From Basics to Advanced: Exploring LangGraph

Building single- and multi-agent workflows with human-in-the-loop interactions

Mariya Mansurova

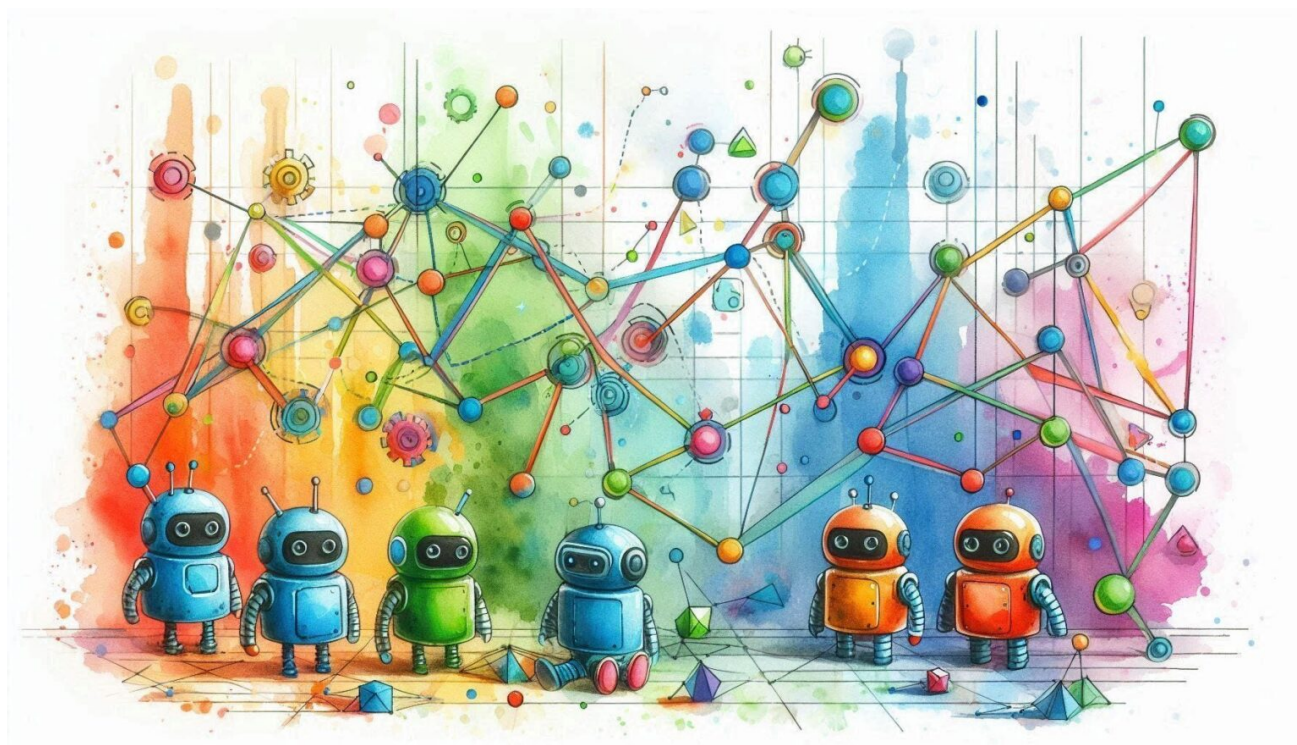Aug 15, 2024 • 25 min read

Image by DALL-E 3

[LangChain](#) is one of the leading frameworks for building applications powered by Lardge Language Models. With the [LangChain Expression Language](#) (LCEL), defining and executing step-by-step action sequences – also known as chains – becomes much simpler. In more technical terms, LangChain allows us to create DAGs (directed acyclic graphs).

As [Llm](#) applications, particularly LLM agents, have evolved, we've

begun to use LLMs not just for execution but also as reasoning engines. This shift has introduced interactions that frequently involve repetition (cycles) and complex conditions. In such scenarios, LCEL is not sufficient, so LangChain implemented a new module – LangGraph.

LangGraph (as you might guess from the name) models all interactions as cyclical graphs. These graphs enable the development of advanced workflows and interactions with multiple loops and if-statements, making it a handy tool for creating both agent and multi-agent workflows.

In this article, I will explore LangGraph's key features and capabilities, including multi-agent applications. We'll build a system that can answer different types of questions and dive into how to implement a human-in-the-loop setup.

In the previous article, we tried using CrewAI, another popular framework for multi-agent systems. LangGraph, however, takes a different approach. While CrewAI is a high-level framework with many predefined features and ready-to-use components, LangGraph operates at a lower level, offering extensive customization and control.

With that introduction, let's dive into the fundamental concepts of LangGraph.

LangGraph basics

LangGraph is part of the LangChain ecosystem, so we will continue using well-known concepts like prompt templates, tools, etc. However, LangGraph brings a bunch of additional concepts. Let's discuss them.

LangGraph is created to define cyclical graphs. Graphs consist of the following elements:

- Nodes represent actual actions and can be either LLMs, agents or functions. Also, a special END node marks the end of execution.

- Edges connect nodes and determine the execution flow of your graph. There are basic edges that simply link one node to another and conditional edges that incorporate if-statements and additional logic.

Another important concept is the state of the graph. The state serves as a foundational element for collaboration among the graph's components. It represents a snapshot of the graph that any part – whether nodes or edges – can access and modify during execution to retrieve or update information.

Additionally, the state plays a crucial role in persistence. It is automatically saved after each step, allowing you to pause and resume execution at any point. This feature supports the development of more complex applications, such as those requiring error correction or incorporating human-in-the-loop interactions.

Single-agent workflow

Building agent from scratch

Let's start simple and try using LangGraph for a basic use case – an agent with tools.

I will try to build similar applications to those we did with CrewAI

in <u>the previous article</u>. Then, we will be able to compare the two frameworks. For this example, let's create an application that can automatically generate documentation based on the table in the database. It can save us quite a lot of time when creating documentation for our data sources.

As usual, we will start by defining the tools for our agent. Since I will use the ClickHouse database in this example, I've defined a function to execute any query. You can use a different database if you prefer, as we won't rely on any database-specific features.

```python
CH_HOST = 'http://localhost:8123' # default address
import requests

def get_clickhouse_data(query, host = CH_HOST, connection_timeout = 1500
    r = requests.post(host, params = {'query': query},
      timeout = connection_timeout)
    if r.status_code == 200:
        return r.text
    else:
        return 'Database returned the following error:n' + r.text
```

It's crucial to make LLM tools reliable and error-prone. If a database returns an error, I provide this feedback to the LLM rather than throwing an exception and halting execution. Then, the LLM agent will have an opportunity to fix an error and call the function again.

Let's define one tool named `execute_sql`, which enables the execution of any SQL query. We use `pydantic` to specify the tool's structure, ensuring that the LLM agent has all the needed information to use the tool effectively.

```python
from langchain_core.tools import tool
from pydantic.v1 import BaseModel, Field
from typing import Optional
```

```python
class SQLQuery(BaseModel):
    query: str = Field(description="SQL query to execute")

@tool(args_schema = SQLQuery)
def execute_sql(query: str) -> str:
    """Returns the result of SQL query execution"""
    return get_clickhouse_data(query)
```

We can print the parameters of the created tool to see what information is passed to LLM.

```python
print(f'''
name: {execute_sql.name}
description: {execute_sql.description}
arguments: {execute_sql.args}
''')

# name: execute_sql
# description: Returns the result of SQL query execution
# arguments: {'query': {'title': 'Query', 'description':
#   'SQL query to execute', 'type': 'string'}}
```

Everything looks good. We've set up the necessary tool and can now move on to defining an LLM agent. As we discussed above, the cornerstone of the agent in LangGraph is its state, which enables the sharing of information between different parts of our graph.

Our current example is relatively straightforward. So, we will only need to store the history of messages. Let's define the agent state.

```python
# useful imports
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import AnyMessage, SystemMessage, HumanMess
```

```python
# defining agent state
class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]
```

We've defined a single parameter in `AgentState` — `messages` — which is a list of objects of the class `AnyMessage` . Additionally, we annotated it with `operator.add` (reducer). This annotation ensures that each time a node returns a message, it is appended to the existing list in the state. Without this operator, each new message would replace the previous value rather than being added to the list.

The next step is to define the agent itself. Let's start with `__init__` function. We will specify three arguments for the agent: model, list of tools and system prompt.

```python
class SQLAgent:
  # initialising the object
  def __init__(self, model, tools, system_prompt = ""):
    self.system_prompt = system_prompt

    # initialising graph with a state
    graph = StateGraph(AgentState)

    # adding nodes
    graph.add_node("llm", self.call_llm)
    graph.add_node("function", self.execute_function)
    graph.add_conditional_edges(
        "llm",
        self.exists_function_calling,
        {True: "function", False: END}
    )
    graph.add_edge("function", "llm")

    # setting starting point
    graph.set_entry_point("llm")
```

```
    self.graph = graph.compile()
    self.tools = {t.name: t for t in tools}
    self.model = model.bind_tools(tools)
```

In the initialisation function, we've outlined the structure of our graph, which includes two nodes: `llm` and `action`. Nodes are actual actions, so we have functions associated with them. We will define functions a bit later.

Additionally, we have one conditional edge that determines whether we need to execute the function or generate the final answer. For this edge, we need to specify the previous node (in our case, `llm`), a function that decides the next step, and mapping of the subsequent steps based on the function's output (formatted as a dictionary). If `exists_function_calling` returns True, we follow to the function node. Otherwise, execution will conclude at the special `END` node, which marks the end of the process.

We've added an edge between `function` and `llm`. It just links these two steps and will be executed without any conditions.

With the main structure defined, it's time to create all the functions outlined above. The first one is `call_llm`. This function will execute LLM and return the result.

The agent state will be passed to the function automatically so we can use the saved system prompt and model from it.

```python
class SQLAgent:
  <...>

  def call_llm(self, state: AgentState):
    messages = state['messages']
    # adding system prompt if it's defined
```

```python
    if self.system_prompt:
        messages = [SystemMessage(content=self.system_prompt)] + message

    # calling LLM
    message = self.model.invoke(messages)

    return {'messages': [message]}
```

As a result, our function returns a dictionary that will be used to update the agent state. Since we used `operator.add` as a reducer for our state, the returned message will be appended to the list of messages stored in the state.

The next function we need is `execute_function` which will run our tools. If the LLM agent decides to call a tool, we will see it in the `message.tool_calls` parameter.

```python
class SQLAgent:
  <...>

  def execute_function(self, state: AgentState):
    tool_calls = state['messages'][-1].tool_calls

    results = []
    for tool in tool_calls:
      # checking whether tool name is correct
      if not t['name'] in self.tools:
      # returning error to the agent
      result = "Error: There's no such tool, please, try again"
      else:
      # getting result from the tool
      result = self.tools[t['name']].invoke(t['args'])

      results.append(
        ToolMessage(
          tool_call_id=t['id'],
          name=t['name'],
          content=str(result)
        )
```

```
        )
        return {'messages': results}
```

In this function, we iterate over the tool calls returned by LLM and either invoke these tools or return the error message. In the end, our function returns the dictionary with a single key `messages` that will be used to update the graph state.

There's only one function left -the function for the conditional edge that defines whether we need to execute the tool or provide the final result. It's pretty straightforward. We just need to check whether the last message contains any tool calls.

```python
class SQLAgent:
  <...>

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

It's time to create an agent and LLM model for it. I will use the new OpenAI GPT 4o mini model (doc) since it's cheaper and better performing than GPT 3.5.

```python
import os

# setting up credentioals
os.environ["OPENAI_MODEL_NAME"]='gpt-4o-mini'
os.environ["OPENAI_API_KEY"] = '<your_api_key>'

# system prompt
prompt = '''You are a senior expert in SQL and data analysis.
So, you can help the team to gather needed data to power their decisions
You are very accurate and take into account all the nuances in data.
Your goal is to provide the detailed documentation for the table in data
that will help users.'''
```

```python
model = ChatOpenAI(model="gpt-4o-mini")
doc_agent = SQLAgent(model, [execute_sql], system=prompt)
```

LangGraph provides us with quite a handy feature to visualise graphs. To use it, you need to install `pygraphviz` .

It's a bit tricky for Mac with M1/M2 chips, so here is the lifehack for you (source):

```
! brew install graphviz
! python3 -m pip install -U --no-cache-dir
    --config-settings="--global-option=build_ext"
    --config-settings="--global-option=-I$(brew --prefix graphviz)/inclu
    --config-settings="--global-option=-L$(brew --prefix graphviz)/lib/"
    pygraphviz
```

After figuring out the installation, here's our graph.

```python
from IPython.display import Image
Image(doc_agent.graph.get_graph().draw_png())
```
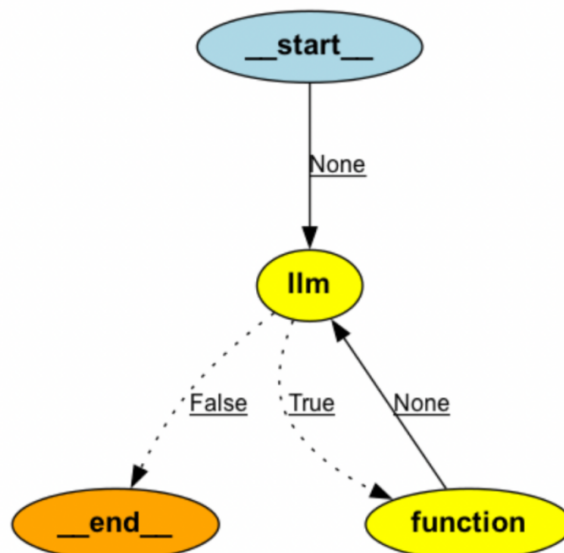

As you can see, our graph has cycles. Implementing something like this with LCEL would be quite challenging.

Finally, it's time to execute our agent. We need to pass the initial set of messages with our questions as `HumanMessage`.

```
messages = [HumanMessage(content="What info do we have in ecommerce_db.u
result = doc_agent.graph.invoke({"messages": messages})
```

In the `result` variable, we can observe all the messages generated during execution. The process worked as expected:

- The agent decided to call the function with the query
  `describe ecommerce.db_users`.

- LLM then processed the information from the tool and
  provided a user-friendly answer.

```
result['messages']


# [
#   HumanMessage(content='What info do we have in ecommerce_db.users tab
#   AIMessage(content='', tool_calls=[{'name': 'execute_sql', 'args': {'
#   ToolMessage(content='user_idtUInt64tttttncountrytStringttttttnis_acti
#   AIMessage(content='The `ecommerce_db.users` table contains the follo
# ]
```

Here's the final result. It looks pretty decent.

```
print(result['messages'][-1].content)

# The `ecommerce_db.users` table contains the following columns:
# 1. **user_id**: `UInt64` - A unique identifier for each user.
# 2. **country**: `String` - The country where the user is located.
# 3. **is_active**: `UInt8` - Indicates whether the user is active (1) o
# 4. **age**: `UInt64` - The age of the user.
```

Using prebuilt agents

We've learned how to build an agent from scratch. However, we can leverage LangGraph's built-in functionality for simpler tasks like this one.

We can use a prebuilt ReAct agent to get a similar result: an agent that can work with tools.

```python
from langgraph.prebuilt import create_react_agent
prebuilt_doc_agent = create_react_agent(model, [execute_sql],
    state_modifier = system_prompt)
```

It is the same agent as we built previously. We will try it out in a second, but first, we need to understand two other important concepts: persistence and streaming.

Persistence and streaming

Persistence refers to the ability to maintain context across different interactions. It's essential for agentic use cases when an application can get additional input from the user.

LangGraph automatically saves the state after each step, allowing you to pause or resume execution. This capability supports the implementation of advanced business logic, such as error recovery or human-in-the-loop interactions.

The easiest way to add persistence is to use an in-memory SQLite database.

```python
from langgraph.checkpoint.sqlite import SqliteSaver
memory = SqliteSaver.from_conn_string(":memory:")
```

For the off-the-shelf agent, we can pass memory as an argument

while creating an agent.

```
prebuilt_doc_agent = create_react_agent(model, [execute_sql],
    checkpointer=memory)
```

If you're working with a custom agent, you need to pass memory as a check pointer while compiling a graph.

```
class SQLAgent:
  def __init__(self, model, tools, system_prompt = ""):
    <...>
    self.graph = graph.compile(checkpointer=memory)
    <...>
```

Let's execute the agent and explore another feature of LangGraph: streaming. With streaming, we can receive results from each step of execution as a separate event in a stream. This feature is crucial for production applications when multiple conversations (or threads) need to be processed simultaneously.

LangGraph supports not only event streaming but also token-level streaming. The only use case I have in mind for token streaming is to display the answers in real-time word by word (similar to ChatGPT implementation).

Let's try using streaming with our new prebuilt agent. I will also use the `pretty_print` function for messages to make the result more readable.

```
# defining thread
thread = {"configurable": {"thread_id": "1"}}
messages = [HumanMessage(content="What info do we have in ecommerce_db.u

for event in prebuilt_doc_agent.stream({"messages": messages}, thread):
    for v in event.values():
```

```
        v['messages'][-1].pretty_print()

# ================================ Ai Message ================================
# Tool Calls:
#  execute_sql (call_YieWiChbFuOlxBg8G1jDJitR)
#  Call ID: call_YieWiChbFuOlxBg8G1jDJitR
#   Args:
#     query: SELECT * FROM ecommerce_db.users LIMIT 1;
# ================================ Tool Message ================================
# Name: execute_sql
# 1000001 United Kingdom 0 70
#
# ================================ Ai Message ================================
#
# The `ecommerce_db.users` table contains at least the following informa
#
# - **User ID** (e.g., `1000001`)
# - **Country** (e.g., `United Kingdom`)
# - **Some numerical value** (e.g., `0`)
# - **Another numerical value** (e.g., `70`)
#
# The specific meaning of the numerical values and additional columns
# is not clear from the single row retrieved. Would you like more detail
# or a broader query?
```

Interestingly, the agent wasn't able to provide a good enough result. Since the agent didn't look up the table schema, it struggled to guess all columns' meanings. We can improve the result by using follow-up questions in the same thread.

```
followup_messages = [HumanMessage(content="I would like to know the colu

for event in prebuilt_doc_agent.stream({"messages": followup_messages},
    for v in event.values():
        v['messages'][-1].pretty_print()

# ================================ Ai Message ================================
# Tool Calls:
#   execute_sql (call_sQKRWtG6aEB38rtOpZszxTVs)
```

```
#  Call ID: call_sQKRWtG6aEB38rtOpZszxTVs
#    Args:
#      query: DESCRIBE ecommerce_db.users;
# ============================== Tool Message ========================
# Name: execute_sql
#
# user_id UInt64
# country String
# is_active UInt8
# age UInt64
#
# ============================== Ai Message ==========================
#
# The `ecommerce_db.users` table has the following columns along with th
#
# | Column Name | Data Type |
# |-------------|-----------|
# | user_id     | UInt64    |
# | country     | String    |
# | is_active   | UInt8     |
# | age         | UInt64    |
#
# If you need further information or assistance, feel free to ask!
```

This time, we got the full answer from the agent. Since we provided the same thread, the agent was able to get the context from the previous discussion. That's how persistence works.

Let's try to change the thread and ask the same follow-up question.

```
new_thread = {"configurable": {"thread_id": "42"}}
followup_messages = [HumanMessage(content="I would like to know the colu

for event in prebuilt_doc_agent.stream({"messages": followup_messages},
    for v in event.values():
        v['messages'][-1].pretty_print()

# ============================== Ai Message ==========================
# Tool Calls:
```

```
#   execute_sql (call_LrmsOGzzusaLEZLP9hGTBGgo)
#  Call ID: call_LrmsOGzzusaLEZLP9hGTBGgo
#   Args:
#     query: DESCRIBE your_table_name;
# ============================== Tool Message ========================
# Name: execute_sql
#
# Database returned the following error:
# Code: 60. DB::Exception: Table default.your_table_name does not exist.
#
# ============================== Ai Message ==========================
#
# It seems that the table `your_table_name` does not exist in the databa
# Could you please provide the actual name of the table you want to desc
```

It was not surprising that the agent lacked the context needed to answer our question. Threads are designed to isolate different conversations, ensuring that each thread maintains its own context.

In real-life applications, managing memory is essential. Conversations might become pretty lengthy, and at some point, it won't be practical to pass the whole history to LLM every time. Therefore, it's worth trimming or filtering messages. We won't go deep into the specifics here, but you can find guidance on it in the LangGraph documentation. Another option to compress the conversational history is using summarization (example).

We've learned how to build systems with single agents using LangGraph. The next step is to combine multiple agents in one application.

Multi-Agent Systems

As an example of a multi-agent workflow, I would like to build an

application that can handle questions from various domains. We will have a set of expert agents, each specializing in different types of questions, and a router agent that will find the best-suited expert to address each query. Such an application has numerous potential use cases: from automating customer support to answering questions from colleagues in internal chats.

First, we need to create the agent state – the information that will help agents to solve the question together. I will use the following fields:

- `question` – initial customer request;
- `question_type` – the category that defines which agent will be working on the request;
- `answer` – the proposed answer to the question;
- `feedback` – a field for future use that will gather some feedback.

```python
class MultiAgentState(TypedDict):
    question: str
    question_type: str
    answer: str
    feedback: str
```

I don't use any reducers, so our state will store only the latest version of each field.

Then, let's create a router node. It will be a simple LLM model that defines the category of question (database, LangChain or general questions).

```python
question_category_prompt = '''You are a senior specialist of analytical
Depending on your answer, question will be routed to the right team, so
There are 3 possible question types:
```

```
   - DATABASE - questions related to our database (tables or fields)
   - LANGCHAIN- questions related to LangGraph or LangChain libraries
   - GENERAL - general questions
   Return in the output only one word (DATABASE, LANGCHAIN or  GENERAL).
   '''


def router_node(state: MultiAgentState):
  messages = [
    SystemMessage(content=question_category_prompt),
    HumanMessage(content=state['question'])
  ]
  model = ChatOpenAI(model="gpt-4o-mini")
  response = model.invoke(messages)
  return {"question_type": response.content}
```

Now that we have our first node – the router – let's build a
simple graph to test the workflow.

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


builder = StateGraph(MultiAgentState)
builder.add_node("router", router_node)

builder.set_entry_point("router")
builder.add_edge('router', END)


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

Let's test our workflow with different types of questions to see
how it performs in action. This will help us evaluate whether the
router agent correctly assigns questions to the appropriate expert
agents.

```
thread = {"configurable": {"thread_id": "1"}}
for s in graph.stream({
    'question': "Does LangChain support Ollama?",
}, thread):
    print(s)
```

```
# {'router': {'question_type': 'LANGCHAIN'}}

thread = {"configurable": {"thread_id": "2"}}
for s in graph.stream({
    'question': "What info do we have in ecommerce_db.users table?",
}, thread):
    print(s)
# {'router': {'question_type': 'DATABASE'}}

thread = {"configurable": {"thread_id": "3"}}
for s in graph.stream({
    'question': "How are you?",
}, thread):
    print(s)

# {'router': {'question_type': 'GENERAL'}}
```

It's working well. I recommend you build complex graphs incrementally and test each step independently. With such an approach, you can ensure that each iteration works expectedly and can save you a significant amount of debugging time.

Next, let's create nodes for our expert agents. We will use the ReAct agent with the SQL tool we previously built as the database agent.

```
# database expert
sql_expert_system_prompt = '''
You are an expert in SQL, so you can help the team
to gather needed data to power their decisions.
You are very accurate and take into account all the nuances in data.
You use SQL to get the data before answering the question.
'''

def sql_expert_node(state: MultiAgentState):
    model = ChatOpenAI(model="gpt-4o-mini")
    sql_agent = create_react_agent(model, [execute_sql],
        state_modifier = sql_expert_system_prompt)
    messages = [HumanMessage(content=state['question'])]
```

```
    result = sql_agent.invoke({"messages": messages})
    return {'answer': result['messages'][-1].content}
```

For LangChain-related questions, we will use the ReAct agent. To enable the agent to answer questions about the library, we will equip it with a search engine tool. I chose Tavily for this purpose as it provides the search results optimised for LLM applications.

If you don't have an account, you can register to use Tavily for free (up to 1K requests per month). To get started, you will need to specify the Tavily API key in an environment variable.

```python
# search expert
from langchain_community.tools.tavily_search import TavilySearchResults
os.environ["TAVILY_API_KEY"] = 'tvly-...'
tavily_tool = TavilySearchResults(max_results=5)

search_expert_system_prompt = '''
You are an expert in LangChain and other technologies.
Your goal is to answer questions based on results provided by search.
You don't add anything yourself and provide only information baked by ot
'''

def search_expert_node(state: MultiAgentState):
    model = ChatOpenAI(model="gpt-4o-mini")
    sql_agent = create_react_agent(model, [tavily_tool],
        state_modifier = search_expert_system_prompt)
    messages = [HumanMessage(content=state['question'])]
    result = sql_agent.invoke({"messages": messages})
    return {'answer': result['messages'][-1].content}
```

For general questions, we will leverage a simple LLM model without specific tools.

```python
# general model
general_prompt = '''You're a friendly assistant and your goal is to answ
Please, don't provide any unchecked information and just tell that you d
'''
```

```python
def general_assistant_node(state: MultiAgentState):
    messages = [
        SystemMessage(content=general_prompt),
        HumanMessage(content=state['question'])
    ]
    model = ChatOpenAI(model="gpt-4o-mini")
    response = model.invoke(messages)
    return {"answer": response.content}
```

The last missing bit is a conditional function for routing. This will be quite straightforward—we just need to propagate the question type from the state defined by the router node.

```python
def route_question(state: MultiAgentState):
    return state['question_type']
```

Now, it's time to create our graph.

```python
builder = StateGraph(MultiAgentState)
builder.add_node("router", router_node)
builder.add_node('database_expert', sql_expert_node)
builder.add_node('langchain_expert', search_expert_node)
builder.add_node('general_assistant', general_assistant_node)
builder.add_conditional_edges(
    "router",
    route_question,
    {'DATABASE': 'database_expert',
     'LANGCHAIN': 'langchain_expert',
     'GENERAL': 'general_assistant'}
)

builder.set_entry_point("router")
builder.add_edge('database_expert', END)
builder.add_edge('langchain_expert', END)
builder.add_edge('general_assistant', END)
graph = builder.compile(checkpointer=memory)
```
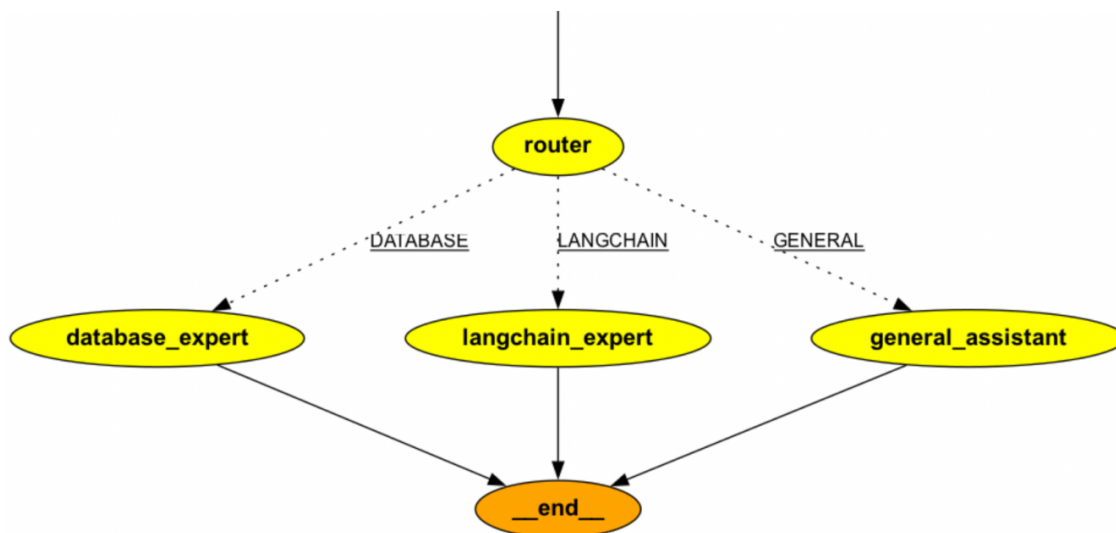
Now, we can test the setup on a couple of questions to see how well it performs.

```
thread = {"configurable": {"thread_id": "2"}}
results = []
for s in graph.stream({
    'question': "What info do we have in ecommerce_db.users table?",
}, thread):
    print(s)
    results.append(s)
print(results[-1]['database_expert']['answer'])

# The `ecommerce_db.users` table contains the following columns:
# 1. **User ID**: A unique identifier for each user.
# 2. **Country**: The country where the user is located.
# 3. **Is Active**: A flag indicating whether the user is active (1 for
# 4. **Age**: The age of the user.
# Here are some sample entries from the table:
#
# | User ID | Country        | Is Active | Age |
# |---------|----------------|-----------|-----|
# | 1000001 | United Kingdom | 0         | 70  |
# | 1000002 | France         | 1         | 87  |
# | 1000003 | France         | 1         | 88  |
# | 1000004 | Germany        | 1         | 25  |
# | 1000005 | Germany        | 1         | 48  |
#
# This gives an overview of the user data available in the table.
```

Good job! It gives a relevant result for the database-related question. Let's try asking about LangChain.

```python
thread = {"configurable": {"thread_id": "42"}}
results = []
for s in graph.stream({
    'question': "Does LangChain support Ollama?",
}, thread):
    print(s)
    results.append(s)

print(results[-1]['langchain_expert']['answer'])

# Yes, LangChain supports Ollama. Ollama allows you to run open-source
# large language models, such as Llama 2, locally, and LangChain provide
# a flexible framework for integrating these models into applications.
# You can interact with models run by Ollama using LangChain, and there
# specific wrappers and tools available for this integration.
#
# For more detailed information, you can visit the following resources:
# - [LangChain and Ollama Integration](https://js.langchain.com/v0.1/doc
# - [ChatOllama Documentation](https://js.langchain.com/v0.2/docs/integr
# - [Medium Article on Ollama and LangChain](https://medium.com/@abonia/
```

Fantastic! Everything is working well, and it's clear that Tavily's search is effective for LLM applications.

Adding human-in-the-loop interactions

We've done an excellent job creating a tool to answer questions. However, in many cases, it's beneficial to keep a human in the loop to approve proposed actions or provide additional feedback. Let's add a step where we can collect feedback from a human before returning the final result to the user.

The simplest approach is to add two additional nodes:

- A `human` node to gather feedback,

- An `editor` node to revisit the answer, taking into account the feedback.

Let's create these nodes:

- **Human node:** This will be a dummy node, and it won't perform any actions.

- **Editor node:** This will be an LLM model that receives all the relevant information (customer question, draft answer and provided feedback) and revises the final answer.

```python
def human_feedback_node(state: MultiAgentState):
    pass


editor_prompt = '''You're an editor and your goal is to provide the fina
You don't add any information on your own. You use friendly and professi
In the output please provide the final answer to the customer without ad
Here's all the information you need.

Question from customer:
----
{question}
----
Draft answer:
----
{answer}
----
Feedback:
----
{feedback}
----
'''

def editor_node(state: MultiAgentState):
```

```python
    messages = [
        SystemMessage(content=editor_prompt.format(question = state['questio
    ]
    model = ChatOpenAI(model="gpt-4o-mini")
    response = model.invoke(messages)
    return {"answer": response.content}
```

Let's add these nodes to our graph. Additionally, we need to introduce an interruption before the human node to ensure that the process pauses for human feedback.

```python
 builder = StateGraph(MultiAgentState)
builder.add_node("router", router_node)
builder.add_node('database_expert', sql_expert_node)
builder.add_node('langchain_expert', search_expert_node)
builder.add_node('general_assistant', general_assistant_node)
builder.add_node('human', human_feedback_node)
builder.add_node('editor', editor_node)

builder.add_conditional_edges(
    "router",
    route_question,
    {'DATABASE': 'database_expert',
    'LANGCHAIN': 'langchain_expert',
    'GENERAL': 'general_assistant'}
)

builder.set_entry_point("router")

builder.add_edge('database_expert', 'human')
builder.add_edge('langchain_expert', 'human')
builder.add_edge('general_assistant', 'human')
builder.add_edge('human', 'editor')
builder.add_edge('editor', END)
graph = builder.compile(checkpointer=memory, interrupt_before = ['human'
```
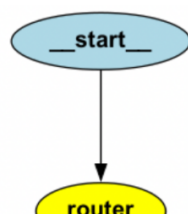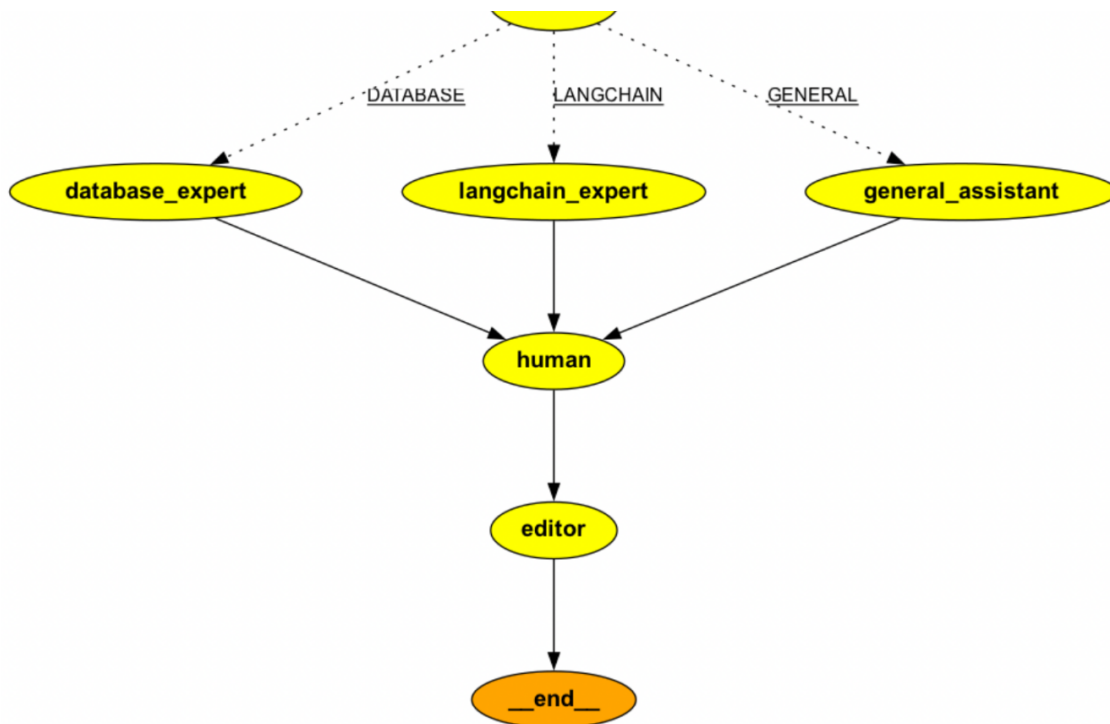
Now, when we run the graph, the execution will be stopped before the human node.

```python
thread = {"configurable": {"thread_id": "2"}}

for event in graph.stream({
    'question': "What are the types of fields in ecommerce_db.users tabl
}, thread):
    print(event)

# {'question_type': 'DATABASE', 'question': 'What are the types of field
# {'router': {'question_type': 'DATABASE'}}
# {'database_expert': {'answer': 'The `ecommerce_db.users` table has the
```

Let's get the customer input and update the state with the feedback.

```python
user_input = input("Do I need to change anything in the answer?")
# Do I need to change anything in the answer?
# It looks wonderful. Could you only make it a bit friendlier please?

graph.update_state(thread, {"feedback": user_input}, as_node="human")
```

We can check the state to confirm that the feedback has been populated and that the next node in the sequence is `editor`.

```
print(graph.get_state(thread).values['feedback'])
# It looks wonderful. Could you only make it a bit friendlier please?

print(graph.get_state(thread).next)
# ('editor',)
```

We can just continue the execution. Passing `None` as input will resume the process from the point where it was paused.

```
for event in graph.stream(None, thread, stream_mode="values"):
  print(event)

print(event['answer'])

# Hello! The `ecommerce_db.users` table has the following fields:
# 1. **user_id**: UInt64
# 2. **country**: String
# 3. **is_active**: UInt8
# 4. **age**: UInt64
# Have a nice day!
```

The editor took our feedback into account and added some polite words to our final message. That's a fantastic result!

We can implement human-in-the-loop interactions in a more agentic way by equipping our editor with the Human tool.

Let's adjust our editor. I've slightly changed the prompt and added the tool to the agent.

```
from langchain_community.tools import HumanInputRun
human_tool = HumanInputRun()

editor_agent_prompt = '''You're an editor and your goal is to provide th
```

```
If you need any clarifications or need feedback, please, use human. Alwa
You don't add any information on your own. You use friendly and professi
In the output please provide the final answer to the customer without ad
Here's all the information you need.

Question from customer:
----
{question}
----
Draft answer:
----
{answer}
----
'''

model = ChatOpenAI(model="gpt-4o-mini")
editor_agent = create_react_agent(model, [human_tool])
messages = [SystemMessage(content=editor_agent_prompt.format(question =
editor_result = editor_agent.invoke({"messages": messages})

# Is the draft answer complete and accurate for the customer's question
# Yes, but could you please make it friendlier.

print(editor_result['messages'][-1].content)
# The `ecommerce_db.users` table has the following fields:
# 1. **user_id**: UInt64
# 2. **country**: String
# 3. **is_active**: UInt8
# 4. **age**: UInt64
#
# If you have any more questions, feel free to ask!
```

So, the editor reached out to the human with the question, "Is the draft answer complete and accurate for the customer's question about the types of fields in the ecommerce_db.users table?". After receiving feedback, the editor refined the answer to make it more user-friendly.

Let's update our main graph to incorporate the new agent instead

of using the two separate nodes. With this approach, we don't need interruptions any more.

```python
def editor_agent_node(state: MultiAgentState):
    model = ChatOpenAI(model="gpt-4o-mini")
    editor_agent = create_react_agent(model, [human_tool])
    messages = [SystemMessage(content=editor_agent_prompt.format(question
    result = editor_agent.invoke({"messages": messages})
    return {'answer': result['messages'][-1].content}

builder = StateGraph(MultiAgentState)
builder.add_node("router", router_node)
builder.add_node('database_expert', sql_expert_node)
builder.add_node('langchain_expert', search_expert_node)
builder.add_node('general_assistant', general_assistant_node)
builder.add_node('editor', editor_agent_node)

builder.add_conditional_edges(
    "router",
    route_question,
    {'DATABASE': 'database_expert',
     'LANGCHAIN': 'langchain_expert',
      'GENERAL': 'general_assistant'}
)

builder.set_entry_point("router")

builder.add_edge('database_expert', 'editor')
builder.add_edge('langchain_expert', 'editor')
builder.add_edge('general_assistant', 'editor')
builder.add_edge('editor', END)

graph = builder.compile(checkpointer=memory)

thread = {"configurable": {"thread_id": "42"}}
results = []

for event in graph.stream({
    'question': "What are the types of fields in ecommerce_db.users table?
}, thread):
    print(event)
    results.append(event)
```

This graph will work similarly to the previous one. I personally prefer this approach since it leverages tools, making the solution more agile. For example, agents can reach out to humans multiple times and refine questions as needed.

That's it. We've built a multi-agent system that can answer questions from different domains and take into account human feedback.

> You can find the complete code on GitHub.

Summary

In this article, we've explored the LangGraph library and its application for building single and multi-agent workflows. We've examined a range of its capabilities, and now it's time to summarise its strengths and weaknesses. Also, it will be useful to compare LangGraph with CrewAI, which we discussed in my previous article.

Overall, I find LangGraph quite a powerful framework for building complex LLM applications:

- LangGraph is a low-level framework that offers extensive customisation options, allowing you to build precisely what you need.

- Since LangGraph is built on top of LangChain, it's seamlessly integrated into its ecosystem, making it easy to leverage existing tools and components.

However, there are areas where LangGrpah could be improved:

- The agility of LangGraph comes with a higher entry barrier. While you can understand the concepts of CrewAI within 15–30 minutes, it takes some time to get comfortable and up to speed with LangGraph.

- LangGraph provides you with a higher level of control, but it misses some cool prebuilt features of CrewAI, such as collaboration or ready-to-use RAG tools.

- LangGraph doesn't enforce best practices like CrewAI does (for example, role-playing or guardrails). So it can lead to poorer results.

I would say that CrewAI is a better framework for newbies and common use cases because it helps you get good results quickly and provides guidance to prevent mistakes.

If you want to build an advanced application and need more control, LangGraph is the way to go. Keep in mind that you'll need to invest time in learning LangGraph and should be fully responsible for the final solution, as the framework won't provide guidance to help you avoid common mistakes.

> Thank you a lot for reading this article. I hope this article was insightful for you. If you have any follow-up questions or comments, please leave them in the comments section.

Reference

This article is inspired by the "AI Agents in LangGraph" short course from DeepLearning.AI.

WRITTEN BY

Mariya Mansurova