Contents

[AGENTIC AI 2](#_Toc206270748)

[TRADITIONAL VERSUS AGENTIC AI 2](#_Toc206270749)

[EXAMPLES 2](#_Toc206270750)

[DESIGN PATTERNS FOR AGENTIC AI 3](#_Toc206270751)

[REFLECTION PATTERN 3](#_Toc206270752)

[TOOL USE PATTERN 4](#_Toc206270753)

[PLANNING PATTERN 4](#_Toc206270754)

[MULTI-AGENT PATTERN 4](#_Toc206270755)

[ReACT PATTERN IN AGENTIC AI 5](#_Toc206270756)

[STEPS OF ReAct 5](#_Toc206270757)

[CORE PILLARS OF AGENTIC AI 6](#_Toc206270758)

[EXAMPLE 7](#_Toc206270759)

[LANGRAPH 8](#_Toc206270760)

[WHY LANGRAPH? 8](#_Toc206270761)

[KEY COMPONENT OF LANGRAPH 8](#_Toc206270762)

[STATEGRAPH AND SCHEMA 8](#_Toc206270763)

[REDUCER 8](#_Toc206270764)

[TOOL NODES 8](#_Toc206270765)

[EDGES 8](#_Toc206270766)

[IMPLEMENTING TOOLS & TOOL CALLING IN LLMS FOR AGENTIC AI WITH LANGCHAIN 8](#_Toc206270767)

[TOOLS FOR AGENTIC AI SYSTEMS 8](#_Toc206270768)

[EXAMPLE 9](#_Toc206270769)

[DETAILS 10](#_Toc206270770)

[CODE 12](#_Toc206270771)

[BUILD A TOOL-USE REACT AGENTIC AI SYSTEM WITH LANGCHAIN AND LANGRAPH 17](#_Toc206270772)

[OBJECTIVE 18](#_Toc206270773)

# AGENTIC AI

A diagram of a tool

AI-generated content may be incorrect.

* **Agentic AI** refers to AI systems that can **autonomously make decisions**, **take actions**, and **pursue goals** with minimal human intervention.
* These systems behave like **agents** — they don’t just respond to prompts, they **act independently** based on objectives, rules, and environmental feedback.

The distinction between **Traditional AI** and **Agentic AI** reflects a major shift in how artificial intelligence systems are designed, interact, and operate. Here's a breakdown:

## TRADITIONAL VERSUS AGENTIC AI

Traditional AI

* **Task-specific**: Designed for narrow, well-defined tasks (e.g., image recognition, spam filtering).
* **Static logic**: Operates based on pre-programmed rules or trained models.
* **No autonomy**: Executes tasks without decision-making or goal-setting.
* **Pipeline-based**: Often part of a larger system, but not self-directed.

**Examples:**

* Machine learning models for fraud detection.
* NLP models for sentiment analysis.
* Recommendation engines.

Agentic AI

* **Goal-driven**: Operates with objectives and can plan steps to achieve them.
* **Autonomous**: Makes decisions, adapts to new information, and can self-correct.
* **Tool-using**: Can call APIs, search the web, write code, and interact with environments.
* **Memory-enabled**: Remembers past interactions and uses them to improve performance.
* **Multi-modal**: Can handle text, images, audio, and more in a unified workflow.

**Examples:**

* AI agents built with **LangGraph**, **AutoGPT**, or **OpenAI's Assistants API**.
* Agents that research, write reports, book meetings, or debug code.
* Agents that collaborate with other agents or humans in real-time.

## EXAMPLES

|  |
| --- |
| AI Health Assistant (Personal Agent)  Scenario: A patient says: "Help me manage my diabetes."  *Agentic AI Actions:*   1. Retrieves your medical history and prescriptions. 2. Sets reminders for medication and meals. 3. Orders refills from the pharmacy. 4. Alerts your doctor if blood sugar readings are abnormal. 5. Adjusts your diet plan based on recent activity.   ✅ Autonomous, goal-driven, multi-step behavior. |
| AI Executive Assistant (e.g., AutoGPT)  Scenario:You say: "Plan a business trip to Mumbai next week."  *Agentic AI Actions:*   1. Searches for flights and books the best one. 2. Reserves a hotel near your meeting location. 3. Schedules meetings with clients. 4. Adds everything to your calendar. 5. Sends confirmation emails.   ✅ It acts like a human assistant, not just a chatbot. |
| 3. DevOps Agent  Scenario: "Monitor my app and scale it if traffic spikes."  *Agentic AI Actions:*   * Monitors server metrics. * Detects a traffic spike. * Automatically scales up resources. * Sends a Slack alert to the team. * Rolls back if errors increase.   ✅ Autonomous infrastructure management. |
| E-commerce Store Manager Bot  Scenario:"Manage my online store."  *Agentic AI Actions:*   1. Updates product listings. 2. Adjusts prices based on competitor data. 3. Responds to customer queries. 4. Flags suspicious orders. 5. Launches a weekend sale campaign.   ✅ Acts with initiative and adapts to changing conditions. |

## DESIGN PATTERNS FOR AGENTIC AI

A diagram of a design

AI-generated content may be incorrect.

### REFLECTION PATTERN

* **What it is**: The agent thinks about what it did, checks for mistakes, and tries to improve i.e. Agents analyze to refine strategies.
* **Real-world analogy**: Like a student reviewing their test answers and correcting errors before submitting.
* **AI example**: You ask an AI to summarize a document. It gives a summary, then reflects: “Did I miss anything important?” It re-reads the document and updates the summary to include missing points.
* **Why it’s useful**: It helps the agent become more accurate , reduces repetitive mistakes and reliable over time.
* **Core Components** 
  + **Memory Module**
  + **Evaluator**
  + **Feedback Loop**

### TOOL USE PATTERN

* **What it is**: The agent selects & uses external tools to get things done. i.e it can connect to the live data
* **Real-world analogy**: Like a person using a calculator, Google Maps, or a dictionary to help with tasks.
* **AI example**: You ask, “What’s the weather in Noida?” The agent doesn’t guess—it uses a weather API to fetch the latest data and gives you the answer.
* **Why it’s useful**: It allows the agent to go beyond its built-in knowledge and access real-time or specialized information.
* **Core Components**
  + **API Connectors**
  + **Data Validators**

### PLANNING PATTERN

* **What it is**: The agent breaks a big task into smaller steps and follows a plan.
* **Real-world analogy**: Like planning a trip—first book flights, then hotels, then create an itinerary.
* **AI example**: You ask an AI to write a research paper. It first creates an outline, then writes each section step-by-step, checking progress along the way.
* **Why it’s useful**: It helps the agent stay organized and handle complex tasks more effectively.

### MULTI-AGENT PATTERN

* **What it is**: Multiple agents work together, each doing a specific job.
* **Real-world analogy**: Like a team project—one person does research, another writes, another edits.
* **AI example**: One agent specializes in medical knowledge, another in summarizing text, and another in generating visuals. You ask for a medical infographic, and they collaborate to produce it.
* **Why it’s useful**: It allows for specialization and teamwork, making the system more powerful and flexible.

How These Patterns Work Together – Home Health System



1. Planning Pattern:
   1. Schedule a video call with the doctor
2. Tool Use Pattern:
   1. Book an appointment with the doctor via clinics API
3. Reflection Pattern:
   1. Meanwhile – the reflection agent Analyze the call outcome to improve future scheduling
4. MultiAgent Pattern:
   1. Transfer the post appointment notes to a diet plan agent (another AI agent)

## ReACT PATTERN IN AGENTIC AI

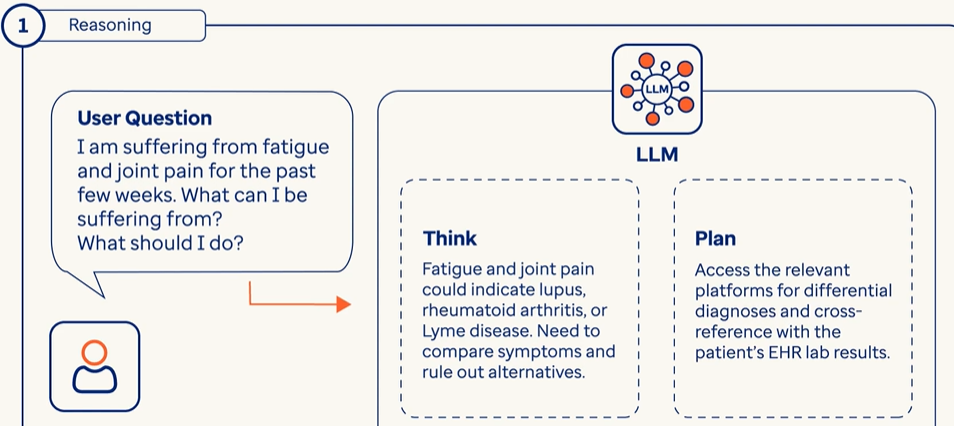
* The ReAct pattern is inspired by the problem solving approach of Humans. For example, A medical doctor diagnosing a patient. The doctor think through the symptoms, observer test results and act prescribes treatment.
* This iterative process of analyzing data , adopting to new information and execute decisions led the foundation of ReAct(Reasoning & Acting Framework) for Agentic AI

|  |  |
| --- | --- |
|  | * Similar to humans AI agent using ReAct will   + Decompose the complex tasks into multiple structured steps   + Integrate real time feedback   + Interact with tools to achieve goals |

### STEPS OF ReAct

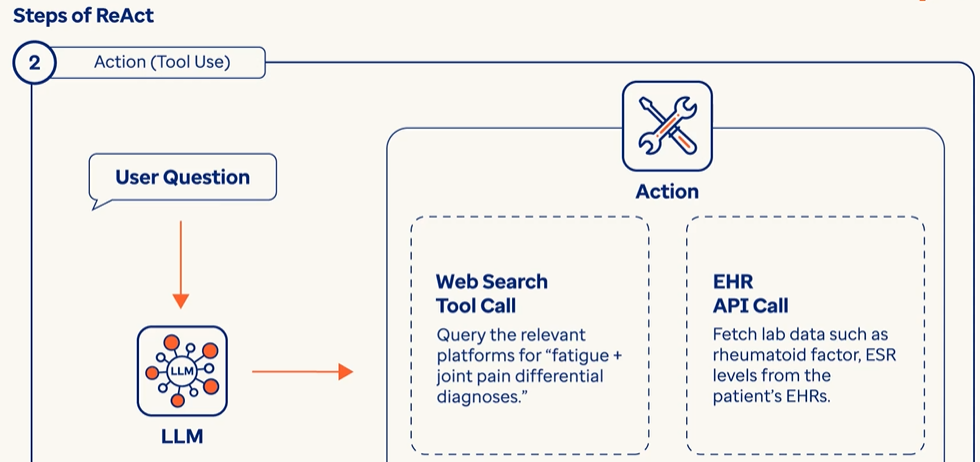
Let us understand the steps of React with an example

#### REASONING



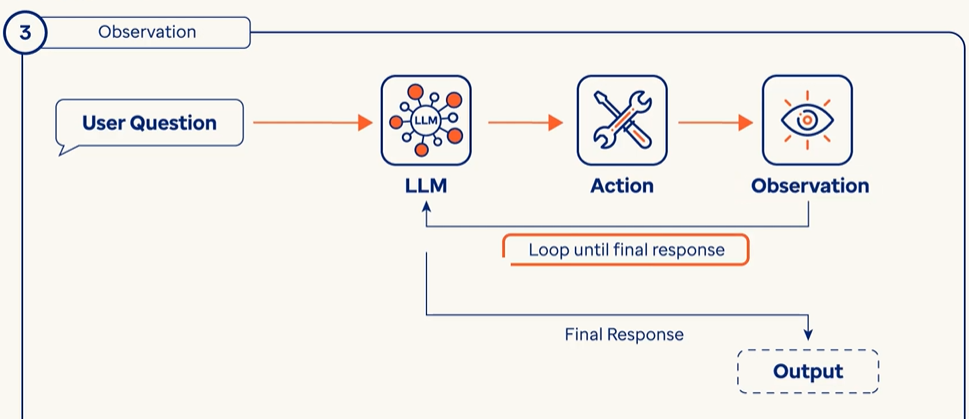
1. Let the user as questions on his medical problem. The LLM will first Think followed by planning.

#### ACTION



* In this step for the user’s query LLM will make use of tolls (tool calling) for web search and tool to fetch Electronic health Record

#### REASONING



Note :

* React uses Chain of thought prompting which enables LLMs to reason better
* Tools & Function calling which enables to take better action

### CORE PILLARS OF AGENTIC AI

Planning

* The agent can **break down goals into steps**, decide what to do next, and adapt its strategy.
* Often uses **chain-of-thought prompting**, **decision trees**, or **LangGraph-style workflows**.
* Enables **autonomous task execution** and multi-step reasoning.

Tools

* Agents can **call external APIs**, **search the web**, **write code**, or **interact with databases**.
* Tools extend the agent’s capabilities beyond language—into action.
* Examples: Tavily search, Python execution, SQL queries, file handling.

Memory

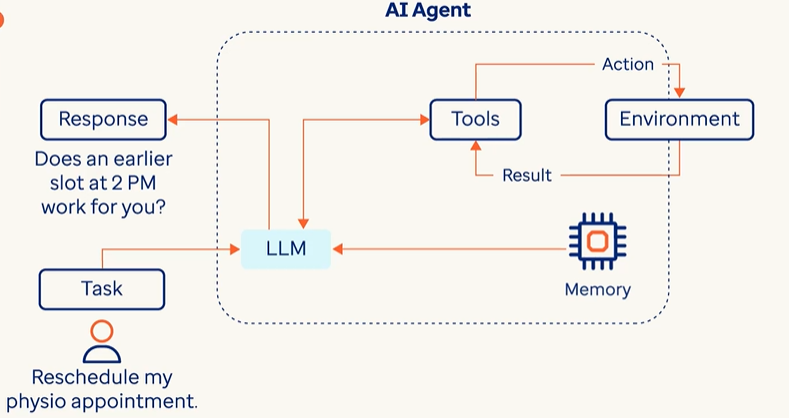
* Agents remember **past interactions**, **user preferences**, and **task history**.
* Supports **context continuity**, personalization, and long-term learning.
* Can be short-term (within a session) or long-term (across sessions).

How They Work Together

Imagine an agent tasked with booking a medical appointment:

* **Planning**: Breaks the task into steps—check history, find slots, confirm preference.
* **Tools**: Uses APIs to access EHR, calendar, and notification systems.
* **Memory**: Remembers patient preferences and past interactions.

### EXAMPLE



Task Initiation (Planning Begins)

* **Input**: *"Reschedule my physio appointment.".* This is the **goal** the agent receives.
* The **LLM** begins **planning** how to achieve this—breaking it into steps like checking availability, confirming preferences, and updating records.

Tool Usage

* The LLM interacts with **Tools** to:
  1. Query the calendar system.
  2. Check provider availability.
  3. Suggest alternate slots (e.g., *"Does an earlier slot at 2 PM work for you?"*).
  4. These tools act on the **Environment**—real-world systems like EHR(Electronic Health Record), scheduling APIs, or notification services.

|  |
| --- |
| * Vector DB and Python list stores this data enabling instant retrieval of data during decision loops |

Memory Integration

After tools act on the environment, the results (e.g., available slots, patient preferences) are stored in **Memory**.

This allows the agent to:

* Remember past appointments.
* Learn user preferences (e.g., prefers afternoon slots).
* Avoid redundant or conflicting actions.

Response Generation

* The LLM uses updated context from **Tools** and **Memory** to generate a meaningful response:
* *"Does an earlier slot at 2 PM work for you?"*
* This closes the loop, completing the agentic cycle.

# LANGRAPH

Langgraph is the python library to build stateful , graph based workflow for Agentic AI systems

## WHY LANGRAPH?

* **Controllability** allows developers to define when and how agents act – *e.g., a medication tracker escalates only after three missed doses.*
* **Human-in-the-Loop** enables seamless oversight – *e.g., a caregiver reviews an AI-proposed treatment plan before execution.*
* **Streaming-First** delivers real-time updates – *e.g., live heart rate trends without critical lag.*

## KEY COMPONENT OF LANGRAPH

* STATEGRAPH AND SCHEMA
* REDUCER
* TOOL NODES
* EDGES

### STATEGRAPH AND SCHEMA

|  |  |
| --- | --- |
| * This defines the graph’s overall state including the schema -variables and datatypes, shared across nodes during agent execution * The state schema the overall schema of data variables shared across the agent ‘s lifecycle. It can be any python data type typically TypeDict or Pydantic base Model |  |

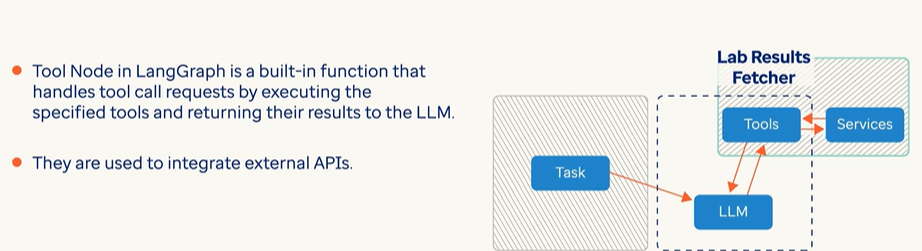
### REDUCER

* Merge updates into the state
* By default, when state variable is updated in multiple nodes , it overrides the previous value , but using reducer we can add/merge the new value to existing state variables
* The reducer function defines how state can be applied to specific keys within the state.

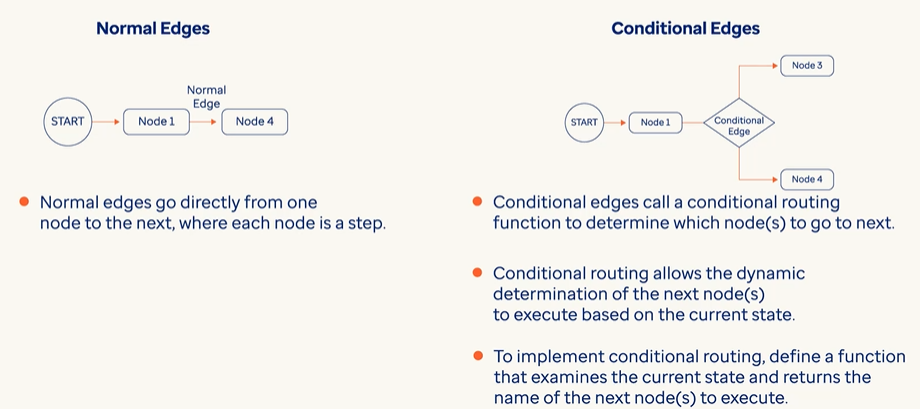
|  |  |
| --- | --- |
| Reducer | Functionality |
| add\_message | * Appends new messages to the existing list, maintaining the conversation history * Additionally, it manages message IDs to prevent duplication and allows for the removal of specific message when needed. * To use , define the state schema and annotate the message key. Then associate the function with node using **add\_conditional\_edges** method |

#### EXAMPLE

### TOOL NODES



### EDGES



# IMPLEMENTING TOOLS & TOOL CALLING IN LLMS FOR AGENTIC AI WITH LANGCHAIN

## TOOLS FOR AGENTIC AI SYSTEMS

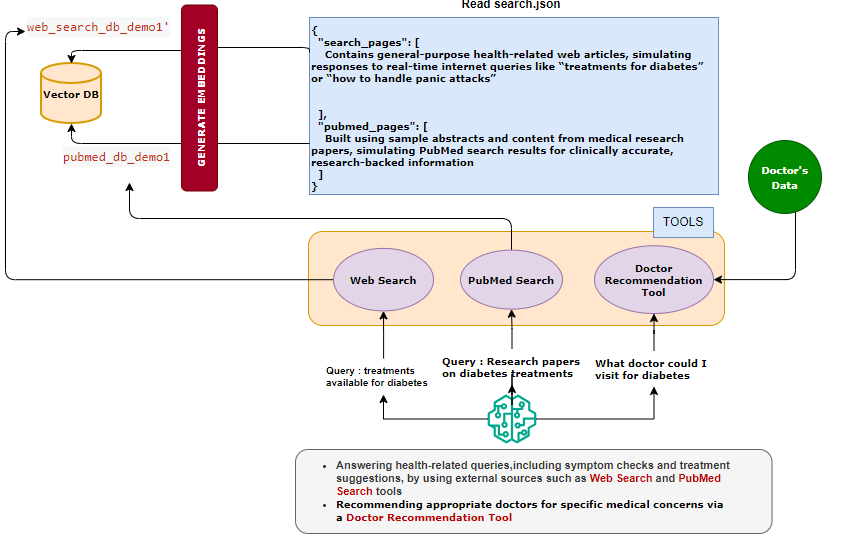
* In Agentic AI systems, tools are functions that allow the large language model (LLM) to do things it cannot do on its own. These include actions like searching for information, looking up data from a database, performing calculations, or calling APIs.
* Tool calling allows the LLM to request a tool when it needs help answering a question. In LangChain, this is done using the **bind\_tools** method. This tells the model which tools are available, how they work, and when to use them when input queries are passed to the LLM.
* LanGraph can also be leveraged to create reactive agents by passing tools, prompts, and LLMs to built-in functions.
* **Note that the LLM will not call the tools automatically; they must be executed manually**

## EXAMPLE

|  |
| --- |
| We aim to build **HealthBuddy**, an Agentic AI system designed to assist users with health-related queries through intelligent reasoning and tool use. Leveraging the **ReAct framework** (Reasoning + Acting), HealthBuddy integrates a **Large Language Model (LLM)** with custom tools and structured instructions to provide accurate, actionable, and personalized responses.  HealthBuddy is capable of:   * Answering health-related queries, including symptom checks and treatment suggestions, by using external sources such as **Web Search** and **PubMed Search** tools * Recommending appropriate doctors for specific medical concerns via a **Doctor Recommendation Tool** * Displaying available **appointment slots** and **booking appointments** after collecting essential user details (name, email, phone) * Maintaining **multi-turn conversations** in a natural, chat-like flow * Supporting **multi-user interactions** with isolated memory contexts for each user to enable personalized and continuous engagement |

A diagram of a computer

AI-generated content may be incorrect.



### DETAILS

STEP 1: Prepare Databases for Simulated Web Search and PubMed Search Tools

* We will load sample data simulating real web page documents and research papers from PubMed. This simulates real-time web or research paper searches using preloaded documents as the source.
* We will create two separate vector databases:
  + web\_search\_db: Contains general-purpose health-related web articles, simulating responses to real-time internet queries like “treatments for diabetes” or “how to handle panic attacks”
  + pubmed\_db: Built using sample abstracts and content from medical research papers, simulating PubMed search results for clinically accurate, research-backed information

Step 2: Preparing Database for Doctor Recommendations Tool

* We will create a small, in-memory database containing information about doctors. This data will be used by our Doctor Recommendation Tool to help users find the right doctor based on their health query or symptoms.
* The database includes a list of doctors along with their:
  + Name
  + Specialization (e.g., Dermatology, Pediatrics, Cardiology)
  + Location
  + Availability
  + Contact information
* We will use a simple Python list of dictionaries to store the doctor data. In a real-world application, this would typically be replaced by a backend database like PostgreSQL, MongoDB, or an external API.

Step 3: Create Tools for AI Agent

* LangChain makes it easy to create and register tools using the Tool class. The tool includes:
* A name and description
* The python function to be called
* An input schema that tells the model what arguments it can use
* When tools are properly defined, they enable the model to solve more complex problems by allowing it to perform actions and access external data. This makes the system more useful and reliable.
* These tools will allow the agent to retrieve information from our preloaded vector databases (web search and PubMed), as well as recommend doctors from our in-memory doctor database.
* The goal is to modularize the logic for different types of tasks into reusable components that can be invoked by the LLM when needed. These include:
  + A **Web Search Tool** that queries the web\_search\_db to simulate general online web search
  + A **PubMed Search Tool** that retrieves information from pubmed\_db for research-grade medical content
  + A **Doctor Recommendation Tool** that finds suitable doctors based on user symptoms or needs
* This tool-based setup is essential for enabling agentic behavior, where the LLM reasons through a problem, decides which action to take, and requests to call the right tools to gather more information or perform a task.

|  |  |
| --- | --- |
| how the LLM is deciding which tool to call in code?   * Tool Binding and Function Calling: The **bind\_tools** method converts the tools into OpenAI function specifications. When we bind tools to the LLM, each tool's metadata (name, description, parameters) becomes available to the model as function specifications. * Tool Descriptions: Each tool has **a descriptive docstring that helps the LLM understand when to use i**t: * **Decision Making Process**: When we send a prompt, the LLM analyzes:   + The prompt content   + The available tools and their descriptions   + The purpose of each tool * Response Types: The code checks for two types of responses:  |  | | --- | | **if result.content:**  **print("No tool call was needed")**  **# LLM decided it can answer directly**    **if result.tool\_calls:**  **print("LLM decided to call tools")**  **# LLM decided it needs to use a tool** |  * Decision Factors: The LLM chooses tools based on:   + If the query is about scientific research → search\_pubmed   + If the query is about finding a doctor → recommend\_doctor   + If the query is about general information → search\_web   + If the query can be answered without tools → direct response * For example:   + "Research papers on diabetes treatments" → search\_pubmed because it explicitly asks for research papers   + "What doctor could I visit" → recommend\_doctor because it's about finding a suitable doctor   + "Explain what is diabetes" → might get a direct response because it's a general knowledge question   **The LLM makes these decisions automatically based on:**   * + **The semantic meaning of the user's query**   + **The tool descriptions provided in the docstrings**   + **The tool's parameter specifications**   + **The expected return types of the tools** |

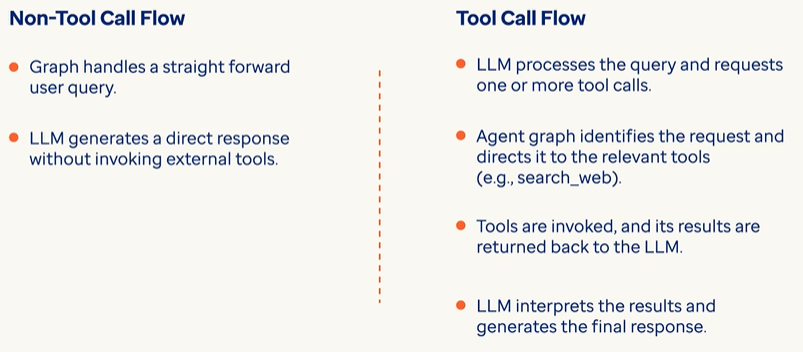
Step 4: Explore LLM Tool Calling with Custom Tools

* An agent is basically an LLM that has the capability to automatically call relevant functions to perform complex or tool-based tasks based on inputs or prompts provided by users.
* Tool calling, also popularly known as function calling, is the ability to reliably enable such LLMs to call external tools and APIs.

|  |
| --- |
| Tool calling in LangChain works by:   1. Registering defined tool functions using @tool decorator 2. Binding the tools to the model using llm.bind\_tools([tool1, tool2, ...]) 3. Passing a user query to the bound model 4. Letting the model decide whether to use a tool or respond directly  * This setup lets the model behave more like an agent that can take actions, observe results, and continue the conversation intelligently. * By using bind\_tools, we give the LLM the ability to understand what tools are available and make requests to use them only when needed. |

**Tool Call Requests**

* LLMs will not call and execute the tools, but will request tool calls based on reasoning on the input (user query) if they feel that they do not have enough information to answer the question directly.
* If the LLM can handle and generate the response without invoking the external tool is called “**Non-tool Call Flow**”



### CODE

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from dotenv import load\_dotenv  import json  from doctors import doctors\_db  from IPython.display import display, Markdown  # Import Chroma vector database for storing and searching embeddings  from langchain\_chroma import Chroma  from langchain\_core.tools import tool  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Read the JSON File  def load\_json\_data\_source():      try:          current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))          json\_file\_path = os.path.join(current\_dir, "search.json")          if not os.path.exists(json\_file\_path):              print(f"Error: search.json not found at {json\_file\_path}")              return None          with open(json\_file\_path, "r") as f:              search\_docs = json.load(f)          return search\_docs      except json.JSONDecodeError as e:          print(f"Error decoding JSON: {e}")          return None      except Exception as e:          print(f"Unexpected error: {e}")          return None  # Load the JSON data  search\_docs = load\_json\_data\_source()  if search\_docs is None:      print("Failed to load search data")  else:      print(f"Major Document Types: {list(search\_docs.keys())}")  # Store the publications and web Articles in in vector Db in different namespaces  web\_search\_db = Chroma.from\_texts(      search\_docs["search\_pages"],      collection\_name="web\_search\_db\_demo1",      embedding=embeddings\_client,  )  pubmed\_db = Chroma.from\_texts(      search\_docs["pubmed\_pages"],      collection\_name="pubmed\_db\_demo1",      embedding=embeddings\_client,  )  ### Tools Definations  @tool  def search\_web(query: str) -> list:      """Search the web for a query. Useful for retrieving general or up-to-date healthcare information."""      # Perform semantic similarity search over the web search vector database      results = web\_search\_db.similarity\_search(query)      docs = [doc.page\_content for doc in results]      return docs  @tool  def search\_pubmed(query: str) -> list:      """Search PubMed for scientific articles related to the query."""      # Perform semantic similarity search over the PubMed vector database      results = pubmed\_db.similarity\_search(query)      docs = [doc.page\_content for doc in results]      return docs  # Tool for recommending a doctor based on user symptoms or health-related queries  @tool  def recommend\_doctor(query: str) -> dict:      """Recommend the most suitable doctor based on the user's symptoms."""      doctors\_list = str(doctors\_db)      # Use the LLM to reason over the list and identify the best match for the user's concern      prompt = f"""You are an assistant helping recommend a doctor based on a patient's health issues.  Here is the list of available doctors:  {doctors\_list}  Given the user's query: "{query}"  Choose the most suitable doctor from the list. Only pick one doctor.  Return only the selected doctor's information in JSON format.  If unsure, recommend the General Physician.  """      response = llm.invoke(prompt)      return {"recommended\_doctor": response.content}  tools = [search\_web, search\_pubmed, recommend\_doctor]  llm\_with\_tools = llm.bind\_tools(tools)  prompts = [      "treatments available for diabetes",      "Research papers on diabetes treatments",      "What doctor could I visit for diabetes",      "Explain what is diabetes in simple terms",  ]  results = []  for prompt in prompts:      result = llm\_with\_tools.invoke(prompt)      results.append(result)  for prompt, result in zip(prompts, results):      # If the model provided a direct response without using any tools      if result.content:          print("No tool call was needed")          print(f"Prompt: {prompt}")          print(f"Direct LLM Response: {result.content}")      # If the model determined that a tool should be called      if result.tool\_calls:          print("LLM decided to call tools")          print(f"Prompt: {prompt}")          print(f"Tool Call Request: {result.tool\_calls}")      print("-" \* 50)      print()  # results = search\_web.invoke("Recent treatments in Diabetes")  # print(f"Total documents: {len(results)}")  # print()  # display(Markdown((results[0][:3000])))  # results = search\_pubmed.invoke("Recent treatments in Diabetes")  # print(f"Total documents: {len(results)}")  # print()  # display(Markdown((results[1][:3000])))  # result = recommend\_doctor.invoke("Treatments for Diabetes")  # print(f"Raw Tool Output:\n{json.dumps(result, indent=2)}")  # print("-" \* 50)  # print(f"\nFormatted Tool Output:\n{result['recommended\_doctor']}") |
| OUTPUT  **Major Document Types: ['search\_pages', 'pubmed\_pages']**  **LLM decided to call tools**  **Prompt: treatments available for diabetes**  **Tool Call Request: [{'name': 'search\_web', 'args': {'query': 'treatments available for diabetes'}, 'id': 'call\_Ntg63iNie58G2bPVuZNotiXl', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **LLM decided to call tools**  **Prompt: Research papers on diabetes treatments**  **Tool Call Request: [{'name': 'search\_pubmed', 'args': {'query': 'diabetes treatments'}, 'id': 'call\_qsGSDhexFNXXbPUGfopzuTDX', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **LLM decided to call tools**  **Prompt: What doctor could I visit for diabetes**  **Tool Call Request: [{'name': 'recommend\_doctor', 'args': {'query': 'diabetes'}, 'id': 'call\_0XQ6raiZjwS8X30LkKvu3U4N', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **No tool call was needed**  **Prompt: Explain what is diabetes in simple terms**  **Direct LLM Response: Diabetes is a health condition that occurs when the body has trouble using sugar (glucose) properly. Glucose is an important source of energy for our cells, but to use it, the body needs a hormone called insulin.**  **In diabetes, either the body doesn't make enough insulin, or it can't use the insulin it makes effectively. This leads to high levels of sugar in the blood, which can cause various health problems over time.**  **There are two main types of diabetes:**  **1. \*\*Type 1 Diabetes\*\*: The body doesn't produce insulin at all. This type usually starts in childhood or young adulthood.**    **2. \*\*Type 2 Diabetes\*\*: The body doesn't use insulin well or doesn't make enough insulin. This type is more common and often develops in adults, but it can also occur in children and teenagers.**  **Managing diabetes typically involves monitoring blood sugar levels, making healthy food choices, exercising, and sometimes taking medication or insulin.**  **--------------------------------------------------** |

# BUILD A TOOL-USE REACT AGENTIC AI SYSTEM WITH LANGCHAIN AND LANGRAPH

A diagram of a health care system

AI-generated content may be incorrect.

## OBJECTIVE

* Build a **ReAct-based Tool-Use Agent** using LanGraph **with built-in utility functions like create\_react\_agent()**
* Equip the agent with multiple tools (web search, PubMed search, doctor recommendation)
* Handle user queries end-to-end—from interpreting intent to delivering a useful, cited, and grounded response
* Simulate a multi-step reasoning and tool-using workflow

## AGENTS

A diagram of a task

AI-generated content may be incorrect.

### HOW AGENTS WORK

1. Task Assignment

* The user gives a task to the agent.
* The agent uses the **LLM as a reasoning engine** to understand the task.

2. Tool Selection

* The agent evaluates which tools it has available.
* It decides which tool is best suited to accomplish the task.

3. Action Execution

* The agent uses the selected tool to perform an action.
* It receives a **response** from the tool.

4. Response Evaluation

* The agent **does not immediately use or return** the response.
* It **observes and evaluates** the response.

5. Reasoning Loop

* If the response is **satisfactory**, it proceeds to generate the final output.
* If **not**, it continues reasoning:
  + It may reuse the same tool.
  + Or it may choose a **different tool**.

6. Final Response

* Once the agent is confident in the result, it uses the best response to generate a final answer.
* The final answer is sent back to the user.

LLMs That Support Agents

* **GPT** by OpenAI
* **Gemini** by Google
* **Mistral** from the open-source community

### SETTING UP

|  |  |
| --- | --- |
| Install The Tools | * Wikipedia: pip install Wikipedia * Duckduckgo Search: pip install duckduckgo-search * ***N****ote :* ***DuckDuckGo is a privacy-focused search engine that helps you find information online—just like Google or Bing*** |

|  |
| --- |
| **import os**  **from langchain\_openai import AzureChatOpenAI**  **# Import hub to pull pre-built prompts from LangChain Hub**  **from langchain import hub**  **# Import agent creation and execution classes for ReAct pattern**  **from langchain.agents import create\_react\_agent, AgentExecutor**  **# Import function to load predefined tools (Wikipedia, DuckDuckGo search)**  **from langchain\_community.agent\_toolkits.load\_tools import load\_tools**  **from dotenv import load\_dotenv**  **load\_dotenv()**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **gpt\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"] # Your GPT model deployment**  **llm = AzureChatOpenAI(**  **azure\_deployment=gpt\_deployment, # Specify which deployment to use**  **api\_version=api\_version, # Set the API version**  **azure\_endpoint=endpoint, # Set the Azure endpoint URL**  **api\_key=subscription\_key, # Provide authentication key**  **)**  **# Pull a pre-built ReAct (Reasoning + Acting) prompt template from LangChain Hub**  **# ReAct prompts help the agent reason through problems step-by-step and take actions**  **prompt = hub.pull("hwchase17/react")**  **# Load predefined tools for the agent to use**  **# "wikipedia": Allows searching Wikipedia for factual information**  **# "ddg-search": Allows searching DuckDuckGo for current web information**  **tools = load\_tools(["wikipedia", "ddg-search"])**  **# Create a ReAct agent that can reason and act using the provided tools**  **agent = create\_react\_agent(llm, tools, prompt)**  **# Create an executor to run the agent with verbose output to see reasoning steps**  **agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)**  **# Get user input for the task they want the agent to perform**  **task = input("Assign me Task")**  **# Check if the user provided a task**  **if task:**  **# Execute the agent with the given task**  **# The agent will reason through the problem and use tools as needed**  **response = agent\_executor.invoke({"input": task})**  **# Print the final output/answer from the agent**  **print(response["output"])** |

### CREATING ReACT(Reasoning -Acting) AGENT USING LANGCHAIN

How the Agent Works

A diagram of a tool execution

AI-generated content may be incorrect.

1. User Query Initiation

* The process begins when a **user submits a query**.
* This query is passed to the **React Agent**, which is a system designed to reason and act using tools and an LLM (Large Language Model).

2. LLM Reasoning (First Pass)

* The agent makes a **special LLM call** to interpret the query.
* The LLM analyzes the intent and decides:
  + Whether a tool is needed.
  + Which tool is appropriate.
  + What parameters might be required.

3. LLM Output Parsing

* The LLM returns a **textual response** (not structured code).
* This response needs to be **parsed** to extract:
  + The tool name.
  + The input parameters.
  + Any intermediate reasoning.

**This parsing step is crucial because LLMs typically return natural language, not executable code.**

4. Tool Execution

* Based on the parsed output, the agent:
  + Selects the correct tool.
  + Executes the tool with the provided input.
* The tool returns a **structured response**.

5. Second Iteration with LLM

* The agent now has:
  + The original query.
  + The tool’s output.
* It runs **another LLM call** to:
  + Decide if another tool is needed.
  + Refine the reasoning.
  + Determine if it can now answer the query.

6. Final Decision

* The agent evaluates:
  + Do I have enough information to answer?
  + Should I call another tool?
  + Should I return the final answer?

This loop continues until the agent is confident it can return a complete and correct response.

Summary of React Agent Workflow

* **React** stands for **Reason + Act**.
* It combines **LLM reasoning** with **tool execution** in a loop:
  1. **Reason** with LLM.
  2. **Act** by calling tools.
  3. **Repeat** until the answer is ready.

#### IMPORTANT NOTES

|  |
| --- |
| * The ReAct Agent always has structured output format. * Below is **Structured Output Format (used by the agent).** This is what the **LLM is expected to generate** during execution. It includes:   ***Question: <user question>***  ***Thought: <reasoning>***  ***Action: <tool name>***  ***Action Input: <input to the tool>***  ***Observation: <result from the tool>***  ***Thought: <updated reasoning>***  ***Final Answer: <final answer>***   * This format is parsed by tools like **ReActSingleInputOutputParser** to extract actionable steps.   Prompt Format (used to instruct the LLM):  This is what **we write and feed into the LLM** to guide it to produce the structured output. For example:  ***template = """***  ***Answer the following questions as best you can. You have access to the following tools:***  ***get\_text\_length(text: str) -> int - Get the length of the text in tokens.***  ***Use the following format:***  ***Question: the input question you must answer***  ***Thought: you should always think about what to do***  ***Action: the action to take, should be one of [get\_text\_length] Action***  ***Action Input: the input to the action***  ***Observation: the result of the action***  ***... (this Thought/Action/Action Input/Observation can repeat N times)***  ***Thought: I now know the final answer***  ***Final Answer: the final answer to the original input question***  ***Begin!***  ***Question: {input}***  ***Thought::***  ***"""***   * **NOTE : This prompt teaches the LLM to follow the ReAct-style output format.** |

#### TOOL CALLING

There are multiple ways to define agent tools

**1. Using the @tool Decorator**

|  |
| --- |
| from langchain.tools import tool  @tool  def get\_text\_length(text: str) -&gt; int:  return len(text)   * This is the **simplest and most declarative** way. * Automatically registers the function as a tool. * You can optionally pass metadata like name, description, etc. |

**2. Using the Tool Class Directly**

|  |
| --- |
| from langchain.tools import Tool  def get\_text\_length(text: str) -&gt; int:  return len(text)  tool = Tool.from\_function(  func=get\_text\_length,  name="GetTextLength",  description="Returns the length of the input text"  )   * Gives **more control** over tool metadata. * Useful when you want to dynamically create tools or wrap existing functions. |

**3. Using StructuredTool for Typed Inputs**

|  |
| --- |
| from langchain.tools import StructuredTool  def get\_text\_length(text: str) -&gt; int:  return len(text)  tool = StructuredTool.from\_function(  func=get\_text\_length,  name="GetTextLength",  description="Returns the length of the input text"  )   * Ensures **structured input/output** using type hints. * Ideal for tools that require **multiple or complex parameters**. |

**4. Using BaseTool for Custom Behavior**

|  |
| --- |
| from langchain.tools import BaseTool  class CustomTool(BaseTool):  name = "CustomTextLength"  description = "Custom tool to get text length"  def \_run(self, text: str) -&gt; int:  return len(text)  def \_arun(self, text: str) -&gt; int:  raise NotImplementedError("Async not supported")  tool = CustomTool()   * Allows **full customization** of tool behavior. * You can override \_run and \_arun for sync/async execution.   Useful for **advanced use cases** like logging, error handling, or chaining. |

**5. Wrapping External APIs or Services**

|  |
| --- |
| We can wrap any external API call as a tool:  import requests  from langchain.tools import Tool  def get\_weather(city: str) -&gt; str:  response = requests.get(f"https://api.weatherapi.com/v1/current.json?q={city}")  return response.json()["current"]["condition"]["text"]  weather\_tool = Tool.from\_function(  func=get\_weather,  name="GetWeather",  description="Returns current weather for a city"  )  • Enables integration with external services.  • Can be used to build tools for databases, REST APIs, etc. |

**6. Using Tool Collections**

|  |
| --- |
| You can group tools together:  from langchain.agents import initialize\_agent  from langchain.agents.agent\_types import AgentType  tools = [tool1, tool2, tool3]  agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)   * Useful for **multi-tool agents**. * Each tool can be defined using any of the above methods. |

#### EXAMPLE

In below example – We will use **@tool** decorator to register a tool

Example- Calling Tools using @tool decorator

If we want to explicitly call the tool (function). We can use the invoke methog

|  |
| --- |
| **@tool**  **def get\_text\_length(text: str) -> int:**  **"""**  **Get the length of the text in tokens.**  **"""**  **return len(text)**  **print(get\_text\_length.invoke({"text":"Hello, world! This is a test of the text length function."})) ## To test**  **print(tools) 🡪** *[StructuredTool(name='get\_text\_length', description='Get the length of the text in tokens.', args\_schema=<class 'langchain\_core.utils.pydantic.get\_text\_length'>, func=<function get\_text\_length at 0x000001CEADD97060>)]* |

Stop Argument

|  |
| --- |
| **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **temperature=0,**  **stop=["\nObservation", "Observation"]**  **)**   * The stop argument in AzureChatOpenAI is used to specify sequences that will cause the model to stop generating further tokens. * It tells the model to stop generating text when it encounters any of the specified sequences * This is particularly useful in agent implementations to prevent the model from trying to generate the observation itself   For Example : stop=["\nObservation", "Observation"]   * "\nObservation": Stops when it encounters "Observation" at the start of a new line * "Observation": Stops when it encounters "Observation" anywhere in the text   Why it's needed here:  In your ReAct agent implementation, the model should:  Generate the "Thought" about what to do  Generate the "Action" and "Action Input"  Then STOP before "Observation"  The actual observation comes from executing the tool, not from the model  Without these stop sequences, the model might try to hallucinate the observation instead of letting the actual tool provide it  Example flow:  Thought: I need to check the length of the text  Action: get\_text\_length  Action Input: "Hello World"  [Model stops here because of stop sequence]  [Tool executes and provides real observation]  This ensures that the model only generates the reasoning and action parts, while the actual tool execution provides the observation, making the agent's behavior more reliable and controlled. |

|  |
| --- |
| ReAct REPL Agent  A ReAct agent works in a loop like this:   1. **Thought** → The agent reasons about what it needs to do next. 2. **Action** → It picks a tool/function and provides the input. 3. **Observation** → The actual output returned by that tool. 4. **Thought** (again) → It reasons based on that observation.   **What is the meaning of Agent Reason?**  **Reasoning = The model’s internal decision-making process → "What do I know? What should I do next? Do I need another tool, or can I answer now?"** |

#### CODE

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain.prompts import PromptTemplate  import os  from dotenv import load\_dotenv  from langchain.tools import tool  from langchain.tools.render import render\_text\_description  from langchain import hub  from langchain.agents.output\_parsers import ReActSingleInputOutputParser  from typing import Union  from langchain.agents.agent import AgentAction, AgentFinish  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,      stop=["\nObservation", "Observation"],  )  @tool  def get\_text\_length(text: str) -> int:      """      Get the length of the text in tokens.      """      text = text.strip("\n").strip('"')      return len(text)  tools = [get\_text\_length]  template = """          Answer the following questions as best you can. You have access to the following tools:  {tools}  Use the following format:  Question: the input question you must answer  Thought: you should always think about what to do  Action: the action to take, should be one of [{tool\_names}]  Action Input: the input to the action  Observation: the result of the action  ... (this Thought/Action/Action Input/Observation can repeat N times)  Thought: I now know the final answer  Final Answer: the final answer to the original input question  Begin!  Question: {input}  Thought:  """  prompt = PromptTemplate.from\_template(template=template).partial(      tools=render\_text\_description(tools), tool\_names=",".join(t.name for t in tools)  )  agent = {"input": lambda input\_dict: input\_dict["input"]} | prompt | llm | ReActSingleInputOutputParser()  agent\_step:Union[AgentAction, AgentFinish] = agent.invoke({"input": "What is the length of the text 'Hello, World!'?"})  def find\_tool\_by\_name(tools, tool\_name):      for tool in tools:          if tool.name == tool\_name:              return tool      raise ValueError(f"Tool with name {tool\_name} not found.")  if isinstance(agent\_step, AgentAction):      tool\_name = agent\_step.tool      tool\_input = agent\_step.tool\_input      tool\_to\_use = find\_tool\_by\_name(tools, tool\_name)      observation = tool\_to\_use.func(str(tool\_input))      print(f"Observation: {observation}") |
| **OUTPUT: Observation: 15** |

**if the code works without bind\_tools(), why use it at all?** Let’s break it down clearly:

**Why Your Code Works Without bind\_tools()**

Your current setup uses a **ReAct-style agent** with a manually crafted prompt and a parser (ReActSingleInputOutputParser) that extracts tool calls from plain text. This works because:

* The prompt explicitly tells the LLM what tools exist and how to use them.
* The LLM generates output in a format like: Thought: I should use a tool. Action: get*text*length Action Input: Hello, World!
* The parser reads this and extracts the tool name and input.
* You manually execute the tool afterward.

So yes — **this works without bind\_tools()** because you're handling everything manually.

**What bind\_tools() Adds (and Why You Might Still Use It)**

bind\_tools() is **not required** for ReAct-style agents, but it becomes valuable when:

**1. You Want Structured Tool Calling**

LangChain supports agents that use **structured outputs** (like OpenAI’s function calling). These agents don’t rely on text parsing — they expect the LLM to return a structured JSON object like:

{

"tool": "get\_text\_length",

"tool\_input": "Hello, World!"

}

To enable this, the LLM needs to know:

* What tools exist.
* What inputs they expect.
* What schema they follow.

That’s exactly what bind\_tools() does.

**2. You Want Better Accuracy and Fewer Hallucinations**

Even in ReAct-style agents, bind\_tools() helps the LLM:

* Avoid hallucinating tool names.
* Use correct input formats.
* Understand tool capabilities better.

This is especially useful when you have **many tools** or **complex input types**.

**3. You Want to Future-Proof Your Agent**

If you later switch to:

* AgentExecutor
* RunnableAgent
* OpenAI tool-calling agents
* LangGraph workflows

…you’ll **need** bind\_tools() to make those work properly.

**🔄 Summary**

| **Feature** | **Without bind\_tools()** | **With bind\_tools()** |
| --- | --- | --- |
| ReAct-style agent | ✅ Works | ✅ Works (better) |
| Structured tool calling | ❌ Not supported | ✅ Required |
| Tool schema awareness | ❌ None | ✅ Full |
| Future agent compatibility | ❌ Limited | ✅ Ready |
| Hallucination prevention | ❌ Manual | ✅ Improved |

**✅ When to Use bind\_tools()**

Use it if:

* You want to switch to structured agents.
* You want better tool usage accuracy.
* You want to scale with more tools.
* You want to future-proof your code.

Would you like me to show how your current ReAct agent could be upgraded to a structured agent using AgentExecutor or RunnableAgent with bind\_tools()?

A screenshot of a computer program

AI-generated content may be incorrect.

Why we need ReActSingleInputOutputParser?

The ReActSingleInputOutputParser is a specialized parser in LangChain designed to interpret outputs from language models that follow the **ReAct prompting style**—a format where the model alternates between reasoning ("Thought") and tool usage ("Action") to solve tasks.

What It Does

This parser:

* **Parses LLM outputs** that follow a specific structure.
* **Returns structured objects** like AgentAction or AgentFinish depending on the output.
* **Supports single tool input**, hence the name.

Expected Output Formats

It expects the LLM output to be in one of two formats:

|  |  |
| --- | --- |
| **Tool Invocation (AgentAction)**:  Thought: I need to look this up.  Action: search  Action Input: what is the temperature in SF? | **Final Answer (AgentFinish)**:  Thought: I now know the final answer.  Final Answer: The temperature is 100 degrees. |

Based on these formats, the parser uses regular expressions to extract the relevant parts and return either:

* **AgentAction(action, action\_input)**
* **AgentFinish(final\_answer)**

A diagram of a computer

AI-generated content may be incorrect.

### AGENTACTION , AGENT FINISH AND ReACT LOOP