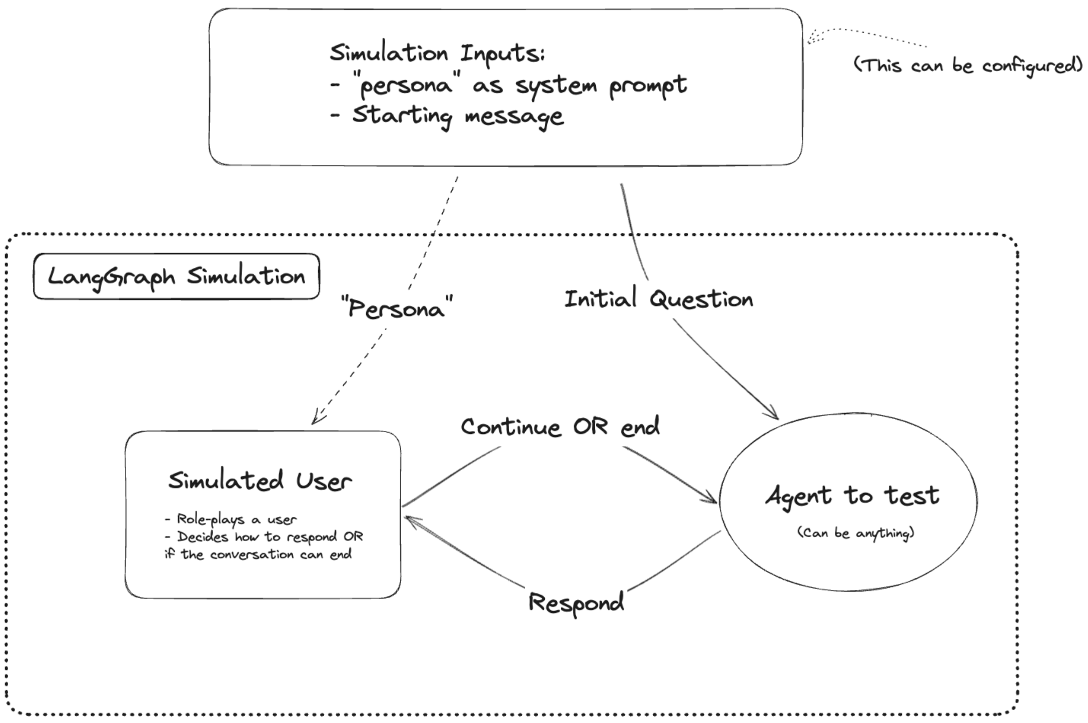
Chat Bot Evaluation as Multi-agent Simulation

When building a chat bot, such as a customer support assistant, it can be hard to properly evaluate your bot's performance. It's time-consuming to have to manually interact with it intensively for each code change.

One way to make the evaluation process easier and more reproducible is to simulate a user interaction.

With LangGraph, it's easy to set this up. Below is an example of how to create a "virtual user" to simulate a conversation.

The overall simulation looks something like this:



Setup

First, let's install the required packages and set our API keys

```
%%capture --no-stderr
%pip install -U langgraph langchain langchain_openai
```

```python
import getpass
import os


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


_set_if_undefined("OPENAI_API_KEY")
```

Set up LangSmith for LangGraph development

Sign up for LangSmith to quickly spot issues and improve the performance of your LangGraph projects. LangSmith lets you use trace data to debug, test, and monitor your LLM apps built with LangGraph — read more about how to get started here.

Define Chat Bot

Next, we will define our chat bot. For this notebook, we assume the bot's API accepts a list of messages and responds with a message. If you want to update this, all you'll have to change is this section and the "get_messages_for_agent" function in the simulator below.

The implementation within `my_chat_bot` is configurable and can even be run on another system (e.g., if your system isn't running in python).

```python
from typing import List

import openai


# This is flexible, but you can define your agent here, or call your agent API 
here.
def my_chat_bot(messages: List[dict]) -> dict:
    system_message = {
        "role": "system",
        "content": "You are a customer support agent for an airline.",
    }
    messages = [system_message] + messages
    completion = openai.chat.completions.create(
        messages=messages, model="gpt-3.5-turbo"
    )
    return completion.choices[0].message.model_dump()
```

```
my_chat_bot([{"role": "user", "content": "hi!"}])
```

```
{'content': 'Hello! How can I assist you today?',
 'role': 'assistant',
 'function_call': None,
 'tool_calls': None}
```

Define Simulated User

We're now going to define the simulated user. This can be anything we want, but we're going to build it as a LangChain bot.

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI

system_prompt_template = """You are a customer of an airline company. \
You are interacting with a user who is a customer support person. \

{instructions}

When you are finished with the conversation, respond with a single word
'FINISHED'"""

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt_template),
        MessagesPlaceholder(variable_name="messages"),
    ]
)
instructions = """Your name is Harrison. You are trying to get a refund for
the trip you took to Alaska. \
You want them to give you ALL the money back. \
This trip happened 5 years ago."""

prompt = prompt.partial(name="Harrison", instructions=instructions)

model = ChatOpenAI()

simulated_user = prompt | model
```

API Reference: ChatPromptTemplate | MessagesPlaceholder | ChatOpenAI

```
from langchain_core.messages import HumanMessage

messages = [HumanMessage(content="Hi! How can I help you?")]
simulated_user.invoke({"messages": messages})
```

API Reference: HumanMessage

```
AIMessage(content='Hi, I would like to request a refund for a trip I took with
your airline company to Alaska. Is it possible to get a refund for that
trip?')
```

Define the Agent Simulation

The code below creates a LangGraph workflow to run the simulation. The main components are:

1. The two nodes: one for the simulated user, the other for the chat bot.
2. The graph itself, with a conditional stopping criterion.

Read the comments in the code below for more information.

Define nodes

First, we define the nodes in the graph. These should take in a list of messages and return a list of messages to ADD to the state. These will be thing wrappers around the chat bot and simulated user we have above.

Note: one tricky thing here is which messages are which. Because both the chat bot AND our simulated user are both LLMs, both of them will resond with AI messages. Our state will be a list of alternating Human and AI messages. This means that for one of the nodes, there will need to be some logic that flips the AI and human roles. In this example, we will assume that HumanMessages are messages from the simulated user. This means that we need some logic in the simulated user node to swap AI and Human messages.

First, let's define the chat bot node

```python
from langchain_community.adapters.openai import convert_message_to_dict
from langchain_core.messages import AIMessage


def chat_bot_node(state):
    messages = state["messages"]
    # Convert from LangChain format to the OpenAI format, which our chatbot
function expects.
    messages = [convert_message_to_dict(m) for m in messages]
    # Call the chat bot
    chat_bot_response = my_chat_bot(messages)
    # Respond with an AI Message
    return {"messages": [AIMessage(content=chat_bot_response["content"])]}
```

API Reference: convert_message_to_dict | AIMessage

Next, let's define the node for our simulated user. This will involve a little logic to swap the roles of the messages.

```python
def _swap_roles(messages):
    new_messages = []
    for m in messages:
        if isinstance(m, AIMessage):
            new_messages.append(HumanMessage(content=m.content))
        else:
            new_messages.append(AIMessage(content=m.content))
    return new_messages


def simulated_user_node(state):
    messages = state["messages"]
    # Swap roles of messages
    new_messages = _swap_roles(messages)
    # Call the simulated user
    response = simulated_user.invoke({"messages": new_messages})
    # This response is an AI message - we need to flip this to be a human
message
    return {"messages": [HumanMessage(content=response.content)]}
```

Define edges

We now need to define the logic for the edges. The main logic occurs after the simulated user goes, and it should lead to one of two outcomes:

- Either we continue and call the customer support bot
- Or we finish and the conversation is over

So what is the logic for the conversation being over? We will define that as either the Human chatbot responds with `FINISHED` (see the system prompt) OR the conversation is more than 6 messages long (this is an arbitrary number just to keep this example short).

```
def should_continue(state):
    messages = state["messages"]
    if len(messages) > 6:
        return "end"
    elif messages[-1].content == "FINISHED":
        return "end"
    else:
        return "continue"
```

Define graph

We can now define the graph that sets up the simulation!

```
from langgraph.graph import END, StateGraph, START
from langgraph.graph.message import add_messages
from typing import Annotated
from typing_extensions import TypedDict
```

```python
class State(TypedDict):
    messages: Annotated[list, add_messages]


graph_builder = StateGraph(State)
graph_builder.add_node("user", simulated_user_node)
graph_builder.add_node("chat_bot", chat_bot_node)
# Every response from  your chat bot will automatically go to the
# simulated user
graph_builder.add_edge("chat_bot", "user")
graph_builder.add_conditional_edges(
    "user",
    should_continue,
    # If the finish criteria are met, we will stop the simulation,
    # otherwise, the virtual user's message will be sent to your chat bot
    {
        "end": END,
        "continue": "chat_bot",
    },
)
# The input will first go to your chat bot
graph_builder.add_edge(START, "chat_bot")
simulation = graph_builder.compile()
```

API Reference: END | StateGraph | START | add_messages

Run Simulation

Now we can evaluate our chat bot! We can invoke it with empty messages (this will simulate letting the chat bot start the initial conversation)

```python
for chunk in simulation.stream({"messages": []}):
    # Print out all events aside from the final end chunk
    if END not in chunk:
        print(chunk)
        print("----")
```

```
{'chat_bot': AIMessage(content='How may I assist you today regarding your
flight or any other concerns?')}
----
{'user': HumanMessage(content='Hi, my name is Harrison. I am reaching out to
request a refund for a trip I took to Alaska with your airline company. The
trip occurred about 5 years ago. I would like to receive a refund for the
entire amount I paid for the trip. Can you please assist me with this?')}
----
{'chat_bot': AIMessage(content="Hello, Harrison. Thank you for reaching out to
us. I understand you would like to request a refund for a trip you took to
Alaska five years ago. I'm afraid that our refund policy typically has a
specific timeframe within which refund requests must be made. Generally,
refund requests need to be submitted within 24 to 48 hours after the booking
is made, or in certain cases, within a specified cancellation
period.\n\nHowever, I will do my best to assist you. Could you please provide
me with some additional information? Can you recall any specific details about
the booking, such as the flight dates, booking reference or confirmation
number? This will help me further look into the possibility of processing a
refund for you.")}
----
{'user': HumanMessage(content="Hello, thank you for your response. I apologize
for not requesting the refund earlier. Unfortunately, I don't have the
specific details such as the flight dates, booking reference, or confirmation
number at the moment. Is there any other way we can proceed with the refund
request without these specific details? I would greatly appreciate your
assistance in finding a solution.")}
----
{'chat_bot': AIMessage(content="I understand the situation, Harrison. Without
specific details like flight dates, booking reference, or confirmation number,
it becomes challenging to locate and process the refund accurately. However, I
can still try to help you.\n\nTo proceed further, could you please provide me
with any additional information you might remember? This could include the
approximate date of travel, the departure and arrival airports, the names of
the passengers, or any other relevant details related to the booking. The more
information you can provide, the better we can investigate the possibility of
processing a refund for you.\n\nAdditionally, do you happen to have any
documentation related to your trip, such as receipts, boarding passes, or
emails from our airline? These documents could assist in verifying your trip
and processing the refund request.\n\nI apologize for any inconvenience
caused, and I'll do my best to assist you further based on the information you
can provide.")}
----
{'user': HumanMessage(content="I apologize for the inconvenience caused.
Unfortunately, I don't have any additional information or documentation
related to the trip. It seems that I am unable to provide you with the
necessary details to process the refund request. I understand that this may
limit your ability to assist me further, but I appreciate your efforts in
trying to help. Thank you for your time. \n\nFINISHED")}
----
{'chat_bot': AIMessage(content="I understand, Harrison. I apologize for any
inconvenience caused, and I appreciate your understanding. If you happen to
locate any additional information or documentation in the future, please don't
hesitate to reach out to us again. Our team will be more than happy to assist
you with your refund request or any other travel-related inquiries. Thank you
for contacting us, and have a great day!")}
----
{'user': HumanMessage(content='FINISHED')}
----
```