Sign in

{{ message }}

langchain-ai / langgraph Public

Build resilient language agents as graphs.

langchain-ai.github.io/langgraph/

MIT license

6.5k stars 1k forks Branches Tags Activity

Star

Notifications

Code

Issues 44

Pull requests 50

Discussions

Actions

Projects 1

Security

Insights

main

Code

View all files

LangGraph

pypi v0.2.44 downloads/month 2M open issues 44 docs latest

Building language agents as graphs

Note

Looking for the JS version? Click here (JS docs).

Overview

LangGraph is a library for building stateful, multi-actor applications with LLMs, used to create agent and
other LLM frameworks, it offers these core benefits: cycles, controllability, and persistence. LangGraph all
cycles, essential for most agentic architectures, differentiating it from DAG-based solutions. As a very low
control over both the flow and state of your application, crucial for creating reliable agents. Additionally,
enabling advanced human-in-the-loop and memory features.

LangGraph is inspired by Pregel and Apache Beam. The public interface draws inspiration from NetworkX
creators of LangChain, but can be used without LangChain.

LangGraph Platform is infrastructure for deploying LangGraph agents. It is a commercial solution for dep
built on the open-source LangGraph framework. The LangGraph Platform consists of several component
development, deployment, debugging, and monitoring of LangGraph applications: LangGraph Server (AP
LangGraph CLI (command line tool for building the server), LangGraph Studio (UI/debugger),

To learn more about LangGraph, check out our first LangChain Academy course, *Introduction to LangGra*

Key Features

- **Cycles and Branching**: Implement loops and conditionals in your apps.
- **Persistence**: Automatically save state after each step in the graph. Pause and resume the graph exec
  recovery, human-in-the-loop workflows, time travel and more.
- **Human-in-the-Loop**: Interrupt graph execution to approve or edit next action planned by the agent
- **Streaming Support**: Stream outputs as they are produced by each node (including token streaming
- **Integration with LangChain**: LangGraph integrates seamlessly with LangChain and LangSmith (bu

LangGraph Platform

LangGraph Platform is a commercial solution for deploying agentic applications to production, built on th
Here are some common issues that arise in complex deployments, which LangGraph Platform addresses:

- **Streaming support**: LangGraph Server provides multiple streaming modes optimized for various a
- **Background runs**: Runs agents asynchronously in the background
- **Support for long running agents**: Infrastructure that can handle long running processes
- **Double texting**: Handle the case where you get two messages from the user before the agent can r
- **Handle burstiness**: Task queue for ensuring requests are handled consistently without loss, even ur

Installation

```
pip install -U langgraph
```

Example

One of the central concepts of LangGraph is state. Each graph execution creates a state that is passed bet
and each node updates this internal state with its return value after it executes. The way that the graph up
the type of graph chosen or a custom function.

Let's take a look at a simple example of an agent that can use a search tool.

```
pip install langchain-anthropic
```

```
export ANTHROPIC_API_KEY=sk-...
```

Optionally, we can set up LangSmith for best-in-class observability.

```
export LANGSMITH_TRACING=true
export LANGSMITH_API_KEY=lsv2_sk_...
from typing import Annotated, Literal, TypedDict

from langchain_core.messages import HumanMessage
from langchain_anthropic import ChatAnthropic
from langchain_core.tools import tool
from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import END, START, StateGraph, MessagesState
from langgraph.prebuilt import ToolNode


# Define the tools for the agent to use
@tool
def search(query: str):
    """Call to surf the web."""
    # This is a placeholder, but don't tell the LLM that...
    if "sf" in query.lower() or "san francisco" in query.lower():
        return "It's 60 degrees and foggy."
    return "It's 90 degrees and sunny."


tools = [search]

tool_node = ToolNode(tools)

model = ChatAnthropic(model="claude-3-5-sonnet-20240620", temperature=0).bind_tools(tools)

# Define the function that determines whether to continue or not
def should_continue(state: MessagesState) -> Literal["tools", END]:
    messages = state['messages']
    last_message = messages[-1]
    # If the LLM makes a tool call, then we route to the "tools" node
    if last_message.tool_calls:
        return "tools"
    # Otherwise, we stop (reply to the user)
    return END


# Define the function that calls the model
def call_model(state: MessagesState):
    messages = state['messages']
    response = model.invoke(messages)
    # We return a list, because this will get added to the existing list
    return {"messages": [response]}


# Define a new graph
workflow = StateGraph(MessagesState)

# Define the two nodes we will cycle between
workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)

# Set the entrypoint as `agent`
# This means that this node is the first one called
workflow.add_edge(START, "agent")

# We now add a conditional edge
workflow.add_conditional_edges(
    # First, we define the start node. We use `agent`.
    # This means these are the edges taken after the `agent` node is called.
    "agent",
    # Next, we pass in the function that will determine which node is called next.
    should_continue,
)

# We now add a normal edge from `tools` to `agent`.
# This means that after `tools` is called, `agent` node is called next.
workflow.add_edge("tools", 'agent')

# Initialize memory to persist state between graph runs
checkpointer = MemorySaver()

# Finally, we compile it!
# This compiles it into a LangChain Runnable,
# meaning you can use it as you would any other runnable.
# Note that we're (optionally) passing the memory when compiling the graph
app = workflow.compile(checkpointer=checkpointer)

# Use the Runnable
final_state = app.invoke(
    {"messages": [HumanMessage(content="what is the weather in sf")]},
    config={"configurable": {"thread_id": 42}}
)
final_state["messages"][-1].content
"Based on the search results, I can tell you that the current weather in San Francisco is:`
```

Now when we pass the same `"thread_id"`, the conversation context is retained via the saved state (i.e. s

```
final_state = app.invoke(
    {"messages": [HumanMessage(content="what about ny")]},
    config={"configurable": {"thread_id": 42}}
)
final_state["messages"][-1].content
"Based on the search results, I can tell you that the current weather in New York City is:`
```

Step-by-step Breakdown

1. ▶ Initialize the model and tools.

2. ▶ Initialize graph with state.
3. ▶ Define graph nodes.
4. ▶ Define entry point and graph edges.
5. ▶ Compile the graph.
6. ▶ Execute the graph.

Documentation

- Tutorials: Learn to build with LangGraph through guided examples.
- How-to Guides: Accomplish specific things within LangGraph, from streaming, to adding memory & (branching, subgraphs, etc.), these are the place to go if you want to copy and run a specific code sni
- Conceptual Guides: In-depth explanations of the key concepts and principles behind LangGraph, sucl
- API Reference: Review important classes and methods, simple examples of how to use the graph anc components and more.
- Cloud (beta): With one click, deploy LangGraph applications to LangGraph Cloud.

Contributing

For more information on how to contribute, see here.

Releases 125

0.2.44 Latest
Nov 2, 2024
+ 124 releases

Packages 0

No packages published

Used by 5.4k

+ 5,361

Contributors 87

+ 73 contributors

Languages

Python 96.0% TypeScript 3.1% Other 0.9%

© 2024 GitHub, Inc.

Terms
Privacy
Security
Status
Docs
Contact
Manage cookies
Do not share my personal information