AI Agent notes

LangChain Basics:

LangChain uses an LLM and a Prompt to chain together actions. You usually declare a LangChain LLM by:

```
os.environ["OPENAI_API_KEY"] = open_ai_api_key

llm = ChatOpenAI(api_key = open_ai_api_key) # The api_key is optional as it is se
```

You can load a webpage or document and vectorize it and use it as a tool by:

```
loader = WebBaseLoader("https://toxigon.com/minecraft-complete-guide-and-w
docs = loader.load()
text_splitter = RecursiveCharacterTextSplitter()
documents = text_splitter.split_documents(docs)
embeddings = OpenAIEmbeddings()
vector = FAISS.from_documents(documents, embeddings)
retriever = vector.as_retriever()
minecraft_guide_tool = create_retriever_tool(retriever, "minecraft_guide_search",
```

This loads the document → splits text into chunks → embeds and factorizes → sets up a retriever tool

Tavily is a common search tool that can be done in the same way

```
os.environ['TAVILY_API_KEY'] = tavily_api_key

search_tool = TavilySearchResults(api_key = tavily_api_key)
```

An agent can be created by combining a prompt, llm, and tools:

```
tools = [minecraft_guide_tool, search_tool]
prompt = hub.pull("hwchase17/openai-functions-agent")
agent = create_openai_functions_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent = agent, tools = tools, verbose = True)
```

Chainlit can be used to visualize the agent in a chat based system easily:

- @cl.on_chat_start is used to setup the agent and page

```
@cl.on_chat_start
async def on_chat_start():
    agent_executor = setup_agent()
    cl.user_session.set("agent_executor", agent_executor)
    await cl.Message(content = "Hi, I'm a Minecraft and General Web Search Ager
```

- @cl.on_message is used to respond to messages

```
@cl.on_message
async def on_message(message: cl.Message):
    agent_executor = cl.user_session.get("agent_executor")
    result = agent_executor.invoke({"input": message.content})
    await cl.Message(content = result["output"]).send()
```

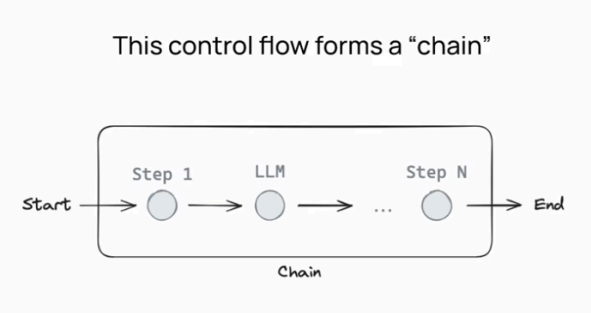
In this, user_session can be used to store values across users, in this case the agent itself. Agent_executor is invoked to respond to the prompt, and output in the json corresponds to the result.

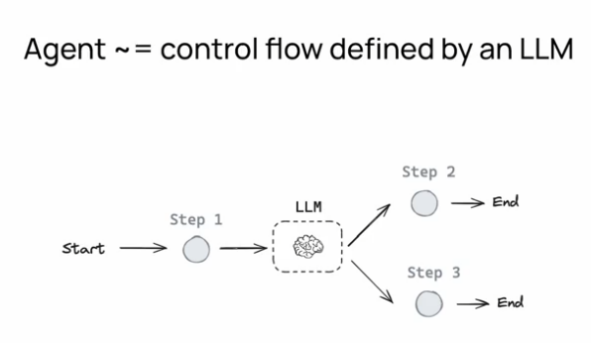
LangGraph Basics

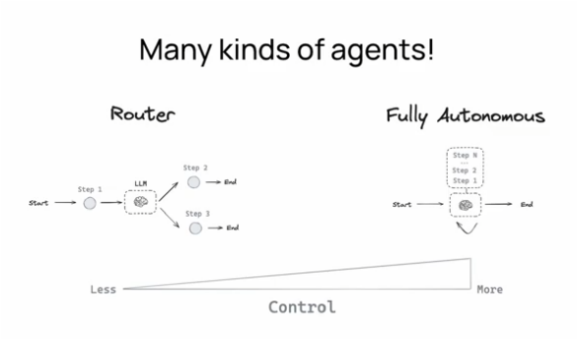
A normal language model is start → llm → end. So a control flow is used, where start → step 1 → llm → ... → step N → end

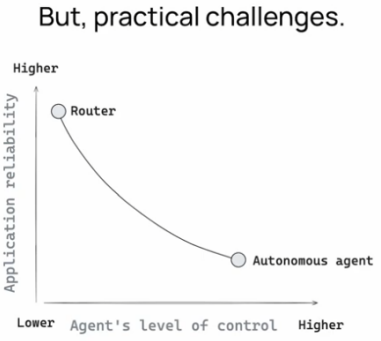
The steps and the llm is called a chain:

These are reliable as they have the same flow every time, but we want an LLM to pick their own control flow

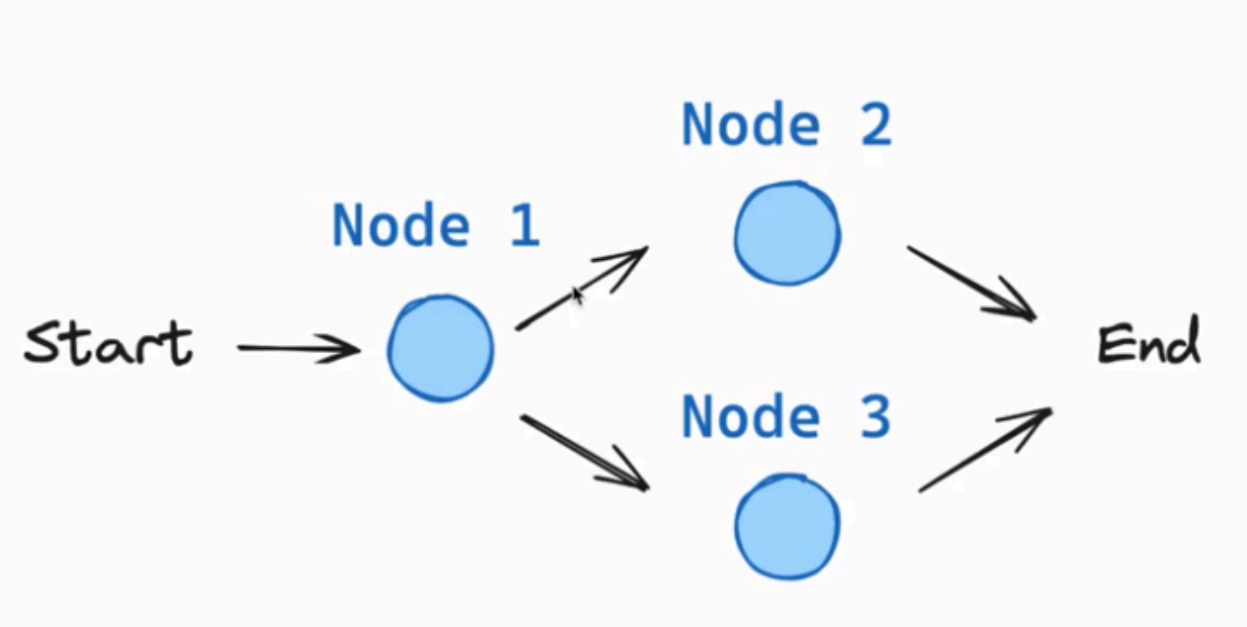








Normal edge is when one edge connects to another, conditional is where there’s a branching path



States are ways to preserve input and memory across the graph:

```
from typing_extensions import TypedDict

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

Nodes are used to modify the value of the state itself:

```
def node_1(state):
    print("---Node 1---")
    return {"graph_state": state['graph_state'] +" I am"}

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

def node_3(state):
```

```
    print("---Node 3---")
    return {"graph_state": state['graph_state'] +" sad!"}
```

Conditional edges take in the graph state and convert to a different node:

```
import random
from typing import Literal

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

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

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

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

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

You can build the complete graph by:

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

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

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

# Add
graph = builder.compile()

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

Now, you can use this graph by invoking it:

```
graph.invoke({"graph_state" : "Hi, this is Lance."})

'''
---Node 1---
---Node 3---
{'graph_state': 'Hi, this is Lance. I am sad!'}
'''
```

Now to build a simple reAct agent:

- act - let model call specific tools
- observer - pass the tool output to model
- reason - let the model reason about the tool output and decide what to do next

```
from langchain_openai import ChatOpenAI

def multiply(a: int, b: int) -> int:
```

```
    """Multiply a and b.

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

# This will be a tool
def add(a: int, b: int) → int:
    """Adds a and b.

    Args:
        a: first int
        b: second int
    """
    return a + b

def divide(a: int, b: int) → float:
    """Divide a and b.

    Args:
        a: first int
        b: second int
    """
    return a / b

tools = [add, multiply, divide]
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools)
```

Functions can be made into tools and attached by setting .bind_tools. These tools can be accessed by the llm if necessary.

```
from langgraph.graph import MessagesState
from langchain_core.messages import HumanMessage, SystemMessage

# System message
sys_msg = SystemMessage(content="You are a helpful assistant tasked with per

# Node
def assistant(state: MessagesState):
   return {"messages": [llm_with_tools.invoke([sys_msg] + state["messages"])]}
```

By giving the LLM a prompt it is more in tune to use tools if necessary, and this sets up an assistant node which can use the llm's tools through invoking and return a message.

```
from langgraph.graph import START, StateGraph
from langgraph.prebuilt import tools_condition, ToolNode
from IPython.display import Image, display

# Graph
builder = StateGraph(MessagesState)

# Define nodes: these do the work
builder.add_node("assistant", assistant)
builder.add_node("tools", ToolNode(tools))

# Define edges: these determine how the control flow moves
builder.add_edge(START, "assistant")
builder.add_conditional_edges(
    "assistant",
    # If the latest message (result) from assistant is a tool call -> tools_condition ro
    # If the latest message (result) from assistant is a not a tool call -> tools_condit
    tools_condition,
)
builder.add_edge("tools", "assistant")
```

```
react_graph = builder.compile()

# Show
display(Image(react_graph.get_graph(xray=True).draw_mermaid_png()))
```

This creates a graph where the assistant connects to the tools as a tool node through a conditional edge, using tools_condition for the assistant to decide if it wants to use the tools or not. Then an edge back to the assistant so it can reason (”reAct”).

This agent doesn’t have memory, so to add this we use the checkpointer with threads:

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

# Specify a thread
config = {"configurable": {"thread_id": "1"}}

# Specify an input
messages = [HumanMessage(content="Add 3 and 4.")]

# Run
messages = react_graph_memory.invoke({"messages": messages},config)
for m in messages['messages']:
    m.pretty_print()

messages = [HumanMessage(content="Multiply that by 2.")]
messages = react_graph_memory.invoke({"messages": messages}, config)
for m in messages['messages']:
    m.pretty_print()
```

Now, by specifying a thread_id, instead of multiply that by 2 not knowing what context to use, it uses the previous message context and answers:

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

Add 3 and 4.
================================== Ai Message ==================================
Tool Calls:
  add (call_MSupVAgej4PShIZs7NXOE6En)
 Call ID: call_MSupVAgej4PShIZs7NXOE6En
  Args:
    a: 3
    b: 4
================================= Tool Message =================================
Name: add

7
================================== Ai Message ==================================

The sum of 3 and 4 is 7.
================================ Human Message =================================

Multiply that by 2.
================================== Ai Message ==================================
Tool Calls:
  multiply (call_fWN7lnSZZm82tAg7RGeuWusO)
 Call ID: call_fWN7lnSZZm82tAg7RGeuWusO
  Args:
    a: 7
    b: 2
================================= Tool Message =================================
Name: multiply

14
================================== Ai Message ==================================

The result of multiplying 7 by 2 is 14.
```

There are multiple says to handle state:

```
from typing import Literal

class TypedDictState(TypedDict):
    name: str
```

```
    mood: Literal["happy","sad"]

def node_1(state):
    print("---Node 1---")
    return {"name": state['name'] + " is ... "}

# OR

from dataclasses import dataclass

@dataclass
class DataclassState:
    name: str
    mood: Literal["happy","sad"]

def node_1(state):
    print("---Node 1---")
    return {"name": state.name + " is ... "}

# These don't have type safety so Pydantic helps

from pydantic import BaseModel, field_validator, ValidationError

class PydanticState(BaseModel):
    name: str
    mood: str # "happy" or "sad"

    @field_validator('mood')
    @classmethod
    def validate_mood(cls, value):
        # Ensure the mood is either "happy" or "sad"
        if value not in ["happy", "sad"]:
            raise ValueError("Each mood must be either 'happy' or 'sad'")
        return value

def node_1(state):
```

```
    print("---Node 1---")
    return {"name": state.name + " is ... "}

try:
    state = PydanticState(name="John Doe", mood="mad")
except ValidationError as e:
    print("Validation Error:", e)

'''
Validation Error: 1 validation error for PydanticState
mood
  Value error, Each mood must be either 'happy' or 'sad' [type=value_error, input_
    For further information visit https://errors.pydantic.dev/2.11/v/value_error
'''
```

State reducers are ways to update state through channels:

```
class State(TypedDict):
    foo: int

def node_1(state):
    print("---Node 1---")
    return {"foo": state['foo'] + 1}

def node_2(state):
    print("---Node 2---")
    return {"foo": state['foo'] + 1}

def node_3(state):
    print("---Node 3---")
    return {"foo": state['foo'] + 1}

# Build graph
builder = StateGraph(State)
```

```
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)
builder.add_node("node_3", node_3)

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

# Add
graph = builder.compile()

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

LangGraph doesn’t know whether to use node_2 or node_3’s state so it throws an error

```
from operator import add
from typing import Annotated

class State(TypedDict):
    foo: Annotated[list[int], add]

def node_1(state):
    print("---Node 1---")
    return {"foo": [state['foo'][0] + 1]}

# This appends: graph.invoke({"foo" : [1]}) → {'foo': [1, 2]}

def reduce_list(left: list | None, right: list | None) → list:
    """Safely combine two lists, handling cases where either or both inputs might
```

```
    Args:
        left (list | None): The first list to combine, or None.
        right (list | None): The second list to combine, or None.

    Returns:
        list: A new list containing all elements from both input lists.
              If an input is None, it's treated as an empty list.
    """
    if not left:
        left = []
    if not right:
        right = []
    return left + right

class DefaultState(TypedDict):
    foo: Annotated[list[int], add]

class CustomReducerState(TypedDict):
    foo: Annotated[list[int], reduce_list]

# This builds a custom reducer that can handle null inputs
```

Messages can be reduced in the same way

```
from langgraph.graph import MessagesState

# Define a custom TypedDict that includes a list of messages with add_message
class CustomMessagesState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
    added_key_1: str
    added_key_2: str
    # etc
```

```
# Use MessagesState, which includes the messages key with add_messages rec
class ExtendedMessagesState(MessagesState):
    # Add any keys needed beyond messages, which is pre-built
    added_key_1: str
    added_key_2: str
    # etc
```

Both of these do the same thing, append messages using add_messages, and have extra keys for other uses

Messages can be rewriten by providing the same id

```
# Initial state
initial_messages = [AIMessage(content="Hello! How can I assist you?", name="M
                    HumanMessage(content="I'm looking for information on marine biol
                   ]

# New message to add
new_message = HumanMessage(content="I'm looking for information on whales

# Test
add_messages(initial_messages , new_message)
```

Messages can be deleted using the RemoveMessage in the reducer:

```
from langchain_core.messages import RemoveMessage

# Message list
messages = [AIMessage("Hi.", name="Bot", id="1")]
messages.append(HumanMessage("Hi.", name="Lance", id="2"))
messages.append(AIMessage("So you said you were researching ocean mamma
messages.append(HumanMessage("Yes, I know about whales. But what others s

# Isolate messages to delete
```

```
delete_messages = [RemoveMessage(id=m.id) for m in messages[:-2]]
print(delete_messages)

This deletes everything except the last two messages
```

Multi schema graphs exist where information can be limited within nodes by providing different states:

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

class OverallState(TypedDict):
    foo: int

class PrivateState(TypedDict):
    baz: int

def node_1(state: OverallState) -> PrivateState:
    print("---Node 1---")
    return {"baz": state['foo'] + 1}

def node_2(state: PrivateState) -> OverallState:
    print("---Node 2---")
    return {"foo": state['baz'] + 1}

# Build graph
builder = StateGraph(OverallState)
builder.add_node("node_1", node_1)
builder.add_node("node_2", node_2)

# Logic
builder.add_edge(START, "node_1")
builder.add_edge("node_1", "node_2")
```

```
builder.add_edge("node_2", END)

# Add
graph = builder.compile()

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

Input and output schema can be used to simplify inner workings by providing only necessary parameters for input and output:

```
class InputState(TypedDict):
    question: str

class OutputState(TypedDict):
    answer: str

class OverallState(TypedDict):
    question: str
    answer: str
    notes: str

def thinking_node(state: InputState):
    return {"answer": "bye", "notes": "... his is name is Lance"}

def answer_node(state: OverallState) -> OutputState:
    return {"answer": "bye Lance"}

graph = StateGraph(OverallState, input=InputState, output=OutputState)
graph.add_node("answer_node", answer_node)
graph.add_node("thinking_node", thinking_node)
graph.add_edge(START, "thinking_node")
graph.add_edge("thinking_node", "answer_node")
graph.add_edge("answer_node", END)
```

```
graph = graph.compile()

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

graph.invoke({"question":"hi"})
```

An overload of messages can create high token usage, so there are many ways to prevent this:

This reduces to keep only the last two messages

```
class InputState(TypedDict):
    question: str

class OutputState(TypedDict):
    answer: str

class OverallState(TypedDict):
    question: str
    answer: str
    notes: str

def thinking_node(state: InputState):
    return {"answer": "bye", "notes": "... his is name is Lance"}

def answer_node(state: OverallState) -> OutputState:
    return {"answer": "bye Lance"}

graph = StateGraph(OverallState, input=InputState, output=OutputState)
graph.add_node("answer_node", answer_node)
graph.add_node("thinking_node", thinking_node)
graph.add_edge(START, "thinking_node")
```

```
graph.add_edge("thinking_node", "answer_node")
graph.add_edge("answer_node", END)

graph = graph.compile()

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

graph.invoke({"question":"hi"})
```

This filters to keep only the last message:

```
# Node
def chat_model_node(state: MessagesState):
    return {"messages": [llm.invoke(state["messages"][-1:])]}

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

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

This clips off based on tokens:

```
from langchain_core.messages import trim_messages

# Node
def chat_model_node(state: MessagesState):
    messages = trim_messages(
```

```
            state["messages"],
            max_tokens=100,
            strategy="last", # Starts from last message
            token_counter=ChatOpenAI(model="gpt-4o"),
            allow_partial=False, # if true allows partial message text
        )
    return {"messages": [llm.invoke(messages)]}

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

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

You can build long term memory instead of using these tricks too by adding summation:

```
from langgraph.graph import MessagesState
class State(MessagesState):
    summary: str

from langchain_core.messages import SystemMessage, HumanMessage, Remov

# Define the logic to call the model
def call_model(state: State):

    # Get summary if it exists
    summary = state.get("summary", "")

    # If there is summary, then we add it
```

```
    if summary:

        # Add summary to system message
        system_message = f"Summary of conversation earlier: {summary}"

        # Append summary to any newer messages
        messages = [SystemMessage(content=system_message)] + state["messag

    else:
        messages = state["messages"]

    response = model.invoke(messages)
    return {"messages": response}

def summarize_conversation(state: State):

    # First, we get any existing summary
    summary = state.get("summary", "")

    # Create our summarization prompt
    if summary:

        # A summary already exists
        summary_message = (
            f"This is summary of the conversation to date: {summary}\n\n"
            "Extend the summary by taking into account the new messages above:"
        )

    else:
        summary_message = "Create a summary of the conversation above:"

    # Add prompt to our history
    messages = state["messages"] + [HumanMessage(content=summary_messag
    response = model.invoke(messages)

    # Delete all but the 2 most recent messages
```

```
    delete_messages = [RemoveMessage(id=m.id) for m in state["messages"][:-2]
    return {"summary": response.content, "messages": delete_messages}

from langgraph.graph import END
# Determine whether to end or summarize the conversation
def should_continue(state: State):

    """Return the next node to execute."""

    messages = state["messages"]

    # If there are more than six messages, then we summarize the conversation
    if len(messages) > 6:
        return "summarize_conversation"

    # Otherwise we can just end
    return END

from IPython.display import Image, display
from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START

# Define a new graph
workflow = StateGraph(State)
workflow.add_node("conversation", call_model)
workflow.add_node(summarize_conversation)

# Set the entrypoint as conversation
workflow.add_edge(START, "conversation")
workflow.add_conditional_edges("conversation", should_continue)
workflow.add_edge("summarize_conversation", END)

# Compile
memory = MemorySaver()
```

```
graph = workflow.compile(checkpointer=memory)
display(Image(graph.get_graph().draw_mermaid_png()))
```

This does a few things:

- The state also gets a summary key added
- If the model is called an a summary is present system text including it gets appended
- Summarize conversation creates a summary message and adds it to state and trims state
- Should_continue only adds summary if more than 6 messages present
- Memory saver adds a checkpointer so memory is saved across threads

Furthermore, you can enhance this by connecting memory to a sqlite database or postgres:

```
import sqlite3
# In memory
conn = sqlite3.connect(":memory:", check_same_thread = False)

# Here is our checkpointer
from langgraph.checkpoint.sqlite import SqliteSaver
memory = SqliteSaver(conn)

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

With this state is persistent through thread_id

Streaming is another way to access outputs from a chatbot:

Updates only stream the newest chunk that has been added in the graph:

```
# Create a thread
config = {"configurable": {"thread_id": "1"}}

# Start conversation - json
for chunk in graph.stream({"messages": [HumanMessage(content="hi! I'm Lance
    print(chunk)

# Start conversation - message
for chunk in graph.stream({"messages": [HumanMessage(content="hi! I'm Lance
    chunk['conversation']["messages"].pretty_print()
```

Values sends the current full state of the graph after every node is called:

```
# Start conversation, again
config = {"configurable": {"thread_id": "2"}}

# Start conversation
input_message = HumanMessage(content="hi! I'm Lance")
for event in graph.stream({"messages": [input_message]}, config, stream_mode=
    for m in event['messages']:
        m.pretty_print()
    print("---"*25)
```

We can stream individual tokens from the llm by specifying a thread and a specific event (node) in the graph:

```
node_to_stream = 'conversation'
config = {"configurable": {"thread_id": "4"}}
input_message = HumanMessage(content="Tell me about the 49ers NFL team")
async for event in graph.astream_events({"messages": [input_message]}, config
    # Get chat model tokens from a particular node
```

```
    if event["event"] == "on_chat_model_stream" and event['metadata'].get('langg
        print(event["data"])
```

Similar code exists in using the API

```
# Streams json values
client = get_client(url="http://127.0.0.1:2024")
thread = await client.threads.create()
# Input message
input_message = HumanMessage(content="Multiply 2 and 3")
async for event in client.runs.stream(thread["thread_id"],
                                      assistant_id="agent",
                                      input={"messages": [input_message]},
                                      stream_mode="values"):
    print(event)

# Streams messages:
from langchain_core.messages import convert_to_messages
thread = await client.threads.create()
input_message = HumanMessage(content="Multiply 2 and 3")
async for event in client.runs.stream(thread["thread_id"], assistant_id="agent", in
    messages = event.data.get('messages',None)
    if messages:
        print(convert_to_messages(messages)[-1])
    print('='*25)

# Formats and prints tool calls:
thread = await client.threads.create()
input_message = HumanMessage(content="Multiply 2 and 3")

def format_tool_calls(tool_calls):
    """
    Format a list of tool calls into a readable string.
```

```
    Args:
        tool_calls (list): A list of dictionaries, each representing a tool call.
            Each dictionary should have 'id', 'name', and 'args' keys.

    Returns:
        str: A formatted string of tool calls, or "No tool calls" if the list is empty.

    """

    if tool_calls:
        formatted_calls = []
        for call in tool_calls:
            formatted_calls.append(
                f"Tool Call ID: {call['id']}, Function: {call['name']}, Arguments: {call['arg
            )
        return "\n".join(formatted_calls)
    return "No tool calls"

async for event in client.runs.stream(
    thread["thread_id"],
    assistant_id="agent",
    input={"messages": [input_message]},
    stream_mode="messages",):

    # Handle metadata events
    if event.event == "metadata":
        print(f"Metadata: Run ID - {event.data['run_id']}")
        print("-" * 50)

    # Handle partial message events
    elif event.event == "messages/partial":
        for data_item in event.data:
            # Process user messages
            if "role" in data_item and data_item["role"] == "user":
                print(f"Human: {data_item['content']}")
            else:
```

```
            # Extract relevant data from the event
            tool_calls = data_item.get("tool_calls", [])
            invalid_tool_calls = data_item.get("invalid_tool_calls", [])
            content = data_item.get("content", "")
            response_metadata = data_item.get("response_metadata", {})

            if content:
                print(f"AI: {content}")

            if tool_calls:
                print("Tool Calls:")
                print(format_tool_calls(tool_calls))

            if invalid_tool_calls:
                print("Invalid Tool Calls:")
                print(format_tool_calls(invalid_tool_calls))

            if response_metadata:
                finish_reason = response_metadata.get("finish_reason", "N/A")
                print(f"Response Metadata: Finish Reason - {finish_reason}")

        print("-" * 50)
```

Breakpoints are ways to add human-in-the-loop into LangGraph:

```
graph = builder.compile(interrupt_before=["tools"], checkpointer=memory)
```

This makes it so before tools are run, there is an interrupt that stops the running of the graph. You can do many things at this point:

```
# Check graph state
state = graph.get_state(thread)
state.next
```

```
# Continue past the interrupt:
for event in graph.stream(None, thread, stream_mode="values"):
    event['messages'][-1].pretty_print()
```

Similar code exists for the API:

```
# Adding interrupts into the streaming method
async for chunk in client.runs.stream(
    thread["thread_id"],
    assistant_id="agent",
    input=initial_input,
    stream_mode="values",
    interrupt_before=["tools"],
):
    print(f"Receiving new event of type: {chunk.event}...")
    messages = chunk.data.get('messages', [])
    if messages:
        print(messages[-1])
    print("-" * 50)
```

Or you could add interrupts into the graph as shown before

You can also edit state while in an interrupt:

```
# No ID so will append message
graph.update_state(
    thread,
    {"messages": [HumanMessage(content="No, actually multiply 3 and 3!")]},
)
```

Similar code exists in the API:

```
# Direct editing
last_message = current_state['values']['messages'][-1]
last_message['content'] = "No, actually multiply 3 and 3!"
last_message

# Appending
await client.threads.update_state(thread['thread_id'], {"messages": last_message
```

Breakpoints can also be set dynamically in a node:

```
def step_2(state: State) -> State:
    # Let's optionally raise a NodeInterrupt if the length of the input is longer than
    if len(state['input']) > 5:
        raise NodeInterrupt(f"Received input that is longer than 5 characters: {state

    print("---Step 2---")
    return state
```

This will pause at the node, and even if continue is sent will not continue till the condition is met, it will keep rerunning the state till the condition is met.

Time traveling is a way to debug in langgraph. You can browse the history by using a list.

```
all_states = [s for s in graph.get_state_history(thread)]
all_states[-2]
```

You can replay from a state using the checkpointer:

```
to_replay = all_states[-2]
```

```
for event in graph.stream(None, to_replay.config, stream_mode="values"):
    event['messages'][-1].pretty_print()
```

This replays from to_replay instead of the current furthest state provided by the thread

You can fork a state by editing a state at the same id and replay from that point:

```
fork_config = graph.update_state(
    to_fork.config,
    {"messages": [HumanMessage(content='Multiply 5 and 3',
                               id=to_fork.values["messages"][0].id)]},
)
```

Rerunning from this can provide different cases for debugging

The LangGraph API has similar code:

```
# Looking at a previous state
states = await client.threads.get_history(thread['thread_id'])
to_replay = states[-2]
to_replay

# Replaying from a previous checkpoint
async for chunk in client.runs.stream(
    thread["thread_id"],
    assistant_id="agent",
    input=None,
    stream_mode="values",
    checkpoint_id=to_replay['checkpoint_id']
):
    print(f"Receiving new event of type: {chunk.event}...")
    print(chunk.data)
```

```
    print("\n\n")


# Forking a state and replaying
states = await client.threads.get_history(thread['thread_id'])
to_fork = states[-2]
to_fork['values']

forked_input = {"messages": HumanMessage(content="Multiply 3 and 3",
                                         id=to_fork['values']['messages'][0]['id'])}

forked_config = await client.threads.update_state(
    thread["thread_id"],
    forked_input,
    checkpoint_id=to_fork['checkpoint_id']
)

async for chunk in client.runs.stream(
    thread["thread_id"],
    assistant_id="agent",
    input=None,
    stream_mode="updates",
    checkpoint_id=forked_config['checkpoint_id']
):
    if chunk.data:
        assisant_node = chunk.data.get('assistant', {}).get('messages', [])
        tool_node = chunk.data.get('tools', {}).get('messages', [])
        if assisant_node:
            print("-" * 20+"Assistant Node"+"-" * 20)
            print(assisant_node[-1])
        elif tool_node:
            print("-" * 20+"Tools Node"+"-" * 20)
            print(tool_node[-1])
```

Nodes can be executed parallel with a fan in and fan out structure

```
builder = StateGraph(State)

# Initialize each node with node_secret
builder.add_node("a", ReturnNodeValue("I'm A"))
builder.add_node("b", ReturnNodeValue("I'm B"))
builder.add_node("c", ReturnNodeValue("I'm C"))
builder.add_node("d", ReturnNodeValue("I'm D"))

# Flow
builder.add_edge(START, "a")
builder.add_edge("a", "b")
builder.add_edge("a", "c")
builder.add_edge("b", "d")
builder.add_edge("c", "d")
builder.add_edge("d", END)
graph = builder.compile()

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

In this graph there’s a fan out for b and c and a fan in to d

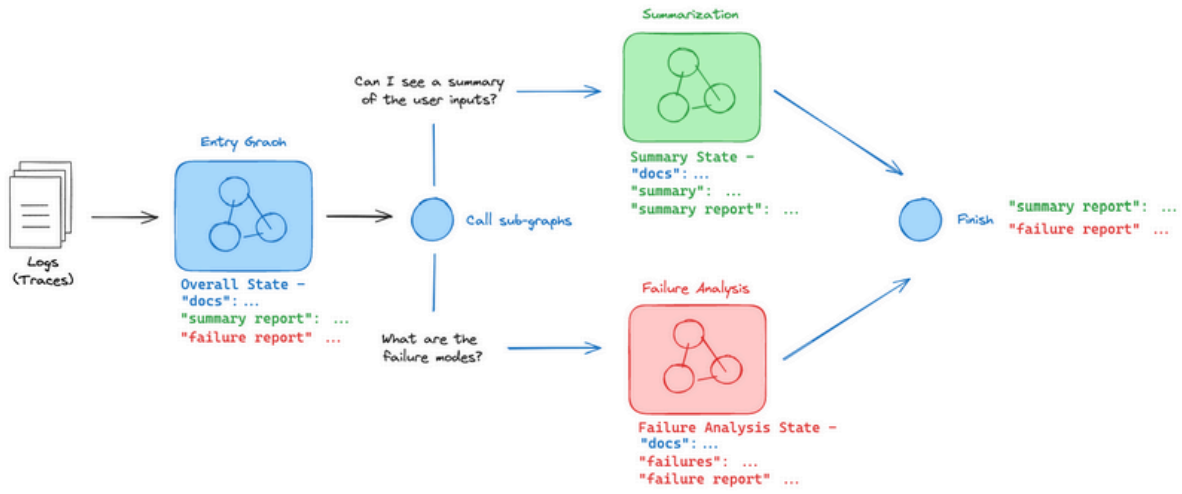
You can solve this by using either a reducer or a custom reducer:

```
import operator
from typing import Annotated

class State(TypedDict):
    # The operator.add reducer fn makes this append-only
    state: Annotated[list, operator.add]

def sorting_reducer(left, right):
    """ Combines and sorts the values in a list"""
    if not isinstance(left, list):
```

```
        left = [left]

    if not isinstance(right, list):
        right = [right]

    return sorted(left + right, reverse=False)

class State(TypedDict):
    # sorting_reducer will sort the values in state
    state: Annotated[list, sorting_reducer]
```

These are some examples of custom reducers

You can create subgraphs to abstract functions



Subgraphs have an input and output state:

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

```
# Failure Analysis Sub-graph
class FailureAnalysisState(TypedDict):
    cleaned_logs: List[Log]
    failures: List[Log]
    fa_summary: str
    processed_logs: List[str]

class FailureAnalysisOutputState(TypedDict):
    fa_summary: str
    processed_logs: List[str]
```

The output state helps when branching nodes fan in and have duplicate variables, so only necessary variables are returned

This graph cleans the logs and has subgraphs for failure analysis and question summarization, and they can use the cleaned_logs that is passed to them. As the output state may also include duplicate cleaned logs, we defined an output state previously so we don’t need a reducer for cleaned_logs

```
# Entry Graph
class EntryGraphState(TypedDict):
    raw_logs: List[Log]
    cleaned_logs: List[Log]
    fa_summary: str # This will only be generated in the FA sub-graph
    report: str # This will only be generated in the QS sub-graph
    processed_logs:  Annotated[List[int], add] # This will be generated in BOTH su

def clean_logs(state):
    # Get logs
    raw_logs = state["raw_logs"]
    # Data cleaning raw_logs -> docs
    cleaned_logs = raw_logs
    return {"cleaned_logs": cleaned_logs}

entry_builder = StateGraph(EntryGraphState)
```

```
entry_builder.add_node("clean_logs", clean_logs)
entry_builder.add_node("question_summarization", qs_builder.compile())
entry_builder.add_node("failure_analysis", fa_builder.compile())

entry_builder.add_edge(START, "clean_logs")
entry_builder.add_edge("clean_logs", "failure_analysis")
entry_builder.add_edge("clean_logs", "question_summarization")
entry_builder.add_edge("failure_analysis", END)
entry_builder.add_edge("question_summarization", END)

graph = entry_builder.compile()

from IPython.display import Image, display

# Setting xray to 1 will show the internal structure of the nested graph
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

Map reduce is a way to auto generate parallelization. We can define states by:

```
import operator
from typing import Annotated
from typing_extensions import TypedDict
from pydantic import BaseModel

class Subjects(BaseModel):
    subjects: list[str]

class BestJoke(BaseModel):
    id: int

class OverallState(TypedDict):
    topic: str
    subjects: list
```

```
    jokes: Annotated[list, operator.add]
    best_selected_joke: str
```

Generate topics generates a topic based on the prompt and creates it in the subject class:

```
def generate_topics(state: OverallState):
    prompt = subjects_prompt.format(topic=state["topic"])
    response = model.with_structured_output(Subjects).invoke(prompt)
    return {"subjects": response.subjects}
```

This is the map: Send as an api fans out per subject in joke and creates parallelized states

```
from langgraph.constants import Send
def continue_to_jokes(state: OverallState):
    return [Send("generate_joke", {"subject": s}) for s in state["subjects"]]

class JokeState(TypedDict):
    subject: str

class Joke(BaseModel):
    joke: str

def generate_joke(state: JokeState):
    prompt = joke_prompt.format(subject=state["subject"])
    response = model.with_structured_output(Joke).invoke(prompt)
    return {"jokes": [response.joke]}
```

Then the reduce takes in the jokes and picks the best one:

```
def best_joke(state: OverallState):
    jokes = "\n\n".join(state["jokes"])
    prompt = best_joke_prompt.format(topic=state["topic"], jokes=jokes)
    response = model.with_structured_output(BestJoke).invoke(prompt)
    return {"best_selected_joke": state["jokes"][response.id]}
```

Structured output is a powerful tool to define LLM output into a class structure:

```
response = model.with_structured_output(BestJoke).invoke(prompt)
```

This is an example of a Research Assistant graph:

This code uses pydantic to enforce types and create analysis, and creates a generate analysts state

```
from typing import List
from typing_extensions import TypedDict
from pydantic import BaseModel, Field

class Analyst(BaseModel):
    affiliation: str = Field(
        description="Primary affiliation of the analyst.",
    )
    name: str = Field(
        description="Name of the analyst."
    )
    role: str = Field(
        description="Role of the analyst in the context of the topic.",
    )
    description: str = Field(
        description="Description of the analyst focus, concerns, and motives.",
    )
```

```
    @property
    def persona(self) -> str:
        return f"Name: {self.name}\nRole: {self.role}\nAffiliation: {self.affiliation}\nD

class Perspectives(BaseModel):
    analysts: List[Analyst] = Field(
        description="Comprehensive list of analysts with their roles and affiliations."
    )

class GenerateAnalystsState(TypedDict):
    topic: str # Research topic
    max_analysts: int # Number of analysts
    human_analyst_feedback: str # Human feedback
    analysts: List[Analyst] # Analyst asking questions
```

This snippet has code for a prompt, creating analysts using an llm and enforcing structured output into perspective classes. There is an empty human feedback node which gets a conditional node for revisions where create analysts is rerun if any feedback is present. Else it continues:

```
from IPython.display import Image, display
from langgraph.graph import START, END, StateGraph
from langgraph.checkpoint.memory import MemorySaver
from langchain_core.messages import AIMessage, HumanMessage, SystemMes

analyst_instructions="""You are tasked with creating a set of AI analyst personas

1. First, review the research topic:
{topic}

2. Examine any editorial feedback that has been optionally provided to guide crea

{human_analyst_feedback}
```

```
3. Determine the most interesting themes based upon documents and / or feedba

4. Pick the top {max_analysts} themes.

5. Assign one analyst to each theme."""

def create_analysts(state: GenerateAnalystsState):

    """ Create analysts """

    topic=state['topic']
    max_analysts=state['max_analysts']
    human_analyst_feedback=state.get('human_analyst_feedback', '')

    # Enforce structured output
    structured_llm = llm.with_structured_output(Perspectives)

    # System message
    system_message = analyst_instructions.format(topic=topic,
                                                 human_analyst_feedback=human_analyst_fe
                                                 max_analysts=max_analysts)

    # Generate question
    analysts = structured_llm.invoke([SystemMessage(content=system_message)

    # Write the list of analysis to state
    return {"analysts": analysts.analysts}

def human_feedback(state: GenerateAnalystsState):
    """ No-op node that should be interrupted on """
    pass

def should_continue(state: GenerateAnalystsState):
    """ Return the next node to execute """

    # Check if human feedback
```

```
    human_analyst_feedback=state.get('human_analyst_feedback', None)
    if human_analyst_feedback:
        return "create_analysts"

    # Otherwise end
    return END

# Add nodes and edges
builder = StateGraph(GenerateAnalystsState)
builder.add_node("create_analysts", create_analysts)
builder.add_node("human_feedback", human_feedback)
builder.add_edge(START, "create_analysts")
builder.add_edge("create_analysts", "human_feedback")
builder.add_conditional_edges("human_feedback", should_continue, ["create_ana

# Compile
memory = MemorySaver()
graph = builder.compile(interrupt_before=['human_feedback'], checkpointer=me

# View
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

The analysts can be gotten and observed by doing this:

```
# Continue the graph execution
for event in graph.stream(None, thread, stream_mode="values"):
    # Review
    analysts = event.get('analysts', '')
    if analysts:
        for analyst in analysts:
            print(f"Name: {analyst.name}")
            print(f"Affiliation: {analyst.affiliation}")
            print(f"Role: {analyst.role}")
```

```
        print(f"Description: {analyst.description}")
        print("-" * 50)
```

A subgraph is created for the question answer cycle. In this a generate question function is made to generate questions based on a system prompt:

```
import operator
from typing import  Annotated
from langgraph.graph import MessagesState

class InterviewState(MessagesState):
    max_num_turns: int # Number turns of conversation
    context: Annotated[list, operator.add] # Source docs
    analyst: Analyst # Analyst asking questions
    interview: str # Interview transcript
    sections: list # Final key we duplicate in outer state for Send() API

class SearchQuery(BaseModel):
    search_query: str = Field(None, description="Search query for retrieval.")

question_instructions = """You are an analyst tasked with interviewing an expert

Your goal is boil down to interesting and specific insights related to your topic.

1. Interesting: Insights that people will find surprising or non-obvious.

2. Specific: Insights that avoid generalities and include specific examples from th

Here is your topic of focus and set of goals: {goals}

Begin by introducing yourself using a name that fits your persona, and then ask y

Continue to ask questions to drill down and refine your understanding of the topi
```

```
When you are satisfied with your understanding, complete the interview with: "Tl

Remember to stay in character throughout your response, reflecting the persona

def generate_question(state: InterviewState):
    """ Node to generate a question """

    # Get state
    analyst = state["analyst"]
    messages = state["messages"]

    # Generate question
    system_message = question_instructions.format(goals=analyst.persona)
    question = llm.invoke([SystemMessage(content=system_message)]+message

    # Write messages to state
    return {"messages": [question]}
```

A search query is made for both web and wiki search to simplify the complex prompt into a simple search question. Then after the search query is invoked the results are formatted:

```
from langchain_core.messages import get_buffer_string

# Search query writing
search_instructions = SystemMessage(content=f"""You will be given a conversa

Your goal is to generate a well-structured query for use in retrieval and / or web-

First, analyze the full conversation.

Pay particular attention to the final question posed by the analyst.

Convert this final question into a well-structured web search query""")
```

```
def search_web(state: InterviewState):

    """ Retrieve docs from web search """

    # Search query
    structured_llm = llm.with_structured_output(SearchQuery)
    search_query = structured_llm.invoke([search_instructions]+state['messages'])

    # Search
    search_docs = tavily_search.invoke(search_query.search_query)

     # Format
    formatted_search_docs = "\n\n---\n\n".join(
        [
            f'<Document href="{doc["url"]}"/>\n{doc["content"]}\n</Document>'
            for doc in search_docs
        ]
    )

    return {"context": [formatted_search_docs]}

def search_wikipedia(state: InterviewState):

    """ Retrieve docs from wikipedia """

    # Search query
    structured_llm = llm.with_structured_output(SearchQuery)
    search_query = structured_llm.invoke([search_instructions]+state['messages'])

    # Search
    search_docs = WikipediaLoader(query=search_query.search_query,
                                  load_max_docs=2).load()

     # Format
    formatted_search_docs = "\n\n---\n\n".join(
```

```
        [
            f'<Document source="{doc.metadata["source"]}" page="{doc.metadata.
            for doc in search_docs
        ]
    )

    return {"context": [formatted_search_docs]}
```

Answer is generated with the context and after max question is hit route_messages is hit as a conditional node and save interview or more questions are called

```
answer_instructions = """You are an expert being interviewed by an analyst.

Here is analyst area of focus: {goals}.

You goal is to answer a question posed by the interviewer.

To answer question, use this context:

{context}

When answering questions, follow these guidelines:

1. Use only the information provided in the context.

2. Do not introduce external information or make assumptions beyond what is ex

3. The context contain sources at the topic of each individual document.

4. Include these sources your answer next to any relevant statements. For examp

5. List your sources in order at the bottom of your answer. [1] Source 1, [2] Sourc
```

6. If the source is: <Document source="assistant/docs/llama3_1.pdf" page="7"/>

[1] assistant/docs/llama3_1.pdf, page 7

And skip the addition of the brackets as well as the Document source preamble i

```
def generate_answer(state: InterviewState):

    """ Node to answer a question """

    # Get state
    analyst = state["analyst"]
    messages = state["messages"]
    context = state["context"]

    # Answer question
    system_message = answer_instructions.format(goals=analyst.persona, contex
    answer = llm.invoke([SystemMessage(content=system_message)]+messages

    # Name the message as coming from the expert
    answer.name = "expert"

    # Append it to state
    return {"messages": [answer]}

def save_interview(state: InterviewState):

    """ Save interviews """

    # Get messages
    messages = state["messages"]

    # Convert interview to a string
    interview = get_buffer_string(messages)

    # Save to interviews key
```

```
    return {"interview": interview}

def route_messages(state: InterviewState,
                   name: str = "expert"):

    """ Route between question and answer """

    # Get messages
    messages = state["messages"]
    max_num_turns = state.get('max_num_turns',2)

    # Check the number of expert answers
    num_responses = len(
        [m for m in messages if isinstance(m, AIMessage) and m.name == name]
    )

    # End if expert has answered more than the max turns
    if num_responses >= max_num_turns:
        return 'save_interview'

    # This router is run after each question - answer pair
    # Get the last question asked to check if it signals the end of discussion
    last_question = messages[-2]

    if "Thank you so much for your help" in last_question.content:
        return 'save_interview'
    return "ask_question"
```

Then a technical writer node is called to format the answer based on the interview, context, and analyst data:

```
section_writer_instructions = """You are an expert technical writer.

Your task is to create a short, easily digestible section of a report based on a set
```

1. Analyze the content of the source documents:
- The name of each source document is at the start of the document, with the <D

2. Create a report structure using markdown formatting:
- Use ## for the section title
- Use ### for sub-section headers

3. Write the report following this structure:
a. Title (## header)
b. Summary (### header)
c. Sources (### header)

4. Make your title engaging based upon the focus area of the analyst:
{focus}

5. For the summary section:
- Set up summary with general background / context related to the focus area of
- Emphasize what is novel, interesting, or surprising about insights gathered from
- Create a numbered list of source documents, as you use them
- Do not mention the names of interviewers or experts
- Aim for approximately 400 words maximum
- Use numbered sources in your report (e.g., [1], [2]) based on information from s

6. In the Sources section:
- Include all sources used in your report
- Provide full links to relevant websites or specific document paths
- Separate each source by a newline. Use two spaces at the end of each line to c
- It will look like:

Sources
[1] Link or Document name
[2] Link or Document name

7. Be sure to combine sources. For example this is not correct:

```
[3] https://ai.meta.com/blog/meta-llama-3-1/
[4] https://ai.meta.com/blog/meta-llama-3-1/

There should be no redundant sources. It should simply be:

[3] https://ai.meta.com/blog/meta-llama-3-1/

8. Final review:
- Ensure the report follows the required structure
- Include no preamble before the title of the report
- Check that all guidelines have been followed"""

def write_section(state: InterviewState):

    """ Node to answer a question """

    # Get state
    interview = state["interview"]
    context = state["context"]
    analyst = state["analyst"]

    # Write section using either the gathered source docs from interview (context)
    system_message = section_writer_instructions.format(focus=analyst.descripti
    section = llm.invoke([SystemMessage(content=system_message)]+[HumanMe

    # Append it to state
    return {"sections": [section.content]}

# Add nodes and edges
interview_builder = StateGraph(InterviewState)
interview_builder.add_node("ask_question", generate_question)
interview_builder.add_node("search_web", search_web)
interview_builder.add_node("search_wikipedia", search_wikipedia)
interview_builder.add_node("answer_question", generate_answer)
interview_builder.add_node("save_interview", save_interview)
interview_builder.add_node("write_section", write_section)
```

```
# Flow
interview_builder.add_edge(START, "ask_question")
interview_builder.add_edge("ask_question", "search_web")
interview_builder.add_edge("ask_question", "search_wikipedia")
interview_builder.add_edge("search_web", "answer_question")
interview_builder.add_edge("search_wikipedia", "answer_question")
interview_builder.add_conditional_edges("answer_question", route_messages,['a
interview_builder.add_edge("save_interview", "write_section")
interview_builder.add_edge("write_section", END)

# Interview
memory = MemorySaver()
interview_graph = interview_builder.compile(checkpointer=memory).with_config

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

A final state is created to split into an intro and conclusion part of the report:

```
import operator
from typing import List, Annotated
from typing_extensions import TypedDict

class ResearchGraphState(TypedDict):
    topic: str # Research topic
    max_analysts: int # Number of analysts
    human_analyst_feedback: str # Human feedback
    analysts: List[Analyst] # Analyst asking questions
    sections: Annotated[list, operator.add] # Send() API key
    introduction: str # Introduction for the final report
    content: str # Content for the final report
    conclusion: str # Conclusion for the final report
    final_report: str # Final report
```

This is the map reduce step that uses the send api to create n analysts, uses them to conduct interviews, summarizes the data, then creates an intro, body, and conclusion for a final report:

```
from langgraph.constants import Send

def initiate_all_interviews(state: ResearchGraphState):
    """ This is the "map" step where we run each interview sub-graph using Send

    # Check if human feedback
    human_analyst_feedback=state.get('human_analyst_feedback')
    if human_analyst_feedback:
        # Return to create_analysts
        return "create_analysts"

    # Otherwise kick off interviews in parallel via Send() API
    else:
        topic = state["topic"]
        return [Send("conduct_interview", {"analyst": analyst,
                                           "messages": [HumanMessage(
                                               content=f"So you said you were writing an article on
                                           )
                                                   ]}) for analyst in state["analysts"]]

report_writer_instructions = """You are a technical writer creating a report on this

{topic}

You have a team of analysts. Each analyst has done two things:

1. They conducted an interview with an expert on a specific sub-topic.
2. They write up their finding into a memo.

Your task:
```

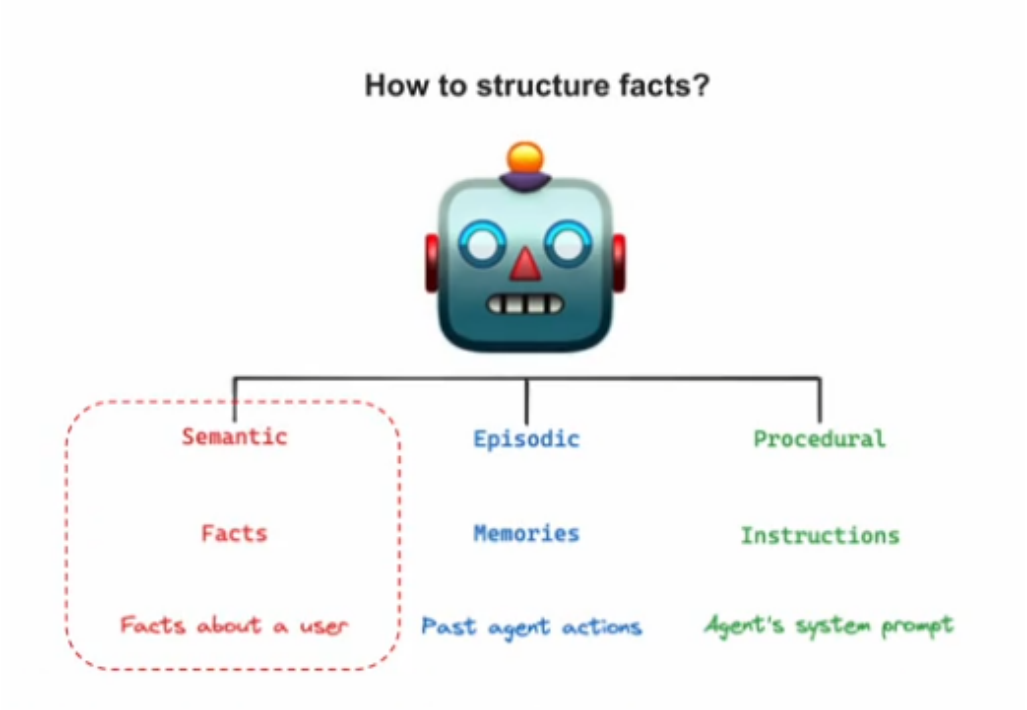
```
1. You will be given a collection of memos from your analysts.
2. Think carefully about the insights from each memo.
3. Consolidate these into a crisp overall summary that ties together the central id
4. Summarize the central points in each memo into a cohesive single narrative.

To format your report:

1. Use markdown formatting.
2. Include no pre-amble for the report.
3. Use no sub-heading.
4. Start your report with a single title header: ## Insights
5. Do not mention any analyst names in your report.
6. Preserve any citations in the memos, which will be annotated in brackets, for e
7. Create a final, consolidated list of sources and add to a Sources section with th
8. List your sources in order and do not repeat.

[1] Source 1
[2] Source 2

Here are the memos from your analysts to build your report from:

{context}"""

def write_report(state: ResearchGraphState):
    # Full set of sections
    sections = state["sections"]
    topic = state["topic"]

    # Concat all sections together
    formatted_str_sections = "\n\n".join([f"{section}" for section in sections])

    # Summarize the sections into a final report
    system_message = report_writer_instructions.format(topic=topic, context=forn
    report = llm.invoke([SystemMessage(content=system_message)]+[HumanMes
    return {"content": report.content}
```

```
intro_conclusion_instructions = """You are a technical writer finishing a report on

You will be given all of the sections of the report.

You job is to write a crisp and compelling introduction or conclusion section.

The user will instruct you whether to write the introduction or conclusion.

Include no pre-amble for either section.

Target around 100 words, crisply previewing (for introduction) or recapping (for c

Use markdown formatting.

For your introduction, create a compelling title and use the # header for the title.

For your introduction, use ## Introduction as the section header.

For your conclusion, use ## Conclusion as the section header.

Here are the sections to reflect on for writing: {formatted_str_sections}"""

def write_introduction(state: ResearchGraphState):
    # Full set of sections
    sections = state["sections"]
    topic = state["topic"]

    # Concat all sections together
    formatted_str_sections = "\n\n".join([f"{section}" for section in sections])

    # Summarize the sections into a final report

    instructions = intro_conclusion_instructions.format(topic=topic, formatted_str_
    intro = llm.invoke([instructions]+[HumanMessage(content=f"Write the report i
    return {"introduction": intro.content}
```

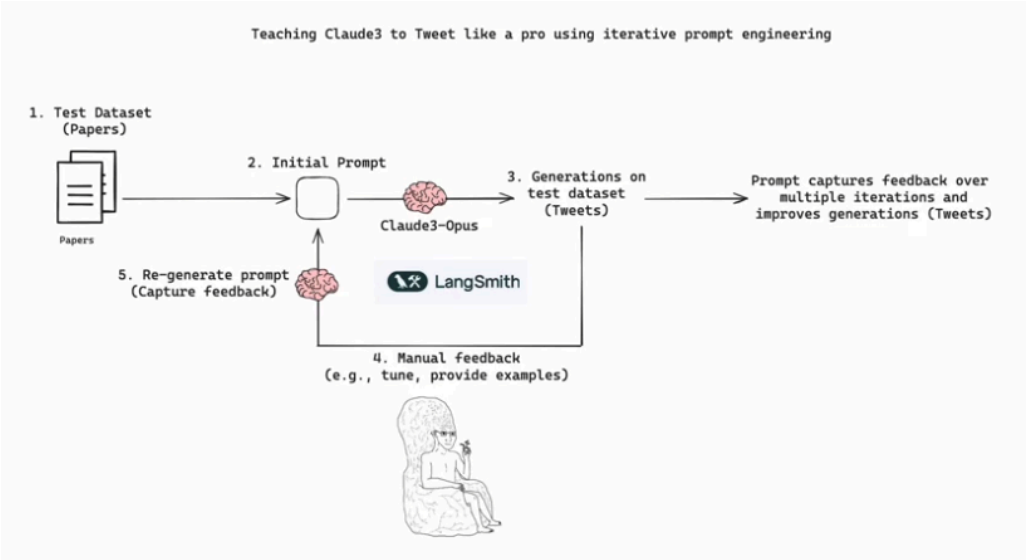
```
def write_conclusion(state: ResearchGraphState):
    # Full set of sections
    sections = state["sections"]
    topic = state["topic"]

    # Concat all sections together
    formatted_str_sections = "\n\n".join([f"{section}" for section in sections])

    # Summarize the sections into a final report

    instructions = intro_conclusion_instructions.format(topic=topic, formatted_str_
    conclusion = llm.invoke([instructions]+[HumanMessage(content=f"Write the r
    return {"conclusion": conclusion.content}

def finalize_report(state: ResearchGraphState):
    """ The is the "reduce" step where we gather all the sections, combine them, a
    # Save full final report
    content = state["content"]
    if content.startswith("## Insights"):
        content = content.strip("## Insights")
    if "## Sources" in content:
        try:
            content, sources = content.split("\n## Sources\n")
        except:
            sources = None
    else:
        sources = None

    final_report = state["introduction"] + "\n\n---\n\n" + content + "\n\n---\n\n" + s
    if sources is not None:
        final_report += "\n\n## Sources\n" + sources
    return {"final_report": final_report}

# Add nodes and edges
builder = StateGraph(ResearchGraphState)
```

```
builder.add_node("create_analysts", create_analysts)
builder.add_node("human_feedback", human_feedback)
builder.add_node("conduct_interview", interview_builder.compile())
builder.add_node("write_report",write_report)
builder.add_node("write_introduction",write_introduction)
builder.add_node("write_conclusion",write_conclusion)
builder.add_node("finalize_report",finalize_report)

# Logic
builder.add_edge(START, "create_analysts")
builder.add_edge("create_analysts", "human_feedback")
builder.add_conditional_edges("human_feedback", initiate_all_interviews, ["creat
builder.add_edge("conduct_interview", "write_report")
builder.add_edge("conduct_interview", "write_introduction")
builder.add_edge("conduct_interview", "write_conclusion")
builder.add_edge(["write_conclusion", "write_report", "write_introduction"], "fina
builder.add_edge("finalize_report", END)

# Compile
memory = MemorySaver()
graph = builder.compile(interrupt_before=['human_feedback'], checkpointer=me
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

Long term memory is important to store data between conversations, not just threads. This is a potential way to store that data.



AIs are good prompt engineers:



LangGraph Store is a way to store information across threads, which in total has a namespace for the object, then a list of keys → dicts in each object. These are

some functions for store:

```
import uuid
from langgraph.store.memory import InMemoryStore
in_memory_store = InMemoryStore()

# Namespace for the memory to save
user_id = "1"
namespace_for_memory = (user_id, "memories")

# Save a memory to namespace as key and value
key = str(uuid.uuid4())

# The value needs to be a dictionary
value = {"food_preference" : "I like pizza"}

# Save the memory
in_memory_store.put(namespace_for_memory, key, value)

# Search
memories = in_memory_store.search(namespace_for_memory)
type(memories)

# Metatdata
memories[0].dict()

# The key, value
print(memories[0].key, memories[0].value)

# Get the memory by namespace and key
memory = in_memory_store.get(namespace_for_memory, key)
memory.dict()
```

Below is a simple graph that has prompts for using and creating new memory. This memory is stored as a string which is rewritten every single time the llm is called:

```
from IPython.display import Image, display

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, MessagesState, START, END
from langgraph.store.base import BaseStore

from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.runnables.config import RunnableConfig

# Chatbot instruction
MODEL_SYSTEM_MESSAGE = """You are a helpful assistant with memory that pr
If you have memory for this user, use it to personalize your responses.
Here is the memory (it may be empty): {memory}"""

# Create new memory from the chat history and any existing memory
CREATE_MEMORY_INSTRUCTION = """You are collecting information about the

CURRENT USER INFORMATION:
{memory}

INSTRUCTIONS:
1. Review the chat history below carefully
2. Identify new information about the user, such as:
   - Personal details (name, location)
   - Preferences (likes, dislikes)
   - Interests and hobbies
   - Past experiences
   - Goals or future plans
3. Merge any new information with existing memory
4. Format the memory as a clear, bulleted list
5. If new information conflicts with existing memory, keep the most recent versio

Remember: Only include factual information directly stated by the user. Do not m
```

```
Based on the chat history below, please update the user information:"""

def call_model(state: MessagesState, config: RunnableConfig, store: BaseStore):

    """Load memory from the store and use it to personalize the chatbot's respons

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve memory from the store
    namespace = ("memory", user_id)
    key = "user_memory"
    existing_memory = store.get(namespace, key)

    # Extract the actual memory content if it exists and add a prefix
    if existing_memory:
        # Value is a dictionary with a memory key
        existing_memory_content = existing_memory.value.get('memory')
    else:
        existing_memory_content = "No existing memory found."

    # Format the memory in the system prompt
    system_msg = MODEL_SYSTEM_MESSAGE.format(memory=existing_memory_

    # Respond using memory as well as the chat history
    response = model.invoke([SystemMessage(content=system_msg)]+state["me

    return {"messages": response}

def write_memory(state: MessagesState, config: RunnableConfig, store: BaseSto

    """Reflect on the chat history and save a memory to the store."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]
```

```
    # Retrieve existing memory from the store
    namespace = ("memory", user_id)
    existing_memory = store.get(namespace, "user_memory")

    # Extract the memory
    if existing_memory:
        existing_memory_content = existing_memory.value.get('memory')
    else:
        existing_memory_content = "No existing memory found."

    # Format the memory in the system prompt
    system_msg = CREATE_MEMORY_INSTRUCTION.format(memory=existing_me
    new_memory = model.invoke([SystemMessage(content=system_msg)]+state[

    # Overwrite the existing memory in the store
    key = "user_memory"

    # Write value as a dictionary with a memory key
    store.put(namespace, key, {"memory": new_memory.content})

# Define the graph
builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_node("write_memory", write_memory)
builder.add_edge(START, "call_model")
builder.add_edge("call_model", "write_memory")
builder.add_edge("write_memory", END)

# Store for long-term (across-thread) memory
across_thread_memory = InMemoryStore()

# Checkpointer for short-term (within-thread) memory
within_thread_memory = MemorySaver()

# Compile the graph with the checkpointer fir and store
```

```
graph = builder.compile(checkpointer=within_thread_memory, store=across_thre

# View
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

There are two types of ways to store long term memory: profiles and collections:

- Profiles map data to a profile object
- Collections add data to a list of strings

Both currently use TrustCall to update schemas as complex data causes error in LLMs.

The code below is an example of using TrustCall to safely write memory

```
from IPython.display import Image, display

from langchain_core.messages import HumanMessage, SystemMessage
from langgraph.graph import StateGraph, MessagesState, START, END
from langchain_core.runnables.config import RunnableConfig
from langgraph.checkpoint.memory import MemorySaver
from langgraph.store.base import BaseStore

# Initialize the model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# Schema
class UserProfile(BaseModel):
    """ Profile of a user """
    user_name: str = Field(description="The user's preferred name")
    user_location: str = Field(description="The user's location")
    interests: list = Field(description="A list of the user's interests")

# Create the extractor
```

```
trustcall_extractor = create_extractor(
    model,
    tools=[UserProfile],
    tool_choice="UserProfile", # Enforces use of the UserProfile tool
)

# Chatbot instruction
MODEL_SYSTEM_MESSAGE = """You are a helpful assistant with memory that pr
If you have memory for this user, use it to personalize your responses.
Here is the memory (it may be empty): {memory}"""

# Extraction instruction
TRUSTCALL_INSTRUCTION = """Create or update the memory (JSON doc) to inc

def call_model(state: MessagesState, config: RunnableConfig, store: BaseStore):

    """Load memory from the store and use it to personalize the chatbot's respons

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve memory from the store
    namespace = ("memory", user_id)
    existing_memory = store.get(namespace, "user_memory")

    # Format the memories for the system prompt
    if existing_memory and existing_memory.value:
        memory_dict = existing_memory.value
        formatted_memory = (
            f"Name: {memory_dict.get('user_name', 'Unknown')}\n"
            f"Location: {memory_dict.get('user_location', 'Unknown')}\n"
            f"Interests: {', '.join(memory_dict.get('interests', []))}"
        )
    else:
        formatted_memory = None
```

```
    # Format the memory in the system prompt
    system_msg = MODEL_SYSTEM_MESSAGE.format(memory=formatted_memo

    # Respond using memory as well as the chat history
    response = model.invoke([SystemMessage(content=system_msg)]+state["me

    return {"messages": response}

def write_memory(state: MessagesState, config: RunnableConfig, store: BaseSto

    """Reflect on the chat history and save a memory to the store."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve existing memory from the store
    namespace = ("memory", user_id)
    existing_memory = store.get(namespace, "user_memory")

    # Get the profile as the value from the list, and convert it to a JSON doc
    existing_profile = {"UserProfile": existing_memory.value} if existing_memory e

    # Invoke the extractor
    result = trustcall_extractor.invoke({"messages": [SystemMessage(content=TR

    # Get the updated profile as a JSON object
    updated_profile = result["responses"][0].model_dump()

    # Save the updated profile
    key = "user_memory"
    store.put(namespace, key, updated_profile)

# Define the graph
builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_node("write_memory", write_memory)
```

```
builder.add_edge(START, "call_model")
builder.add_edge("call_model", "write_memory")
builder.add_edge("write_memory", END)

# Store for long-term (across-thread) memory
across_thread_memory = InMemoryStore()

# Checkpointer for short-term (within-thread) memory
within_thread_memory = MemorySaver()

# Compile the graph with the checkpointer fir and store
graph = builder.compile(checkpointer=within_thread_memory, store=across_thre

# View
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

The code below is an example of using TrustCall to safely create collections:

```
from IPython.display import Image, display

import uuid

from langgraph.graph import StateGraph, MessagesState, START, END
from langgraph.store.memory import InMemoryStore
from langchain_core.messages import merge_message_runs
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.runnables.config import RunnableConfig
from langgraph.checkpoint.memory import MemorySaver
from langgraph.store.base import BaseStore

# Initialize the model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# Memory schema
```

```
class Memory(BaseModel):
    content: str = Field(description="The main content of the memory. For exampl

# Create the Trustcall extractor
trustcall_extractor = create_extractor(
    model,
    tools=[Memory],
    tool_choice="Memory",
    # This allows the extractor to insert new memories
    enable_inserts=True,
)

# Chatbot instruction
MODEL_SYSTEM_MESSAGE = """You are a helpful chatbot. You are designed to

You have a long term memory which keeps track of information you learn about t

Current Memory (may include updated memories from this conversation):

{memory}"""

# Trustcall instruction
TRUSTCALL_INSTRUCTION = """Reflect on following interaction.

Use the provided tools to retain any necessary memories about the user.

Use parallel tool calling to handle updates and insertions simultaneously:"""

def call_model(state: MessagesState, config: RunnableConfig, store: BaseStore):

    """Load memories from the store and use them to personalize the chatbot's re

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve memory from the store
```

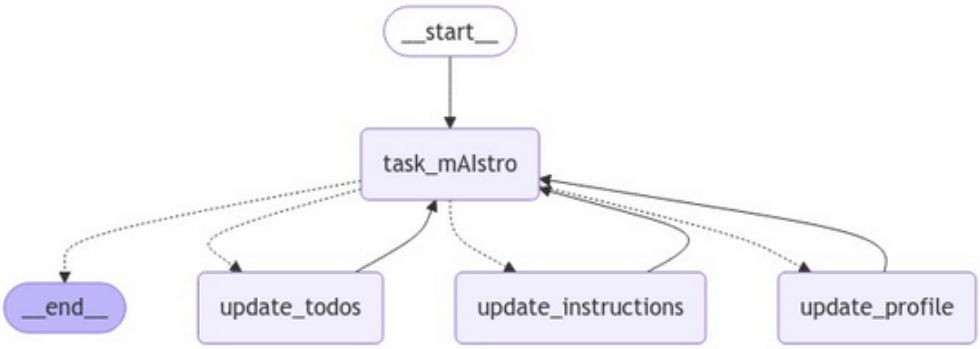
```
    namespace = ("memories", user_id)
    memories = store.search(namespace)

    # Format the memories for the system prompt
    info = "\n".join(f"- {mem.value['content']}" for mem in memories)
    system_msg = MODEL_SYSTEM_MESSAGE.format(memory=info)

    # Respond using memory as well as the chat history
    response = model.invoke([SystemMessage(content=system_msg)]+state["me

    return {"messages": response}

def write_memory(state: MessagesState, config: RunnableConfig, store: BaseSto

    """Reflect on the chat history and update the memory collection."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Define the namespace for the memories
    namespace = ("memories", user_id)

    # Retrieve the most recent memories for context
    existing_items = store.search(namespace)

    # Format the existing memories for the Trustcall extractor
    tool_name = "Memory"
    existing_memories = ([(existing_item.key, tool_name, existing_item.value)
                          for existing_item in existing_items]
                          if existing_items
                          else None
                        )

    # Merge the chat history and the instruction
    updated_messages=list(merge_message_runs(messages=[SystemMessage(c
```

```
    # Invoke the extractor
    result = trustcall_extractor.invoke({"messages": updated_messages,
                                         "existing": existing_memories})

    # Save the memories from Trustcall to the store
    for r, rmeta in zip(result["responses"], result["response_metadata"]):
        store.put(namespace,
                  rmeta.get("json_doc_id", str(uuid.uuid4())),
                  r.model_dump(mode="json"),
            )

# Define the graph
builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_node("write_memory", write_memory)
builder.add_edge(START, "call_model")
builder.add_edge("call_model", "write_memory")
builder.add_edge("write_memory", END)

# Store for long-term (across-thread) memory
across_thread_memory = InMemoryStore()

# Checkpointer for short-term (within-thread) memory
within_thread_memory = MemorySaver()

# Compile the graph with the checkpointer fir and store
graph = builder.compile(checkpointer=within_thread_memory, store=across_thre

# View
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

This is the reAct architecture agent to build that has long term memory.

- Update todos is a collection of todos

- Instructions is a collection of instructions
- Profile is a profile that stores user details



Another new concept is spy which can be attached to tool binded llms to see the tool call's data in more depth:

```
from trustcall import create_extractor
from langchain_openai import ChatOpenAI

# Inspect the tool calls made by Trustcall
class Spy:
    def __init__(self):
        self.called_tools = []

    def __call__(self, run):
        # Collect information about the tool calls made by the extractor.
        q = [run]
        while q:
            r = q.pop()
            if r.child_runs:
                q.extend(r.child_runs)
            if r.run_type == "chat_model":
                self.called_tools.append(
                    r.outputs["generations"][0][0]["message"]["kwargs"]["tool_calls"]
                )
```

```
# Initialize the spy
spy = Spy()

# Initialize the model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# Create the extractor
trustcall_extractor = create_extractor(
    model,
    tools=[Memory],
    tool_choice="Memory",
    enable_inserts=True,
)

# Add the spy as a listener
trustcall_extractor_see_all_tool_calls = trustcall_extractor.with_listeners(on_end=s
```

For example:

```
result = trustcall_extractor.invoke({"messages": [SystemMessage(content=instru

# Messages contain the tool calls
for m in result["messages"]:
    m.pretty_print()

spy.called_tools
```

```
[[{'name': 'PatchDoc',
   'args': {'json_doc_id': '0',
    'planned_edits': '1. Replace the existing content with the updated memory t
hat includes the new activities: going to Tartine for a croissant and thinking ab
out going back to Japan this winter.',
    'patches': [{'op': 'replace',
```

```
      'path': '/content',
      'value': 'Lance had a nice bike ride in San Francisco this morning. Afterwar
d, he went to Tartine and ate a croissant. He was also thinking about his trip to
Japan and going back this winter.'}]},
   'id': 'call_bF0w0hE4YZmGyDbuJVe1mh5H',
   'type': 'tool_call'},
  {'name': 'Memory',
   'args': {'content': 'Lance went to Tartine and ate a croissant. He was also thi
nking about his trip to Japan and going back this winter.'},
   'id': 'call_fQAxxRypV914Xev6nJ9VKw3X',
   'type': 'tool_call'}]]
```

This can be used to get details about tool calls to return back to the user:

```
def extract_tool_info(tool_calls, schema_name="Memory"):
    """Extract information from tool calls for both patches and new memories.

    Args:
        tool_calls: List of tool calls from the model
        schema_name: Name of the schema tool (e.g., "Memory", "ToDo", "Profile")
    """

    # Initialize list of changes
    changes = []

    for call_group in tool_calls:
        for call in call_group:
            if call['name'] == 'PatchDoc':
                changes.append({
                    'type': 'update',
                    'doc_id': call['args']['json_doc_id'],
                    'planned_edits': call['args']['planned_edits'],
                    'value': call['args']['patches'][0]['value']
                })
            elif call['name'] == schema_name:
                changes.append({
```

```
                'type': 'new',
                'value': call['args']
            })

    # Format results as a single string
    result_parts = []
    for change in changes:
        if change['type'] == 'update':
            result_parts.append(
                f"Document {change['doc_id']} updated:\n"
                f"Plan: {change['planned_edits']}\n"
                f"Added content: {change['value']}"
            )
        else:
            result_parts.append(
                f"New {schema_name} created:\n"
                f"Content: {change['value']}"
            )

    return "\n\n".join(result_parts)

# Inspect spy.called_tools to see exactly what happened during the extraction
schema_name = "Memory"
changes = extract_tool_info(spy.called_tools, schema_name)
print(changes)
```

```
Document 0 updated:
Plan: 1. Replace the existing content with the updated memory that includes th
e new activities: going to Tartine for a croissant and thinking about going back
to Japan this winter.
Added content: Lance had a nice bike ride in San Francisco this morning. Afte
rward, he went to Tartine and ate a croissant. He was also thinking about his tr
ip to Japan and going back this winter.

New Memory created:
```

```
Content: {'content': 'Lance went to Tartine and ate a croissant. He was also thi
nking about his trip to Japan and going back this winter.'}
```

In this model the task of updating todos, profile, or instructions needs to be picked. A trick used in this is by making a memory tool where a selection of three tools can be picked:

```
from typing import TypedDict, Literal

# Update memory tool
class UpdateMemory(TypedDict):
    """ Decision on what memory type to update """
    update_type: Literal['user', 'todo', 'instructions']
```

And then this makes it so the llm has to pick one of the options or null:

```
# Chatbot instruction for choosing what to update and what tools to call
MODEL_SYSTEM_MESSAGE = """You are a helpful chatbot.

You are designed to be a companion to a user, helping them keep track of their T

You have a long term memory which keeps track of three things:
1. The user's profile (general information about them)
2. The user's ToDo list
3. General instructions for updating the ToDo list

Here is the current User Profile (may be empty if no information has been collect
<user_profile>
{user_profile}
</user_profile>

Here is the current ToDo List (may be empty if no tasks have been added yet):
<todo>
```

```
{todo}
</todo>

Here are the current user-specified preferences for updating the ToDo list (may b
<instructions>
{instructions}
</instructions>

Here are your instructions for reasoning about the user's messages:

1. Reason carefully about the user's messages as presented below.

2. Decide whether any of the your long-term memory should be updated:
- If personal information was provided about the user, update the user's profile b
- If tasks are mentioned, update the ToDo list by calling UpdateMemory tool with
- If the user has specified preferences for how to update the ToDo list, update th

3. Tell the user that you have updated your memory, if appropriate:
- Do not tell the user you have updated the user's profile
- Tell the user them when you update the todo list
- Do not tell the user that you have updated instructions

4. Err on the side of updating the todo list. No need to ask for explicit permission

5. Respond naturally to user user after a tool call was made to save memories, or

# Node definitions
def task_mAIstro(state: MessagesState, config: RunnableConfig, store: BaseStor

    """Load memories from the store and use them to personalize the chatbot's re

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve profile memory from the store
    namespace = ("profile", user_id)
```

```
    memories = store.search(namespace)
    if memories:
        user_profile = memories[0].value
    else:
        user_profile = None

    # Retrieve task memory from the store
    namespace = ("todo", user_id)
    memories = store.search(namespace)
    todo = "\n".join(f"{mem.value}" for mem in memories)

    # Retrieve custom instructions
    namespace = ("instructions", user_id)
    memories = store.search(namespace)
    if memories:
        instructions = memories[0].value
    else:
        instructions = ""

    system_msg = MODEL_SYSTEM_MESSAGE.format(user_profile=user_profile, t

    # Respond using memory as well as the chat history
    response = model.bind_tools([UpdateMemory], parallel_tool_calls=False).invok

    return {"messages": [response]}
```

As no tool_choice is used in the .bind_tools, a tool does not necessarily have to be called.

The entire code is below and it uses spy and conditional routing like discussed before:

```
import uuid
from IPython.display import Image, display
```

```python
from datetime import datetime
from trustcall import create_extractor
from typing import Optional
from pydantic import BaseModel, Field

from langchain_core.runnables import RunnableConfig
from langchain_core.messages import merge_message_runs, HumanMessage, S

from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, MessagesState, END, START
from langgraph.store.base import BaseStore
from langgraph.store.memory import InMemoryStore

from langchain_openai import ChatOpenAI

# Initialize the model
model = ChatOpenAI(model="gpt-4o", temperature=0)

# User profile schema
class Profile(BaseModel):
    """This is the profile of the user you are chatting with"""
    name: Optional[str] = Field(description="The user's name", default=None)
    location: Optional[str] = Field(description="The user's location", default=None
    job: Optional[str] = Field(description="The user's job", default=None)
    connections: list[str] = Field(
        description="Personal connection of the user, such as family members, frie
        default_factory=list
    )
    interests: list[str] = Field(
        description="Interests that the user has",
        default_factory=list
    )

# ToDo schema
class ToDo(BaseModel):
    task: str = Field(description="The task to be completed.")
```

```
    time_to_complete: Optional[int] = Field(description="Estimated time to comple
    deadline: Optional[datetime] = Field(
        description="When the task needs to be completed by (if applicable)",
        default=None
    )
    solutions: list[str] = Field(
        description="List of specific, actionable solutions (e.g., specific ideas, servi
        min_items=1,
        default_factory=list
    )
    status: Literal["not started", "in progress", "done", "archived"] = Field(
        description="Current status of the task",
        default="not started"
    )

# Create the Trustcall extractor for updating the user profile
profile_extractor = create_extractor(
    model,
    tools=[Profile],
    tool_choice="Profile",
)

# Chatbot instruction for choosing what to update and what tools to call
MODEL_SYSTEM_MESSAGE = """You are a helpful chatbot.

You are designed to be a companion to a user, helping them keep track of their T

You have a long term memory which keeps track of three things:
1. The user's profile (general information about them)
2. The user's ToDo list
3. General instructions for updating the ToDo list

Here is the current User Profile (may be empty if no information has been collect
<user_profile>
{user_profile}
</user_profile>
```

```
Here is the current ToDo List (may be empty if no tasks have been added yet):
<todo>
{todo}
</todo>

Here are the current user-specified preferences for updating the ToDo list (may b
<instructions>
{instructions}
</instructions>

Here are your instructions for reasoning about the user's messages:

1. Reason carefully about the user's messages as presented below.

2. Decide whether any of the your long-term memory should be updated:
- If personal information was provided about the user, update the user's profile b
- If tasks are mentioned, update the ToDo list by calling UpdateMemory tool with
- If the user has specified preferences for how to update the ToDo list, update th

3. Tell the user that you have updated your memory, if appropriate:
- Do not tell the user you have updated the user's profile
- Tell the user them when you update the todo list
- Do not tell the user that you have updated instructions

4. Err on the side of updating the todo list. No need to ask for explicit permission

5. Respond naturally to user user after a tool call was made to save memories, or

# Trustcall instruction
TRUSTCALL_INSTRUCTION = """Reflect on following interaction.

Use the provided tools to retain any necessary memories about the user.

Use parallel tool calling to handle updates and insertions simultaneously.
```

```
System Time: {time}"""

# Instructions for updating the ToDo list
CREATE_INSTRUCTIONS = """Reflect on the following interaction.

Based on this interaction, update your instructions for how to update ToDo list ite

Use any feedback from the user to update how they like to have items added, etc

Your current instructions are:

<current_instructions>
{current_instructions}
</current_instructions>"""

# Node definitions
def task_mAIstro(state: MessagesState, config: RunnableConfig, store: BaseStor

    """Load memories from the store and use them to personalize the chatbot's re

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Retrieve profile memory from the store
    namespace = ("profile", user_id)
    memories = store.search(namespace)
    if memories:
        user_profile = memories[0].value
    else:
        user_profile = None

    # Retrieve task memory from the store
    namespace = ("todo", user_id)
    memories = store.search(namespace)
    todo = "\n".join(f"{mem.value}" for mem in memories)
```

```
    # Retrieve custom instructions
    namespace = ("instructions", user_id)
    memories = store.search(namespace)
    if memories:
        instructions = memories[0].value
    else:
        instructions = ""

    system_msg = MODEL_SYSTEM_MESSAGE.format(user_profile=user_profile, t

    # Respond using memory as well as the chat history
    response = model.bind_tools([UpdateMemory], parallel_tool_calls=False).invok

    return {"messages": [response]}

def update_profile(state: MessagesState, config: RunnableConfig, store: BaseSto

    """Reflect on the chat history and update the memory collection."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Define the namespace for the memories
    namespace = ("profile", user_id)

    # Retrieve the most recent memories for context
    existing_items = store.search(namespace)

    # Format the existing memories for the Trustcall extractor
    tool_name = "Profile"
    existing_memories = ([(existing_item.key, tool_name, existing_item.value)
                          for existing_item in existing_items]
                          if existing_items
                          else None
                        )
```

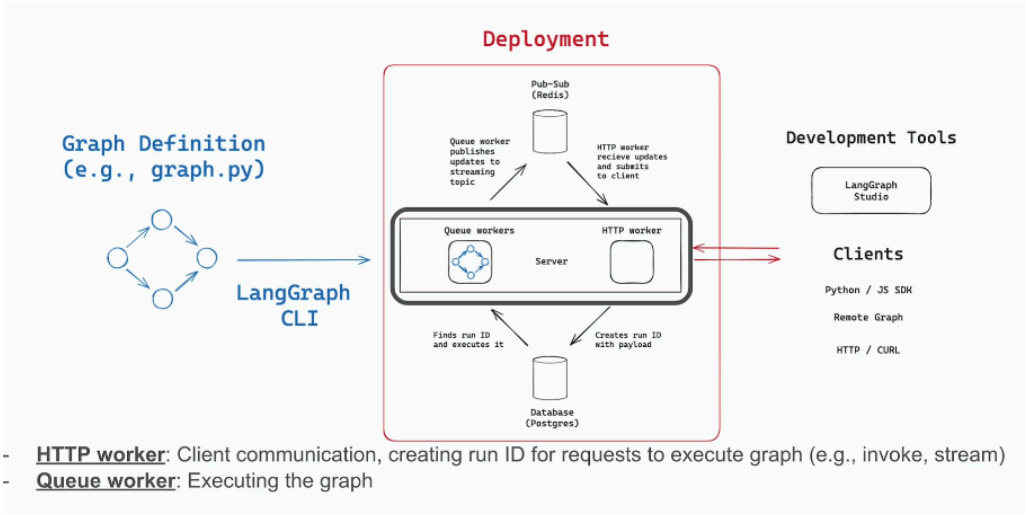
```
    # Merge the chat history and the instruction
    TRUSTCALL_INSTRUCTION_FORMATTED=TRUSTCALL_INSTRUCTION.format
    updated_messages=list(merge_message_runs(messages=[SystemMessage(c

    # Invoke the extractor
    result = profile_extractor.invoke({"messages": updated_messages,
                                       "existing": existing_memories})

    # Save the memories from Trustcall to the store
    for r, rmeta in zip(result["responses"], result["response_metadata"]):
        store.put(namespace,
                  rmeta.get("json_doc_id", str(uuid.uuid4())),
                  r.model_dump(mode="json"),
            )
    tool_calls = state['messages'][-1].tool_calls
    return {"messages": [{"role": "tool", "content": "updated profile", "tool_call_id"

def update_todos(state: MessagesState, config: RunnableConfig, store: BaseSto

    """Reflect on the chat history and update the memory collection."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    # Define the namespace for the memories
    namespace = ("todo", user_id)

    # Retrieve the most recent memories for context
    existing_items = store.search(namespace)

    # Format the existing memories for the Trustcall extractor
    tool_name = "ToDo"
    existing_memories = ([(existing_item.key, tool_name, existing_item.value)
                          for existing_item in existing_items]
                          if existing_items
                          else None
```

```
                )

    # Merge the chat history and the instruction
    TRUSTCALL_INSTRUCTION_FORMATTED=TRUSTCALL_INSTRUCTION.format
    updated_messages=list(merge_message_runs(messages=[SystemMessage(c

    # Initialize the spy for visibility into the tool calls made by Trustcall
    spy = Spy()

    # Create the Trustcall extractor for updating the ToDo list
    todo_extractor = create_extractor(
    model,
    tools=[ToDo],
    tool_choice=tool_name,
    enable_inserts=True
    ).with_listeners(on_end=spy)

    # Invoke the extractor
    result = todo_extractor.invoke({"messages": updated_messages,
                                    "existing": existing_memories})

    # Save the memories from Trustcall to the store
    for r, rmeta in zip(result["responses"], result["response_metadata"]):
        store.put(namespace,
                  rmeta.get("json_doc_id", str(uuid.uuid4())),
                  r.model_dump(mode="json"),
            )

    # Respond to the tool call made in task_mAIstro, confirming the update
    tool_calls = state['messages'][-1].tool_calls

    # Extract the changes made by Trustcall and add the the ToolMessage returne
    todo_update_msg = extract_tool_info(spy.called_tools, tool_name)
    return {"messages": [{"role": "tool", "content": todo_update_msg, "tool_call_id

def update_instructions(state: MessagesState, config: RunnableConfig, store: Ba
```

```
    """Reflect on the chat history and update the memory collection."""

    # Get the user ID from the config
    user_id = config["configurable"]["user_id"]

    namespace = ("instructions", user_id)

    existing_memory = store.get(namespace, "user_instructions")

    # Format the memory in the system prompt
    system_msg = CREATE_INSTRUCTIONS.format(current_instructions=existing_
    new_memory = model.invoke([SystemMessage(content=system_msg)]+state[

    # Overwrite the existing memory in the store
    key = "user_instructions"
    store.put(namespace, key, {"memory": new_memory.content})
    tool_calls = state['messages'][-1].tool_calls
    return {"messages": [{"role": "tool", "content": "updated instructions", "tool_ca

# Conditional edge
def route_message(state: MessagesState, config: RunnableConfig, store: BaseSt

    """Reflect on the memories and chat history to decide whether to update the m
    message = state['messages'][-1]
    if len(message.tool_calls) ==0:
        return END
    else:
        tool_call = message.tool_calls[0]
        if tool_call['args']['update_type'] == "user":
            return "update_profile"
        elif tool_call['args']['update_type'] == "todo":
            return "update_todos"
        elif tool_call['args']['update_type'] == "instructions":
            return "update_instructions"
        else:
```

```
            raise ValueError

# Create the graph + all nodes
builder = StateGraph(MessagesState)

# Define the flow of the memory extraction process
builder.add_node(task_mAIstro)
builder.add_node(update_todos)
builder.add_node(update_profile)
builder.add_node(update_instructions)
builder.add_edge(START, "task_mAIstro")
builder.add_conditional_edges("task_mAIstro", route_message)
builder.add_edge("update_todos", "task_mAIstro")
builder.add_edge("update_profile", "task_mAIstro")
builder.add_edge("update_instructions", "task_mAIstro")

# Store for long-term (across-thread) memory
across_thread_memory = InMemoryStore()

# Checkpointer for short-term (within-thread) memory
within_thread_memory = MemorySaver()

# We compile the graph with the checkpointer and store
graph = builder.compile(checkpointer=within_thread_memory, store=across_thre

# View
display(Image(graph.get_graph(xray=1).draw_mermaid_png()))
```

Deployment works where the is inside a deployment container where the client ge5ts a run, which goes to postgres, which gets a run id to execute, and publishes this in a queue, which gets sent back to the client:



- **HTTP worker**: Client communication, creating run ID for requests to execute graph (e.g., invoke, stream)
- **Queue worker**: Executing the graph

To create a langgraph container we need:

- A LangGraph API Config file
- The graphs that exist
- Requirements.txt
- A .env or docker-compose.yml

Below is an example of a langgraph api config file:

```
{
   "dockerfile_lines": [],
   "graphs": {
     "task_maistro": "./task_maistro.py:graph"
   },
   "python_version": "3.11",
   "dependencies": [
     "."
   ]
}
```

After we have a redis and postgresql file, we need to get both of their uri's and use:

```
docker run \
    --env-file .env \
    -p 8123:8000 \
    -e REDIS_URI="foo" \
    -e DATABASE_URI="bar" \
    -e LANGSMITH_API_KEY="baz" \
    my-image
```

We can also use docker-compose.yml where we create three containers for redis, postgres, and the langgraph api

```
langgraph build my-image
```

Then we fill out the docker-compose.yml:

```
volumes:
  langgraph-data:
    driver: local
services:
  langgraph-redis:
    image: redis:6
    healthcheck:
      test: redis-cli ping
      interval: 5s
      timeout: 1s
      retries: 5
    ports:
      - "6379:6379"
  langgraph-postgres:
    image: postgres:16
    ports:
```

```
    - "5432:5432"
  environment:
    POSTGRES_DB: postgres
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  volumes:
    - langgraph-data:/var/lib/postgresql/data
  healthcheck:
    test: pg_isready -U postgres
    start_period: 10s
    timeout: 1s
    retries: 5
    interval: 5s
langgraph-api:
  image: "my-image"
  ports:
    - "8123:8000"
  depends_on:
    langgraph-redis:
      condition: service_healthy
    langgraph-postgres:
      condition: service_healthy
  environment:
    REDIS_URI: redis://langgraph-redis:6379
    OPENAI_API_KEY: "your_openai_api_key"
    LANGSMITH_API_KEY: "your_langchain_api_key"
    POSTGRES_URI: postgres://postgres:postgres@langgraph-postgres:5432/po
```

Fill this out with your details and then run

```
cd module-6/deployment
docker compose up
```

You can connect to the deployment using the SDK and use functions like before:

```
from langgraph_sdk import get_client

# Connect via SDK
url_for_cli_deployment = "http://localhost:8123"
client = get_client(url=url_for_cli_deployment)
```

Remote graph is an easy way to work with this too:

```
from langgraph.pregel.remote import RemoteGraph
from langchain_core.messages import convert_to_messages
from langchain_core.messages import HumanMessage, SystemMessage

# Connect via remote graph
url = "http://localhost:8123"
graph_name = "task_maistro"
remote_graph = RemoteGraph(graph_name, url=url)
```

Runs are a single execution of your graph, you can do many things with runs:

```
# Create a thread
thread = await client.threads.create()
thread

# Check any existing runs on a thread
thread = await client.threads.create()
runs = await client.runs.list(thread["thread_id"])
print(runs)

# Do a non-blocking run, the print will tell the status of the run and it will be store
user_input = "Add a ToDo to finish booking travel to Hong Kong by end of next w
config = {"configurable": {"user_id": "Test"}}
graph_name = "task_maistro"
run = await client.runs.create(thread["thread_id"], graph_name, input={"message
print(await client.runs.get(thread["thread_id"], run["run_id"]))
```

```
# Wait until the run completes before continuing
# Start the run:
run = await client.runs.create(assistant_id="asst_123", input={"prompt": "Hello"})
# Wait synchronously for its completion:
result = await client.runs.join(run_id=run.id)
print("Final output:", result.output)
```

Each time a client makes a streaming request:

1. The HTTP worker generates a unique run ID
2. The Queue worker begins work on the run
3. During execution, the Queue worker publishes update to Redis
4. The HTTP worker subscribes to updates from Redis for ths run, and returns them to the client

```
user_input = "What ToDo should I focus on first."
async for chunk in client.runs.stream(thread["thread_id"],
                                      graph_name,
                                      input={"messages": [HumanMessage(content=user_input
                                      config=config,
                                      stream_mode="messages-tuple"):

    if chunk.event == "messages":
        print("".join(data_item['content'] for data_item in chunk.data if 'content' in da
```

Other functions with thread, human in loop, and state are:

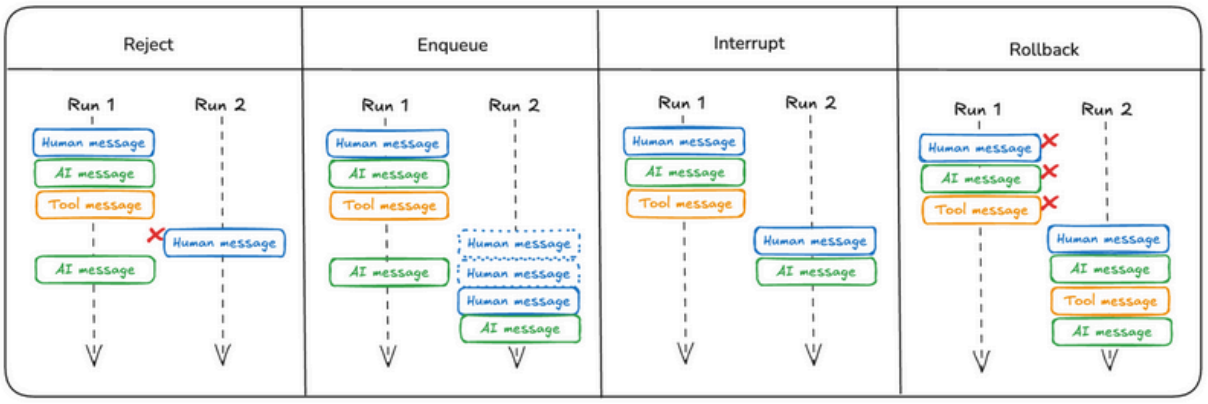
```
# Copy the thread
copied_thread = await client.threads.copy(thread['thread_id'])

# Check the state of the copied thread
```

```
copied_thread_state = await client.threads.get_state(copied_thread['thread_id'])
for m in convert_to_messages(copied_thread_state['values']['messages']):
    m.pretty_print()

# Get the history of the thread
states = await client.threads.get_history(thread['thread_id'])

# Pick a state update to fork
to_fork = states[-2]
to_fork['values']

to_fork['values']['messages'][0]['id']
to_fork['next']
to_fork['checkpoint_id']

forked_input = {"messages": HumanMessage(content="Give me a summary of a
                                id=to_fork['values']['messages'][0]['id'])}

# Update the state, creating a new checkpoint in the thread
forked_config = await client.threads.update_state(
    thread["thread_id"],
    forked_input,
    checkpoint_id=to_fork['checkpoint_id']
)

# Run the graph from the new checkpoint in the thread
async for chunk in client.runs.stream(thread["thread_id"],
                                graph_name,
                                input=None,
                                config=config,
                                checkpoint_id=forked_config['checkpoint_id'],
                                stream_mode="messages-tuple"):

    if chunk.event == "messages":
        print("".join(data_item['content'] for data_item in chunk.data if 'content' in da
```

```
# Search items
items = await client.store.search_items(
    ("todo", "general", "Test"),
    limit=5,
    offset=0
)
items['items']

# Add items
from uuid import uuid4
await client.store.put_item(
    ("testing", "Test"),
    key=str(uuid4()),
    value={"todo": "Test SDK put_item"},
)

items = await client.store.search_items(
    ("testing", "Test"),
    limit=5,
    offset=0
)
items['items']

# Delete items
[item['key'] for item in items['items']]

await client.store.delete_item(
       ("testing", "Test"),
        key='3de441ba-8c79-4beb-8f52-00e4dcba68d4',
    )

items = await client.store.search_items(
    ("testing", "Test"),
    limit=5,
    offset=0
```

```
)
items['items']
```

Double texting is an important use case to handle in a real world scenario:




```
# REJECT
import httpx
from langchain_core.messages import HumanMessage

# Create a thread
thread = await client.threads.create()

# Create to dos
user_input_1 = "Add a ToDo to follow-up with DI Repairs."
user_input_2 = "Add a ToDo to mount dresser to the wall."
config = {"configurable": {"user_id": "Test-Double-Texting"}}
graph_name = "task_maistro"

run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_1)]},
```

```
    config=config,
)
try:
    await client.runs.create(
        thread["thread_id"],
        graph_name,
        input={"messages": [HumanMessage(content=user_input_2)]},
        config=config,
        multitask_strategy="reject",
    )
except httpx.HTTPStatusError as e:
    print("Failed to start concurrent run", e)

'''
================================ Human Message ==============

Add a ToDo to follow-up with DI Repairs.
================================== Ai Message ==============
Tool Calls:
  UpdateMemory (call_6xqHubCPNufS0bg4tbUxC0FU)
 Call ID: call_6xqHubCPNufS0bg4tbUxC0FU
  Args:
    update_type: todo
================================= Tool Message ==============

New ToDo created:
Content: {'task': 'Follow-up with DI Repairs', 'time_to_complete': 30, 'deadline': N
================================== Ai Message ==============

I've added a task to follow-up with DI Repairs to your ToDo list. If there's anythin
'''
```

```
# Enqueue
# Create a new thread
```

```
thread = await client.threads.create()

# Create new ToDos
user_input_1 = "Send Erik his t-shirt gift this weekend."
user_input_2 = "Get cash and pay nanny for 2 weeks. Do this by Friday."
config = {"configurable": {"user_id": "Test-Double-Texting"}}
graph_name = "task_maistro"

first_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_1)]},
    config=config,
)

second_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_2)]},
    config=config,
    multitask_strategy="enqueue",
)

# Wait until the second run completes
await client.runs.join(thread["thread_id"], second_run["run_id"])

# Get the state of the thread
state = await client.threads.get_state(thread["thread_id"])
for m in convert_to_messages(state["values"]["messages"]):
    m.pretty_print()

'''
================================ Human Message =============

Send Erik his t-shirt gift this weekend.
================================== Ai Message ==============
```

```
Tool Calls:
  UpdateMemory (call_svTeXPmWGTLY8aQ8EifjwHAa)
 Call ID: call_svTeXPmWGTLY8aQ8EifjwHAa
  Args:
    update_type: todo
================================== Tool Message ==============

New ToDo created:
Content: {'task': 'Send Erik his t-shirt gift', 'time_to_complete': 30, 'deadline': '20
================================== Ai Message ==============

I've updated your ToDo list to send Erik his t-shirt gift this weekend. If there's any
================================ Human Message ==============

Get cash and pay nanny for 2 weeks. Do this by Friday.
================================== Ai Message ==============
Tool Calls:
  UpdateMemory (call_Cq0Tfn6yqccHH8n0DOucz5OQ)
 Call ID: call_Cq0Tfn6yqccHH8n0DOucz5OQ
  Args:
    update_type: todo
================================== Tool Message ==============

New ToDo created:
Content: {'task': 'Get cash and pay nanny for 2 weeks', 'time_to_complete': 15, 'd

Document af1fe011-f3c5-4c1c-b98b-181869bc2944 updated:
Plan: Update the deadline for sending Erik his t-shirt gift to this weekend, which i
Added content: 2024-11-17T23:59:00
================================== Ai Message ==============

I've updated your ToDo list to ensure you get cash and pay the nanny for 2 week
'''
```

```
# Interrupt
import asyncio

# Create a new thread
thread = await client.threads.create()

# Create new ToDos
user_input_1 = "Give me a summary of my ToDos due tomrrow."
user_input_2 = "Never mind, create a ToDo to Order Ham for Thanksgiving by ne
config = {"configurable": {"user_id": "Test-Double-Texting"}}
graph_name = "task_maistro"

interrupted_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_1)]},
    config=config,
)

# Wait for some of run 1 to complete so that we can see it in the thread
await asyncio.sleep(1)

second_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_2)]},
    config=config,
    multitask_strategy="interrupt",
)

# Wait until the second run completes
await client.runs.join(thread["thread_id"], second_run["run_id"])

# Get the state of the thread
state = await client.threads.get_state(thread["thread_id"])
```

```
for m in convert_to_messages(state["values"]["messages"]):
    m.pretty_print()

'''
================================ Human Message =============

Give me a summary of my ToDos due tomrrow.
================================ Human Message =============

Never mind, create a ToDo to Order Ham for Thanksgiving by next Friday.
================================== Ai Message ==============
Tool Calls:
  UpdateMemory (call_Rk80tTSJzik2oY44tyUWk8FM)
 Call ID: call_Rk80tTSJzik2oY44tyUWk8FM
  Args:
    update_type: todo
================================= Tool Message =============

New ToDo created:
Content: {'task': 'Order Ham for Thanksgiving', 'time_to_complete': 30, 'deadline'
================================== Ai Message ==============

I've added the task "Order Ham for Thanksgiving" to your ToDo list with a deadli
'''
```

```
# Rollback
# Create a new thread
thread = await client.threads.create()

# Create new ToDos
user_input_1 = "Add a ToDo to call to make appointment at Yoga."
user_input_2 = "Actually, add a ToDo to drop by Yoga in person on Sunday."
config = {"configurable": {"user_id": "Test-Double-Texting"}}
graph_name = "task_maistro"
```

```
rolled_back_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_1)]},
    config=config,
)

second_run = await client.runs.create(
    thread["thread_id"],
    graph_name,
    input={"messages": [HumanMessage(content=user_input_2)]},
    config=config,
    multitask_strategy="rollback",
)

# Wait until the second run completes
await client.runs.join(thread["thread_id"], second_run["run_id"])

# Get the state of the thread
state = await client.threads.get_state(thread["thread_id"])
for m in convert_to_messages(state["values"]["messages"]):
    m.pretty_print()

'''
# Confirm that the original run was deleted
try:
    await client.runs.get(thread["thread_id"], rolled_back_run["run_id"])
except httpx.HTTPStatusError as _:
    print("Original run was correctly deleted")
'''
```

This is the contents of the task_mAIstro config file:

```
import os
from dataclasses import dataclass, field, fields
from typing import Any, Optional

from langchain_core.runnables import RunnableConfig
from typing_extensions import Annotated
from dataclasses import dataclass


@dataclass(kw_only=True)
class Configuration:
    """The configurable fields for the chatbot."""

    user_id: str = "default-user"
    todo_category: str = "general"
    task_maistro_role: str = (
        "You are a helpful task management assistant. You help you create, organiz
    )

    @classmethod
    def from_runnable_config(
        cls, config: Optional[RunnableConfig] = None
    ) -> "Configuration":
        """Create a Configuration instance from a RunnableConfig."""
        configurable = (
            config["configurable"]
            if config and "configurable" in config
            else {}
        )
        values: dict[str, Any] = {
            f.name: os.environ.get(f.name.upper(), configurable.get(f.name))
            for f in fields(cls)
            if f.init
```

```
        }
        return cls(**{k: v for k, v in values.items() if v})
```

You can import the assistant by:

```
from langgraph_sdk import get_client
url_for_cli_deployment = "http://localhost:8123"
client = get_client(url=url_for_cli_deployment)
```

You can create assistants by:

```
personal_assistant = await client.assistants.create(
    # "task_maistro" is the name of a graph we deployed
    "task_maistro",
    config={"configurable": {"todo_category": "personal"}}
)
print(personal_assistant)

task_maistro_role = """You are a friendly and organized personal task assistant. Y

- Help track and organize personal tasks
- When providing a 'todo summary':
  1. List all current tasks grouped by deadline (overdue, today, this week, future)
  2. Highlight any tasks missing deadlines and gently encourage adding them
  3. Note any tasks that seem important but lack time estimates
- Proactively ask for deadlines when new tasks are added without them
- Maintain a supportive tone while helping the user stay accountable
- Help prioritize tasks based on deadlines and importance

Your communication style should be encouraging and helpful, never judgmental.

When tasks are missing deadlines, respond with something like "I notice [task] d
```

```
configurations = {"todo_category": "personal",
                  "user_id": "lance",
                  "task_maistro_role": task_maistro_role}

personal_assistant = await client.assistants.update(
    personal_assistant["assistant_id"],
    config={"configurable": configurations}
)
print(personal_assistant)
```

Assistants can be accessed like:

```
assistants = await client.assistants.search()
for assistant in assistants:
    print({
        'assistant_id': assistant['assistant_id'],
        'version': assistant['version'],
        'config': assistant['config']
    })

await client.assistants.delete("assistant_id")

work_assistant_id = assistants[0]['assistant_id']
personal_assistant_id = assistants[1]['assistant_id']
```

They can be used by choosing their id:

```
user_input = "Create ToDos: 1) Check on swim lessons for the baby this weekend
thread = await client.threads.create()
async for chunk in client.runs.stream(thread["thread_id"],
                                      personal_assistant_id,
                                      input={"messages": [HumanMessage(content=user_input
                                      stream_mode="values"):
```

```
    if chunk.event == 'values':
        state = chunk.data
        convert_to_messages(state["messages"])[-1].pretty_print()

user_input = "Give me a todo summary."
thread = await client.threads.create()
async for chunk in client.runs.stream(thread["thread_id"],
                                      personal_assistant_id,
                                      input={"messages": [HumanMessage(content=user_input
                                      stream_mode="values"):

    if chunk.event == 'values':
        state = chunk.data
        convert_to_messages(state["messages"])[-1].pretty_print()
```

Basic LangGraph Project Setup:

✅ 1. Create and Enter Your Project Folder

```
mkdir langgraph-demo
cd langgraph-demo
```

✅ 2. Set Up a Virtual Environment

```
python -m venv .venv
source .venv/bin/activate  # On Windows: .venv\Scripts\activate
```

✅ 3. Install All Required Packages

```
pip install langgraph langchain langchain_community openai python-dotenv ju
```

```
pyter
```

✅ 4. Create Your Project Files

```
touch .env langgraph_app.py langgraph_notebook.ipynb
```

✅ 5. Add Your OpenAI Key to `.env`

Open `.env` and paste this (replace with your key):

```
OPENAI_API_KEY=sk-your-openai-key
```

✅ 7. Start Jupyter Notebook

```
jupyter notebook
```

✅ Final Structure

```
langgraph-demo/
├── .venv/
├── .env
├── langgraph_app.py
├── langgraph_notebook.ipynb
```

✅ Optional: Save Your Environment

```
pip freeze > requirements.txt
```

To reinstall later:

```
pip install -r requirements.txt
```

```
import os
from dotenv import load_dotenv
from langchain.chat_models import ChatOpenAI
from langgraph.graph import StateGraph

load_dotenv()

llm = ChatOpenAI(openai_api_key=os.getenv("OPENAI_API_KEY"), model="gpt-3
```

Basic project structure

You can open up langgraph studio by:

```
pip install --upgrade "langgraph-cli[inmem]"
langgraph dev
```