Building Smarter Agents with LangGraph Tools, Memory & Workflows

A quick guide to using tools and memory in your agent workflows

R
RAJNEESH JHA
JUN 05, 2025 · PAID

8 2 Share

In my recent articles, I've been diving into various tools and libraries for develop
multi-agent systems.

We started with the **OpenAI Agent SDK**, where I shared how to build your own Research Agent:
🔗 Build Your Own Deep Research Agent using OpenAI's Agent SDK

Then we explored **CrewAI**, a powerful framework for creating collaborative AI a
🔗 Build Your First AI Coding Buddy with CrewAI

Today, let's take a look at another exciting and widely adopted framework — LangGraph. This graph-based library is designed for building complex, structure
agent workflows with ease. In the next few minutes, we'll see how to use tools an
memory with LangGraph.

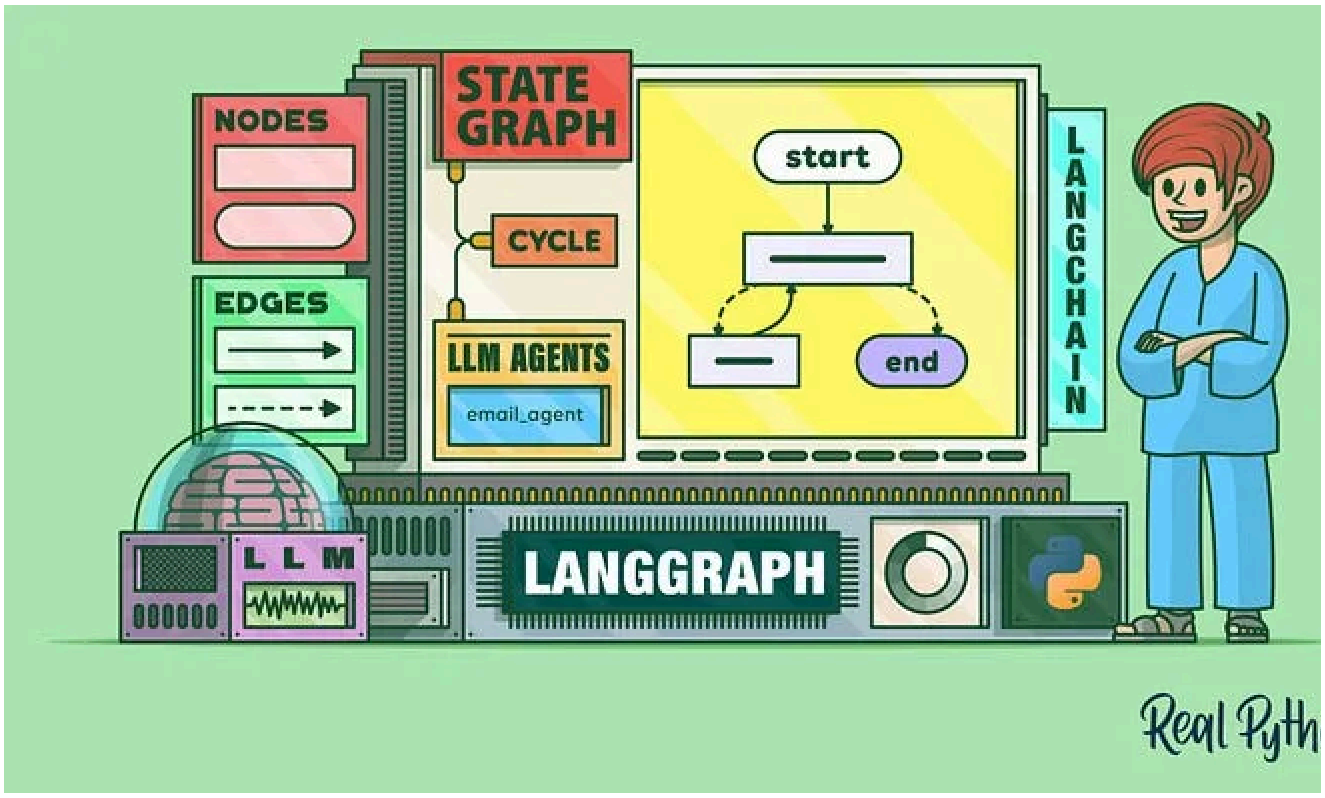

Image Source: https://realpython.com/langgraph-python/

1. What is LangGraph ?

LangGraph is a stateful, graph-based framework designed to build complex, mul agent applications using LLMs. Unlike traditional linear workflows, LangGraph leverages directed graphs where each node represents an agent or task, and edge define their interactions. This structure facilitates dynamic, iterative, and memo aware agentic systems.

Core Features of LangGraph:

- **Graph-Oriented Workflow**: Design flexible, branching workflows with loop conditionals, enabling complex agent interactions.
- **Stateful Execution**: Automatically persist agent states at each step, allowing error recovery, time travel, and human-in-the-loop interventions.
- **Integrated Memory**: Support for both short-term (session-based) and long-t (persistent across sessions) memory, enabling agents to recall and adapt to u interactions.

- **Human-in-the-Loop Control**: Pause execution to seek human approval or modification before proceeding, enhancing oversight and control .
- **Seamless Integration with LangChain**: While built on LangChain, LangGra can function independently, offering flexibility in deployment.
- **Open-Source and Extensible**: Released under the MIT license, LangGraph i to use and can be extended to meet specific needs.

2. Key Terminology in LangGraph:

1. The **State** serves as the central data structure in LangGraph, encapsulating a relevant information during the execution of the workflow. It can be as simp dictionary or a more structured model using tools like Pydantic. Each node i graph receives and updates this state, ensuring continuity and context throu the process.
   When defining the state in LangGraph, two important concepts from LangC play a crucial role:

- `Annotated` **from Python's** `typing` **the module** is used to attach metadata t hints. This metadata usually specifies a **reducer function**, which dictates ho updates to a particular state field should be merged with existing data.
- **Reducers** are functions responsible for determining how updates to individu state fields are applied. Each field can have its own reducer, allowing for customized merging behaviour. If no reducer is specified, the default behavi to overwrite the existing value with the new update.

```
from typing import Annotated
from typing_extensions import TypedDict
from operator import add

class State(TypedDict):
    messages: Annotated[list[str], add]
```

In this example, the `messages` field is a list of strings, and the `add` function fro `operator` the module is used as a reducer. This means that when new messages added, they will be concatenated to the existing list rather than replacing it.

2. Nodes are the fundamental units of computation within LangGraph. Each noc represents a specific task or action, such as:

- Processing user input
- Interacting with external APIs
- Making decisions based on conditions
- Generating responses using LLMs

Nodes are typically implemented as Python functions that accept the current sta input and return an updated state.

3. Edges define the flow of execution between nodes, determining the sequence i which tasks are performed. There are different types of edges:

- **Normal Edges**: Establish a straightforward, unidirectional connection betwe nodes.
- **Conditional Edges**: Introduce decision-making into the workflow, directing flow based on specific conditions or logic.

These edges enable the creation of complex workflows with branching logic and

4. Unlike traditional DAGs, LangGraph supports **cycles**, allowing nodes to revisi previous steps. This feature is particularly useful for tasks that require iterative refinement or repeated evaluation, such as debates between agents or recursive problem-solving.

5. LangGraph provides built-in persistence mechanisms to save the state of the workflow at various points. This capability enables:

- **Checkpointing**: Saving snapshots of the graph's state to resume execution la

- **Human-in-the-Loop Interactions**: Allowing manual intervention during the workflow.
- **Fault Tolerance**: Recovering from errors by restoring to a previous state.

Persistence is achieved through check pointers and storage systems that manage across different threads and sessions.

6. In LangGraph, a **thread** represents an individual session or conversation. Each thread maintains its own state, enabling personalized interactions and context retention across multiple sessions. This is particularly beneficial for applications chatbots or virtual assistants that require memory of past interactions.

7. LangGraph includes a **storage** system that allows for the saving and retrieval c across different threads. This feature supports:

- **Cross-Thread Persistence**: Maintaining information across multiple session
- **Flexible Name spacing**: Organizing data for different users or contexts.
- **Document Storage**: Saving data in JSON format for easy manipulation and retrieval.

The storage system facilitates the creation of long-term memories and knowledg bases.

In the next few sections, let's create some tools and integrate with agents.

3. Tool Creation:

Serper Web-Search Tool: Serper is a web search API tool that provides real-tim Google-like search results. It's often used in AI and agent frameworks like Lang to allow language models to access up-to-date information from the web.

Explore more here: https://serper.dev/

```
from langchain_community.utilities import GoogleSerperAPIWrapper

serper = GoogleSerperAPIWrapper()
serper.run("What is the capital of India?")

from langchain.agents import Tool

tool_search = Tool(
    name="search",
    func=serper.run,
    description=(
        "A web search tool powered by Serper.dev. Use this tool when
need to retrieve real-time "
        "information from the internet, such as current events, rece
news, public data, or facts that "
        "are not part of the model's training knowledge. Ideal for
answering up-to-date queries, verifying information, "
        "or supplementing static knowledge with fresh content from l
web sources."
    )
)

tool_search.invoke("What is the capital of India?")
```

Push Over Tool: **Pushover** is a simple and reliable push notification service that allows you to send real-time alerts to your phone, tablet, or desktop. It’s widely u developers and system administrators to notify users or themselves of important events like server errors, completed tasks, or AI agent actions.

Explore more here: https://pushover.net/

```
import os
import requests
from langchain.agents import Tool

# Load credentials from environment variables
pushover_token = os.getenv("PUSHOVER_TOKEN")
pushover_user = os.getenv("PUSHOVER_USER")
pushover_url = "https://api.pushover.net/1/messages.json"
```

```
def push(text: str):
    """Send a push notification using the Pushover API."""
    if not pushover_token or not pushover_user:
        raise ValueError("Pushover credentials are missing. Check
environment variables.")

    payload = {
        "token": pushover_token,
        "user": pushover_user,
        "message": text
    }

    response = requests.post(pushover_url, data=payload)

    if response.status_code != 200:
        raise Exception(f"Failed to send push notification:
{response.text}")

tool_push = Tool(
    name="send_push_notification",
    func=push,
    description=(
        "Useful for sending real-time push notifications to the user
the Pushover API. "
        "Use this when you need to alert the user with important
messages or status updates."
    )
)

tool_push.invoke("Hello, Good People !")
# Check the notification in your pushover app on your phone.
```

Now that the tools are set up, let's build a basic graph with nodes powered by an that can use these tools.

4. Define Workflow:

In this section, we'll create a basic **LangGraph** powered by an **LLM** that can inte with external tools. We've already defined two tools:

- A **web search tool** (using Serper) for real-time internet queries
- A **push notification tool** (using Pushover) to send alerts

Now, we'll build a simple graph where:

- An LLM node processes user input
- If tool use is needed, the graph routes to a tool node
- After the tool is invoked, control returns to the LLM to decide what to do ne

```
from typing import Annotated
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages
from dotenv import load_dotenv
from IPython.display import Image, display
import gradio as gr
from langgraph.prebuilt import ToolNode, tools_condition
import requests
import os
from langchain_openai import ChatOpenAI
from typing import TypedDict

load_dotenv(override=True)

# Create a list of Tools
tools = [tool_search, tool_push]

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

# Start the Graph Building process
graph_builder = StateGraph(State)

# Define the LLM to be used
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools)

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

# Add Nodes
graph_builder.add_node("chatbot", chatbot)
graph_builder.add_node("tools", ToolNode(tools=tools))

# If any tool is called, we return to the chatbot to decide the next
step
graph_builder.add_conditional_edges( "chatbot", tools_condition,
"tools")
graph_builder.add_edge("tools", "chatbot")

graph_builder.add_edge(START, "chatbot")

# Compile the Graph
graph = graph_builder.compile()
display(Image(graph.get_graph().draw_mermaid_png()))
```

We can use a simple Gradio Chat interface to test this graph quickly:

Note: Before running, make sure you have below things in environment variable (LangSmith is optional, if you want to trace the agent in a better way.)

- OPENAI_API_KEY
- LANGSMITH_TRACING=true
- LANGSMITH_ENDPOINT
- LANGSMITH_API_KEY
- LANGSMITH_PROJECT
- SERPER_API_KEY
- PUSHOVER_USER
- PUSHOVER_TOKEN
- PUSHOVER_URL

```
def chat(user_input: str, history):
    result = graph.invoke({"messages": [{"role": "user", "content":
```

```
user_input}]})
    return result["messages"][-1].content


gr.ChatInterface(chat, type="messages").launch()
```

5. Adding Memory:

You might be wondering, *"If LangGraph maintains and updates state between nodes, do we still need memory?"*

LangGraph processes workflows in **units called "super-steps"**.

- A **super-step** is a single pass through the graph running one or more nodes (possibly in parallel), based on the current state.
- Each time you call `graph.invoke()`, you're running **one super-step**.
- **Reducers** handle state updates within this super-step they control how data merged or modified while the graph is running.

Once the super-step completes, the graph's in-memory state is gone unless you explicitly save it.

This means:

- LangGraph manages **short-term state** during execution (via the state object reducers)
- But it doesn't **automatically persist memory** between interactions or sessior

That's where **checkpointing and memory storage** come in.

To retain state between graph invocations, LangGraph offers **persistence mecha** like:

- **Checkpointing**: Save the state after each super-step so it can be resumed or referenced later

- **Storage Backends**: For managing long-term memory or user-specific conversations across sessions

This enables:

- Stateful, ongoing conversations
- Time travel (restoring to previous states)
- Human-in-the-loop interventions
- Memory-aware agents

Let's now enhance the LangGraph with **memory support** using a **checkpointing system**. This allows the agent to retain and recall conversation history across sup steps (individual runs of the graph).

```
from langgraph.checkpoint.memory import MemorySaver

# MemorySaver is a built-in checkpointing utility.
# It stores the state of the graph after each run, allowing recovery
# continuation later.

memory = MemorySaver()

# This means the graph automatically saves state after each interact
(super
# step).
graph = graph_builder.compile(checkpointer=memory)
display(Image(graph.get_graph().draw_mermaid_png()))

# This is the session ID or conversation thread ID. Think of it like
# chatroom. All messages in the same thread share memory.
config = {"configurable": {"thread_id": "1"}}

def chat(user_input: str, history):
    result = graph.invoke({"messages": [{"role": "user", "content":
user_input}]}, config=config)
    return result["messages"][-1].content
```

```
gr.ChatInterface(chat, type="messages").launch()

graph.get_state(config)

list(graph.get_state_history(config))

# By specifying a checkpoint_id, you can restore the graph to a prev
point # in time.
# config = {"configurable": {"thread_id": "1", "checkpoint_id": ...}
# graph.invoke(None, config=config)
```

Persisting Memory:

Previously, we used `MemorySaver` for temporary in-memory persistence (RAM
We can also use **SQLite**, which gives LangGraph application **disk-based**, **long-te
memory**. This means the agent can remember past interactions **even after restar
the program**.

```
import sqlite3
from langgraph.checkpoint.sqlite import SqliteSaver


# SqliteSaver is LangGraph's utility to store state checkpoints in a
table.
# This allows for long-term persistence, cross-session recall, and s
# restoration from disk.

db_path = "memory.db"
conn = sqlite3.connect(db_path, check_same_thread=False)
sql_memory = SqliteSaver(conn)

graph = graph_builder.compile(checkpointer=sql_memory)
display(Image(graph.get_graph().draw_mermaid_png()))

# Thread ID "3" defines a unique session. Each session has its own
memory.
# This allows the agent to carry on context from earlier messages ac
# restarts.
```

```
config = {"configurable": {"thread_id": "3"}}

def chat(user_input: str, history):
    result = graph.invoke({"messages": [{"role": "user", "content":
user_input}]}, config=config)
    return result["messages"][-1].content


gr.ChatInterface(chat, type="messages").launch()
```

This should give the agent a persistent memory that remains intact even after a k
restart. Try mentioning your name, then restart the kernel the agent should still
remember it

6. Conclusion:

In this post, we went through the fundamentals of LangGraph a powerful, graph
framework for building structured, multi-agent AI workflows.

We explored:

- What LangGraph is and how it differs from linear agent frameworks
- Key features like stateful execution, memory integration, and human-in-the-
control
- Core concepts such as State, Nodes, Edges, and how they work together to c
dynamic workflows
- Building a simple agent that is integrated with multiple tools
- Adding memory to that agent and persisting the memory

As agentic systems continue to evolve, frameworks like LangGraph will play a ke
in making them more reliable, transparent, and customizable. Stay tuned! In the
few posts, we'll dive into hands-on examples of building real-world agent workfl
with tools and memory using LangGraph.

Thanks for Reading. Have any questions or concerns? Feel free to connect, and s
tuned for more insightful content. Let's connect on LinkedIn:
https://www.linkedin.com/in/rajneesh407/

8 Likes · 2 Restacks

Previous Ne

A guest post by
Rajneesh Jha
Interested in solving business problems using Machine Learning and AI. https://www.linkedin.com/in/rajneesh407/

Subscribe to Rajr

Discussion about this post

Comments Restacks

Write a comment...