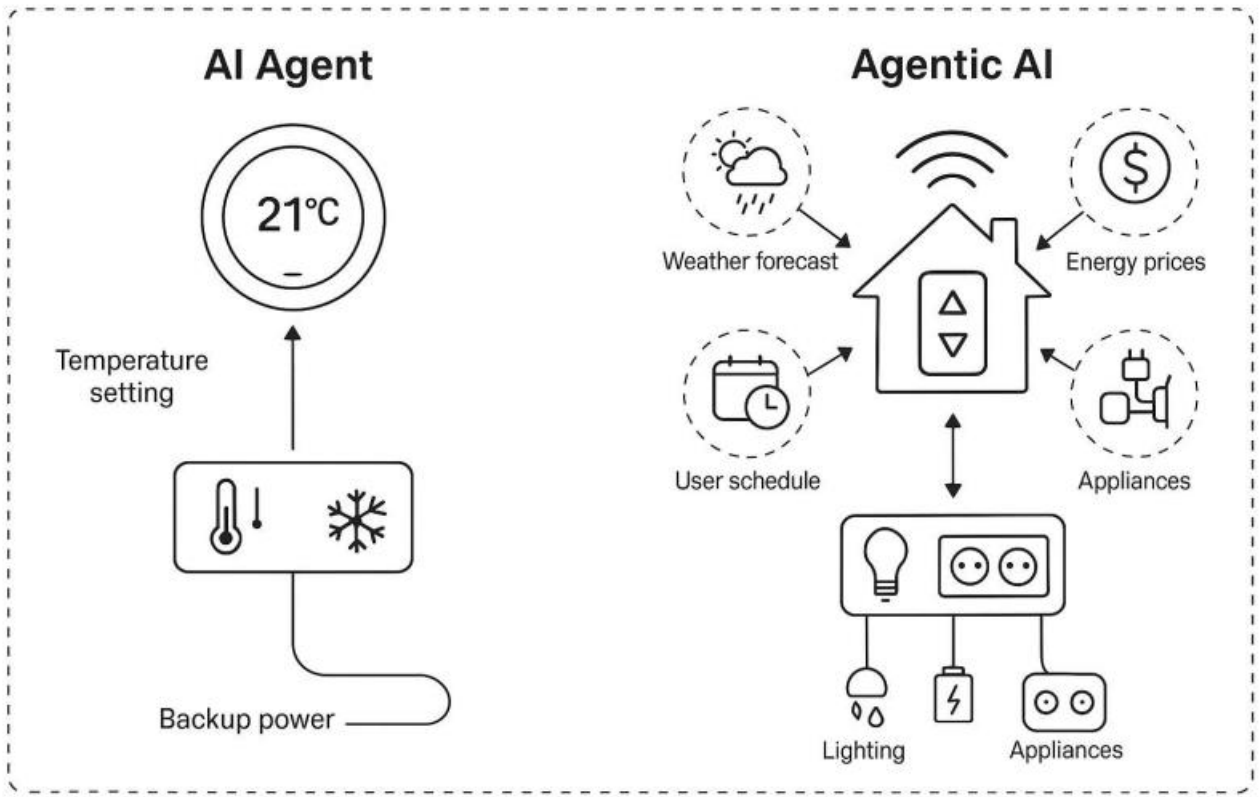
Agentic AI with LangChain and LangGraph

What Is Agentic AI?

Agentic AI takes things a step further. Instead of just one agent doing one job, Agentic AI brings multiple agents together into a team. These agents coordinate tasks, exchange information, adapt roles dynamically, and share memory. Key features include:

- **Task Decomposition**: Goals are split into subtasks automatically.
- **Inter-Agent Communication**: Agents share updates and results via messaging or shared memory.
- **Memory and Reflection**: Agents remember past steps and learn from outcomes.
- **Orchestration**: A lead agent or system coordinates the team.

Example: Planning a vacation—one agent books the flight, another finds hotels and a third checks visa requirements. A coordinator agent makes sure everything matches your preferences.



Key Architectural Differences between an AI Agent and Agentic AI

A structured taxonomy helps clarify the differences:

- **AI Agent**: Single entity, handles one task, uses tools, operates in narrow contexts
- **Agentic AI**: Multi-agent systems, goal-driven orchestration, adapts across time and context, supports parallel task execution

Feature	AI Agent	Agentic AI
Design	One agent, one task	Multiple agents with distinct roles
Communication	No coordination with others	Constant communication and coordination
Memory	Stateless or minimal history	Persistent memory of tasks, outcomes, and strategies
Reasoning	Linear logic (do step A → B)	Iterative planning and re-planning with advanced reasoning
Scalability	Limited to task size	Can scale to handle multi-agent, multi-stage problems
Typical Applications	Chatbots, virtual assistants, workflow helpers	Supply chain coordination, enterprise optimization, virtual team leaders

From Single to Multiple Agents:

- Rather than operating as single units, Agentic AI systems consist of multiple agents, each assigned specialized functions or tasks (such as summarization, retrieval, or planning).
- These agents interact via communication channels like message queues, blackboards, or shared memory.

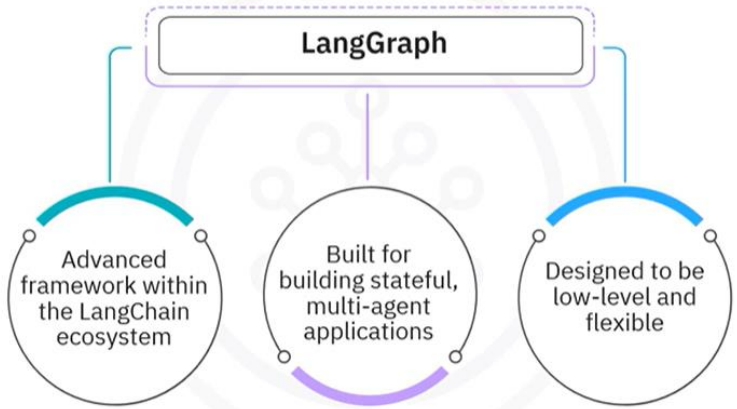
Advanced Reasoning Capabilities

- Agentic systems integrate iterative reasoning capabilities using frameworks such as "ReAct (Reasoning and Acting)", "Chain-of-Thought" prompting, and "Tree of Thoughts." These mechanisms allow agents to break down complex tasks into multiple reasoning stages, evaluate intermediate results, and re-plan actions dynamically.

Persistent Memory Systems

- Unlike traditional agents, Agentic AI incorporates memory subsystems to preserve and persist knowledge across task cycles or agent sessions.
- Memory types include episodic memory (task-specific history), semantic memory (long-term facts or structured data), and vector-based memory for retrieval-augmented generation.

Introduction

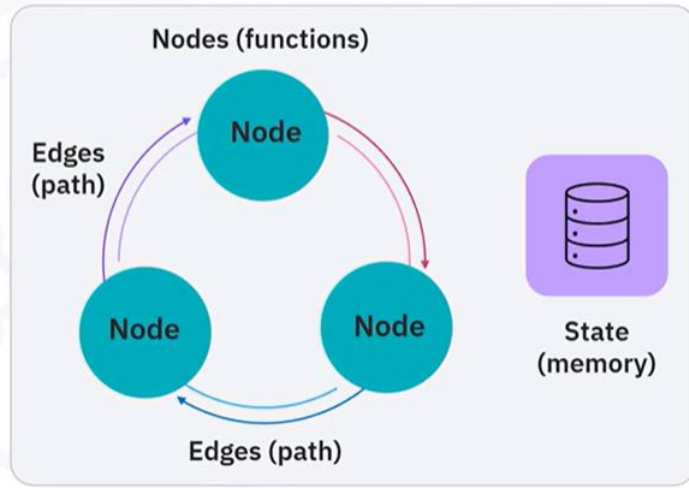


How LangGraph works

Nodes: Individual steps or functions that do the actual computation

Edges: Defines how the execution flows

State: Remembers everything across all the nodes



Powerful capabilities of LangGraph

Looping and branching: Agents can make dynamic decisions as they go

State persistence: AI maintains context over long interactions

Human-in-the-loop: Humans can intervene when needed

Time travel: Facilitates debugging by rewinding to previous states

LangGraph versus Traditional Loops and 'if' statements

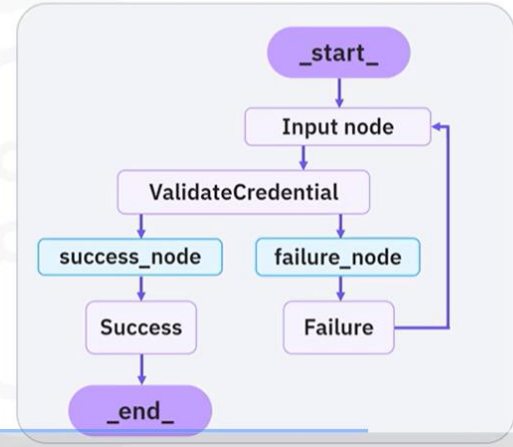
Traditional loops and 'if' statements	versus	LangGraph
• Repeat a block of code until a certain condition is met • Evaluate conditions to decide what happens next • Effective for simple, repetitive tasks • Lack the flexibility needed for complex, stateful workflows		• Explicit state management to maintain context across different nodes • Conditional transitions to make decisions at runtime • Modularity to promote reusable components • Enhanced observability to provide clear insights into the workflow's execution path

Why LangGraph outperforms traditional loops

Traditional loop (while)	LangGraph workflow
Keeps asking until valid input	Can branch, loop, wait for human, then continue
No memory of past topics	Maintains full conversational memory

Visualizing LangGraph workflows

- Mermaid diagrams help debug graph structures
- Primitives (nodes and edges) are clearly represented
- Primitives help construct intricate workflows



Recap

- LangGraph is an advanced framework built for building stateful, multi-agent applications.
- Nodes are functions that do the actual computation. Edges define how the execution flows from one step to the next. State is a shared memory that remembers everything across nodes.
- LangGraph's unique capabilities include looping and branching for making dynamic decisions, state persistence to maintain context over long interactions, human-in-the-loop functionality for timely human interventions, and time travel to facilitate convenient debugging.

Recap

- LangGraph offers state management, allowing the workflow to maintain and modify context across different nodes. It also offers conditional transitions, enabling the workflow to make decisions at runtime and branch accordingly.
- A LangGraph workflow can branch, loop, wait for a human, and then continue later, all while keeping that full conversational memory.
- LangGraph graphs can be visualized using Mermaid diagrams with core primitives such as nodes and edges clearly represented.

Introduction

Recent developments in AI have produced powerful frameworks for building applications around large language models (LLMs). LangChain (released 2022) and LangGraph (released 2023) are two such frameworks from LangChain Inc.

They take different approaches:

- LangChain uses a linear "chain" of components (prompts, models, tools).
- LangGraph uses a graph-based orchestration of stateful, multi-agent workflows.

In practice, LangChain is ideal for straightforward, sequential tasks (for example, a simple QA or RAG pipeline), whereas LangGraph is designed for complex, adaptive systems (for example, coordinating multiple AI agents, maintaining long-term context, or human-in-the-loop approval).

What is LangChain?

LangChain is an open-source framework for developing LLM-driven applications. It provides tools and APIs (in Python and JavaScript) to simplify building chatbots, virtual assistants, Retrieval-Augmented Generation (RAG) pipelines, and other LLM-based workflows. At its core, LangChain uses a "chain" or directed graph structure: you define a sequence of steps (prompts → model calls → outputs) that execute in order.

For example, a RAG workflow might chain: (1) retrieve relevant documents, (2) summarize, (3) generate an answer. The LangChain ecosystem includes prebuilt components for prompts, memory buffers, tools (such as search or calculator), and agents that can pick actions. It also integrates with dozens of LLM providers. LangChain's flexible, modular design has made it popular for quickly prototyping AI apps with minimal coding.

The LangChain framework is layered: a core of abstractions (chat models, vectors, tools), surrounded by higher-level chains and agents that implement workflows.

LangChain's architecture is **modular**. The base `langchain-core` defines interfaces for models, prompts, memory and tools. The main `langchain` package adds chains and agents that form the "cognitive architecture". Popular LLM providers (OpenAI, Google, etc.) have their own integration packages, making it easy to switch models. There is also a `langchain-community` package for third-party extensions. Overall, LangChain serves as a high-level orchestration layer for LLMs and data. It handles inputs/outputs and connects components, but it remains mostly stateless by default (conversation histories can be passed, but long-term memory must be explicitly managed).

What is LangGraph?

LangGraph is a newer framework (built by the same team) focused on stateful multi-agent orchestration. LangGraph is an extension of LangChain aimed at building robust and stateful multi-actor applications with LLMs by modeling steps as edges and nodes. In simple terms, while LangChain chains operations, LangGraph lets you build a graph of agents and tools. Each node in the graph can be an LLM call or an agent (with its own prompt, model, and tools), and edges define how data and control flow between them. This architecture natively supports loops, branches, and complex control flows.

LangGraph is explicitly designed for long-running, complex workflows. Its core benefits include durable execution (agents can pause and resume after failures), explicit human-in-the-loop controls, and persistent memory across sessions. For instance, LangGraph provides "comprehensive memory" that records both short-term reasoning steps and long-term facts. It even offers built-in inspection and rollback features so developers can "time-travel" through an agent's state to debug or adjust its course. In practice, LangGraph is used to build applications where many agents work together – for example, one agent might retrieve documents, another summarizes them, a third plans next steps, etc. These agents communicate using shared "state" (a kind of global memory) and can be arranged hierarchically or in parallel. Major companies (LinkedIn, Uber, Klarna, and so on) have begun using LangGraph to build sophisticated agentic applications in production.

Key Architectural Differences

Feature	LangChain	LangGraph
Type	LLM orchestration framework based on chains and agents.	AI agent orchestration framework based on stateful graphs.
Workflow Structure	Linear/DAG workflows (sequence of steps with no cycles). Good for "prompt → model → output" flows	Graph-based workflows (nodes and edges allow loops, branches, and dynamic transitions). Suited for complex flows.
State Management	Implicit/pass-through data. Chains carry inputs forward, but long-term state is limited by default	Explicit global state ("memory bank") that all agents access. State is persistently stored and updated at each step.
Task Complexity	Best for simple to medium tasks: chatbots, RAG pipelines, sequential reasoning.	Designed for complex, multi-step tasks and workflows that evolve over time (for example, multi-agent assistants).
Agents and Collaboration	Typically single-agent or linear chain; agents operate independently without inter-communication.	Multi-agent. Agents (nodes) can call each other using the graph, share memory, or be arranged hierarchically.

The above table summarizes the LangChain vs. LangGraph trade-offs.

Pros and Cons

Framework	Pros	Cons
LangChain	- Easy and quick to set up for common LLM tasks - Extensive community and prebuilt components (for example, QA chains, map-reduce, memory buffers) - Excellent for RAG workflows and chatbots - Implicit chaining model requires minimal boilerplate code	- Not well-suited for long-running or highly interactive processes - Lacks built-in persistent memory and multi-agent orchestration - Workflows cannot natively loop or branch dynamically - Debugging is harder due to opaque state passing between steps
LangGraph	- Built for complexity and scale - Agents can run concurrently or sequentially with shared context - Supports durable execution (resume from point of failure) - Deep visibility into internal state and execution path (using LangSmith) - Human-in-the-loop support is first class - Ideal for orchestrating multi-step business processes	- More complex to learn and set up - Requires explicit definition of states, nodes, and edges - Slower to develop simple use cases compared to LangChain - Ecosystem is newer with fewer templates and extensions - Overhead may be unnecessary for simple tasks

When to use which framework?

Use Case	Use LangChain When…	Use LangGraph When…
Workflow Complexity	You have a clearly-defined, linear workflow.	You need complex workflows with branching logic or conditional steps.
Development Speed	You want to build something quickly—ideal for prototyping and MVPs.	You're building a production-grade system where reliability, traceability, and durability are essential.
Memory Requirements	Stateless or light memory needs (for example, current conversation only).	Long-term memory is needed across interactions or agents (for example, remembering context across sessions).
Interaction Style	Simple LLM tool use (for example, retrieval, transformation, response).	Multi-turn or human-in-the-loop interactions requiring persistent state and coordination.
System Design	Linear pipelines such as document Q&A, summarization, or format conversion.	Multi-agent architectures, process automation, or workflows with retries, dependencies, or approvals.
Team Collaboration	Individual developer exploring LLM capabilities quickly.	Teams designing modular, orchestrated systems with accountability and version control.

Conclusion

The LangChain and LangGraph frameworks represent two evolving approaches to building with LLMs. LangChain offers a simple, powerful abstraction for chaining prompts and tools in sequence, while LangGraph offers a flexible, stateful architecture for orchestrating complex agent workflows. Developers should choose between them based on the complexity and requirements of their project: use LangChain for straightforward pipelines and experimentation, and adopt LangGraph when you need durable, multi-agent orchestration and fine-grained control. As both frameworks grow, they will likely continue to influence each other. LangChain is already integrating more stateful features (for example, LangGraph memory), and LangGraph can leverage LangChain's components. The right choice depends on the use case at hand, and understanding the trade-offs above will help you select the best tool for your LLM application.

Understand state in LangGraph

- **Counter example:** State includes an integer ('n') and a random letter.
- State is defined with TypedDict
- It can be lists, nested structures, or message sequences

Note: Structures like TypedDict only contain one value per key at a time

n	letter
1	v
2	b
:	p
12	q
13	a

Define state structure with TypedDict

- Class is a subtype of TypedDict

n	letter
1	v

```
import random
import string
from typing import TypedDict

from langgraph.graph import StateGraph, END

class ChainState(TypedDict):
    n: int
    letter: str

initial_state = ChainState(n=1, letter='a')

##### output of code
initial_state:{'n': 1, 'letter': 's'}
```

Function-based nodes

- Nodes link to a function that processes state

```
import random
import string

def add(state: ChainState) -> ChainState:
    random_letter = random.choice(string.ascii_lowercase)
    return {**state,"n": state["n"] + 1,"letter": random_letter,}
```

Function-based nodes

- Create a StateGraph object initialized with the ChainState class

```
from langgraph.graph import StateGraph

workflow = StateGraph(ChainState)
```

Function-based nodes

- Nodes contain logic to process state


```
from langgraph.graph import StateGraph

workflow = StateGraph(ChainState)
```

Function-based nodes

- Edges define transitions between nodes based on conditions


```
from langgraph.graph import StateGraph

workflow = StateGraph(ChainState)
```

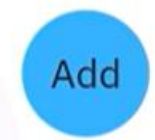
The role of the end state

- End state indicates that the workflow is complete


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

Add nodes to your workflow

- Incorporate the add function using add_node




```
from langgraph.graph import StateGraph

workflow = StateGraph(ChainState)
workflow.add_node("add", add)
```

Connect nodes with edges

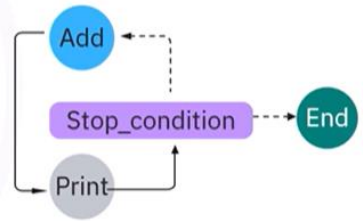
- Add an edge between “add” and “print_out” nodes

```
from langgraph.graph import StateGraph

workflow = StateGraph(ChainState)
workflow.add_node("add", add)
workflow.add_node("print", print_out)
workflow.add_edge("add", "print")
```

Conditional edge implementation

- State has a value of n = 10 and letter = 's'

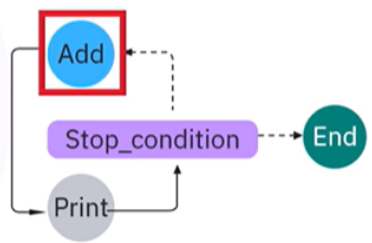
```
def stop_condition(state: ChainState) -> bool:
    return state["n"] >= 13

workflow.add_conditional_edges("print", stop_condition,
{
    True: END,
    False: "add",
})
```



Alternative routing approach

- Route back to the ‘add’ node

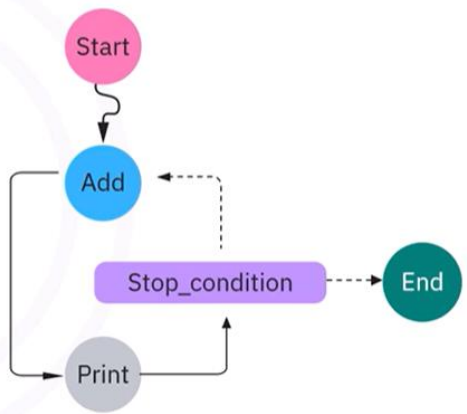
```
def stop_condition(state: ChainState) -> bool:
    if state["n"] >= 13:
        return END
    return "add"

workflow.add_conditional_edges("print", should_continue)
```



Set entry points

- Use the app by passing a starting state

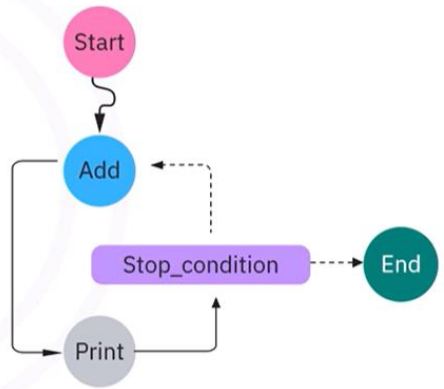
```
workflow.set_entry_point("add")

app = workflow.compile()
```



Invoke the workflow

- Final state in the ‘result’ variable

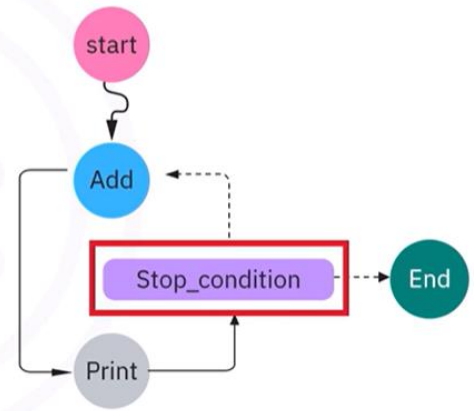
```
result = app.invoke({"n": 1, "letter": ""})

state:{'n': 1, 'letter': ''}
```



Workflow completion

- Condition evaluates to True

```
result = app.invoke({"n": 1, "letter": ""})

state:{'n': 13, 'letter': 's'}

Current n: 2, Current letter: 'v'
Current n: 3, Current letter: 'p'
.
Current n: 12, Current letter: 'g'
Current n: 13, Current letter: 's'
```

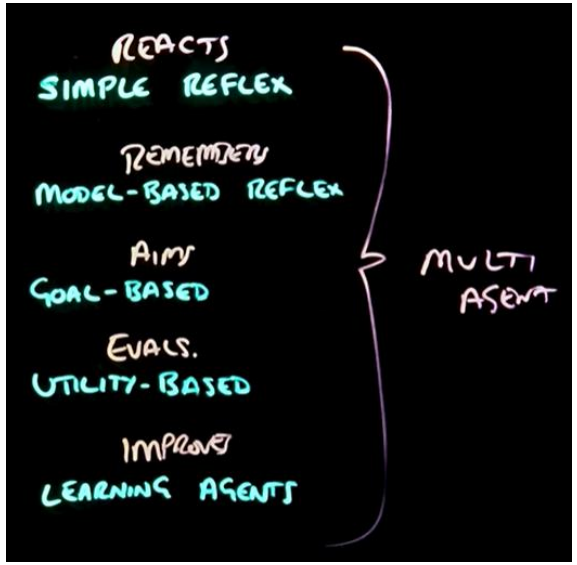


Recap

- State in LangGraph is a constantly changing variable that can store all inputs, intermediate values, and outputs
- Nodes are functions that process the current state. Some nodes modify the state, while others are used for side effects
- Edges define how the execution flows between nodes, passing the updated state from one step to the next
- Conditional edges allow the workflow to make dynamic decisions, routing the state to different nodes

Recap

- Building a LangGraph application involves creating a StateGraph object, incorporating nodes, connecting them, setting an entry point, and then compiling the graph into a runnable application
- Running a LangGraph workflow is done by invoking the compiled application with an initial state
- Workflow visualization helps to understand the execution flow and how state progresses through different nodes



Introduction

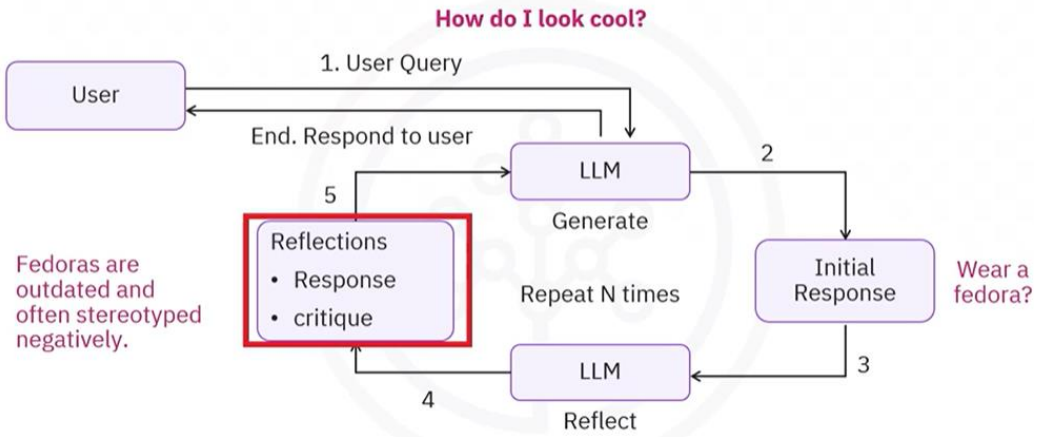
Reflection agents

- Core concept: AI improving by learning from mistakes
- Designed to analyze performance critically

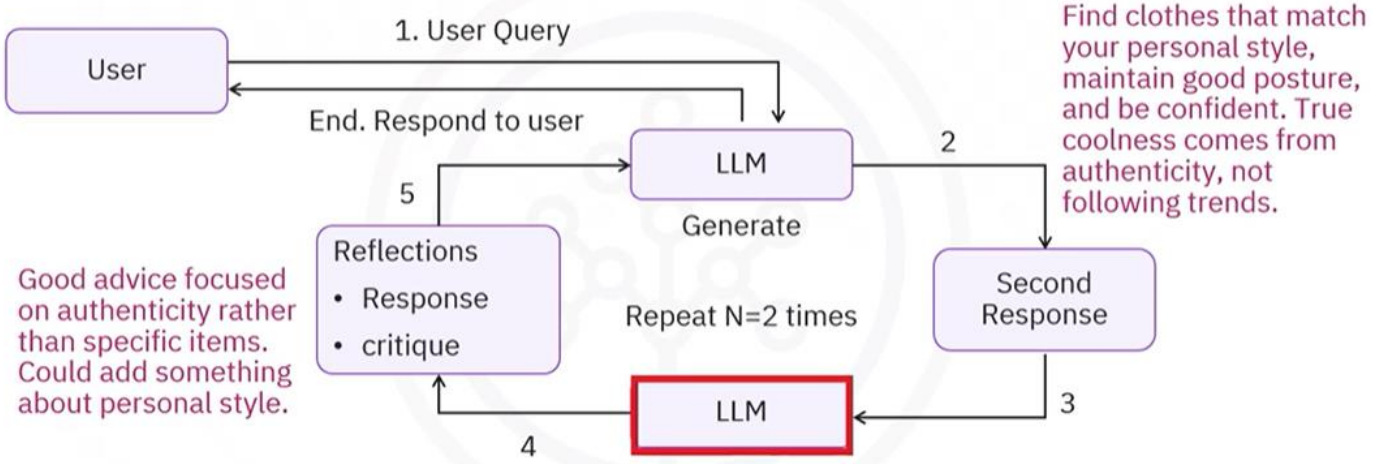
Introduction

- Basic reflection agent
- Reflexion agent
- Language Agent Tree Search or LATS

Example: Basic reflection agent

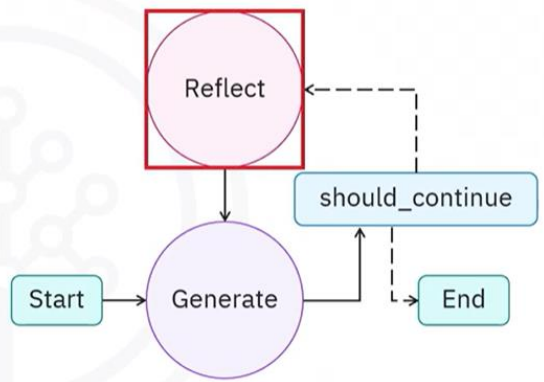


Example: Basic reflection agent

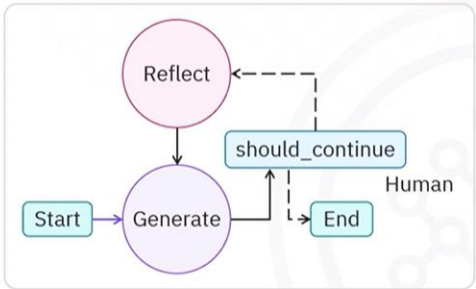


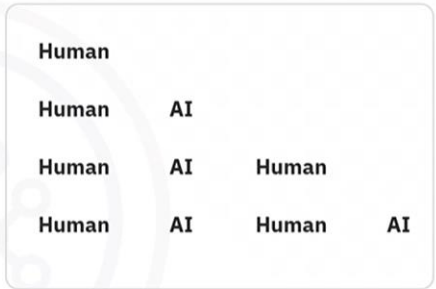
Build a LinkedIn post optimization agent with reflection

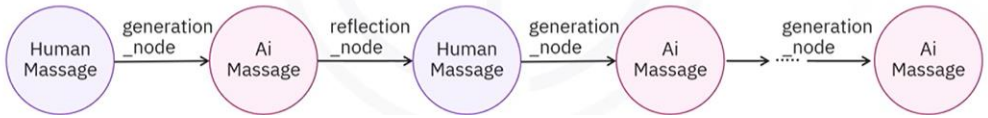
- Build a LinkedIn post optimization agent
- AI will generate a post and critique it
- Refine the content iteratively
- Build with a post generation phase
- Reflection or AI review phase



Build a LinkedIn post optimization agent with reflection







Load the LLM

- Initialize the LLM
- Use IBM's Granite model using LangChain

```
from langchain_ibm import ChatWatsonx

llm = ChatWatsonx(
    model_id="ibm/granite-3-3-8b-instruct",
    url="https://us-south.ml.cloud.ibm.com",
    project_id="skills-network"
)
```

Create the chain for post generation

- Use LangChain's ChatPromptTemplate
- System message defines the LLM's role
- MessagesPlaceholder serves as memory
- Use the pipe operator (|) to connect the structured prompt to the LLM

```
        MessagesPlaceholder(variable_name="messages"),

    ]

)

generate_chain = generation_prompt | llm
```

Create the chain for post generation

- Create a prompt for the reflection LLM
- Define a reflection prompt using ChatPromptTemplate
- System message frames the model as a LinkedIn strategist
- Include a MessagesPlaceholder that injects the post to be critiqued

```
        """
    ),
    MessagesPlaceholder(variable_name="messages")
])

reflect_chain = reflection_prompt | llm
```

Define the agent state

- Use LangGraph to build the conversational workflow
- Define an agent state
- Use prebuilt component MessageGraph
- HumanMessage
- AIMessage
- SystemMessage

```
from langgraph.graph import MessageGraph

# Initialize a predefined MessageGraph

graph = MessageGraph()
```

Define the agent state

- MessageGraph: Specialized StateGraph that accumulates different message types.
- Each user turn adds a HumanMessage
- Then adds an AIMessage

This code is for demonstration purposes only

```
from langchain_core.messages import BaseMessage

class MessagesState(TypedDict):
    messages: List[BaseMessage]

graph = StateGraph(MessagesState)
```

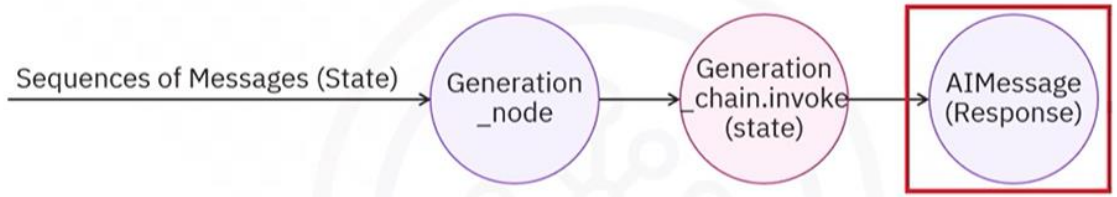
Define the generation node

- Import BaseMessage, HumanMessage, and AIMessage
- Import List and Sequence
- Returns a list-like object of BaseMessages

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

from typing import List, Sequence

def generation_node(state: Sequence[BaseMessage]) -> List[BaseMessage]:
    generated_tweet = generate_chain.invoke({"messages": state})
    return [AIMessage(content=generated_tweet.content)]
```

Define the generation node

"Wear a fedora and sunglasses in your profile pic"




```
def generation_node(state: Sequence[BaseMessage]) -> List[BaseMessage]:
    generated_tweet = generate_chain.invoke({"messages": state})
    return [AIMessage(content=generated_tweet.content)]
```

Define the generation node

- State contains the input message
- Node updates the state, appending the AIMessage
- LangGraph handles this with a merging mechanism

```
    return [AIMessage(content=generated_tweet.content)]

state : [HumanMessage(content="Make me look cool on LinkedIn"),
         AIMessage(content="Wear a fedora and sunglasses in your profile pic")
]
```

Define the reflection node

- reflection_node improves response from generation_node.
- Analyzes the conversation context and provides feedback
- "messages" is a sequence of BaseMessage objects

```
def reflection_node(messages: Sequence[BaseMessage])-> List[BaseMessage]:

    res = reflect_chain.invoke({"messages": messages})

    return [HumanMessage(content=res.content)]
```

Define the reflection node

- "messages" is passed to reflect_chain
- Chain returns a thoughtful critique
- Critique is wrapped in a HumanMessage
- An AIMessage would break the feedback loop
- The reflection agent considers the generation node to be a user
- Requests refinement of the existing input

```
def reflection_node(messages: Sequence[BaseMessage])-> List[BaseMessage]:

    res = reflect_chain.invoke({"messages": messages})

    return [HumanMessage(content=res.content)]
```

Add nodes to the graph

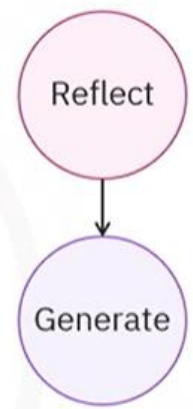
- Add the generate node using add_node
- Function takes two arguments: generate and generation_node
- Similarly, add the reflect node

```
graph.add_node("generate", generation_node)
graph.add_node("reflect", reflection_node)
```

Add edges to the graph

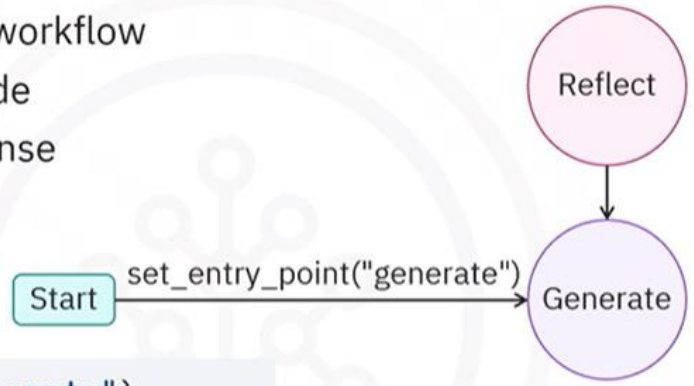
- Define the flow of execution
- add_edge creates a one-way connection
- Creates connection between reflection and generation phases

```
graph.add_edge("reflect", "generate")
```



Add an entry point to the graph

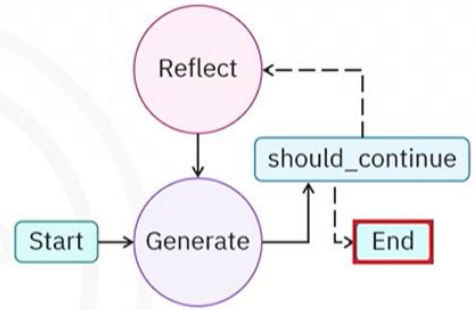
- Set the entry point of the workflow
- Specify the "generate" code
- Start with the initial response




```
graph.set_entry_point("generate")
```

Add the router node for decision-making

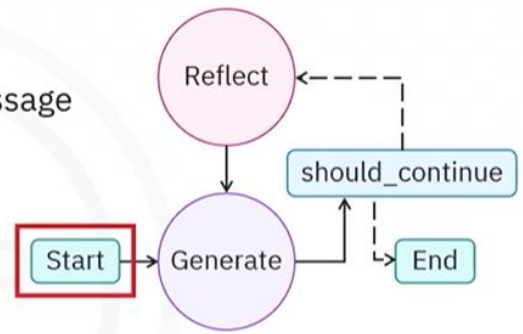
- Add a router node to control the graph's flow
- should_continue function checks the message history
- add_conditional_edges links "generate" to "reflect" or END




```
        return END

    return "reflect"


graph.add_conditional_edges("generate", should_continue)
```

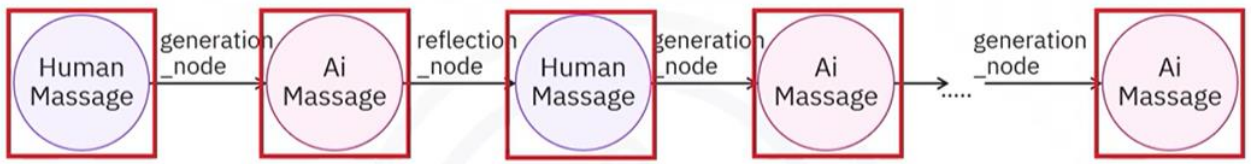
Compile and execute the workflow

- Compile the workflow
- Define the initial user input using a HumanMessage
- Run the workflow




```
inputs = HumanMessage(content="Write a LinkedIn post on getting a software
developer job at IBM under 160 characters")


response = workflow.invoke(inputs)
```

Reflection agent in action



- Final AIMessage is the refined post

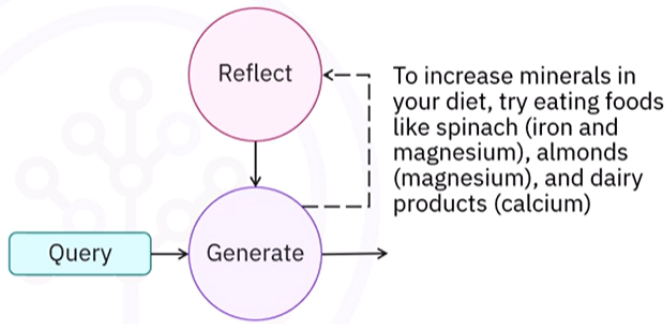
```
    HumanMessage(content="The provided post is..."),  # Critique 1

    . . .

    AIMessage(content="🚀Exhilarated to start @IBM as
Software  Developer...")  # Final post
]
```

Recap

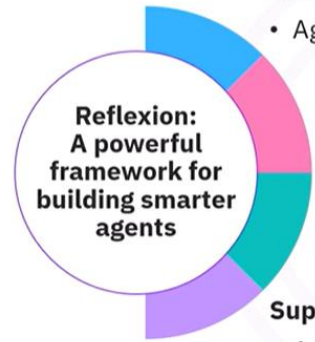
- Reflection agents iteratively improve AI outputs by critically analyzing their performance through a feedback loop
- The generator produces content while the reflector provides critical feedback
- Prompt engineering with LangChain guides LLMs in content generation and structured reflection using dynamic ChatPromptTemplates and message placeholders
- Agent state in LangGraph is defined using MessageGraph. It tracks conversation, accumulating messages and context across iterations
- Graph construction involves defining nodes, connecting them with edges, setting an entry point, and using router nodes for dynamic decision-making and iterative loops.

The reflection process

- The query like “I need more minerals in my diet” is passed through a cycle
- The output is reviewed every iteration
- The system produces a response
- Reflexion allows the system to learn from new information



Why Reflexion agents?



Framework for self-improving agents
- Agents improve with every iteration

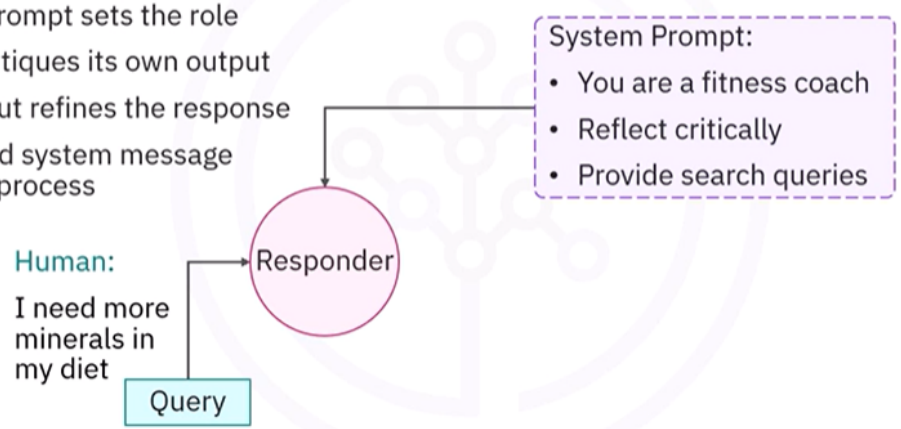
Finds and fixes weaknesses
- Agents review and adjust its reasoning

Incorporates external information
- Agents use web search or APIs

Supports and justifies outputs
- Agents generate responses backed by evidence

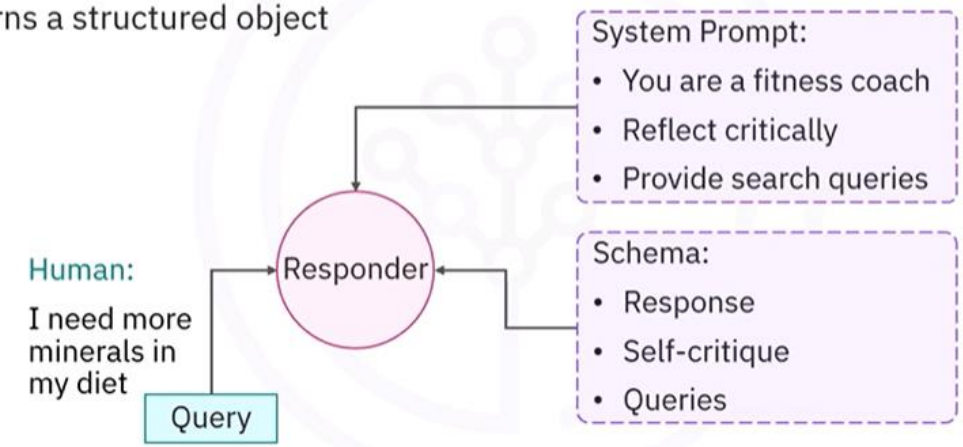
How Reflexion agents work

- Start with a query
- The LLM creates the initial response
- A system prompt sets the role
- The LLM critiques its own output
- A tool output refines the response
- A structured system message guides the process



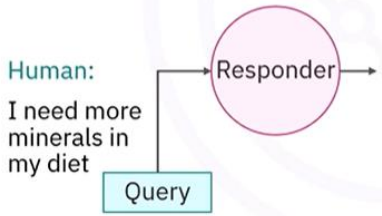
Structuring the responder’s output

- The output is formatted
- Label the response, critique, and query
- LLM returns a structured object



Schema and object representation

- The LLM outputs a structured object
- The object follows a schema
- Each field maps to an object attribute
- The entire structure becomes an AI message



AI:

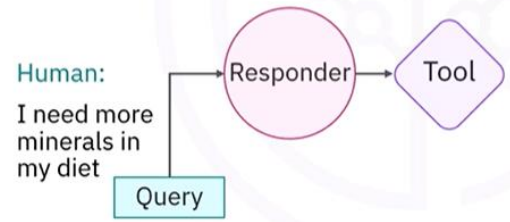
Fields	Response 1
Response	You can get more minerals by eating rocks.
Self-critique	This is misleading — humans can't digest minerals from rocks.
Queries	What are good food sources of minerals? Are there specific deficiencies to address?

Humans can't digest or absorb nutrients from rocks.

This suggestion could be misleading or harmful

Passing output to tools

- The output is passed to a search engine
- The tool extracts the search query
- HumanMessage and AIMessage saved to a list



Fields	Response 1
Response	You can get more minerals by eating rocks.
Self-critique	This is misleading — humans can't digest minerals from rocks.
Queries	What are good food sources of minerals? Are there specific deficiencies to address?

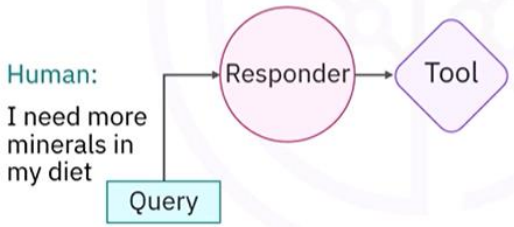
response_list

Human 1	AI 1

Retrieving results from tools

- It might include the title, content, and URL
- Append the tool call result to the response list

Elements	What are good food sources of minerals?
Title	Minerals are Good but don't Eat Rocks!
Content	Minerals are amazing
URL	www.donteatrocks.com

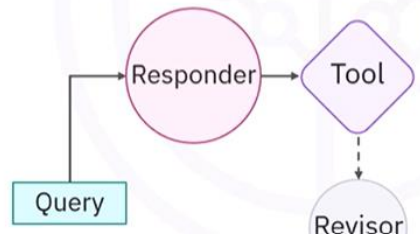


response_list

Human 1	AI 1	Tool 1

Passing results to the revisor

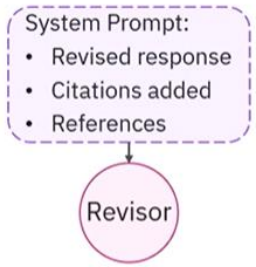
- Tool passes its output to the revisor
- Revisor uses the response list and the self-critique



Field	Response 1
Response	You can get more minerals by eating rocks.
Self-critique	This is misleading — humans can't digest minerals from rocks.
Queries	What are good food sources of minerals? Are there specific deficiencies to address?

Human 1	AI 1	Tool 1

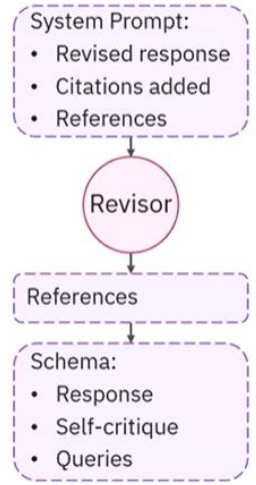
Revisor adds context and citations



Revisor

- Modifies the input from the responder
- Follows a set of instructions to:
  - Revise the response
  - Incorporate citations from the tool
  - Add references for the citations

Build a structured revision with references



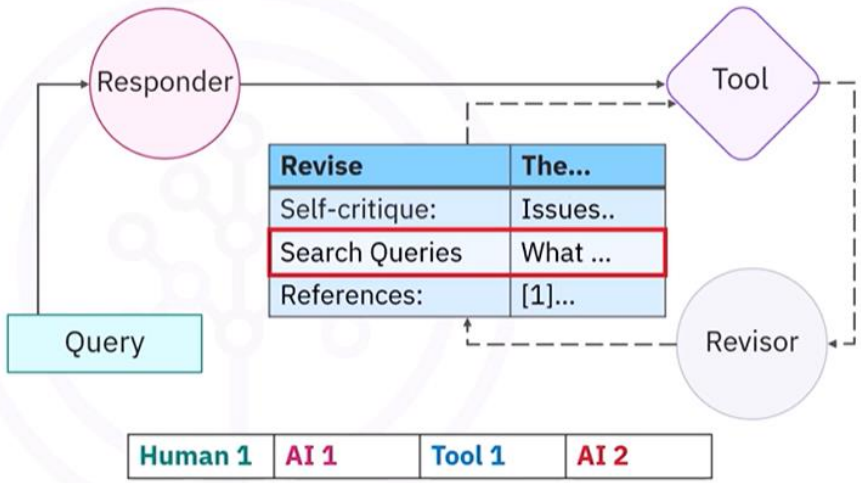
- Response now includes refined references

Revised response	Eat mineral-rich foods: Leafy greens, nuts, dairy products
Self-critique	Corrected dangerous misinformation; needs deficiency specifics
Search Queries:	"daily mineral requirements" "common mineral deficiencies“ "food mineral bioavailability"

References:	Harvard T.H. Chan School of Public Health – National Institutes of Health - Office

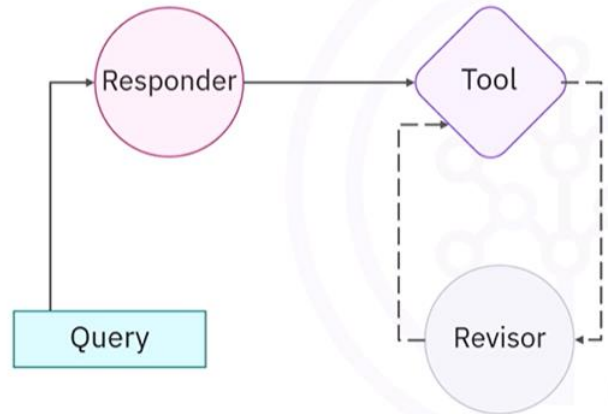
Send queries and store outputs

- Output is passed to the tool
- Output is also stored in the tool response



Tool output added to the response list

- Response processed by the revisor

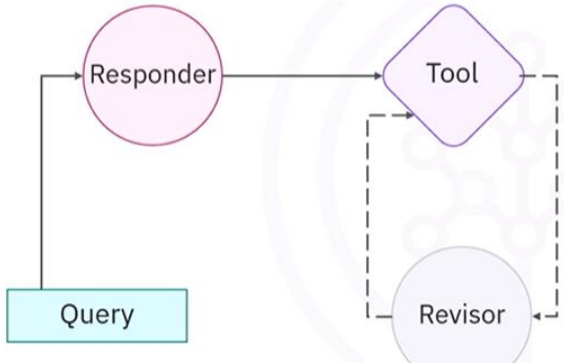


Revise	The...
Self-critique:	Issues..
Search Queries	What ...
References:	[1]...

Human 1	AI 1	Tool 1	AI 2	Tool 2

Track the revision history

- Response includes past revisor AI messages

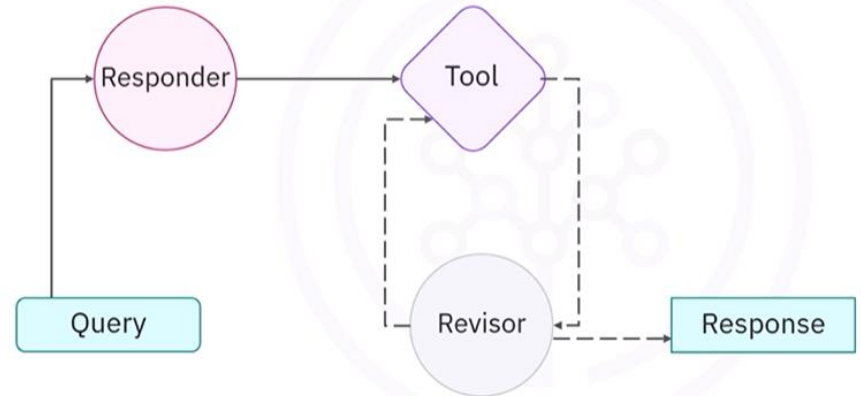


Elements	What are good food sources of minerals?
Title	Next
Content	Medical..
URL	www.D.com

Human 1	AI 1	Tool 1	AI 2	Tool 2

The iterative cycle

- Cycle continues for a predetermined number of iterations



To increase minerals in your diet, try eating foods like spinach (iron and magnesium), almonds (magnesium), and dairy products (calcium).

Recap

- Reflexion agents build on reflection agents by iteratively improving responses
- The reflection process involves a loop of generation, critique, and revision
- Reflexion agents can identify and fix their own weaknesses, improving with each cycle
- They can incorporate real-time data by calling external tools like web search APIs
- Structured schema-based output helps agents distinguish between different components

Recap

- The responder produces an object with fields such as query and response
- The revisor refines the response by revising it, integrating tool outputs, and adding references
- This entire process operates in an iterative cycle, with outputs and feedback passed through tools and stored in a response list

Define the schema

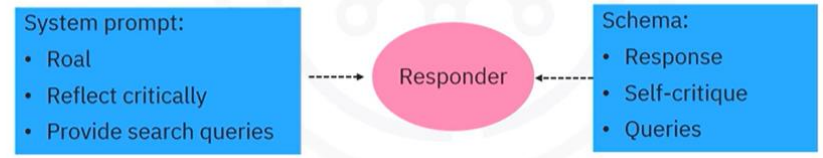
Reflection class is instantiated

```
class Reflection(BaseModel):
    missing: str = Field(description="What information is missing")
    superfluous: str = Field(description="What information is unnecessary")

class AnswerQuestion(BaseModel):
    answer: str = Field(description="Main response to the question")
    reflection: Reflection = Field(description="Self-critique of the answer")
    search_queries: List[str] = Field(description="Queries for additional research")
```

Bind the tool to the LLM

Bind the LLM to the AnswerQuestion schema

```
initial_chain = first_responder_prompt| llm.bind_tools(tools=[AnswerQuestion])
```



Invoke the chain

Result is an AIMessage class

```
response=initial_chain.invoke({"messages":[HumanMessage(question)]})


"""

Breakfast is an excellent opportunity to fuel your body with nutrient-
dense, animal-based foods. A carnivore breakfast can feature nutrient-rich
options
like scrambled eggs cooked in grass-fed butter, which provide essential
amino acids, healthy fats, and fat-soluble vitamins A, D, E, and K. Pairing
the eggs
with liver pâté delivers a concentrated source of nutrients, including
vitamin B12, iron, and retinol, which are often lacking in plant foods. For
a change
of pace, you might consider an omelet filled with various organ meats—such
as beef heart or kidney—which are among the most nutrient-dense foods
```

Invoke the chain

Name of the class of the key-value pairs

```
response.tool_calls:
{
  "tool_call": {
    "name": "AnswerQuestion",
    "args": {

      "answer": "A high-protein all-meat breakfast can serve as an unconventional
but highly effective way to kickstart your metabolism. This includes options like
steak, eggs, or even organ meats such as liver. The rationale for this choice lies
in the thermic effect of food (TEF) and satiety. Protein has a higher TEF compared
to fats and carbohydrates, meaning you burn more calories digesting it, which can
aid in weight management. Additionally, a high-protein breakfast may stabilize blood
```

Store and extract response data

Extract the search queries

```
response_list=[]
response_list.append(HumanMessage(content=question))
response_list.append(response)
```

Human 1

Human 1 | AI 1

```
tool_call=response.tool_calls[0]
search_queries = tool_call["args"].get("search_queries", [])
search_queries:
['effect of antinutrients on nutrient absorption',
 'bioavailability of nutrients in animal foods',
 'evolutionary nutrition guidelines']
```

Create a LangGraph node for search queries

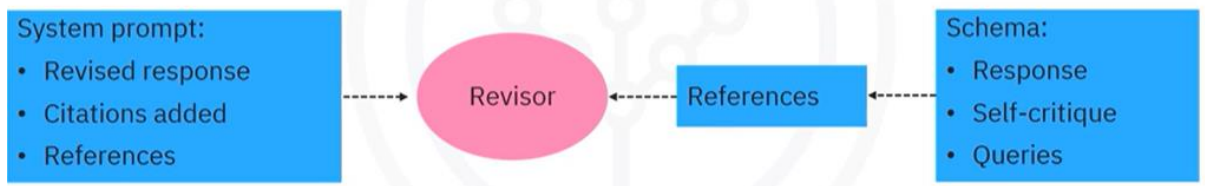
Create a function for search queries

```
import json
from typing import List, Dict, Any
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage

def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
    last_ai_message: AIMessage = state[-1]

    # Extract tool calls from the AI message
    if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
        return []

    # Process the AnswerQuestion or ReviseAnswer tool calls to extract search queries
```

Bind the revisor node

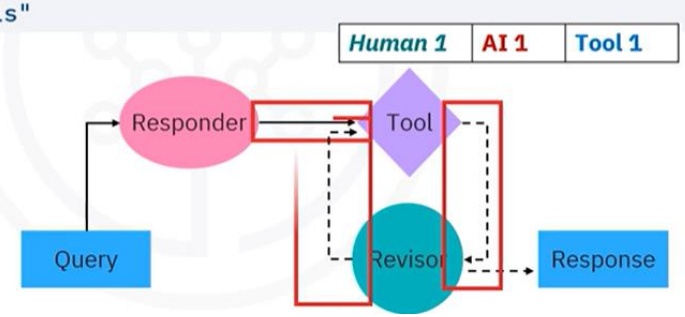
Chain the system message

```
revisor_chain = revisor_prompt | llm.bind_tools(tools=[ReviseAnswer])
```



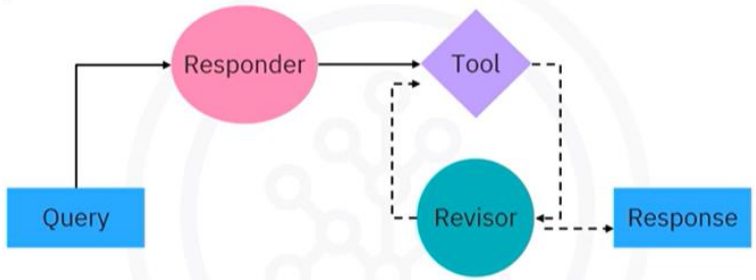
Control iteration count

```
def event_loop(state: List[BaseMessage]) -> str:
  count_tool_visits = sum(isinstance(item, ToolMessage) for item in state)
  num_iterations = count_tool_visits
  if num_iterations >= MAX_ITERATIONS:
      return END
  return "execute_tools"
```



Build the MessageGraph

Set the entry point




```
graph.add_conditional_edges("revisor", event_loop)
graph.set_entry_point("draft")
```

Recap

- A search tool like Tavily can be configured and invoked to provide external data that enhances AI-generated responses
- Prompt engineering and schema design work together to shape the LLM's outputs, enabling structured reflections and targeted answers
- The AnswerQuestion and Reflection schemas define how the agent captures answers, identifies missing or irrelevant details, and generates search queries
- Tool outputs such as tool_calls and schema fields can be parsed to extract structured insights from AI messages
- LangGraph chains responder and revisor nodes to form an iterative feedback loop, using prompt updates and evidence-based revisions
- The complete Reflexion agent is orchestrated through a MessageGraph, which handles node routing, iteration limits, and conditional control flow

Introduction

ReAct agents

- Designed for complex tasks
- Offer step-by-step reasoning

Introduction

Example of a prompt:

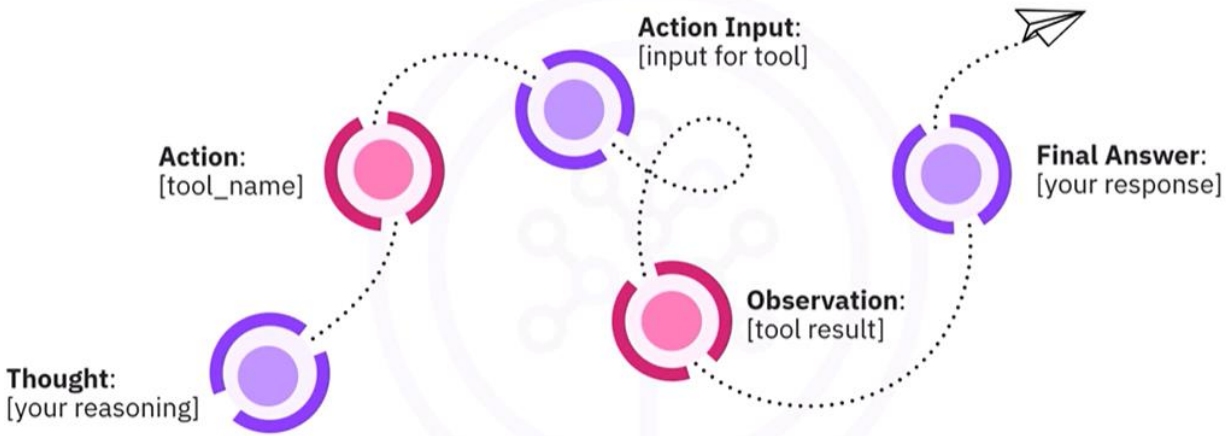
"You are a helpful AI assistant that thinks step by step and uses tools when needed."

Introduction

Prompt the LLM to respond as follows:

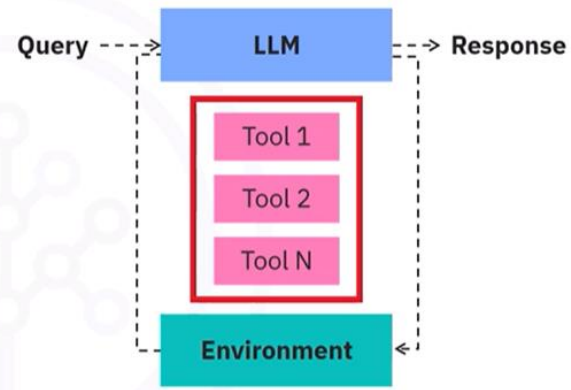
- Identify the information needed
- Use tools when appropriate
- Provide clear and useful answers
- Walk through its reasoning process

Example of a system prompt



Example: Using tools with an LLM

- The LLM is shown in blue
- The green box represents the environment
- The LLM must interact with the environment
- Reasons step-by-step



Example: Using tools with an LLM

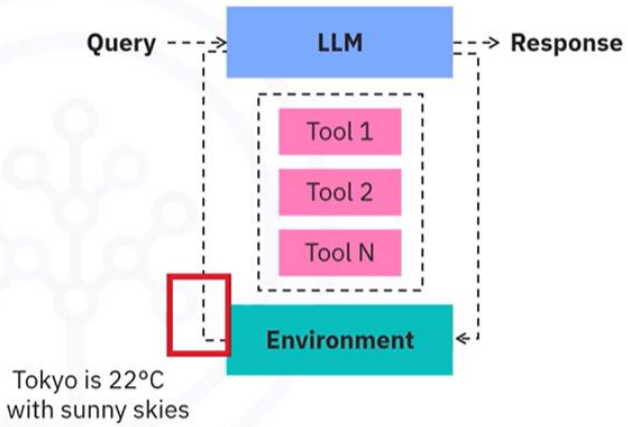
Question: What's the weather in Tokyo, and what should I wear?

Thought: I need to look up the weather in Tokyo first.

Action: search_tool

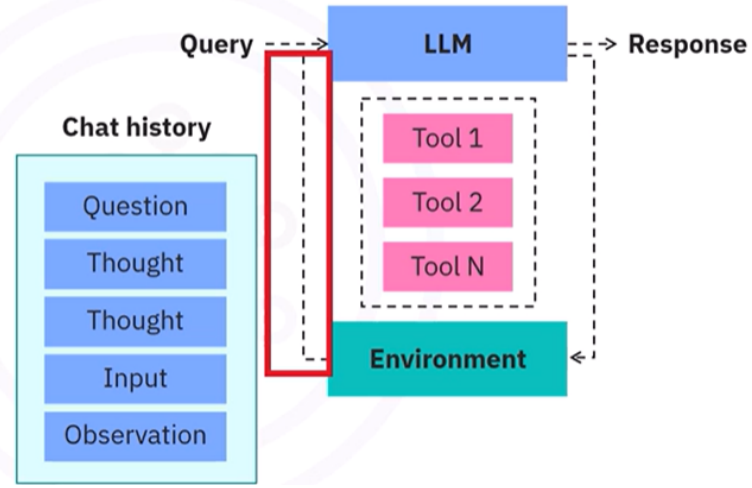
Input: Tokyo weather today

Observation: Tokyo is 22°C with sunny skies



Example: Using tools with an LLM

- Observation and chat history are passed back into the LLM.



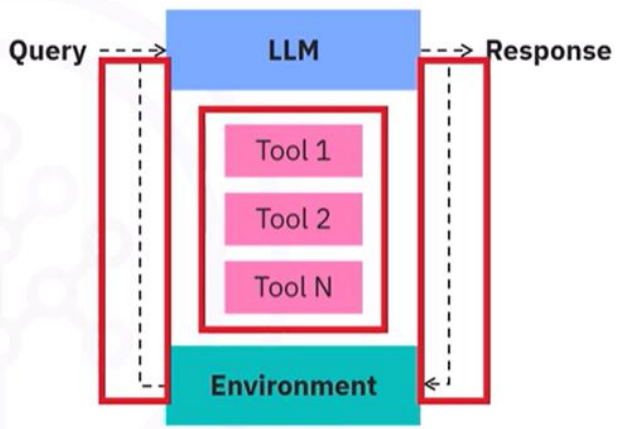
Example: Using tools with an LLM

Thought: Now I should recommend clothing for 22°C sunny weather

Action: recommend_clothing

Action input: 22°C sunny weather

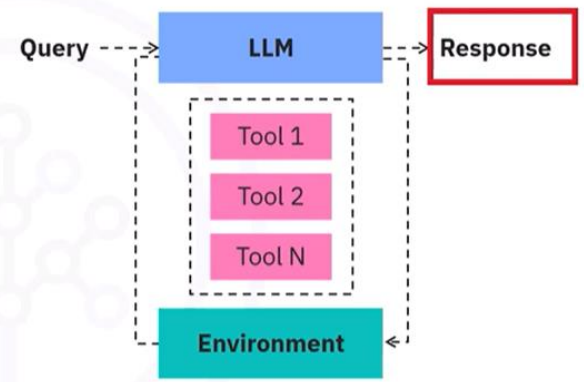
Observation: Light clothing recommended - t-shirt, shorts, sunglasses



Example: Using tools with an LLM

Thought: I now have both weather and clothing recommendation

Final answer: Tokyo is 22°C and sunny today. I recommend wearing light clothing like a t-shirt, shorts, and sunglasses.

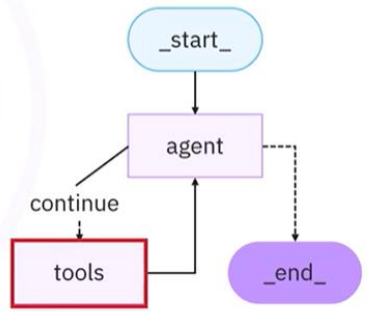


ReAct with LangGraph

Create a new graph using AgentState

- Add a node named “tools”

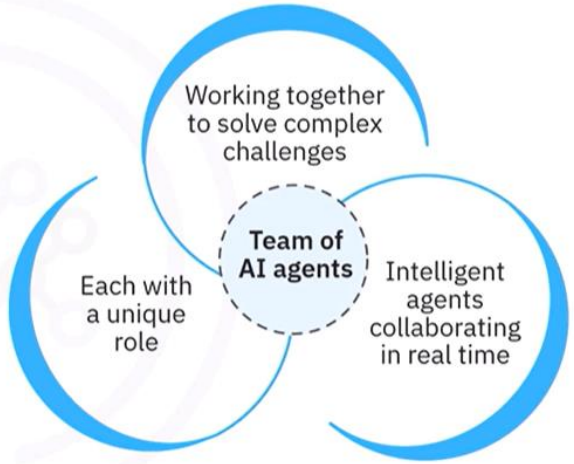
```
from langgraph.graph import StateGraph, END

# 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)

# Add edges between nodes
workflow.add_edge("tools", "agent")
```



Recap

- ReAct agents perform step-by-step reasoning and use tools to answer complex queries.
- Their output follows a structured format: Thought → Action → Action Input → Observation → Final Answer.
- Tool results (observations) feed back into the reasoning process to guide next steps.
- LangGraph is used to implement the ReAct workflow, connecting reasoning and tool nodes.
- The process continues until a Final Answer is generated, with no further tool calls required.

Introduction



What are multi-agent systems?

The core idea

- Organize agents into specialized, collaborate teams
- Focus on organized specialization
- Assign the right agent to the right task

Multi-Agent System

- Consists of multiple autonomous agents
- Interacts within an environment to achieve goals
- Each agent operates independently

Core components of multi-agent systems

Fleet of warehouse robots

- Constantly update each other
- Signals others to avoid redundant effort
- Have dynamic, real-time coordination

Core components of multi-agent systems

Components

Agents: Autonomous units with specific capabilities and goals

Environment: Context within which agents operate and interact

Communication protocols: Standards that enable agents to share information

Agent specialization concepts

Capability boundaries
Each agent has a well-defined scope

Expertise depth vs. Breadth
Balance deep specialist agents with broad generalist coordinators

Interface standardization
Agents communicate through structured inputs and outputs

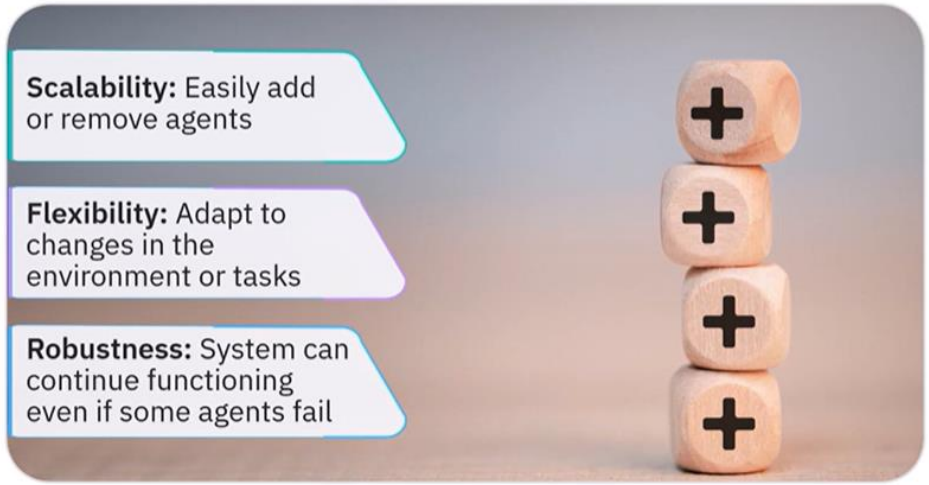
Handoff patterns
Agents pass tasks when outside their expertise

An example of a multi-agent system

Research Assistant System:

1. **Retriever Agent:** Pulls relevant documents from various sources
2. **Summarizer Agent:** Condenses documents and extracts key insights
3. **Critique Agent:** Evaluates the summarized agent
4. **Compiler Agent:** Generates the final report

Advantages of multi-agent systems



Scalability: Easily add or remove agents

Flexibility: Adapt to changes in the environment or tasks

Robustness: System can continue functioning even if some agents fail

How agents work

- Agents interact in Graph-Structured Systems
- **Pipeline patterns:** Agents perform sequential handoffs
  - Example: Research Agent sends data to Editor Agent
- **Hub-and-Spoke Pattern:** Central coordinator dispatches tasks to specialists
  - Example: Content Manager assigns tasks to Writer, Fact-Checker, SEO Optimizer
- Many more powerful interaction patterns exist

Communication protocols in multi-agent systems

Model Context Protocol (MCP)

- Standardizes how AI models access and share context
- Acts like a universal connector for AI applications

Agent Communication Protocol (ACP)

- Provides a standardized communication method for AI agents
- Enables seamless integration across different systems

Orchestrating Multi-Agent systems

LangGraph

Framework that helps define custom multi-agent workflows

CrewAI

Open-source Python framework designed to develop and manage multi-agent AI systems

AutoGen

Microsoft-developed framework that enables the creation of multi-agent AI applications through conversational interfaces

IBM BeeAI

Open-source framework for building and deploying AI agents

Challenges and considerations

Coordination complexity:

Ensuring agents work harmoniously

Communication overhead:

Frequent interactions may strain resources

Security concerns:

Protecting the system from malicious agents

Recap

- Multi-agent systems focus on assigning the right agent to the right task. These systems consist of multiple autonomous agents.
- Orchestration frameworks such as LangGraph, CrewAI, AutoGen, and IBM BeeAI Framework are used to manage complex interactions among AI agents.
- Model Context Protocol (MCP): Standardizes how AI models access and share context with external tools and data sources.
- Agent Communication Protocol (ACP): Provides a standardized method for AI agents to communicate and collaborate.
- Challenges in building Multi-Agent Systems include coordination complexity, communication overhead, and security concerns.