Integrating Ollama and LangGraph with GraphState for a Time-Aware Agent

1. Introduction: Ollama, LangGraph, GraphState, and the Time-Aware Agent

Ollama has emerged as a significant tool for developers and researchers seeking to run open-source large language models locally.[1] It simplifies the process by bundling the necessary model weights, configuration files, and data into a single package.[1] This approach streamlines the setup process, including the optimization for GPU usage, thus making sophisticated language models more accessible for local experimentation and application development.[1] Ollama supports a diverse range of models, including popular choices like Llama 2, Llama 3, and Mistral, as well as models with multi-modal capabilities.[1] Functioning as a local server, Ollama typically listens for API requests on http://localhost:11434.[8] The ease of deploying LLMs locally through Ollama reflects a growing trend towards edge AI, addressing concerns related to data privacy and consistent access, which is particularly relevant in research environments with strict data handling protocols. The increasing support for multi-modal models within Ollama indicates a future where local AI agents can process various data formats beyond text, potentially leading to more versatile and context-aware applications.

Complementing Ollama, LangGraph, developed by LangChain, is a powerful framework for constructing stateful, multi-actor AI applications that leverage large language models.[8] Drawing inspiration from graph computation models like Pregel and Apache Beam, LangGraph offers a user-friendly interface similar to NetworkX.[8] This framework provides fine-grained control over an application's flow and state, making it well-suited for developing reliable and advanced agents with features such as cyclical processes, enhanced controllability, and the ability to maintain state.[8] The graph-based approach adopted by LangGraph for agent design signifies a move towards a more structured and manageable methodology for handling intricate AI workflows. This offers improved control and observability compared to more traditional programming approaches, which is especially beneficial for researchers investigating complex agent behaviors and interactions.

At the core of LangGraph's architecture is the GraphState, which acts as a central hub for managing the application's state. This state is dynamically updated by the individual processing steps, or nodes, within the graph.[13] The GraphState serves as a shared data structure, capturing the current information snapshot of the application.[14] Nodes within the graph contribute to this state by returning operations that modify its attributes, essentially functioning as a key-value store.[16] These attributes can be either completely replaced with new values or updated incrementally by adding to their existing content.[13] The GraphState is crucial for maintaining context and enabling seamless information exchange between the various components of a LangGraph workflow. This centralized state management is fundamental for building agents capable of remembering past interactions and making contextually relevant decisions throughout their operation.

The user's primary objective is to develop a LangGraph that utilizes Ollama and GraphState to accurately respond to the query "what is the current time." This necessitates the integration of a mechanism for retrieving the current time and incorporating it into the agent's response. Furthermore, the user requires guidance on setting up and integrating a ChromaDB container into this system. The specific requirement to answer time-related queries indicates the need to integrate external tools or functions within the LangGraph workflow to access real-time information. This also implies the need for conditional logic to appropriately route and handle such queries, showcasing LangGraph's capability to integrate external resources and manage specific user intents. While the immediate utility of ChromaDB for a simple time retrieval task might not be evident, its inclusion in the user's requirements suggests a potential for future expansion or more complex functionalities, such as Retrieval Augmented Generation (RAG), where ChromaDB would play a vital role in storing and retrieving relevant knowledge to enhance the agent's capabilities in broader scenarios.

2. LangGraph Architecture and GraphState in Detail

The foundation of any LangGraph application lies in its core components: State, Nodes, and Edges.[11] The State is the central data structure, typically defined using TypedDict or a Pydantic BaseModel, that holds the current snapshot of information relevant to the agent's operation.[11] While other Python types can be used, TypedDict or Pydantic models are recommended for their ability to enforce a clear schema, ensuring data integrity and facilitating better type checking and validation.[11] The State is dynamic and evolves as the graph executes, with different nodes updating its attributes based on their specific tasks.[13] This flexibility in defining the State's schema allows developers to precisely tailor the information tracked by the LangGraph to the specific needs of their AI agent, enabling the management of diverse and complex data structures relevant to the application domain.

Nodes represent the individual processing steps or actions within the LangGraph workflow.[11] These are implemented as Python functions or LangChain Expression Language (LCEL) runnables that accept the current State as input.[11] Each node performs a specific computation or side effect, such as interacting with an LLM, an external tool, or another API, and returns an updated version of the State.[11] The modularity inherent in defining each processing step as a distinct node promotes code reusability and simplifies the development and debugging process for complex AI agents, allowing for more effective isolation and testing of individual workflow components.

Edges define the connections and flow of information between the nodes in the LangGraph.[11] Implemented as Python functions or LCEL runnables, they determine which Node to execute next based on the current State.[11] Edges can be fixed transitions (normal edges) that directly link one node to another or conditional branches (conditional edges) that use functions to evaluate the current State and dynamically determine the subsequent node to execute.[11] The flexibility of defining both fixed and conditional edges provides a robust mechanism for orchestrating complex agent behaviors, with conditional edges being particularly important for implementing dynamic decision-making processes that allow the agent to adapt its actions based on user input or intermediate computation results. LangGraph also defines two special nodes: the START node, which marks the entry point of the workflow [11], and the END node, which signifies the termination point.[11]

The GraphState is initialized when a StateGraph object is created, requiring an initial state definition that specifies the attributes the state will manage and their corresponding data types.[13] Nodes within the LangGraph update the GraphState by returning dictionaries where the keys correspond to the state attributes and the values represent the new or modified data.[13] For more complex state management, LangGraph offers the concept of reducers – functions that define how state updates are applied, especially useful for list-like attributes where updates might involve appending, extending, or modifying specific elements.[17] LangGraph includes a built-in add_messages reducer specifically designed for conveniently managing lists of message objects, facilitating the tracking of conversational history.[14] If no reducer is explicitly defined for a state attribute, the default behavior is to completely overwrite the existing value with the new update.[17] State reducers provide a granular level of control over how the GraphState is manipulated by different nodes, enabling the implementation of sophisticated data management strategies and ensuring consistency in state updates, particularly when dealing with complex data structures.

A common practice in building conversational AI agents with LangGraph is to manage the conversation history as a list of BaseMessage objects (e.g., HumanMessage, AIMessage) within the GraphState.[14] The MessagesState class provides a specialized and convenient way to define a state schema that primarily focuses on managing this message history.[14] This dedicated support for managing messages within the

GraphState, including the MessagesState class, highlights LangGraph's suitability for developing conversational AI applications that require maintaining context and history across multiple turns of interaction. Once the nodes and edges of the LangGraph have been defined, the StateGraph must be compiled into a LangChain Runnable using the compile() method.[12] This compilation process transforms the graph structure into an executable object that can then be invoked with an initial state to begin the agent's workflow.[12] The compilation of the LangGraph into a LangChain Runnable facilitates seamless integration with the broader LangChain ecosystem, allowing developers to leverage other LangChain components and utilities within their agent workflows, enhancing the flexibility and power of LangGraph in building complex AI applications.

3. Integrating Ollama as the Language Model

The first step in utilizing Ollama within LangGraph is to download and install the Ollama application from its official website.[1] After installation, the desired large language model can be downloaded using the command ollama pull <model_name> (for example, ollama pull llama3).[1] To view a list of the models currently downloaded, the command ollama list can be used.[1] Furthermore, it's possible to interact with a downloaded model directly from the terminal using ollama run <model_name>.[2] By default, the Ollama server operates on the address http://localhost:11434.[8]

To integrate Ollama with LangChain and LangGraph, the langchain-ollama package needs to be installed using pip: pip install -qU langchain-ollama.[1] Once installed, the ChatOllama class can be imported from the langchain_ollama module.[1] To utilize a specific Ollama model, an instance of the ChatOllama class is created, providing the model name as a parameter (e.g., ChatOllama(model="llama3")).[1] The model's behavior can be further configured by setting other parameters during instantiation, such as temperature, num_predict, and top_p.[1] The ChatOllama class within the langchain-ollama package offers a simplified and intuitive way to interact with locally hosted LLMs managed by Ollama within the LangChain and LangGraph framework, abstracting away the complexities of network communication and API details.

To use Ollama within a LangGraph workflow, a LangGraph node needs to be created that accepts the GraphState as input. Inside this node, an instance of the ChatOllama class is instantiated with the desired model specified. Then, the message(s) to be sent to the LLM are constructed, typically by retrieving relevant information from the current GraphState. Finally, the invoke() or ainvoke() method of the ChatOllama instance is used to obtain the LLM's response.[1] The response from the LLM is then used to update the GraphState.

Certain models served by Ollama offer native support for tool calling.[1] This allows the LLM to determine when it needs to invoke external functions to fulfill a user's request. To define a tool in LangChain, the @tool decorator from the langchain_core.tools module can be used.[1] Once tools are defined, they can be bound to a ChatOllama instance using the bind_tools() method.[1] When the LLM identifies the need to use a tool, it will output a structured call to that tool. The integration of tool calling capabilities within Ollama, when combined with LangGraph, enables the development of more advanced and autonomous AI agents that can interact with the real world by invoking external functions based on the LLM's reasoning and the user's instructions.

4. Building the Core LangGraph Workflow

For the specific task of creating a time-aware agent, the GraphState should, at a minimum, include the history of messages exchanged with the user and the user's current input. It may also be beneficial to include a flag to indicate if the user's query is specifically asking for the time and a field to store the retrieved current time. An example of such a GraphState definition using TypedDict is:

Python

```python
from typing import TypedDict, List
from langchain_core.messages import BaseMessage

class AgentState(TypedDict):
    messages: List
    user_input: str
    is_time_query: bool
    current_time: str
```

The LangGraph workflow will require nodes to handle user input and interact with the language model. An **Input Node** will be responsible for receiving the user's input and updating the user_input field within the GraphState. It might also initialize other relevant fields in the state. The **LLM Node**, as discussed in the previous section, will take the current messages from the GraphState, append the user_input, send the updated message list to the configured Ollama model, and then update the messages in the GraphState with the LLM's response.

To begin, a basic sequential workflow can be implemented where the user's input is processed, then passed to the LLM, and the response is generated. This can be achieved using StateGraph to define the graph, add_node() to add the input and LLM processing nodes, and add_edge() to connect them in the desired sequence (Input Node -> LLM Node -> End). The starting node for the graph needs to be set using set_entry_point(). Finally, the graph is compiled into a Runnable using compile(). Starting with a simplified sequential workflow allows for a foundational understanding of the interaction between the core components (input handling and LLM interaction) before introducing more complex logic for tool use and conditional branching. This iterative development approach makes it easier to manage complexity and debug the initial setup.

5. Implementing the Current Time Retrieval Tool

To retrieve the current time, a Python function utilizing the datetime module can be defined. This function will get the current date and time and format it into a user-friendly string. An example of such a function is:

Python

```python
from datetime import datetime

def get_current_time():
    now = datetime.now()
    current_time = now.strftime("%Y-%m-%d %H:%M:%S")
    return current_time
```

While a direct function call is possible, it is often cleaner and more aligned with LangChain's principles to wrap the time retrieval function as a LangChain tool using the @tool decorator.[1] This involves providing a descriptive string for the tool that the LLM can understand. An example of defining the get_current_time function as a LangChain tool is:

Python

```python
from langchain_core.tools import tool
from datetime import datetime

@tool
def get_current_time():
    """Returns the current date and time."""
    now = datetime.now()
    current_time = now.strftime("%Y-%m-%d %H:%M:%S")
    return current_time
```

If the tool approach is used, a dedicated LangGraph node will be needed to handle the invocation of the get_current_time tool. This node would typically check if the LLM's response indicates a call to this specific tool. If a tool call is detected, the node will execute the tool and update the GraphState with the tool's output (the current time). However, for this specific scenario, a simpler approach might involve directly calling the get_current_time() function within a LangGraph node that is triggered by the conditional logic identifying a time-related query. This avoids the overhead of formal tool definition and invocation for a relatively straightforward task. Regardless of whether a tool or a direct function call is used, the LangGraph node responsible for retrieving the current time must update the current_time field in the GraphState with the obtained value. Additionally, it should likely format a user-friendly response message containing the current time and update the messages list in the state with this response.

6. Conditional Node for Time Queries

To handle the specific query about the current time, a conditional function is required. This function will take the current GraphState as input and examine the user_input field. It should contain logic to identify if the user's input is a request for the current time, looking for keywords or patterns like "what is the time" or "current time" (case-insensitive). Based on this check, the function should return a string that indicates which node in the LangGraph should be executed next (e.g., "get_time" if it's a time query, or "continue_normal" otherwise). An example of such a conditional function is:

Python

```
def should_handle_time_query(state):
    user_input = state.get("user_input", "").lower()
    if "what is the time" in user_input or "current time" in user_input:
        return "get_time"
    else:
        return "continue_normal"
```

Once the conditional function is defined, it needs to be integrated into the LangGraph using add_conditional_edges().[11] This involves specifying the node from which the conditional logic should be applied (likely the input processing node), the conditional function itself, and a mapping that dictates which node to transition to based on the output of the conditional function. A conceptual example of the resulting graph structure is:

Python

```
graph_builder = StateGraph(AgentState)
graph_builder.add_node("input", input_node)
graph_builder.add_node("llm", llm_node)
graph_builder.add_node("get_time", get_time_node) # Node to fetch current time
graph_builder.set_entry_point("input")
graph_builder.add_conditional_edges(
    "input",
    should_handle_time_query,
    {
        "get_time": "get_time",
        "continue_normal": "llm",
    },
)
graph_builder.add_edge("get_time", "end")
graph_builder.add_edge("llm", "end")
graph = graph_builder.compile()
```

The get_time_node is the LangGraph node that will be executed when the conditional function determines that the user is asking for the current time. This node will contain the logic to call the get_current_time() function (or invoke the get_current_time tool if that approach is taken). After retrieving the current time, this node will update the current_time field in the GraphState and also format a response message to the user, including the current time, which will be added to the messages list in the state.

7. Setting Up and Running ChromaDB Container

Before integrating ChromaDB, ensure that Docker is installed and running on your system. To download the official ChromaDB Docker image, open your terminal or command prompt and execute the command: docker pull chromadb/chroma. Once the image is downloaded, you can run a ChromaDB container using the docker run command. For a basic setup, you can map the default ChromaDB port (8000) to a port on your host machine. Consider using a persistent volume if you need to preserve the ChromaDB data across container restarts. A basic docker run command is:

Bash

```bash
docker run -d --name chromadb -p 8000:8000 chromadb/chroma
```

In this command, -d runs the container in detached mode (in the background), --name chromadb assigns the name "chromadb" to the container, -p 8000:8000 maps port 8000 on your host machine to port 8000 inside the container, and chromadb/chroma specifies the Docker image to use. After running this command, you can verify that the ChromaDB container is up and running by using the command docker ps. This will list all running Docker containers, and you should see an entry for the "chromadb" container with a status of "Up". ChromaDB, when run in a container, can be accessed via its HTTP API on the port you specified (default is 8000). You can also use Python client libraries like the chromadb package to interact with the database programmatically from your LangGraph application. Containerizing ChromaDB using Docker provides a consistent, isolated, and easily reproducible environment for the vector database, simplifying its deployment and management, which is particularly beneficial in research and development settings where environment consistency is crucial.

8. Connecting LangGraph to ChromaDB (Potential Use Cases)

The primary use case for ChromaDB in the context of LangGraph is likely to be Retrieval Augmented Generation (RAG), where ChromaDB serves as a vector store for embedding and retrieving relevant documents based on user queries.[8] LangGraph can be designed to incorporate nodes that interact with ChromaDB to fetch these relevant documents, which can then be used to augment the responses generated by the LLM. To integrate ChromaDB into a LangGraph workflow, a specific LangGraph node would be created to connect to the running ChromaDB instance using a ChromaDB client library. This node would then take the user's query (or a refined version of it) from the GraphState and perform a vector similarity search against the documents stored in ChromaDB. The retrieved documents would then be added to the GraphState.

Consider a scenario where the user asks a question about a specific topic. The LangGraph could employ a conditional node to identify this topic-related query. It would then route the flow to a dedicated retrieval node that interacts with ChromaDB to find relevant documents. These documents, along with the original query, would then be passed to an LLM node to generate a more informed and contextually accurate answer. However, for the specific and relatively simple task of retrieving the current time, ChromaDB is not directly applicable or necessary, as the time can be obtained through a standard Python function call. The user might consider future enhancements where historical time-related queries and their corresponding responses are stored in ChromaDB. This could enable the agent to learn patterns in time-related queries or perform analysis on past interactions involving time, which would involve embedding the queries and storing them in ChromaDB for later retrieval and analysis. While ChromaDB is not directly utilized in the immediate task of time retrieval, its integration into the LangGraph ecosystem provides a foundation for implementing more advanced functionalities, particularly in the realm of knowledge retrieval and augmentation, allowing the agent to access and leverage external information sources to enhance its responses to a broader range of user inquiries in the future.

9. End-to-End Implementation Example

Python

```python
from typing import TypedDict, List
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_ollama import ChatOllama
from langgraph.graph import StateGraph, END
from datetime import datetime
```

```
# 1. Define the GraphState
class AgentState(TypedDict):
  messages: List
  user_input: str
  current_time: str

# 2. Set up Ollama
ollama_llm = ChatOllama(model="llama3")

# 3. Implement the time retrieval function
def get_current_time_func():
  now = datetime.now()
  current_time = now.strftime("%Y-%m-%d %H:%M:%S")
  return current_time

# 4. Create LangGraph nodes
def user_input_node(state: AgentState):
  return {"messages": state['messages'] + [HumanMessage(content=state['user_input'])]}

def llm_node(state: AgentState):
  response = ollama_llm.invoke(state['messages'])
  return {"messages": state['messages'] + [response], "current_time": state['current_time']}

def get_time_node(state: AgentState):
  current_time = get_current_time_func()
  response = AIMessage(content=f"The current time is: {current_time}")
  return {"messages": state['messages'] + [response], "current_time": current_time}

# 5. Define the conditional function
def should_handle_time_query(state: AgentState):
  user_input = state.get("user_input", "").lower()
  if "what is the current time" in user_input or "what time is it" in user_input:
    return "get_time"
  else:
    return "llm"

# 6. Construct the StateGraph
builder = StateGraph(AgentState)
builder.add_node("user_input", user_input_node)
builder.add_node("llm", llm_node)
builder.add_node("get_time", get_time_node)
builder.set_entry_point("user_input")

builder.add_conditional_edges(
    "user_input",
    should_handle_time_query,
    {
        "get_time": "get_time",
        "llm": "llm",
    },
)

builder.add_edge("get_time", END)
builder.add_edge("llm", END)

graph = builder.compile()

# 7. Run the graph
inputs = {"messages":, "user_input": "what is the current time"}
result = graph.invoke(inputs)
print(result['messages'][-1].content)

inputs_normal = {"messages":, "user_input": "Tell me a joke"}
result_normal = graph.invoke(inputs_normal)
print(result_normal['messages'][-1].content)

# Instructions for running ChromaDB container:
# docker run -d --name chromadb -p 8000:8000 chromadb/chroma

# Basic example of connecting to ChromaDB (for demonstration purposes):
# import chromadb
# client = chromadb.HttpClient(host="localhost", port=8000)
# print(client.heartbeat())
```

This example defines the AgentState, sets up the ChatOllama client, implements the get_current_time_func, and creates LangGraph nodes for user input, LLM interaction, and time retrieval. The should_handle_time_query function routes the workflow based on the user's input. The StateGraph is constructed with these nodes and conditional edges, and then invoked with a time-related query and a normal query to demonstrate its functionality. The code also includes instructions for running the ChromaDB container using Docker and a basic example of connecting to it using the chromadb Python client. This end-to-end example provides a practical template for integrating Ollama, LangGraph, and conditional logic to create a time-aware agent.

10. Advanced Concepts and Considerations

LangGraph provides mechanisms for more intricate state management through state reducers, which allow for custom logic in how state updates are applied, going beyond simple overwriting.[17] This is particularly useful for managing complex data structures within the GraphState. For applications requiring human oversight, LangGraph's stateful nature and the ability to insert breakpoints into the workflow enable the creation of human-in-the-loop systems where human review or intervention can occur at specific stages, enhancing control and reliability.[27] To build agents that can maintain context over longer periods or across multiple sessions, LangGraph offers options for persisting the GraphState using databases or built-in checkpointing mechanisms, allowing agents to remember past interactions.[17] Ensuring the robustness of LangGraph applications involves implementing effective error handling within the nodes and configuring retry mechanisms for operations that might be unreliable, thus improving the agent's stability.[33] For deploying LangGraph applications at scale, considerations around handling increased user loads and managing complex workflows become important, with platforms like LangGraph Platform and self-hosted solutions offering potential deployment strategies.[34] Finally, for monitoring the performance of LangGraph agents, debugging issues, and gaining insights into their behavior during execution, tools like LangSmith are invaluable.[1] Understanding these advanced concepts empowers users to build more sophisticated, reliable, and scalable AI agent applications using LangGraph.

11. Conclusion

Integrating Ollama as a local language model within the LangGraph framework, while leveraging GraphState for state management and incorporating conditional logic, provides a robust foundation for creating sophisticated AI agents, as demonstrated with the time-aware agent example. The potential integration of ChromaDB into this architecture opens up possibilities for even more advanced functionalities, such as Retrieval Augmented Generation, which can significantly enhance the agent's ability to handle a wider range of knowledge-based queries. LangGraph's flexibility and power make it a compelling framework for building complex and stateful AI agent workflows using locally hosted large language models via Ollama. Further exploration of the advanced features and capabilities of both LangGraph and Ollama will undoubtedly unlock even more sophisticated AI application development possibilities.

Table: Ollama and ChromaDB Setup Commands

Command	Description
docker pull chromadb/chroma	Downloads the ChromaDB Docker image.
docker run -d --name chromadb -p 8000:8000 chromadb/chroma	Runs the ChromaDB container.
ollama pull <model_name>	Downloads a specific LLM model from Ollama.
ollama list	Lists the Ollama models currently available on your local system.
ollama run <model_name>	Starts an interactive session with a specified Ollama model in your terminal.
pip install -qU langchain-ollama	Installs the necessary LangChain integration for Ollama.

Table: Key Parameters for ChatOllama Instantiation

Parameter	Description	Example Value
model	Specifies the name of the Ollama model to use.	"llama3"
temperature	Controls the randomness of the model's output (0.0 for deterministic).	0.7
num_predict	The maximum number of tokens to predict when generating text.	256
top_p	Controls the nucleus sampling; considers tokens whose probability exceeds top_p.	0.9

base_url	The URL of the Ollama server.	"http://localhost:11434"

Works cited

1. ChatOllama - LangChain, accessed on April 13, 2025, https://python.langchain.com/docs/integrations/chat/ollama/
2. Ollama - LangChain, accessed on April 13, 2025, https://python.langchain.com/v0.1/docs/integrations/llms/ollama/
3. ChatOllama - LangChain.js, accessed on April 13, 2025, https://js.langchain.com/v0.1/docs/integrations/chat/ollama/
4. ChatOllama - LangChain.js, accessed on April 13, 2025, https://js.langchain.com/docs/integrations/chat/ollama
5. langchain/docs/docs/integrations/chat/ollama.ipynb at master - GitHub, accessed on April 13, 2025, https://github.com/langchain-ai/langchain/blob/master/docs/docs/integrations/chat/ollama.ipynb
6. ChatOllama — LangChain documentation, accessed on April 13, 2025, https://python.langchain.com/api_reference/community/chat_models/langchain_community.chat_models.ollama.ChatOllama.html
7. ChatOllama — LangChain documentation, accessed on April 13, 2025, https://python.langchain.com/api_reference/ollama/chat_models/langchain_ollama.chat_models.ChatOllama.html
8. Building Local AI Agents: A Guide to LangGraph, AI Agents, and Ollama | DigitalOcean, accessed on April 13, 2025, https://www.digitalocean.com/community/tutorials/local-ai-agents-with-langgraph-and-ollama
9. Why my LLM could not fill tool_calls field? · langchain-ai langgraph · Discussion #3284, accessed on April 13, 2025, https://github.com/langchain-ai/langgraph/discussions/3284
10. Instead of OpenAI, Can I use llamaIndex and Ollama with Langgraph - DeepLearning.AI, accessed on April 13, 2025, https://community.deeplearning.ai/t/instead-of-openai-can-i-use-llamaindex-and-ollama-with-langgraph/708171
11. Quick Primer on LangGraph - PolarSPARC, accessed on April 13, 2025, https://www.polarsparc.com/xhtml/LangGraph.html
12. LangGraph: Build Stateful AI Agents in Python, accessed on April 13, 2025, https://realpython.com/langgraph-python/
13. LangGraph Simplified - Kaggle, accessed on April 13, 2025, https://www.kaggle.com/code/marcinrutecki/langgraph-simplified
14. LangGraph Tutorial with Practical Example - Getting Started with Artificial Intelligence, accessed on April 13, 2025, https://www.gettingstarted.ai/langgraph-tutorial-with-example/
15. Llama 3.1 Agent using LangGraph and Ollama - Pinecone, accessed on April 13, 2025, https://www.pinecone.io/learn/langgraph-ollama-llama/

16. LangGraph - LangChain Blog, accessed on April 13, 2025, https://blog.langchain.dev/langgraph/
17. LangGraph Glossary - GitHub Pages, accessed on April 13, 2025, https://langchain-ai.github.io/langgraph/concepts/low_level/
18. LangGraph state - YouTube, accessed on April 13, 2025, https://www.youtube.com/watch?v=DBXdE_5Jces
19. Langgraph adaptive rag local - GitHub Pages, accessed on April 13, 2025, https://langchain-ai.github.io/langgraph/tutorials/rag/langgraph_adaptive_rag_local/
20. langgraph/examples/rag/langgraph_crag_local.ipynb at main - GitHub, accessed on April 13, 2025, https://github.com/langchain-ai/langgraph/blob/main/examples/rag/langgraph_crag_local.ipynb
21. Self-RAG using local LLMs - GitHub Pages, accessed on April 13, 2025, https://langchain-ai.github.io/langgraph/tutorials/rag/langgraph_self_rag_local/
22. How to update graph state from nodes - GitHub Pages, accessed on April 13, 2025, https://langchain-ai.github.io/langgraph/how-tos/state-reducers/
23. langgraph/examples/state-model.ipynb at main - GitHub, accessed on April 13, 2025, https://github.com/langchain-ai/langgraph/blob/main/examples/state-model.ipynb
24. Nodes, Edges, States & Graph in LangGraph — Basics | by Jayasree M - Medium, accessed on April 13, 2025, https://medium.com/@official.hardcodeconcepts/nodes-edges-states-graph-in-langgraph-basics-3bdc7e9954b6
25. Getting started with LangGraph #langgraph #nodes #edges - YouTube, accessed on April 13, 2025, https://www.youtube.com/watch?v=lVQynpvI9Ek
26. LangChain-LangGraph-Ollama-Tools/README.md at main - GitHub, accessed on April 13, 2025, https://github.com/chetan25/LangChain-LangGraph-Ollama-Tools/blob/main/README.md
27. How to view and update past graph state - GitHub Pages, accessed on April 13, 2025, https://langchain-ai.github.io/langgraph/how-tos/human_in_the_loop/time-travel/
28. LangGraph Agents - Human-in-the-Loop: Editing Graph State - YouTube, accessed on April 13, 2025, https://www.youtube.com/watch?v=ndPrCjRCSGo
29. python - langchain ollama module difference - Stack Overflow, accessed on April 13, 2025, https://stackoverflow.com/questions/78921530/langchain-ollama-module-difference
30. ChatOllama — LangChain documentation, accessed on April 13, 2025, https://api.python.langchain.com/en/latest/ollama/chat_models/langchain_ollama.chat_models.ChatOllama.html
31. langchain_community.chat_models.ollama.ChatOllama — LangChain 0.2.17, accessed on April 13, 2025, https://api.python.langchain.com/en/latest/chat_models/langchain_community.ch

at_models.ollama.ChatOllama.html
32. langgraph/examples/rag/langgraph_crag.ipynb at main - GitHub, accessed on April 13, 2025, https://github.com/langchain-ai/langgraph/blob/main/examples/rag/langgraph_crag.ipynb
33. Enhanced state management & retries in LangGraph Python - LangChain - Changelog, accessed on April 13, 2025, https://changelog.langchain.com/announcements/enhanced-state-management-retries-in-langgraph-python
34. LangGraph - LangChain, accessed on April 13, 2025, https://www.langchain.com/langgraph
35. Building a fully local research assistant from scratch with Ollama - YouTube, accessed on April 13, 2025, https://www.youtube.com/watch?v=XGuTzHoqlj8
36. langchain-ai/langgraph: Build resilient language agents as graphs. - GitHub, accessed on April 13, 2025, https://github.com/langchain-ai/langgraph