

langchain-ai / langgraph-swarm-py

<> Code

Issues 3

Pull requests

Actions

Projects

Security

Insights

For your multi-agent needs

[langchain-ai.github.io/langgraph/concepts/multi_agent/](#)

MIT license

1k stars

138 forks

12 watching

Branches

Activity

Custom properties

Tags

Public repository

26 Branches

12 Tags

Go to file

Go to file

+

Add file

<> Code

eyurtsev

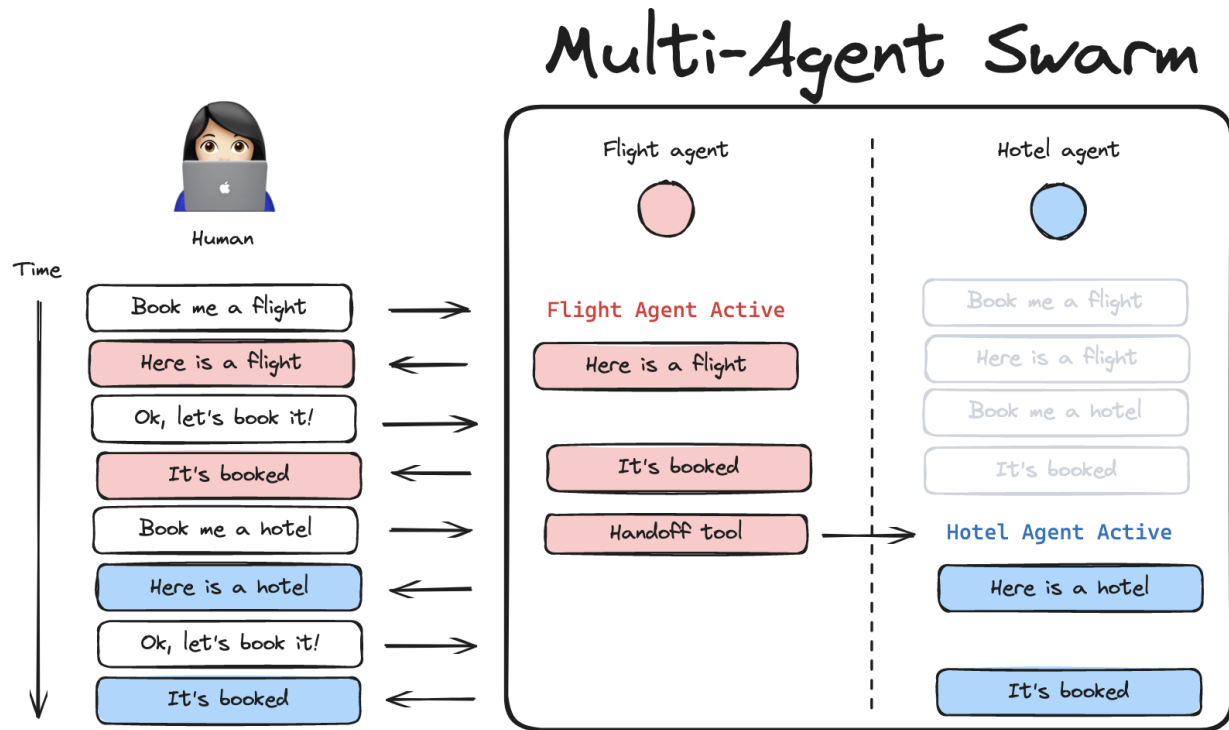
chore: turn on mypy, add py.typed (#90)

fc8e2c4 · 5 days ago

.github	initial implementation (#1)	5 months ago
examples	Apply stricter linting (#89)	5 days ago
langgraph_swarm	chore: turn on mypy, add py.typed (#90)	5 days ago
static/img	update readme image (#3)	5 months ago
tests	chore: turn on mypy, add py.typed (#90)	5 days ago
.gitignore	Apply stricter linting (#89)	5 days ago
LICENSE	first commit	5 months ago
Makefile	chore: turn on mypy, add py.typed (#90)	5 days ago
README.md	update handoff to include names (#51)	3 months ago
pyproject.toml	chore: turn on mypy, add py.typed (#90)	5 days ago
uv.lock	chore: turn on mypy, add py.typed (#90)	5 days ago

LangGraph Multi-Agent Swarm

A Python library for creating swarm-style multi-agent systems using [LangGraph](#). A swarm is a type of [multi-agent](#) architecture where agents dynamically hand off control to one another based on their specializations. The system remembers which agent was last active, ensuring that on subsequent interactions, the conversation resumes with that agent.



Features

- **Multi-agent collaboration** - Enable specialized agents to work together and hand off context to each other
- **Customizable handoff tools** - Built-in tools for communication between agents

This library is built on top of [LangGraph](#), a powerful framework for building agent applications, and comes with out-of-box support for [streaming](#), [short-term and long-term memory](#) and [human-in-the-loop](#)

Installation

```
pip install langgraph-swarm
```

Quickstart

```
pip install langgraph-swarm langchain-openai
```

```
export OPENAI_API_KEY=<your_api_key>
```

```
from langchain_openai import ChatOpenAI

from langgraph.checkpoint.memory import InMemorySaver
from langgraph.prebuilt import create_react_agent
from langgraph_swarm import create_handoff_tool, create_swarm

model = ChatOpenAI(model="gpt-4o")

def add(a: int, b: int) -> int:
```

```

    """Add two numbers"""
    return a + b

alice = create_react_agent(
    model,
    [add, create_handoff_tool(agent_name="Bob")],
    prompt="You are Alice, an addition expert.",
    name="Alice",
)

bob = create_react_agent(
    model,
    [create_handoff_tool(agent_name="Alice", description="Transfer to Alice, she can help with
math")],
    prompt="You are Bob, you speak like a pirate.",
    name="Bob",
)

checkpointer = InMemorySaver()
workflow = create_swarm(
    [alice, bob],
    default_active_agent="Alice"
)
app = workflow.compile(checkpointer=checkpointer)

config = {"configurable": {"thread_id": "1"}}
turn_1 = app.invoke(
    {"messages": [{"role": "user", "content": "i'd like to speak to Bob"}]},
    config,
)
print(turn_1)
turn_2 = app.invoke(
    {"messages": [{"role": "user", "content": "what's 5 + 7?"}]},
    config,
)
print(turn_2)

```

Memory

You can add [short-term](#) and [long-term memory](#) to your swarm multi-agent system. Since `create_swarm()` returns an instance of `StateGraph` that needs to be compiled before use, you can directly pass a [checkpointer](#) or a [store](#) instance to the `.compile()` method:

```

from langgraph.checkpoint.memory import InMemorySaver
from langgraph.store.memory import InMemoryStore

# short-term memory
checkpointer = InMemorySaver()
# long-term memory
store = InMemoryStore()

model = ...
alice = ...
bob = ...

workflow = create_swarm(
    [alice, bob],
    default_active_agent="Alice"

```



```
)

# Compile with checkpointer/store
app = workflow.compile(
    checkpointer=checkpointer,
    store=store
)
```

⚠ Important

Adding [short-term memory](#) is crucial for maintaining conversation state across multiple interactions. Without it, the swarm would "forget" which agent was last active and lose the conversation history. Make sure to always compile the swarm with a checkpointer if you plan to use it in multi-turn conversations; e.g.,

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

How to customize

You can customize multi-agent swarm by changing either the [handoff tools](#) implementation or the [agent implementation](#).

Customizing handoff tools

📖 README 📄 MIT license



modify the default implementation:

- change tool name and/or description
- add tool call arguments for the LLM to populate, for example a task description for the next agent
- change what data is passed to the next agent as part of the handoff: by default `create_handoff_tool` passes **full** message history (all of the messages generated in the swarm up to this point), as well as a tool message indicating successful handoff.

Here is an example of what a custom handoff tool might look like:

```
from typing import Annotated

from langchain_core.tools import tool, BaseTool, InjectedToolCallId
from langchain_core.messages import ToolMessage
from langgraph.types import Command
from langgraph.prebuilt import InjectedState

def create_custom_handoff_tool(*, agent_name: str, name: str | None, description: str | None) ->
BaseTool:

    @tool(name, description=description)
    def handoff_to_agent(
        # you can add additional tool call arguments for the LLM to populate
        # for example, you can ask the LLM to populate a task description for the next agent
        task_description: Annotated[str, "Detailed description of what the next agent should do,
including all of the relevant context."],
        # you can inject the state of the agent that is calling the tool
        state: Annotated[dict, InjectedState],
        tool_call_id: Annotated[str, InjectedToolCallId],
    ):
```



```

    tool_message = ToolMessage(
        content=f"Successfully transferred to {agent_name}",
        name=name,
        tool_call_id=tool_call_id,
    )
    # you can use a different messages state key here, if your agent uses a different schema
    # e.g., "alice_messages" instead of "messages"
    messages = state["messages"]
    return Command(
        goto=agent_name,
        graph=Command.PARENT,
        # NOTE: this is a state update that will be applied to the swarm multi-agent graph
        # (i.e., the PARENT graph)
        update={
            "messages": messages + [tool_message],
            "active_agent": agent_name,
            # optionally pass the task description to the next agent
            "task_description": task_description,
        },
    )

    return handoff_to_agent

```

⚠ Important

If you are implementing custom handoff tools that return `Command`, you need to ensure that:

- (1) your agent has a tool-calling node that can handle tools returning `Command` (like LangGraph's prebuilt [ToolNode](#))
- (2) both the swarm graph and the next agent graph have the [state schema](#) containing the keys you want to update in `Command.update`

Customizing agent implementation

By default, individual agents are expected to communicate over a single `messages` key that is shared by all agents and the overall multi-agent swarm graph. This means that messages from **all** of the agents will be combined into a single, shared list of messages. This might not be desirable if you don't want to expose an agent's internal history of messages. To change this, you can customize the agent by taking the following steps:

1. use custom [state schema](#) with a different key for messages, for example `alice_messages`
2. write a wrapper that converts the parent graph state to the child agent state and back (see this [how-to](#) guide)

```

from typing_extensions import TypedDict, Annotated

from langchain_core.messages import AnyMessage
from langgraph.graph import StateGraph, add_messages
from langgraph_swarm import SwarmState

class AliceState(TypedDict):
    alice_messages: Annotated[list[AnyMessage], add_messages]

# see this guide to learn how you can implement a custom tool-calling agent
# https://langchain-ai.github.io/langgraph/how-tos/react-agent-from-scratch/
alice = (
    StateGraph(AliceState)
    .add_node("model", ...)
    .add_node("tools", ...)

```

```

    .add_edge(...)
    ...
    .compile()
)

# wrapper calling the agent
def call_alice(state: SwarmState):
    # you can put any input transformation from parent state -> agent state
    # for example, you can invoke "alice" with "task_description" populated by the LLM
    response = alice.invoke({"alice_messages": state["messages"]})
    # you can put any output transformation from agent state -> parent state
    return {"messages": response["alice_messages"]}

def call_bob(state: SwarmState):
    ...

```

Then, you can create the swarm manually in the following way:

```


from langgraph_swarm import add_active_agent_router

workflow = (
    StateGraph(SwarmState)
    .add_node("Alice", call_alice, destinations=("Bob",))
    .add_node("Bob", call_bob, destinations=("Alice",))
)
# this is the router that enables us to keep track of the last active agent
workflow = add_active_agent_router(
    builder=workflow,
    route_to=["Alice", "Bob"],
    default_active_agent="Alice",
)

# compile the workflow
app = workflow.compile()

```

Releases 12


 langgraph-swarm==0.0.12 Latest
5 days ago

[+ 11 releases](#)

Packages

No packages published

Contributors 4

 **lancemartin** Lance Martin
 **eyurtsev** Eugene Yurtsev