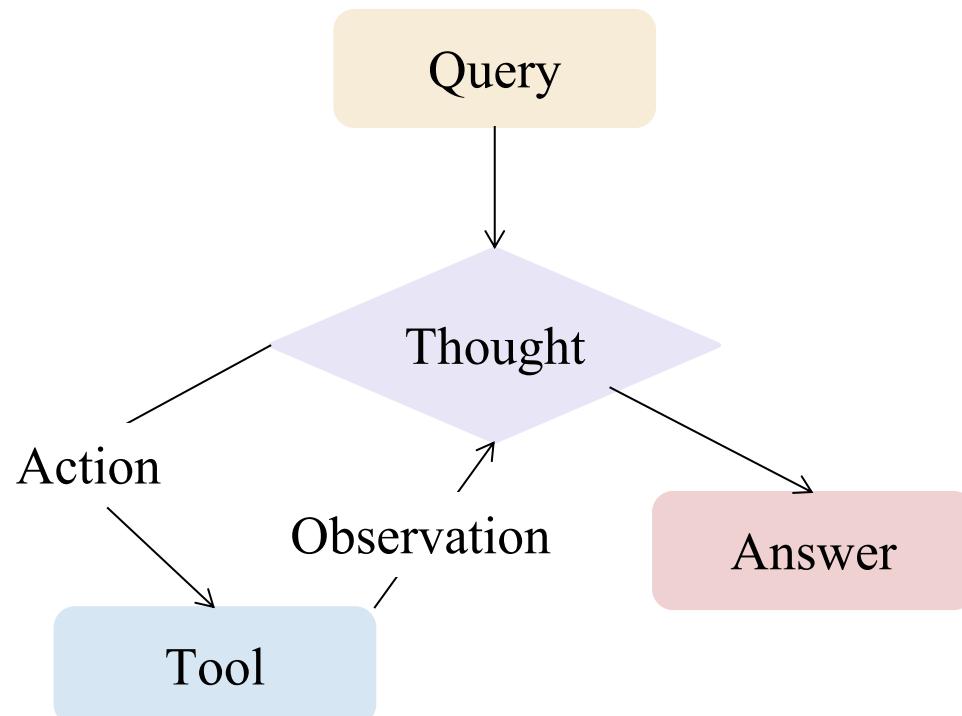
Action: your action

Question: If I have seven trillion dollars today, can I buy Apple, Nvidia, and Microsoft? If not, how much more money do I need?

Reasoning Agents



Example: Travel Planning Agent



⌚ Task: Book a flight from Hanoi to Ho Chi Minh City for tomorrow morning.

Loop 1:

🧠 THOUGHT:

"Let me find the earliest flight available tomorrow."

💪 ACTION

Search for flights Hanoi → HCMC, sorted by departure time.

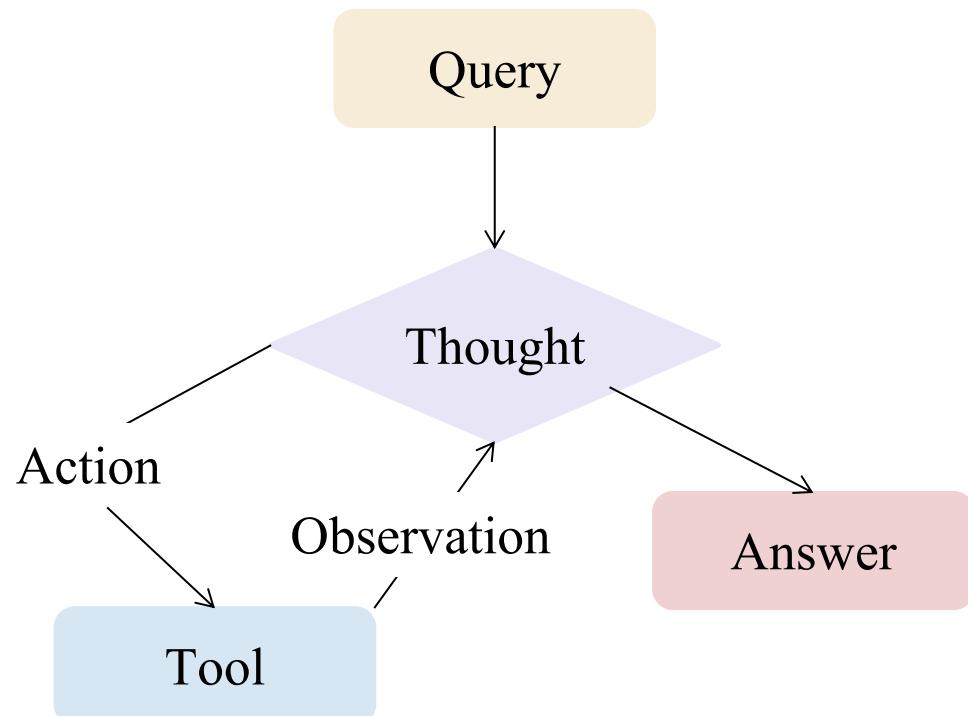
👀 OBSERVATION

Finds 5:30 AM flight on Airline A.

Reasoning Agents



Example: Travel Planning Agent



⌚ Task: Book a flight from Hanoi to Ho Chi Minh City for tomorrow morning.

Loop 2:

🧠 THOUGHT:
"This is the earliest. I'll go ahead and book it."

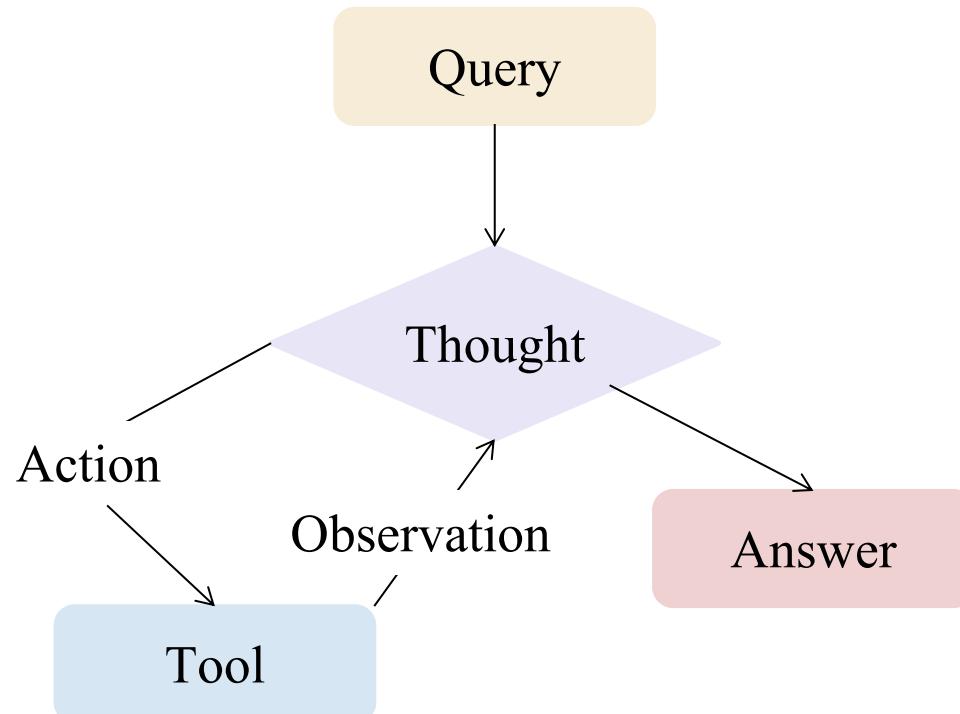
💪 ACTION
Attempt to book 5:30 AM flight on Airline A.

👀 OBSERVATION
Booking failed: "This flight is fully booked."

Reasoning Agents



Example: Travel Planning Agent



⌚ Task: Book a flight from Hanoi to Ho Chi Minh City for tomorrow morning.

Loop 3:

🧠 THOUGHT:

"Ah, I need to pick the next earliest available flight instead."

💪 ACTION

Book the 6:15 AM flight on Airline B.

👀 OBSERVATION

Booking confirmed successfully.

Reasoning Agents



Acting without Reasoning

You are in the middle of a room. Looking quickly around you, you see a cabinet 6, a cabinet 1, a coffee machine 1, a countertop 3, a stove burner 1, and a toaster 1.

Your task is to: Put some pepper shaker on a drawer.

Act only

Act 1: Go to drawer 1

Obs 1: The drawer 1 is closed.

Act 2: Open drawer 1

Obs 2: You open the drawer 1. The drawer 1 is open. In it, you see a dishponge 2, and a spoon 1.

Act 3: Go to sinkbasin 1

Obs 3: On the sinkbasin 1, you see a dishponge 3, a spatula 1, and a spoon 2.

Act 4: Take peppershaker 1 from sinkbasin 1

Obs 4: Nothing happens.

Act 5: Take peppershaker 1 from sinkbasin 1

Obs 5: Nothing happens.

.....

X

Reasoning Agents



ReAct Enables Systematic Exploration

You are in the middle of a room. Looking quickly around you, you see a cabinet 6, a cabinet 1, a coffee machine 1, a countertop 3, a stove burner 1, and a toaster 1.

Your task is to: Put some pepper shaker on a drawer.

ReAct

Act 1: Think[First I need to find a peppe shaker...more likely to appear in cabinets (1-6), countertops (1-3), ...]

Act 2: Go to cabinet 1

Obs 2: On the cabinet 1, you see a vase 2.

(Here the agent goes to cabinet 1, 2, 3, 4, 5, 6, then countertop 1 and 2)

Act 9: Go to countertop 3

Obs 9: On the countertop 3, you see a apple 1, a bread 1, a pepper shaker 1, and a vase 3.

Act 10: Take pepper shaker 1 from countertop 3

Obs 10: You pick up the pepper shaker 1 from the countertop 3.

Act 11: Think[Now I find a pepper shaker 1. Next, I need to put it in/on drawer 1.]

Act 12: Go to drawer 1

Obs 12: Drawer 1 is closed.

Act 13: Open drawer 1

Obs 13: You open Drawer 1 ...

Act 14: Put pepper shaker 1 in/on drawer 1

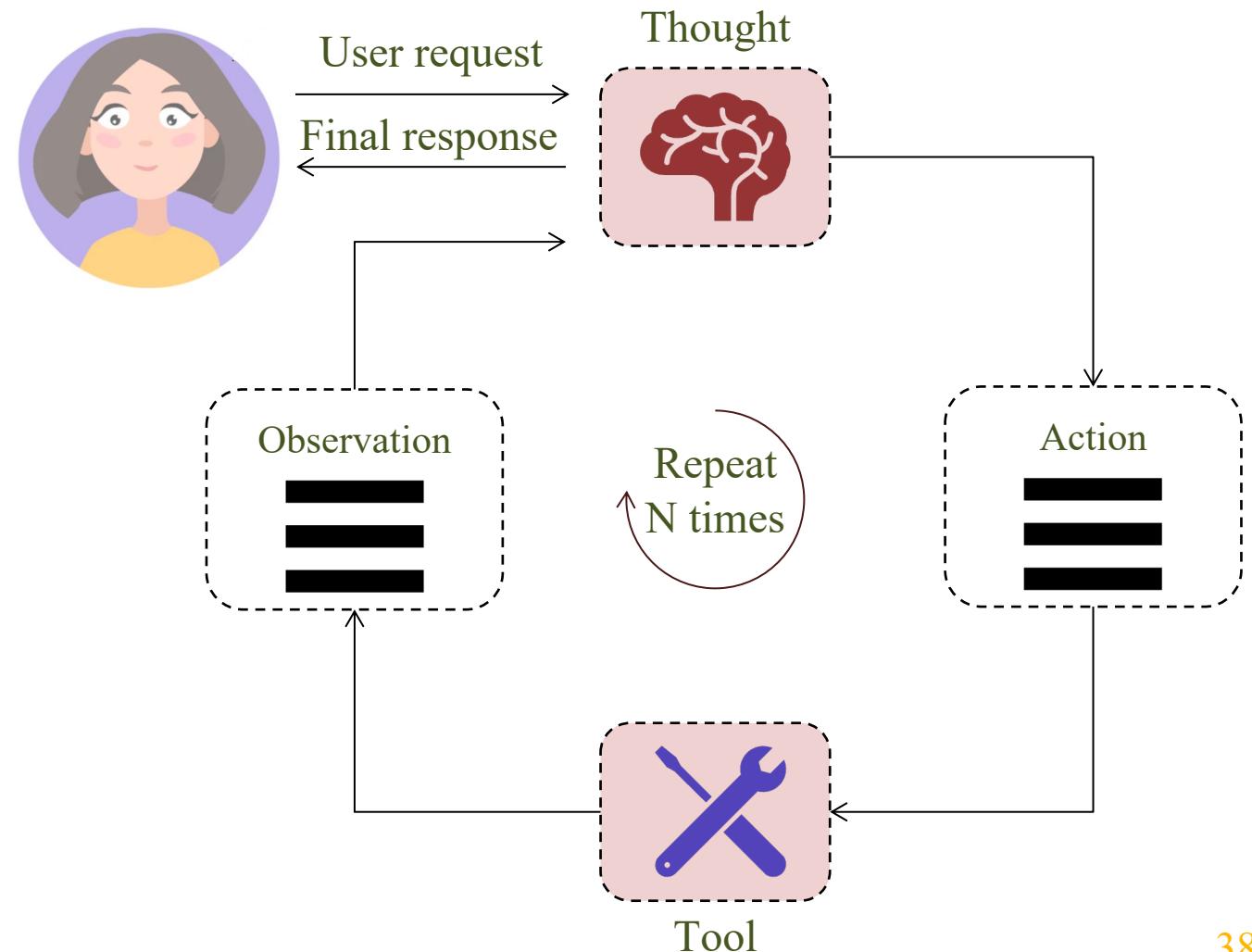
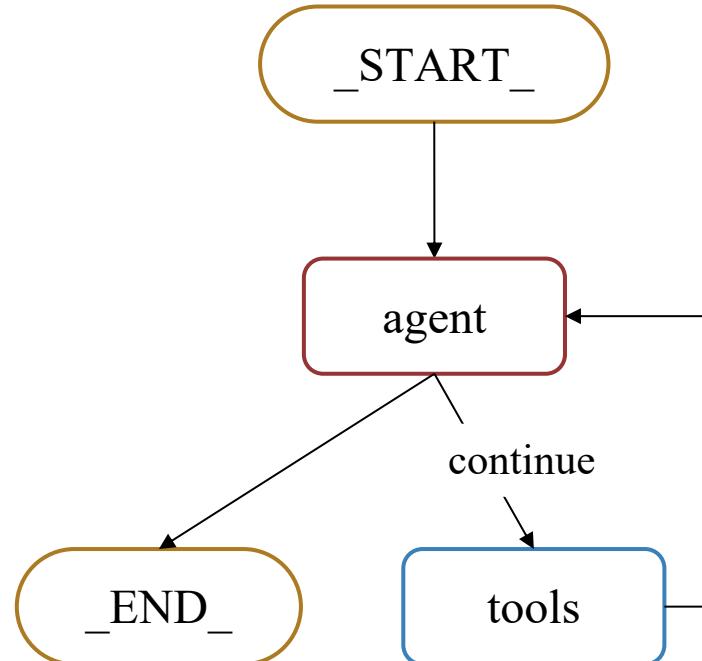
Obs 14: You put pepper shaker 1 in/on the drawer 1.



Reasoning Agents



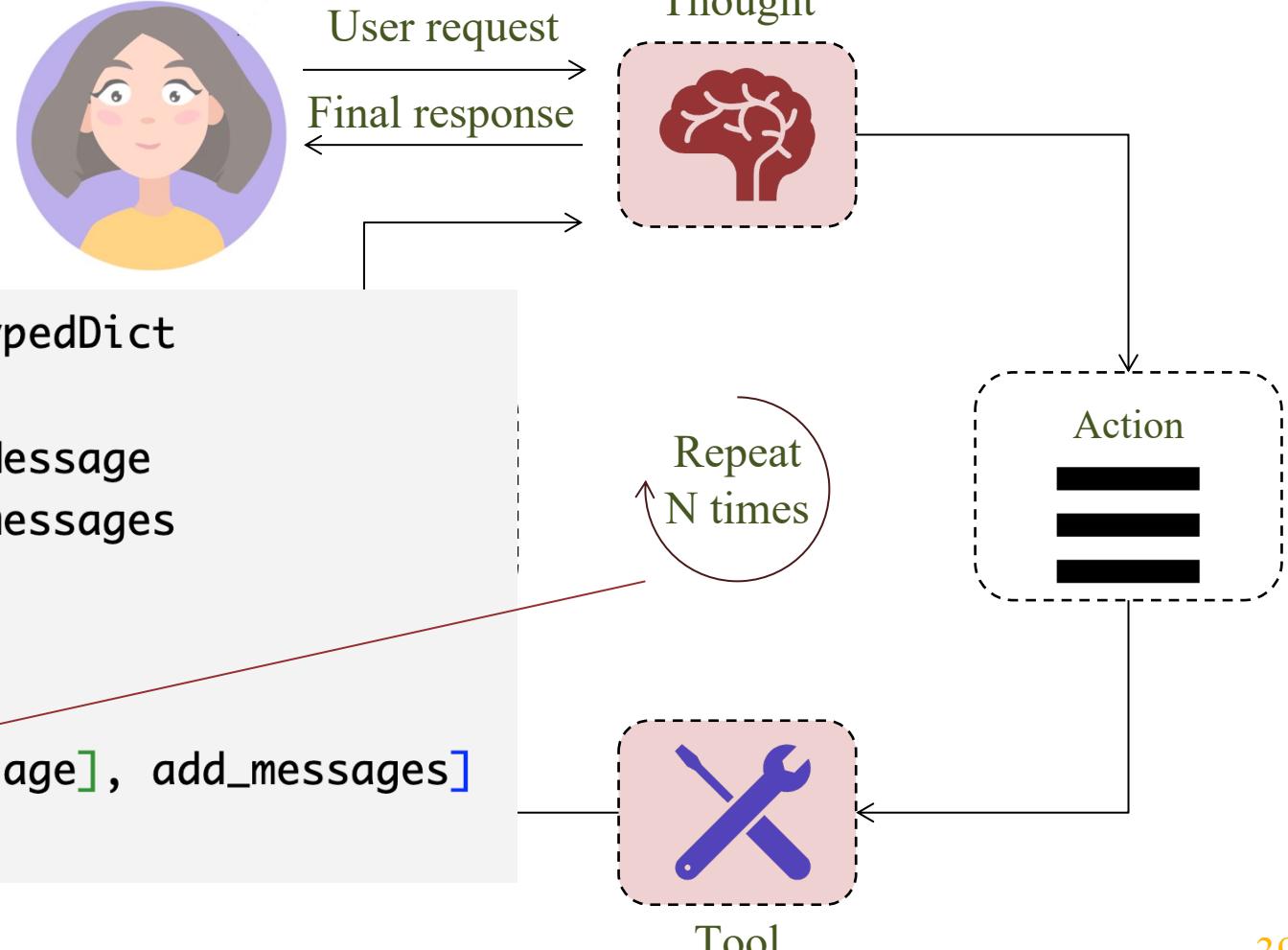
ReAct Agent



Reasoning Agents



ReAct Agent

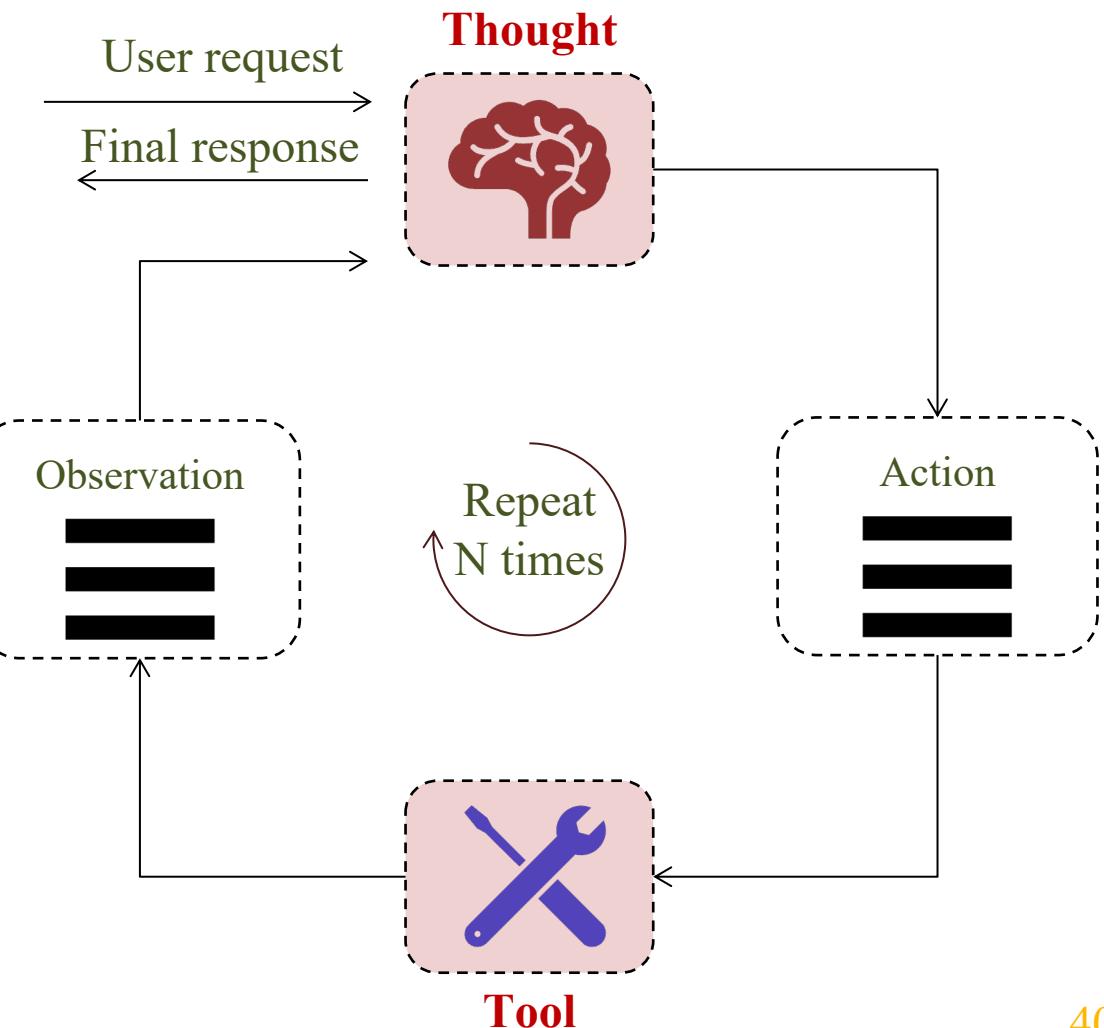


Reasoning Agents



ReAct Agent

```
model = ChatOpenAI(model="gpt-4.1-nano")  
  
from langchain_tavily import TavilySearch  
from langchain_core.tools import tool  
  
@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]
```



Reasoning Agents



ReAct Agent

```
from langchain_core.messages import SystemMessage
```

```
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

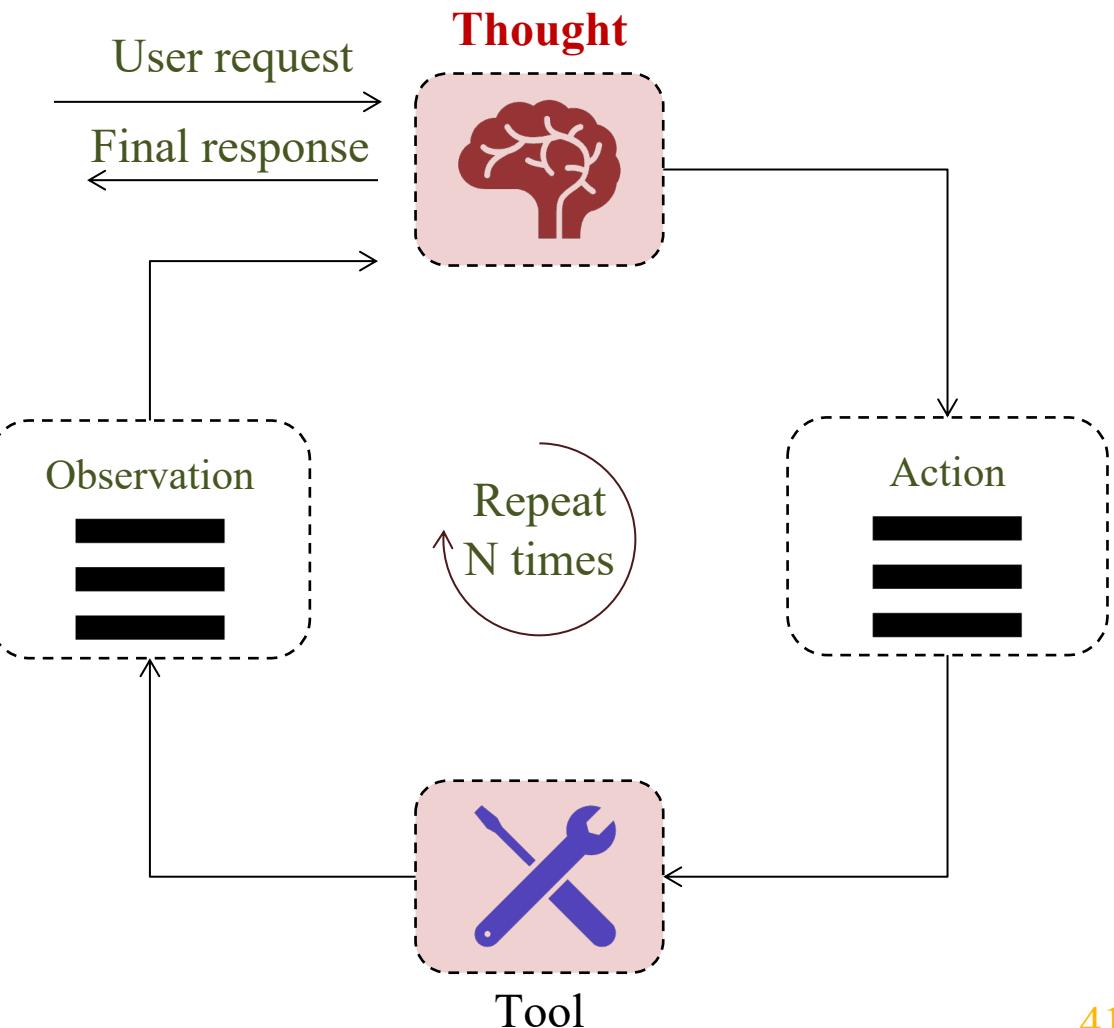
(Repeat Thought/Action/Observation if needed)

Final Answer: your answer to the user

Question: {input}

```
"""
```

```
)
```



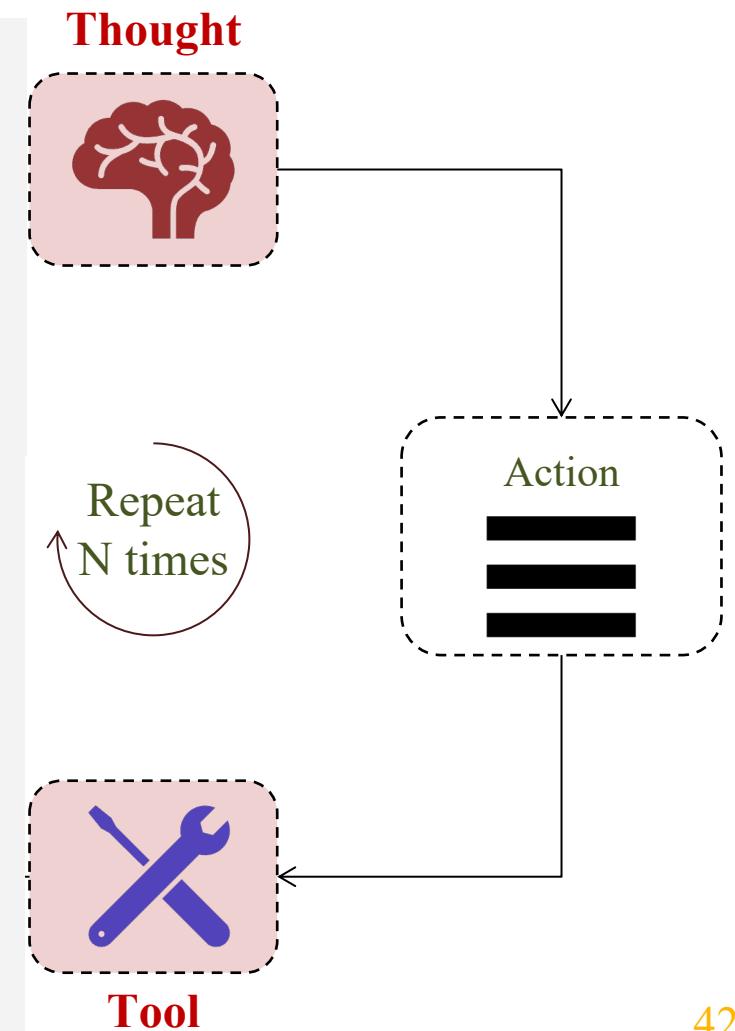
Reasoning Agents



ReAct Agent

```
# 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 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}
```

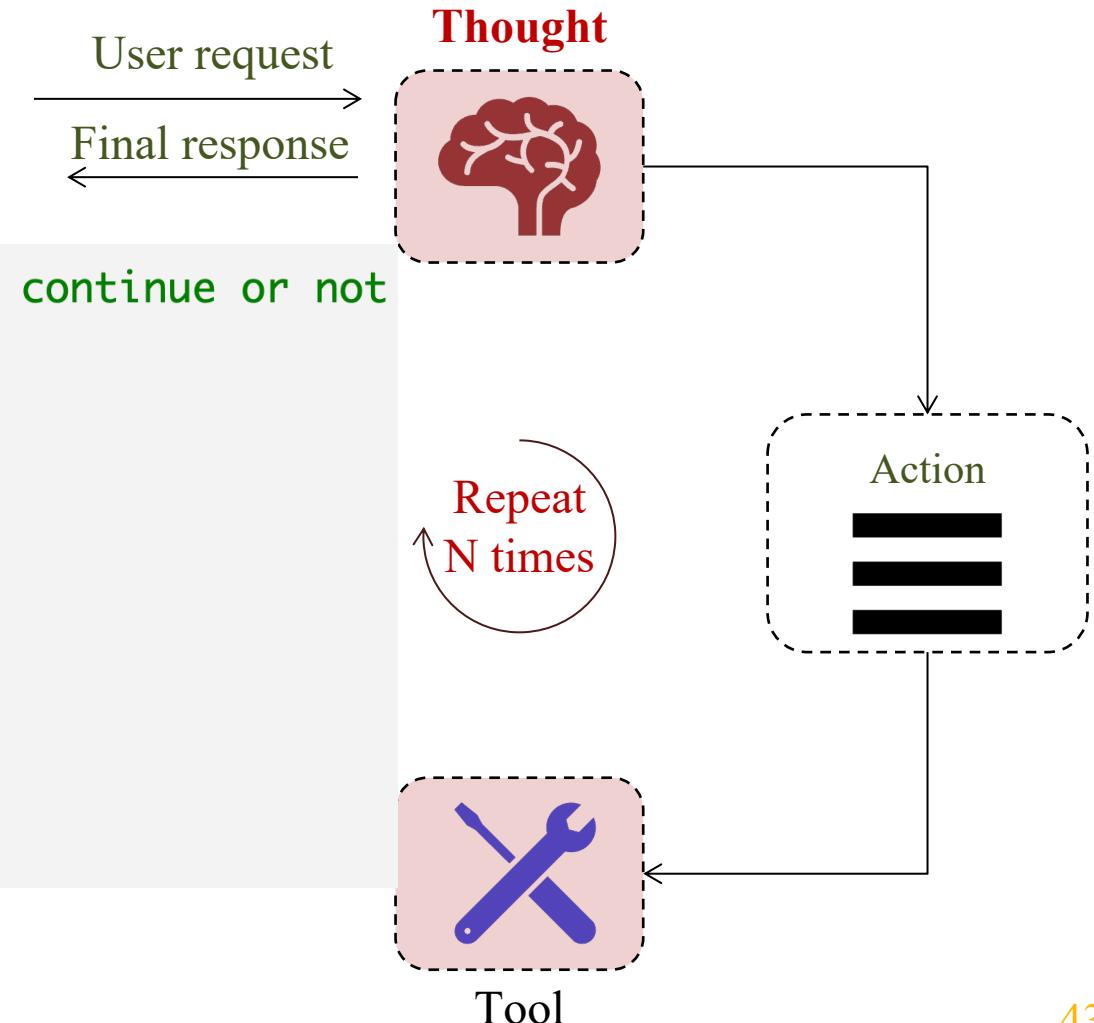


Reasoning Agents



ReAct Agent

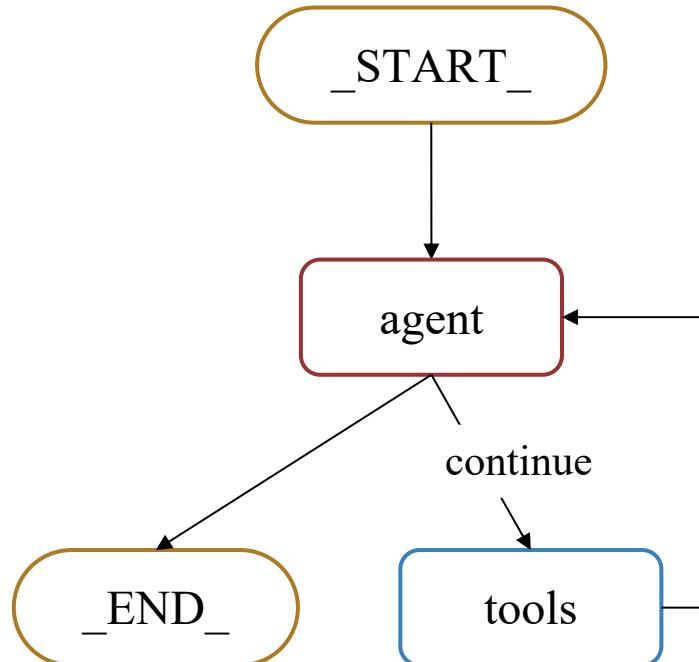
```
# Define the conditional edge that determines whether to continue or not
def should_continue(state: AgentState):
    messages = state["messages"]
    last_message = messages[-1]
    # If there is no function call, then we finish
    if not last_message.tool_calls:
        return "end"
    # Otherwise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



Reasoning Agents



ReAct Agent



```
from langgraph.graph import StateGraph, END

# Define a new graph
workflow = StateGraph(AgentState)

workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)

workflow.set_entry_point("agent")
workflow.add_conditional_edges(
    "agent",
    {
        "continue": "tools",
        "end": END,
    },
)

workflow.add_edge("tools", "agent")
graph = workflow.compile()
```

QUIZ TIME

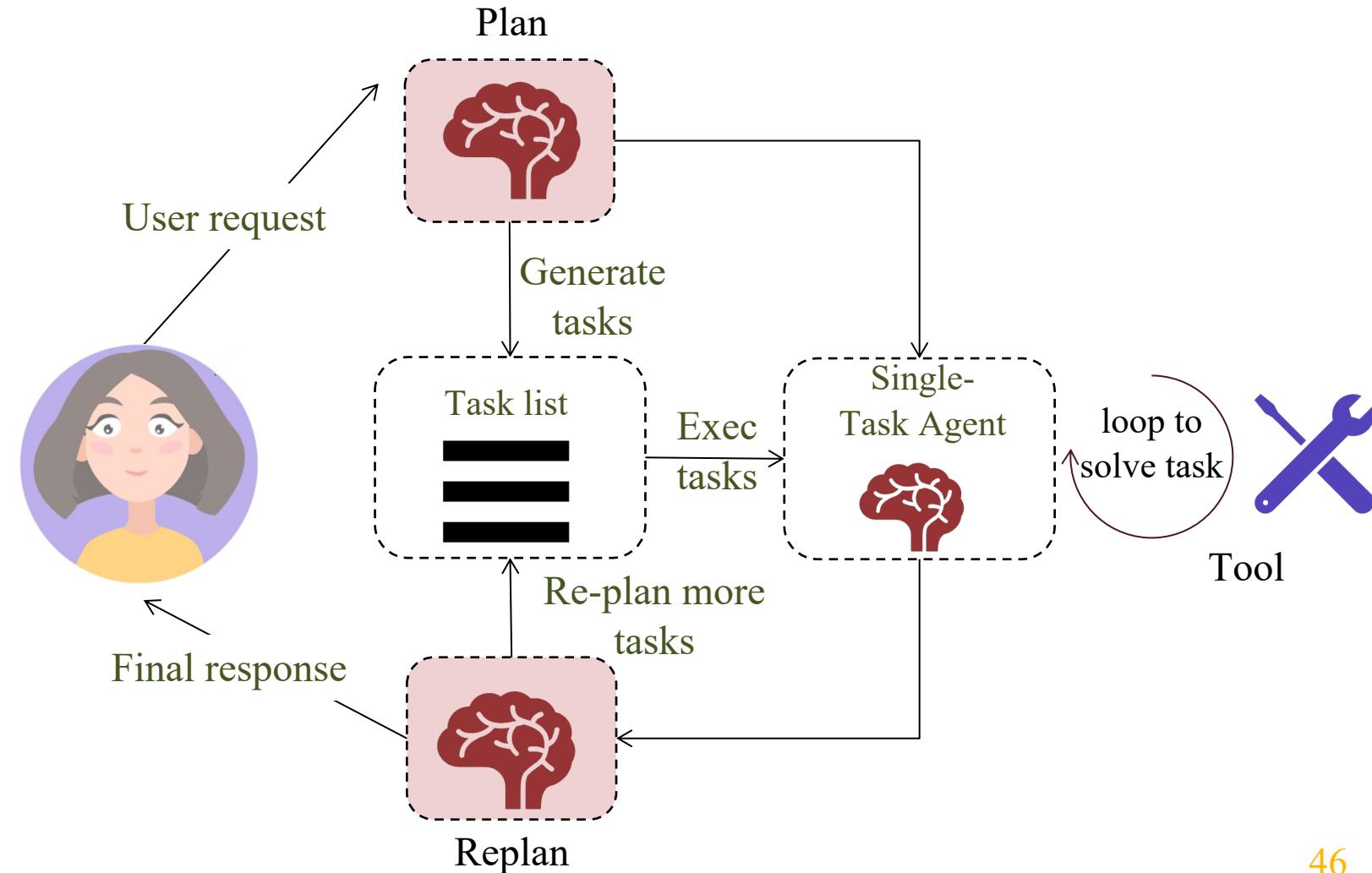
Reasoning Agents



Plan-and-Execute Agent

➤ Two basic components:

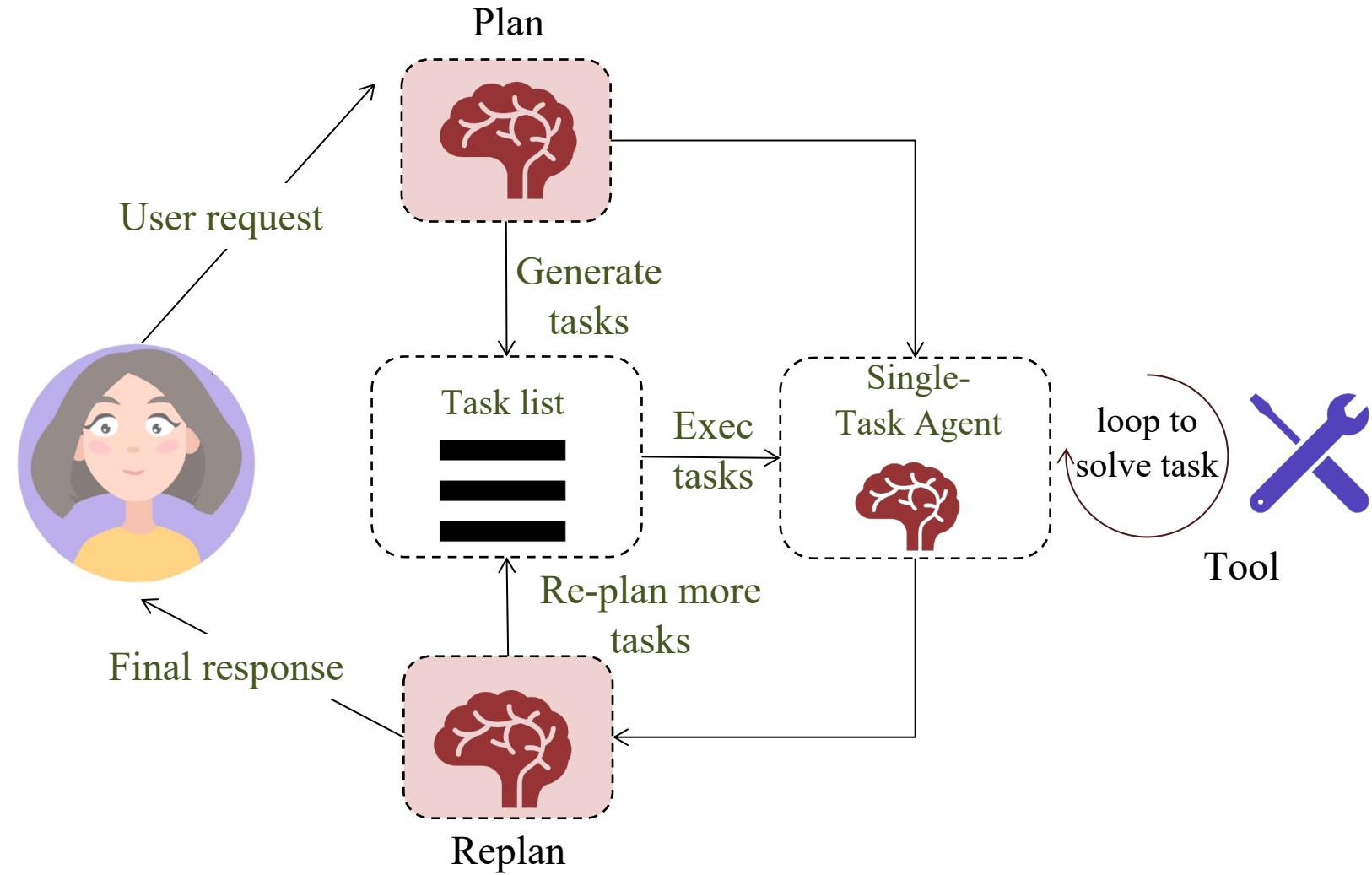
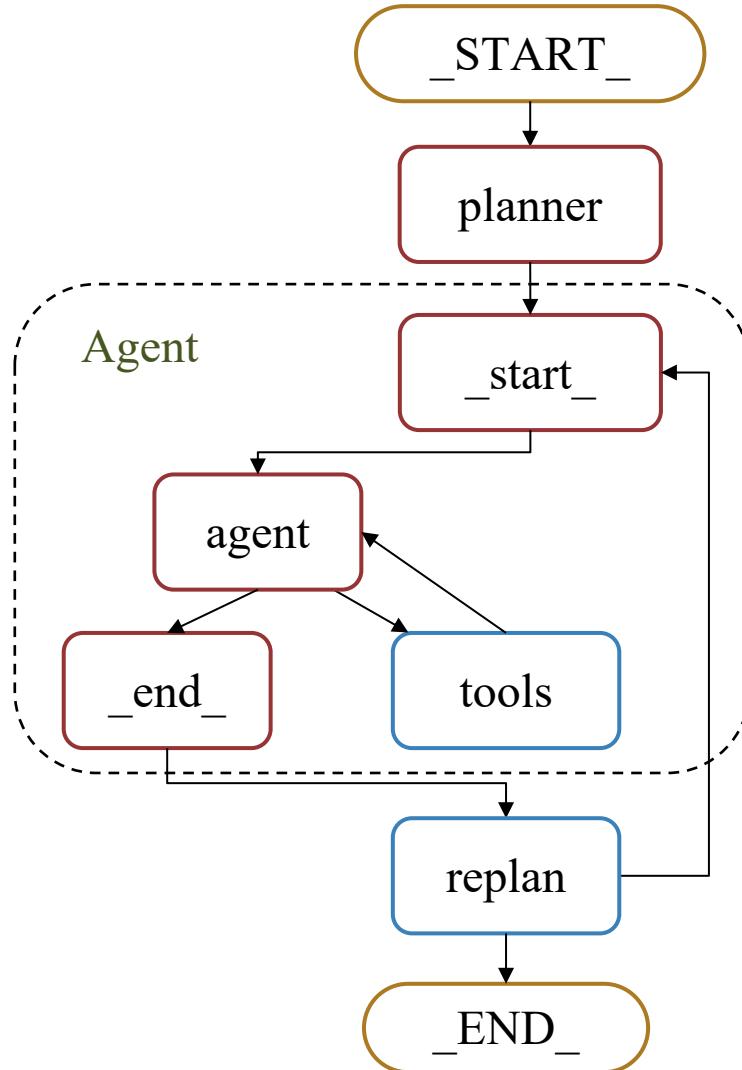
- A planner, which prompts an LLM to generate a multi-step plan to complete a large task.
- Executor(s), which accept the user query and a step in the plan and invoke 1 or more tools to complete that task.



Reasoning Agents



Plan-and-Execute Agent



Reasoning Agents



Plan-and-Execute Agent

Q: In a dance class of 20 students, 20% enrolled in contemporary dance, 25% of the remaining enrolled in jazz dance, and the rest enrolled in hip-hop dance. What percentage of the entire students enrolled in hip-hop dance?

A: *Let's first understand the problem and devise a plan to solve the problem.*

Then, let's carry out the plan and solve the problem step by step.

→ LLMs →

Plan:

Step 1: Calculate the total number of students who enrolled in contemporary and jazz dance.
Step 2: Calculate the total number of students who enrolled in hip-hop dance.
Step 3: Calculate the percentage of students who enrolled in hip-hop dance.

Solution:

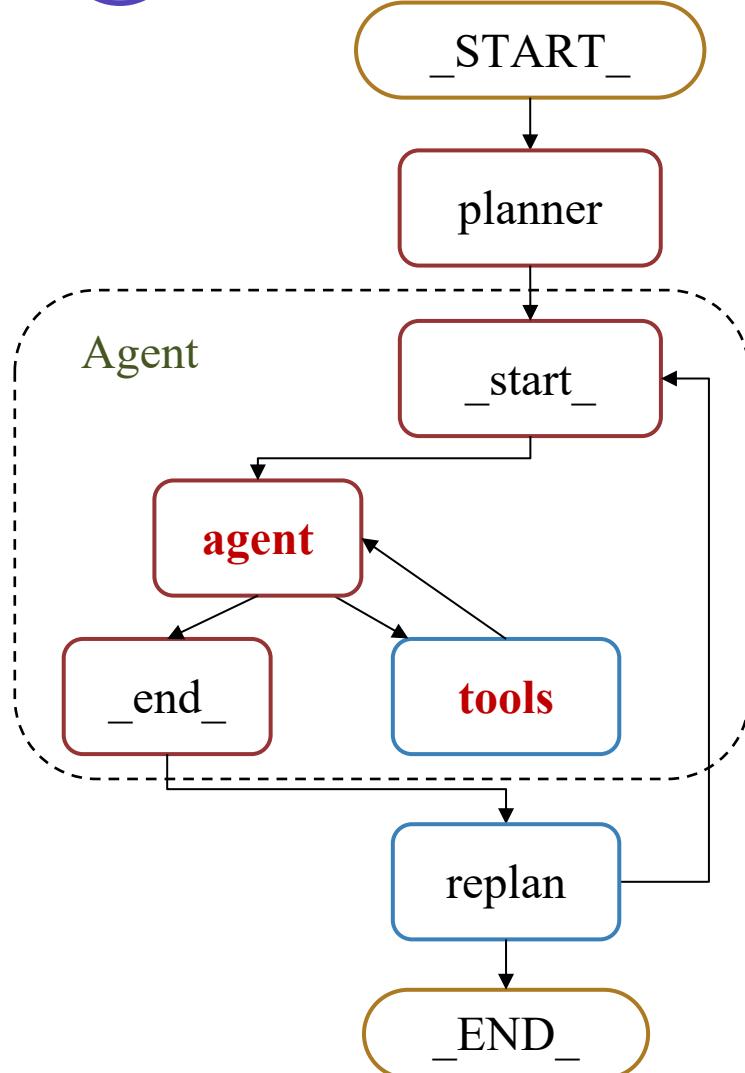
Step 1: 20% of 20 students is 4 students. 25% of the remaining 16 students is 4 students. So, a total of 8 students enrolled in contemporary and jazz dance.
Step 2: The remaining 12 students enrolled in hip-hop dance.
Step 3: The percentage of students who enrolled in hip-hop dance is $12/20 = 60\%$.



Reasoning Agents



Plan-and-Execute Agent



```
from langchain_openai import ChatOpenAI

from langgraph.prebuilt import create_react_agent

# Choose the LLM that will drive the agent
llm = ChatOpenAI(model="gpt-4o")
prompt = "You are a helpful assistant."
agent_executor = create_react_agent(llm, tools, prompt=prompt)
```

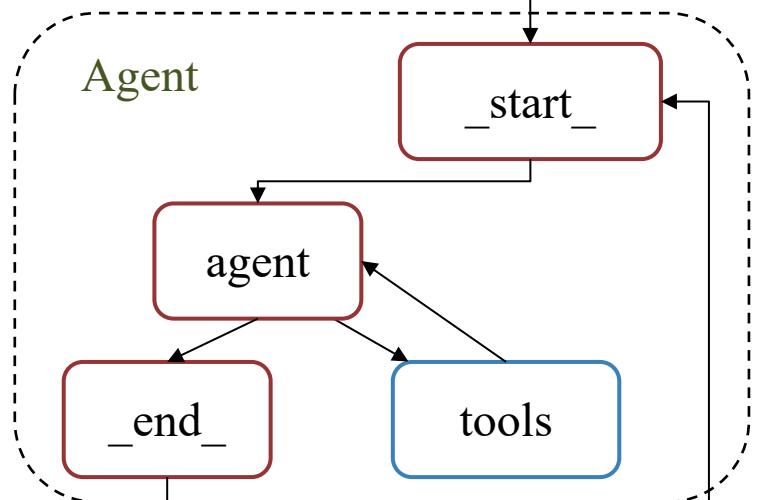
```
from langchain_tavily import TavilySearch

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

Reasoning Agents



Plan-and-Execute Agent



Agent

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

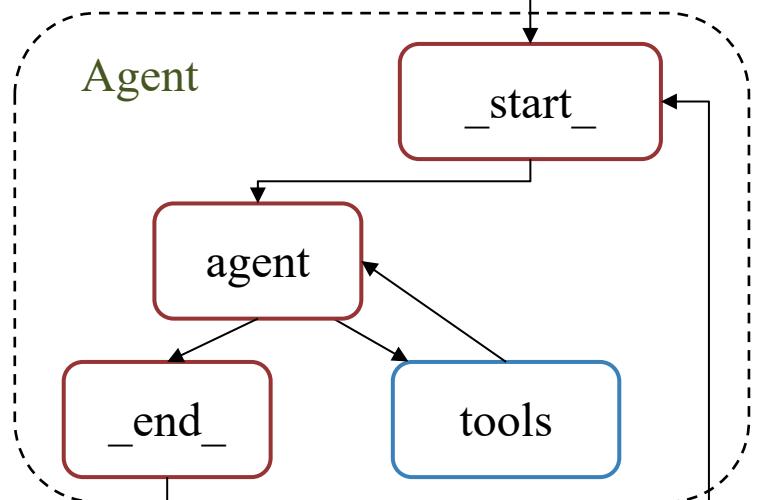
    steps: List[str] = Field(
        description="different steps to follow, should be in sorted order"
    )

    planner_prompt = ChatPromptTemplate.from_messages([
        (
            "system",
            """For the given objective, come up with a simple step by step plan""",
        ),
        ("placeholder", "{messages}"),
    ])
    planner = planner_prompt | ChatOpenAI(
        model="gpt-4o", temperature=0
    ).with_structured_output(Plan)
```

Reasoning Agents



Plan-and-Execute Agent



```
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."
    )

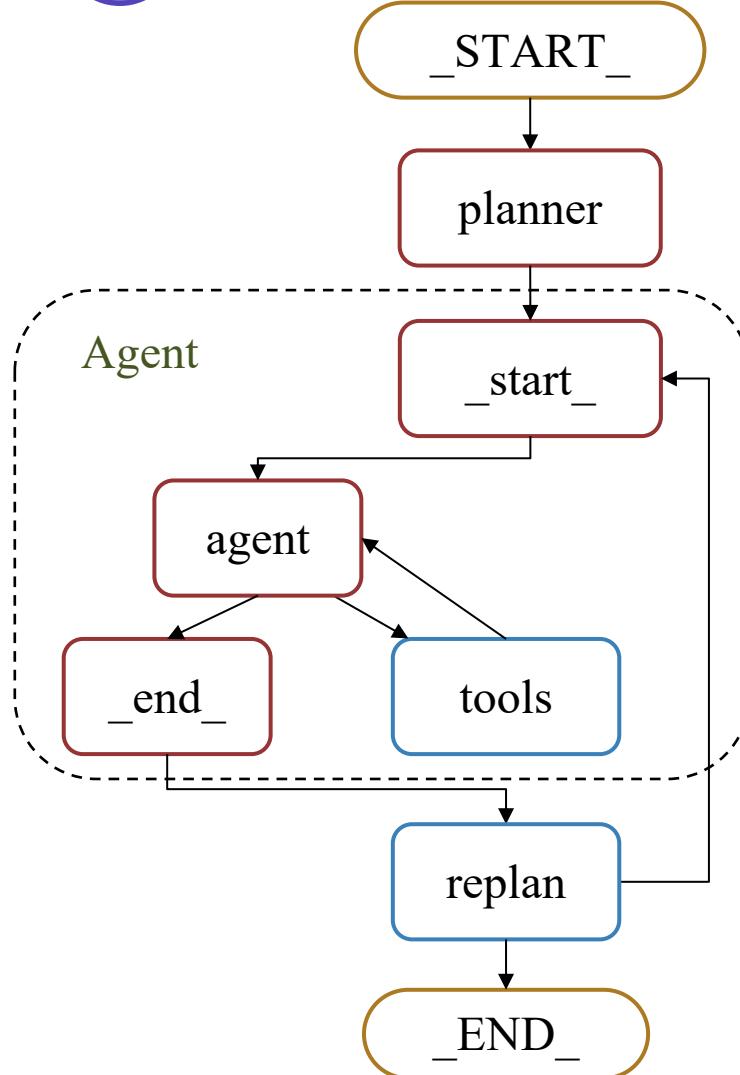
replanner_prompt = ChatPromptTemplate.from_template(
    """Update plan"""
)

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

Reasoning Agents



Plan-and-Execute Agent



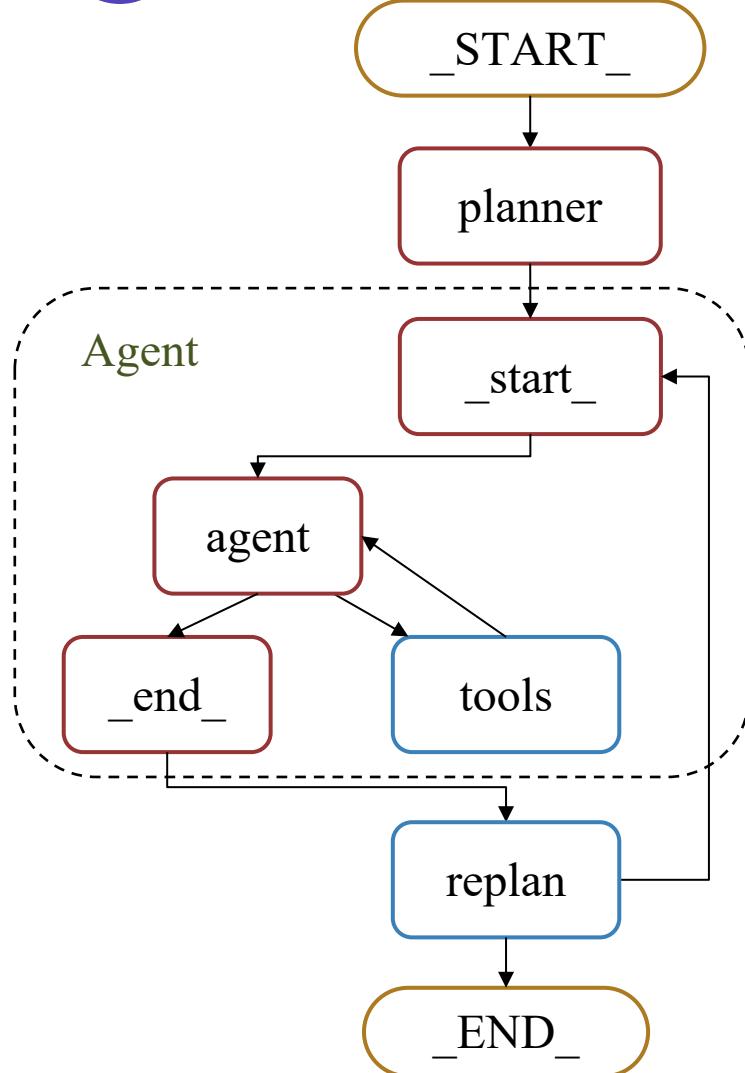
```
async def execute_step(state: PlanExecute):
    plan = state["plan"]
    plan_str = "\n".join(f"{i+1}. {step}" for i, step in enumerate(plan))
    task = plan[0]
    task_formatted = f"""For the following plan:
    {plan_str}\n\nYou are tasked with executing step {1}, {task}."""
    agent_response = await agent_executor.invoke(
        {"messages": [("user", task_formatted)]}
    )
    return {
        "past_steps": [task, agent_response["messages"][-1].content],
    }

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

Reasoning Agents



Plan-and-Execute Agent

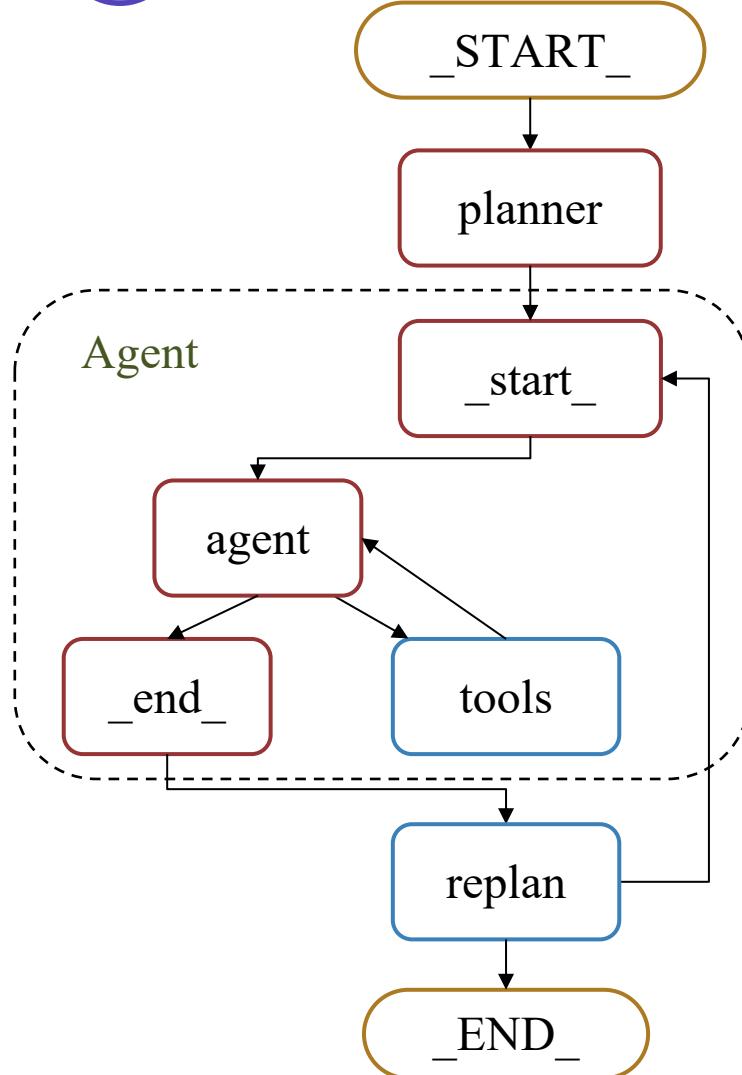


```
# Define the conditional edge that determines whether to continue
def should_continue(state: AgentState):
    messages = state["messages"]
    last_message = messages[-1]
    # If there is no function call, then we finish
    if not last_message.tool_calls:
        return "end"
    # Otherwise if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

Reasoning Agents



Plan-and-Execute Agent



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

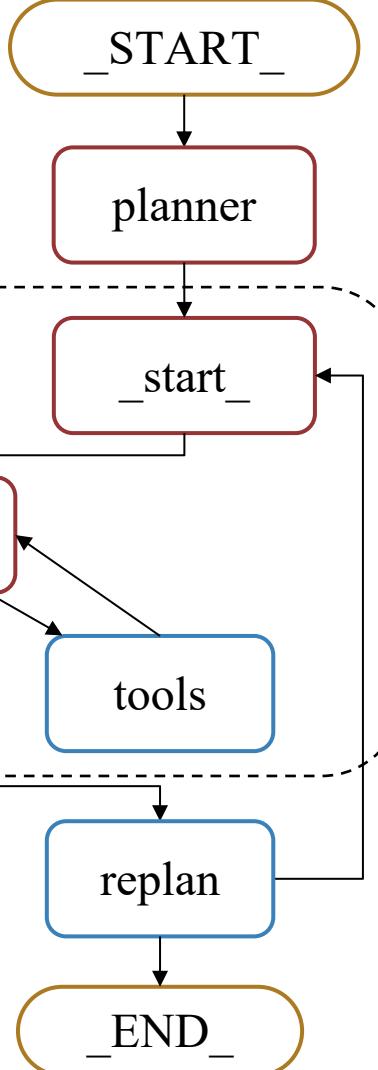
def should_end(state: PlanExecute):
    if "response" in state and state["response"]:
        return END
    else:
        return "agent"
```

Reasoning Agents



Plan-and-Execute Agent

Agent

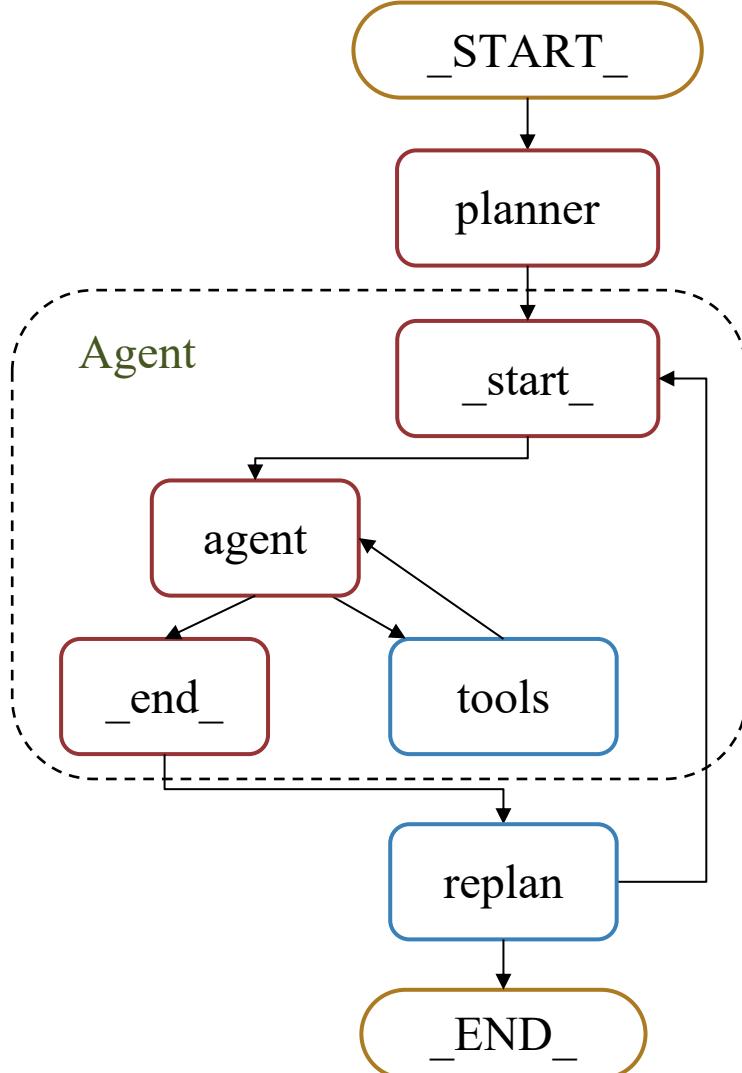


```
workflow = StateGraph(PlanExecute)  
  
workflow.add_node("planner", plan_step)  
workflow.add_node("agent", execute_step)  
workflow.add_node("replan", replan_step)  
  
workflow.add_edge(START, "planner")  
workflow.add_edge("planner", "agent")  
workflow.add_edge("agent", "replan")  
  
workflow.add_conditional_edges(  
    "replan",  
    should_end,  
    ["agent", END],  
)  
  
app = workflow.compile()
```

Reasoning Agents



Plan-and-Execute Agent



▼ **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 $\{ \sqrt{x - 2} = 3 \}$...

▼ 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 $\{ \sqrt{x - 2} = 3 \}$. 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.

Thanks!

Any questions?