The Pythoneers

Building AI agent systems with LangGraph

Vishnu Sivan Follow 11 min read · Oct 22, 2024

Artificial Intelligence (AI) has advanced significantly, and language models (LLMs) are now performing complex tasks and making dynamic decisions. However, the infrastructure supporting these advanced capabilities often struggles to keep pace. Retrieval-augmented generation (RAG) systems, while useful for simple queries, fall short when it comes to managing complex and multi-step processes.

LangGraph is a game-changing library within the LangChain ecosystem designed to streamline the creation of sophisticated AI applications. LangGraph provides a framework for efficiently defining, coordinating, and executing multiple LLM agents (or chains) in a structured, cyclical manner.

In this article, we will explore how LangGraph helps to build robust, scalable and flexible multi-agent systems, making the development of intelligent AI workflows smoother and more efficient.

Getting Started

Table of contents

What is LangGraph

Before LangGraph, the agent executor class in LangChain was the primary tool for building AI agents. It operated by using an agent in a loop to make decisions, execute actions, and log observations. However, this class had limited flexibility, constraining developers to a fixed pattern of tool calling and error handling, making it difficult to build dynamic and adaptable agent runtimes.

LangGraph is an advanced library within the LangChain ecosystem addresses these limitations by introducing cyclic computational capabilities. While LangChain supports Directed Acyclic Graphs (DAGs) for linear workflows, LangGraph enables the creation of cycles, allowing LLM agents to dynamically loop through processes and make decisions based on evolving conditions. This framework empowers developers to build more complex, flexible, and adaptive agent systems.

Overview

- LangGraph is a library built on top of LangChain, designed to create cyclic graphs for LLM-based AI agents.
- It enables cyclic graph topologies for workflows, allowing more flexible and nuanced agent behaviors than linear models.
- Utilizes key elements:
  — Nodes: Represent functions or LangChain runnable items.

— Edges: Define execution and data flow.
— Stateful graphs: Manage persistent data across execution cycles.

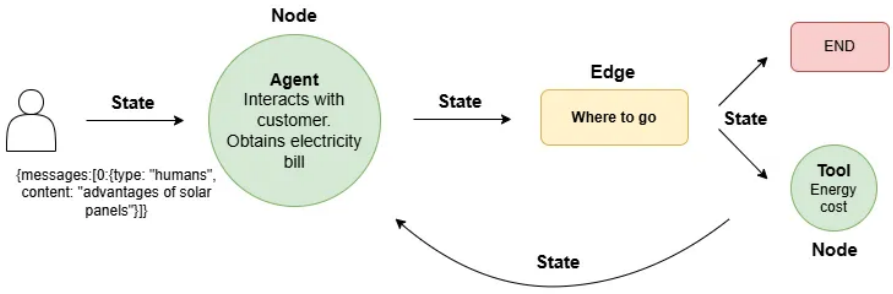
- Supports multi-agent coordination, allowing each agent to have its own prompt, LLM, tools and custom code within a single graph.
- Introduces a chat agent executor that represents agent state as a list of messages, ideal for chat-based models.

Key Components

- **State** maintains and updates the context or memory as the process advances and enabling each step to access relevant information from earlier steps for dynamic decision-making.
- **Nodes** represent individual computation steps or tasks, performing specific functions like data processing, decision-making or system interactions.
- **Edges** connect nodes, dictating the flow of computation and allowing conditional logic to guide the workflow based on the current state, supporting complex and multi-step operations.
- **Stateful graph** is central to its architecture, where each node represents a step in the computation. The graph maintains and updates a shared state as the process advances.

How LangGraph works

LangGraph enables cyclic LLM call execution with state persistence, essential for agentic behavior. LangGraph Inspired by Pregel and Apache Beam and modeled after the NetworkX library for user-friendly graph-based programming offers a more advanced approach to agent runtimes. Unlike its predecessors, which rely on basic loops, LangGraph supports intricate systems of nodes and edges, allowing developers to create more complex decision-making processes and action sequences.



The diagram illustrates how LangGraph facilitates a dynamic, cyclic workflow for AI agents. The process begins with an initial **state**, which contains input data or context (in this case, a message from the customer asking about the advantages of solar panels). This **state** is passed to an **agent** node, which

interacts with the customer to gather information, such as obtaining their electricity bill. After this interaction, the **state** is updated with the gathered information and passed to an **edge**, which represents a decision point. At this point, the system evaluates the state and decides where to proceed next, depending on the updated information. This could lead to a tool interaction or move directly towards an end state.

If the decision is to engage with a **tool**, another **node** is triggered, where the tool (in this case, an energy cost calculator) processes the data. This step involves a function that executes specific operations, again updating the **state** as the workflow continues. Finally, the system can either reach an **end** state, concluding the process, or cycle back, repeating the steps with the updated state if further actions are required. This **feedback loop** allows for ongoing, dynamic decision-making based on evolving conditions, showcasing how LangGraph's cyclic graphs enable flexible and iterative agent behaviors.

Experimenting with LangGraph

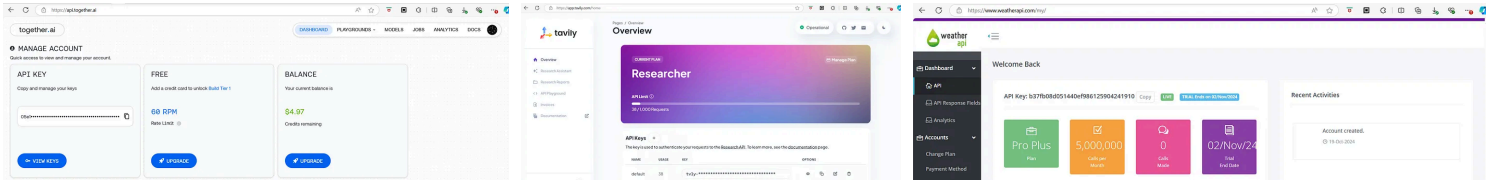
Let's build an agent using LangGraph to gain a deeper understanding. We will start by implementing tool calls, then utilize a pre-built agent, and finally, create our own custom agent within LangGraph.

Pre-requisites

This tutorial is built using OpenAI, the Weather API, Together, and Tavily. If you have an OpenAI API key, you can use that throughout the tutorial, or you can work with other powerful proprietary models such as GPT-4 and Gemini Pro. You can also opt for open-access models such as Mixtral and Llama-3.2. For LLM inferencing, there are several platforms available, such as Abacus, Anyscale and Together.

In this tutorial, we will be using Together AI to infer Llama 3.1, so make sure to obtain an API key from Together before proceeding. Additionally, you will need API keys from WeatherAPI to access weather data and from Tavily to optimize search for AI agents.

- Dashboard — WeatherAPI.com
- api.together.ai
- Tavily AI



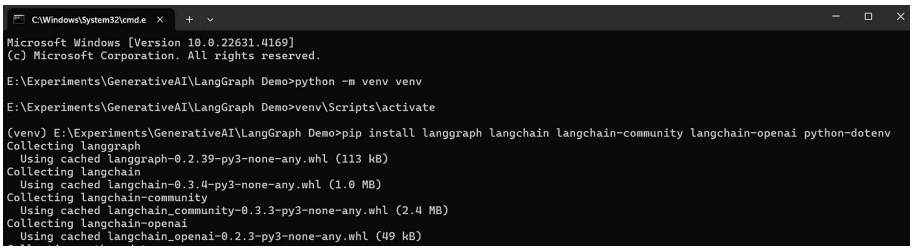
Installing dependencies

- Create and activate a virtual environment by executing the following command.

```
python -m venv venv
source venv/bin/activate #for ubuntu
venv/Scripts/activate #for windows
```

- Install `langgraph` , `langchain` , `langchain-community`, `langchain-openai` and `python-dotenv` libraries using pip.

```
pip install langgraph langchain langchain-community langchain-openai python-dote
```



Setting up the environment variables

- Begin by creating a new folder for your project. Choose a name that reflects the purpose of your project.
- Inside your new project folder, create a file named `.env` . This file will store your environment variables, including your Weather, Together and Tavily API keys.
- Open the `.env` file and add the following code to specify your Gemini API key:

```
WEATHER_API_KEY=b37fb08d051440ef......
TAVILY_API_KEY=tvly-iSIylB0XSj.......
TOGETHER_API_KEY=05e1a66f6........
```

1. Tool Calling in LangGraph

Importing keys

Lets import the API keys to the code.

```
# Import the keys
import os
from dotenv import load_dotenv
load_dotenv('.env')

WEATHER_API_KEY = os.environ['WEATHER_API_KEY']
TAVILY_API_KEY = os.environ['TAVILY_API_KEY']
TOGETHER_API_KEY = os.environ['TOGETHER_API_KEY']
```

Importing necessary libraries

Lets import the necessary libraries and modules needed for the project.

```
# Import the required libraries and methods
import requests
from typing import List, Literal
from langchain_community.tools.tavily_search import TavilySearchResults
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
```

Defining tools

We will create two tools:

- For fetching weather data for weather-related queries.
- For internet searches when the LLM lacks the answer.

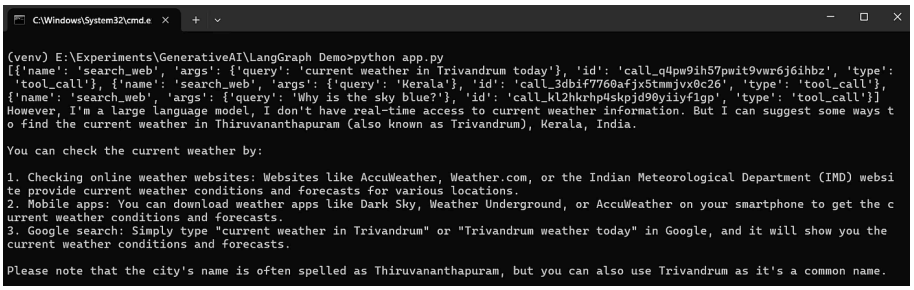
```
@tool
def get_weather(query: str) -> list:
    """Search weatherapi to get the current weather"""
    endpoint = f"http://api.weatherapi.com/v1/current.json?key={WEATHER_API_KEY}
    response = requests.get(endpoint)
    data = response.json()

    if data.get("location"):
        return data
    else:
        return "Weather Data Not Found"

@tool
def search_web(query: str) -> list:
    """Search the web for a query"""
    tavily_search = TavilySearchResults(api_key=TAVILY_API_KEY, max_results=2, s
    results = tavily_search.invoke(query)
    return results
```

To make these tools accessible to the LLM, we can bind them to the model as shown below. In this example, we're using the Together API to utilize the Llama 3 model. If you have an OpenAI key, you can proceed with that by specifying your key instead.

```
llm = ChatOpenAI(base_url="https://api.together.xyz/v1",
    api_key=TOGETHER_API_KEY,
    model="meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo"

# If you have OpenAI key
# llm = ChatOpenAI(model="gpt-4o-mini", api_key="sk-U7tijaa4jwHvhVWGr....", temp

tools = [search_web, get_weather]
llm_with_tools = llm.bind_tools(tools)
```

Now, lets invoke the LLM with a prompt to see the results:

```
prompt = """
    Given only the tools at your disposal, mention tool calls for the following
    Do not change the query given for any search tasks
        1. What is the current weather in Trivandrum today
        2. Can you tell me about Kerala
        3. Why is the sky blue?
    """

results = llm_with_tools.invoke(prompt)

print(results.tool_calls)

query = "What is the current weather in Trivandrum today"
response = llm.invoke(query)
print(response.content)
```



The Llama 3 / GPT model does not have real-time access to current weather information, so it cannot provide up-to-date weather details.

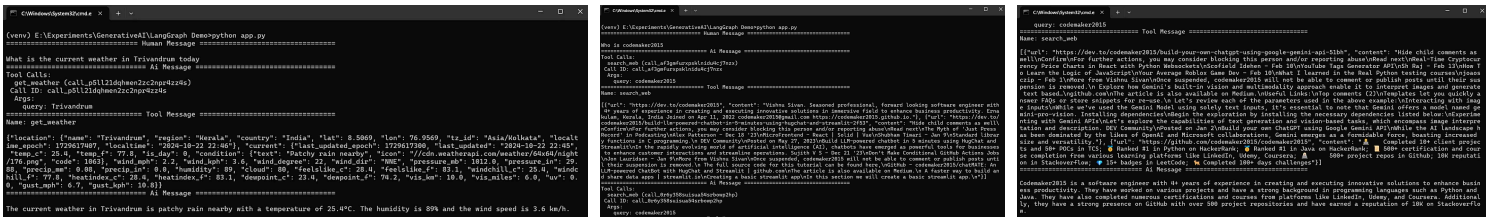
2. Using pre-built agent

LangGraph offers a pre-built React(Reason and Act) agent, designed to streamline decision-making and action execution.

Let's explore how it works.

```
from langgraph.prebuilt import create_react_agent

# system prompt is used to inform the tools available to when to use each
system_prompt = """Act as a helpful assistant.
    Use the tools at your disposal to perform tasks as needed.
        - get_weather: whenever user asks get the weather of a place.
        - search_web: whenever user asks for information on current events or if
    Use the tools only if you don't know the answer.
    """

# we can initialize the agent using the llama3 model, tools, and system prompt.
agent = create_react_agent(model=llm, tools=tools, state_modifier=system_prompt)

# Let's query the agent to see the result.
def print_stream(stream):
    for s in stream:
        message = s["messages"][-1]
        if isinstance(message, tuple):
            print(message)
        else:
            message.pretty_print()

inputs = {"messages": [("user", "What is the current weather in Trivandrum today

print_stream(agent.stream(inputs, stream_mode="values"))
```

Now, let's run the script to see the results:



The output demonstrates how the LLM leverages external tools to answer user queries. The LLM invoked the `search_web` tool and `get_weather` API to retrieve relevant information. The `search_web` tool successfully fetched a URL and extracted content, which the LLM then used to generate a response to the query. Similarly, the LLM utilized the `get_weather` API to fetch current weather information and incorporated it into its output for providing an accurate answer.

3. Building a custom agent

Now, let's create an agent using LangGraph.

- Import the required libraries.

```
from langgraph.prebuilt import ToolNode
```

```
from langgraph.graph import StateGraph, MessagesState, START, END
```

- Define a `tool_node` with the available tools. The `ToolNode` is a LangChain Runnable that takes graph state, which includes a list of messages as input and produces an updated state as the results of the tool calls.

```
tools = [search_web, get_weather]
tool_node = ToolNode(tools)
```

- Define functions to call the model or tools.

```
def call_model(state: MessagesState):
    messages = state["messages"]
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}

def call_tools(state: MessagesState) -> Literal["tools", END]:
    messages = state["messages"]
    last_message = messages[-1]
    if last_message.tool_calls:
        return "tools"
    return END
```

The `call_model` function takes messages from the state as input, which can include queries, prompts or tool content and returns a response. The `call_tools` function also accepts state messages as input and returns the tools node if the last message includes tool calls else it terminates.

- Now, let's create the nodes and edges.

```
# initialize the workflow from StateGraph
workflow = StateGraph(MessagesState)

# add a node named LLM, with call_model function. This node uses an LLM to make
workflow.add_node("LLM", call_model)

# Our workflow starts with the LLM node
workflow.add_edge(START, "LLM")

# Add a tools node
workflow.add_node("tools", tool_node)

# Add a conditional edge from LLM to call_tools function. It can go tools node o
workflow.add_conditional_edges("LLM", call_tools)

# tools node sends the information back to the LLM
workflow.add_edge("tools", "LLM")

agent = workflow.compile()
```

The above code initializes a workflow using the `StateGraph` class with the `MessagesState`. It first adds a node named LLM that utilizes the `call_model` function to make decisions based on the provided input. The workflow begins with the LLM node and establishes a connection to a tools node, which is created using the `tool_node`. Conditional edges are then added, allowing the workflow to either proceed to the tools node or terminate based on the output of the LLM node, which leads back to the LLM for further processing.

- Now, let's query the agent.

```
for chunk in agent.stream(
    {"messages": [("user", "Will it rain in Trivandrum today?")]},
    stream_mode="values",):
    chunk["messages"][-1].pretty_print()
```

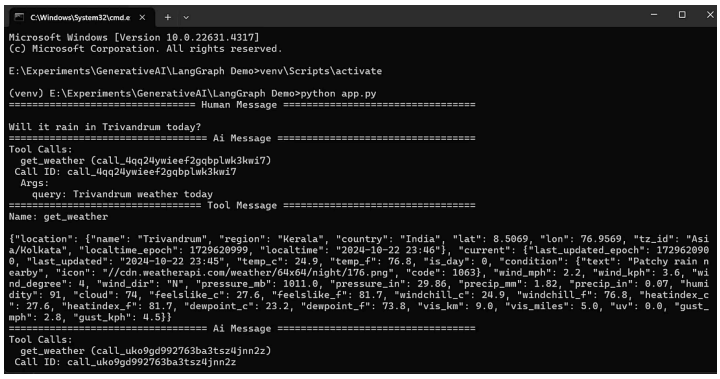
- The complete code is provided below.

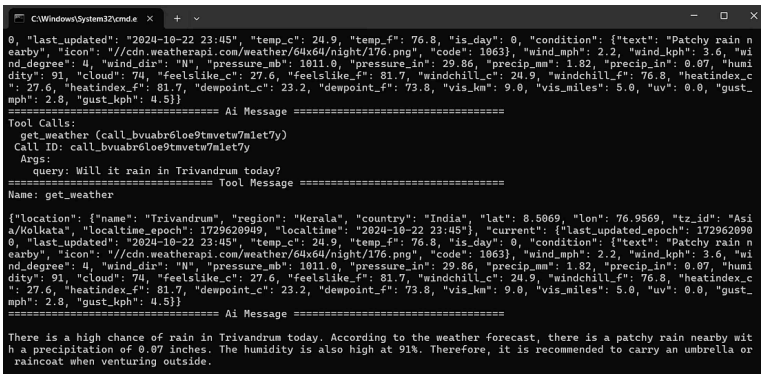
```
# import the required methods
from langgraph.prebuilt import ToolNode
from langgraph.graph import StateGraph, MessagesState, START, END

# define a tool_node with the available tools
tools = [search_web, get_weather]
tool_node = ToolNode(tools)

# define functions to call the LLM or the tools
def call_model(state: MessagesState):
    messages = state["messages"]
    response = llm_with_tools.invoke(messages)
    return {"messages": [response]}

def call_tools(state: MessagesState) -> Literal["tools", END]:
    messages = state["messages"]
    last_message = messages[-1]
    if last_message.tool_calls:
        return "tools"
    return END

# initialize the workflow from StateGraph
workflow = StateGraph(MessagesState)

# add a node named LLM, with call_model function. This node uses an LLM to make
workflow.add_node("LLM", call_model)

# Our workflow starts with the LLM node
workflow.add_edge(START, "LLM")

# Add a tools node
workflow.add_node("tools", tool_node)

# Add a conditional edge from LLM to call_tools function. It can go tools node o
workflow.add_conditional_edges("LLM", call_tools)

# tools node sends the information back to the LLM
workflow.add_edge("tools", "LLM")

agent = workflow.compile()
# display(Image(agent.get_graph().draw_mermaid_png()))
```

```
for chunk in agent.stream(
    {"messages": [("user", "Will it rain in Trivandrum today?")]},
    stream_mode="values",):
    chunk["messages"][-1].pretty_print()
```

Now, lets run the code to see the results.

The workflow is compiled into an agent, which streams responses to the user query about the weather in Trivandrum, displaying each message in a readable format as it is received.





In this example, the `get_weather` tool was invoked to retrieve various weather-related values, leading the LLM to conclude that it is likely to rain. This demonstrates how different tools can be integrated with the LLM, enhancing its ability to address queries that it might not be able to answer on its own.

Applications of LangGraph

LangGraph can be utilized to create a wide range of applications such as:

- **Chatbots**: Ideal for developing sophisticated chatbots that can manage various user requests, process natural language queries, and maintain context for a coherent user experience.
- **Autonomous Agents**: Enables the creation of agents capable of independent decision-making, executing complex workflows, and adapting to new information for tasks like automated customer support and system monitoring.
- **Multi-Agent Systems**: Facilitates the collaboration of multiple agents to achieve common goals, such as managing inventory and processing orders in supply chain management, enhancing operational efficiency.
- **Workflow Automation Tools**: Simplifies the automation of business processes, allowing intelligent agents to handle tasks like document processing and data analysis, thus reducing errors and increasing productivity.

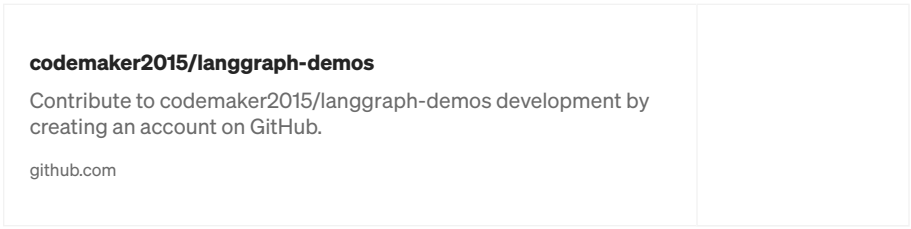
- **Recommendation Systems**: Supports personalized recommendation systems by analyzing user behavior and preferences through multiple agents, delivering tailored suggestions for products and services.

Thanks for reading this article !!

Thanks Gowri M Bhatt for reviewing the content.

If you enjoyed this article, please click on the clap button 👏 and share to help others find it!

The full source code for this tutorial can be found here,



Resources

- https://langchain-ai.github.io/langgraph
- https://youtu.be/5h-JBkySK34
- https://youtu.be/hvAPnpSfSGo
- https://www.udemy.com/course/langgraph

Langgraph | Langchain | Langchain Agents | Python | Ai Agent

Published in The Pythoneers

14.4K Followers · Last published 15 hours ago

Follow

Your home for innovative tech stories about Python and its limitless possibilities. Discover, learn, and get inspired.

Written by Vishnu Sivan

961 Followers · 181 Following

Follow

Try not to become a man of SUCCESS but rather try to become a man of VALUE

Responses (3)

Thomas Modern

What are your thoughts?

Brent Costa
Dec 10, 2024

Vishnu,

In terms of building an AI Assistant/tutor for K-8th graders, what hierarchical structure best fits?

May I have your email address?

2 1 reply Reply

Yethu Krishnan
Feb 28

workflow.add_conditional_edges("LLM", call_tools)

Instead of the call_tools custom function , we can use Langraph's prebuilt "tools_condition" function

workflow.add_conditional_edges("LLM", tools_condition,

if_true="tools",

if_false=END

)... more

Reply

Ramesh Adavi
Feb 18

Great lesson

Reply

More from the list: "LLM"

Curated by Thomas Modern

Edwin Lisowski
What Every AI Engineer Should Know About A2...
Apr 23

In Google Cloud... by Heik...
Getting Started with Google A2A: A Hands-o...
Apr 15

Minyang Chen
Google's A2A Protocol: Seamless Agent...
Apr 18

CPlog
Int... La...
Feb

View list

More from Vishnu Sivan and The Pythoneers

Vishnu Sivan

Introducing uv: Next-Gen Python Package Manager

Python evolution has been closely tied to advancements in package management, fro...

Dec 15, 2024 266 3

In The Pythoneers by Abhay Parashar

How to Explain Each Core Machine Learning Model in an Interview

From Regression to Clustering to CNNs: A Brief Guide to 25+ Machine Learning Models

Mar 11 2.1K 26

In The Pythoneers by Aashish Kumar

9 Modern Python Libraries You Must Know in 2025! 🚀

I have discovered these few game changer modern set of libraries that you should must...

Mar 2 702 8

In The Pythoneers by Vishnu Sivan

Building a multi agent system using CrewAI

AI agents are transforming industries by independently analyzing data, making...

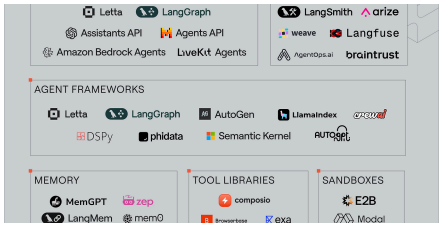
Nov 4, 2024 207 2

See all from Vishnu Sivan

See all from The Pythoneers

Recommended from Medium

Lovelyn David



Vipra Singh

LangGraph Series-2-Creating a Conversational Bot with Memory...

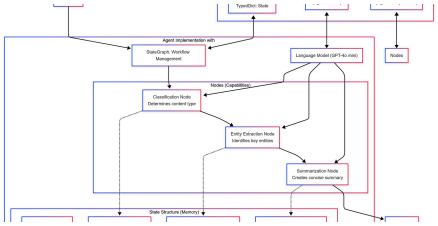
In the previous article, we explored how to construct a graph using LangGraph. In this...

Apr 11 16 1

AI Agents: Introduction (Part-1)

Discover AI agents, their design, and real-world applications.

Feb 2 2K 42

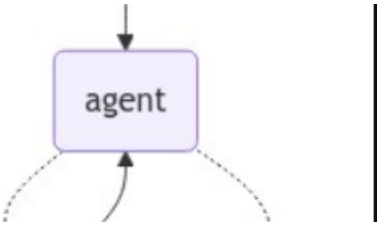


In Data Science Collective by Paolo Perrone

The Complete Guide to Building Your First AI Agent with...

Three months into building my first commercial AI agent, everything collapsed...

Mar 11 3.1K 66



Story_Teller

Introduction to Tool Use with LangGraph's ToolNode

Modern AI applications often require seamless integration of external tools to...

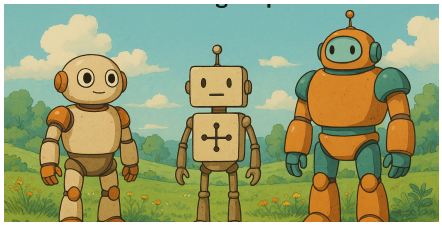
Dec 14, 2024 11 4

Prabhudev Guntur

Choosing Your AI Agent Framework: Google ADK vs....

The world of AI is buzzing with the potential of multi-agent systems—frameworks where...

Apr 15 16 3



In ProjectPro by Khushbu Shah

AutoGen vs. LangGraph vs. CrewAI:Who Wins?

AI agents are taking over workflows—from research automation to customer service...

Mar 31 29 1

See more recommendations