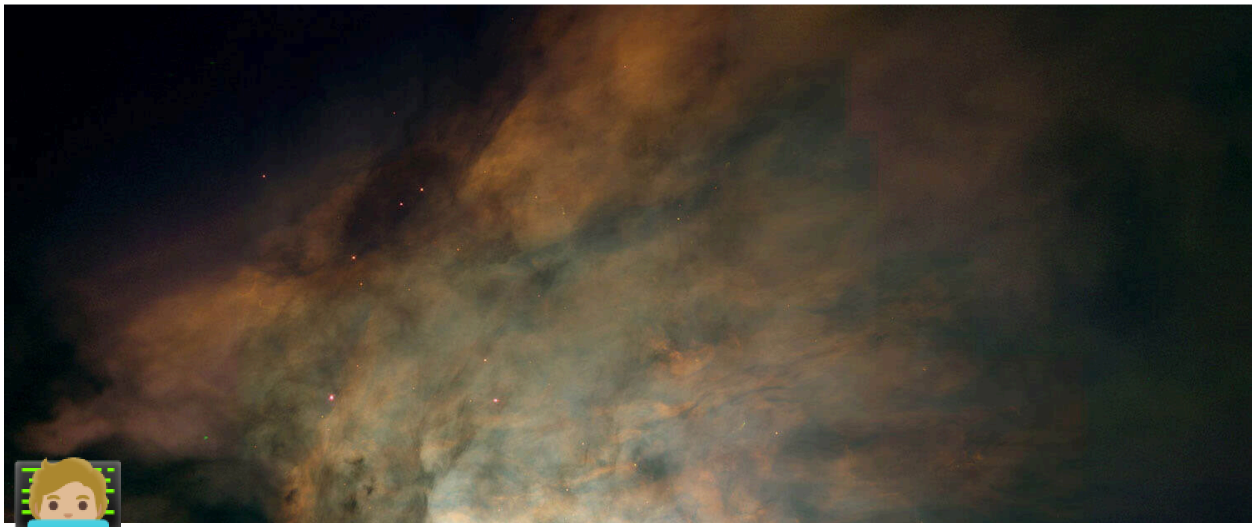
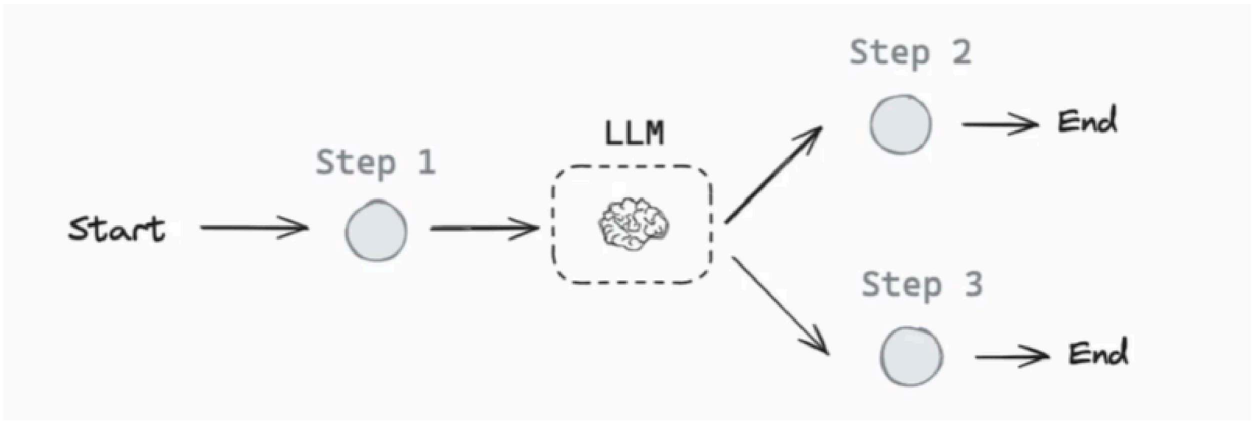


Module 1-Langgraph by Hunar Bhatia

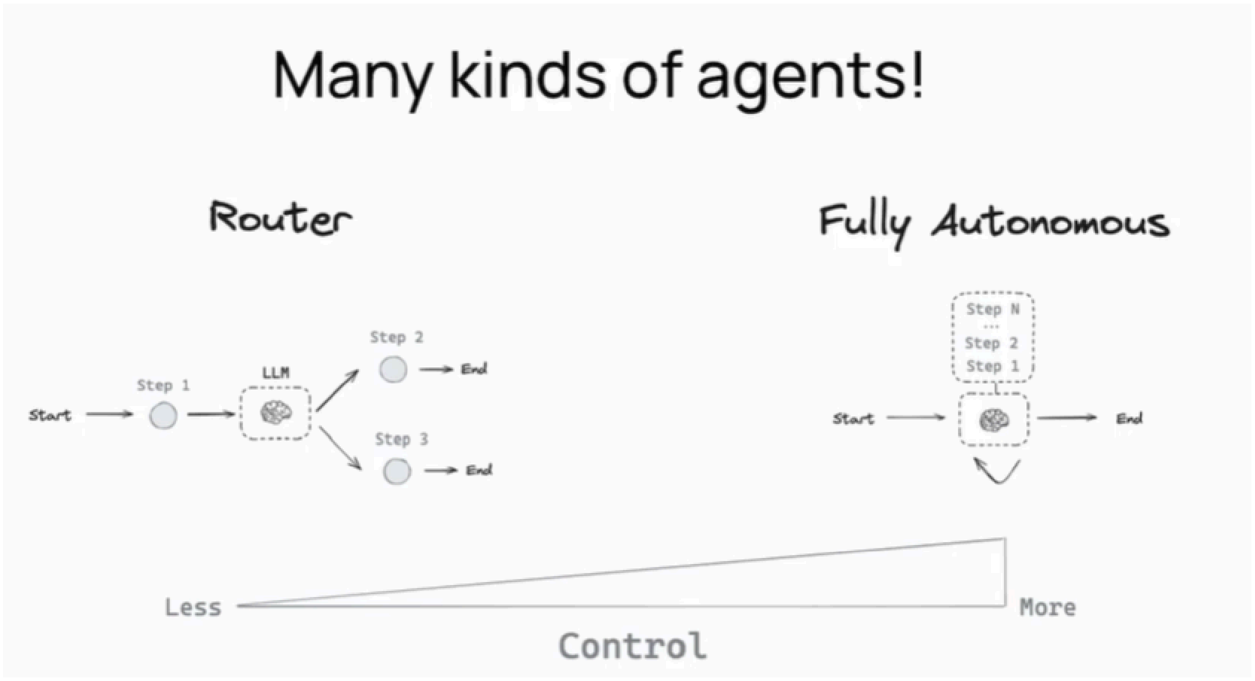
Lesson 1: Motivation

A solitary model alone is fairly limited as it does not have any access to tools, external content and multi-step workflows. So many applications use control flow before and after LLM calls which could be tool calls, retrieval steps and so forth. This control flow forms a chain which you can think of as some set of steps before and after an LLM call.
Now, the nice thing about chains is that they are very reliable so the same set of steps occurs whenever a chain is invoked. But, we do want LLM systems that can pick their very own control flow for certain kinds of problems. This is really what an agent is →



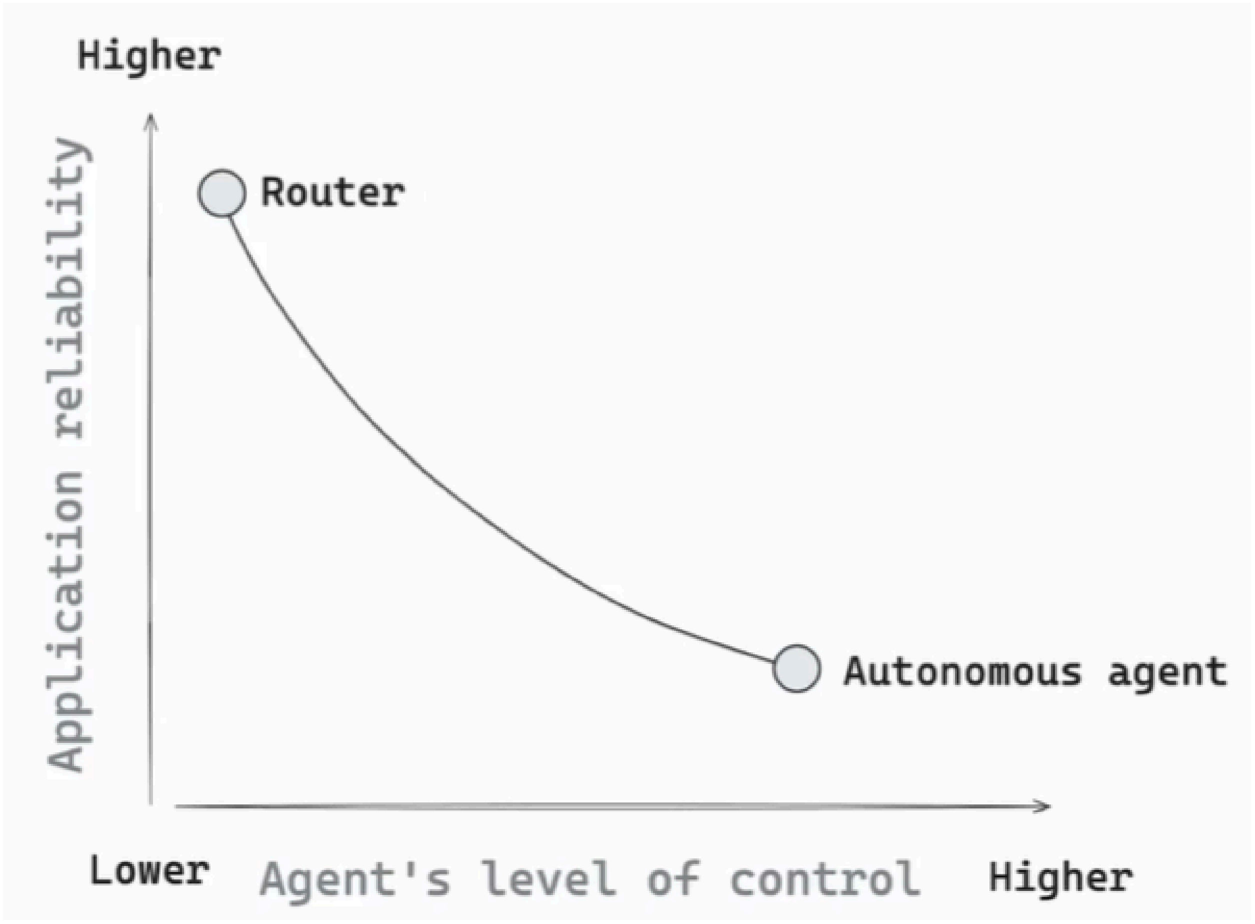
Agent→ Its control flow that is actually defined by an LLM.

So, you have chains which are fixed control flows set by developers and you have agents which are LLM defined control flows.
Now, the thing to think about is that there are many different kinds of workflows so you can think about dialing the control you give to the LLM from low to high. In a router an LLM chooses to perform a single step in a flow and it may choose between a narrow set of options. On the other hand, you have a fully autonomous agent that can choose a sequence of steps through some set of given options, or even it can generate its own steps it can take which is basically auto-generate its next own move based on some potentially available resources.

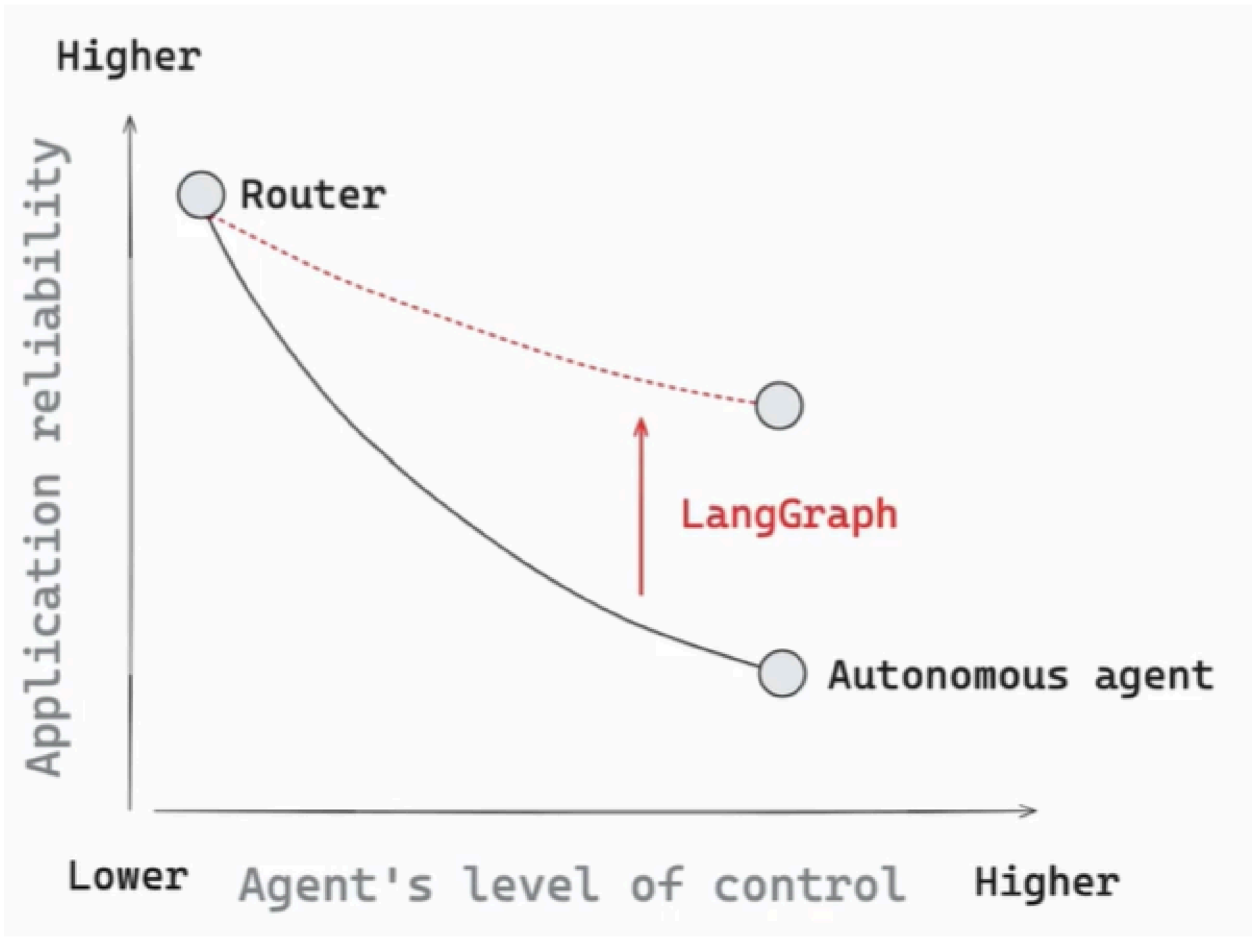


But, we do have some practical challenges here→

We've seen as you ramp up the level of control you give to the LLM, the reliability drops so going from something simpler like a router to going to something much more complicated like a fully autonomous agent, the application does degrade in terms of reliability.

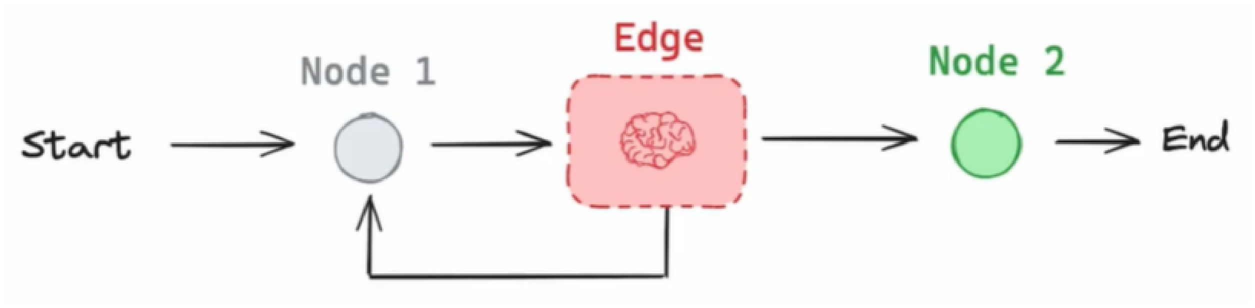


So, this is where Langgraph comes into use → It helps use bend the curve by allowing us to build agents that maintain reliability, even as you push out the level of control you actually give to the LLM or Agent.



Let’s get started with →

In many applications, we want to kind of combine developer intuitions with LLM control, and so you can very easily specify in your application certain steps that you always want to be fixed. We always start at step one and end at step two but we can also inject LLM at certain points turning it into an agent. These are expressed as graphs which contain nodes that represent the steps in our application and edges are just the connectivity between the nodes and there’s a lot of flexibilty for how you lay out nodes and edges.



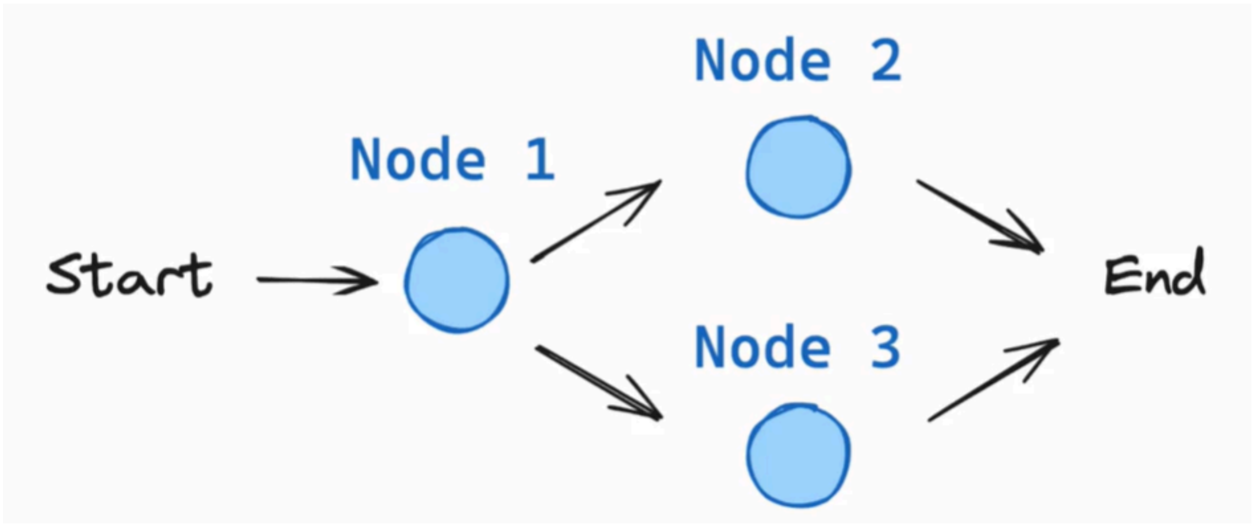
So, there's certain pillars in Langgraph that help us achieve the goals that we are talking about here→

- Persistence
- Streaming
- Human-in-the-loop
- Controllability

These pillars are going to be the cornerstones of the modules in this course and we'll be going through these a lot as we proceed in this code.

Lesson 2- Simple Graph

So let's build a simple graph to introduce the core components of LangGraph.



The edge between start → node1 is a normal edge. The next edge from node1 to node2 and node3 is called a conditional edge meaning that based on some condition we define, we either choose the branch leading to node2 or node3.
Next, we install the langgraph module and then we define the state that is baiscally the object that we pass between the nodes and edges of our graph. Here, we define the state as a simple dictionary and it is going to have one key graph_state which is shown as follows-:

```
from typing_extensions import TypedDict

class State(TypedDict):
    graph_state: str
```

Now, next we need to define our nodes we are going to be taking 3 nodes here →

```
def node_1(state):
    print("---Node 1---")
    return {"graph_state": state['graph_state'] +" My fav color is→"}

def node_2(state):
    print("---Node 2---")
    return {"graph_state": state['graph_state'] +"Red"}

def node_3(state):
    print("---Node 3---")
    return {"graph_state": state['graph_state'] +"Blue"}
```

Remember that State is a dictionary so here we override the value of "graph_state" and append something new(of our own).

We know for a fact that edges are how we connect nodes.

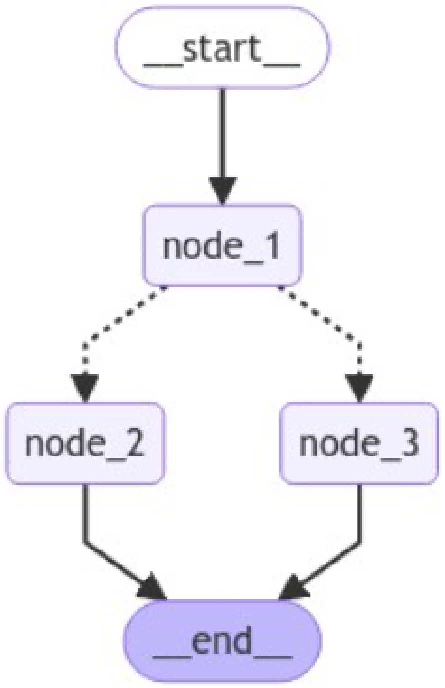
```
import random
from typing import Literal

def decide_mood(state) → Literal["node_2", "node_3"]:

    # Often, we will use state to decide on the next node to visit
    user_input = state['graph_state']

    # Here, let's just do a 50 / 50 split between nodes 2, 3
    if random.random() < 0.5:

        # 50% of the time, we return Node 2
        return "node_2"

    # 50% of the time, we return Node 3
    return "node_3"
```

So here we were using a random function to decide which node to go to next from the conditional edge.

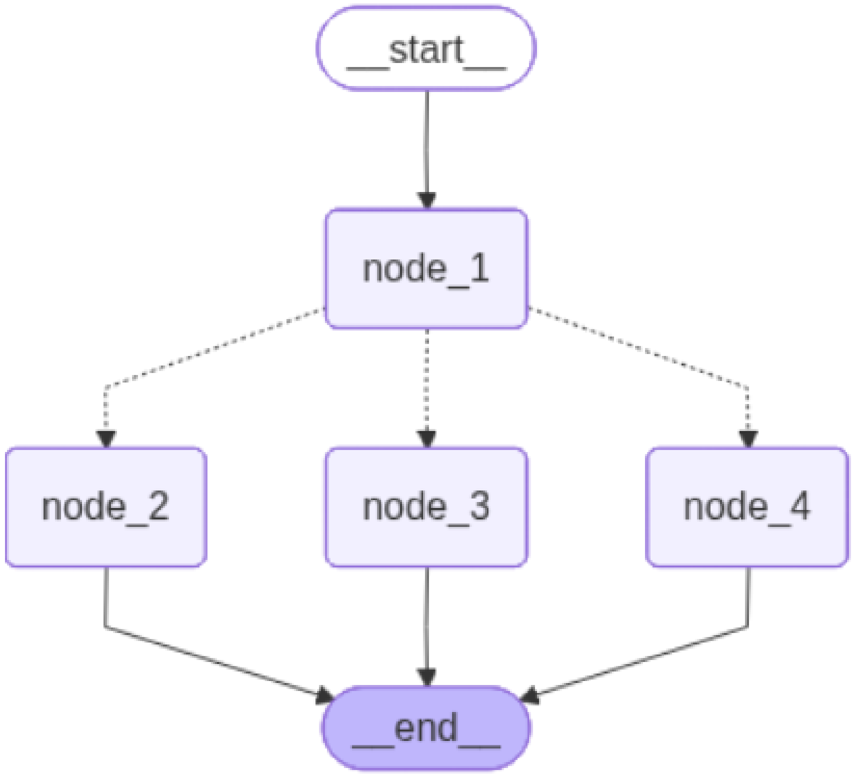
Now, we are going to put all that together into a graph as follows-:
We would be using the `StateGraph` class to do that and we'll initialize it with `State` that we defined above and we'll call it `builder` .Then, we add our nodes and name them accordingly. Finally, we define our logic for our starting from START and finishing up at END before we compile our graph and perform the basic checks and display it.

```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END
```

```
# Build graph
builder = StateGraph(State)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

# Logic
builder.add_edge(START, "node_1")
builder.add_conditional_edges("node_1", decide_mood)
builder.add_edge("node_2", END)
builder.add_edge("node_3", END)

# Add
graph = builder.compile()

# View
    display(Image(graph.get_graph().draw_mermaid_png()))
```



Now, the ***TWEAKING PART***→ I tried doing this with 3 nodes in the conditional edge to see how it turns out. In order to do this I changed the conditional statement a bit. This is how it turned out to be→



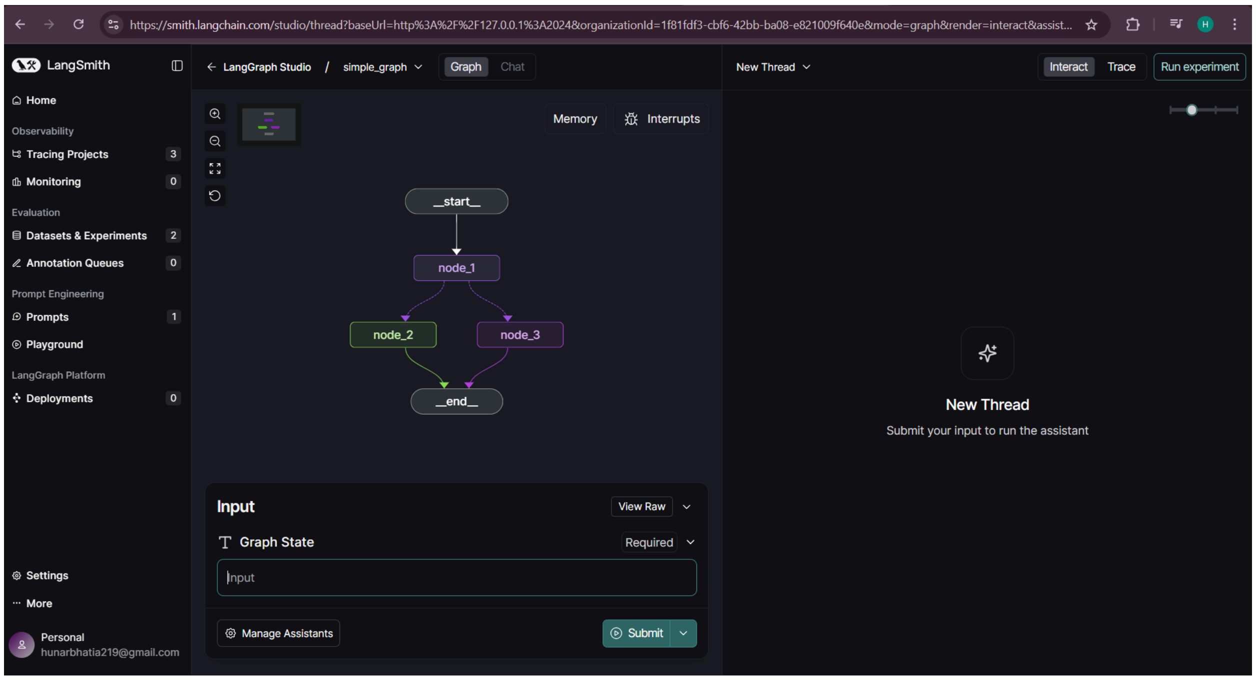
Now, graphs implement a runnable protocol. This is just a standard way to execute various langchain components. So, this protocol is a few standard methods which is higly convenient and one of them is `invoke()` .All we need to do is invoke our graph with an initial condition or an initial starting value for our state.So, recall our state is a dictionary and it has one key graph_state and we can just simply invoke it with a starting condition as follows →

```
graph.invoke({"graph_state" : "You wanna know what my fav color is?"})
```

```
---Node 1---
---Node 2---
{'graph_state': 'You wanna know what my fav color is? My fav color is->Red'}
```

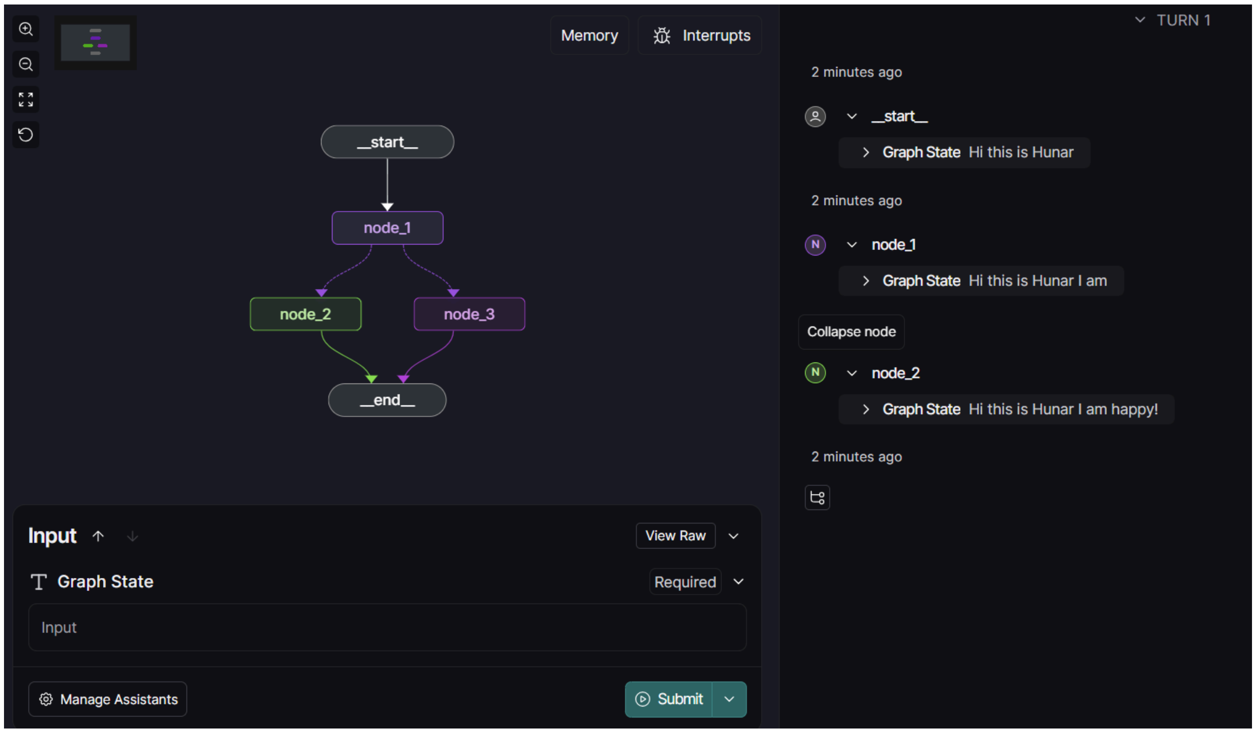
Lesson 3:LangSmith Studio

Something went wrong with my OpenAI API key and it took me literal 2 hours to get here→



This is the LangSmith studio interface.

So, we'll be using simple_graph for now. There's an input field which allows us to input values to the state just like we did in the notebook in the first video except that it is an IDE this time so we can visually interact with it.
The thread on the right side basically groups different invocations in my graph together.
It is like basically the history of any run of the graph.
On clicking submit, we see that we have a thread which shows us what happened in a brief way where we can see what each node does by expanding on the arrow.



2 minutes ago

__start__

Graph State Hi this is Hunar

2 minutes ago

node_1

Graph State Hi this is Hunar I am

node_2

Graph State Hi this is Hunar I am happy!

2 minutes ago

Graph State

So, it is a really nice way to visualize what happens in your graphs and various runs of your graphs.

So, this was a basic introduction to how we're gonna be using studio.

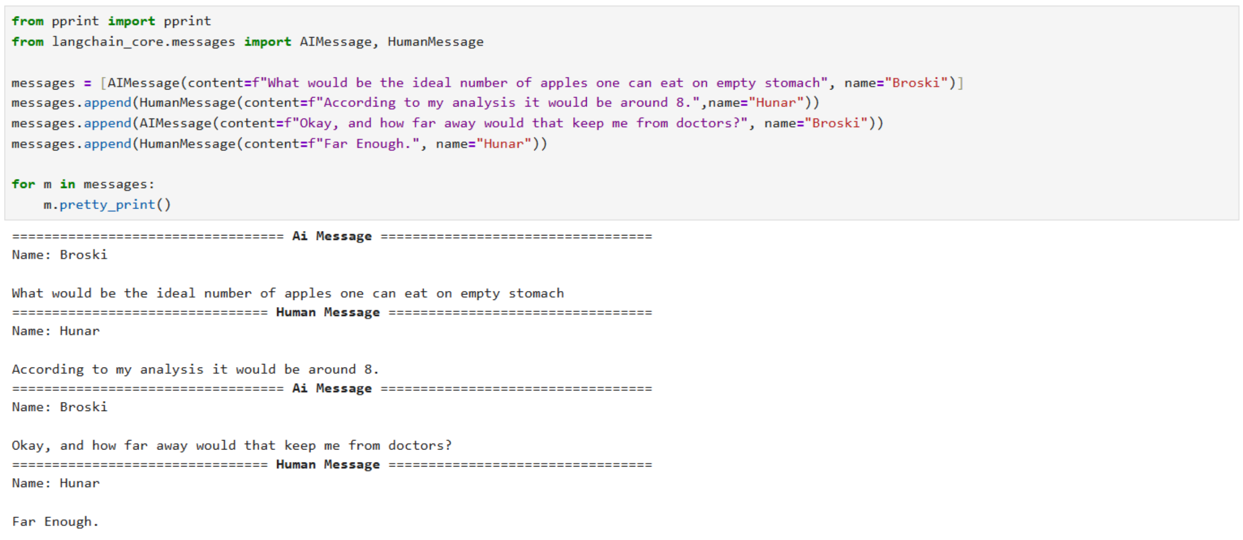
Lesson 4:Chain

So we previously built a normal graph with nodes, normal edges and conditional edges.

Now, let's build up to a simple chain that combines 4 key concepts-:

- Using chat messages in our graph.
- Using chat models.
- Binding tools to our LLM.
- Executing tool calls in our graph.

Getting started, first is messages → So, chat models interact with messages. Here's an example -: I can create a list of messages which in our given case was the conversation between an AI and a human. We can assign a name to each message as well as content and just print it out. We change this as follows →

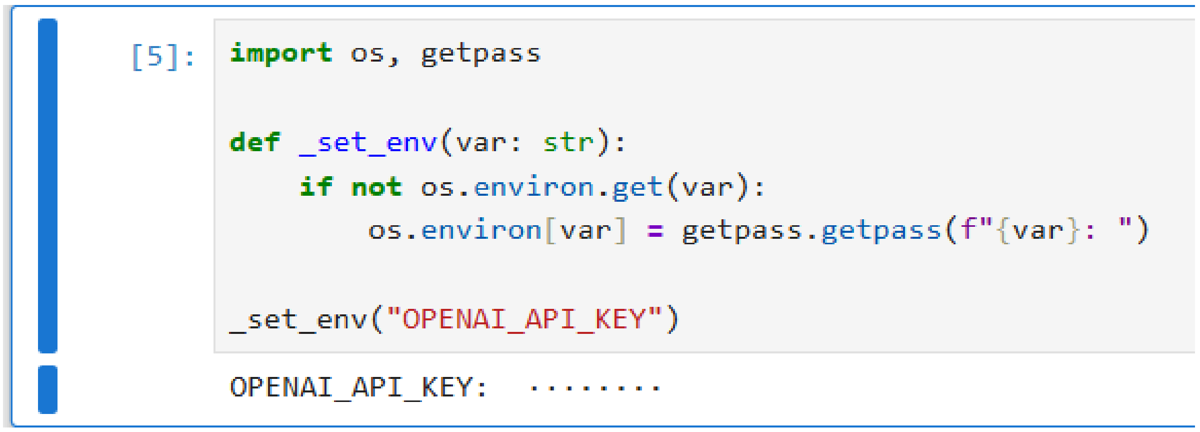


```
from pprint import pprint
from langchain_core.messages import AIMessage, HumanMessage

messages = [AIMessage(content=f"What would be the ideal number of apples one can eat on empty stomach", name="Broski")]
messages.append(HumanMessage(content=f"According to my analysis it would be around 8.",name="Hunar"))
messages.append(AIMessage(content=f"Okay, and how far away would that keep me from doctors?", name="Broski"))
messages.append(HumanMessage(content=f"Far Enough.", name="Hunar"))

for m in messages:
    m.pretty_print()
```

```
================================== Ai Message ==================================
Name: Broski

What would be the ideal number of apples one can eat on empty stomach
================================ Human Message =================================
Name: Hunar

According to my analysis it would be around 8.
================================== Ai Message ==================================
Name: Broski

Okay, and how far away would that keep me from doctors?
================================ Human Message =================================
Name: Hunar

Far Enough.
```

I changed the conversation in the part where we learned about messages into a funny conversation on apples and doctors and observed the output.

Going further, I can take this list of messages and pass it directly to a chat model. First we make sure our OpenAI key is set, then we make some imports and then we specify the LLM model to work with which in this case was GPT 4o.

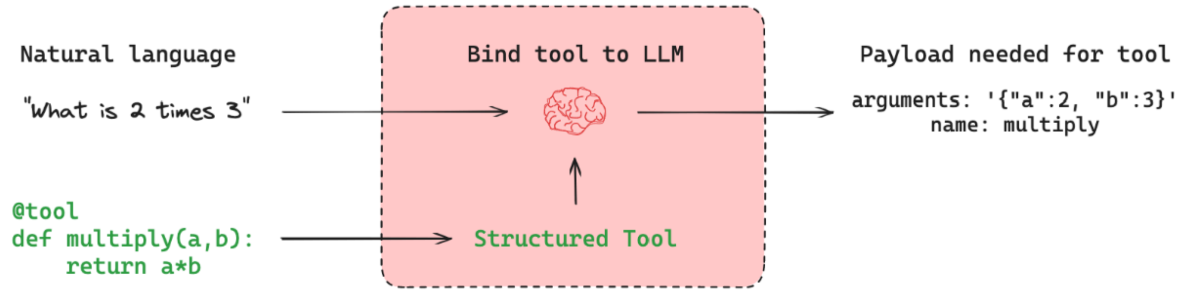


```
[5]: import os, getpass

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

     _set_env("OPENAI_API_KEY")

     OPENAI_API_KEY:  ········
```

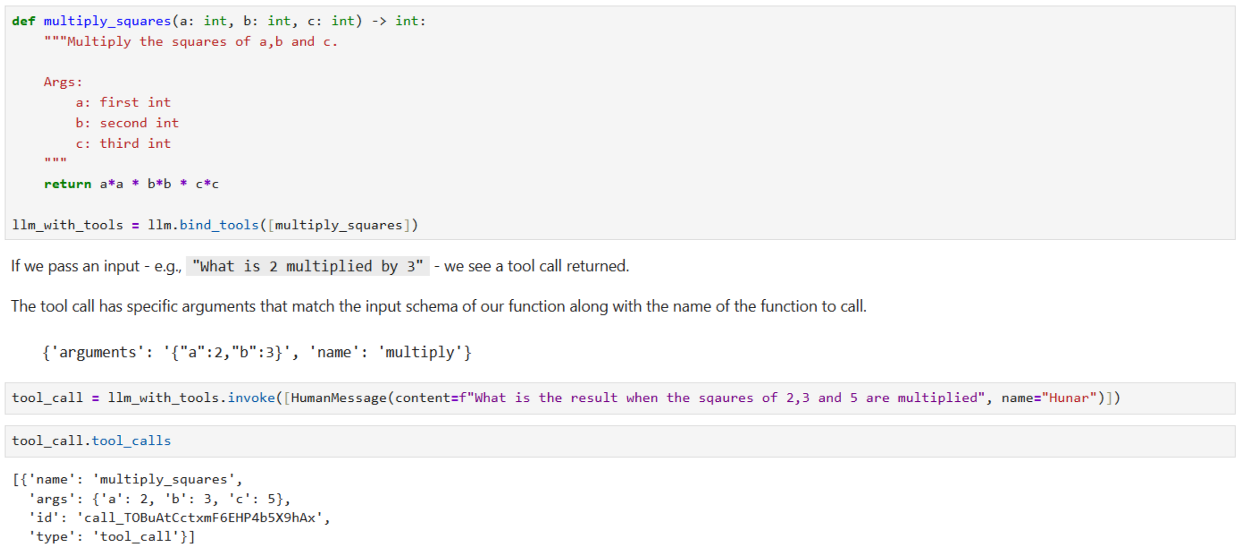
We then get the result and it is basically an AI message and the content which is a string from the LLM and then in the next cell we get the response metadata.

Now, let’s introduce the idea of tools which is another way to use chat models. The idea is simple→ Sometimes, we want to connect our chat model with some external tool like an API that requires particular payload to run. For example-: We take a function and define it and bind it to the LLM and now we’ll have an LLM that would have access to awareness of that function.

As shown in the diagram below, we can take natural language in and it can produce the payload necessary to actually run that tool or function out.



This is what we get from our own function that we defined in the notebook-:



```
def multiply_squares(a: int, b: int, c: int) -> int:
    """Multiply the squares of a,b and c.

    Args:
        a: first int
        b: second int
        c: third int
    """
    return a*a * b*b * c*c

llm_with_tools = llm.bind_tools([multiply_squares])
```

If we pass an input - e.g., `"What is 2 multiplied by 3"` - we see a tool call returned.

The tool call has specific arguments that match the input schema of our function along with the name of the function to call.

```
{'arguments': '{"a":2,"b":3}', 'name': 'multiply'}
```

```
tool_call = llm_with_tools.invoke([HumanMessage(content=f"What is the result when the sqaures of 2,3 and 5 are multiplied", name="Hunar")])
```

```
tool_call.tool_calls
```

```
[{'name': 'multiply_squares',
  'args': {'a': 2, 'b': 3, 'c': 5},
  'id': 'call_TOBuAtCctxmF6EHP4b5X9hAx',
  'type': 'tool_call'}]
```

Here, we saw how to bind tools to chat models to produce tool calls outputs.

Now, let’s start rolling these pieces all into Langgraph. The first idea is how we can use messages as a graph state.

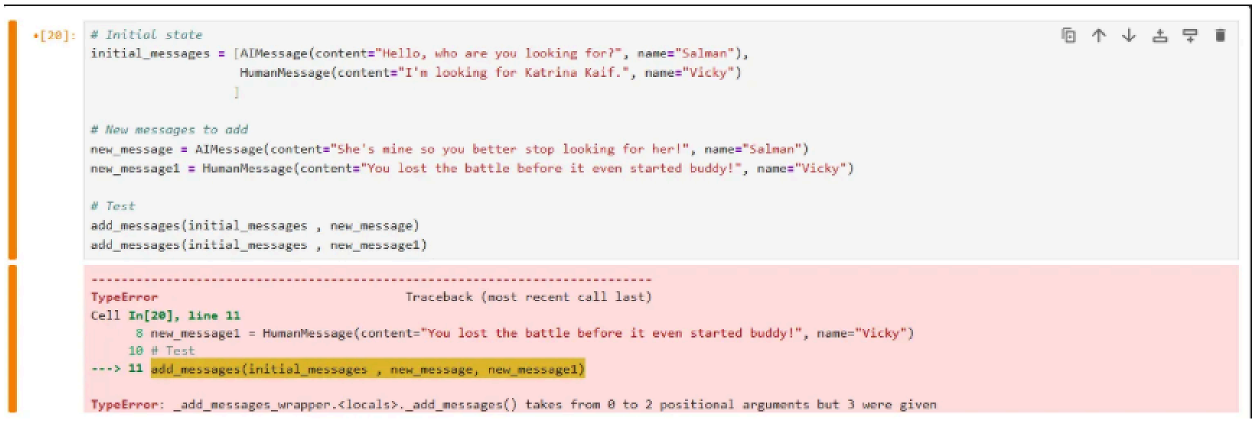
Now, we define this class messages_state and it is a typed_dict and it has one key messages which is just a list of messages. Do not forget that we overwrite the value of this key when we perform state updates by default and LangGraph but in our particular case we don’t wanna do that. Instead, we wanna append this list everytime. For example our chat model produces an output, we wanna append to a state so it preserves a full history of the conversation. This is what motivates the idea of reducer functions.

So, when we define our state in LangGraph, we can have a single key messages, we can have it but we can actually annotate it with what we call reducer function, which tells LangGraph to append to this messages list when it recieves a new message.
Since having a list of messages in graph state is so common, LangGraph has a pre-built `MessagesState` !

`MessagesState` is defined:

- With a pre-build single `messages` key.
- This is a list of `AnyMessage` objects.
- It uses the `add_messages` reducer.

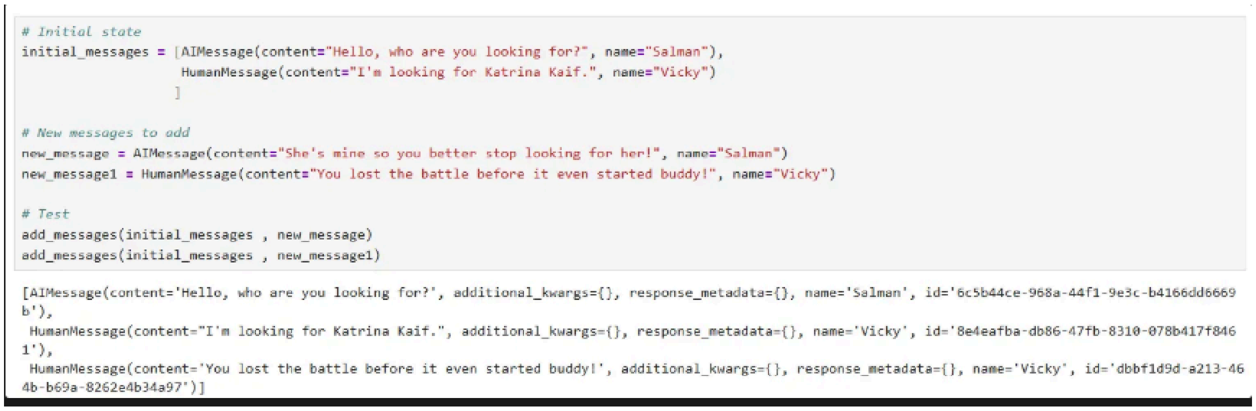
Tested the reducer with my custom messages based on a bollywood joke to see how the reducer function appends the response. I also tried adding two messages to the function which gave an error where I learned that the reducer function takes 0-2 arguments and 3 were given to it.



```
•[20]: # Initial state
       initial_messages = [AIMessage(content="Hello, who are you looking for?", name="Salman"),
                           HumanMessage(content="I'm looking for Katrina Kaif.", name="Vicky")
                           ]

       # New messages to add
       new_message = AIMessage(content="She's mine so you better stop looking for her!", name="Salman")
       new_message1 = HumanMessage(content="You lost the battle before it even started buddy!", name="Vicky")

       # Test
       add_messages(initial_messages , new_message)
       add_messages(initial_messages , new_message1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 11
      8 new_message1 = HumanMessage(content="You lost the battle before it even started buddy!", name="Vicky")
     10 # Test
---> 11 add_messages(initial_messages , new_message, new_message1)

TypeError: _add_messages_wrapper.<locals>._add_messages() takes from 0 to 2 positional arguments but 3 were given
```

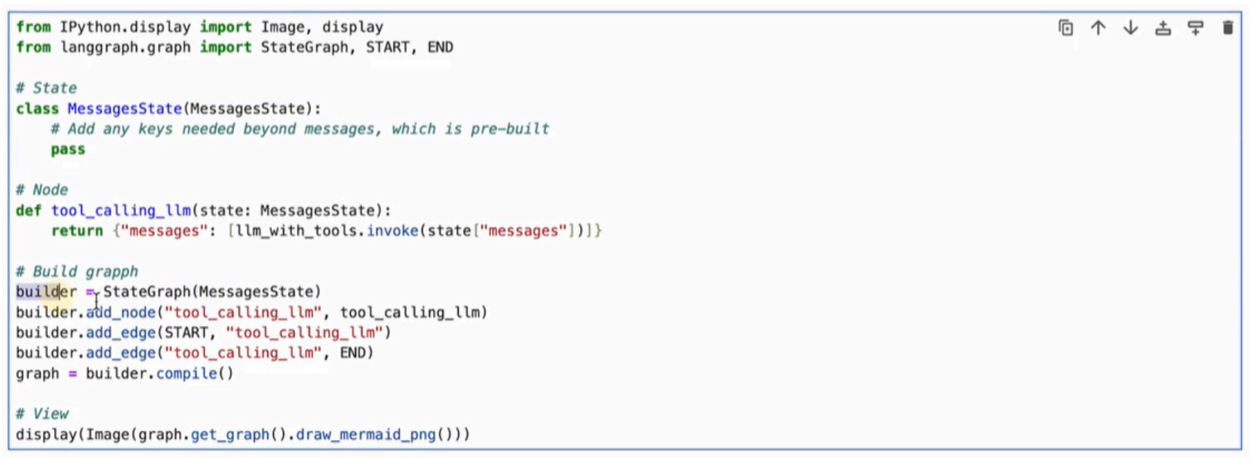
So, instead of doing this, I wrote two functions and then tried executing it which led me to the result that it showed output for only the most recently called function.



```
# Initial state
initial_messages = [AIMessage(content="Hello, who are you looking for?", name="Salman"),
                    HumanMessage(content="I'm looking for Katrina Kaif.", name="Vicky")
                    ]

# New messages to add
new_message = AIMessage(content="She's mine so you better stop looking for her!", name="Salman")
new_message1 = HumanMessage(content="You lost the battle before it even started buddy!", name="Vicky")

# Test
add_messages(initial_messages , new_message)
add_messages(initial_messages , new_message1)
```

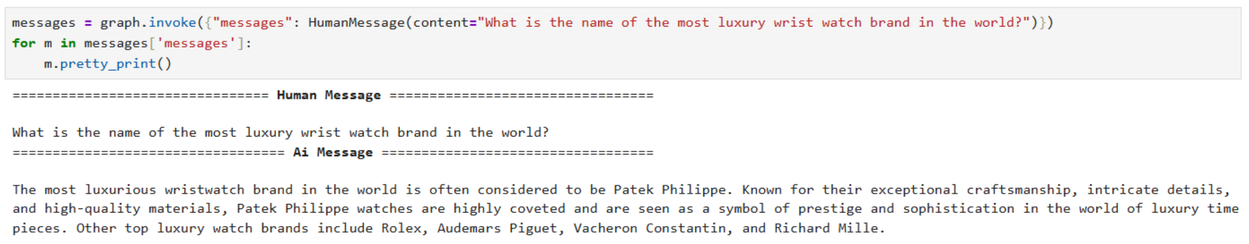
```
[AIMessage(content='Hello, who are you looking for?', additional_kwargs={}, response_metadata={}, name='Salman', id='6c5b44ce-968a-44f1-9e3c-b4166dd6669
b'),
 HumanMessage(content="I'm looking for Katrina Kaif.", additional_kwargs={}, response_metadata={}, name='Vicky', id='8e4eafba-db86-47fb-8310-078b417f846
1'),
 HumanMessage(content='You lost the battle before it even started buddy!', additional_kwargs={}, response_metadata={}, name='Vicky', id='dbbf1d9d-a213-46
4b-b69a-8262e4b34a97')]
```

Finally, we roll this into a graph as follows→



```
from IPython.display import Image, display
from langgraph.graph import StateGraph, START, END

# State
class MessagesState(MessagesState):
    # Add any keys needed beyond messages, which is pre-built
    pass

# Node
def tool_calling_llm(state: MessagesState):
    return {"messages": [llm_with_tools.invoke(state["messages"])]}

# Build grapph
builder = StateGraph(MessagesState)
builder.add_node("tool_calling_llm", tool_calling_llm)
builder.add_edge(START, "tool_calling_llm")
builder.add_edge("tool_calling_llm", END)
graph = builder.compile()

# View
display(Image(graph.get_graph().draw_mermaid_png()))
```

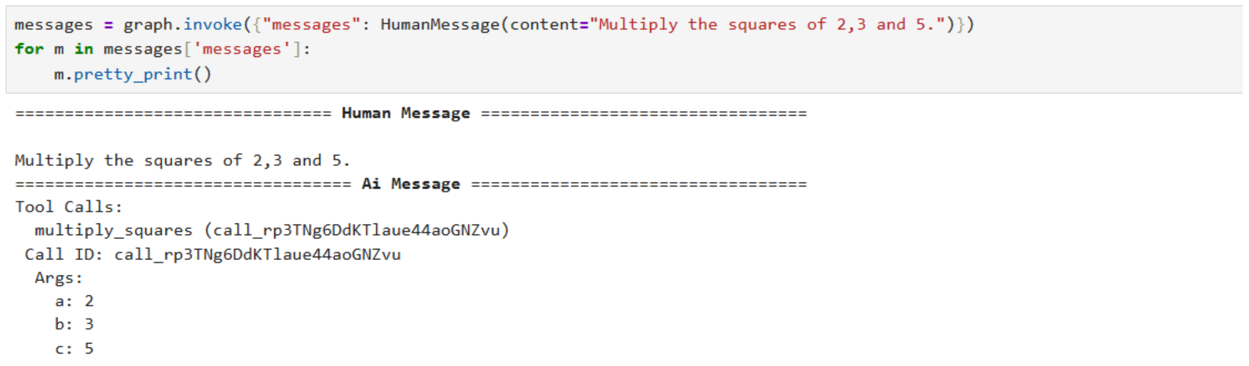
Now, let’s try invoking out graph with 2 different inputs(of my own) →



```
messages = graph.invoke({"messages": HumanMessage(content="What is the name of the most luxury wrist watch brand in the world?")})
for m in messages['messages']:
    m.pretty_print()
```

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

What is the name of the most luxury wrist watch brand in the world?
================================== Ai Message ==================================

The most luxurious wristwatch brand in the world is often considered to be Patek Philippe. Known for their exceptional craftsmanship, intricate details,
and high-quality materials, Patek Philippe watches are highly coveted and are seen as a symbol of prestige and sophistication in the world of luxury time
pieces. Other top luxury watch brands include Rolex, Audemars Piguet, Vacheron Constantin, and Richard Mille.
```

In the second invoke function, we try tool calling with our custom function and it worked as follows→



```
messages = graph.invoke({"messages": HumanMessage(content="Multiply the squares of 2,3 and 5.")})
for m in messages['messages']:
    m.pretty_print()
```

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

Multiply the squares of 2,3 and 5.
================================== Ai Message ==================================
Tool Calls:
  multiply_squares (call_rp3TNg6DdKTlaue44aoGNZvu)
 Call ID: call_rp3TNg6DdKTlaue44aoGNZvu
  Args:
    a: 2
    b: 3
    c: 5
```

Lesson 5:Router

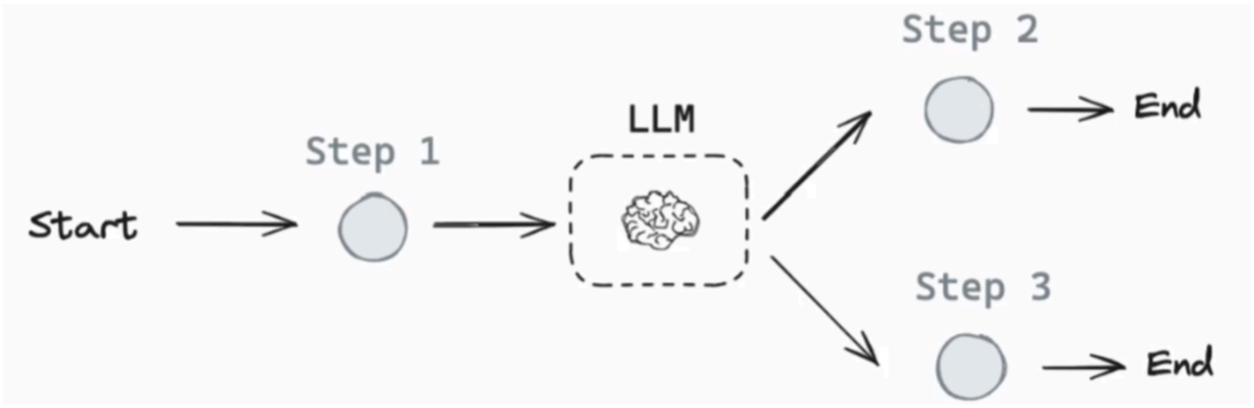
We built a grpah that uses messages as state and a chat model with bound tools.

To do either of the two things-:

- Return a tool call.
- Return a natural language response.

If an input is relevant to the tool, it returns a tool call otherwise just responds directly.Now, you can think of this as a general router as the chat model is routing between a tool call or a direct response.

This is what a typical router looks like where the LLM chooses one or two potential paths based on the input-:



So, let's actually extend what we did using two new ideas-:

1. We'll add a node that'll actually call our tool so if the model responds with a tool call we actually can execute that call in a separate node.
2. We can add a conditional edge that let's you look at the chat model output and make a decision.

Starting with the notebook→

We first call the API and make the imports required and then we set our custom function for tool calling.I altered the function that we were calling as a tool and changed it from multiply to weighted random number chooser for 3 inputs as follows→

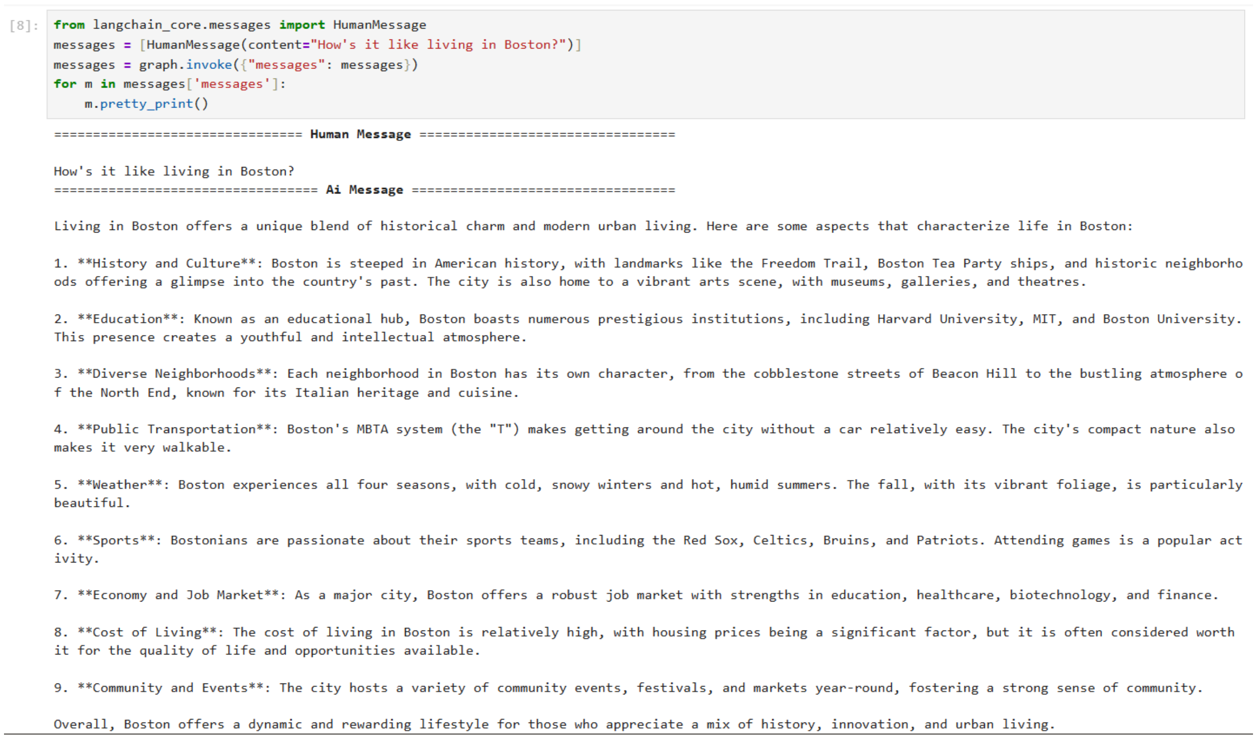
```
from langchain_openai import ChatOpenAI
import random

def weighted_choice(a: int, b: int, c: int) → str:
    """Choose one of A, B, or C randomly, weighted by their values.

    Args:
        a: weight for option A
        b: weight for option B
        c: weight for option C
    """
    total = a + b + c
    r = random.uniform(0, total)
    if r < a:
        return "A wins!"
    elif r < a + b:
        return "B wins!"
    else:
        return "C wins!"

llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools([weighted_choice])
```

We have the built-in ToolNode of LangGraph which is what we use to call our tool and all we need to do is just pass that function to our ToolNode.
We also have the pre-built tools_condition and basically it is a conditional edge, so it’s gonna look at the output of our LLM, and if that output is a tool call, it’ll route to our ToolNode.

```python
from langchain_core.messages import HumanMessage
messages = [HumanMessage(content="Choose a weighted random from 4,7,8.")]
messages = graph.invoke({"messages": messages})
for m in messages['messages']:
    m.pretty_print()
```

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

Choose a weighted random from 4,7,8.
================================== Ai Message ==================================
Tool Calls:
  weighted_choice (call_Tc9Z8iYgHmTdGIWmo40SwySt)
 Call ID: call_Tc9Z8iYgHmTdGIWmo40SwySt
  Args:
    a: 4
    b: 7
    c: 8
  weighted_choice (call_h2W15NUHrHZ5RsnySipuMC1X)
 Call ID: call_h2W15NUHrHZ5RsnySipuMC1X
  Args:
    a: 4
    b: 7
    c: 8
================================= Tool Message =================================
Name: weighted_choice

A wins!
================================= Tool Message =================================
Name: weighted_choice

B wins!
```

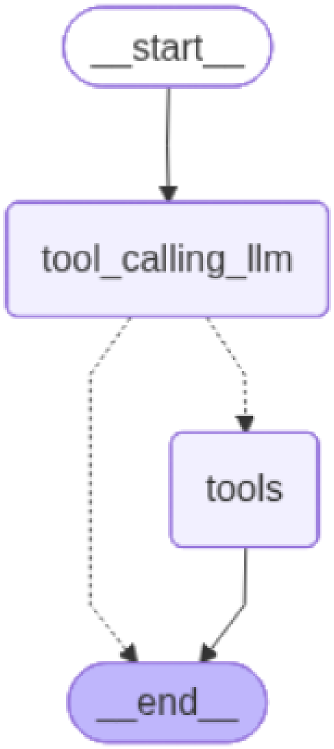
And instead of having the tool call, if we provide it with a direct message inside the content we'll see that it responds normally.



```
[8]: from langchain_core.messages import HumanMessage
     messages = [HumanMessage(content="How's it like living in Boston?")]
     messages = graph.invoke({"messages": messages})
     for m in messages['messages']:
         m.pretty_print()
```

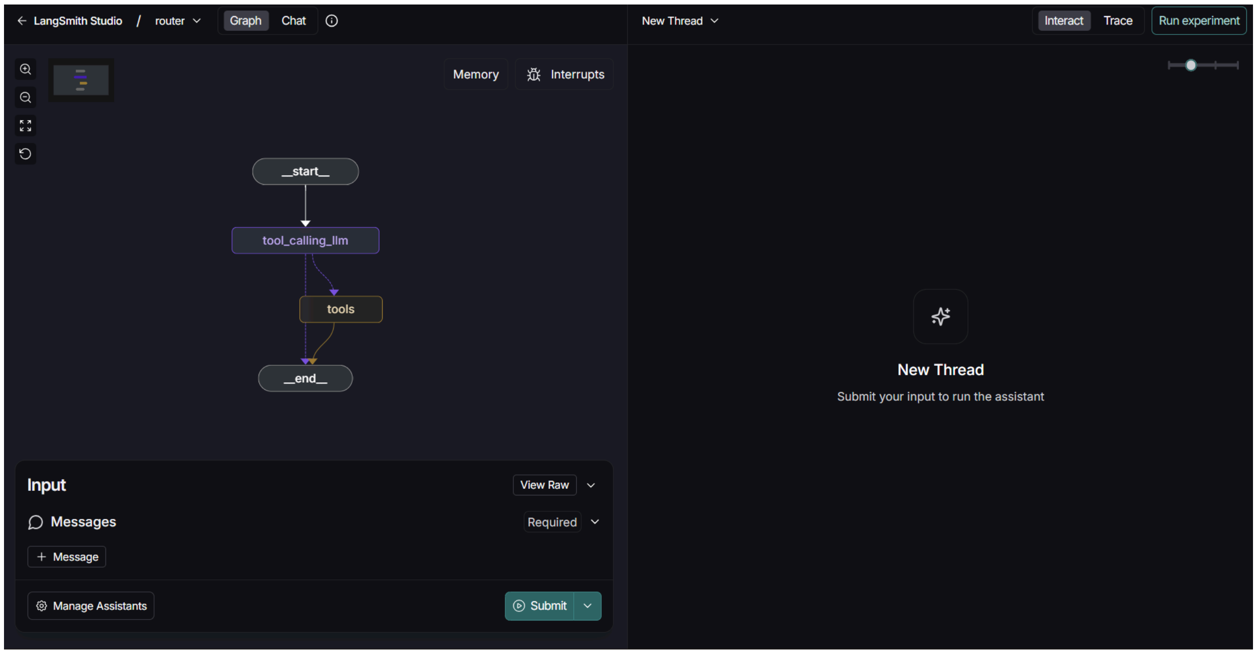
```
================================ Human Message =================================

How's it like living in Boston?
================================== Ai Message ==================================

Living in Boston offers a unique blend of historical charm and modern urban living. Here are some aspects that characterize life in Boston:

1. **History and Culture**: Boston is steeped in American history, with landmarks like the Freedom Trail, Boston Tea Party ships, and historic neighborho
ods offering a glimpse into the country's past. The city is also home to a vibrant arts scene, with museums, galleries, and theatres.

2. **Education**: Known as an educational hub, Boston boasts numerous prestigious institutions, including Harvard University, MIT, and Boston University.
This presence creates a youthful and intellectual atmosphere.

3. **Diverse Neighborhoods**: Each neighborhood in Boston has its own character, from the cobblestone streets of Beacon Hill to the bustling atmosphere o
f the North End, known for its Italian heritage and cuisine.

4. **Public Transportation**: Boston's MBTA system (the "T") makes getting around the city without a car relatively easy. The city's compact nature also
makes it very walkable.

5. **Weather**: Boston experiences all four seasons, with cold, snowy winters and hot, humid summers. The fall, with its vibrant foliage, is particularly
beautiful.

6. **Sports**: Bostonians are passionate about their sports teams, including the Red Sox, Celtics, Bruins, and Patriots. Attending games is a popular act
ivity.

7. **Economy and Job Market**: As a major city, Boston offers a robust job market with strengths in education, healthcare, biotechnology, and finance.

8. **Cost of Living**: The cost of living in Boston is relatively high, with housing prices being a significant factor, but it is often considered worth
it for the quality of life and opportunities available.

9. **Community and Events**: The city hosts a variety of community events, festivals, and markets year-round, fostering a strong sense of community.

Overall, Boston offers a dynamic and rewarding lifestyle for those who appreciate a mix of history, innovation, and urban living.
```

So, basically, the below graph depicts the the control flow of the entire process where the LLM decides whether to make the tool call or end the process.



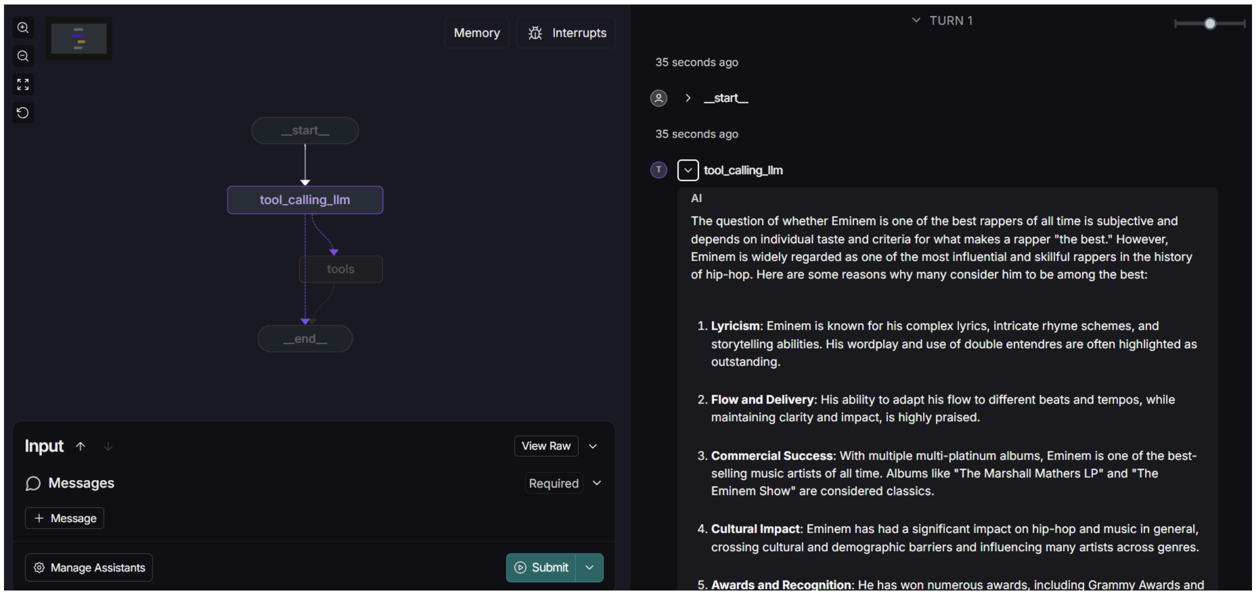
Now, we’ll see how it works in the studio as follows →

So, we’ll open our studio and go over to router and the interface should be looking something like this→



So, we write a new message of our own to test how it works with a router via threads→

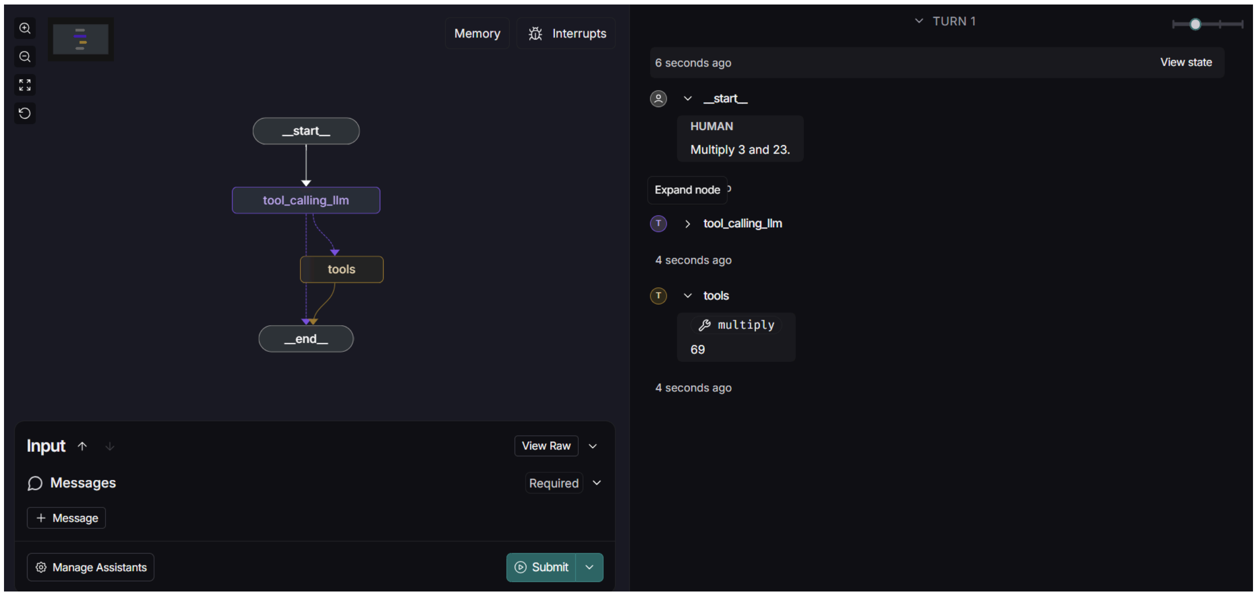
Here we see that tool_calling_llm does not use the tools part from the threads as it was a direct response and the tool wasn’t needed.



The question of whether Eminem is one of the best rappers of all time is subjective and depends on individual taste and criteria for what makes a rapper "the best." However, Eminem is widely regarded as one of the most influential and skillful rappers in the history of hip-hop. Here are some reasons why many consider him to be among the best:

1. **Lyricism**: Eminem is known for his complex lyrics, intricate rhyme schemes, and storytelling abilities. His wordplay and use of double entendres are often highlighted as outstanding.
2. **Flow and Delivery**: His ability to adapt his flow to different beats and tempos, while maintaining clarity and impact, is highly praised.
3. **Commercial Success**: With multiple multi-platinum albums, Eminem is one of the best-selling music artists of all time. Albums like "The Marshall Mathers LP" and "The Eminem Show" are considered classics.
4. **Cultural Impact**: Eminem has had a significant impact on hip-hop and music in general, crossing cultural and demographic barriers and influencing many artists across genres.
5. **Awards and Recognition**: He has won numerous awards, including Grammy Awards and

On the other hand, when we pass it a message having something to do with the tool, it uses the tools part in the thread shown as follows→



6 seconds ago

__start__

HUMAN

Multiply 3 and 23.

tool_calling_llm

4 seconds ago

tools

multiply

69

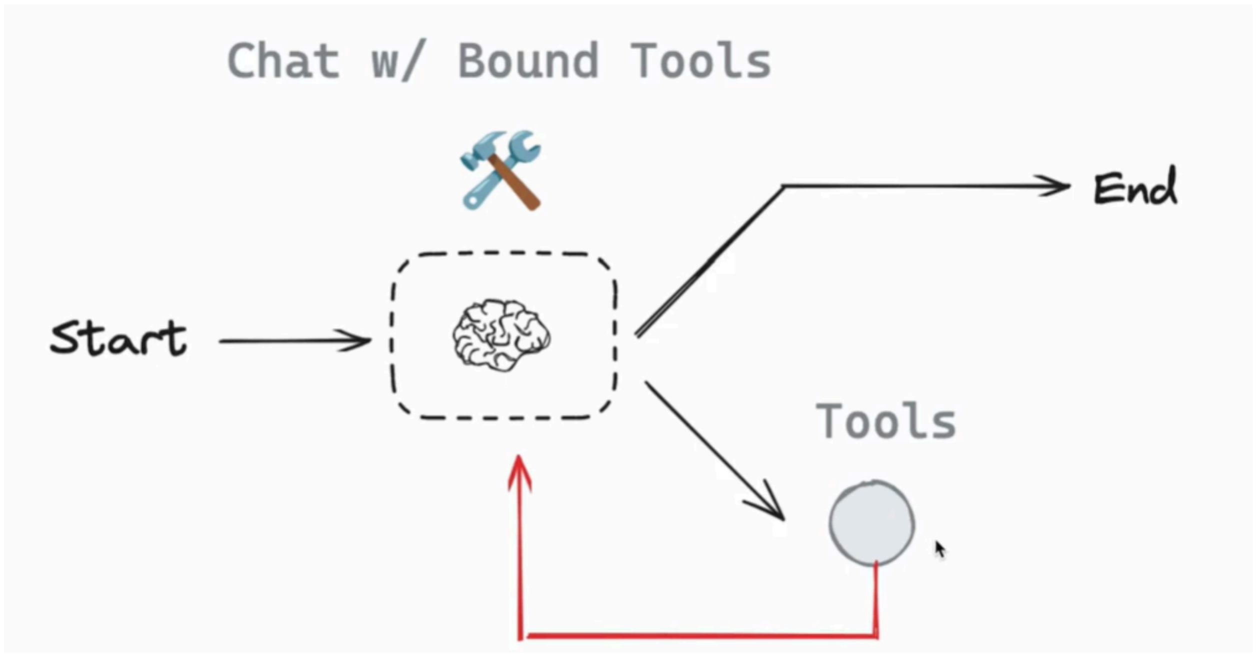
4 seconds ago

Lesson 6:Agent

We are already done with the basics of a router. We can make one simple modification to this router to turn it into one of the more famous generic agent architectures.

So, from the router if we send the tools based information back to the LLM it would turn it into ReAct architecture which has 3 components →

1. act→ let the model call specific functions.
2. observe→ pass the tool output back to the model.
3. reason→let the model reason about the tool output to decide what to do next.

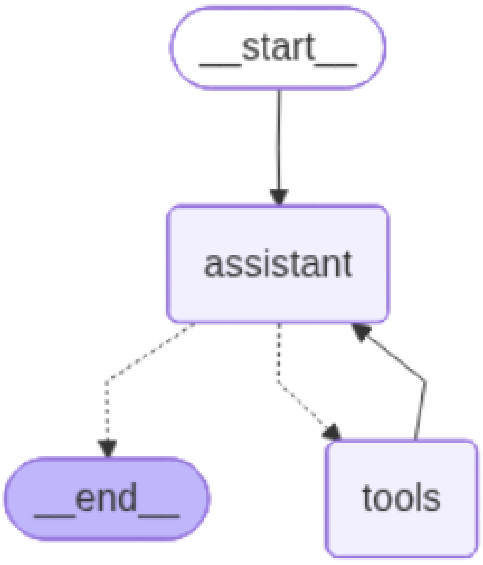


This general purpose architecture can be applied to many types of tools.

So, basically the model can continue to call tool until it sees a solution fit for the purpose or determines that the problem is solved and then it can just return natural language response and then end. This does have cases where we can apply a max limit for recursion but this was the intuition behind the whole concept and let’s go further now.

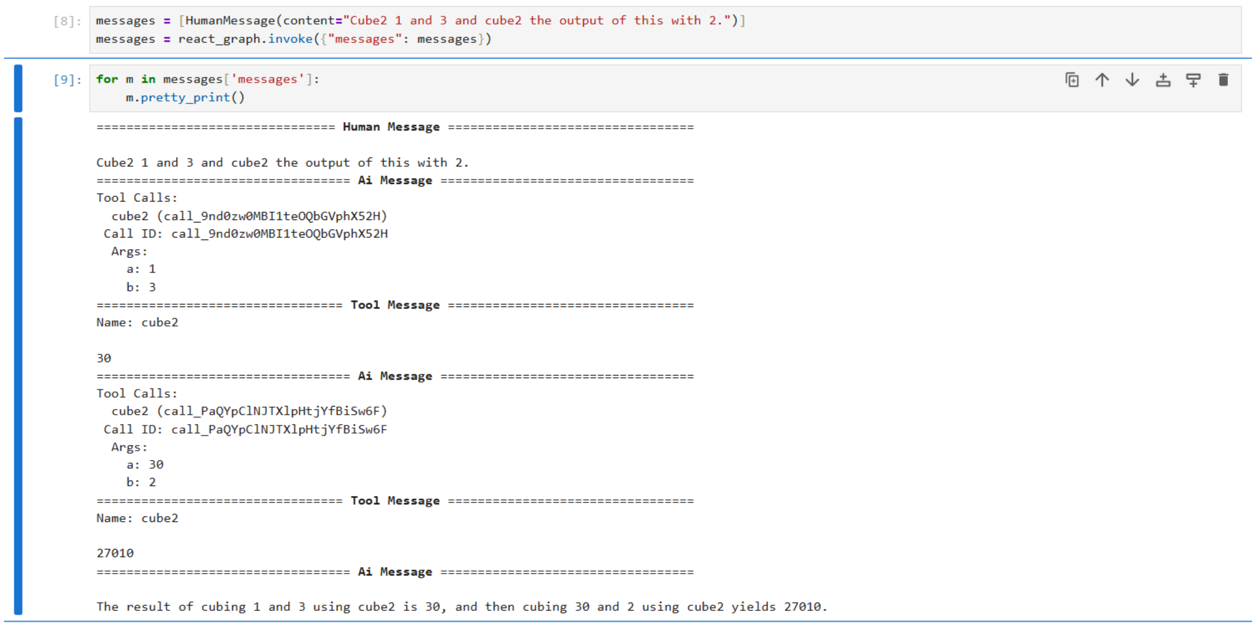
After the installations and imports, we create 3 tools (of our own) and bind them to the model.

After this, we work with the system messages and go ahead and build our graph but we see that this time it looks a little bit different→



We can clearly see that here our assistant is the chat model and whenever it goes to access tools, its output gets redirected back towards the assistant.

I changed the content in the human message and tailored it according to my self-built tools.



```
[8]: messages = [HumanMessage(content="Cube2 1 and 3 and cube2 the output of this with 2.")]
     messages = react_graph.invoke({"messages": messages})
```

```
[9]: for m in messages['messages']:
         m.pretty_print()
```

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

Cube2 1 and 3 and cube2 the output of this with 2.
================================== Ai Message ==================================
Tool Calls:
  cube2 (call_9nd0zw0MBI1teOQbGVphX52H)
 Call ID: call_9nd0zw0MBI1teOQbGVphX52H
  Args:
    a: 1
    b: 3
================================= Tool Message =================================
Name: cube2

30
================================== Ai Message ==================================
Tool Calls:
  cube2 (call_PaQYpClNJTXlpHtjYfBiSw6F)
 Call ID: call_PaQYpClNJTXlpHtjYfBiSw6F
  Args:
    a: 30
    b: 2
================================= Tool Message =================================
Name: cube2

27010
================================== Ai Message ==================================

The result of cubing 1 and 3 using cube2 is 30, and then cubing 30 and 2 using cube2 yields 27010.
```

For the rest of the video, we examine the traces in LangSmith. This allowed us to go a little bit further under the hood and look at the various steps in the process.

Lesson 7:Agent with Memory

Previously, we built an agent that can act, observe and reason. In this video, we get introduced to the idea of memory.

I am not making any new tools of my own in this video as this follows the format of the last video where we already did this and hence it would only be repetitive.

Yet again, we define tools and have a system message and get a graph that represents our agent.

Here, we want our agent to have memory i.e. for it to be able to access the conversation history for even more complex computations and deeper conversations.

To address the problem of the state being transient to a single graph execution to access the previous message, persistence comes in play as it introduces the idea of memory.

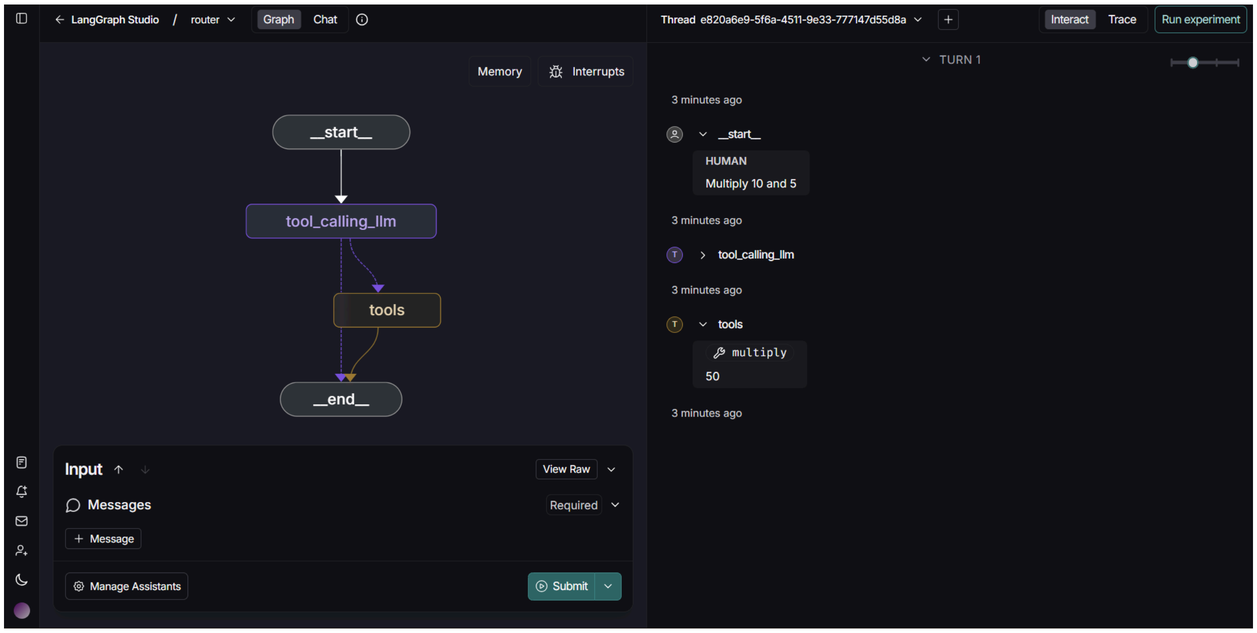
So, all we need to do is just import MemorySaver and then compile our graph with checkpointer set to memory in this case, MemorySaver() →

```
from langgraph.checkpoint.memory import MemorySaver
memory = MemorySaver()
react_graph_memory = builder.compile(checkpointer=memory)
```

What checkpointers actually do is that they save the state of the graph at each point as this checkpoint. So, the checkpoint contains things like the state but it also has other stuff like the next node to go to and some metadata and has a checkpoint_id.

Now these checkpoints can be associated together in what we call a thread.

So, when we specify a thread, specify an input and run → it works normally except for now when we run the follow-up question with the same thread_id, it answers it they way it should be answering by referencing.



3 minutes ago

__start__

HUMAN

Multiply 10 and 5

3 minutes ago

tool_calling_llm

3 minutes ago

tools

multiply

50

3 minutes ago