ReAct Agent

💡 **ReAct = Reasoning + Acting**

- It is a method for building agents where the LLM doesn't just give a direct answer, but:
  1. **Thinks step by step** (Reasoning)
  2. **Takes actions** using tools (Acting)
  3. **Uses results from tools** to continue reasoning
  4. **Finally answers the user**

Why is it needed?

- Normal LLMs can hallucinate (make up wrong answers).
- With **ReAct**, the agent can:
  - Call a **tool** (like Google, Wikipedia, Calculator, Database, API).
  - Check the tool's result.
  - Use that real information in the answer.

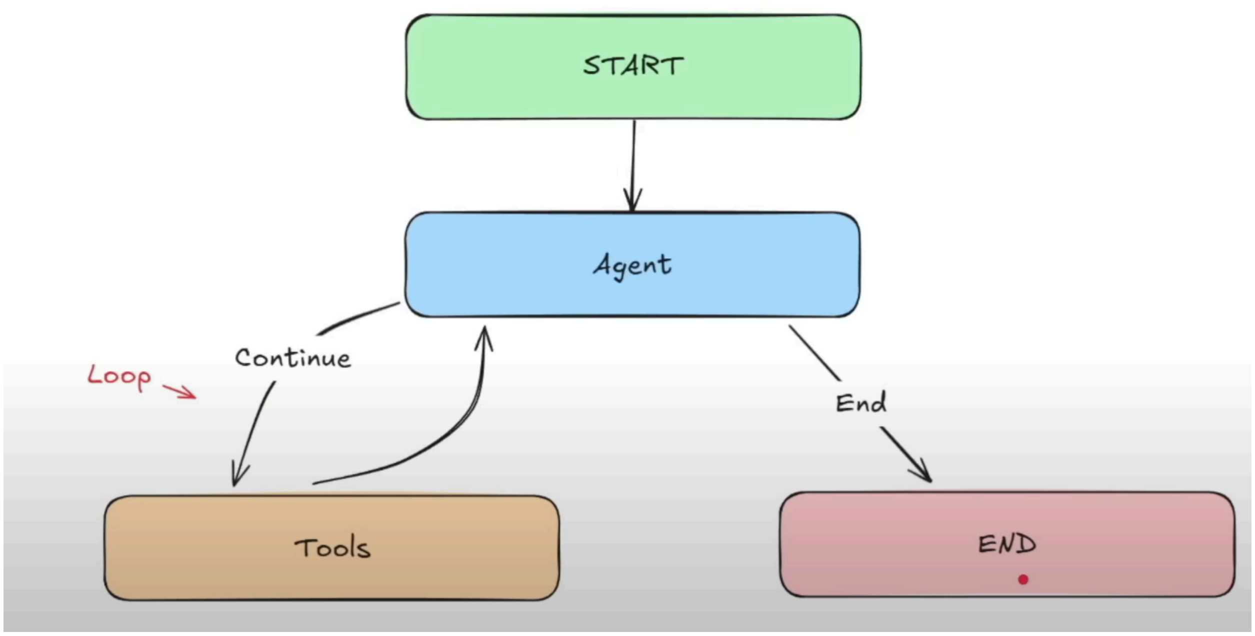
This makes it **more reliable**.

How does it work internally?

Imagine a conversation:

1. **User**: "What is the population of France divided by 2?"
2. **Agent (Reasoning)**: I don't know directly. Step 1: I should look up France's population.
3. **Agent (Action)**: Calls a tool (Wikipedia API).

4. **Tool Result**: "France population = 67 million"
5. **Agent (Reasoning)**: Now divide 67 million by 2 = 33.5 million.
6. **Agent (Final Answer)**: "It's about 33.5 million."



Code

```
from typing import Annotated, Sequence, TypedDict
from dotenv import load_dotenv
from langchain_core.messages import BaseMessage # The foundational class
for all message types in LangGraph
from langchain_core.messages import ToolMessage # Passes data back to LL
M after it calls a tool such as the content and the tool_call_id
from langchain_core.messages import SystemMessage
from langchain_groq import ChatGroq
from langchain_core.tools import tool
from langgraph.graph.message import add_messages
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode
```

```
load_dotenv()
```

`Annotated` → Allows you to **attach extra metadata** to a type hint. I

`Sequence` → It's used to say: "This function accepts any ordered collection, not just a list."

✅ Works with:

- `["a", "b", "c"]` (list)
- `("a", "b", "c")` (tuple)
- `"abc"` (string)

📌 Why use `Sequence` instead of `List`?

- `List[str]` only accepts lists.
- `Sequence[str]` is **more flexible**, and works with any indexable iterable (list, tuple, str, etc.).

`ToolMessage`

- A special message used when a **tool has returned a result** and you're sending that result back to the LLM.
- Typically follows a **tool call** that the LLM made.

🧠 **Purpose:**

- Bridges the gap between the **LLM asking for a tool** and the **tool's actual response**.

✅ `BaseMessage`

- `BaseMessage` is the **abstract base class** for all message types in LangChain.
- Other message types (like `HumanMessage`, `AIMessage`, `SystemMessage`, `ToolMessage`, etc.) **inherit from** this.
- **`BaseMessage` is the parent class**

⚙️ `SystemMessage`

- A message type used to pass **instructions, role-setting, or context** to the LLM.

`add_messages`

- This is a **reducer function**
- LangGraph agents typically maintain a **state** that contains a list of messages (like `HumanMessage` , `AIMessage` , etc.). `add_messages` makes it easier to **append new messages** to that list in a structured and consistent way.

  ```
  from langgraph.graph.message import add_messages
  from langchain_core.messages import AIMessage

  def some_node(state):
      response = AIMessage(content="Sure, here's the info you asked for.")
      return add_messages(state, [response])
  ```

`ToolMessage` →

- A message type specifically used when an LLM calls a **tool**.
  - It contains the **result of a tool execution**.
  - Also carries the `tool_call_id` so LangChain knows which request this belongs to.

Example flow:

1. LLM says: *"Use calculator with input 2+2"*.
2. Tool is executed → result = `4`.
3. This result is wrapped in a `ToolMessage` and passed back to the LLM.

```
from langchain_core.tools import tool
```

`@tool` → Turns a function into a Tool

- This is a **decorator**. You put `@tool` above a Python function to magically convert it into a tool that the AI can recognize and call.

`ToolNode` →

A **ready-made LangGraph node** that handles **tool execution**.

- Instead of writing your own node to call a tool and process its result, you just use `ToolNode`.
- It takes care of:
  - Running the tool
  - Wrapping the result in `ToolMessage`
  - Sending it back to the LLM

A **pre-built node** that handles all the complexity of:

1. Receiving a list of tool calls from the AI.
2. Finding the correct tool.
3. Executing it.
4. Packaging the results into objects.

It saves you from writing this logic yourself.

```
class AgentState(TypedDict):
    message: Annotated[Sequence[BaseMessage], add_messages]
```

`Sequence[BaseMessage]`

- `Sequence` = any ordered, indexable collection (like a `list` or `tuple`).
- `BaseMessage` = parent class for `HumanMessage`, `AIMessage`, `SystemMessage`, etc.

So, `"message"` will hold a **list of messages**:

```
[
  HumanMessage(content="Hello"),
  AIMessage(content="Hi, how can I help?")
]
```

`Annotated[..., add_messages]`

In LangGraph, that metadata (`add_messages`) tells the framework:

> "When this node returns new messages, use the `add_messages` function to merge them into this field."

The **special instruction** **`add_messages`**, LangGraph knows:

> "Whenever I add new messages, I should not overwrite old ones, but append them to the list."

Create Tool

```
@tool
def add(a: int, b:int):
    """This is an addition function that adds 2 numbers together"""

    return a + b

@tool
def subtract(a: int, b: int):
    """Subtraction function"""
    return a - b

@tool
def multiply(a: int, b: int):
    """Multiplication function"""
    return a * b
```

💡 **!!If the doc string is removed, the function will not work.**

Create a list of tools

```
#Create a list of tools

tools = [add, subtract, multiply]
```

Model + Tools:

```
#Model

model =ChatGroq(model="llama-3.3-70b-versatile").bind_tools(tools)
```

- In this way, model has access to the tools

Model Call Node:

```
def model_call(state:AgentState) → AgentState:
    system_prompt = SystemMessage(content="You are my AI assistant, pleas
e answer my query to the best of your ability.")
    response = model.invoke([system_prompt] + state["messages"])
    return {"messages": [response]}
```

```
Output:

AIMessage(content="I'll do my best to provide a helpful and accurate resp
onse. What's your question?", additional_kwargs={}, response_metadata=
{'token_usage': {'completion_tokens': 19, 'prompt_tokens': 47, 'total_token
s': 66, 'completion_time': 0.046528529, 'prompt_time': 0.017985182, 'queu
e_time': 0.047918769, 'total_time': 0.064513711}, 'model_name': 'llama-3.3-
70b-versatile', 'system_fingerprint': 'fp_2ddfbb0da0', 'finish_reason': 'sto
```

```
p', 'logprobs': None}, id='run--3883e593-2eb1-4480-b079-6ea4c53ca20
3-0', usage_metadata={'input_tokens': 47, 'output_tokens': 19, 'total_token
s': 66})
```

`return {"messages": [response]}` → **Compact way to update the state.**

> 💡 **We are not returning a string because (`response.content`)→ `"messages"` field must hold a list of `BaseMessage` objects, not plain strings.**

> 💡 We just write `"messages": [response]` because the `add_messages` function handles the appending of messages.

Conditional Edge

```
def should_continue(state:AgentState)→ AgentState:

    messages = state["messages"]

    last_message = messages[-1]

    if not last_message.tool_calls:
        return "end"
    else:
        return "continue"
```

`messages[-1]` gets the **most recent message** (last one in the list)

What is `.tool_calls`?

- Some messages (usually from the assistant) may contain **tool calls**.

- Example: the assistant might decide: *"I need to use the Wikipedia API tool"*.
- In that case, the message will have something like:

```
{
  "role": "assistant",
  "tool_calls": [{"id": "1", "name": "WikipediaAPIWrapper", "arguments": {"query": "cats"}}]
}
```

> If `.tool_calls` is **empty**, it means the assistant is not calling any tool and is just giving a final response.

- If the **last message has no tool calls** → that means the assistant is done → return `"end"`.
- If the **last message contains tool calls** → that means the assistant wants to run tools (like search, calculator, DB query) → return `"continue"`.

Graph

```
graph = StateGraph(AgentState)

graph.add_node("our_agent", model_call)

#Tool Node
tool_node = ToolNode(tools=tools)
graph.add_node("tools", tool_node) #Contains multiple tools

graph.set_entry_point("our_agent")

graph.add_conditional_edges(
    "our_agent",
    should_continue,
    {
        "continue": "tools",
```

```
        "end" : END
    }
)

#Edge that goes back to our tool
graph.add_edge("tools", "our_agent")

app = graph.compile()
```

`graph.add_edge("tools", "our_agent"):` Edge that goes back to our tool.

- But we also wanted **`tools` → `our_agent`**

Stream

```
def print_stream(stream):
    for s in stream:
        message = s["messages"][-1]
        if isinstance(message, tuple):
            print(message)
        else:
            message.pretty_print()
```

`for s in stream:`

- This means: go through each **item** (`s`) inside `stream`.
- If `stream` is a list (or an iterable object), this loop will pick one item at a time.

Think of `stream` like a **queue of updates/messages** coming from an AI pipeline — you're checking them one by one.

`message = s["messages"][-1]`

- `s` is expected to be a dictionary (key-value pairs).
- Inside `s`, there is a key `"messages"`.
- `s["messages"]` is a list of messages.

- `[-1]` means "last item in the list".

`if isinstance(message, tuple):`

- `isinstance(x, tuple)` checks if `message` is a **tuple** (a fixed group of values like `(a, b)` in Python).
- Why? Because sometimes the last message might be stored as a **tuple** instead of a custom object.
- If `message` is not a tuple, it must be some kind of **object** (likely a `ChatMessage` or `AIMessage` from LangChain).
- **That object has a built-in method `.pretty_print()` which formats it nicely for display.**

```
inputs = {"messages": [("user", "Add 40 + 12")]}
print_stream(app.stream(inputs, stream_mode="values"))
```

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

Add 40 + 12
================================== Ai Message ==================================
Tool Calls:
  add (6gjdhgmks)
 Call ID: 6gjdhgmks
  Args:
    a: 40
    b: 12
================================= Tool Message =================================
Name: add

52
================================== Ai Message ==================================

The result of the addition is 52.
```

- `.stream(inputs, stream_mode="values")` runs the pipeline but **instead of giving you the final result all at once**, it yields outputs **step by step** as a **stream** (like a generator).

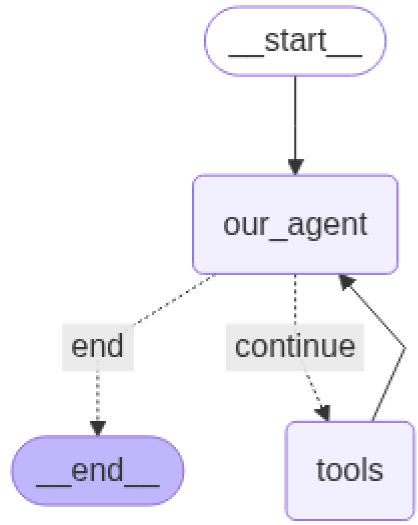
- Each yielded value (`s`) is a dictionary containing intermediate + final messages.

```
inputs = {"messages": [("user", "Add 40 + 12. add 5+7")]}
print_stream(app.stream(inputs, stream_mode="values"))
```

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

Add 40 + 12. add 5+7
================================== Ai Message ==================================
Tool Calls:
  add (asx4mnnn7)   1
 Call ID: asx4mnnn7
  Args:
    a: 40
    b: 12
  add (fkhb2dq20)   2
 Call ID: fkhb2dq20
  Args:
    a: 5
    b: 7
================================= Tool Message =================================
Name: add

12
================================== Ai Message ==================================

The results of the additions are 52 and 12.
```

Visualize

```
from IPython.display import Image, display
display(Image(app.get_graph().draw_mermaid_png()))
```


__start__
our_agent
end
continue
__end__
tools