Why LangGraph Stands Out as an Exceptional Agent Framework

Hao Lin · Follow

6 min read · Mar 11, 2024

Image sourced from the official LangGraph page

The recent post on the Berkeley Artificial Intelligence Research (BAIR) Lab blog, "The Shift from Models to Compound AI Systems," provides an insightful review on Compound AI Systems. These systems address intricate tasks by leveraging a variety of interacting components, such as multiple model calls, information retrievers, and external tools. This multifaceted approach is becoming increasingly preferred for developing sophisticated virtual assistants, copilots, and AI agents for several reasons:

Enhanced Task Performance through System Design: While Large Language Models (LLMs) improve with larger scale and more compute power, system design can offer better return on investment and faster improvements. For example, engineering a system to sample and test multiple solutions from a model can significantly boost performance, as evidenced by projects like AlphaCode 2, making it more reliable for applications like coding contests.

Enhanced Dynamics: LLMs, trained on static datasets, have fixed "knowledge." Compound systems can incorporate real-time and managed data through additional components like search, retrieval, and access controls, enabling up-to-date responses and applications with nuanced access management.

Improved Control and Trust: Controlling the behavior of LLMs is challenging. Compound systems allow for tighter control over outputs, such as filtering,

providing citations, explaining reasoning, and performing fact-checking. This reduces model hallucination and increases user trust.

Flexible Performance Goals: The fixed quality level and cost of individual AI models may not meet all application needs. Compound systems enable the use of both large and small models for different tasks to balance performance and cost effectively. This approach is adept at serving a broad spectrum of needs, from those emphasizing cost efficiency to scenarios requiring premium outputs.

. . .

As AI continues to evolve, I believe compound systems are set to become a leading paradigm. Now you might wonder which frameworks are optimal for constructing such compound systems. In my role, while developing a virtual assistant, I delved into various frameworks to find the one that best suited our needs. Following an extensive evaluation and practical application, LangGraph emerged as my top choice and here's why.

1. Advanced Workflow Control

LangGraph, built upon LangChain, structures applications through a network of nodes linked by either normal or conditional edges. Nodes, essentially functional units, can employ LLMs, conventional machine learning models, or pure code logic. A normal edge always directs the output of one node (the source) to another (the target). However, with conditional edges connecting a source node to multiple targets, the flow depends on the source's output, determining the path dynamically at runtime.

This architecture facilitates complex decision-making. Consider a LangGraph application comprising three nodes: an Orchestrator node utilizing an LLM for function calls, a Python node generating code via a specialized LLM, and a Validation node that evaluates the Python code and flags any errors. The application begins with the Orchestrator node, which processes user requests and depending on the need for code generation, activates the Python node through a conditional connection. The output from the Python node then moves to the Validation node via a normal edge. If the code contains errors, a conditional edge routes the invalid code and error information back to the Python node for correction, establishing a loop for refining the code. Conversely, if the code is

error-free or exceeds the attempt limit for corrections, either the successful code or all the generated codes and associated validation feedback are sent to the Orchestrator node. This node then finalizes the response to the user, offering the best possible solution along with an explanation of any issues encountered. This workflow ensures that the application not only delivers responses of high quality but also optimizes for efficiency and enhances the user experience.

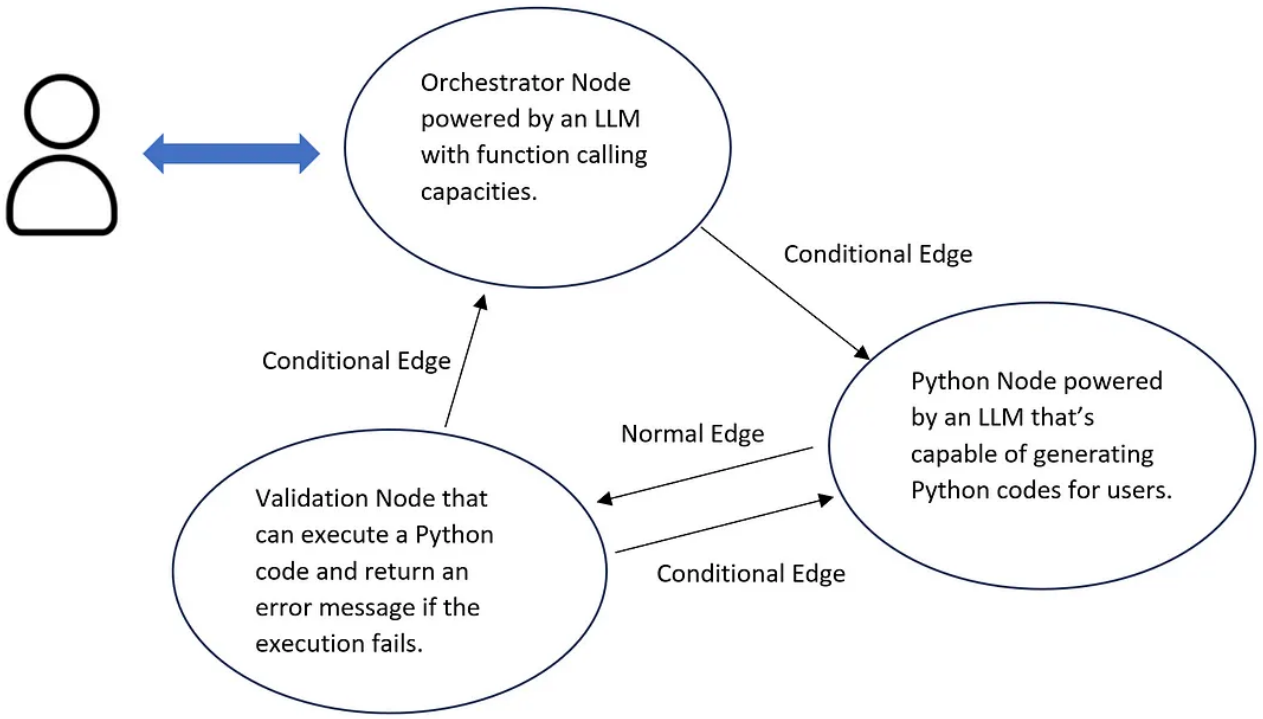


Image provided by the author

2. Enhanced Function Calling Control

LLM function calling capacity is an impressive advancement in AI, enhancing an agent's ability to plan and execute tasks. Typically, developers annotate tools with clear descriptions and define input requirements using Pydantic, alongside providing system prompts to LLMs for in-context learning or few-shot guidance. This helps direct LLMs on when and how to utilize each tool. Despite these efforts to refine LLM function calling, concerns remain about potential mismatches in tool selection or inadequate inputs. This is akin to how, in Retrieval-Augmented Generation (RAG), the effectiveness of a semantic search depends heavily on how well the search query is formulated.

LangGraph's stateful design helps address these issues, which involves passing around a "state" object among the nodes in the graph as they activate. This object, which records all messages exchanged up to that point, is then updated based on

the operations returned by each node. These updates can either replace specific state attributes (e.g., overwriting existing messages) or add new information to them (e.g., appending new messages).

As the LLM assesses a task and selects a tool to use, it adds an AIMessage into the state object. This message contains no content, only function calling parameters including the tool name and input arguments. Before the selected tool is executed, developers have the flexibility to review the current state (the complete message history so far) to verify the tool choice and tool inputs crafted by the LLM. If needed, they can overwrite the tool name or refine the inputs using methods such as term normalization or the HyDE technique for query transformation before processing, ensuring more accurate and effective tool utilization.

3. Flexible Message Types and Parameters

LangGraph also provides a versatile way to handle message types and parameters. Each node can produce messages of any type: AI messages, function messages, or system messages. These messages can also include a variety of parameters beyond just the basic 'content' parameter. Here's why this flexibility is beneficial:

In the self-correction scenario mentioned above, sending erroneous code back to the Python node as a system message, rather than a function message, might significantly affect the correction quality by the LLM within that node. Furthermore, LangGraph facilitates the streaming of node outputs, including the streaming of LLM tokens for nodes incorporating LLMs. This feature, combined with the ability to customize output parameters, is particularly valuable for user interfaces designed to separately present reasoning steps, answers, and explanations. By effectively organizing and streaming data, artifacts, or "breadcrumbs" within your output parameters, you're able to significantly enhance the user experience, offering detailed and timely information as it becomes available.

4. Efficient State and LLM Token Management

While I anticipate that LLMs will eventually overcome the current limitations on context windows, providing only pertinent information as inputs remains a key strategy for enhancing LLM response quality and optimizing costs. Within LangGraph, thanks to its architecture where each node can access the entire

current state (e.g., the accumulated list of chat messages), developers can precisely customize the message list forwarded to LLMs. This customization might involve trimming unnecessary messages or focusing on the most recent messages. For instance, in RAG, once an LLM has produced a response based on the raw retrieved data, this data can be removed from the state before the next LLM request. This streamlined approach ensures both the optimization of resource use and the maintenance of high-quality interactions.

Conclusion

LangGraph's advanced workflow control, the ability to ensure quality in function calling, the customization of message types and parameters, and convenient state and LLM token management are just a few reasons why it stood out to me among the rest of agent frameworks. These features not only streamline the development process but also ensure that the end product is flexible, reliable, and user centric.

Furthermore, the integration of human feedback directly into the workflow of a LangGraph-based virtual assistant presents an exciting opportunity. Instead of the conventional offline learning methods, like compiling Preference and Prompt datasets for LLM fine-tuning via RLHF, one can consider leveraging the real-time interaction analysis between humans and the assistant and training an RL agent to, for example, smartly add helpful system prompts to the workflow, influence routing decisions, and oversee function calls to improve user experiences. Additionally, the option to include a complex state object — capturing other user activities from web or mobile applications besides a list of messages — can offer further personalization and enhancement of interactions, bringing virtual assistants to the next level.

Follow me for the latest insights on agent frameworks, generative AI, and best practices in LLM application development!

. . .

References:

The Shift from Models to Compound AI Systems

The BAIR Blog

bair.berkeley.edu

🦜🕸️ **LangGraph | 🦜🔗 Langchain**

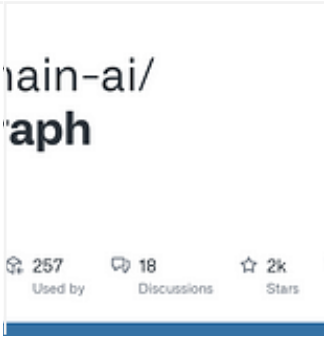
⚡ Building language agents as graphs ⚡

python.langchain.com

langgraph/examples at main · langchain-ai/langgraph

Contribute to langchain-ai/langgraph development by creating an account on GitHub.

github.com



Precise Zero-Shot Dense Retrieval without Relevance Labels

While dense retrieval has been shown effective and efficient across tasks and languages, it remains difficult to create...

arxiv.org

Langgraph AI Agents Framework Llm

Follow

Written by Hao Lin

31 Followers · 15 Following

Senior Machine Learning Engineer sharing insights and resources on mastering big data technologies, machine learning, deep learning, LLMs, and generative AI.