

[Home](#) [Guides](#) [Tutorials](#) [Quick Start](#)

LangGraph Quickstart

In this tutorial, we will build a support chatbot in LangGraph that can:

- ✅ **Answer common questions** by searching the web
- ✅ **Maintain conversation state** across calls
- ✅ **Route complex queries** to a human for review
- ✅ **Use custom state** to control its behavior
- ✅ **Rewind and explore** alternative conversation paths

We'll start with a **basic chatbot** and progressively add more sophisticated capabilities, introducing key LangGraph concepts along the way. Let's dive in! 🚀

Setup

First, install the required packages and configure your environment:

```
%%capture --no-stderr
%pip install -U langgraph langsmith langchain_anthropic
```

```
import getpass
import os

def _set_env(var: str):
    if not os.environ.get(var):
        os.environ[var] = getpass.getpass(f"{var}: ")

_set_env("ANTHROPIC_API_KEY")
```



Set up LangSmith for LangGraph development

Sign up for LangSmith to quickly spot issues and improve the performance of your LangGraph projects. LangSmith lets you use trace data to debug, test, and monitor your LLM apps built with LangGraph — read more about how to get started [here](#).

Part 1: Build a Basic Chatbot

We'll first create a simple chatbot using LangGraph. This chatbot will respond directly to user messages. Though simple, it will illustrate the core concepts of building with LangGraph. By the end of this section, you will have a built rudimentary chatbot.

Start by creating a `StateGraph`. A `StateGraph` object defines the structure of our chatbot as a "state machine". We'll add `nodes` to represent the llm and functions our chatbot can call and `edges` to specify how the bot should transition between these functions.

```
from typing import Annotated

from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

class State(TypedDict):
    # Messages have the type "list". The `add_messages` function
    # in the annotation defines how this state key should be updated
    # (in this case, it appends messages to the list, rather than overwriting
    # them)
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)
```

API Reference: [StateGraph](#) | [START](#) | [END](#) | [add_messages](#)

Our graph can now handle two key tasks:

1. Each `node` can receive the current `State` as input and output an update to the state.
2. Updates to `messages` will be appended to the existing list rather than overwriting it, thanks to the prebuilt `add_messages` function used with the `Annotated` syntax.



Concept

When defining a graph, the first step is to define its `State`. The `State` includes the graph's schema and [reducer functions](#) that handle state updates. In our example, `State` is a `TypedDict` with one key: `messages`. The `add_messages` reducer function is used to append new messages to the list instead of overwriting it. Keys without a reducer annotation will overwrite previous values. Learn more about state, reducers, and related concepts in [this guide](#).

Next, add a "chatbot" node. Nodes represent units of work. They are typically regular python functions.

```
from langchain_anthropic import ChatAnthropic

llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")

def chatbot(state: State):
    return {"messages": [llm.invoke(state["messages"])]}

# The first argument is the unique node name
# The second argument is the function or object that will be called whenever
# the node is used.
graph_builder.add_node("chatbot", chatbot)
```

API Reference: [ChatAnthropic](#)

Notice how the `chatbot` node function takes the current `State` as input and returns a dictionary containing an updated `messages` list under the key "messages". This is the basic pattern for all LangGraph node functions.

The `add_messages` function in our `State` will append the llm's response messages to whatever messages are already in the state.

Next, add an `entry` point. This tells our graph **where to start its work** each time we run it.

```
graph_builder.add_edge(START, "chatbot")
```

Similarly, set a `finish` point. This instructs the graph **"any time this node is run, you can exit."**

```
graph_builder.add_edge("chatbot", END)
```

Finally, we'll want to be able to run our graph. To do so, call "`compile()`" on the graph builder. This creates a "`CompiledGraph`" we can use invoke on our state.

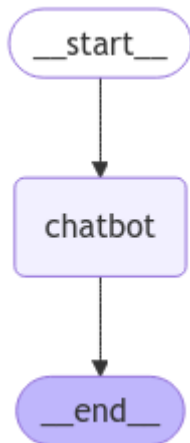
```
graph = graph_builder.compile()
```

You can visualize the graph using the `get_graph` method and one of the "draw" methods, like `draw_ascii` or `draw_png`. The `draw` methods each require additional dependencies.

```
from IPython.display import Image, display

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

pass



Now let's run the chatbot!

Tip: You can exit the chat loop at any time by typing "quit", "exit", or "q".

```

def stream_graph_updates(user_input: str):
    for event in graph.stream({"messages": [{"role": "user", "content":
user_input}]}):
        for value in event.values():
            print("Assistant:", value["messages"][-1].content)

while True:
    try:
        user_input = input("User: ")
        if user_input.lower() in ["quit", "exit", "q"]:
            print("Goodbye!")
            break

        stream_graph_updates(user_input)
    except:
        # fallback if input() is not available
        user_input = "What do you know about LangGraph?"
        print("User: " + user_input)
        stream_graph_updates(user_input)
        break
  
```

Assistant: LangGraph is a library designed to help build stateful multi-agent applications using language models. It provides tools for creating workflows and state machines to coordinate multiple AI agents or language model interactions. LangGraph is built on top of LangChain, leveraging its components while adding graph-based coordination capabilities. It's particularly useful for developing more complex, stateful AI applications that go beyond simple query-response interactions. Goodbye!

Congratulations! You've built your first chatbot using LangGraph. This bot can engage in basic conversation by taking user input and generating responses using an LLM. You can inspect a [LangSmith Trace](#) for the call above at the provided link.

However, you may have noticed that the bot's knowledge is limited to what's in its training data. In the next part, we'll add a web search tool to expand the bot's knowledge and make it more capable.

Below is the full code for this section for your reference:

Full Code

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")

def chatbot(state: State):
    return {"messages": [llm.invoke(state["messages"])]}

# The first argument is the unique node name
# The second argument is the function or object that will be called whenever
# the node is used.
graph_builder.add_node("chatbot", chatbot)
graph_builder.set_entry_point("chatbot")
graph_builder.set_finish_point("chatbot")
graph = graph_builder.compile()
```

API Reference: [ChatAnthropic](#) | [StateGraph](#) | [add_messages](#)

Part 2: Enhancing the Chatbot with Tools

To handle queries our chatbot can't answer "from memory", we'll integrate a web search tool. Our bot can use this tool to find relevant information and provide better responses.

Requirements

Before we start, make sure you have the necessary packages installed and API keys set up:

First, install the requirements to use the [Tavily Search Engine](#), and set your `TAVILY_API_KEY`.

```
%%capture --no-stderr
%pip install -U tavily-python langchain_community
```

```
_set_env("TAVILY_API_KEY")
```

```
TAVILY_API_KEY: .....
```

Next, define the tool:

```
from langchain_community.tools.tavily_search import TavilySearchResults

tool = TavilySearchResults(max_results=2)
tools = [tool]
tool.invoke("What's a 'node' in LangGraph?")
```

API Reference: [TavilySearchResults](#)

```
[{'url': 'https://medium.com/@cplog/introduction-to-langgraph-a-beginners-guide-14f9be027141',
  'content': 'Nodes: Nodes are the building blocks of your LangGraph. Each node represents a function or a computation step. You define nodes to perform specific tasks, such as processing input, making ...'},
 {'url': 'https://saksheepatil05.medium.com/demystifying-langgraph-a-beginner-friendly-dive-into-langgraph-concepts-5ffe890ddac0',
  'content': 'Nodes (Tasks): Nodes are like the workstations on the assembly line. Each node performs a specific task on the product. In LangGraph, nodes are Python functions that take the current state, do some work, and return an updated state. Next, we define the nodes, each representing a task in our sandwich-making process.'}]
```

The results are page summaries our chat bot can use to answer questions.

Next, we'll start defining our graph. The following is all **the same as in Part 1**, except we have added `bind_tools` on our LLM. This lets the LLM know the correct JSON format to use if it wants to use our search engine.

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
```

```

from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
# Modification: tell the LLM which tools it can call
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

graph_builder.add_node("chatbot", chatbot)

```

API Reference: [ChatAnthropic](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#)

Next we need to create a function to actually run the tools if they are called. We'll do this by adding the tools to a new node.

Below, we implement a `BasicToolNode` that checks the most recent message in the state and calls tools if the message contains `tool_calls`. It relies on the LLM's `tool_calling` support, which is available in Anthropic, OpenAI, Google Gemini, and a number of other LLM providers.

We will later replace this with LangGraph's prebuilt `ToolNode` to speed things up, but building it ourselves first is instructive.

```

import json

from langchain_core.messages import ToolMessage

class BasicToolNode:
    """A node that runs the tools requested in the last AIMessage."""

    def __init__(self, tools: list) -> None:
        self.tools_by_name = {tool.name: tool for tool in tools}

    def __call__(self, inputs: dict):
        if messages := inputs.get("messages", []):
            message = messages[-1]
        else:
            raise ValueError("No message found in input")
        outputs = []

```

```

        for tool_call in message.tool_calls:
            tool_result = self.tools_by_name[tool_call["name"]].invoke(
                tool_call["args"]
            )
            outputs.append(
                ToolMessage(
                    content=json.dumps(tool_result),
                    name=tool_call["name"],
                    tool_call_id=tool_call["id"],
                )
            )
        return {"messages": outputs}

tool_node = BasicToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

```

API Reference: [ToolMessage](#)

With the tool node added, we can define the `conditional_edges`.

Recall that **edges** route the control flow from one node to the next. **Conditional edges** usually contain "if" statements to route to different nodes depending on the current graph state. These functions receive the current graph `state` and return a string or list of strings indicating which node(s) to call next.

Below, call define a router function called `route_tools`, that checks for `tool_calls` in the chatbot's output. Provide this function to the graph by calling `add_conditional_edges`, which tells the graph that whenever the `chatbot` node completes to check this function to see where to go next.

The condition will route to `tools` if tool calls are present and `END` if not.

Later, we will replace this with the prebuilt `tools_condition` to be more concise, but implementing it ourselves first makes things more clear.

```

def route_tools(
    state: State,
):
    """
    Use in the conditional_edge to route to the ToolNode if the last message
    has tool calls. Otherwise, route to the end.
    """
    if isinstance(state, list):
        ai_message = state[-1]
    elif messages := state.get("messages", []):
        ai_message = messages[-1]
    else:
        raise ValueError(f"No messages found in input state to tool_edge: {state}")
    if hasattr(ai_message, "tool_calls") and len(ai_message.tool_calls) > 0:
        return "tools"
    return END

```



```
# The `tools_condition` function returns "tools" if the chatbot asks to use a
# tool, and "END" if
# it is fine directly responding. This conditional routing defines the main
# agent loop.
graph_builder.add_conditional_edges(
    "chatbot",
    route_tools,
    # The following dictionary lets you tell the graph to interpret the
    # condition's outputs as a specific node
    # It defaults to the identity function, but if you
    # want to use a node named something else apart from "tools",
    # You can update the value of the dictionary to something else
    # e.g., "tools": "my_tools"
    {"tools": "tools", END: END},
)
# Any time a tool is called, we return to the chatbot to decide the next step
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")
graph = graph_builder.compile()
```

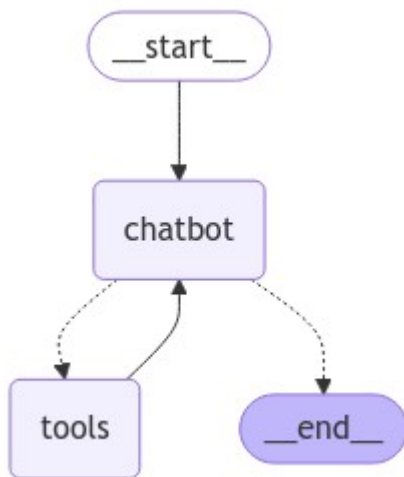
Notice that conditional edges start from a single node. This tells the graph "any time the 'chatbot' node runs, either go to 'tools' if it calls a tool, or end the loop if it responds directly.

Like the prebuilt `tools_condition`, our function returns the `END` string if no tool calls are made. When the graph transitions to `END`, it has no more tasks to complete and ceases execution. Because the condition can return `END`, we don't need to explicitly set a `finish_point` this time. Our graph already has a way to finish!

Let's visualize the graph we've built. The following function has some additional dependencies to run that are unimportant for this tutorial.

```
from IPython.display import Image, display

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



Now we can ask the bot questions outside its training data.

```

while True:
    try:
        user_input = input("User: ")
        if user_input.lower() in ["quit", "exit", "q"]:
            print("Goodbye!")
            break

        stream_graph_updates(user_input)
    except:
        # fallback if input() is not available
        user_input = "What do you know about LangGraph?"
        print("User: " + user_input)
        stream_graph_updates(user_input)
        break
  
```

Assistant: [{'text': "To provide you with accurate and up-to-date information about LangGraph, I'll need to search for the latest details. Let me do that for you.", 'type': 'text'}, {'id': 'toolu_01Q588CszHaSvvP2MxRq9zRD', 'input': {'query': 'LangGraph AI tool information', 'name': 'tavily_search_results_json', 'type': 'tool_use'}}]

Assistant: [{"url": "https://www.langchain.com/langgraph", "content": "LangGraph sets the foundation for how we can build and scale AI workloads \u2014 from conversational agents, complex task automation, to custom LLM-backed experiences that 'just work'. The next chapter in building complex production-ready features with LLMs is agentic, and with LangGraph and LangSmith, LangChain delivers an out-of-the-box solution ..."}, {"url": "https://github.com/langchain-ai/langgraph", "content": "Overview. LangGraph is a library for building stateful, multi-actor applications with LLMs, used to create agent and multi-agent workflows. Compared to other LLM frameworks, it offers these core benefits: cycles, controllability, and persistence. LangGraph allows you to define flows that involve cycles, essential for most agentic architectures ..."}]

Assistant: Based on the search results, I can provide you with information about LangGraph:

1. Purpose:

LangGraph is a library designed for building stateful, multi-actor applications with Large Language Models (LLMs). It's particularly useful for creating agent and multi-agent workflows.

2. Developer:

LangGraph is developed by LangChain, a company known for its tools and frameworks in the AI and LLM space.

3. Key Features:

- Cycles: LangGraph allows the definition of flows that involve cycles, which is essential for most agentic architectures.
- Controllability: It offers enhanced control over the application flow.
- Persistence: The library provides ways to maintain state and persistence in LLM-based applications.

4. Use Cases:

LangGraph can be used for various applications, including:

- Conversational agents
- Complex task automation
- Custom LLM-backed experiences

5. Integration:

LangGraph works in conjunction with LangSmith, another tool by LangChain, to provide an out-of-the-box solution for building complex, production-ready features with LLMs.

6. Significance:

LangGraph is described as setting the foundation for building and scaling AI workloads. It's positioned as a key tool in the next chapter of LLM-based application development, particularly in the realm of agentic AI.

7. Availability:

LangGraph is open-source and available on GitHub, which suggests that developers can access and contribute to its codebase.

8. Comparison to Other Frameworks:

LangGraph is noted to offer unique benefits compared to other LLM frameworks, particularly in its ability to handle cycles, provide controllability, and maintain persistence.

LangGraph appears to be a significant tool in the evolving landscape of LLM-based application development, offering developers new ways to create more complex, stateful, and interactive AI systems.

Goodbye!

Congrats! You've created a conversational agent in langgraph that can use a search engine to retrieve updated information when needed. Now it can handle a wider range of user queries. To inspect all the steps your agent just took, check out this [LangSmith trace](#).

Our chatbot still can't remember past interactions on its own, limiting its ability to have coherent, multi-turn conversations. In the next part, we'll add **memory** to address this.

The full code for the graph we've created in this section is reproduced below, replacing our `BasicToolNode` for the prebuilt `ToolNode`, and our `route_tools` condition with the prebuilt `tools_condition`



Full Code



```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
# Any time a tool is called, we return to the chatbot to decide the next step
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
graph = graph_builder.compile()
```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [BaseMessage](#) | [StateGraph](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#)

Part 3: Adding Memory to the Chatbot

Our chatbot can now use tools to answer user questions, but it doesn't remember the context of previous interactions. This limits its ability to have coherent, multi-turn conversations.

LangGraph solves this problem through **persistent checkpointing**. If you provide a `checkpointer` when compiling the graph and a `thread_id` when calling your graph, LangGraph automatically saves the state after each step. When you invoke the graph again using the same `thread_id`, the graph loads its saved state, allowing the chatbot to pick up where it left off.

We will see later that **checkpointing** is *much* more powerful than simple chat memory - it lets you save and resume complex state at any time for error recovery, human-in-the-loop workflows, time travel interactions, and more. But before we get too ahead of ourselves, let's add checkpointing to enable multi-turn conversations.

To get started, create a `MemorySaver` checkpointer.

```
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()
```

API Reference: [MemorySaver](#)

Notice we're using an in-memory checkpointer. This is convenient for our tutorial (it saves it all in-memory). In a production application, you would likely change this to use `SqliteSaver` or `PostgresSaver` and connect to your own DB.

Next define the graph. Now that you've already built your own `BasicToolNode`, we'll replace it with LangGraph's prebuilt `ToolNode` and `tools_condition`, since these do some nice things like parallel API execution. Apart from that, the following is all copied from Part 2.

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)
```

```

tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
# Any time a tool is called, we return to the chatbot to decide the next step
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [BaseMessage](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#)

Finally, compile the graph with the provided checkpointer.

```
graph = graph_builder.compile(checkpointer=memory)
```

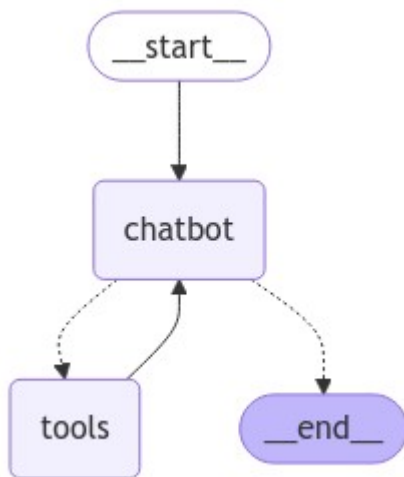
Notice the connectivity of the graph hasn't changed since Part 2. All we are doing is checkpointing the `State` as the graph works through each node.

```

from IPython.display import Image, display

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

```



Now you can interact with your bot! First, pick a thread to use as the key for this conversation.

```
config = {"configurable": {"thread_id": "1"}}
```

Next, call your chat bot.

```
user_input = "Hi there! My name is Will."

# The config is the **second positional argument** to stream() or invoke()!
events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    config,
    stream_mode="values",
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
===== [1m Human Message
[0m=====

Hi there! My name is Will.
===== [1m Ai Message
[0m=====

Hello Will! It's nice to meet you. How can I assist you today? Is there
anything specific you'd like to know or discuss?
```

Note: The config was provided as the **second positional argument** when calling our graph. It importantly is *not* nested within the graph inputs ({'messages': []}).

Let's ask a followup: see if it remembers your name.

```
user_input = "Remember my name?"

# The config is the **second positional argument** to stream() or invoke()!
```

```
events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    config,
    stream_mode="values",
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
===== [1m Human Message
[0m=====

Remember my name?
===== [1m Ai Message
[0m=====

Of course, I remember your name, Will. I always try to pay attention to
important details that users share with me. Is there anything else you'd like
to talk about or any questions you have? I'm here to help with a wide range of
topics or tasks.
```

Notice that we aren't using an external list for memory: it's all handled by the checkpointer! You can inspect the full execution in this [LangSmith trace](#) to see what's going on.

Don't believe me? Try this using a different config.

```
# The only difference is we change the `thread_id` here to "2" instead of "1"
events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    {"configurable": {"thread_id": "2"}},
    stream_mode="values",
)
for event in events:
    event["messages"][-1].pretty_print()
```

```
===== [1m Human Message
[0m=====

Remember my name?
===== [1m Ai Message
[0m=====

I apologize, but I don't have any previous context or memory of your name. As
an AI assistant, I don't retain information from past conversations. Each
interaction starts fresh. Could you please tell me your name so I can address
you properly in this conversation?
```

Notice that the **only** change we've made is to modify the `thread_id` in the config. See this call's [LangSmith trace](#) for comparison.

By now, we have made a few checkpoints across two different threads. But what goes into a checkpoint? To inspect a graph's `state` for a given config at any time, call `get_state(config)`.


```
snapshot = graph.get_state(config)
snapshot
```

```
StateSnapshot(values={'messages': [HumanMessage(content='Hi there! My name is Will.', additional_kwargs={}, response_metadata={}, id='8c1ca919-c553-4ebf-95d4-b59a2d61e078'), AIMessage(content='Hello Will! It's nice to meet you. How can I assist you today? Is there anything specific you'd like to know or discuss?', additional_kwargs={}, response_metadata={'id': 'msg_01WTQebPhNwmMrrmWojJ9KXJ', 'model': 'claude-3-5-sonnet-20240620', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 405, 'output_tokens': 32}}, id='run-58587b77-8c82-41e6-8a90-d62c444a261d-0', usage_metadata={'input_tokens': 405, 'output_tokens': 32, 'total_tokens': 437}), HumanMessage(content='Remember my name?', additional_kwargs={}, response_metadata={}, id='daba7df6-ad75-4d6b-8057-745881cea1ca'), AIMessage(content='Of course, I remember your name, Will. I always try to pay attention to important details that users share with me. Is there anything else you'd like to talk about or any questions you have? I'm here to help with a wide range of topics or tasks.', additional_kwargs={}, response_metadata={'id': 'msg_01E41KitY74HpENRgXx94vag', 'model': 'claude-3-5-sonnet-20240620', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 444, 'output_tokens': 58}}, id='run-ffeaee5c-4d2d-4ddb-bd59-5d5cbf2a5af8-0', usage_metadata={'input_tokens': 444, 'output_tokens': 58, 'total_tokens': 502})]], next=(), config={'configurable': {'thread_id': '1', 'checkpoint_ns': '', 'checkpoint_id': '1ef7d06e-93e0-6acc-8004-f2ac846575d2'}}, metadata={'source': 'loop', 'writes': {'chatbot': {'messages': [AIMessage(content='Of course, I remember your name, Will. I always try to pay attention to important details that users share with me. Is there anything else you'd like to talk about or any questions you have? I'm here to help with a wide range of topics or tasks.', additional_kwargs={}, response_metadata={'id': 'msg_01E41KitY74HpENRgXx94vag', 'model': 'claude-3-5-sonnet-20240620', 'stop_reason': 'end_turn', 'stop_sequence': None, 'usage': {'input_tokens': 444, 'output_tokens': 58}}, id='run-ffeaee5c-4d2d-4ddb-bd59-5d5cbf2a5af8-0', usage_metadata={'input_tokens': 444, 'output_tokens': 58, 'total_tokens': 502})]]}, 'step': 4, 'parents': {}}, created_at='2024-09-27T19:30:10.820758+00:00', parent_config={'configurable': {'thread_id': '1', 'checkpoint_ns': '', 'checkpoint_id': '1ef7d06e-859f-6206-8003-e1bd3c264b8f'}}, tasks=())
```

```
snapshot.next # (since the graph ended this turn, `next` is empty. If you
fetch a state from within a graph invocation, next tells which node will
execute next)
```

```
()
```

The snapshot above contains the current state values, corresponding config, and the `next` node to process. In our case, the graph has reached an `END` state, so `next` is empty.

Congratulations! Your chatbot can now maintain conversation state across sessions thanks to LangGraph's checkpointing system. This opens up exciting possibilities for more natural, contextual interactions. LangGraph's checkpointing even handles **arbitrarily complex graph states**, which is much more expressive and powerful than simple chat memory.

In the next part, we'll introduce human oversight to our bot to handle situations where it may need guidance or verification before proceeding.

Check out the code snippet below to review our graph from this section.



Full Code



```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.set_entry_point("chatbot")
memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)
```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [BaseMessage](#) | [MemorySaver](#) | [StateGraph](#) | [add_messages](#) | [ToolNode](#)

Part 4: Human-in-the-loop

Agents can be unreliable and may need human input to successfully accomplish tasks. Similarly, for some actions, you may want to require human approval before running to ensure that everything is running as intended.

LangGraph's [persistence](#) layer supports human-in-the-loop workflows, allowing execution to pause and resume based on user feedback. The primary interface to this functionality is the [interrupt](#) function. Calling `interrupt` inside a node will pause execution. Execution can be resumed, together with new input from a human, by passing in a [Command](#). `interrupt` is ergonomically similar to Python's built-in `input()`, [with some caveats](#). We demonstrate an example below.

First, start with our existing code from Part 3. We will make one change, which is to add a simple `human_assistance` tool accessible to the chatbot. This tool uses `interrupt` to receive information from a human.

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.tools import tool
from typing_extensions import TypedDict

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition

from langgraph.types import Command, interrupt


class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)


@tool
def human_assistance(query: str) -> str:
    """Request assistance from a human."""
    human_response = interrupt({"query": query})
    return human_response["data"]

tool = TavilySearchResults(max_results=2)
tools = [tool, human_assistance]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    message = llm_with_tools.invoke(state["messages"])
    # Because we will be interrupting during tool execution,
```

```
# we disable parallel tool calling to avoid repeating any
# tool invocations when we resume.
assert len(message.tool_calls) <= 1
return {"messages": [message]}
```

```
graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=tools)
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")
```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [tool](#) | [MemorySaver](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#) | [Command](#) | [interrupt](#)



Tip

Check out the [Human-in-the-loop section](#) of the How-to Guides for more examples of Human-in-the-loop workflows, including how to [review and edit tool calls](#) before they are executed.

We compile the graph with a checkpointer, as before:

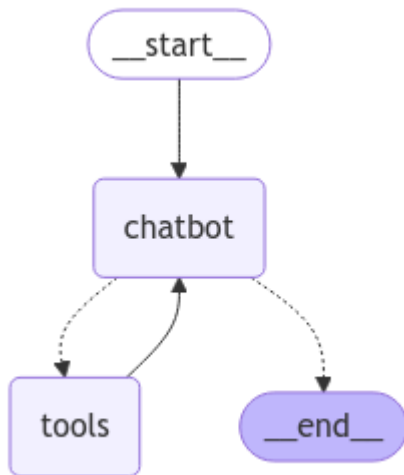
```
memory = MemorySaver()

graph = graph_builder.compile(checkpointer=memory)
```

Visualizing the graph, we recover the same layout as before. We have just added a tool!

```
from IPython.display import Image, display

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



Let's now prompt the chatbot with a question that will engage the new `human_assistance` tool:

```

user_input = "I need some expert guidance for building an AI agent. Could you request assistance for me?"
config = {"configurable": {"thread_id": "1"}}

events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    config,
    stream_mode="values",
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
  
```

```

===== [1m Human Message
[0m=====

I need some expert guidance for building an AI agent. Could you request
assistance for me?
===== [1m Ai Message
[0m=====

[{'text': "Certainly! I'd be happy to request expert assistance for you
regarding building an AI agent. To do this, I'll use the human_assistance
function to relay your request. Let me do that for you now.", 'type': 'text'},
{'id': 'toolu_01ABUqneqnuHNuo1vhfDFQCW', 'input': {'query': 'A user is
requesting expert guidance for building an AI agent. Could you please provide
some expert advice or resources on this topic?'}, 'name': 'human_assistance',
'type': 'tool_use'}]
Tool Calls:
  human_assistance (toolu_01ABUqneqnuHNuo1vhfDFQCW)
Call ID: toolu_01ABUqneqnuHNuo1vhfDFQCW
Args:
  query: A user is requesting expert guidance for building an AI agent.
Could you please provide some expert advice or resources on this topic?
  
```

The chatbot generated a tool call, but then execution has been interrupted! Note that if we inspect the graph state, we see that it stopped at the tools node:

```
snapshot = graph.get_state(config)
snapshot.next
```

```
('tools',)
```

Let's take a closer look at the `human_assistance` tool:

```
@tool
def human_assistance(query: str) -> str:
    """Request assistance from a human."""
    human_response = interrupt({"query": query})
    return human_response["data"]
```

Similar to Python's built-in `input()` function, calling `interrupt` inside the tool will pause execution. Progress is persisted based on our choice of `checkpointer`--so if we are persisting with Postgres, we can resume at any time as long as the database is alive. Here we are persisting with the in-memory checkpointer, so we can resume any time as long as our Python kernel is running.

To resume execution, we pass a `Command` object containing data expected by the tool. The format of this data can be customized based on our needs. Here, we just need a dict with a key `"data"`:

```
human_response = (
    "We, the experts are here to help! We'd recommend you check out LangGraph to build your agent."
    " It's much more reliable and extensible than simple autonomous agents."
)

human_command = Command(resume={"data": human_response})

events = graph.stream(human_command, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
===== [1m Ai Message
[0m=====

[{'text': "Certainly! I'd be happy to request expert assistance for you regarding building an AI agent. To do this, I'll use the human_assistance function to relay your request. Let me do that for you now.", 'type': 'text'}, {'id': 'toolu_01ABUqneqnuHNuo1vhfDFQCW', 'input': {'query': 'A user is requesting expert guidance for building an AI agent. Could you please provide some expert advice or resources on this topic?'}, 'name': 'human_assistance', 'type': 'tool_use'}]
Tool Calls:
```

```

human_assistance (toolu_01ABUqneqnuHNuo1vhfDFQCW)
Call ID: toolu_01ABUqneqnuHNuo1vhfDFQCW
Args:
  query: A user is requesting expert guidance for building an AI agent.
  Could you please provide some expert advice or resources on this topic?
===== [1m Tool Message
[0m=====
Name: human_assistance

We, the experts are here to help! We'd recommend you check out LangGraph to
build your agent. It's much more reliable and extensible than simple
autonomous agents.
===== [1m Ai Message
[0m=====

Thank you for your patience. I've received some expert advice regarding your
request for guidance on building an AI agent. Here's what the experts have
suggested:

The experts recommend that you look into LangGraph for building your AI agent.
They mention that LangGraph is a more reliable and extensible option compared
to simple autonomous agents.

LangGraph is likely a framework or library designed specifically for creating
AI agents with advanced capabilities. Here are a few points to consider based
on this recommendation:

1. Reliability: The experts emphasize that LangGraph is more reliable than
simpler autonomous agent approaches. This could mean it has better stability,
error handling, or consistent performance.

2. Extensibility: LangGraph is described as more extensible, which suggests
that it probably offers a flexible architecture that allows you to easily add
new features or modify existing ones as your agent's requirements evolve.

3. Advanced capabilities: Given that it's recommended over "simple autonomous
agents," LangGraph likely provides more sophisticated tools and techniques for
building complex AI agents.

To get started with LangGraph, you might want to:

1. Search for the official LangGraph documentation or website to learn more
about its features and how to use it.
2. Look for tutorials or guides specifically focused on building AI agents
with LangGraph.
3. Check if there are any community forums or discussion groups where you can
ask questions and get support from other developers using LangGraph.

If you'd like more specific information about LangGraph or have any questions
about this recommendation, please feel free to ask, and I can request further
assistance from the experts.

```

Our input has been received and processed as a tool message. Review this call's [LangSmith trace](#) to see the exact work that was done in the above call. Notice that the state is loaded in the first step so that our chatbot can continue where it left off.

Congrats! You've used an `interrupt` to add human-in-the-loop execution to your chatbot, allowing for human oversight and intervention when needed. This opens up the potential UIs you can create with your AI systems. Since we have already added a **checkpoint**, as long as the underlying persistence layer is running, the graph can be paused **indefinitely** and resumed at any time as if nothing had happened.

Human-in-the-loop workflows enable a variety of new workflows and user experiences. Check out [this section](#) of the How-to Guides for more examples of Human-in-the-loop workflows, including how to [review and edit tool calls](#) before they are executed.



Full Code



```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.tools import tool
from typing_extensions import TypedDict

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.types import Command, interrupt

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

@tool
def human_assistance(query: str) -> str:
    """Request assistance from a human."""
    human_response = interrupt({"query": query})
    return human_response["data"]

tool = TavilySearchResults(max_results=2)
tools = [tool, human_assistance]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    message = llm_with_tools.invoke(state["messages"])
    assert(len(message.tool_calls) <= 1)
    return {"messages": [message]}

graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=tools)
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)
```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [tool](#) | [MemorySaver](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#) | [Command](#) | [interrupt](#)

Part 5: Customizing State

So far, we've relied on a simple state with one entry--a list of messages. You can go far with this simple state, but if you want to define complex behavior without relying on the message list, you can add additional fields to the state. Here we will demonstrate a new scenario, in which the chatbot is using its search tool to find specific information, and forwarding them to a human for review. Let's have the chatbot research the birthday of an entity. We will add `name` and `birthday` keys to the state:

```
from typing import Annotated

from typing_extensions import TypedDict

from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list, add_messages]
    name: str
    birthday: str
```

API Reference: [add_messages](#)

Adding this information to the state makes it easily accessible by other graph nodes (e.g., a downstream node that stores or processes the information), as well as the graph's persistence layer.

Here, we will populate the state keys inside of our `human_assistance` tool. This allows a human to review the information before it is stored in the state. We will again use `Command`, this time to issue a state update from inside our tool. Read more about use cases for `Command` [here](#).

```
from langchain_core.messages import ToolMessage
from langchain_core.tools import InjectedToolCallId, tool

from langgraph.types import Command, interrupt

@tool
# Note that because we are generating a ToolMessage for a state update, we
# generally require the ID of the corresponding tool call. We can use
# LangChain's InjectedToolCallId to signal that this argument should not
# be revealed to the model in the tool's schema.
```

```
def human_assistance(
    name: str, birthday: str, tool_call_id: Annotated[str, InjectedToolCallId]
) -> str:
    """Request assistance from a human."""
    human_response = interrupt(
        {
            "question": "Is this correct?",
            "name": name,
            "birthday": birthday,
        },
    )
    # If the information is correct, update the state as-is.
    if human_response.get("correct", "").lower().startswith("y"):
        verified_name = name
        verified_birthday = birthday
        response = "Correct"
    # Otherwise, receive information from the human reviewer.
    else:
        verified_name = human_response.get("name", name)
        verified_birthday = human_response.get("birthday", birthday)
        response = f"Made a correction: {human_response}"

    # This time we explicitly update the state with a ToolMessage inside
    # the tool.
    state_update = {
        "name": verified_name,
        "birthday": verified_birthday,
        "messages": [ToolMessage(response, tool_call_id=tool_call_id)],
    }
    # We return a Command object in the tool to update our state.
    return Command(update=state_update)
```

API Reference: [ToolMessage](#) | [InjectedToolCallId](#) | [tool](#) | [Command](#) | [interrupt](#)

Otherwise, the rest of our graph is the same:

```
from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode, tools_condition

tool = TavilySearchResults(max_results=2)
tools = [tool, human_assistance]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    message = llm_with_tools.invoke(state["messages"])
    assert len(message.tool_calls) <= 1
    return {"messages": [message]}
```

```

graph_builder = StateGraph(State)
graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=tools)
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)

```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [MemorySaver](#) | [StateGraph](#) | [START](#) | [END](#) | [ToolNode](#) | [tools_condition](#)

Let's prompt our application to look up the "birthday" of the LangGraph library. We will direct the chatbot to reach out to the `human_assistance` tool once it has the required information. Note that setting `name` and `birthday` in the arguments for the tool, we force the chatbot to generate proposals for these fields.

```

user_input = (
    "Can you look up when LangGraph was released? "
    "When you have the answer, use the human_assistance tool for review."
)
config = {"configurable": {"thread_id": "1"}}

events = graph.stream(
    {"messages": [{"role": "user", "content": user_input}]},
    config,
    stream_mode="values",
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()

```

```

===== [1m Human Message
[0m=====

Can you look up when LangGraph was released? When you have the answer, use the
human_assistance tool for review.
===== [1m Ai Message
[0m=====

[{'text': "Certainly! I'll start by searching for information about
LangGraph's release date using the Tavily search function. Then, I'll use the
human_assistance tool for review.", 'type': 'text'}, {'id':
'toolu_01JoXQPgTVJXiuma8xMVwqAi', 'input': {'query': 'LangGraph release
date'}, 'name': 'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
    tavily_search_results_json (toolu_01JoXQPgTVJXiuma8xMVwqAi)

```

```

Call ID: toolu_01JoXPgTVJXiuma8xMVwqAi
Args:
  query: LangGraph release date
===== [1m Tool Message
[0m=====
Name: tavily_search_results_json

[{"url": "https://blog.langchain.dev/langgraph-cloud/", "content": "We also
have a new stable release of LangGraph. By LangChain 6 min read Jun 27, 2024
(Oct '24) Edit: Since the launch of LangGraph Cloud, we now have multiple
deployment options alongside LangGraph Studio - which now fall under LangGraph
Platform. LangGraph Cloud is synonymous with our Cloud SaaS deployment
option."}, {"url": "https://changelog.langchain.com/announcements/langgraph-
cloud-deploy-at-scale-monitor-carefully-iterate-boldly", "content": "LangChain
- Changelog | 🚀 LangGraph Cloud: Deploy at scale, monitor LangChain
LangSmith LangGraph LangChain LangSmith LangGraph LangChain LangSmith
LangGraph LangChain Changelog Sign up for our newsletter to stay up to date
DATE: The LangChain Team LangGraph LangGraph Cloud 🚀 LangGraph Cloud:
Deploy at scale, monitor carefully, iterate boldly DATE: June 27, 2024 AUTHOR:
The LangChain Team LangGraph Cloud is now in closed beta, offering scalable,
fault-tolerant deployment for LangGraph agents. LangGraph Cloud also includes
a new playground-like studio for debugging agent failure modes and quick
iteration: Join the waitlist today for LangGraph Cloud. And to learn more,
read our blog post announcement or check out our docs. Subscribe By clicking
subscribe, you accept our privacy policy and terms and conditions."}]
===== [1m Ai Message
[0m=====

[{'text': "Based on the search results, it appears that LangGraph was already
in existence before June 27, 2024, when LangGraph Cloud was announced.
However, the search results don't provide a specific release date for the
original LangGraph. \n\nGiven this information, I'll use the human_assistance
tool to review and potentially provide more accurate information about
LangGraph's initial release date.", 'type': 'text'}, {'id':
'toolu_01JDQAV7nPqMkHHhNs3j3XoN', 'input': {'name': 'Assistant', 'birthday':
'2023-01-01'}, 'name': 'human_assistance', 'type': 'tool_use'}]
Tool Calls:
  human_assistance (toolu_01JDQAV7nPqMkHHhNs3j3XoN)
Call ID: toolu_01JDQAV7nPqMkHHhNs3j3XoN
Args:
  name: Assistant
  birthday: 2023-01-01

```

We've hit the `interrupt` in the `human_assistance` tool again. In this case, the chatbot failed to identify the correct date, so we can supply it:



```

human_command = Command(
    resume={
        "name": "LangGraph",
        "birthday": "Jan 17, 2024",
    },
)

events = graph.stream(human_command, config, stream_mode="values")
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()

```

```

===== [1m Ai Message
[0m=====

[{'text': "Based on the search results, it appears that LangGraph was already
in existence before June 27, 2024, when LangGraph Cloud was announced.
However, the search results don't provide a specific release date for the
original LangGraph. \n\nGiven this information, I'll use the human_assistance
tool to review and potentially provide more accurate information about
LangGraph's initial release date.", 'type': 'text'}, {'id':
'toolu_01JDQAV7nPqMkHHhNs3j3XoN', 'input': {'name': 'Assistant', 'birthday':
'2023-01-01'}, 'name': 'human_assistance', 'type': 'tool_use'}}]
Tool Calls:
  human_assistance (toolu_01JDQAV7nPqMkHHhNs3j3XoN)
Call ID: toolu_01JDQAV7nPqMkHHhNs3j3XoN
Args:
  name: Assistant
  birthday: 2023-01-01
===== [1m Tool Message
[0m=====
Name: human_assistance

Made a correction: {'name': 'LangGraph', 'birthday': 'Jan 17, 2024'}
===== [1m Ai Message
[0m=====

Thank you for the human assistance. I can now provide you with the correct
information about LangGraph's release date.

LangGraph was initially released on January 17, 2024. This information comes
from the human assistance correction, which is more accurate than the search
results I initially found.

To summarize:
1. LangGraph's original release date: January 17, 2024
2. LangGraph Cloud announcement: June 27, 2024

It's worth noting that LangGraph had been in development and use for some time
before the LangGraph Cloud announcement, but the official initial release of
LangGraph itself was on January 17, 2024.

```

Note that these fields are now reflected in the state:

```
snapshot = graph.get_state(config)
```

```
{k: v for k, v in snapshot.values.items() if k in ("name", "birthday")}
```

```
{'name': 'LangGraph', 'birthday': 'Jan 17, 2024'}
```

This makes them easily accessible to downstream nodes (e.g., a node that further processes or stores the information).

Manually updating state

LangGraph gives a high degree of control over the application state. For instance, at any point (including when interrupted), we can manually override a key using

`graph.update_state`:

```
graph.update_state(config, {"name": "LangGraph (library)"})
```

```
{'configurable': {'thread_id': '1',
  'checkpoint_ns': '',
  'checkpoint_id': '1efd4ec5-cf69-6352-8006-9278f1730162'}}
```

If we call `graph.get_state`, we can see the new value is reflected:

```
snapshot = graph.get_state(config)
```

```
{k: v for k, v in snapshot.values.items() if k in ("name", "birthday")}
```

```
{'name': 'LangGraph (library)', 'birthday': 'Jan 17, 2024'}
```

Manual state updates will even [generate a trace](#) in LangSmith. If desired, they can also be used to control human-in-the-loop workflows, as described in [this guide](#). Use of the `interrupt` function is generally recommended instead, as it allows data to be transmitted in a human-in-the-loop interaction independently of state updates.

Congratulations! You've added custom keys to the state to facilitate a more complex workflow, and learned how to generate state updates from inside tools.

We're almost done with the tutorial, but there is one more concept we'd like to review before finishing that connects `checkpointing` and `state updates`.

This section's code is reproduced below for your reference.



Full Code



```

from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.messages import ToolMessage
from langchain_core.tools import InjectedToolCallId, tool
from typing_extensions import TypedDict

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.types import Command, interrupt

class State(TypedDict):
    messages: Annotated[list, add_messages]
    name: str
    birthday: str

@tool
def human_assistance(
    name: str, birthday: str, tool_call_id: Annotated[str, InjectedToolCallId]
) -> str:
    """Request assistance from a human."""
    human_response = interrupt(
        {
            "question": "Is this correct?",
            "name": name,
            "birthday": birthday,
        },
    )
    if human_response.get("correct", "").lower().startswith("y"):
        verified_name = name
        verified_birthday = birthday
        response = "Correct"
    else:
        verified_name = human_response.get("name", name)
        verified_birthday = human_response.get("birthday", birthday)
        response = f"Made a correction: {human_response}"

    state_update = {
        "name": verified_name,
        "birthday": verified_birthday,
        "messages": [ToolMessage(response, tool_call_id=tool_call_id)],
    }
    return Command(update=state_update)

tool = TavilySearchResults(max_results=2)
tools = [tool, human_assistance]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

```

```
def chatbot(state: State):
    message = llm_with_tools.invoke(state["messages"])
    assert(len(message.tool_calls) <= 1)
    return {"messages": [message]}

graph_builder = StateGraph(State)
graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=tools)
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)
```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [ToolMessage](#) | [InjectedToolCallId](#) | [tool](#) | [MemorySaver](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#) | [Command](#) | [interrupt](#)

Part 6: Time Travel

In a typical chat bot workflow, the user interacts with the bot 1 or more times to accomplish a task. In the previous sections, we saw how to add memory and a human-in-the-loop to be able to checkpoint our graph state and control future responses.

But what if you want to let your user start from a previous response and "branch off" to explore a separate outcome? Or what if you want users to be able to "rewind" your assistant's work to fix some mistakes or try a different strategy (common in applications like autonomous software engineers)?

You can create both of these experiences and more using LangGraph's built-in "time travel" functionality.

In this section, you will "rewind" your graph by fetching a checkpoint using the graph's `get_state_history` method. You can then resume execution at this previous point in time.

For this, let's use the simple chatbot with tools from [Part 3](#):

```
from typing import Annotated

from langchain_anthropic import ChatAnthropic
from langchain_community.tools.tavily_search import TavilySearchResults
```

```

from langchain_core.messages import BaseMessage
from typing_extensions import TypedDict

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from langgraph.prebuilt import ToolNode, tools_condition

class State(TypedDict):
    messages: Annotated[list, add_messages]

graph_builder = StateGraph(State)

tool = TavilySearchResults(max_results=2)
tools = [tool]
llm = ChatAnthropic(model="claude-3-5-sonnet-20240620")
llm_with_tools = llm.bind_tools(tools)

def chatbot(state: State):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

graph_builder.add_node("chatbot", chatbot)

tool_node = ToolNode(tools=[tool])
graph_builder.add_node("tools", tool_node)

graph_builder.add_conditional_edges(
    "chatbot",
    tools_condition,
)
graph_builder.add_edge("tools", "chatbot")
graph_builder.add_edge(START, "chatbot")

memory = MemorySaver()
graph = graph_builder.compile(checkpointer=memory)

```

API Reference: [ChatAnthropic](#) | [TavilySearchResults](#) | [BaseMessage](#) | [MemorySaver](#) | [StateGraph](#) | [START](#) | [END](#) | [add_messages](#) | [ToolNode](#) | [tools_condition](#)

Let's have our graph take a couple steps. Every step will be checkpointed in its state history:

```

config = {"configurable": {"thread_id": "1"}}
events = graph.stream(
    {
        "messages": [
            {
                "role": "user",
                "content": (
                    "I'm learning LangGraph. "

```

```

        "Could you do some research on it for me?"
    ),
    },
],
},
config,
stream_mode="values",
)
for event in events:
    if "messages" in event:
        event["messages"][-1].pretty_print()

```

```

===== [1m Human Message
[0m=====

I'm learning LangGraph. Could you do some research on it for me?
===== [1m Ai Message
[0m=====

[{'text': "Certainly! I'd be happy to research LangGraph for you. To get the
most up-to-date and accurate information, I'll use the Tavily search engine to
look this up. Let me do that for you now.", 'type': 'text'}, {'id':
'toolu_01BscbfJJB9EWJFqGrN6E54e', 'input': {'query': 'LangGraph latest
information and features'}, 'name': 'tavily_search_results_json', 'type':
'tool_use'}}]
Tool Calls:
  tavily_search_results_json (toolu_01BscbfJJB9EWJFqGrN6E54e)
Call ID: toolu_01BscbfJJB9EWJFqGrN6E54e
Args:
  query: LangGraph latest information and features
===== [1m Tool Message
[0m=====
Name: tavily_search_results_json

[{"url": "https://blockchain.news/news/langchain-new-features-upcoming-events-
update", "content": "LangChain, a leading platform in the AI development
space, has released its latest updates, showcasing new use cases and
enhancements across its ecosystem. According to the LangChain Blog, the
updates cover advancements in LangGraph Cloud, LangSmith's self-improving
evaluators, and revamped documentation for LangGraph."}, {"url":
"https://blog.langchain.dev/langgraph-platform-announce/", "content": "With
these learnings under our belt, we decided to couple some of our latest
offerings under LangGraph Platform. LangGraph Platform today includes
LangGraph Server, LangGraph Studio, plus the CLI and SDK. ... we added
features in LangGraph Server to deliver on a few key value areas. Below, we'll
focus on these aspects of LangGraph Platform."}]
===== [1m Ai Message
[0m=====

Thank you for your patience. I've found some recent information about
LangGraph for you. Let me summarize the key points:

1. LangGraph is part of the LangChain ecosystem, which is a leading platform
in AI development.

2. Recent updates and features of LangGraph include:

```

a. LangGraph Cloud: This seems to be a cloud-based version of LangGraph, though specific details weren't provided in the search results.

b. LangGraph Platform: This is a newly introduced concept that combines several offerings:

- LangGraph Server
- LangGraph Studio
- CLI (Command Line Interface)
- SDK (Software Development Kit)

3. LangGraph Server: This component has received new features to enhance its value proposition, though the specific features weren't detailed in the search results.

4. LangGraph Studio: This appears to be a new tool in the LangGraph ecosystem, likely providing a graphical interface for working with LangGraph.

5. Documentation: The LangGraph documentation has been revamped, which should make it easier for learners like yourself to understand and use the tool.

6. Integration with LangSmith: While not directly part of LangGraph, LangSmith (another tool in the LangChain ecosystem) now features self-improving evaluators, which might be relevant if you're using LangGraph as part of a larger LangChain project.

As you're learning LangGraph, it would be beneficial to:

1. Check out the official LangChain documentation, especially the newly revamped LangGraph sections.
2. Explore the different components of the LangGraph Platform (Server, Studio, CLI, and SDK) to see which best fits your learning needs.
3. Keep an eye on LangGraph Cloud developments, as cloud-based solutions often provide an easier starting point for learners.
4. Consider how LangGraph fits into the broader LangChain ecosystem, especially its interaction with tools like LangSmith.

Is there any specific aspect of LangGraph you'd like to know more about? I'd be happy to do a more focused search on particular features or use cases.

```
events = graph.stream(
    {
        "messages": [
            {
                "role": "user",
                "content": (
                    "Ya that's helpful. Maybe I'll "
                    "build an autonomous agent with it!"
                ),
            },
        ],
    },
    config,
    stream_mode="values",
)
for event in events:
```

```
if "messages" in event:
    event["messages"][-1].pretty_print()
```

```
===== [1m Human Message
[0m=====

Ya that's helpful. Maybe I'll build an autonomous agent with it!
===== [1m Ai Message
[0m=====

[{'text': "That's an exciting idea! Building an autonomous agent with
LangGraph is indeed a great application of this technology. LangGraph is
particularly well-suited for creating complex, multi-step AI workflows, which
is perfect for autonomous agents. Let me gather some more specific information
about using LangGraph for building autonomous agents.", 'type': 'text'},
{'id': 'toolu_01QWNHhUaeeWcGXvA4eHT7Zo', 'input': {'query': 'Building
autonomous agents with LangGraph examples and tutorials'}, 'name':
'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
  tavily_search_results_json (toolu_01QWNHhUaeeWcGXvA4eHT7Zo)
Call ID: toolu_01QWNHhUaeeWcGXvA4eHT7Zo
Args:
  query: Building autonomous agents with LangGraph examples and tutorials
===== [1m Tool Message
[0m=====
Name: tavily_search_results_json

[{"url": "https://towardsdatascience.com/building-autonomous-multi-tool-
agents-with-gemini-2-0-and-langgraph-ad3d7bd5e79d", "content": "Building
Autonomous Multi-Tool Agents with Gemini 2.0 and LangGraph | by Youness Mansar
| Jan, 2025 | Towards Data Science Building Autonomous Multi-Tool Agents with
Gemini 2.0 and LangGraph A practical tutorial with full code examples for
building and running multi-tool agents Towards Data Science LLMs are
remarkable – they can memorize vast amounts of information, answer general
knowledge questions, write code, generate stories, and even fix your grammar.
In this tutorial, we are going to build a simple LLM agent that is equipped
with four tools that it can use to answer a user's question. This Agent will
have the following specifications: Follow Published in Towards Data Science --
----- Your home for data science and AI. Follow
Follow Follow"}, {"url": "https://github.com/anmolaman20/Tools_and_Agents",
"content": "GitHub - anmolaman20/Tools_and_Agents: This repository provides
resources for building AI agents using Langchain and Langgraph. This
repository provides resources for building AI agents using Langchain and
Langgraph. This repository provides resources for building AI agents using
Langchain and Langgraph. This repository serves as a comprehensive guide for
building AI-powered agents using Langchain and Langgraph. It provides hands-on
examples, practical tutorials, and resources for developers and AI enthusiasts
to master building intelligent systems and workflows. AI Agent Development:
Gain insights into creating intelligent systems that think, reason, and adapt
in real time. This repository is ideal for AI practitioners, developers
exploring language models, or anyone interested in building intelligent
systems. This repository provides resources for building AI agents using
Langchain and Langgraph."}]
===== [1m Ai Message
[0m=====
```

Great idea! Building an autonomous agent with LangGraph is definitely an exciting project. Based on the latest information I've found, here are some insights and tips for building autonomous agents with LangGraph:

1. **Multi-Tool Agents:** LangGraph is particularly well-suited for creating autonomous agents that can use multiple tools. This allows your agent to have a diverse set of capabilities and choose the right tool for each task.
2. **Integration with Large Language Models (LLMs):** You can combine LangGraph with powerful LLMs like Gemini 2.0 to create more intelligent and capable agents. The LLM can serve as the "brain" of your agent, making decisions and generating responses.
3. **Workflow Management:** LangGraph excels at managing complex, multi-step AI workflows. This is crucial for autonomous agents that need to break down tasks into smaller steps and execute them in the right order.
4. **Practical Tutorials Available:** There are tutorials available that provide full code examples for building and running multi-tool agents. These can be incredibly helpful as you start your project.
5. **Langchain Integration:** LangGraph is often used in conjunction with Langchain. This combination provides a powerful framework for building AI agents, offering features like memory management, tool integration, and prompt management.
6. **GitHub Resources:** There are repositories available (like the one by [anmolaman20](#)) that provide comprehensive resources for building AI agents using Langchain and LangGraph. These can be valuable references as you develop your agent.
7. **Real-time Adaptation:** LangGraph allows you to create agents that can think, reason, and adapt in real-time, which is crucial for truly autonomous behavior.
8. **Customization:** You can equip your agent with specific tools tailored to your use case. For example, you might include tools for web searching, data analysis, or interacting with specific APIs.

To get started with your autonomous agent project:

1. Familiarize yourself with LangGraph's documentation and basic concepts.
2. Look into tutorials that specifically deal with building autonomous agents, like the one mentioned from Towards Data Science.
3. Decide on the specific capabilities you want your agent to have and identify the tools it will need.
4. Start with a simple agent and gradually add complexity as you become more comfortable with the framework.
5. Experiment with different LLMs to find the one that works best for your use case.
6. Pay attention to how you structure the agent's decision-making process and workflow.
7. Don't forget to implement proper error handling and safety measures, especially if your agent will be interacting with external systems or making important decisions.

Building an autonomous agent is an iterative process, so be prepared to refine and improve your agent over time. Good luck with your project! If you need any more specific information as you progress, feel free to ask.

Now that we've had the agent take a couple steps, we can `replay` the full state history to see everything that occurred.

```
to_replay = None
for state in graph.get_state_history(config):
    print("Num Messages: ", len(state.values["messages"]), "Next: ",
          state.next)
    print("-" * 80)
    if len(state.values["messages"]) == 6:
        # We are somewhat arbitrarily selecting a specific state based on the
        # number of chat messages in the state.
        to_replay = state
```

```
Num Messages:  8 Next:  ()
-----
--
Num Messages:  7 Next:  ('chatbot',)
-----
--
Num Messages:  6 Next:  ('tools',)
-----
--
Num Messages:  5 Next:  ('chatbot',)
-----
--
Num Messages:  4 Next:  ('__start__',)
-----
--
Num Messages:  4 Next:  ()
-----
--
Num Messages:  3 Next:  ('chatbot',)
-----
--
Num Messages:  2 Next:  ('tools',)
-----
--
Num Messages:  1 Next:  ('chatbot',)
-----
--
Num Messages:  0 Next:  ('__start__',)
-----
--
```

Notice that checkpoints are saved for every step of the graph. This **spans invocations** so you can rewind across a full thread's history. We've picked out `to_replay` as a state to resume from. This is the state after the `chatbot` node in the second graph invocation above.

Resuming from this point should call the **action** node next.


```
print(to_replay.next)
print(to_replay.config)
```

```
('tools',)
{'configurable': {'thread_id': '1', 'checkpoint_ns': '', 'checkpoint_id':
'1efd43e3-0c1f-6c4e-8006-891877d65740'}}
```

Notice that the checkpoint's config (`to_replay.config`) contains a `checkpoint_id` **timestamp**. Providing this `checkpoint_id` value tells LangGraph's checkpointer to **load** the state from that moment in time. Let's try it below:

```
# The `checkpoint_id` in the `to_replay.config` corresponds to a state we've
persisted to our checkpointer.
for event in graph.stream(None, to_replay.config, stream_mode="values"):
    if "messages" in event:
        event["messages"][-1].pretty_print()
```

```
===== [1m Ai Message
[0m=====

[{'text': "That's an exciting idea! Building an autonomous agent with
LangGraph is indeed a great application of this technology. LangGraph is
particularly well-suited for creating complex, multi-step AI workflows, which
is perfect for autonomous agents. Let me gather some more specific information
about using LangGraph for building autonomous agents.", 'type': 'text'},
{'id': 'toolu_01QWNHhUaeeWcGXvA4eHT7Zo', 'input': {'query': 'Building
autonomous agents with LangGraph examples and tutorials'}, 'name':
'tavily_search_results_json', 'type': 'tool_use'}]
Tool Calls:
  tavily_search_results_json (toolu_01QWNHhUaeeWcGXvA4eHT7Zo)
Call ID: toolu_01QWNHhUaeeWcGXvA4eHT7Zo
Args:
  query: Building autonomous agents with LangGraph examples and tutorials
===== [1m Tool Message
[0m=====
Name: tavily_search_results_json

[{"url": "https://towardsdatascience.com/building-autonomous-multi-tool-
agents-with-gemini-2-0-and-langgraph-ad3d7bd5e79d", "content": "Building
Autonomous Multi-Tool Agents with Gemini 2.0 and LangGraph | by Youness Mansar
| Jan, 2025 | Towards Data Science Building Autonomous Multi-Tool Agents with
Gemini 2.0 and LangGraph A practical tutorial with full code examples for
building and running multi-tool agents Towards Data Science LLMs are
remarkable – they can memorize vast amounts of information, answer general
knowledge questions, write code, generate stories, and even fix your grammar.
In this tutorial, we are going to build a simple LLM agent that is equipped
with four tools that it can use to answer a user's question. This Agent will
have the following specifications: Follow Published in Towards Data Science --
----- Your home for data science and AI. Follow
Follow Follow"}, {"url": "https://github.com/anmolaman20/Tools_and_Agents",
"content": "GitHub - anmolaman20/Tools_and_Agents: This repository provides
resources for building AI agents using Langchain and Langgraph. This
repository provides resources for building AI agents using Langchain and
```

Langgraph. This repository provides resources for building AI agents using Langchain and Langgraph. This repository serves as a comprehensive guide for building AI-powered agents using Langchain and Langgraph. It provides hands-on examples, practical tutorials, and resources for developers and AI enthusiasts to master building intelligent systems and workflows. AI Agent Development: Gain insights into creating intelligent systems that think, reason, and adapt in real time. This repository is ideal for AI practitioners, developers exploring language models, or anyone interested in building intelligent systems. This repository provides resources for building AI agents using Langchain and Langgraph."}]

===== [1m Ai Message
[0m=====

Great idea! Building an autonomous agent with LangGraph is indeed an excellent way to apply and deepen your understanding of the technology. Based on the search results, I can provide you with some insights and resources to help you get started:

1. Multi-Tool Agents:

LangGraph is well-suited for building autonomous agents that can use multiple tools. This allows your agent to have a variety of capabilities and choose the appropriate tool based on the task at hand.

2. Integration with Large Language Models (LLMs):

There's a tutorial that specifically mentions using Gemini 2.0 (Google's LLM) with LangGraph to build autonomous agents. This suggests that LangGraph can be integrated with various LLMs, giving you flexibility in choosing the language model that best fits your needs.

3. Practical Tutorials:

There are tutorials available that provide full code examples for building and running multi-tool agents. These can be invaluable as you start your project, giving you a concrete starting point and demonstrating best practices.

4. GitHub Resources:

There's a GitHub repository (github.com/anmolaman20/Tools_and_Agents) that provides resources for building AI agents using both Langchain and Langgraph. This could be a great resource for code examples, tutorials, and understanding how LangGraph fits into the broader LangChain ecosystem.

5. Real-Time Adaptation:

The resources mention creating intelligent systems that can think, reason, and adapt in real-time. This is a key feature of advanced autonomous agents and something you can aim for in your project.

6. Diverse Applications:

The materials suggest that these techniques can be applied to various tasks, from answering questions to potentially more complex decision-making processes.

To get started with your autonomous agent project using LangGraph, you might want to:

1. Review the tutorials mentioned, especially those with full code examples.
2. Explore the GitHub repository for hands-on examples and resources.
3. Decide on the specific tasks or capabilities you want your agent to have.

4. Choose an LLM to integrate with LangGraph (like GPT, Gemini, or others).
5. Start with a simple agent that uses one or two tools, then gradually expand its capabilities.
6. Implement decision-making logic to help your agent choose between different tools or actions.
7. Test your agent thoroughly with various inputs and scenarios to ensure robust performance.

Remember, building an autonomous agent is an iterative process. Start simple and gradually increase complexity as you become more comfortable with LangGraph and its capabilities.

Would you like more information on any specific aspect of building your autonomous agent with LangGraph?

Notice that the graph resumed execution from the `**action**` node. You can tell this is the case since the first value printed above is the response from our search engine tool.

Congratulations! You've now used time-travel checkpoint traversal in LangGraph. Being able to rewind and explore alternative paths opens up a world of possibilities for debugging, experimentation, and interactive applications.

Next Steps

Take your journey further by exploring deployment and advanced features:

Server Quickstart

- [LangGraph Server Quickstart](#): Launch a LangGraph server locally and interact with it using the REST API and LangGraph Studio Web UI.

LangGraph Cloud

- [LangGraph Cloud QuickStart](#): Deploy your LangGraph app using LangGraph Cloud.

LangGraph Framework

- [LangGraph Concepts](#): Learn the foundational concepts of LangGraph.
- [LangGraph How-to Guides](#): Guides for common tasks with LangGraph.

LangGraph Platform

Expand your knowledge with these resources:

- [LangGraph Platform Concepts](#): Understand the foundational concepts of the LangGraph Platform.

- **LangGraph Platform How-to Guides:** Guides for common tasks with LangGraph Platform.



LangChain Expression Language (LCEL)

❗ PREREQUISITES

- [Runnable Interface](#)

The LangChain Expression Language (LCEL) takes a **declarative** approach to building new **Runnables** from existing Runnables.

This means that you describe what *should* happen, rather than *how* it should happen, allowing LangChain to optimize the run-time execution of the chains.

We often refer to a `Runnable` created using LCEL as a "chain". It's important to remember that a "chain" is `Runnable` and it implements the full **Runnable Interface**.

❗ NOTE

- The [LCEL cheatsheet](#) shows common patterns that involve the Runnable interface and LCEL expressions.
- Please see the following list of [how-to guides](#) that cover common tasks with LCEL.
- A list of built-in `Runnables` can be found in the [LangChain Core API Reference](#). Many of these Runnables are useful when composing custom "chains" in LangChain using LCEL.

Benefits of LCEL

LangChain optimizes the run-time execution of chains built with LCEL in a number of ways:

- **Optimized parallel execution:** Run Runnable in parallel using [RunnableParallel](#) or run multiple inputs through a given chain in parallel using the [Runnable Batch API](#). Parallel execution can significantly reduce the latency as processing can be done in parallel instead of sequentially.
- **Guaranteed Async support:** Any chain built with LCEL can be run asynchronously using the [Runnable Async API](#). This can be useful when running chains in a server environment where you want to handle large number of requests concurrently.
- **Simplify streaming:** LCEL chains can be streamed, allowing for incremental output as the chain is executed. LangChain can optimize the streaming of the output to minimize the time-to-first-token (time elapsed until the first chunk of output from a [chat model](#) or [llm](#) comes out).

Other benefits include:

- **Seamless LangSmith tracing** As your chains get more and more complex, it becomes increasingly important to understand what exactly is happening at every step. With LCEL, all steps are automatically logged to [LangSmith](#) for maximum observability and debuggability.
- **Standard API:** Because all chains are built using the Runnable interface, they can be used in the same way as any other Runnable.
- **Deployable with LangServe:** Chains built with LCEL can be deployed using for production use.

Should I use LCEL?

LCEL is an [orchestration solution](#) -- it allows LangChain to handle run-time execution of chains in an optimized way.

While we have seen users run chains with hundreds of steps in production, we generally recommend using LCEL for simpler orchestration tasks. When the application requires complex state management, branching, cycles or multiple agents, we recommend that users take advantage of [LangGraph](#).

In LangGraph, users define graphs that specify the application's flow. This allows users to keep using LCEL within individual nodes when LCEL is needed, while making it easy to define complex orchestration logic that is more readable and maintainable.

Here are some guidelines:

- If you are making a single LLM call, you don't need LCEL; instead call the underlying `chat model` directly.
- If you have a simple chain (e.g., prompt + llm + parser, simple retrieval set up etc.), LCEL is a reasonable fit, if you're taking advantage of the LCEL benefits.
- If you're building a complex chain (e.g., with branching, cycles, multiple agents, etc.) use `LangGraph` instead. Remember that you can always use LCEL within individual nodes in LangGraph.

Composition Primitives

LCEL chains are built by composing existing `Runnables` together. The two main composition primitives are `RunnableSequence` and `RunnableParallel`.

Many other composition primitives (e.g., `RunnableAssign`) can be thought of as variations of these two primitives.

NOTE

You can find a list of all composition primitives in the [LangChain Core API Reference](#).

RunnableSequence

`RunnableSequence` is a composition primitive that allows you "chain" multiple runnables sequentially, with the output of one runnable serving as the input to the next.

```
from langchain_core.runnables import RunnableSequence
chain = RunnableSequence([runnable1, runnable2])
```

API Reference: [RunnableSequence](#)

Invoking the `chain` with some input:

```
final_output = chain.invoke(some_input)
```

corresponds to the following:

```
output1 = runnable1.invoke(some_input)
final_output = runnable2.invoke(output1)
```

NOTE

`runnable1` and `runnable2` are placeholders for any `Runnable` that you want to chain together.

RunnableParallel

`RunnableParallel` is a composition primitive that allows you to run multiple runnables concurrently, with the same input provided to each.

```
from langchain_core.runnables import RunnableParallel
chain = RunnableParallel({
    "key1": runnable1,
    "key2": runnable2,
})
```

API Reference: [RunnableParallel](#)

Invoking the `chain` with some input:

```
final_output = chain.invoke(some_input)
```

Will yield a `final_output` dictionary with the same keys as the input dictionary, but with the values replaced by the output of the corresponding runnable.


```
{  
    "key1": runnable1.invoke(some_input),  
    "key2": runnable2.invoke(some_input),  
}
```

Recall, that the runnables are executed in parallel, so while the result is the same as dictionary comprehension shown above, the execution time is much faster.

NOTE

`RunnableParallel` supports both synchronous and asynchronous execution (as all `Runnables` do).

- For synchronous execution, `RunnableParallel` uses a `ThreadPoolExecutor` to run the runnables concurrently.
- For asynchronous execution, `RunnableParallel` uses `asyncio.gather` to run the runnables concurrently.

Composition Syntax

The usage of `RunnableSequence` and `RunnableParallel` is so common that we created a shorthand syntax for using them. This helps to make the code more readable and concise.

The `|` operator

We have **overloaded** the `|` operator to create a `RunnableSequence` from two `Runnables`.

```
chain = runnable1 | runnable2
```

is Equivalent to:

```
chain = RunnableSequence([runnable1, runnable2])
```

The `.pipe` method

If you have moral qualms with operator overloading, you can use the `.pipe` method instead. This is equivalent to the `|` operator.

```
chain = runnable1.pipe(runnable2)
```

Coercion

LCEL applies automatic type coercion to make it easier to compose chains.

If you do not understand the type coercion, you can always use the `RunnableSequence` and `RunnableParallel` classes directly.

This will make the code more verbose, but it will also make it more explicit.

Dictionary to RunnableParallel

Inside an LCEL expression, a dictionary is automatically converted to a `RunnableParallel`.

For example, the following code:

```
mapping = {  
    "key1": runnable1,  
    "key2": runnable2,  
}  
  
chain = mapping | runnable3
```

It gets automatically converted to the following:

```
chain = RunnableSequence([RunnableParallel(mapping), runnable3])
```

CAUTION

You have to be careful because the `mapping` dictionary is not a `RunnableParallel` object, it is just a dictionary. This means that the following code will raise an `AttributeError`:

```
mapping.invoke(some_input)
```

Function to RunnableLambda

Inside an LCEL expression, a function is automatically converted to a `RunnableLambda`.

```
def some_func(x):  
    return x  
  
chain = some_func | runnable1
```

It gets automatically converted to the following:

```
chain = RunnableSequence([RunnableLambda(some_func), runnable1])
```

⚠ CAUTION

You have to be careful because the lambda function is not a `RunnableLambda` object, it is just a function. This means that the following code will raise an `AttributeError`:

```
lambda x: x + 1.invoke(some_input)
```

Legacy chains

LCEL aims to provide consistency around behavior and customization over legacy subclassed chains such as `LLMChain` and `ConversationalRetrievalChain`. Many of these legacy chains hide important details like prompts, and as a wider variety of viable models emerge, customization has become more and more important.

If you are currently using one of these legacy chains, please see [this guide for guidance on how to migrate](#).

For guides on how to do specific tasks with LCEL, check out [the relevant how-to guides](#).

[✎ Edit this page](#)

Was this page helpful?





An error occurred: API rate limit exceeded for langchain-ai/langchain. Sign in to increase the

Write

Preview

Aa

Sign in to comment

Sign in with GitHub

This is documentation for LangChain **v0.1**, which is no longer actively maintained.

For the current stable version, see [this version](#) (Latest).

[Composition](#)[Agents](#)[Quickstart](#)

Quickstart

To best understand the agent framework, let's build an agent that has two tools: one to look things up online, and one to look up specific data that we've loaded into a index.

This will assume knowledge of [LLMs](#) and [retrieval](#) so if you haven't already explored those sections, it is recommended you do so.

Setup: LangSmith

By definition, agents take a self-determined, input-dependent sequence of steps before returning a user-facing output. This makes debugging these systems particularly tricky, and observability particularly important. [LangSmith](#) is especially useful for such cases.

When building with LangChain, all steps will automatically be traced in LangSmith. To set up LangSmith we just need set the following environment variables:

```
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_API_KEY="<your-api-key>"
```

Define tools

We first need to create the tools we want to use. We will use two tools: [Tavily](#) (to search online) and then a retriever over a local index we will create

Tavily

We have a built-in tool in LangChain to easily use Tavily search engine as tool. Note that this requires an API key - they have a free tier, but if you don't have one or don't want to create

one, you can always ignore this step.

Once you create your API key, you will need to export that as:

```
export TAVILY_API_KEY="..."
```

```
from langchain_community.tools.tavily_search import  
TavilySearchResults
```

API Reference:

- [TavilySearchResults](#)

```
search = TavilySearchResults()
```

```
search.invoke("what is the weather in SF")
```

```
[{'url': 'https://www.weatherapi.com/',  
  'content': '{"location": {"name": "San Francisco", "region":  
'California', 'country': 'United States of America', 'lat': 37.78,  
'lon': -122.42, 'tz_id': 'America/Los_Angeles', 'localtime_epoch':  
1712847697, 'localtime': '2024-04-11 8:01'}, 'current':  
{'last_updated_epoch': 1712847600, 'last_updated': '2024-04-11 08:00',  
'temp_c': 11.1, 'temp_f': 52.0, 'is_day': 1, 'condition': {'text':  
'Partly cloudy', 'icon':  
'//cdn.weatherapi.com/weather/64x64/day/116.png', 'code': 1003},  
'wind_mph': 2.2, 'wind_kph': 3.6, 'wind_degree': 10, 'wind_dir': 'N',  
'pressure_mb': 1015.0, 'pressure_in': 29.98, 'precip_mm': 0.0,  
'precip_in': 0.0, 'humidity': 97, 'cloud': 25, 'feelslike_c': 11.5,  
'feelslike_f': 52.6, 'vis_km': 14.0, 'vis_miles': 8.0, 'uv': 4.0,  
'gust_mph': 2.8, 'gust_kph': 4.4}}"}},  
 {'url': 'https://www.yahoo.com/news/april-11-2024-san-francisco-  
122026435.html',  
  'content': "2024 NBA Mock Draft 6.0: Projections for every pick  
following March Madness With the NCAA tournament behind us, here's an  
updated look at Yahoo Sports' first- and second-round projections for  
the ..."},  
 {'url': 'https://world-weather.info/forecast/usa/san_francisco/april-  
2024/',  
  'content': 'Extended weather forecast in San Francisco. Hourly Week  
10 days 14 days 30 days Year. Detailed ↗ San Francisco Weather Forecast
```

```
for April 2024 – day/night 🌡️ temperatures, precipitations – World-
Weather.info.'},
{'url': 'https://www.wunderground.com/hourly/us/ca/san-
francisco/94144/date/date/2024-4-11',
'content': 'Personal Weather Station. Inner Richmond (KCASANFR1685)
Location: San Francisco, CA. Elevation: 207ft. Nearby Weather Stations.
Hourly Forecast for Today, Thursday 04/11Hourly for Today, Thu 04/11
...'},
{'url': 'https://weatherspark.com/h/y/557/2024/Historical-Weather-
during-2024-in-San-Francisco-California-United-States',
'content': 'San Francisco Temperature History 2024\nHourly
Temperature in 2024 in San Francisco\nCompare San Francisco to another
city:\nCloud Cover in 2024 in San Francisco\nDaily Precipitation in
2024 in San Francisco\nObserved Weather in 2024 in San Francisco\nHours
of Daylight and Twilight in 2024 in San Francisco\nSunrise & Sunset
with Twilight and Daylight Saving Time in 2024 in San Francisco\nSolar
Elevation and Azimuth in 2024 in San Francisco\nMoon Rise, Set & Phases
in 2024 in San Francisco\nHumidity Comfort Levels in 2024 in San
Francisco\nWind Speed in 2024 in San Francisco\nHourly Wind Speed in
2024 in San Francisco\nHourly Wind Direction in 2024 in San
Francisco\nAtmospheric Pressure in 2024 in San Francisco\nData
Sources\n See all nearby weather stations\nLatest Report – 3:56
PM\nWed, Jan 24, 2024\xa0\xa0\xa0\xa013 min ago\xa0\xa0\xa0\xa0UTC
23:56\nCall Sign KSF0\nTemp.\n60.1°F\nPrecipitation\nNo
Report\nWind\n6.9 mph\nCloud Cover\nMostly Cloudy\n1,800 ft\nRaw: KSF0
242356Z 18006G19KT 10SM FEW015 BKN018 BKN039 16/12 A3004 RMK A02 SLP171
T01560122 10156 20122 55001\n While having the tremendous advantages of
temporal and spatial completeness, these reconstructions: (1) are based
on computer models that may have model-based errors, (2) are coarsely
sampled on a 50 km grid and are therefore unable to reconstruct the
local variations of many microclimates, and (3) have particular
difficulty with the weather in some coastal areas, especially small
islands.\n We further caution that our travel scores are only as good
as the data that underpin them, that weather conditions at any given
location and time are unpredictable and variable, and that the
definition of the scores reflects a particular set of preferences that
may not agree with those of any particular reader.\n 2024 Weather
History in San Francisco California, United States\nThe data for this
report comes from the San Francisco International Airport.'}]
```

Retriever

We will also create a retriever over some data of our own. For a deeper explanation of each step here, see [this section](#).

```

from langchain_community.document_loaders import WebBaseLoader
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter

loader = WebBaseLoader("https://docs.smith.langchain.com/overview")
docs = loader.load()
documents = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200
).split_documents(docs)
vector = FAISS.from_documents(documents, OpenAIEmbeddings())
retriever = vector.as_retriever()

```

API Reference:

- [WebBaseLoader](#)
- [FAISS](#)
- [OpenAIEmbeddings](#)
- [RecursiveCharacterTextSplitter](#)

```
retriever.invoke("how to upload a dataset")[0]
```

```

Document(page_content='import Clientfrom langsmith.evaluation import
evaluateclient = Client()# Define dataset: these are your test
casesdataset_name = "Sample Dataset"dataset =
client.create_dataset(dataset_name, description="A sample dataset in
LangSmith.")client.create_examples(    inputs=[        {"postfix": "to
LangSmith"},        {"postfix": "to Evaluations in LangSmith"},    ],
    outputs=[        {"output": "Welcome to LangSmith"},        {"output":
"Welcome to Evaluations in LangSmith"},    ],
    dataset_id=dataset.id,)# Define your evaluatordef exact_match(run,
example):    return {"score": run.outputs["output"] ==
example.outputs["output"]}experiment_results = evaluate(    lambda
input: "Welcome " + input['postfix'], # Your AI system goes here
    data=dataset_name, # The data to predict and grade over    evaluators=
[exact_match], # The evaluators to score the results
    experiment_prefix="sample-experiment", # The name of the experiment
    metadata={        "version": "1.0.0",        "revision_id":', metadata=
{'source': 'https://docs.smith.langchain.com/overview', 'title':
'Getting started with LangSmith | 🦜🔧 LangSmith', 'description':
'Introduction', 'language': 'en'})

```


Now that we have populated our index that we will do doing retrieval over, we can easily turn it into a tool (the format needed for an agent to properly use it)

```
from langchain.tools.retriever import create_retriever_tool
```

API Reference:

- [create_retriever_tool](#)

```
retriever_tool = create_retriever_tool(  
    retriever,  
    "langsmith_search",  
    "Search for information about LangSmith. For any questions about  
    LangSmith, you must use this tool!",  
)
```

Tools

Now that we have created both, we can create a list of tools that we will use downstream.

```
tools = [search, retriever_tool]
```

Create the agent

Now that we have defined the tools, we can create the agent. We will be using an OpenAI Functions agent - for more information on this type of agent, as well as other options, see [this guide](#).

First, we choose the LLM we want to be guiding the agent.

```
from langchain_openai import ChatOpenAI  
  
llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=0)
```

API Reference:

- [ChatOpenAI](#)

Next, we choose the prompt we want to use to guide the agent.

If you want to see the contents of this prompt and have access to LangSmith, you can go to:

<https://smith.langchain.com/hub/hwchase17/openai-functions-agent>

```
from langchain import hub

# Get the prompt to use – you can modify this!
prompt = hub.pull("hwchase17/openai-functions-agent")
prompt.messages
```

```
[SystemMessagePromptTemplate(prompt=PromptTemplate(input_variables=[],
template='You are a helpful assistant')),
 MessagesPlaceholder(variable_name='chat_history', optional=True),
 HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=
['input'], template='{input}')),
 MessagesPlaceholder(variable_name='agent_scratchpad')]
```

Now, we can initialize the agent with the LLM, the prompt, and the tools. The agent is responsible for taking in input and deciding what actions to take. Crucially, the Agent does not execute those actions - that is done by the AgentExecutor (next step). For more information about how to think about these components, see our [conceptual guide](#).

```
from langchain.agents import create_tool_calling_agent

agent = create_tool_calling_agent(llm, tools, prompt)
```

API Reference:

- [create_tool_calling_agent](#)

Finally, we combine the agent (the brains) with the tools inside the AgentExecutor (which will repeatedly call the agent and execute tools).

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

API Reference:

- **AgentExecutor**

Run the agent

We can now run the agent on a few queries! Note that for now, these are all **stateless** queries (it won't remember previous interactions).

```
agent_executor.invoke({"input": "hi!"})
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3mHello! How can I assist you today? [0m

[1m> Finished chain. [0m
```

```
{'input': 'hi!', 'output': 'Hello! How can I assist you today?'}
```

```
agent_executor.invoke({"input": "how can langsmith help with
testing?"})
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3m
Invoking: `langsmith_search` with `{'query': 'how can LangSmith help
with testing'}`
```

```
[0m [33;1m [1;3mGetting started with LangSmith |  LangSmith
```

Skip to main contentLangSmith API DocsSearchGo to AppQuick StartUser GuideTracingEvaluationProduction Monitoring & AutomationsPrompt HubProxyPricingSelf-HostingCookbookQuick StartOn this pageGetting started with LangSmithIntroductionLangSmith is a platform for building production-grade LLM applications. It allows you to closely monitor and evaluate your application, so you can ship quickly and with confidence. Use of LangChain is not necessary – LangSmith works on its own!Install

LangSmithWe offer Python and Typescript SDKs for all your LangSmith needs. PythonTypeScriptpip install -U langsmithyarn add langchain langsmithCreate an API keyTo create an API key head to the setting pages. Then click Create API Key.Setup your environmentShellexport LANGCHAIN_TRACING_V2=trueexport LANGCHAIN_API_KEY=<your-api-key># The below examples use the OpenAI API, though it's not necessary in generalexport OPENAI_API_KEY=<your-openai-api-key>Log your first trace We provide multiple ways to log traces

Learn about the workflows LangSmith supports at each stage of the LLM application lifecycle.Pricing: Learn about the pricing model for LangSmith.Self-Hosting: Learn about self-hosting options for LangSmith.Proxy: Learn about the proxy capabilities of LangSmith.Tracing: Learn about the tracing capabilities of LangSmith.Evaluation: Learn about the evaluation capabilities of LangSmith.Prompt Hub Learn about the Prompt Hub, a prompt management tool built into LangSmith.Additional ResourcesLangSmith Cookbook: A collection of tutorials and end-to-end walkthroughs using LangSmith.LangChain Python: Docs for the Python LangChain library.LangChain Python API Reference: documentation to review the core APIs of LangChain.LangChain JS: Docs for the TypeScript LangChain libraryDiscord: Join us on our Discord to discuss all things LangChain!FAQHow do I migrate projects between organizations?Currently we do not support project migration between organizations. While you can manually imitate this by

team deals with sensitive data that cannot be logged. How can I ensure that only my team can access it?If you are interested in a private deployment of LangSmith or if you need to self-host, please reach out to us at sales@langchain.dev. Self-hosting LangSmith requires an annual enterprise license that also comes with support and formalized access to the LangChain team.Was this page helpful?NextUser GuideIntroductionInstall LangSmithCreate an API keySetup your environmentLog your first traceCreate your first evaluationNext StepsAdditional ResourcesFAQHow do I migrate projects between organizations?Why aren't my runs aren't showing up in my project?My team deals with sensitive data that cannot be logged. How can I ensure that only my team can access it?CommunityDiscordTwitterGitHubDocs CodeLangSmith SDKPythonJS/TSMoreHomepageBlogLangChain Python DocsLangChain JS/TS DocsCopyright © 2024 LangChain, Inc. [0m [32;1m [1;3mLangSmith is a platform for building production-grade LLM applications that can help with testing in the following ways:

1. ****Tracing****: LangSmith provides tracing capabilities that allow you to closely monitor and evaluate your application during testing. You can log traces to track the behavior of your application and identify

any issues.

2. ****Evaluation****: LangSmith offers evaluation capabilities that enable you to assess the performance of your application during testing. This helps you ensure that your application functions as expected and meets the required standards.

3. ****Production Monitoring & Automations****: LangSmith allows you to monitor your application in production and automate certain processes, which can be beneficial for testing different scenarios and ensuring the stability of your application.

4. ****Prompt Hub****: LangSmith includes a Prompt Hub, a prompt management tool that can streamline the testing process by providing a centralized location for managing prompts and inputs for your application.

Overall, LangSmith can assist with testing by providing tools for monitoring, evaluating, and automating processes to ensure the reliability and performance of your application during testing phases. [0m

[1m> Finished chain. [0m

```
{'input': 'how can langsmith help with testing?',
 'output': 'LangSmith is a platform for building production-grade LLM
 applications that can help with testing in the following ways:\n\n1.
 **Tracing**\n\nLangSmith provides tracing capabilities that allow you to
 closely monitor and evaluate your application during testing. You can
 log traces to track the behavior of your application and identify any
 issues.\n\n2. **Evaluation**\n\nLangSmith offers evaluation capabilities
 that enable you to assess the performance of your application during
 testing. This helps you ensure that your application functions as
 expected and meets the required standards.\n\n3. **Production
 Monitoring & Automations**\n\nLangSmith allows you to monitor your
 application in production and automate certain processes, which can be
 beneficial for testing different scenarios and ensuring the stability
 of your application.\n\n4. **Prompt Hub**\n\nLangSmith includes a Prompt
 Hub, a prompt management tool that can streamline the testing process
 by providing a centralized location for managing prompts and inputs for
 your application.\n\nOverall, LangSmith can assist with testing by
 providing tools for monitoring, evaluating, and automating processes to
 ensure the reliability and performance of your application during
 testing phases.'}
```

```
agent_executor.invoke({"input": "whats the weather in sf?"})
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3m
Invoking: `tavily_search_results_json` with `{ 'query': 'weather in San Francisco'}`

[0m [36;1m [1;3m[{ 'url': 'https://www.weatherapi.com/', 'content': "
{'location': {'name': 'San Francisco', 'region': 'California',
'country': 'United States of America', 'lat': 37.78, 'lon': -122.42,
'tz_id': 'America/Los_Angeles', 'localtime_epoch': 1712847697,
'localtime': '2024-04-11 8:01'}, 'current': {'last_updated_epoch':
1712847600, 'last_updated': '2024-04-11 08:00', 'temp_c': 11.1,
'temp_f': 52.0, 'is_day': 1, 'condition': {'text': 'Partly cloudy',
'icon': '//cdn.weatherapi.com/weather/64x64/day/116.png', 'code':
1003}, 'wind_mph': 2.2, 'wind_kph': 3.6, 'wind_degree': 10, 'wind_dir':
'N', 'pressure_mb': 1015.0, 'pressure_in': 29.98, 'precip_mm': 0.0,
'precip_in': 0.0, 'humidity': 97, 'cloud': 25, 'feelslike_c': 11.5,
'feelslike_f': 52.6, 'vis_km': 14.0, 'vis_miles': 8.0, 'uv': 4.0,
'gust_mph': 2.8, 'gust_kph': 4.4}}"}], { 'url':
'https://www.yahoo.com/news/april-11-2024-san-francisco-
122026435.html', 'content': "2024 NBA Mock Draft 6.0: Projections for
every pick following March Madness With the NCAA tournament behind us,
here's an updated look at Yahoo Sports' first- and second-round
projections for the ..."}, { 'url':
'https://www.weathertab.com/en/c/e/04/united-states/california/san-
francisco/', 'content': 'Explore comprehensive April 2024 weather
forecasts for San Francisco, including daily high and low temperatures,
precipitation risks, and monthly temperature trends. Featuring detailed
day-by-day forecasts, dynamic graphs of daily rain probabilities, and
temperature trends to help you plan ahead. ... 11 65°F 49°F 18°C 9°C
29% 12 64°F 49°F ...'}, { 'url':
'https://weatherspark.com/h/y/557/2024/Historical-Weather-during-2024-
in-San-Francisco-California-United-States', 'content': 'San Francisco
Temperature History 2024\nHourly Temperature in 2024 in San
Francisco\nCompare San Francisco to another city:\nCloud Cover in 2024
in San Francisco\nDaily Precipitation in 2024 in San
Francisco\nObserved Weather in 2024 in San Francisco\nHours of Daylight
and Twilight in 2024 in San Francisco\nSunrise & Sunset with Twilight
and Daylight Saving Time in 2024 in San Francisco\nSolar Elevation and
Azimuth in 2024 in San Francisco\nMoon Rise, Set & Phases in 2024 in
San Francisco\nHumidity Comfort Levels in 2024 in San Francisco\nWind
```

Speed in 2024 in San Francisco\nHourly Wind Speed in 2024 in San Francisco\nHourly Wind Direction in 2024 in San Francisco\nAtmospheric Pressure in 2024 in San Francisco\nData Sources\n See all nearby weather stations\nLatest Report – 3:56 PM\nWed, Jan 24, 2024\n13 min ago\nUTC 23:56\nCall Sign KSF0\nTemp.\n60.1°F\nPrecipitation\nNo Report\nWind\n6.9 mph\nCloud Cover\nMostly Cloudy\n1,800 ft\nRaw: KSF0 242356Z 18006G19KT 10SM FEW015 BKN018 BKN039 16/12 A3004 RMK A02 SLP171 T01560122 10156 20122 55001\nWhile having the tremendous advantages of temporal and spatial completeness, these reconstructions: (1) are based on computer models that may have model-based errors, (2) are coarsely sampled on a 50 km grid and are therefore unable to reconstruct the local variations of many microclimates, and (3) have particular difficulty with the weather in some coastal areas, especially small islands.\n We further caution that our travel scores are only as good as the data that underpin them, that weather conditions at any given location and time are unpredictable and variable, and that the definition of the scores reflects a particular set of preferences that may not agree with those of any particular reader.\n 2024 Weather History in San Francisco California, United States\nThe data for this report comes from the San Francisco International Airport.', {'url': 'https://www.msn.com/en-us/weather/topstories/april-11-2024-san-francisco-bay-area-weather-forecast/vi-BB1lrXDb', 'content': 'April 11, 2024 San Francisco Bay Area weather forecast. Posted: April 11, 2024 | Last updated: April 11, 2024 ...'}] [0m [32;1m [1;3mThe current weather in San Francisco is partly cloudy with a temperature of 52.0°F (11.1°C). The wind speed is 3.6 kph coming from the north, and the humidity is at 97%. [0m

[1m> Finished chain. [0m

```
{'input': 'whats the weather in sf?',
 'output': 'The current weather in San Francisco is partly cloudy with a temperature of 52.0°F (11.1°C). The wind speed is 3.6 kph coming from the north, and the humidity is at 97%.'}
```

Adding in memory

As mentioned earlier, this agent is stateless. This means it does not remember previous interactions. To give it memory we need to pass in previous `chat_history`. Note: it needs to be called `chat_history` because of the prompt we are using. If we use a different prompt, we could change the variable name

```
# Here we pass in an empty list of messages for chat_history because
it is the first message in the chat
agent_executor.invoke({"input": "hi! my name is bob", "chat_history":
[]})
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3mHello Bob! How can I assist you today? [0m

[1m> Finished chain. [0m
```

```
{'input': 'hi! my name is bob',
 'chat_history': [],
 'output': 'Hello Bob! How can I assist you today?'}
```

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

API Reference:

- [AIMessage](#)
- [HumanMessage](#)

```
agent_executor.invoke(
    {
        "chat_history": [
            HumanMessage(content="hi! my name is bob"),
            AIMessage(content="Hello Bob! How can I assist you
today?"),
        ],
        "input": "what's my name?",
    }
)
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3mYour name is Bob. How can I assist you, Bob? [0m
```



```
[1m> Finished chain. [0m
```

```
{'chat_history': [HumanMessage(content='hi! my name is bob'),
  AIMessage(content='Hello Bob! How can I assist you today?')],
 'input': "what's my name?",
 'output': 'Your name is Bob. How can I assist you, Bob?'}
```

If we want to keep track of these messages automatically, we can wrap this in a `RunnableWithMessageHistory`. For more information on how to use this, see [this guide](#).

```
from langchain_community.chat_message_histories import
ChatMessageHistory
from langchain_core.runnables.history import
RunnableWithMessageHistory
```

API Reference:

- [ChatMessageHistory](#)
- [RunnableWithMessageHistory](#)

```
message_history = ChatMessageHistory()
```

```
agent_with_chat_history = RunnableWithMessageHistory(
    agent_executor,
    # This is needed because in most real world scenarios, a session
    id is needed
    # It isn't really used here because we are using a simple in
    memory ChatMessageHistory
    lambda session_id: message_history,
    input_messages_key="input",
    history_messages_key="chat_history",
)
```

```
agent_with_chat_history.invoke(
    {"input": "hi! I'm bob"},
    # This is needed because in most real world scenarios, a session
    id is needed
    # It isn't really used here because we are using a simple in
```

```
memory ChatMessageHistory
    config={"configurable": {"session_id": "<foo>"}}
)
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3mHello Bob! How can I assist you today? [0m

[1m> Finished chain. [0m
```

```
{'input': "hi! I'm bob",
 'chat_history': [],
 'output': 'Hello Bob! How can I assist you today?'}
```

```
agent_with_chat_history.invoke(
    {"input": "what's my name?"},
    # This is needed because in most real world scenarios, a session
    id is needed
    # It isn't really used here because we are using a simple in
    memory ChatMessageHistory
    config={"configurable": {"session_id": "<foo>"}}
)
```

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3mYour name is Bob! How can I help you, Bob? [0m

[1m> Finished chain. [0m
```

```
{'input': "what's my name?",
 'chat_history': [HumanMessage(content="hi! I'm bob"),
                  AIMessage(content='Hello Bob! How can I assist you today?')],
 'output': 'Your name is Bob! How can I help you, Bob?'}
```

Conclusion

That's a wrap! In this quick start we covered how to create a simple agent. Agents are a complex topic, and there's a lot to learn! Head back to the [main agent page](#) to find more resources on conceptual guides, different types of agents, how to create custom tools, and more!

Help us out by providing feedback on this documentation page:



This is documentation for LangChain **v0.1**, which is no longer actively maintained.

For the current stable version, see [this version](#) (Latest).

[Composition](#)[Agents](#)[Concepts](#)

Concepts

The core idea of agents is to use a language model to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

There are several key components here:

Schema

LangChain has several abstractions to make working with agents easy.

AgentAction

This is a dataclass that represents the action an agent should take. It has a `tool` property (which is the name of the tool that should be invoked) and a `tool_input` property (the input to that tool)

AgentFinish

This represents the final result from an agent, when it is ready to return to the user. It contains a `return_values` key-value mapping, which contains the final agent output. Usually, this contains an `output` key containing a string that is the agent's response.

Intermediate Steps

These represent previous agent actions and corresponding outputs from this CURRENT agent run. These are important to pass to future iteration so the agent knows what work it has already done. This is typed as a `List[Tuple[AgentAction, Any]]`. Note that observation is currently left as type `Any` to be maximally flexible. In practice, this is often a string.

Agent

This is the chain responsible for deciding what step to take next. This is usually powered by a language model, a prompt, and an output parser.

Different agents have different prompting styles for reasoning, different ways of encoding inputs, and different ways of parsing the output. For a full list of built-in agents see [agent types](#). You can also **easily build custom agents**, should you need further control.

Agent Inputs

The inputs to an agent are a key-value mapping. There is only one required key: `intermediate_steps`, which corresponds to `Intermediate Steps` as described above.

Generally, the `PromptTemplate` takes care of transforming these pairs into a format that can best be passed into the LLM.

Agent Outputs

The output is the next action(s) to take or the final response to send to the user (`AgentActions` or `AgentFinish`). Concretely, this can be typed as `Union[AgentAction, List[AgentAction], AgentFinish]`.

The output parser is responsible for taking the raw LLM output and transforming it into one of these three types.

AgentExecutor

The agent executor is the runtime for an agent. This is what actually calls the agent, executes the actions it chooses, passes the action outputs back to the agent, and repeats. In pseudocode, this looks roughly like:

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

While this may seem simple, there are several complexities this runtime handles for you, including:

1. Handling cases where the agent selects a non-existent tool
2. Handling cases where the tool errors
3. Handling cases where the agent produces output that cannot be parsed into a tool invocation
4. Logging and observability at all levels (agent decisions, tool calls) to stdout and/or to [LangSmith](#).

Tools

Tools are functions that an agent can invoke. The `Tool` abstraction consists of two components:

1. The input schema for the tool. This tells the LLM what parameters are needed to call the tool. Without this, it will not know what the correct inputs are. These parameters should be sensibly named and described.
2. The function to run. This is generally just a Python function that is invoked.

Considerations

There are two important design considerations around tools:

1. Giving the agent access to the right tools
2. Describing the tools in a way that is most helpful to the agent

Without thinking through both, you won't be able to build a working agent. If you don't give the agent access to a correct set of tools, it will never be able to accomplish the objectives you give it. If you don't describe the tools well, the agent won't know how to use them properly.

LangChain provides a wide set of built-in tools, but also makes it easy to define your own (including custom descriptions). For a full list of built-in tools, see the [tools integrations section](#)

Toolkits

For many common tasks, an agent will need a set of related tools. For this LangChain provides the concept of toolkits - groups of around 3-5 tools needed to accomplish specific objectives. For example, the GitHub toolkit has a tool for searching through GitHub issues, a tool for reading a file, a tool for commenting, etc.

LangChain provides a wide set of toolkits to get started. For a full list of built-in toolkits, see the [toolkits integrations section](#)

Help us out by providing feedback on this documentation page:

