A Comprehensive Guide to LangGraph: Managing Agent State with Tools

Joey O'Neill Follow 25 min read · Aug 7, 2025

image source: https://awstip.com/built-with-langgraph-1-hello-world-7f1ea87ed5b9

Introduction

Hello all, my name is Joey O'Neill — I am an AI/ML Engineering Consultant with a focus in Generative AI applications. This article is an in-depth technical walkthrough of state management through tool functions in the Python version of LangGraph (`v0.5.4`).

To get the most out of this guide, you should have a general understanding of LangChain and basic familiarity with LangGraph — especially things like graph flow, nodes, edges, tools, and ReAct agents. If you need to brush up on these fundamentals, I'd recommend starting with the LangGraph documentation or the LangChain Academy's online course for LangGraph, which I found incredibly helpful for grasping the basics.

What We Will Cover

Ever tried to build a LangGraph agent where your tools need to track usage statistics, maintain session data, or coordinate complex workflows? If so, you've probably hit the same wall I did: LangGraph's default ToolNode simply can't handle tools that need to read from or write to the graph's state. The moment you try to use `InjectedState` or return `Command` objects, your tools get completely ignored.

In this guide, I'll walk you through the problem I encountered while building a production ReAct agent, the dead ends I hit trying to solve it, and ultimately what I figured out to get it working. We'll dive deep into building flexible tool nodes that can handle both regular tool returns and complex state operations, implement proper reducers for concurrent execution, and create a complete working system that gives you full control over state management in your LangGraph applications.

By the end, you'll have a robust pattern for building agents that can seamlessly mix tools with different capabilities — some that just return data, others that track metrics, and still others that coordinate complex multi-step processes. All while maintaining the performance and reliability you need for production systems.

Let's get into it!

Where It All Started...

I was working on a proof-of-concept beta feature for an application where we transformed a static retrieval-augmented generation (RAG) workflow into a ReAct-styled agent for a more dynamic approach to information retrieval and post-interaction tracking. My only real prior experience with LangGraph came from documentation, LangChain Academy courses, and some bare-minimum testing projects — just enough to get a feel for the framework, not for actual production use-cases. Still, this wouldn't deter me from attempting to build a scalable system ready for real user usage.

When the opportunity arose within our project, I jumped at the chance to create a foundational ReAct agent for a small beta testing group. This agent handled dynamic information retrieval and reranking to supply informed responses across multiple different corpuses of non-overlapping information — different storages about different information. Effectively, the retrieval process could iterate and decide which specific indexes to search based on a user's query.

But then I hit a wall — one problem that became a persistent thorn in my side that I could not figure out for the life of me.

The Core Challenge

The feature requirements were straightforward but tricky: we needed to track usage and retrieval statistics accurately while also accessing pre-generated information to reduce repeated token usage. This meant we not only needed to access the LangGraph agent's state directly from within tool functions, but we also needed to write to that state simultaneously. The requirements varied depending on which tool was called and executed, meaning there was no uniformity across tools in terms of parameters or return values.

With no prior knowledge of this specific issue, I dove into the LangGraph documentation and discovered two key components:

1. **InjectedState** — a special type annotation that tells LangGraph to automatically pass the current graph state as a parameter to your tool function, giving you read access to any state variables you need.
2. **Command object** — a powerful return type that lets you break out of the standard linear flow by explicitly updating state fields and optionally routing to specific nodes, essentially giving you programmatic control over what happens next in the graph.

So we had the pieces we needed: InjectedState for reading state and the Command object for writing to it from within our tool functions.

Where Things Fall Apart...

We were using LangGraph's prebuilt ToolNode — honestly, it was the only approach I knew at the time. All the tutorials, documentation, and examples I'd encountered used the default ToolNode. In hindsight, I didn't even know custom tool nodes were an option. With that in mind, here's where things got frustrating: anytime you added anything that reached out to state from within a tool, the agent would completely ignore that tool. It didn't matter whether you were only passing in InjectedState, returning a Command object, or using both together — any interaction with the graph's state from within the tool caused it to be completely ignored by the LLM, as if it wasn't in the tool list at all.

Here's an example of a tool that utilizes both InjectedState and Command object return:

```
# Example of tool that utilizes Injected State & Command object return
@tool
def search_index_a(
  query: str,
  state: Annotated[CustomAgentState, InjectedState],
  tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    """
    Performs a hybrid search across index A based on the user's query.
```

```
    Returns the 3 most relevant chunks of information.
    """
    ret = perform_index_search(query)
    return Command(update = {
        "data": ret,
        "messages": [
            ToolMessage(
                content=str(ret),
                name="search_index_a",
                tool_call_id=tool_call_id
            )
        ]
    })
```

This is actually a more refined version than what I started with, but the fundamental issue remained the same. Add a tool like this to your tool list, pass it to LangGraph's default ToolNode, as well as bind it to your LLM, and watch as the function fails to register — effectively robbing you of the functionality you desperately need.

That's when I realized I needed to dig deeper and find a different approach.

The Search for Answers (Spoiler: It Was Rough)

Trying to solve this problem led me to many different avenues of relief. Of course, I first asked different LLM chat applications, then the classic approach of asking on Stack Overflow — both with no luck, leading to the even more antiquated method: sifting through the weeds via regular internet searching, which still yielded minimal information or direction on this specific issue.

Then I stumbled upon a GitHub discussion thread on the LangGraph QA board where another developer, Dominik Römer, was essentially asking the same question: how do you update graph state with tool output? The thread had several replies, but three responses became the breadcrumbs that eventually led me to a full solution.

First, Isaac Francisco dropped this crucial insight: "The prebuilt `ToolNode` will only append to the `messages` key in your state, but you can define a custom node [to handle non-message based state updates]." This was the first hint that I'd been thinking about the problem all wrong—the default ToolNode was fundamentally limited in what it could do with state. In hindsight, this explained why I'd been banging my head against the wall trying to pass Command objects directly to the prebuilt ToolNode and wondering why nothing was executing properly. The prebuilt ToolNode automatically wraps tool function responses and returns them as a ToolMessage, essentially only ever updating the `messages` key in the agent's state.

Next, Vadym Barda answered the thread with more details about how the Command object could be leveraged for more complex state operations. Then, from within that answer's reply thread, Matheus Rodrigues shared some workarounds he'd been using to update tool call arguments and pass state into tool calls. While his specific approach had its own limitations, it opened my eyes to different ways of thinking about the state flow problem.

While individually these answers don't paint a full picture, they essentially put me on the right path to fully solving the challenge of simultaneous tool usage and state read/write operations. I couldn't find a comprehensive explanation or starting point anywhere else on the internet, so huge thanks to Isaac, Vadym, and Matheus — along with the other users who posted to the thread — for at least giving me a foothold on tackling this issue.

The Solution: Commands, Custom Tool Nodes, and Reducers

Now that you understand the problem we faced, let's dive into the technical solution. In this section, I'll walk you through building a custom tool node that can handle different types of tool outputs — both the standard tool messages wrapper and the more powerful Command objects that let you update state.

Here's what we'll cover step by step:

- How to read state from within a tool function.
- How to write to state using Command objects from within a tool function.
- Building a custom tool node that works with both approaches.
- Setting up annotations and reducers to handle concurrent tool updates to state arguments.

By the end, you'll have a working example that gives you key insights to maintain full control over state management in your LangGraph agents.

Environment Set Up

For this tutorial, we will be using the following version of python:

```
python==3.13.5
```

Create a virtual python environment with the following command:

```
python -m venv .venv
```

Activate the virtual environment with the following command:

For Windows Users:

```
.\.venv\Scripts\activate
```

For Mac Users:

```
source .venv/bin/activate
```

For the packages, we will mainly be using the following:

```
langchain-core==0.3.72
langchain-openai==0.3.28
langgraph==0.5.4
python-dotenv==1.1.1
```

Install the packages using *pip* either directly or copy them into a *requirements.txt* file.

Let's create an environment file (*.env*) to securely store environment variables like API keys and other sensitive information. This file keeps your credentials safe by allowing you to separate them from the main codebase. Later, we'll add it to a *.gitignore* file to ensure that any keys or secrets won't be exposed if you upload the project to a public repository.

In the *.env* file, create a variable to store your OpenAI API key and ensure you use the same variable name as below. Please note, you will need to get your own API key from OpenAI and have credits on your account.

```
OPENAI_API_KEY='<YOUR API KEY HERE>'
```

With that created, save it and let's move on to the *.gitignore* file.

Create the *.gitignore* file and include the following and save it:

```
*.env
.env
.venv/
__pycache__/
```

Finally, create the python file that you will be working in. You can do this in a *.py* file, but I find it easier to follow along using a python notebook file (*.ipynb*).

Now, let's load the environment variables into the python file and save it. This is required to allow access to the OpenAI API Key we saved in it previously.

```
# imports
from dotenv import load_dotenv

# load in the .env variables
load_dotenv()
```

With all these files created, you should have a directory that looks something similar to this:

```
project-folder/
├── .env
├── .gitignore
├── main.ipynb
└── .venv/
```

And our environment is set up and ready to go!

Our Example: A Math Operations Agent

In this walkthrough, we will be creating a simple reAct agent that handles basic mathematical operations — addition, subtraction, multiplication, and division. I picked this example for its underlying simplicity to focus on the core LangGraph concepts you will need to know.

How will we do this? For each time a mathematical operation tool is executed, we will track the amount of steps in our graph's state. We'll also build a fifth tool that reads from state to report back these usage statistics. This way, you will see both reading and writing to state within tool functions in action.

Drafting Our Agent State

First, let's set up our state schema. We need the standard `messages` attribute that every LangGraph agent requires, plus individual counters for each mathematical operation:

```
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages

# Agent Graph State
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    add_n: int
    subtract_n: int
    multiply_n: int
    divide_n: int
```

Pretty straightforward — we're just tracking how many times each operation gets called.

Reading State: Creating the Check Operations Tool

Now let's build our first tool, which demonstrates how to read state from within a tool function. This tool will check how many operations have been performed and report back to the user:

```
from langchain_core.tools import tool
from langgraph.prebuilt import InjectedState

# Practice reading in state tool
@tool
def check_operations(
    state: Annotated[AgentState, InjectedState]
) -> Command:
    """
    A simple tool that reads state and returns the amount of mathematical
    operations that have been performed in its execution history by tool call.
    Use this if a user asks how many operations something took to run.
    """
    # Get operation counts from state
    add_count = state.get("add_n", 0)
    subtract_count = state.get("subtract_n", 0)
    multiply_count = state.get("multiply_n", 0)
    divide_count = state.get("divide_n", 0)

    # Create returnable operations list
    operations = []
    if add_count > 0:
        operations.append(f"addition ({add_count}x)")
    if subtract_count > 0:
        operations.append(f"subtraction ({subtract_count}x)")
    if multiply_count > 0:
        operations.append(f"multiplication ({multiply_count}x)")
    if divide_count > 0:
        operations.append(f"division ({divide_count}x)")

    return operations
```

Notice a few key things here: we're using `InjectedState` to automatically get the current state passed into our tool function. The state parameter gives us access to the current state, which at its core is just a dictionary (TypedDict). We read the values for each operation counter using `.get()` with a default of zero in case they don't exist yet.

Since we're only reading from state (not writing to it), we can return a simple list of strings. Each string represents an operation count, and we only include operations that have actually been used. This is important because we're returning a regular Python object — not a Command object.

Our custom tool node will need to recognize this type of return and handle it the same way the default ToolNode would: take our list, convert it to a string, wrap it in a ToolMessage, and append that message to the state. This keeps the behavior consistent for tools that don't need to modify state directly.

Creating The Mathematical Operations Tools

Now we'll create the main tools that handle the actual operations. Let's focus on just the addition function first, since all the math operations will work the same way — they just perform different calculations.

If we were building a simple tool that just takes two numbers and returns the result, it would be straightforward. But we need to complicate things by both reading from and writing to the graph's state to track usage statistics.

For these functions, we need to inject the `ToolCallId` because we're returning a Command object to write to state rather than just passing data. When we use Commands, we have to manually create the ToolMessage that would normally be generated automatically. This means we need to know the `ToolCallId` from the execution so we can properly construct the message.

For writing to state, we'll use the Command object to send an update notification that gets handled after our custom tool node executes. We'll be updating two attributes in state: the operational counter (`add_n` in this case) to track how many times the addition tool runs, and the `messages` attribute where we'll append our manually created ToolMessage.

```
from langchain_core.messages import ToolMessage
from langchain_core.tools import InjectedToolCallId
from langgraph.types import Command

# Addition Tool
@tool
def addition(
    a: float,
    b: float,
    tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    """Adds two floats together and returns the value."""
    ans = a + b
    return Command(update = {
        "add_n": 1,
        "messages": [
            ToolMessage(
                content=f"{a} + {b} = {ans}",
                name="addition",
                tool_call_id=tool_call_id
            )
        ]
    })
```

Another thing to note, as you can see, rather than reading the current `add_n` from state and adding one to it — something like this:

```
"add_n": state.get("add_n", 0) + 1
```

We are only passing in a `1`. And why do we do this? In the next section, we will create our custom Tool Node, and the reasoning behind it will all make sense.

For reference, the other three mathematical operations follow the exact same pattern — they just update different counter fields (`subtract_n`, `multiply_n`, `divide_n`) and perform different calculations. Here they are:

```
# Subtraction Tool
@tool
def subtraction(
    a: float,
    b: float,
    tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    """Subtracts float 'b' from float 'a' and returns the value."""
    ans = a - b
    return Command(update = {
        "subtract_n": 1,
        "messages": [
            ToolMessage(
                content=f"{a} - {b} = {ans}",
                name="subtraction",
                tool_call_id=tool_call_id
            )
        ]
    })

# Multiplication Tool
@tool
def multiplication(
    a: float,
    b: float,
    tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    """Multiplies 2 floats together and returns the value."""
    ans = a * b
    return Command(update = {
        "multiply_n": 1,
        "messages": [
            ToolMessage(
                content=f"{a} * {b} = {ans}",
                name="multiplication",
                tool_call_id=tool_call_id
            )
        ]
    })

# Division Tool
@tool
def division(
    a: float,
    b: float,
    tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    """Divides float 'a' by float 'b' and returns the value."""
    ans = a / b
    return Command(update = {
        "divide_n": 1,
        "messages": [
            ToolMessage(
                content=f"{a} / {b} = {ans}",
                name="division",
```

```
                tool_call_id=tool_call_id
            )
        ]
    })
```

Finally, with all of our tools created, let's create a tools list that we'll use later to bind tools to our LLM and a tools dictionary that we'll use in our custom tool node for fast lookup of tool functions by name:

```
# Tools List for LLM Tools Bind
tools = [addition, subtraction, multiplication, division, check_operations]

# Tools Dictionary for O(1) lookup by name
tools_by_name = {tool.name: tool for tool in tools}
```

Creating the Custom ToolNode Function & Adding Reducers to State

Now we'll create the custom ToolNode function that handles our mixed return types. Looking back at the previous section, we need to remember that our tools return both Commands and regular data. This flexibility is important if you're incorporating this approach into an existing project where some tools already exist and return regular data that needs to be wrapped in ToolMessages. Our custom tool node needs to handle both scenarios seamlessly.

Here's something else to keep in mind: our reasoner node (which uses the LLM to decide what tool calls to make in ReAct format) can result in zero to many tool calls. Obviously, if there are no tool calls, our tool node won't execute. But the reasoner can also decide to send either a single tool call or multiple ones in a single run.

If multiple tool calls are sent in a single run, LangGraph interprets this as a concurrent operation and will return multiple Commands in a single list. This makes things difficult because any state object that gets updated from tools utilizing Command objects MUST use a reducer function so it knows how to handle one or many return types. Therefore, we MUST update our state so that we can handle concurrent runs while updating the operation step counting fields.

Why We Need Reducers

Remember how we're only passing `1` to our counter fields instead of reading the current value and incrementing it? This is because when tools run concurrently, multiple Commands might try to update the same state field at the same time. Without reducers, the last update would overwrite the others, and we'd lose count data.

With reducers, LangGraph knows how to combine multiple updates to the same field. For our counters, we want to add all the values together, so we'll use `operator.add` as our reducer function. This operator, in the case of the integer type we provide it, will take all the integer values being sent to a field and add them to the existing value in state, giving us the total sum of all steps. This is why we only pass a `1` value to the operation steps within our tool Command returns—it denotes the addition of another single step for each Command returned.

Let's update our state schema to include reducers:

```
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages
import operator

# Agent Graph State
class AgentState(TypedDict):
```

```
    messages: Annotated[Sequence[BaseMessage], add_messages]
    add_n: Annotated[int, operator.add]
    subtract_n: Annotated[int, operator.add]
    multiply_n: Annotated[int, operator.add]
    divide_n: Annotated[int, operator.add]
```

This is all fine and dandy if you're just summing up information, but what if you need custom logic to handle state updates? Maybe you need to track the maximum value instead of a sum, or handle different data types in a specific way. In that case, you would create a custom reducer.

A custom reducer is just a function that defines how LangGraph should combine multiple updates to the same state field. It receives the current state value and the new update(s), then returns the final value that should be stored in state. The beauty is that you have complete control over this logic.

Here's an example of a custom reducer that does the same thing as `operator.add`, but handles different scenarios more explicitly. I'm showing this because it's important that you understand you're not limited to the basic pre-built functions—you can create any logic you need:

```
def flexible_counter_reducer(current: int, updates) -> int:
    if isinstance(updates, int):
        # Simple case: just add the integer
        return current + updates
    elif isinstance(updates, list):
        # Handle multiple updates at once
        return current + sum(updates)
    else:
        # Fallback: treat as 0
        print(f"Unknown update type: {type(updates)}, treating as 0")
        return current
```

This reducer handles both single updates and lists of updates (which happen during concurrent tool execution), plus it includes error handling to gracefully deal with unexpected data types.

Now, lets update our state one more time to include our custom reducer to at least one of our operations counter fields for later testing:

```
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage
from langgraph.graph.message import add_messages
import operator

# Agent Graph State
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    add_n: Annotated[int, operator.add]
    subtract_n: Annotated[int, operator.add]
    multiply_n: Annotated[int, operator.add]
    divide_n: Annotated[int, flexible_counter_reducer]
```

The Custom ToolNode

Now that we have our state updated to handle concurrent updates, we can successfully implement our custom ToolNode function. If we're following the design principles of a ReAct agent, we should expect a list of tool calls coming from the last updated state — which was the output from our LLM reasoner node.

The idea is that if the LLM generates one to many tool calls (we'll set up a conditional edge for this in the next section), we can check the last message and then execute those expected tool calls to return tool information that helps the LLM answer the user's query.

We set up the function like any other graph node function, where we start by passing in our custom state type as a parameter. We'll need to access the last message in state to know which tools we need to execute. On top of that, we need to create a placeholder list to append all the commands returned by executing these tools, which we'll return to pass on to the reasoner node for state updates.

```
# Define custom ToolNode
def call_tool(state: AgentState):

    # Access the last message in state
    last_msg = state["messages"][-1]

    # Initialize Returnable Commands List
    commands = []
```

Now we need to iterate over the tool calls that are expected. From the last message, we get the `tool_calls` using `getattr()` to return an empty list if the message has no tool calls. The `getattr()` function safely accesses an attribute on an object and returns a default value (in this case, an empty list `[]`) if that attribute doesn't exist. This prevents errors when processing messages that don't contain tool calls.

The `tool_calls` attribute is a list of dictionaries generated by the LLM to specify which tools should be executed. Each dictionary contains several attributes, but the main ones we need are: the name of the tool to call, the generated arguments for the tool call, and the tool call ID which we need for creating ToolMessages.

For each individual `tool_call`, we get the executable tool function from the tools dictionary we created earlier. Then, we create a clean copy of the tool call's arguments. It's essential that you create a clean copy rather than using the tool call arguments directly—we'll discuss the reasoning behind this in more detail shortly.

With those copied arguments, we invoke the tool and get the result. As we talked about earlier, we must account for both Command objects being returned as well as handle all other types by wrapping their content in a ToolMessage and adding that to the messages state. This approach means you don’t need to rely on Command objects being returned every time and stays consistent with how the prebuilt ToolNode handles different return types.

We use `isinstance()` to check if the result is of type `Command`, and if it is, we append it directly to our return list. Otherwise, we take the returned data, stringify it using `str()`, wrap it in a `ToolMessage` object, then wrap that in a `Command` object that updates the messages field of our state by appending that ToolMessage to it. Finally, we add that Command to our returnable list so it will be executed. This is how we handle writing to state from tools in the tool node—these Commands NEED to be returned here since they're expected to be passed between nodes to carry out the state updates they're designed to perform.

```
# Define custom ToolNode
def call_tool(state: AgentState):

    # Access the last message in state
    last_msg = state["messages"][-1]

    # Initialize Returnable Commands List
    commands = []

    # Iterate over tool calls and return commands
    for tool_call in getattr(last_msg, "tool_calls", []):

        # Get Executable from Tools Dictionary
        tool = tools_by_name[tool_call["name"]]

        # CREATE A CLEAN COPY of arguments for invocation
        invocation_args = tool_call["args"].copy()

        # Invoke tool for result
        result = tool.invoke(invocation_args)

        # Check if result is a Command
        if isinstance(result, Command):

            # append result to commands return list
            commands.append(result)

        # Wrap other typed results in ToolMessage & append with command update
        else:
            commands.append(
                Command(update={"messages": [
                    ToolMessage(
                        content=str(result),
                        name=tool_call["name"],
```

```
                        tool_call_id=tool_call["id"]
                    )
                ]})
            )

    # Return List of Commands
    return commands
```

There's one more crucial step to complete our custom ToolNode function: we need to check the tool functions for any state attributes required, allowing us to inject the state into our tool invokes. To do this, we use Python's built-in `inspect` package.

The `inspect` package provides utilities for examining live objects like modules, classes, and functions. In our case, we use `inspect.signature()` to analyze the parameters that our tool functions expect. This allows us to dynamically determine which tools need state injection and which need the tool call ID, then add those parameters to our invocation arguments accordingly.

We check if the tool signature has parameters named `state` or `tool_call_id`, and if so, we add the appropriate values to our copied arguments. This automated injection means our tools get exactly what they need without us having to manually configure each one.

Now here's the crucial part about why we copy the arguments: if you attempt to edit the original tool call arguments directly (like `tool_call["args"]["state"] = state`), you'll encounter a JSON serialization error during execution. LangGraph can't handle certain complex data types like message objects when they're added to the original request structure. By working with a copy, the original remains untouched and serialization-safe, while our copy gets the extra parameters it needs for execution.

```
import inspect

# Define custom ToolNode
def call_tool(state: AgentState):

    # Access the last message in state
    last_msg = state["messages"][-1]

    # Initialize Returnable Commands List
    commands = []

    # Iterate over tool calls and return commands
    for tool_call in getattr(last_msg, "tool_calls", []):

        # Get Executable from Tools Dictionary
        tool = tools_by_name[tool_call["name"]]

        # check if it needs injected state params
        tool_signature = inspect.signature(
            tool.func if hasattr(tool, 'func') else tool
        )

        # CREATE A CLEAN COPY of arguments for invocation
        invocation_args = tool_call["args"].copy()

        # Add Custom Parameter Updates if needed
        if 'state' in tool_signature.parameters:
            # tool_call["args"]["state"] = state << DO NOT DO IT LIKE THIS
            invocation_args["state"] = state
        if 'tool_call_id' in tool_signature.parameters:
            invocation_args["tool_call_id"] = tool_call["id"]

        # Invoke tool for result
        result = tool.invoke(invocation_args)

        # Check if result is a Command
        if isinstance(result, Command):

            # append result to commands return list
            commands.append(result)

        # Wrap other typed results in ToolMessage & append with command update
        else:
            commands.append(
                Command(update={"messages": [
                    ToolMessage(
                        content=str(result),
                        name=tool_call["name"],
                        tool_call_id=tool_call["id"]
                    )
                ]})
            )
```

```
    # Return List of Commands
    return commands
```

With our custom ToolNode complete, we now have a flexible function that handles tools returning Command objects for state updates as well as tools returning regular data types, all while automatically injecting state parameters as needed. Next, let's put it all together and build the complete agent for testing.

Assembling the ReAct Agent

With our tools and custom tool node function complete and able to handle reading from and writing to our graph's state, we can now put together the reasoner node and build out the complete ReAct agent.

First, let's set up our LLM and bind our tools to it so the reasoner node can use them:

```
from langchain_openai import ChatOpenAI

# Instantiate LLM
llm = ChatOpenAI(model="gpt-4o-mini")

# Bind tools to LLM
llm_with_tools = llm.bind_tools(tools)
```

Now we can create the reasoner node. This node is the "brain" of our ReAct agent — it analyzes the current conversation state and decides whether to generate tool calls or output a final response if it thinks it has enough information to answer the user's query. It makes these decisions using the ReAct framework of reasoning and acting.

```
# Reasoner Node
def call_model(state: AgentState):
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}
```

Next, we create the conditional edge function that determines the flow of our agent. This function checks the most recent message, which should be an AIMessage from our reasoner node, to see if it has a `tool_calls` attribute with any tool calls in it.

If the message contains tool calls, we route to our custom tool node to execute them. If there are no tool calls or the list is empty, the LLM will generate a final output and we end the interaction by routing to the `END` node.

```
# Custom Conditional Tools Edge
def should_continue(state: AgentState):
    messages = state["messages"]
    if not messages[-1].tool_calls:
        return "end"
    return "continue"
```

Finally, let's assemble our entire ReAct agent with all the functionality we've created. If you're not familiar with what we're doing here, I'd recommend going back and understanding the basics of LangGraph and creating an agent graph first.

Here's what we're doing at a high level: we're creating a `StateGraph` that defines the flow of our agent, adding our nodes (the reasoner and tool executor), and connecting them with edges that control the execution flow.

The conditional edge creates the ReAct loop: the agent reasons about what to do, potentially uses tools, then reasons about the tool results, and repeats until it has enough information to provide a final answer.

```
from langgraph.graph import StateGraph, START, END

# Define a new graph with our state
workflow = StateGraph(AgentState)

# Nodes
workflow.add_node("llm", call_model)
workflow.add_node("tools",  call_tool)

# Edges
workflow.add_edge(START, "llm")
workflow.add_conditional_edges(
    "llm",
    should_continue,
    {
        "continue": "tools",
        "end": END
    }
)
workflow.add_edge("tools", "llm")

# Compile Graph
graph = workflow.compile()
```

Now we have compiled our graph, and are ready to test it out!

Testing Our Agent

Test Run 1 — Tool Concurrency

Before we add in a system prompt, we can test that the custom tool node we created is handling multiple tool calls. Granted, it will generate the tool calls incorrectly and give us the wrong answer, but it will show us if there are any concurrency issues in our agent.

Let's create our initial state and pass messages with a single user message containing a math problem that has two or more operations. We'll start the operation counters at zero so they count correctly from the beginning.

```
# Create Intial State for Graph Execution
initial_state = {
    "messages": [
        ("user", "What is 3+3*2?")
    ],
    "add_n": 0,
    "subtract_n": 0,
    "multiply_n": 0,
    "divide_n": 0,
}
```

Now we can stream our graph execution and watch the message history as it's created in real-time:

```
# Call our graph with streaming to see the steps
last_state = None
for state in graph.stream(initial_state, stream_mode="values"):
    last_state = state
    last_message = state["messages"][-1]
    last_message.pretty_print()
```

We also capture the `last_state` variable so you can examine the final state after graph execution if you want to dig deeper into what happened:

```
# Check final state
from pprint import pprint
pprint(last_state)
```

From this run, it should have successfully executed — although the answer will most likely be incorrect — with concurrent tool calls and a final answer should have been outputted. If you run into any errors, double-check your code against the examples above to see what might have gone wrong.

Test Run 2 — Operations Tool Check

We can check that our operations tracking tool is working by passing a simple initial state with a user asking about the mathematical operations count. We'll put some dummy data into the operations count state variables to simulate previous executions:

```
initial_state = {
    "messages": [
        ("user", "How many mathematical operations were executed in history?")
    ],
    "add_n": 2,
    "subtract_n": 0,
    "multiply_n": 3,
    "divide_n": 0,
}
```

Executing the graph using this `initial_state` should return a correct answer along the lines of five operations with two additions and three multiplications.

Test Run 3 — Complete Agent Check

For this final test, we can evaluate a combination of the mathematical operation tools, their accuracy, and the operations usage tracking tool all working together. First, we'll add a system prompt to better explain the agent's purpose and guide it to solve problems step-by-step using proper order of operations:

```
system_prompt = """You are a helpful assistant that can perform mathematical cal

IMPORTANT RULES FOR MATH:
1. Always follow the order of operations (PEMDAS/BODMAS)
2. Only use ONE tool call at a time
3. Think step-by-step through complex expressions
4. For expressions like "3+3*2", first do multiplication (3*2=6), then addition
5. Wait for each calculation result before proceeding to the next step

Example approach for "3+3*2":
1. "I need to follow order of operations. First, I'll multiply 3*2"
2. [Use multiplication tool: 3*2]
3. "Now I have the result 6. Next I'll add 3+6"
4. [Use addition tool: 3+6]
5. "The final answer is 9"

Always explain your reasoning and work through problems one step at a time."""
```

Now we create an `initial_state` that includes the system prompt and a different mathematical expression to see if our complete system works correctly:

```
initial_state = {
    "messages": [
        ("system", system_prompt),
        ("user", "What is 100/25/2? After you answer that, tell me how many oper
    ],
    "add_n": 0,
    "subtract_n": 0,
    "multiply_n": 0,
    "divide_n": 0,
}
```

Once again, we execute the graph and stream the output to watch the step-by-step process:

```
# call our graph with streaming to see the steps
last_state = None
for state in graph.stream(initial_state, stream_mode="values"):
    last_state = state
    last_message = state["messages"][-1]
    last_message.pretty_print()
```

When I ran this test, I ended up with a final output of: “The result of 100/25/2 is 2 and it took 2 operations.” This shows our agent is now working correctly — it performed the calculations step-by-step, got the right answer, and accurately tracked how many operations it used, showing that is is properly reading and writing to the agent’s state from our tool functions.

Final Thoughts

That was a long one — my apologies, haha. Throughout this guide, we’ve tackled one of the more challenging (and not as widely documented) aspects of LangGraph: how to manage the graph’s state through tools specifically. What started as a frustrating limitation with the prebuilt ToolNode led me to this comprehensive solution that gives you complete control over how your tools interact with agent state.

The key takeaways from this implementation are the importance of reducers for handling concurrent operations, the flexibility that Command objects provide for complex state updates, and how custom tool nodes can seamlessly handle mixed return types. This approach isn’t just useful for mathematical operations — you can apply these same patterns to any scenario where you need tools to track usage statistics, maintain session data, or coordinate complex workflows. Whether you’re building retrieval systems that need to track search patterns or multi-step processes that require state coordination, the patterns we’ve covered here will serve as a solid foundation for your more advanced LangGraph applications.

Thank you so much for taking the time to make it through this article. If you find any issues or have better insights on this topic, please feel free to share them with me or in the comments below. If you’ve enjoyed this tutorial and want to stay updated on upcoming articles, be sure to follow me here on Medium and connect with me on LinkedIn. Your support motivates me to keep sharing insights and tutorials. Exciting content is on the way — stay tuned! 🚀

For additional help, here are some links to documentation that was used in creating this article
https://ai.google.dev/gemini-api/docs/langgraph-example
https://langchain-ai.github.io/langgraph/how-tos/tool-calling/#short-term-memory
https://langchain-ai.github.io/langgraph/concepts/low_level/#reducers

My Personal Links:
GitHub: **https://github.com/joeyoneill/**
LinkedIn: **https://www.linkedin.com/in/joseph-o-neill-131592164/**
Portfolio: **https://joeyoneill.neocities.org/**

TL;DR

The Problem: LangGraph’s default ToolNode can’t handle tools that need to read from or write to the graph’s state. If you try to use `InjectedState` or return `Command` objects from tools, the default ToolNode completely ignores those tools.

The Solution: Build a custom tool node that handles both regular tool returns and Command objects, plus implement reducers to manage concurrent state updates.

Key Components You Need:

1. State Schema with Reducers

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], add_messages]
    counter: Annotated[int, operator.add]  # Reducer for concurrent updates
```

2. Tools That Read/Write State

```python
@tool
def my_tool(
    param: str,
    state: Annotated[AgentState, InjectedState],
    tool_call_id: Annotated[str, InjectedToolCallId]
) -> Command:
    # Do something with the state
    return Command(update={
        "counter": 1,  # Increment counter
        "messages": [ToolMessage(content="result", name="my_tool", tool_call_id=
    })
```

3. Custom Tool Node Function

```python
def call_tool(state: AgentState):
    commands = []
    for tool_call in getattr(state["messages"][-1], "tool_calls", []):
        tool = tools_by_name[tool_call["name"]]

        # Check if tool needs state injection & Inject Args
        tool_signature = inspect.signature(tool.func if hasattr(tool, 'func') el
        invocation_args = tool_call["args"].copy()
        if 'state' in tool_signature.parameters:
            invocation_args["state"] = state
        if 'tool_call_id' in tool_signature.parameters:
            invocation_args["tool_call_id"] = tool_call["id"]

        # Invoke Tool With Tool Call Args
        result = tool.invoke(invocation_args)

        # Handle both Command objects and regular returns
        if isinstance(result, Command):
            commands.append(result)
        else:
            commands.append(Command(update={"messages": [
                ToolMessage(content=str(result), name=tool_call["name"], tool_ca
            ]}))

    return commands
```

Why This Matters:

- **State Management:** Track usage statistics, session data, or any custom metrics
- **Concurrency:** Reducers handle multiple tools updating the same state fields simultaneously
- **Production Ready:** Works with existing LangGraph patterns and scales properly

The Bottom Line:

If you need tools that can read from or write to LangGraph state, you MUST build a custom tool node. The default ToolNode simply can't handle it, and this approach gives you complete control over state management in your agents.

Langgraph Ai Agent Python Langchain AI

Written by Joey O'Neill

149 followers · 3 following

Developer of Things | GenAI | Python Enthusiast | AI Consulting
joeyoneill.neocities.org https://www.linkedin.com/in/joseph-o-neill-131592164/

Follow

No responses yet

Write a response

What are your thoughts?

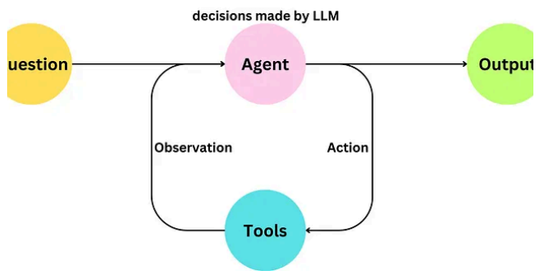
More from Joey O'Neill

Joey O'Neill

Full-stack RAG: FastAPI Backend (Part 1)

All of my articles are 100% free to read. Non-members can read for free by clicking this...

Jan 28 133 1

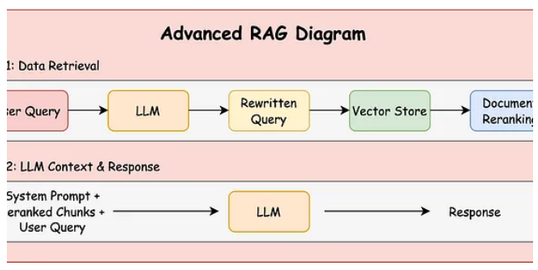


Joey O'Neill

Agentic RAG with Python & LangGraph

All of my articles are 100% free to read. Non-members can read for free by clicking this...

Jan 21 217 2



Joey O'Neill

Advanced RAG with Python & LangChain

In this article, we incorporate pre-retrieval query rewriting and post-retrieval document...

Jan 14 61

Joey O'Neill

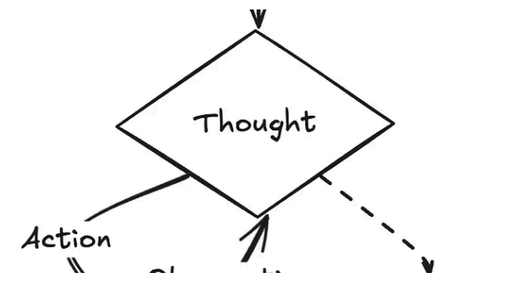
Full-stack RAG: Streaming with FastAPI & React (Part 2)

All of my articles are 100% free to read. Non-members can read for free by clicking this...

Feb 4 6

See all from Joey O'Neill

Recommended from Medium

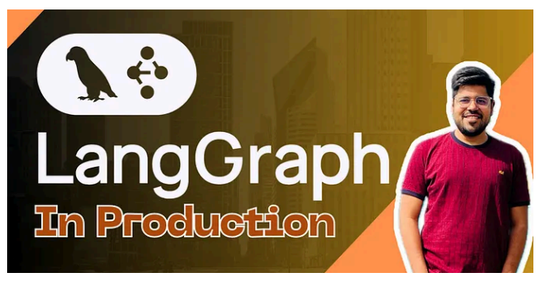


In Artificial Intelligence in Plain E... by Okan Yeni...

Built with LangGraph! #13: ReAct Agent

ReAct Agents Using LangGraph

Jul 22 2



In AlgoMart by Yash Jain

Bringing AI Agents into Production: Why LangGraph Is...

Prototyping an AI agent is fast. Getting one into production—where real-world...

Aug 21 116

In Level Up Coding by Mercy Babatope

Pankaj

How to Build a Multi-Agent Supervisor System with...

Agents are autonomous entities that perceive their environment and take actions to achiev...

May 2 22 1

Open Deep Research by LangChain: The Next Step for AI-...

Open Deep Research is an open-source, multi-agent AI system by LangChain that...

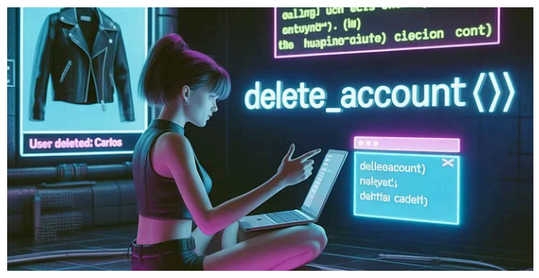
Jul 19 112 1

Pankaj Kumar

Building a ReAct Agent with LangGraph

A hands-on guide to reasoning + acting with LangGraph, one tool call at a time

Aug 24 1



In System Weakness by Aditya Bhatt

BurpSuite Lab: Indirect Prompt Injection 💀🗿

Turned a product review into a silent assassin—Carlos never saw it coming.

Aug 17

See more recommendations