AI Agents in LangGraph

Last updated: July 4, 2024

Contents

1 Introduction

AI agents are systems designed to *perform tasks autonomously* by interacting with each other and their environment. These agents follow a structured workflow, known as agent workflow, which allows them to achieve better results compared to one-shot requests by *acting iteratively*.

1.1 Agent Workflow

The key design patterns of an agentic workflow include:

- **Planning**: Identifying the steps needed to build and execute the workflow.
- **Tool Use**: Knowing which tools are available and how to utilize them effectively.
- **Reflection**: Iteratively improving results, often using multiple LLMs to provide useful suggestions for enhancing the editing cycle.
- **Multi-agent Communication**: Each LLM can be assigned a specific role, with a unique prompt to perform that role.
- **Memory**: Tracking progress and results over multiple steps to maintain continuity.

Some of these capabilities are inherent to the LLMs themselves, while others are implemented outside of the LLMs by the framework they operate within.

There are two additional capabilities useful in constructing an agent:

- *Human Input Reception*: Allowing agents to receive guidance at critical points.
- *Persistence*: The ability to store the current state of information so that it can be referenced later.

2 Build an Agent from Scratch

We will build an agent based on the ReAct pattern (Reason + Act). First, an LLM thinks about what to do and then decides on the actions to take. This action is then executed in an environment, and an observation is returned. With this observation, the LLM repeats (thinks again about what to do, decides on another action, and so on) until it decides that it is done.

2.1 Generic Agent

To start, let's import all the necessary libraries:

```
import openai
import re
import httpx
import os
from dotenv import load_dotenv

_ = load_dotenv()
from openai import OpenAI
```

Next, we initialize the LLM (we use OpenAI in this case):

```
client = OpenAI()
```

Now, let's create the actual agent by defining a class for the agent:

```
class Agent:
    def __init__(self, system=""):
        self.system = system
        self.messages = []
        if self.system:
            self.messages.append({"role": "system", "content": system})

    def __call__(self, message):
        self.messages.append({"role": "user", "content": message})
        result = self.execute()
        self.messages.append({"role": "assistant", "content": result})
        return result

    def execute(self):
        completion = client.chat.completions.create(
                        model="gpt-4o",
                        temperature=0,
                        messages=self.messages)
        return completion.choices[0].message.content
```

The agent needs to be parameterized by a system agent, so we allow the user to pass that in and save it as an attribute. We also keep track of the list of messages over time (the first being the system message).

The *__call__* method *takes the user's message, appends it to the list*, executes the *execute* function which *returns a response, and appends the response to the list*. The *execute* method is what actually calls the LLM with all the accumulated messages so far and returns the model's response. This is our generic agent.

2.2 ReAct Agent

The ReAct agent requires a specific system prompt:

```
prompt = """
You run in a loop of Thought, Action, PAUSE, Observation.
At the end of the loop you output an Answer
Use Thought to describe your thoughts about the question you have been asked.
Use Action to run one of the actions available to you - then return PAUSE.
Observation will be the result of running those actions.

Your available actions are:

calculate:
e.g. calculate: 4 * 7 / 3
Runs a calculation and returns the number - uses Python so be sure to /
```

```
use floating point syntax if necessary

average_dog_weight:
e.g. average_dog_weight: Collie
returns average weight of a dog when given the breed

Example session:

Question: How much does a Bulldog weigh?
Thought: I should look the dogs weight using average_dog_weight
Action: average_dog_weight: Bulldog
PAUSE

You will be called again with this:

Observation: A Bulldog weights 51 lbs

You then output:

Answer: A bulldog weights 51 lbs
""".strip()
```

Besides explaining the cycle of thinking, acting, and observing the result of the action, it's important to explain in the prompt what actions the agent can take and provide an example of how to use them. In this case, these are two functions we define here:

```
def calculate(what):
    return eval(what)

def average_dog_weight(name):
    if name in "Scottish Terrier":
        return("Scottish Terriers average 20 lbs")
    elif name in "Border Collie":
        return("a Border Collies average weight is 37 lbs")
    elif name in "Toy Poodle":
        return("a toy poodles average weight is 7 lbs")
    else:
        return("An average dog weights 50 lbs")

known_actions = {
    "calculate": calculate,
    "average_dog_weight": average_dog_weight
}
```

If we try it now, we first need to call the agent and then ask a question:

```
abot = Agent(prompt)
result = abot("How much does a toy poodle weigh?")
print(result)
```

The response will be something like: *Thought: I should look up the dog's weight using average_dog_weight for a Toy Poodle. Action: average_dog_weight: Toy Poodle. PAUSE*

This means that the function with that parameter needs to be called:

```
result = average_dog_weight("Toy Poodle")
print(result)
```

We will get the result *a toy poodles average weight is 7 lbs*; we can pass this as the next prompt to the LLM:

```
next_prompt = "Observation: {}".format(result)
abot(next_prompt)
```

If we want more details on what happened throughout the entire process, we can use the *messages* attribute to print all the messages in the conversation:

```
abot.messages
```

The final output will be *Answer: A Toy Poodle weighs 7 lbs.*

Now, let's see a more complex use case:

```
abot = Agent(prompt)
question = """I have 2 dogs, a border collie and a scottish terrier. \
What is their combined weight"""
abot(question)
```

The response will be *Thought: To find the combined weight of a Border Collie and a Scottish Terrier, I need to first find the average weight of each breed and then add those weights together. I'll start by finding the average weight of a Border Collie. Action: average_dog_weight: Border Collie PAUSE*

Execute the action, insert it into a new prompt, and call the agent again:

```
next_prompt = "Observation: {}".format(average_dog_weight("Border Collie"))
abot(next_prompt)
```

The response will be *Thought: Now that I know a Border Collie's average weight is 37 lbs, I need to find the average weight of a Scottish Terrier to calculate the combined weight. Action: average_dog_weight: Scottish Terrier PAUSE*

Again, execute the action, insert it into a new prompt, and call the agent:

```
next_prompt = "Observation: {}".format(average_dog_weight("Scottish Terrier"))
abot(next_prompt)
```

The response will be *Thought: With the average weight of a Border Collie being 37 lbs and a Scottish Terrier being 20 lbs, I can now calculate their combined weight. Action: calculate: 37 + 20 PAUSE*

Execute this last action, which now calls a different function:

```
next_prompt = "Observation: {}".format(calculate("37 + 20"))
abot(next_prompt)
```

The final response will be *Answer: The combined weight of a Border Collie and a Scottish Terrier is 57 lbs.*

2.3 Add Loop

What we've seen so far is the process that needs to be executed. Now we want to put it into a *loop to automate it*.

The first step is to create a *regex* (regular expression) to match the action string. A regular expression is a sequence of characters that define a search pattern. It can be used for string matching, extraction, and replacement. In this case, the regex function will look for specific patterns in the text to determine the action to be taken:

```
action_re = re.compile('^Action: (\w+): (.*)$')
# python regular expression to selection action
```

This splits the LLM response and determines whether we want to execute an action or if it is the final answer.

Now, let's create a *query* function. It takes a question and follows the same process we did manually:

```
def query(question, max_turns=5):
    i = 0
    bot = Agent(prompt) # initialize the agent with the default system prompt
    next_prompt = question
    while i < max_turns:
        i += 1
        result = bot(next_prompt) # call the agent with the last prompt
        print(result)
        actions = [
            action_re.match(a)
            for a in result.split('\n')
            if action_re.match(a)
        ] # parse the responcse and get a list of actions
        if actions:
            # There is an action to run
            action, action_input = actions[0].groups() # function to call and the input
            if action not in known_actions:
                raise Exception("Unknown action: {}: {}".format(action, action_input))
            print(" -- running {} {}".format(action, action_input))
            # First look up in the action dictionary the action to take and
            # we get back a function that we call with the inputs
            observation = known_actions[action](action_input)
            print("Observation:", observation)
            next_prompt = "Observation: {}".format(observation)
        else:
            # If there aren't any action we have the final answer
            return
```

Calling this function, for example, with the following code, will activate all the processes we previously did manually and get the final answer:

```
question = """I have 2 dogs, a border collie and a scottish terrier. \
What is their combined weight"""
query(question)
```

3 LangGraph Components

In the previous lesson, we built an agent from scratch; now we will create a similar one using *LangGraph*, introducing some of its components and features.

3.1 Prompt Templates

PromptTemplate allows you to create reusable prompts. This means you can create a prompt like this and use it multiple times with arbitrary variables.

```
from langchain.prompts import PromptTemplate
prompt_template = PromptTemplate.from_template(
"Tell me a {adjective} joke about {content}.")
```

In the *LangChain* Hub, there are various examples of such prompts, uploaded by users.

3.2 Tools

There are also many tools created by the community. An example is *TavilySearch*, which will be the search tool we use from now on.

```
from langchain_community.tools.tavily_search import TavilySearchResults
tool = TavilySearchResults(max_results=2)

self.tools = {t.name: t for t in tools}
self.model = model.bind_tools([tool])
```

3.3 Introduction to LangGraph

LangGraph is an extension of *LangChain* that supports graphs. The fundamental role of *LangGraph* is to describe and *orchestrate the control flow*, i.e., all the relationships between the components we previously created manually.

Specifically, it allows creating cyclic graphs, which support looping and relationships between elements. Single and multi-agent flows are described and represented as graphs, allowing for extremely controlled "flows."

Additionally, it comes with built-in persistence, which is excellent for maintaining multiple conversations simultaneously or remembering previous interactions and actions. Persistence also allows for introducing human-in-the-loop features.

3.3.1 Graphs

Graphs are composed of three basic elements:

- *Nodes*: Agents or functions.
- *Edges*: Connect nodes.
- *Conditional edges*: Used when you need to make decisions about which node to go to next.

3.3.2 State

One of the most important aspects to understand in *LangGraph* is the state that is tracked over time, often called the **agent state**. This is *accessible to all parts* of the graph and can be stored in the persistence layer.

A simple state is:

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

where we have just a list of messages. The variable *messages* is a sequence of *BaseMessage*, which is a *LangChain* type.

A more complex state is:

```
class AgentState(TypedDict):
    input: str
    chat_history: list[BaseMessage]
    agent_outcome: Union[AgentAction, AgentFinish, None]
    intermediate_steps: Annotated[list[tuple[AgentAction, str]], operator.add]
```

Each of these four variables is of a different type. The first three are not annotated in any way, meaning when a new update is pushed to that variable, it overrides the existing value. Instead, *intermediate_steps*, through the *add* operator, adds the new update without removing the previous ones.

3.4 Building an Agent

Now we will actually build the agent, composed as follows: the first node is *call_openai* which calls the LLM; the second is a conditional edge which checks if there is an action to be taken and we will call it *exists_action*. From here it splits into two paths: either there is no action and it goes to the final node, or there is an action and it goes to the respective action node, then loops back to *call_openai*.

First, load an environment variable, our OpenAI key:

```
from dotenv import load_dotenv
_ = load_dotenv()
```

Next, import all necessary libraries:

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, ToolMessage
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
```

Although *langchain_openai* may seem specific to OpenAI, it can be easily swapped with any other supported model without changing anything else in the code.

The search tool we will use is *Tavily*:

```
tool = TavilySearchResults(max_results=4)
print(type(tool))
print(tool.name)
```

First, initialize it with *max_results=4*, which means we will get a maximum of 4 responses from the search API. The name of the tool is *tavily_search_results_json*, and this is the name the LLM will use to call this tool.

Now, create the agent state, simply an annotated list of messages:

```
class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]
```

Next, create the actual agent. As we have seen, we will need three functions: the first to call OpenAI, the second to check if there is an action to take, and the last to take that action:

```
class Agent:

    def __init__(self, model, tools, system=""):
        self.system = system
        graph = StateGraph(AgentState)
        graph.add_node("llm", self.call_openai)
        graph.add_node("action", self.take_action)
        graph.add_conditional_edges(
            "llm",
            self.exists_action,
            {True: "action", False: END}
        )
        graph.add_edge("action", "llm")
        graph.set_entry_point("llm")
        self.graph = graph.compile()
        self.tools = {t.name: t for t in tools}
        self.model = model.bind_tools(tools)

    def exists_action(self, state: AgentState):
        result = state['messages'][-1]
        return len(result.tool_calls) > 0

    def call_openai(self, state: AgentState):
        messages = state['messages']
        if self.system:
            messages = [SystemMessage(content=self.system)] + messages
        message = self.model.invoke(messages)
        return {'messages': [message]}

    def take_action(self, state: AgentState):
        tool_calls = state['messages'][-1].tool_calls
        results = []
        for t in tool_calls:
            print(f"Calling: {t}")
```

```
            if not t['name'] in self.tools:      # check for bad tool name from LLM
                print("\n ....bad tool name....")
                result = "bad tool name, retry"  # instruct LLM to retry if bad
            else:
                result = self.tools[t['name']].invoke(t['args'])
            results.append(ToolMessage(tool_call_id=t['id'], name=t['name'], content=str(result)))
        print("Back to the model!")
        return {'messages': results}
```

The *agent* is parameterized by three things: the *model* to use, the *tools* to call, and the *system message* (saved as an attribute).

First, initialize the graph with the *AgentState*.

Then, add the node that calls the LLM and the action node (the order in which they are added is not important since there are no links yet). Finally, add the conditional edge. This edge follows the LLM call node and checks if there is an action to perform. If an action is needed, it routes to the action node; if not, it routes to the end node. The *first argument* of the *conditional edge* is the *starting node*, the *second* argument is the *function* that *determines the next step*, and the *third* argument is a *dictionary* that maps the function's response to the next node. Next, add a standard edge between the action node and the LLM node, and establish the entry point of the graph as the LLM node. After completing the setup, compile the graph to transform it into a *LangChain* runnable. Save the compiled graph as a class attribute, along with the tools and model. For the tools, create a dictionary that maps each tool's name to the tool itself. For the model, call *bind_tools*, passing in the list of tools; this informs the model of the available tools.

For the LLM node, create the *call_openai* function: retrieve the list of messages from the agent state, add the system message, and call the model. Finally, return a dictionary with the message obtained from the LLM.

For the action node, create the *take_action* function: if we are in this node, we know that one or more tools need to be called. The *tool_calls* attribute will be present in the last message, which we retrieve. Loop through these tool calls, find the correct tool (using the name in the dictionary), and invoke it with the arguments from the tool call. Add a *ToolMessage* to the state with the results of the invocation and return the new messages obtained from this call. If the *tool_name* is incorrect, print an error message, indicating to the model that the provided tool name is invalid and to retry.

For the conditional edge, create the *exists_action* function: take the last message and return whether there are tools to call (*True* if there are, *False* if not).

This defines our agent; now we can use it:

```
prompt = """You are a smart research assistant. Use the search engine to look up information. \
You are allowed to make multiple calls (either together or in sequence). \
Only look up information when you are sure of what you want. \
If you need to look up some information before asking a follow up question, you are allowed to do that!
"""

model = ChatOpenAI(model="gpt-3.5-turbo")  #reduce inference cost
abot = Agent(model, [tool], system=prompt)
```

We have defined the system prompt and the model to use; for the list of tools, we pass only one, our *TavilySearch* tool.

This graph we have created can also be visualized graphically:

```
from IPython.display import Image

Image(abot.graph.get_graph().draw_png())
```

Now, let's call the agent:

```
messages = [HumanMessage(content="What is the weather in sf?")]
result = abot.graph.invoke({"messages": messages})
```

The *HumanMessage* represents the user message; we add it as the sole element of the message list because the agent state takes a list of messages.

The result we obtain is a list of calls that have been made (first the tool, then back to the model); the final response is the last message, obtained by looking at the *content* attribute:

```
result['messages'][-1].content
```

If we ask a more complex question:

```
messages = [HumanMessage(content="What is the weather in SF and LA?")]
result = abot.graph.invoke({"messages": messages})
```

The tool is called twice (once for each search, through *parallel tool calling*) and then returns to the model.

Let's try one last, even more complex example:

```
query = "Who won the super bowl in 2024? In what state is the winning team headquarters located? \
What is the GDP of that state? Answer each question."
messages = [HumanMessage(content=query)]

model = ChatOpenAI(model="gpt-4o")  # requires more advanced model
abot = Agent(model, [tool], system=prompt)
result = abot.graph.invoke({"messages": messages})
```

First, it calls the tool to find out who won. Then it returns to the model. Next, it calls the tool again to find out the location of the headquarters. It returns to the model and finally makes one last call to find the state's GDP and returns to the model for the final answer.

In this case, the tool calls are *not parallelizable* because the *response* from the *first call* is *needed* to formulate the *second*.

4 Agentic Search Tools

This section explores the differences between agentic search and standard search and how to use the agentic search tool.

4.1 How an Agent Uses Search

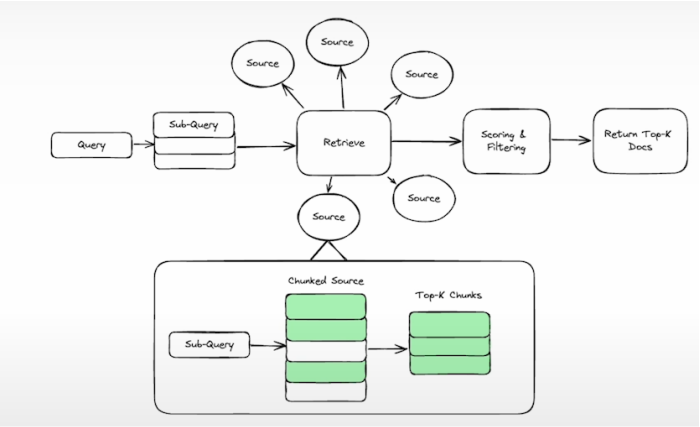

Figure 1: Example of a basic search tool implementation.

In zero-shot learning, a *LLM* receives a prompt and produces a response based on the weights of its model.

However, this has two *limitations*:

- *Time*: If the model is not trained on recent events, such as yesterday's game results, it cannot provide an answer.
- *Source Verification*: We often want to know the sources of the obtained information, for example, to reduce hallucinations.

Let's analyze the example in Fig. 1 step by step. If the agent wants to, it sends the query to the search tool; the first step is to understand the question and *break it into sub-questions* if necessary. This way, it can handle complex queries. Then, for each sub-query, the search tool looks for the best source, choosing among many. From the best source, only the most relevant information for the sub-query is extracted. A basic implementation of this can be achieved by *chunking* the sources and then using *vector search* to extract the top K chunks. After extracting data from the sources, the search tool would then *score the results* and filter out less relevant information.

All these steps are done automatically when, for example, we execute the following search:

```
from dotenv import load_dotenv
import os
from tavily import TavilyClient

# load environment variables from .env file
_ = load_dotenv()

# connect
client = TavilyClient(api_key=os.environ.get("TAVILY_API_KEY"))

# run search
result = client.search("What is in Nvidia's new Blackwell GPU?",
                       include_answer=True)

# print the answer
result["answer"]
```

Now let's look in more detail at the difference between regular search and agentic search.

4.1.1 Regular Search

Let's create a simple query about the weather in a specific location:

```
city = "San Francisco"

query = f"""
    what is the current weather in {city}?
    Should I travel there today?
"""
```

We will test a regular search using *DuckDuckGo* and we want to get the links that contain the answer:

```
import requests
from bs4 import BeautifulSoup
from duckduckgo_search import DDGS
import re

ddg = DDGS()
```

```

def search(query, max_results=6):
    try:
        results = ddg.text(query, max_results=max_results)
        return [i["href"] for i in results]
    except Exception as e:
        print(f"returning previous results due to exception reaching ddg.")
        results = [ # cover case where DDG rate limits due to high deeplearning.ai volume
            "https://weather.com/weather/today/l/USCA0987:1:US",
            "https://weather.com/weather/hourbyhour/l/54f9d8baac32496f6b5497b4bf7a277c3e2e6cc5625de69680e6169e7e38e9a8"
        ]
        return results

for i in search(query):
    print(i)
```

From the links obtained with this response, we want to extract the actual information to answer the question:

```
def scrape_weather_info(url):
    """Scrape content from the given URL"""
    if not url:
        return "Weather information could not be found."

    # fetch data
    headers = {'User-Agent': 'Mozilla/5.0'}
    response = requests.get(url, headers=headers)
    if response.status_code != 200:
        return "Failed to retrieve the webpage."

    # parse result
    soup = BeautifulSoup(response.text, 'html.parser')
    return soup

# use DuckDuckGo to find websites and take the first result
url = search(query)[0]

# scrape first wesbsite
soup = scrape_weather_info(url)

print(f"Website: {url}\n\n")
print(str(soup.body)[:50000]) # limit long outputs
```

BeautifulSoup extracts the HTML, then we print the output; this is very long and contains a lot of unnecessary information. We want to extract the headers and some content:

```
# extract text
weather_data = []
for tag in soup.find_all(['h1', 'h2', 'h3', 'p']):
    text = tag.get_text(" ", strip=True)
    weather_data.append(text)

# combine all elements into a single string
weather_data = "\n".join(weather_data)

# remove all spaces from the combined text
weather_data = re.sub(r'\s+', ' ', weather_data)

print(f"Website: {url}\n\n")
print(weather_data)
```

Now the output is much better; we get readable and understandable text that answers the question, but it is still much longer than necessary.

Now let's try with agentic search.

4.1.2 Agentic Search

Let's perform the same query and call *Tavily* to get an answer:

```
# run search
result = client.search(query, max_results=1)

```

```
# print first result
data = result["results"][0]["content"]
print(data)
```

The first response (from the search tool) is a simple JSON with a lot of information about the weather in San Francisco.

We can further analyze it and highlight the information more clearly:

```
import json
from pygments import highlight, lexers, formatters

# parse JSON
parsed_json = json.loads(data.replace("'", '"'))

# pretty print JSON with syntax highlighting
formatted_json = json.dumps(parsed_json, indent=4)
colorful_json = highlight(formatted_json,
                          lexers.JsonLexer(),
                          formatters.TerminalFormatter())

print(colorful_json)
```

We still get a JSON, but it is well-formatted. It might not be what a human wants to see, but this is the exact answer an agent would want: *structured data*.

5 Persistence and Streaming

When working with agents, we often need to handle longer running tasks. For these types of tasks, two important concepts are persistence and streaming.

Persistence allows you to maintain the state of an agent at a particular point in time. This means you can return to that state and resume from it in future interactions. This is crucial for long-running applications.

Streaming, on the other hand, enables you to emit a series of signals about what is happening at any given moment. This provides real-time updates on the actions the agent is taking.

5.1 Agent with Persistence and Streaming

First, we create the agent, as done in previous sections:

```python
from dotenv import load_dotenv

_ = load_dotenv()

from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, ToolMessage
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults

tool = TavilySearchResults(max_results=2)

class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]

    from langgraph.checkpoint.sqlite import SqliteSaver

memory = SqliteSaver.from_conn_string(":memory:")

class Agent:
    def __init__(self, model, tools, checkpointer, system=""):
        self.system = system
        graph = StateGraph(AgentState)
        graph.add_node("llm", self.call_openai)
        graph.add_node("action", self.take_action)
        graph.add_conditional_edges("llm", self.exists_action, {True: "action", False: END})
        graph.add_edge("action", "llm")
        graph.set_entry_point("llm")
        self.graph = graph.compile(checkpointer=checkpointer)
        self.tools = {t.name: t for t in tools}
        self.model = model.bind_tools(tools)

    def call_openai(self, state: AgentState):
        messages = state['messages']
        if self.system:
            messages = [SystemMessage(content=self.system)] + messages
        message = self.model.invoke(messages)
        return {'messages': [message]}

    def exists_action(self, state: AgentState):
        result = state['messages'][-1]
        return len(result.tool_calls) > 0

    def take_action(self, state: AgentState):
        tool_calls = state['messages'][-1].tool_calls
        results = []
        for t in tool_calls:
            print(f"Calling: {t}")
            result = self.tools[t['name']].invoke(t['args'])
            results.append(ToolMessage(tool_call_id=t['id'], name=t['name'], content=str(result)))
        print("Back to the model!")
        return {'messages': results}
```

Compared to previous implementations, we have added a new line of code. To add persistence, we rely on the concept

of a *checkpointer*, which is responsible for checkpointing the state after and between every node. To add persistence to this agent, we used a *SqliteSaver* which uses SQLite, a built-in database under the hood. Here, we use an in-memory database; this database is very simple, and if the notebook is refreshed, it disappears. However, it can easily be connected to an external database. Alternatively, other checkpointers can use *Redis*, *Postgres*, and other more persistent databases.

Once this checkpointer is initialized, it can be used by passing it to *graph.compile*. To facilitate this, we added a new parameter to the agent: the checkpointer.

Now we can create our agent:

```
prompt = """You are a smart research assistant. Use the search engine to look up information. \
You are allowed to make multiple calls (either together or in sequence). \
Only look up information when you are sure of what you want. \
If you need to look up some information before asking a follow up question, you are allowed to do that!
"""
model = ChatOpenAI(model="gpt-4o")
abot = Agent(model, [tool], system=prompt, checkpointer=memory)
```

To use our agent more effectively, we need to introduce the concept of streaming. There are two important aspects of streaming:

1. *Streaming individual messages*: This involves streaming both the AI message that determines which action to take and the observation message that represents the result of that action. This ensures real-time updates on the actions being taken by the AI and their outcomes.
2. *Streaming tokens*: For each token generated by the LLM call, we might want to stream the output. This provides real-time updates of the generated text, token by token.

For now, let's start by focusing only on streaming individual messages. This will allow us to observe the decision-making process of the AI in real-time, as it receives new observations and decides on subsequent actions.

Let's create the human message:

```
messages = [HumanMessage(content="What is the weather in sf?")]
```

Add the concept of *thread config* which will be used to track the different threads inside the persistence checkpointer. This allows us to have *multiple conversations* going on at the same time, necessary for production applications where there are typically many users, and also to *resume* a *conversation* we had *previously*.

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

This thread config is simply a dictionary with a key *configurable* containing the thread ID.

Now let's call the graph; instead of using *invoke*, we use *stream*:

```
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v['messages'])
```

We pass it the dictionary of messages and the thread. What we get is a stream of events, which represent updates to the state over time. Since our state has only one key, the messages key, we loop through and print it.

Executing this code, we get a stream of results. First, we get an *AIMessage*, which tells us to call *Tavily*. Next, we have the *ToolMessage*, the result of the *Tavily* call. Finally, we get another *AIMessage* with the final answer from the LLM.

With this stream method, we get all intermediate results and have a clear understanding of what is happening.

Now let's call the agent with another message:

```
messages = [HumanMessage(content="What about in la?")]
thread = {"configurable": {"thread_id": "1"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
```

Using the same thread ID as before, we expect to continue the previous conversation about the weather in SF. Indeed, the first message is an *AIMessage* indicating a need to look up the weather in LA. It knows we are still talking about the weather because we have introduced the concept of persistence with the checkpointer.

Again, if we use the same thread ID, the conversation continues, and we can ask questions without having to re-establish the context:

```
messages = [HumanMessage(content="Which one is warmer?")]
thread = {"configurable": {"thread_id": "1"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
```

However, if we change the thread ID, it is as if we change the conversation, and the model no longer knows what we are talking about:

```
messages = [HumanMessage(content="Which one is warmer?")]
thread = {"configurable": {"thread_id": "2"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
```

5.1.1 Streaming Tokens

To track tokens, we use the *astream_events* method that comes on all *LangChain* and *LangGraph* objects; this is an asynchronous method, meaning we need to use an asynchronous checkpointer.

To do this, we can import *AsyncSqliteSaver* and pass it to the agent (it is very similar to the one used before but asynchronous):

```
from langgraph.checkpoint.aiosqlite import AsyncSqliteSaver

memory = AsyncSqliteSaver.from_conn_string(":memory:")
abot = Agent(model, [tool], system=prompt, checkpointer=memory)
```

We use a new thread ID to start a new conversation:

```
messages = [HumanMessage(content="What is the weather in SF?")]
thread = {"configurable": {"thread_id": "4"}}
async for event in abot.graph.astream_events({"messages": messages}, thread, version="v1"):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        content = event["data"]["chunk"].content
        if content:
            # Empty content in the context of OpenAI means
            # that the model is asking for a tool to be invoked.
            # So we only print non-empty content
            print(content, end="|")
```

We iterate over a different type of event that represents updates from the underlying stream. We want to look for events corresponding to new tokens; this type of event is called *on_chat_model_stream*. When we see this type of event, we want to view it and print its content (separated with "|").

The output of this code first shows a call to the search tool; in this case, it didn't stream out anything because there is effectively no content to stream, it is just a function call. Next, we have the final response, and in this case, we see it token by token, separated by "|", exactly as we asked to print them.

6 Human in the Loop

There are many situations where it is beneficial to have a human in the loop to monitor and approve an agent's actions. This can be easily implemented with *LangGraph*.

Note: The code in this section is running in a later version of *LangGraph*. The later version has a couple of key additions:

- Additional state information is stored to memory and displayed when using *get_state()* or *get_state_history()*.
- State is additionally stored every state transition while previously it was stored at an interrupt or at the end. These change the command output slightly, but are a useful addtion to the information available.

6.1 Setting Up the Agent

Let's start with what we already know:

```
from dotenv import load_dotenv

_ = load_dotenv()

from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, ToolMessage
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
from langgraph.checkpoint.sqlite import SqliteSaver

memory = SqliteSaver.from_conn_string(":memory:")
```

When creating the *AgentState*, let's make a small modification. In previous examples, we annotated the messages list with the operator *add()*, which added new messages to the existing list. For human-in-the-loop interactions, however, *we might want to replace existing messages*. To achieve this, we create a function that checks for messages with the same ID and replaces the old one with the new one; otherwise, it adds the new messages as usual.

```
from uuid import uuid4
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, AIMessage

def reduce_messages(left: list[AnyMessage], right: list[AnyMessage]) -> list[AnyMessage]:
    # assign ids to messages that don't have them
    for message in right:
        if not message.id:
            message.id = str(uuid4())
    # merge the new messages with the existing messages
    merged = left.copy()
    for message in right:
        for i, existing in enumerate(merged):
            # replace any existing messages with the same id
            if existing.id == message.id:
                merged[i] = message
                break
        else:
            # append any new messages to the end
            merged.append(message)
    return merged

class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], reduce_messages]
```

Next, we use the same tool we have used before:

```
tool = TavilySearchResults(max_results=2)
```

6.2 Manual Human Approval

For the agent creation, we make a slight modification:

```
class Agent:
    def __init__(self, model, tools, system="", checkpointer=None):
```

```
        self.system = system
        graph = StateGraph(AgentState)
        graph.add_node("llm", self.call_openai)
        graph.add_node("action", self.take_action)
        graph.add_conditional_edges("llm", self.exists_action, {True: "action", False: END})
        graph.add_edge("action", "llm")
        graph.set_entry_point("llm")
        self.graph = graph.compile(
            checkpointer=checkpointer,
            interrupt_before=["action"]
        )
        self.tools = {t.name: t for t in tools}
        self.model = model.bind_tools(tools)

    def call_openai(self, state: AgentState):
        messages = state['messages']
        if self.system:
            messages = [SystemMessage(content=self.system)] + messages
        message = self.model.invoke(messages)
        return {'messages': [message]}

    def exists_action(self, state: AgentState):
        print(state)
        result = state['messages'][-1]
        return len(result.tool_calls) > 0

    def take_action(self, state: AgentState):
        tool_calls = state['messages'][-1].tool_calls
        results = []
        for t in tool_calls:
            print(f"Calling: {t}")
            result = self.tools[t['name']].invoke(t['args'])
            results.append(ToolMessage(tool_call_id=t['id'], name=t['name'], content=str(result)))
        print("Back to the model!")
        return {'messages': results}
```

When we compile the graph, besides passing the checkpointer, we also pass the *interrupt_before* parameter with the value *["action"]*. This causes the process to *interrupt before reaching the action node*. This is done because we will add a step requiring human approval before executing the tools, ensuring they are executed correctly.

This interruption happens before executing any tool; it is also possible to specify interruptions only before certain tools and not others.

Initialize the agent and call it, also passing the thread configuration:

```
prompt = """You are a smart research assistant. Use the search engine to look up information. \
You are allowed to make multiple calls (either together or in sequence). \
Only look up information when you are sure of what you want. \
If you need to look up some information before asking a follow up question, you are allowed to do that!
"""
model = ChatOpenAI(model="gpt-3.5-turbo")
abot = Agent(model, [tool], system=prompt, checkpointer=memory)

messages = [HumanMessage(content="Whats the weather in SF?")]
thread = {"configurable": {"thread_id": "1"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
```

We get an interruption after the *AIMessage* because the next step would be to call the tool.

We can view the current state of the graph for this thread:

```
abot.graph.get_state(thread)
```

We can also see what the next step would be, i.e., the node to be called next:

```
abot.graph.get_state(thread).next
```

6.2.1 Continue After Interrupt

To continue, simply call the stream again with the same thread configuration and pass *None* as the input:

```
for event in abot.graph.stream(None, thread):
    for v in event.values():
        print(v)
```

This way, we get the *ToolMessage* and the final response.

If we now execute:

```
abot.graph.get_state(thread).next
```

We get an empty response, meaning there are no more steps to take.

We can combine everything in a loop, asking the user whether to proceed:

```
messages = [HumanMessage("Whats the weather in LA?")]
thread = {"configurable": {"thread_id": "2"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
while abot.graph.get_state(thread).next:
    print("\n", abot.graph.get_state(thread),"\n")
    _input = input("proceed?")
    if _input != "y":
        print("aborting")
        break
    for event in abot.graph.stream(None, thread):
        for v in event.values():
            print(v)
```

6.3 Snapshot and State Management

As a graph is executing, a snapshot of each state is stored in memory. This snapshot includes the agent state and other useful information, such as the thread and a unique identifier for each snapshot. This allows access to specific snapshots.

There are commands for accessing memory:

- *.get_state(thread)* returns the current state if only the thread ID is provided.
- *.get_state_history(thread)* returns an iterator over all the state snapshots. This iterator can be used to access all the unique identifiers for each state.

An example usage is, given the thread ID and the unique identifier *thread_ts*, one can access a specific state and use it with the command *.invoke(None, thread, thread_ts)*; this will use that state as the starting point for the rest of the graph. This is akin to time travel. If *thread_ts* is not provided, the current state will be used as the starting point.

The *thread_ts* can be used to access a specific state, modify it, and then use *.update_state(thread, state.values)* to save it to memory. Then, using stream or invoke, the modified state will be used as the starting point.

Let's see how to implement these solutions.

6.3.1 Modify State

Let's see an example of modifying a state.

```
messages = [HumanMessage("Whats the weather in LA?")]
thread = {"configurable": {"thread_id": "3"}}
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v)
```

If we check the state, we will see two messages: *HumanMessage* and *AIMessage*.

```
abot.graph.get_state(thread)
```

Suppose we want to modify the last message to check the weather in Louisiana instead of LA. First, we save the current state of the graph.

```
current_values = abot.graph.get_state(thread)
```

The last message will be the *AIMessage*.

```
current_values.values['messages'][-1]
```

We can also see the list of tool calls associated with this message.

```
current_values.values['messages'][-1].tool_calls
```

Let's update this tool call.

```
_id = current_values.values['messages'][-1].tool_calls[0]['id']
current_values.values['messages'][-1].tool_calls = [
    {'name': 'tavily_search_results_json',
  'args': {'query': 'current weather in Louisiana'},
  'id': _id}
]
```

First, we get the ID associated with this tool call. Then we update the tool calls property to be a list, which will have only one element because there is only one call. The keys are the same as before, but the arguments are different.

This does not take effect until we call *update_state*.

```
abot.graph.update_state(thread, current_values.values)
```

It takes the thread to know which conversation we are referring to and the new values we want to overwrite.

If we then continue, we will see that *Tavily* is called with the weather query for Louisiana, returns the information, and the model responds correctly.

```
for event in abot.graph.stream(None, thread):
    for v in event.values():
        print(v)
```

6.3.2 Time Travel

One important thing to note is that we're actually keeping a *running list of all these states*. So when we modify a state, we are actually creating a new state, which becomes the current state.

This allows us to go back and visit previous states. To do this, we call *get_state_history* and save all the states in a variable.

```
states = []
for state in abot.graph.get_state_history(thread):
    print(state)
    print('--')
    states.append(state)
```

Suppose we want to go back to the weather search for LA; we need to find the correct state and restart from that point with the stream.

```
to_replay = states[-3]

for event in abot.graph.stream(None, to_replay.config):
    for k, v in event.items():
        print(v)
```

6.3.3 Go Back in Time and Edit

Another thing we can do is go back in time and then edit from there.

We can do the same as before, where we modify the state, but this time we modify a previous state instead of the last one.

```
_id = to_replay.values['messages'][-1].tool_calls[0]['id']
to_replay.values['messages'][-1].tool_calls = [{'name': 'tavily_search_results_json',
  'args': {'query': 'current weather in LA, accuweather'},
  'id': _id}]
```

We modified it to search for the weather information on *accuweather* instead of other sites.

We can update the state of this *to_replay* and get back a new branch state.

```
branch_state = abot.graph.update_state(to_replay.config, to_replay.values)
```

Then, we call the stream from this *branch_state*.

```
for event in abot.graph.stream(None, branch_state):
    for k, v in event.items():
        if k != "__end__":
            print(v)
```

6.3.4 Add Message to a State

Another thing we can do is add messages to a state at a given time.

Suppose that, starting from *to_replay*, instead of calling *Tavily*, we wanted to mock a response. We can do this by adding a new message to the state to mock out a response.

First, we get the ID of the tool call.

```
_id = to_replay.values['messages'][-1].tool_calls[0]['id']
```

Then, we create the *state_update*, which is a list of messages. The structure is that of a *ToolMessage* to which we add the content, as if it were the actual response obtained from the tool.

```
state_update = {"messages": [ToolMessage(
    tool_call_id=_id,
    name="tavily_search_results_json",
    content="54 degree celcius",
)]}
```

This is a new message, so when we add it, it will not replace any existing messages but will be added.

Now, we update the graph's state.

```
branch_and_add = abot.graph.update_state(
    to_replay.config,
    state_update,
    as_node="action")
```

Because we're adding and pretending that an action has taken place, as opposed to modifying the existing state, we need to add this *as_node="action"* parameter.

What we're doing is indicating that the *state_update* is not just a modification but that we are performing this update as if we were the action node.

If we now call the stream in this new configuration, it does not perform any action (because the role of the action is taken by the message we added); instead, it calls the model and responds to the question with an *AIMessage*.

```
for event in abot.graph.stream(None, branch_and_add):
    for k, v in event.items():
        print(v)
```

7 Essay Writer

In this section, we will create a compact version of an AI researcher, specifically an essay writer.

7.1 Overview

Here is the general workflow of this agent:

- First, we create a *plan* for the essay; this will happen once upfront.
- Then, based on that plan, we will conduct some *research*; this involves calling *Tavily* and obtaining relevant documents.
- From there, we move into *generation*; following the plan and using the researched documents, we will write the essay.
- After generating the draft, we will either finish or continue.
- If we continue, we will enter a *reflection* phase where we generate a critique of the current essay.
- Based on the critique, we will conduct *additional research* with *Tavily* to gather more information.
- We will append the new information to the existing set of documents and return to the *generation* phase, continuing this cycle until certain criteria are met and we conclude.

7.2 Implementation

First, we import the environment variables and standard functions, and set up our in-memory SQLite checkpointers:

```
from dotenv import load_dotenv

_ = load_dotenv()

from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated, List
import operator
from langgraph.checkpoint.sqlite import SqliteSaver
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, AIMessage, ChatMessage

memory = SqliteSaver.from_conn_string(":memory:")
```

Next, we define the agent state:

```
class AgentState(TypedDict):
    task: str
    plan: str
    draft: str
    critique: str
    content: List[str]
    revision_number: int
    max_revisions: int
```

We want to keep track of several things:

- *task*: The user-provided topic for the essay.
- *plan*: The outline of the essay generated by the planning agent.
- *draft*: The current draft of the essay.
- *critique*: Feedback and recommendations from the critique agent.
- *content*: A list of documents obtained through *Tavily* research.
- *revision_number*: The number of revisions made so far.
- *max_revisions*: The maximum number of revisions allowed.

We initialize the model:

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

Now, we create the prompts for each stage of the agent. First, the planning prompt:

```
PLAN_PROMPT = """You are an expert writer tasked with writing a high level outline of an essay. \
Write such an outline for the user provided topic. Give an outline of the essay along with any relevant notes \
or instructions for the sections."""
```

Next, the writer prompt, which will write the essay given the content researched:

```
WRITER_PROMPT = """You are an essay assistant tasked with writing excellent 5-paragraph essays.\
Generate the best essay possible for the user's request and the initial outline. \
If the user provides critique, respond with a revised version of your previous attempts. \
Utilize all the information below as needed:

------

{content}"""
```

We also need a reflection prompt for critiquing the draft:

```
REFLECTION_PROMPT = """You are a teacher grading an essay submission. \
Generate critique and recommendations for the user's submission. \
Provide detailed recommendations, including requests for length, depth, style, etc."""
```

Next, the research plan prompt for generating queries to pass to *Tavily* after the planning step:

```
RESEARCH_PLAN_PROMPT = """You are a researcher charged with providing information that can \
be used when writing the following essay. Generate a list of search queries that will gather \
any relevant information. Only generate 3 queries max."""
```

Lastly, the research critique prompt for generating queries after receiving critique:

```
RESEARCH_CRITIQUE_PROMPT = """You are a researcher charged with providing information that can \
be used when making any requested revisions (as outlined below). \
Generate a list of search queries that will gather any relevant information. Only generate 3 queries max."""
```

To ensure we get a list of strings from the LLM, we use function calling with a *pydantic* model, that just represents the result that we want to get back:

```
from langchain_core.pydantic_v1 import BaseModel

class Queries(BaseModel):
    queries: List[str]
```

We import the *Tavily client* instead of *Tavily tool*, because we're working with it in a slightly unconventional way:

```
from tavily import TavilyClient
import os
tavily = TavilyClient(api_key=os.environ["TAVILY_API_KEY"])
```

Now we can create the agents for each node, starting with the planning agent:

```
def plan_node(state: AgentState):
    messages = [
        SystemMessage(content=PLAN_PROMPT),
        HumanMessage(content=state['task'])
    ]
    response = model.invoke(messages)
    return {"plan": response.content}
```

This node takes the state, creates a list of messages, one of which is the planning prompt (as the system prompt), and the other is a *HumanMessage* representing the task. We pass these messages to the model, which returns a message. We take the content of that message and set it as the plan. This updates the plan key to be this response.

The next node is the agent that takes the plan and performs some research:

```
def research_plan_node(state: AgentState):
    queries = model.with_structured_output(Queries).invoke([
        SystemMessage(content=RESEARCH_PLAN_PROMPT),
        HumanMessage(content=state['task'])
    ])
    content = state['content'] or []
    for q in queries.queries:
        response = tavily.search(query=q, max_results=2)
        for r in response['results']:
            content.append(r['content'])
    return {"content": content}
```

This node takes the agent state and first generates some queries. By using *model.with_structured_output(Queries)*, we ensure the response follows the pydantic object that we defined, which is a list of queries. We then obtain a list of documents (and if it's empty, we create an empty list). We loop over the generated queries and search them in *Tavily*. We append the results to the content and update the content key.

Next, we create the node that writes the draft:

```python
def generation_node(state: AgentState):
    content = "\n\n".join(state['content'] or [])
    user_message = HumanMessage(
        content=f"{state['task']}\n\nHere is my plan:\n\n{state['plan']}")
    messages = [
        SystemMessage(
            content=WRITER_PROMPT.format(content=content)
        ),
        user_message
        ]
    response = model.invoke(messages)
    return {
        "draft": response.content,
        "revision_number": state.get("revision_number", 1) + 1
    }
```

First, we prepare the content by joining the list of strings into one. Then, we create the user message by combining the task and the plan. We create a list of messages that includes the *SystemMessage* and the combined task and plan. We pass these messages to the model and obtain a response. We then update two things in the state: the draft (which will be the model's response) and the revision number (which will be incremented by 1).

Next, we need the reflection node:

```python
def reflection_node(state: AgentState):
    messages = [
        SystemMessage(content=REFLECTION_PROMPT),
        HumanMessage(content=state['draft'])
    ]
    response = model.invoke(messages)
    return {"critique": response.content}
```

This node sets the system prompt to the reflection prompt and the newly created draft. We pass these messages to the model and take the response. Finally, we update the critique with the value of the response.

Lastly, we create the node for research after the critique:

```python
def research_critique_node(state: AgentState):
    queries = model.with_structured_output(Queries).invoke([
        SystemMessage(content=RESEARCH_CRITIQUE_PROMPT),
        HumanMessage(content=state['critique'])
    ])
    content = state['content'] or []
    for q in queries.queries:
        response = tavily.search(query=q, max_results=2)
        for r in response['results']:
            content.append(r['content'])
    return {"content": content}
```

This node is very similar to the previous research node, but it takes the critique as input.

The last thing we need is the *should_continue* condition, which checks the revision number. If it is greater than the maximum number, we end; otherwise, we continue to reflect:

```python
def should_continue(state):
    if state["revision_number"] > state["max_revisions"]:
        return END
    return "reflect"
```

Now that we have all the nodes and conditional edges, we need to assemble them into a graph. First, we initialize the graph with the agent state:

```python
builder = StateGraph(AgentState)
```

We then add all the created nodes and set the entry point:

```python
builder.add_node("planner", plan_node)
builder.add_node("generate", generation_node)
builder.add_node("reflect", reflection_node)
builder.add_node("research_plan", research_plan_node)
builder.add_node("research_critique", research_critique_node)

builder.set_entry_point("planner")
```

We add the conditional edge:

```
builder.add_conditional_edges(
    "generate",
    should_continue,
    {END: END, "reflect": "reflect"}
)
```

And finally, we add all the standard edges:

```
builder.add_edge("planner", "research_plan")
builder.add_edge("research_plan", "generate")

builder.add_edge("reflect", "research_critique")
builder.add_edge("research_critique", "generate")
```

Now we can compile the graph and pass in the checkpointer we created earlier:

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

For better understanding and to verify everything is set up correctly, we can visualize the created graph:

```
from IPython.display import Image

Image(graph.get_graph().draw_png())
```

Finally, we can test our writing agent:

```
thread = {"configurable": {"thread_id": "1"}}
for s in graph.stream({
    'task': "what is the difference between langchain and langsmith",
    "max_revisions": 2,
    "revision_number": 1,
}, thread):
    print(s)
```

We call *graph.stream* to observe everything that happens.

8 LangChain Resources

In this course, we have covered only the basics of implementing an agent. This section aims to guide you on where to find further and more in-depth information.

The first useful resource is definitely the *LangChain* documentation. It provides a great high-level overview of all the packages and services in the *LangChain* ecosystem. There are also many tutorials useful for building various applications with agents. Additionally, it offers an introduction to LangServe, which is an easy way to turn your *LangChain* application into a web server. Helping out with all of this is LangSmith: it can assist with debugging, prompt management, and also with monitoring in production, and it also has a playground.

Another useful resource is the *LangChain GitHub* repository, which has many good resources and templates to get started. The *LangGraph GitHub* repository has in-depth documentation: reference docs, tutorials, and how-to guides.

The prompt hub is a good place for inspiration and to see what other expert prompters are doing.

9 Conclusion

In this course, we have explored how to build agents, including some rather complex ones. In this final section, we will look at some additional agent flows that we haven't covered but are great to know about.

9.1 Multi-Agent Architecture

We touched on this concept when creating the essay writer agent. This architecture involves multiple agents working on the *same shared state*. These agents could simply be different prompts and LLMs, or they could each have different tools that they can call. The crucial aspect is that they all work on the same shared state, passing it from one agent to the next. This allows for a highly collaborative and integrated approach to task management.

9.2 Supervisor Architecture

In this architecture, there is a supervisor agent that oversees and coordinates several *sub-agents*. The supervisor determines the input for each sub-agent, which can have their own *distinct states*. These sub-agents might operate as independent graphs rather than sharing a single state. The key difference from the multi-agent framework is the emphasis on the supervisor's role in routing and coordinating the sub-agents. This setup is useful for complex workflows where different tasks require specialized handling.

9.3 Plan and Execute Architecture

This architecture begins with an explicit planning step. The agent formulates a plan outlining the steps needed to complete a task. It then starts *executing the plan step-by-step*. The agent may perform a step, return to update the plan if necessary, and then proceed to the next step. This cycle continues until the plan is fully executed. Afterward, the agent checks if the task was successfully completed. If not, it replans and repeats the process until the objective is met. This method ensures a structured approach to task execution and allows for adjustments as new information becomes available.

9.4 Language Agent Tree Search Architecture

This architecture involves a tree search over possible actions. The agent generates an action and then reflects on its consequences. Based on this reflection, it generates subsequent sub-actions and reflects further. Through these reflections, the agent can determine the *optimal path by backpropagating and updating parent nodes* with new information. This iterative process helps inform future decisions from previous states, allowing for a dynamic and flexible approach to problem-solving.