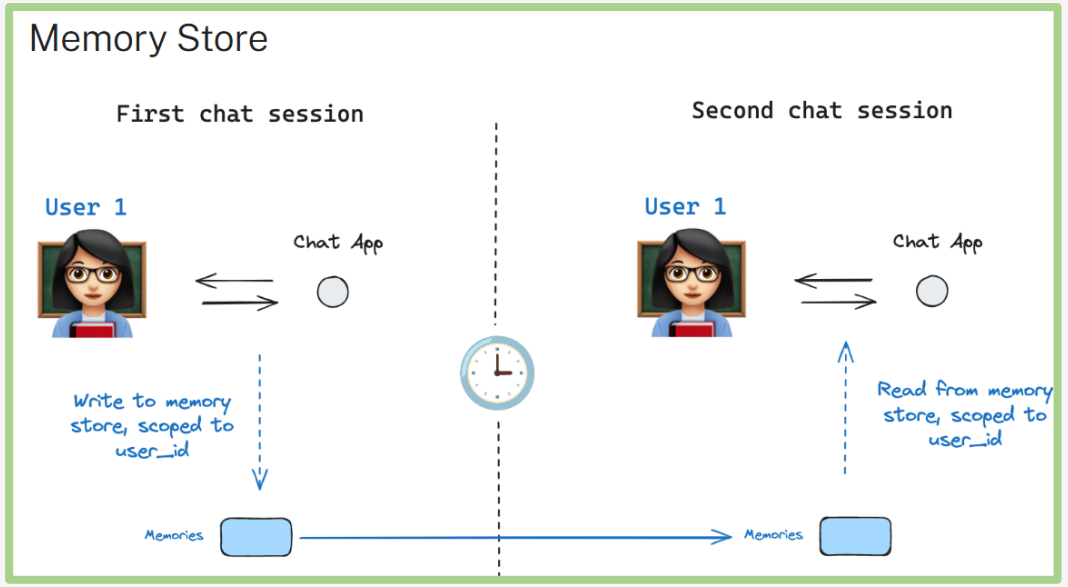
Session#4
Developing Agentic Apps with LangGraph

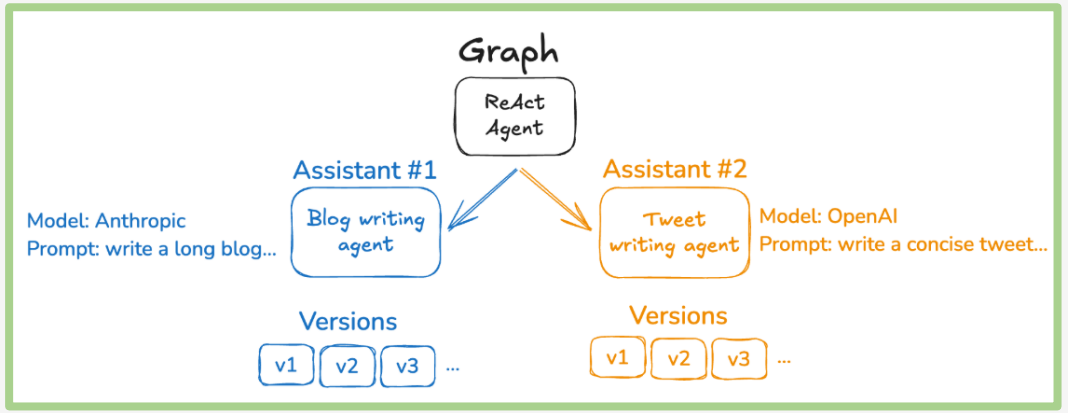
Palacode Narayana Iyer Anantharaman

4th Dec 2025

Recap: Core Agentic Features

- Autonomy, Reasoning, Planning and Adaptability
- Tool Use
- Memory
- Human In the Loop (HITL)
- Topologies – Linear, Hierarchical, Graphs, Custom
- Design Patterns
- Guard Rails
- Agentic RAG





Comparison Chart: OpenAI, CrewAI, Autogen, LangGraph

Aspect	OpenAI Agent SDK	CrewAI	Autogen	LangGraph
Core Purpose	Build **tools-augmented agents** for retrieval, actions, workflows. Integrated with OpenAI's APIs.	Multi-agent collaboration framework. Focuses on **task delegation** between agents.	Framework for building **multi-agent conversations** for problem-solving and autonomous loops.	Build **stateful, multi-step AI workflows** with clear graphs and memory. Orchestrates agents deterministically.
Architecture Style	**Tool-augmented agent APIs**. Agent calls tools/functions explicitly.	Agents as **roles in a crew**. Task planning & handoffs.	**Conversational agents** in loops. Message-based.	**State machine graph** of nodes. Declarative control flow. **Cycles** are a common pattern.
Human-in-the-Loop (HITL)	Supported via tool interfaces but **manual**	Supported via **HumanAgent**	Supported via **UserProxyAgent**	Supported via explicit **nodes for human input**
Strengths	Tight integration with OpenAI APIs. Best for **tool-using assistants.** Simple to adopt.	Clear **multi-agent orchestration**. Focuses on collaboration between specialized roles.	Flexible for **complex agent conversations**. Easy for research and prototypes.	**Deterministic control flow**. Fine-grained state management. Complex workflows with memory.
Weaknesses	Less flexible for complex agent systems or long workflows.	Less formal structure, can become spaghetti-like with complexity.	Conversations can get messy. Less structured for production workflows.	Requires explicit graph design; initial learning curve.
State Management	Implicit (via tool calls & memory APIs)	Implicit task states; no formal state machine	Loosely structured via chat history	**Explicit** state, memory, transitions
Tool/Function Calling	Native via OpenAI Functions	Supports tools via agents	Native via tools/functions	Native, plus arbitrary tools per node
Async / Parallelism	Limited; sequential agent calls	Supports concurrent agents via roles	Sequential by default	**Parallel branches** supported
Visualization	None (yet)	None officially	None officially	**Graph visualization** of flows
Integration Maturity	Early-stage, evolving fast	Community project, gaining adoption	Research-originated, robust community	More mature for **workflow orchestration**
Primary Language	Python	Python	Python	Python
Best For…	OpenAI ecosystem tools, APIs, **function-calling LLMs.**	Role-based agent teams, clear collaboration patterns.	Autonomous agents in conversation. **Research, experiments.**	Complex workflows, **state machines**, production pipelines.

Summary Recommendations

If you want…	Recommended Tool
Easiest to get started	OpenAI Agent SDK
Role-based multi-agent teamwork	CrewAI
Conversational agent simulations	Autogen
Structured, production workflows	LangGraph

Additional Insights

- **Agent SDK** is best when you rely on OpenAI tools and want retrieval / action pipelines. Easiest to get started.
- **CrewAI** shines when modeling teams of specialized agents (Planner, Researcher, Executor) working together.
- **Autogen** is flexible for simulating autonomous dialogues or for academic experiments where multiple agents "talk.“
- **LangGraph** is optimal when you need reliable, traceable, debuggable flows—great for production, HITL, or regulated environments. Implementation of **Persistence and state management, support for low level control, flexible topologies** are very strong. E.g. You can explicitly design branches, loops, guards, retries, and fallbacks.

LangGraph Basics

Graph: Nodes and Edges G = (V, E)

A node can be an agent or a tool

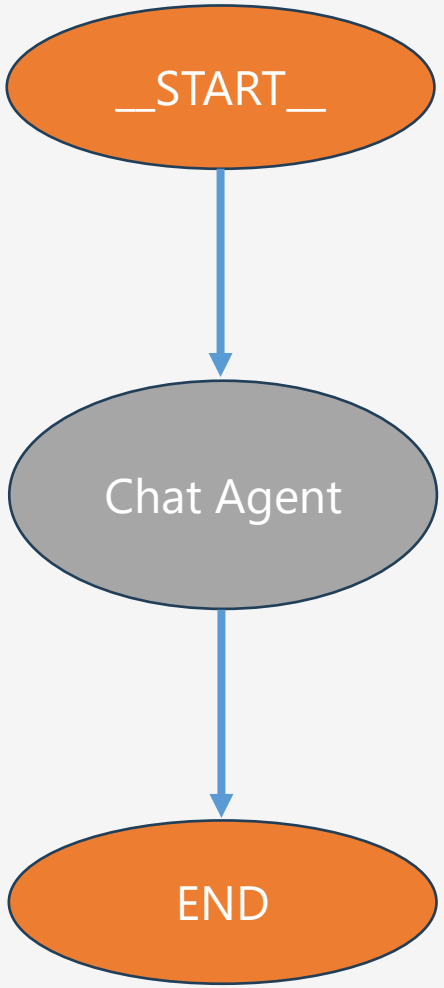
The agent typically is based on LLM while a tool is a function

Edges connect nodes, i.e the output of a node is passed to the input of another. Edges can be conditional.

- A code generator send the code for review and the reviewer returns the feedback
- Invoking a node can be conditional. An LLM can respond to a prompt without invoking a search tool if no web search is needed.

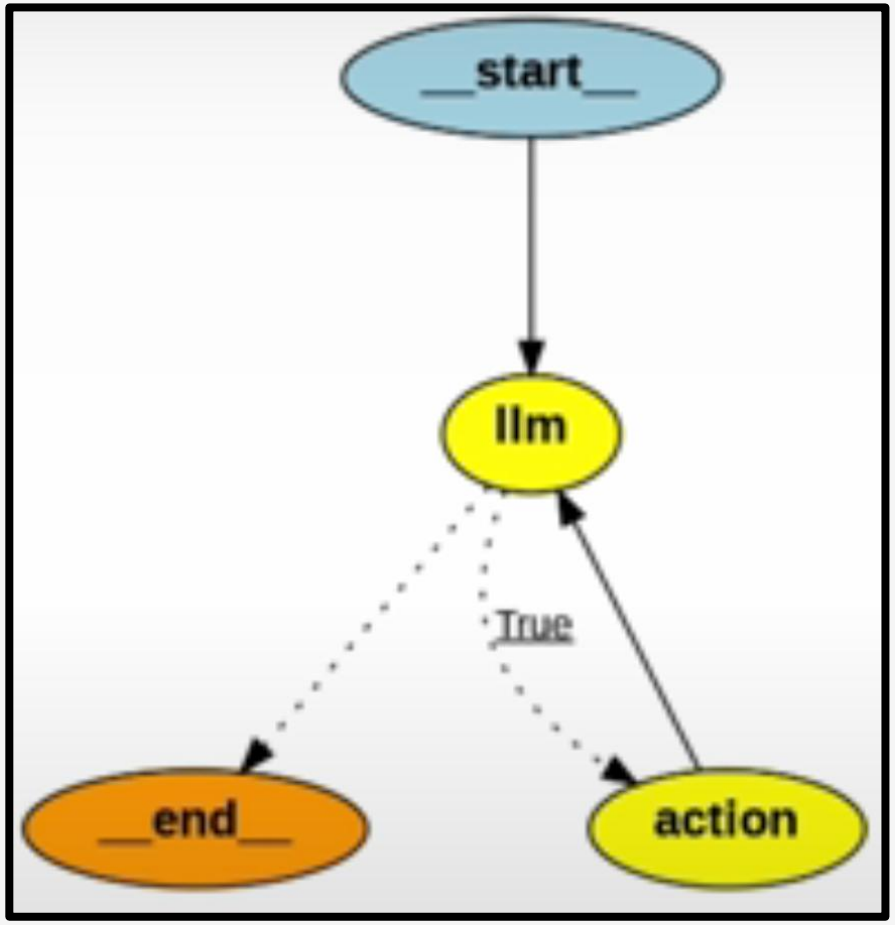
The state stores application information, while nodes access and modify it.

- E.g. The user query is stored in the state, the planner produces a response (e.g. A web service request) and that is stored in the state. The tool processes this request and writes its response to the state.



LangGraph: Conditional Edges

- The user query and system prompt is given to the LLM
- LLM is provided a set of tools
- LLM decides if it needs to use a tool and if so invokes it, otherwise returns the result
- The tool is a function that corresponds to an action
- END node signifies completion.
- Agent's state is maintained throughout in AgentState



Code Walkthrough

Please see:

- http://localhost:8888/notebooks/simple_chatbot_graph.ipynb
- http://localhost:8888/notebooks/my_langgraph_example.ipynb
- tool_call_example.py

Hands On 1: Your First LangGraph agent

1. **Reproduce:** Run the notebook: simple_chatbot_graph.ipynb that takes a prompt and uses an LLM to produce a result

2. **Reproduce:** Build a 2 node system: 1 LLM agent and 1 Tool agent using the file: my_langgraph_example.ipynb

3. **Replace:** Instead of search API used in the provided code, use a Weather API, keep the same 2 agent architecture. If some query was issued where LLM agent can't answer, you must say "I don't know the answer". Modify the prompt to answer weather related questions accurately. Update the state.

LangChain versus LangGraph Agents and Tools

- Contrast the agentic app generation approach of LangGraph versus LangChain
- Compare the way the tools are handled: See the code in react_example_for_exercises.py (This hands on was done in Session 2)
- LangGraph requires additional steps but scales well for multiple agents. Also has features like Checkpointers.
- LangChain is more convenient for ReAct style single agents: that is 1 reasoning agent + 1 or more tools

LangChain Tool Call

Key points:

- Define tools with `@tool` decorator or as callable classes.
- Invoke tools inside graph nodes using `.invoke({"arg_name": value})`.
- Pass tool arguments as a dictionary.
- The node returns updated state including the tool output.

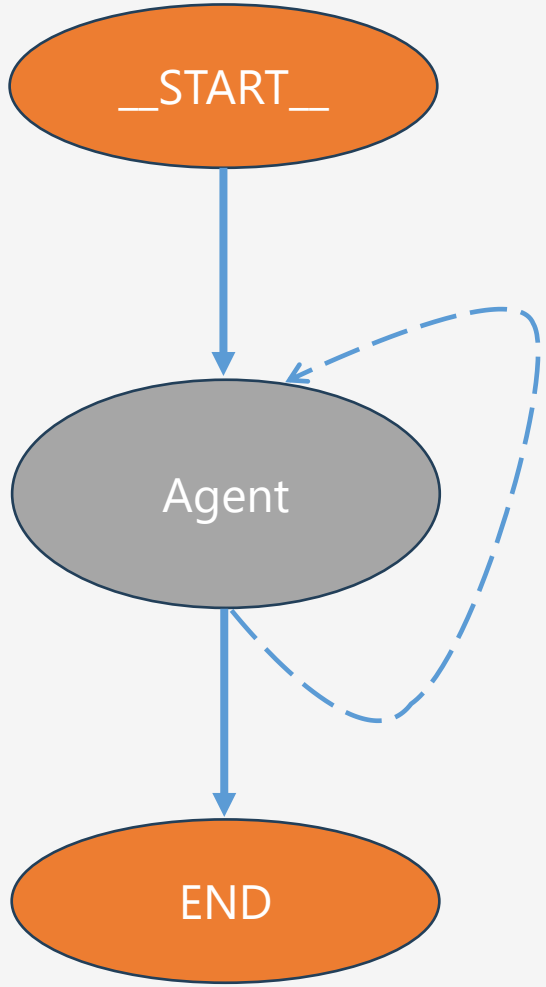
This approach integrates tool calls seamlessly into LangGraph workflows.

Reference:

- LangGraph tool calling guide: https://langchain-ai.github.io/langgraph/how-tos/tool-calling/
- LangChain custom tools: https://python.langchain.com/docs/how_to/custom_tools/

Hands On 2: LangGraph agent with a loop pattern – see PDF

- We have one node: **Agent**
- Assume the input to come from a file of text or a folder (e.g. folder of pngs)
- In the state we can add a **loop count**: Decrement after every prompt/response.
- Exit the loop after a preset number of iterations are completed
- **Later**: Replace the loop counter agent with a Chatbot
- Use the **Human In the Loop (HITL)** feature to facilitate termination.
- Experiment by implementing chat history, modifying it by HITL



Example: RAG and Web Search using Langgraph

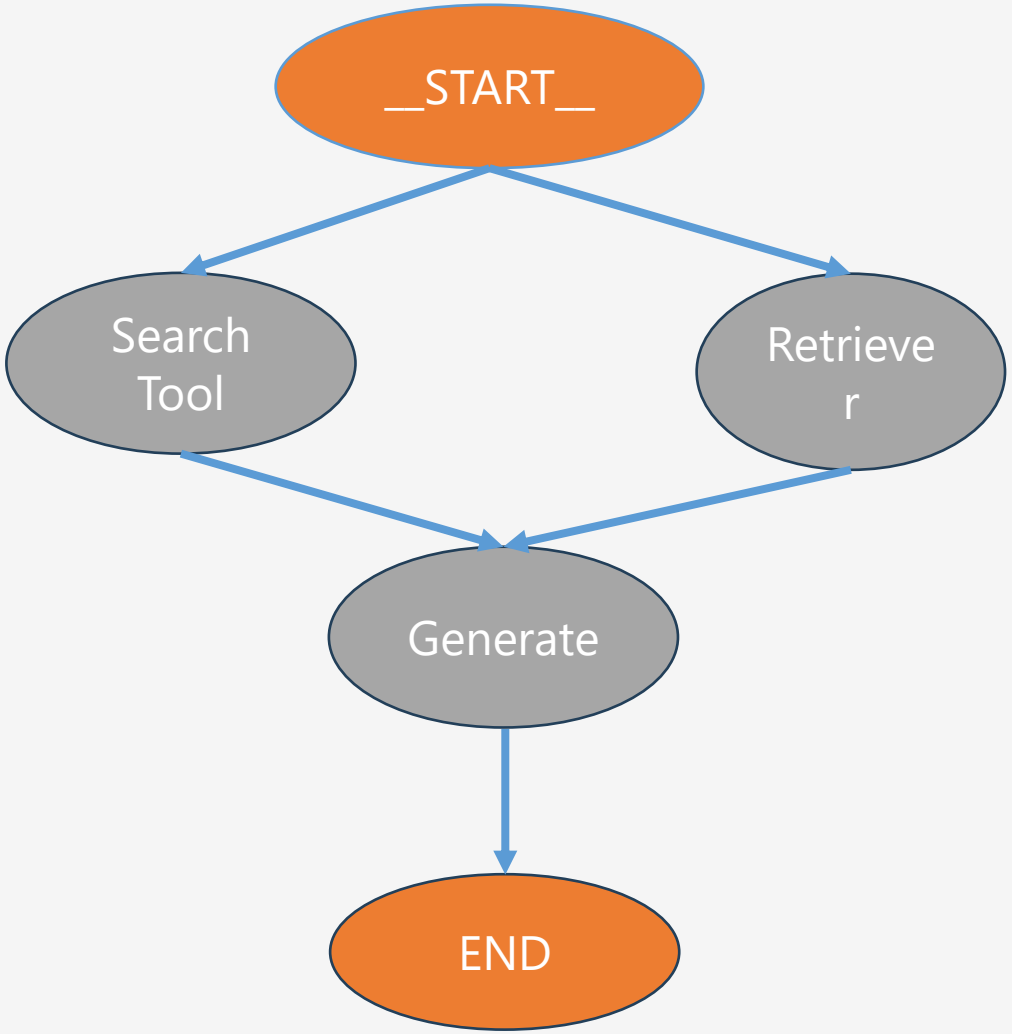
See: http://localhost:8888/notebooks/agent_and_tool.ipynb

We have a vector database containing all the lecture notes

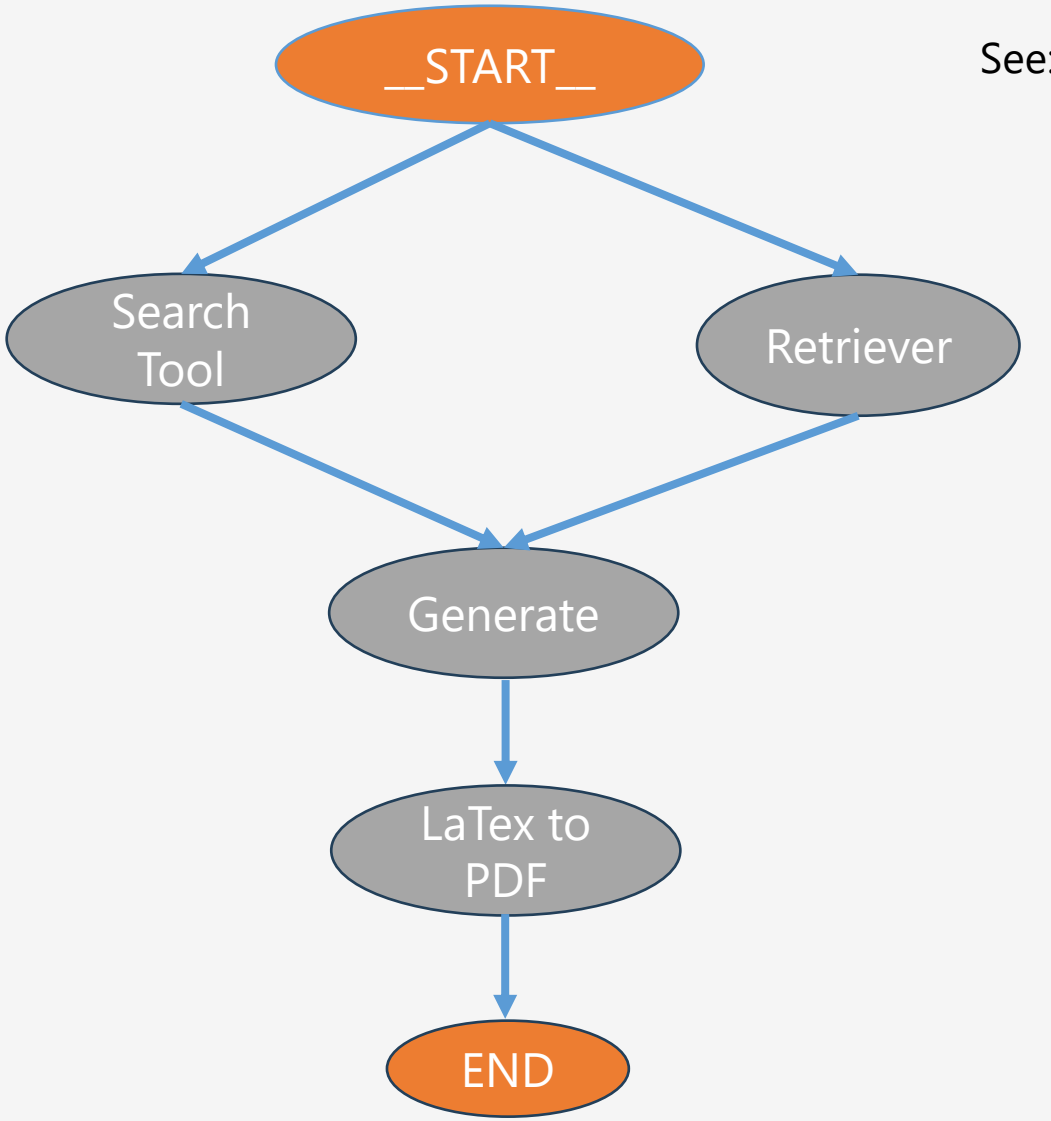
We need an agent system that invokes the RAG for queries concerning LLMs and GenAI and for other queries it should use a web search tool.

Build an agentic application for this purpose

Use a router to direct the control flow: either web search or RAG



Exercise: Extend this app to generate PDF



See: http://localhost:8888/notebooks/agent_and_tool_with_pdf.ipynb

Optional Reading: Tool Calling Internals (theory)

Exercises:

1. Go through the prompt templates and tool calling special tokens for Llama 3.x and gemma models.

2. Look at the prompt templates of Multimodal models that support audio, video, images, text (e.g. Microsoft Phi 4) and find out how the different data types such as images are passed to the LLM. E.g. https://build.nvidia.com/microsoft/phi-4-multimodal-instruct/modelcard

Tool Calling Internals: Llama 3.2 Templates

User and assistant conversation

Here is a regular multi-turn user assistant conversation and how its formatted.

Input Prompt Format

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful assistant<|eot_id|><|start_header_id|>user<|end_header_id|>

Who are you?<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Model Response Format

```
I'm an AI assistant, which means I'm a computer program designed to simulate conversation and answer questions to the best of my

I don't have a personal name, but I'm often referred to as a "virtual assistant" or a "chatbot." I'm a machine learning model, \

I can help with a wide range of topics, from general knowledge and trivia to more specialized subjects like science, history, a

So, what can I help you with today?<|eot_id|>
```

Tool Calling Internals: Llama 3.2 Templates

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are an expert in composing functions. You are given a question and a set of possible functions.
Based on the question, you will need to make one or more function/tool calls to achieve the purpose.
If none of the function can be used, point it out. If the given question lacks the parameters required by the function,
also point it out. You should only return the function call in tools call sections.

If you decide to invoke any of the function(s), you MUST put it in the format of [func_name1(params_name1=params_value1, params_
You SHOULD NOT include any other text in the response.

Here is a list of functions in JSON format that you can invoke.

[
    {
        "name": "get_weather",
        "description": "Get weather info for places",
        "parameters": {
            "type": "dict",
            "required": [
                "city"
            ],
            "properties": {
                "city": {
                    "type": "string",
                    "description": "The name of the city to get the weather for"
                },
                "metric": {
                    "type": "string",
                    "description": "The metric for weather. Options are: celsius, fahrenheit",
                    "default": "celsius"
                }
            }
        }
    }
]<|eot_id|><|start_header_id|>user<|end_header_id|>

What is the weather in SF and Seattle?<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Ref: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/text_prompt_format.md

Model Response:

[get_weather(city='San Francisco', metric='celsius'), get_weather(city='Seattle', metric='celsius')]<|eot_id|>

Zero shot Function Calling in User Message

```
<|begin_of_text|><|start_header_id|>user<|end_header_id|>

Questions: Can you retrieve the details for the user with the ID 7890, who has black as their special request?
Here is a list of functions in JSON format that you can invoke:
[
    {
        "name": "get_user_info",
        "description": "Retrieve details for a specific user by their unique identifier. Note that the provided function is in P
        "parameters": {
            "type": "dict",
            "required": [
                "user_id"
            ],
            "properties": {
                "user_id": {
                "type": "integer",
                "description": "The unique identifier of the user. It is used to fetch the specific user details from the databa
            },
            "special": {
                "type": "string",
                "description": "Any special information or parameters that need to be considered while fetching user details.",
                "default": "none"
                }
            }
        }
    }
]

Should you decide to return the function call(s),Put it in the format of [func1(params_name=params_value, params_name2=params_va

NO other text MUST be included.<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Ref: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/text_prompt_format.md

Model Response:

[get_user_info(user_id=7890, special='black')]<|eot_id|>

Enterprise Example

Case Study: Enterprise Scale Research Intelligence Pipeline

Context:

A large AI research organization wants to build an internal "Research Intelligence Assistant" to help scientists and engineering managers stay on top of rapidly growing ML literature (e.g., ICLR 2024–2025).

The organization has **18,000 research papers (ICLR 2024-25)**, each with:

- Paper metadata (title, abstract, authors, submission date)
- Review summaries and reviewer scores
- Author rebuttals
- The full PDF

Value Proposition

- It helps researchers quickly identify the most promising and relevant research papers for a given topic by combining AI summarization plus review-based evaluation.
- Related work survey
- Identification of open problems for future research
- While this problem is illustrated for a research organization, it can be mapped easily for a business use case.

Category	Example Sources
Competitive Intelligence	Competitor product pages, pricing pages, press releases
Financials	10-K, 10-Q, investor calls
Customer sentiment	Amazon reviews, TrustPilot, Reddit, X
Analyst research	Gartner, Forrester, IDC
Internal data	Sales call transcripts, CRM notes
Macro trends	Industry reports, patents, regulations

Product Goals

The goal is to create a *deep research agent* that can:

- Process the full dataset reliably
- Extract insights after multi-step analysis
- Allow humans to intervene or guide decisions
- Save intermediate progress
- Recover from failures or resume long multi-hour workflows
- Provide auditable, reproducible outputs
- LangGraph is a perfect fit because of its **agent state**, **durable checkpoints**, **interrupt-friendly execution**, and **persistent memory**.

Key Requirements and Challenges

Design a LangGraph-based Research Intelligence Pipeline that processes a large corpus of scientific papers, performs multi-stage analysis, supports human intervention, and persists both intermediate outputs and long-term memory.

The assistant must:

Incrementally process 18k papers without restarting

Use persistent memory to avoid recomputing embeddings and summaries

Save checkpoints after each paper

Interrupt execution and request human input when models indicate uncertainty

Resume after interruptions without losing progress

Produce a final set of ranked paper recommendations

Implementing Memory

Review the ipynb notebooks on Chat history

Memory and Persistence

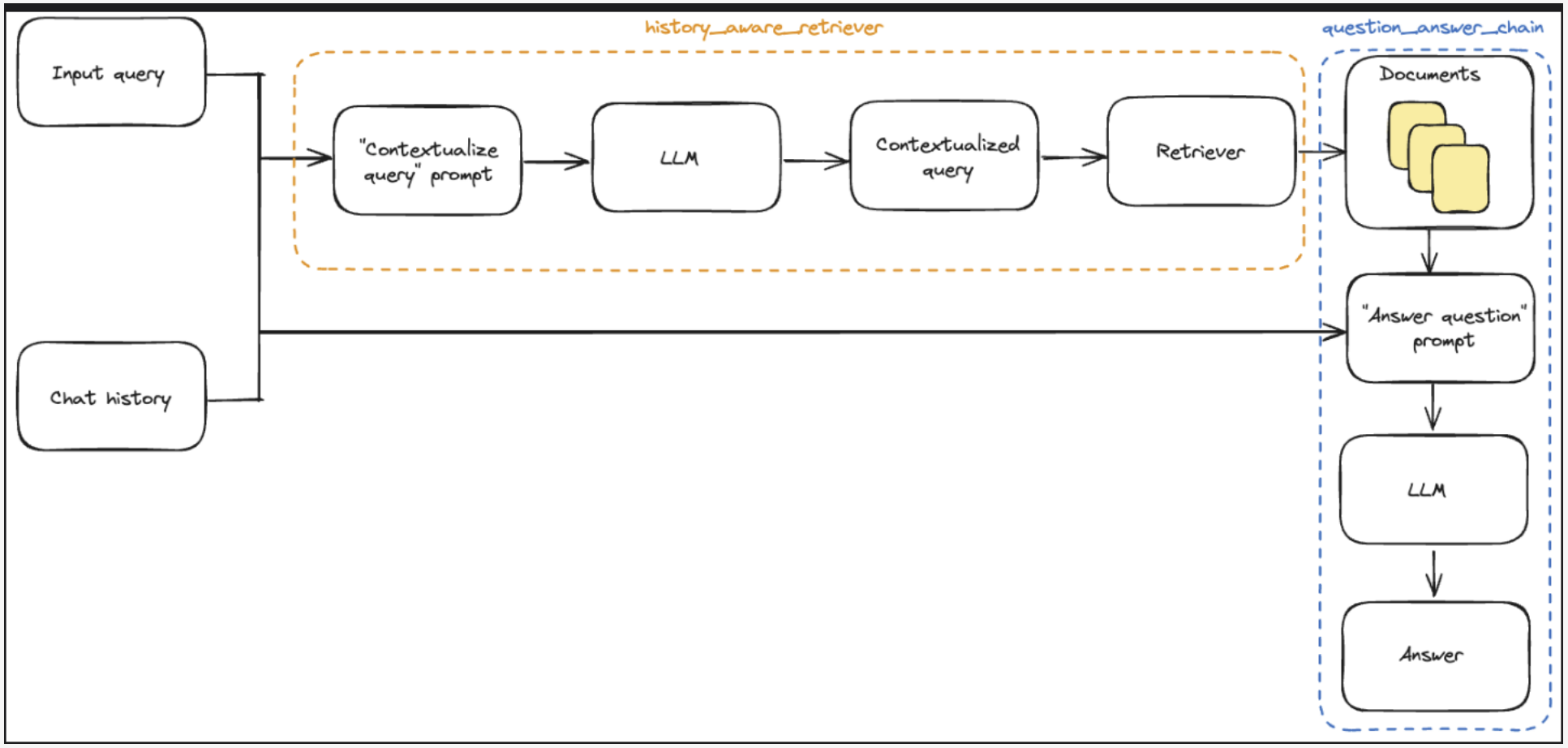
- Implementing Chat History: session wide memory, in-memory storage, trimmer
- Chat history and RAG
- LangGraph Persistence

Demos and Code walkthroughs

- demo2_agent_with_memory.py : OpenAI Agent SDK application example
- http://localhost:8888/notebooks/basic_concepts_chat_history.ipynb : LangGraph session and in-memory buffer example
- manage_conversation_history.ipynb : trimmer demo

Optional Theory: How this is applied for Advanced RAG – Query Rewriting?

Supporting chat history RAG example

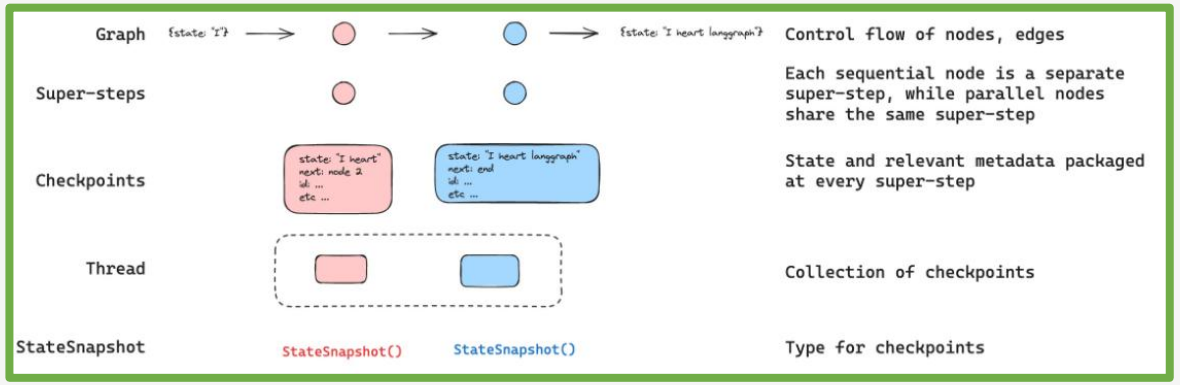


Condensing questions with another LLM

- This chain has two steps. First, it condenses the current question and the chat history into a standalone question.
- This is necessary to create a standalone vector to use for retrieval.
- After that, it does retrieval and then answers the question using retrieval augmented generation with a separate model.
- Part of the power of the declarative nature of LangChain is that you can easily use a separate language model for each call.
- This can be useful to use a cheaper and faster model for the simpler task of condensing the question, and then a more expensive model for answering the question.

Persistence in LangGraph

- **Built-in Persistence**: LangGraph includes a persistence layer via *checkpointers*.
- **State Checkpoints**: Checkpointers save the graph state at every super-step.
- **Thread Storage**: Checkpoints are stored in a *thread* for post-execution access.
- **Post-Execution Access**: Threads enable retrieval of the graph's state after execution.
- **Enables Advanced Capabilities**:
  - ❑ Human-in-the-loop workflows
  - ❑ Long-term memory
  - ❑ Time-travel through prior states
  - ❑ Fault-tolerance and recovery




```python
# Assume graph is compiled with a checkpointer and thread_id is known

# 1. Get history of states (checkpoints)
history = graph.get_state_history({"configurable": {"thread_id": "1"}})

# 2. Select a checkpoint to time travel to
checkpoint = history[[2]](https://langchain-ai.github.io/langgraph/concepts/persisten

# 3. Resume execution from that checkpoint
config = {
    "configurable": {
        "thread_id": "1",
        "checkpoint_id": checkpoint.config["configurable"]["checkpoint_id"]
    }
}
new_state = graph.invoke(None, config=config)
```

Using Checkpointers: Simple code example

See checkpoint_example.py

Optional: Parallel Execution and Super Step

In this example, nodes "b" and "c" run in parallel after "a" finishes, then "d" runs after both "b" and "c" complete.

- **Defer node execution:** You can mark nodes as deferred (`defer=True`) to delay their execution until all other pending tasks complete, useful when branches have different lengths.
- **Transactional super-steps:** All nodes in a super-step execute concurrently, and the entire super-step is transactional—if any node fails, none of the updates apply.
- **Exception handling:** You can handle exceptions inside node functions or use retry policies to retry failing nodes without redoing successful ones.
- **Async support:** LangGraph supports async node functions and async graph invocation (`ainvoke`), enabling concurrent IO-bound operations like API calls.
- **Example of parallel LLM calls and aggregation:**

```python
from langgraph.graph import StateGraph, START, END
from typing_extensions import TypedDict
import operator

class State(TypedDict):
    aggregate: list

def a(state: State):
    return {"aggregate": state["aggregate"] + ["A"]}

def b(state: State):
    return {"aggregate": state["aggregate"] + ["B"]}

def c(state: State):
    return {"aggregate": state["aggregate"] + ["C"]}

def d(state: State):
    return {"aggregate": state["aggregate"] + ["D"]}

builder = StateGraph(State)
builder.add_node(a)
builder.add_node(b)
builder.add_node(c)
builder.add_node(d)

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()
result = graph.invoke({"aggregate": []})
print(result)
```

Threads

Thread in LangGraph:

- A *thread* is a unique ID assigned to each checkpoint saved by a checkpointer.
- It accumulates the state across a sequence of runs.

State Persistence:

- The graph's state is persisted to the thread during execution.
- Thread state includes both *current* and *historical* states.

Configuration:

- Specify the thread_id in the config when invoking a graph with a checkpointer: {"configurable": {"thread_id": "1"}}

Thread Management:

- Threads must be created *before* running the graph to enable persistence.
- LangGraph Platform API provides endpoints to create, retrieve, and manage threads and their state

Threads

Using threads in LangGraph

- Threads group traces (runs) of a conversation or workflow.
- Each run updates the thread's state, saved as checkpoints.
- You can retrieve, inspect, and edit thread history in the LangGraph Studio UI.
- Threads enable multi-turn conversations by linking traces with a unique thread ID.

Persistence in LangGraph: Debuggability

Persistence features not only provide rich set of functionality (e.g. Human In The Loop) but also strong testability and debuggability

Checkpoints

- A **Checkpoint** is a snapshot of the graph state at a specific point in time.

- Each checkpoint is represented by a StateSnapshot object.

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

config = {"configurable": {"thread_id": "1"}}
graph.invoke({"foo": ""}, config)
# Checkpoints are saved automatically at each super-step
```

Key Properties of a Checkpoint:

- **config**: Configuration tied to this checkpoint.
- **metadata**: Metadata associated with the checkpoint.
- **values**: Current state channel values at this step.
- **next**: Tuple of node names scheduled for next execution.
- **tasks**: Tuple of PregelTask objects:
  - Contains information about upcoming tasks.
  - Includes error information if previous attempts failed.
  - Stores interrupt-related data if the graph was dynamically interrupted.

```
import sqlite3
from langgraph.checkpoint.sqlite import SqliteSaver

conn = sqlite3.connect("checkpoints.sqlite", check_same_thread=False)
checkpointer = SqliteSaver(conn)
graph = workflow.compile(checkpointer=checkpointer)
```

Purpose:

- Checkpoints are **persisted**.
- They enable **state restoration** to any prior point in a thread's execution.

Checkpoints – Sample code

```python
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import InMemorySaver
from typing import Annotated
from typing_extensions import TypedDict
from operator import add

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

def node_a(state: State):
    return {"foo": "a", "bar": ["a"]}

def node_b(state: State):
    return {"foo": "b", "bar": ["b"]}


workflow = StateGraph(State)
workflow.add_node(node_a)
workflow.add_node(node_b)
workflow.add_edge(START, "node_a")
workflow.add_edge("node_a", "node_b")
workflow.add_edge("node_b", END)

checkpointer = InMemorySaver()
graph = workflow.compile(checkpointer=checkpointer)

config = {"configurable": {"thread_id": "1"}}
graph.invoke({"foo": ""}, config)
```

After we run the graph, we expect to see exactly 4 checkpoints:

- empty checkpoint with `START` as the next node to be executed
- checkpoint with the user input `{'foo': '', 'bar': []}` and `node_a` as the next node to be executed
- checkpoint with the outputs of `node_a` `{'foo': 'a', 'bar': ['a']}` and `node_b` as the next node to be executed
- checkpoint with the outputs of `node_b` `{'foo': 'b', 'bar': ['a', 'b']}` and no next nodes to be executed

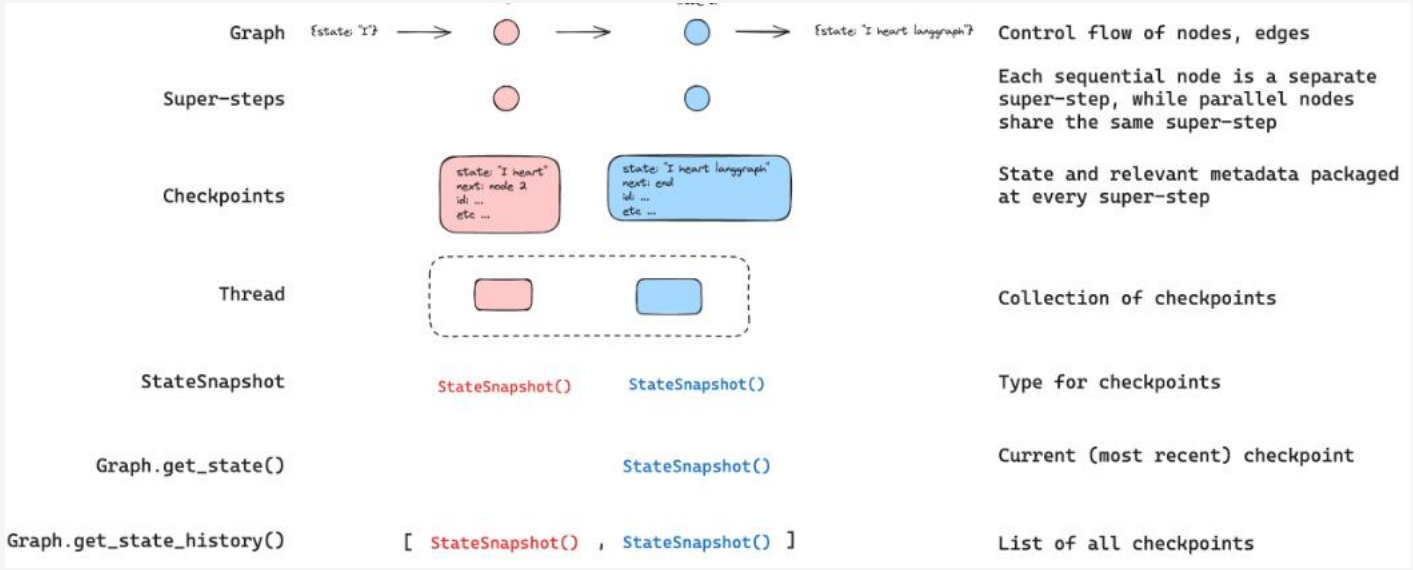
Note that the `bar` channel values contain outputs from both nodes as we have a reducer for `bar` channel.

REF: https://langchain-ai.github.io/langgraph/concepts/persistence/#checkpoints

Getting States

```
# get the latest state snapshot

config = {"configurable": {"thread_id": "1"}}

graph.get_state(config)

# get a state snapshot for a specific
checkpoint_id

config = {"configurable": {"thread_id": "1",
"checkpoint_id": "1ef663ba-28fe-6528-8002-
5a559208592c"}}

graph.get_state(config)
```

Use get_state_history to get all snapshots




```
StateSnapshot(
    values={'foo': 'b', 'bar': ['a', 'b']},
    next=(),
    config={'configurable': {'thread_id': '1', 'checkpoint_ns': '', 'checkpoint
    metadata={'source': 'loop', 'writes': {'node_b': {'foo': 'b', 'bar': ['b']}
    created_at='2024-08-29T19:19:38.821749+00:00',
    parent_config={'configurable': {'thread_id': '1', 'checkpoint_ns': '', 'che
)
```

Replay from checkpoints

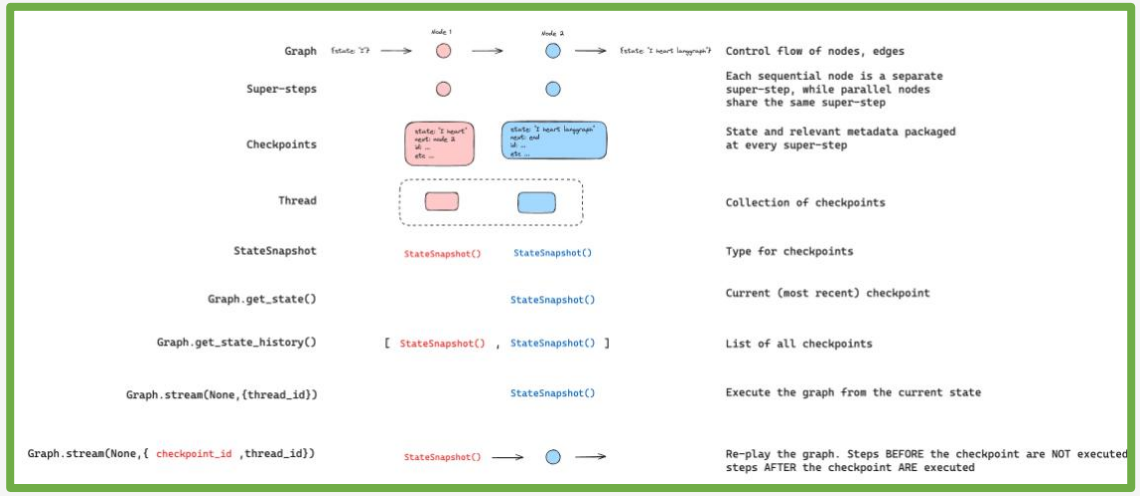
Replay In LangGraph

- **Replay** allows playback of a prior graph execution from any checkpoint.
- Use thread_id to specify the thread and checkpoint_id to specify the checkpoint.
- When invoked, LangGraph:
  - Replays steps before the checkpoint_id (without re-executing).
  - Executes steps after the checkpoint_id as a new fork (even if previously run).

Purpose of Replay:

- Enables **partial re-execution** from any checkpoint.
- Supports **experimentation, debugging, and branching** behaviors.
- Facilitates **time-travel-like exploration** of graph workflows.

```
config = {
    "configurable": {
        "thread_id": "1",
        "checkpoint_id": "0c62ca34-ac19-445d-bbb0-5b4984975b2a"
    }
}
graph.invoke(None, config=config)
```



Hands On 3: Using Checkpointers to save/resume execution

1. Use the single agent notebook that we saw in simple_chatbot_graph.ipynb

2. Implement a checkpoint save and resume: Use in-memory storage

3. Discuss how this can be used to implement a session history similar to chatgpt. Assume only one thread/session and describe how to augment this application to implement chat history.

Heads Up: Using Interrupts

```
from langgraph.types import interrupt, Command

def agent_node(state: AgentState):
    # Pause execution and wait for human approval
    approval = interrupt("Approve this action?")
    if approval:
        return {"messages": state["messages"] + [AIMessage(content="Action approved")]}
    return {"messages": state["messages"] + [AIMessage(content="Action rejected")]}

# Pause execution
graph.invoke({"messages": [HumanMessage(content="Delete database")]}, config)

# Resume after human decision
graph.invoke(Command(resume=True), config)
```

Enterprise Example

See multi_agents_real_world.py (not uploaded to shared drive)

Time Travel

How it works:

- Each execution of a LangGraph graph with persistence enabled saves checkpoints representing the full graph state at discrete super-steps.
- These checkpoints are stored in a thread identified by a thread_id.
- You can retrieve the history of checkpoints for a thread using graph.get_state_history(config) or via the LangGraph API.
- To time travel, you invoke the graph again specifying the thread_id and a chosen checkpoint_id to resume from that prior state.
- Optionally, you can modify the state before resuming to explore alternative trajectories.

Update State in LangGraph

- **Purpose**: Manually modify the state of a running or completed graph.
- Uses the method: graph.update_state()

Arguments

- Config: thread_id, checkpoint_id (optional)
- Values:
  - Key-value pairs to update graph state
  - Behaves like updates from graph nodes
  - **Reducer behavior respected**: If a reducer exists (e.g., list append), update is combined. Otherwise value is overwritten

Initial State:

python Copy Edit

```python
{"foo": 1, "bar": ["a"]}
```

Update Command:

python Copy Edit

```python
graph.update_state(config, {"foo": 2, "bar": ["b"]})
```

Resulting State:

python Copy Edit

```python
{"foo": 2, "bar": ["a", "b"]}
```

- `foo`: Overwritten to `2`.
- `bar`: Appended `"b"` via reducer.

Code Example: checkpoint_update_state.py

As_node in LangGraph

- **Purpose**:

Specify which **node** the state update should appear to originate from.

- **Why it matters**:

LangGraph determines **next execution steps** based on the **last node** that updated the state.

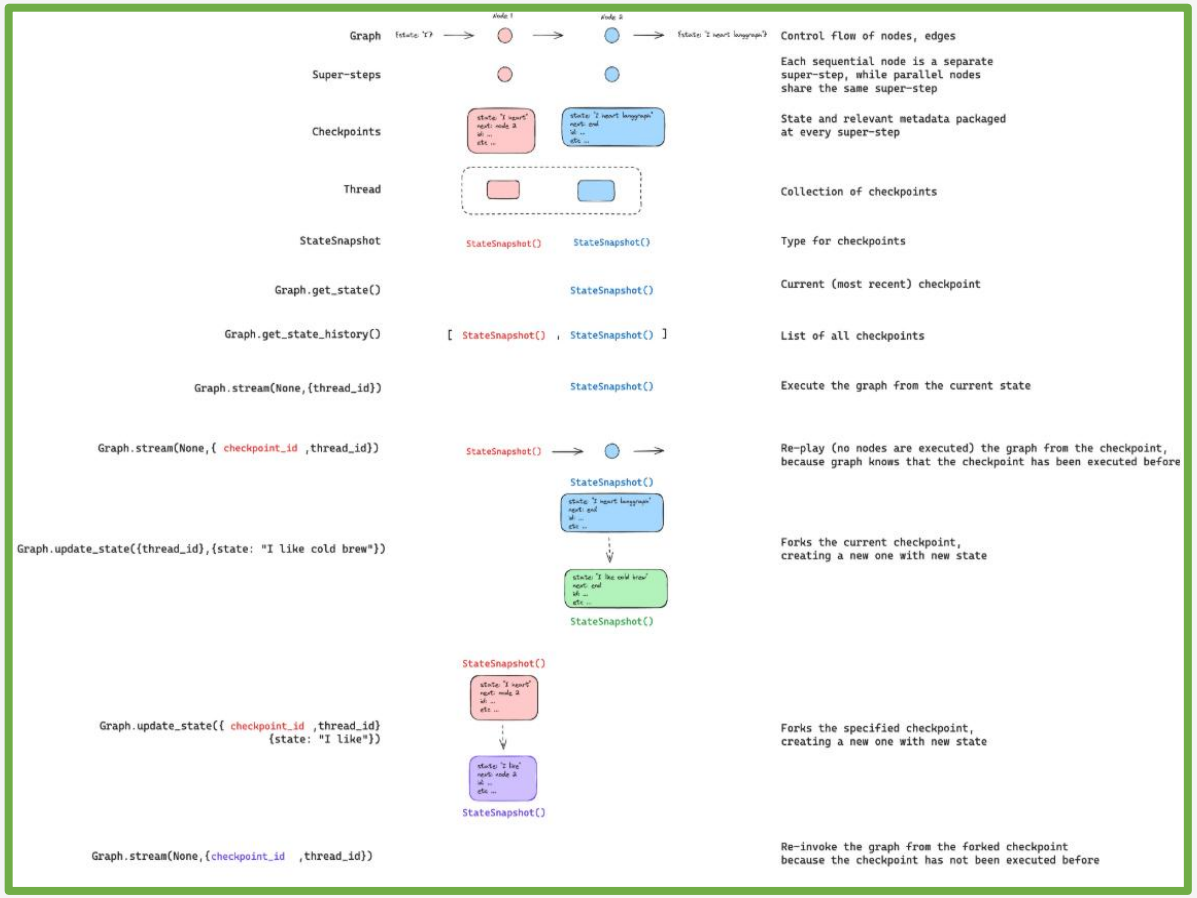
- **Behavior**:

If as_node is provided:

Defaults to the last node that updated the state (if unambiguous).

Otherwise, LangGraph may raise ambiguity issues.

- **Use Case**: Control future graph execution flow when forking state, replaying, or injecting manual updates.



LangGraph Capabilities Enabled by Checkpoints

✅ Human-in-the-loop

- Enables inspection, interruption, and approval of graph steps by humans.
- Supports workflows where humans review or modify state mid-execution.
- Graph can safely **resume after human updates**.

✅ Memory

- **Persists state across sessions** (threads retain memory of prior steps).
- Ideal for **conversational memory** and ongoing interactions.
- Supports follow-up queries with awareness of past context.

✅ Time Travel

- Allows **replay of previous executions** for debugging or review.
- Supports **forking from any checkpoint** to explore alternative paths.
- Enables systematic analysis of graph behaviors.

✅ Fault-tolerance

- Checkpoints allow graphs to **resume from the last successful step** after failure.
- Partial progress within a superstep is saved (no need to re-run successful nodes).
- Robust error recovery mechanism.

✅ Pending Writes Handling

- Stores successful node updates within a superstep, even if other nodes fail.
- Ensures resumed executions **avoid redundant computation**.

Interrupts

Key Capabilities: Interrupts & Persistent Execution

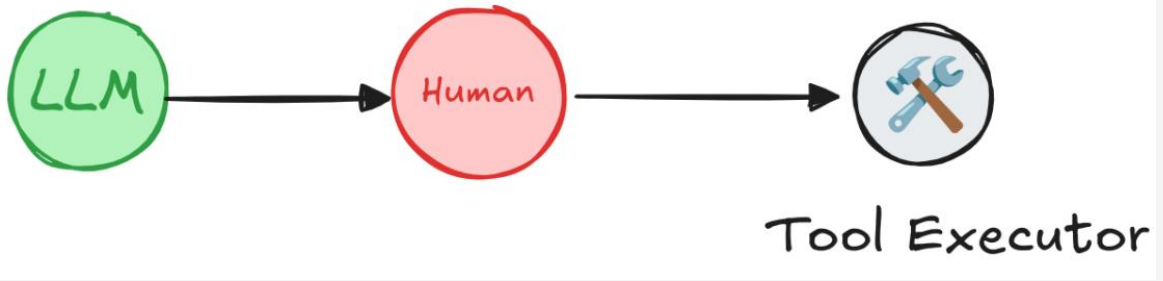
🔄 Persistent Execution State

- **Graph execution can pause indefinitely** using LangGraph's persistence layer.
- **Checkpoints after each step** capture full execution context.
- Supports **asynchronous human review/input** with no time limits.
- Workflow resumes from **exact point of interruption**.

⏸️ Two Ways to Pause a Graph

- **Dynamic Interrupts:**
  - Triggered **inside a node** based on graph state.
- **Static Interrupts:**
  - Use `interrupt_before` / `interrupt_after` to pause **before or after specific nodes.**

- **Flexible integration points**: Human-in-the-loop logic can be introduced at any point in the workflow. This allows targeted human involvement, such as approving API calls, correcting outputs, or guiding conversations.



Hands On 4: Static Interrupts

- Imagine that you have the single chatbot agent. After every prompt-response we would like to say either “exit” or “redo” or provide a new prompt
- If new prompt is given the chatbot should execute the new prompt
- If exit, end the application
- If redo repeat the same prompt in order to provide an improved response
- Use static interrupts along with the chatbot agent app to implement this.

Human In the Loop - Patterns

✅ 1. Approve or Reject

- Pause graph **before critical actions** (e.g., API calls).
- Human approves or rejects the step.
- Can **route execution** based on decision.

🖍️ 2. Edit Graph State

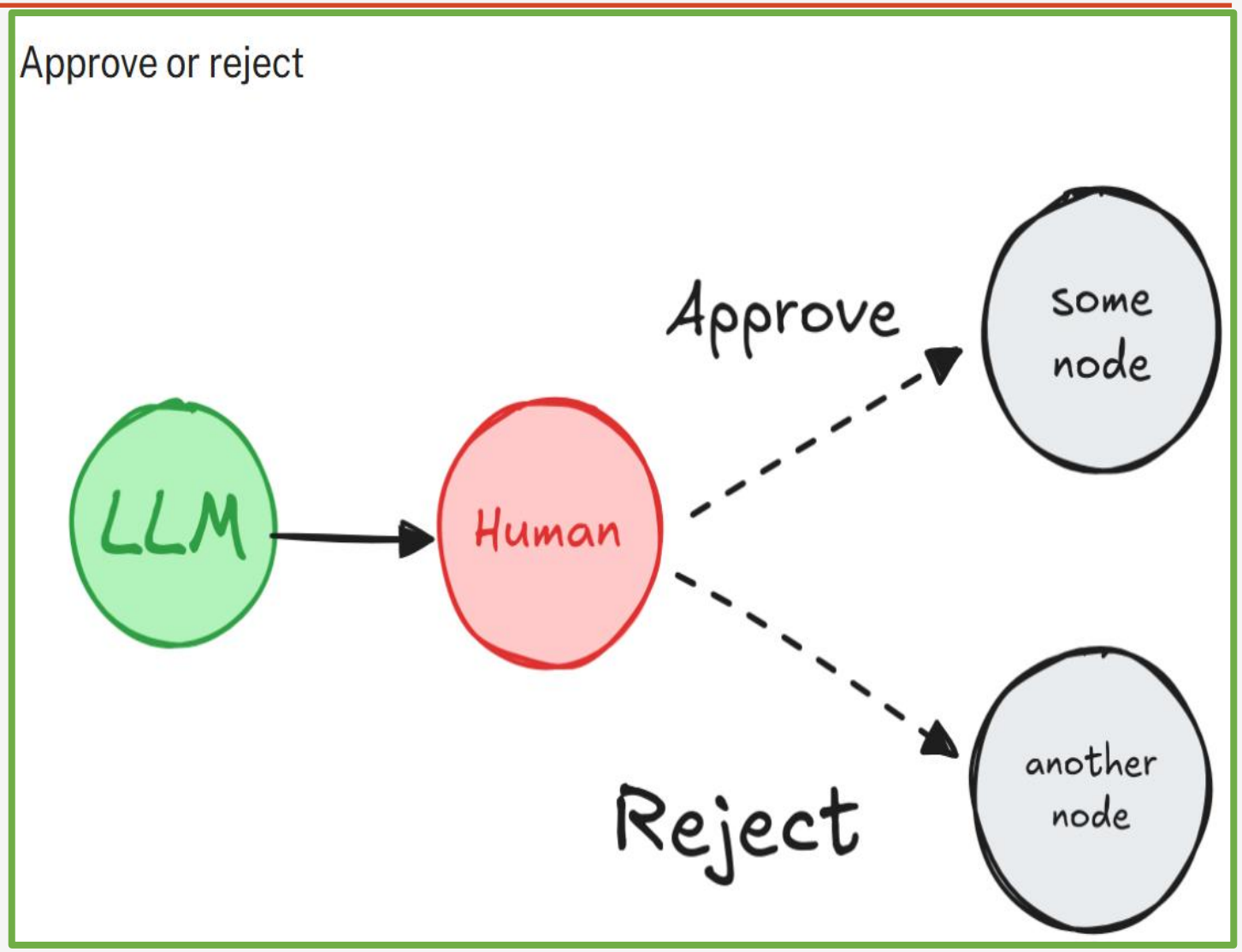
- Pause graph to **review or modify state.**
- Correct errors or add missing information.
- Resume with **updated state from human input.**

🔍 3. Review Tool Calls

- Pause graph to **inspect and modify tool calls** (e.g., API requests, LLM tools).
- Ensures correctness before tool execution.

🛡️ 4. Validate Human Input

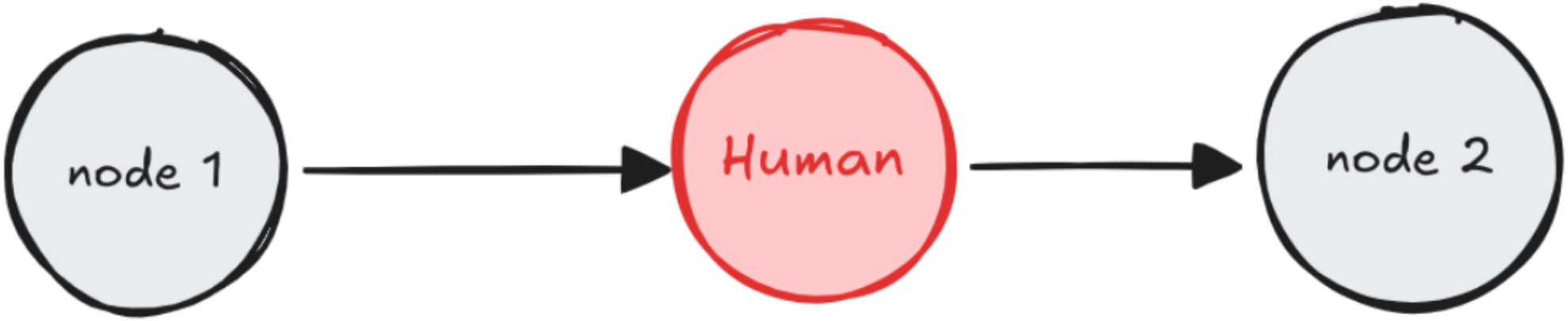
- Pause to **verify human input** before continuing.
- Ensures valid, complete data before next step.



Industry Example: Curating Multimodal LLM Outputs

- Suppose you use a multimodal LLM to process images and extract important information, based on which there is an action
- Before such an action, a human review may be preferred
- Human reviews can turn an invalid data in to a approved data
- Very crucial for enterprise use cases

Edit State



A human can review and edit the state of the graph. This is useful for correcting mistakes or updating the state with additional information.

See the code in hitl_edit_state_example.py

Hands On 5: HITL for Edit State

- Review the code in hitl_edit_state_example.py
- Use this as a boilerplate and implement a summarizer application: In our case, take a page from OpenAI Agent SDK documentation and summarize it for RAG later.
- Review the generated summary using HITL
- Put it to downstream use (e.g. RAG)

Demo

See: dynamic_interrupts_example.py,

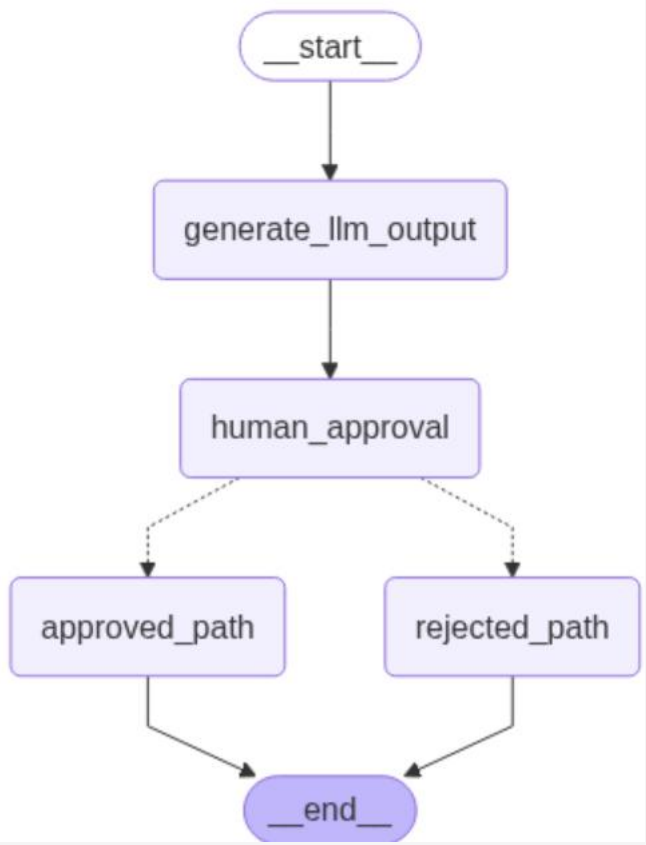
static_interrupts_example.py

Demo

See: hitl_approve_reject_pattern.py,

hitl_review_tool_calls_example.py

hitl_edit_state_example.py



Real World Case Study: Chapter Outline Generator

Goal: Given a slide deck autonomously generate a high quality chapter outline, output it in JSON format.

Preprocessing:

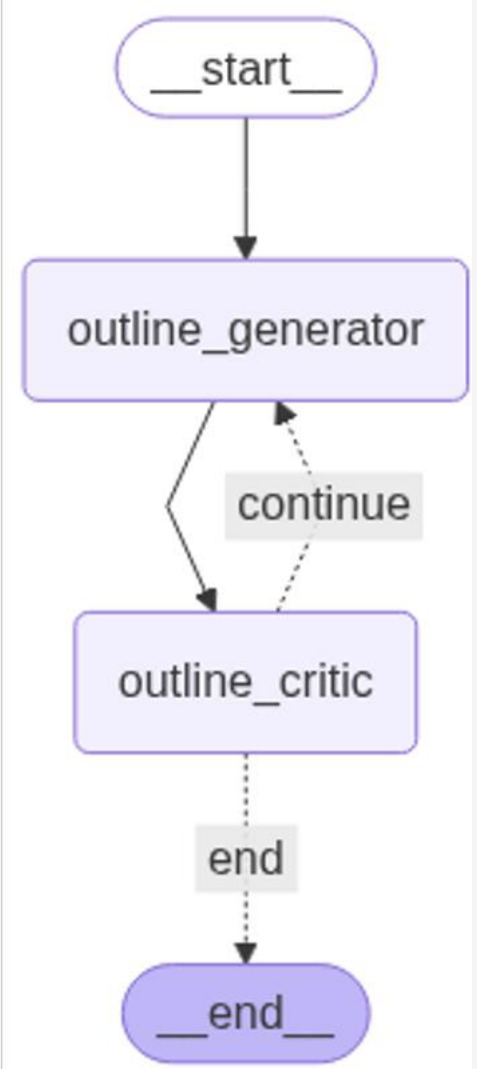
- Extract content from PPT: Uses UnstructuredPowerPointLoader from LangChain

Helper Functions:

- Generate outline for the given contents : create_outline_from_slides
- Critically evaluate the generated outline : outline_critic

Agents in Relection Pattern:

- Outline_generator
- Outline_critic



Real World Case Study: Extending the outline generator to HITL

Goals, Preprocessing, Helper Functions: Same as previous case

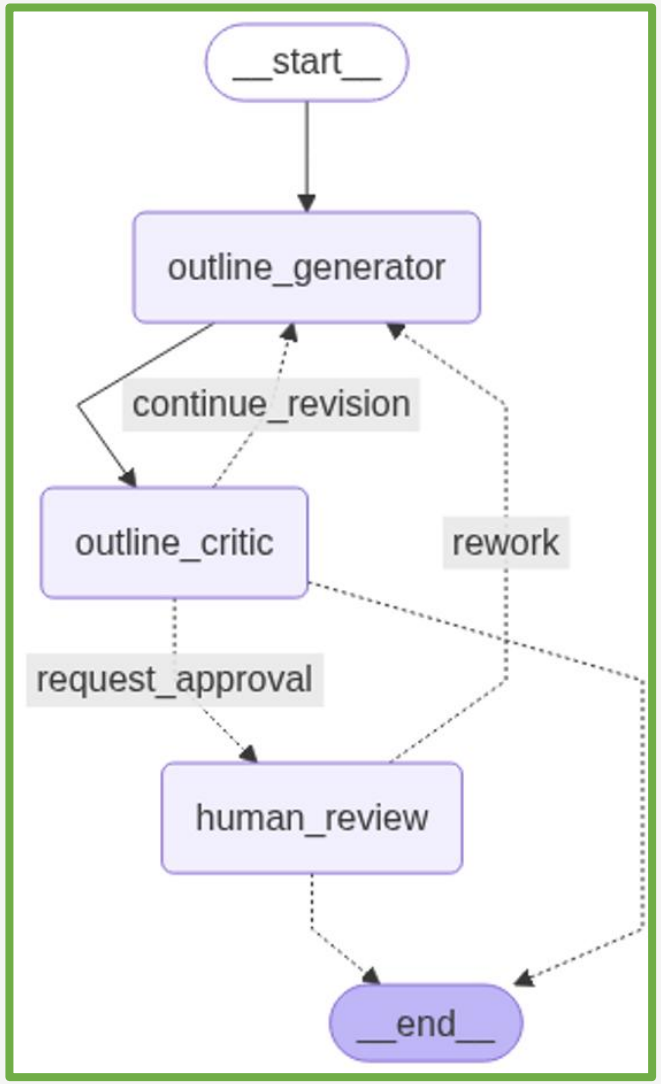
Agents:

- Outline_generator
- Outline_critic
- Human_review

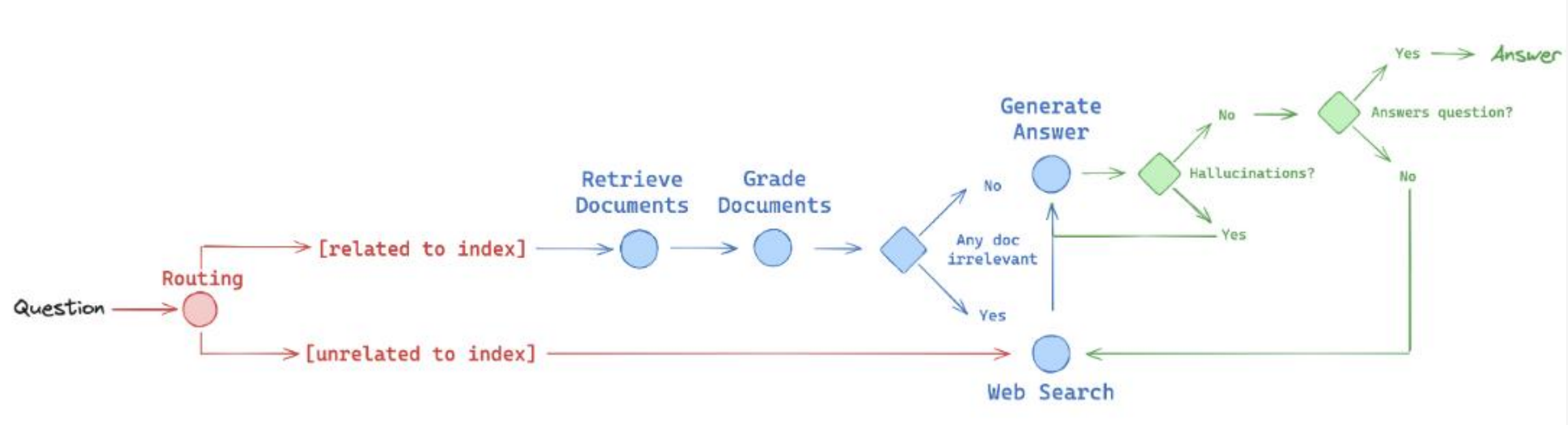
Agentic AI using HITL Pattern:

After completing the generation process that involves outline generator and critic agents, present it to a Human Review. If the human accepts it, return the final outline. Otherwise iterate all over again.

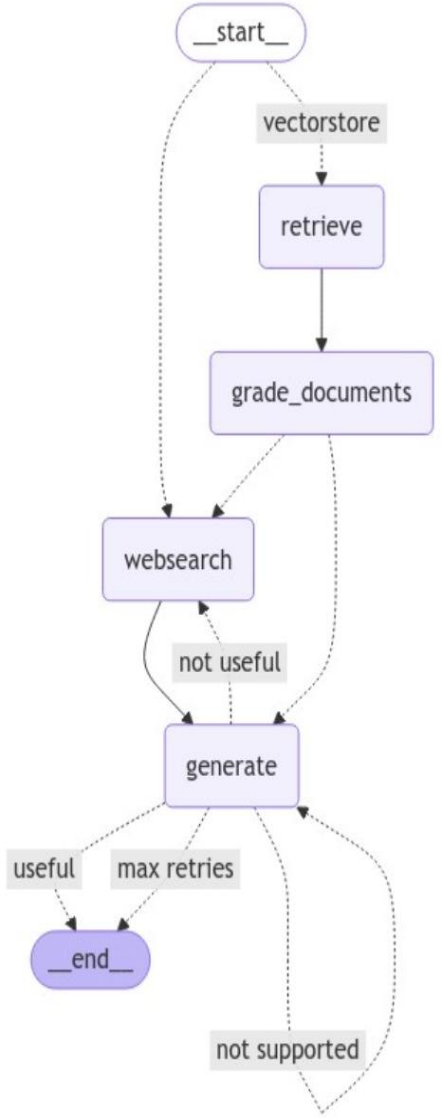
Outline Generator agent is re-entrant from critic as well as human review processes.



RAG Agents – Router Pattern



Graph and Edges



Router determines if the query is answerable by the RAG

Retriever node (tool) retrieves the chunks for the given query

Grader node (agent) evaluates the chunks against the query for relevance (context relevance as in TruLens) and grades them.

Web search node (tool) is invoked if grader doesn't find the context relevant or if the router decided to invoke web search.

Generate node (agent) generates an answer if the context if acceptable or upon web search.

Generated answer is checked for "answer relevance" and if not supported does a retry (up to some limit, say 3 retries)

The workflow finishes either upon useful answer or on exhausting maximum retries.

Recap and Looking Forward

- Today we reviewed core features of LangGraph in detail, contrasted it with other frameworks

- LangGraph is very strong in persistence support, enabling the developer to have fine grained control including editing states through a human process.

- The case studies presented provided implementation examples of several Agentic Patterns.

During the next session, we look forward to:

- A brief discussion on LangSmith

- Advanced Features involving persistence, guardrails etc

- Hands On: Real World project

Thank You!