Build a custom RAG agent

Copy page

LangGraph v1.0

Welcome to the new LangGraph documentation! If you encounter any issues or have feedback, please **open an issue** so we can improve. Archived v0 documentation can be found **here**.

See the **release notes** and **migration guide** for a complete list of changes and instructions on how to upgrade your code.

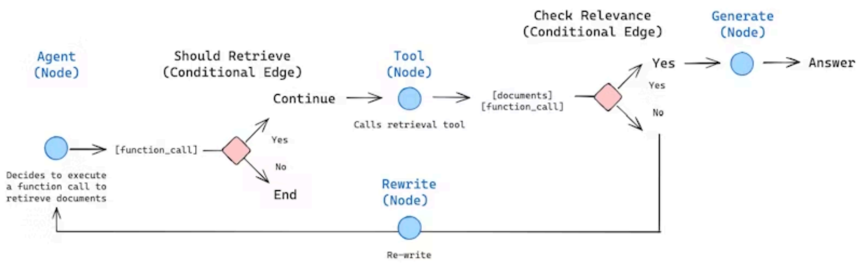
Overview

In this tutorial we will build a **retrieval** agent using LangGraph.

LangChain offers built-in **agent** implementations, implemented using **LangGraph** primitives. If deeper customization is required, agents can be implemented directly in LangGraph. This guide demonstrates an example implementation of a retrieval agent. **Retrieval** agents are useful when you want an LLM to make a decision about whether to retrieve context from a vectorstore or respond to the user directly.

By the end of the tutorial we will have done the following:

1. Fetch and preprocess documents that will be used for retrieval.
2. Index those documents for semantic search and create a retriever tool for the agent.
3. Build an agentic RAG system that can decide when to use the retriever tool.



Concepts

We will cover the following concepts:

Retrieval using **document loaders**, **text splitters**, **embeddings**, and **vector stores**

The LangGraph **Graph API**, including state, nodes, edges, and conditional edges.

Setup

Let's download the required packages and set our API keys:

```
%%capture --no-stderr
pip install -U --quiet langgraph "langchain[openai]" langchain-communi
```

```
import getpass
import os


def _set_env(key: str):
    if key not in os.environ:
        os.environ[key] = getpass.getpass(f"{key}:")


_set_env("OPENAI_API_KEY")
```

Sign up for LangSmith to quickly spot issues and improve the performance of your LangGraph projects. **LangSmith** lets you use trace data to debug, test, and monitor your LLM apps built with LangGraph.

1. Preprocess documents

1. Fetch documents to use in our RAG system. We will use three of the most recent pages from **Lilian Weng's excellent blog**. We'll start by fetching the content of the pages using `WebBaseLoader` utility:

```
from langchain_community.document_loaders import WebBaseLoader

urls = [
    "https://lilianweng.github.io/posts/2024-11-28-reward-hacking/",
    "https://lilianweng.github.io/posts/2024-07-07-hallucination/",
    "https://lilianweng.github.io/posts/2024-04-12-diffusion-video/",
]

docs = [WebBaseLoader(url).load() for url in urls]
```

```
docs[0][0].page_content.strip()[:1000]
```

2. Split the fetched documents into smaller chunks for indexing into our vectorstore:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=100, chunk_overlap=50
)
doc_splits = text_splitter.split_documents(docs_list)
```

```
doc_splits[0].page_content.strip()
```

2. Create a retriever tool

Now that we have our split documents, we can index them into a vector store that we'll use for semantic search.

1. Use an in-memory vector store and OpenAI embeddings:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings

vectorstore = InMemoryVectorStore.from_documents(
    documents=doc_splits, embedding=OpenAIEmbeddings()
)
retriever = vectorstore.as_retriever()
```

2. Create a retriever tool using LangChain's prebuilt `create_retriever_tool`:

```
from langchain.tools.retriever import create_retriever_tool

retriever_tool = create_retriever_tool(
    retriever,
    "retrieve_blog_posts",
    "Search and return information about Lilian Weng blog posts.",
)
```

3. Test the tool:

```
retriever_tool.invoke({"query": "types of reward hacking"})
```

3. Generate query

Now we will start building components (**nodes** and **edges**) for our agentic RAG graph.

Note that the components will operate on the `MessagesState` — graph state that contains a `messages` key with a list of **chat messages**.

1. Build a `generate_query_or_respond` node. It will call an LLM to generate a response based on the current graph state (list of messages). Given the input messages, it will decide to retrieve using the retriever tool, or respond directly to the user. Note that we're giving the chat model access to the `retriever_tool` we created earlier via `.bind_tools`:

```python
from langgraph.graph import MessagesState
from langchain.chat_models import init_chat_model

response_model = init_chat_model("openai:gpt-4o", temperature=0)


def generate_query_or_respond(state: MessagesState):
    """Call the model to generate a response based on the current stat
    the question, it will decide to retrieve using the retriever tool,
    """
    response = (
        response_model
        .bind_tools([retriever_tool]).invoke(state["messages"])
    )
    return {"messages": [response]}
```

2. Try it on a random input:

```python
input = {"messages": [{"role": "user", "content": "hello!"}]}
generate_query_or_respond(input)["messages"][-1].pretty_print()
```

Output:

```
================================== Ai Message =======================

Hello! How can I help you today?
```

3. Ask a question that requires semantic search:

```python
input = {
    "messages": [
        {
            "role": "user",
            "content": "What does Lilian Weng say about types of rewar
        }
    ]
}
generate_query_or_respond(input)["messages"][-1].pretty_print()
```

Output:

```
================================== Ai Message =======================
Tool Calls:
retrieve_blog_posts (call_tYQxgfIlnQUDMdtAhdbXNwIM)
Call ID: call_tYQxgfIlnQUDMdtAhdbXNwIM
Args:
    query: types of reward hacking
```

4. Grade documents

1. Add a **conditional edge** — `grade_documents` — to determine whether the retrieved documents are relevant to the question. We will use a model with

a structured output schema `GradeDocuments` for document grading. The `grade_documents` function will return the name of the node to go to based on the grading decision (`generate_answer` or `rewrite_question`):

```
from pydantic import BaseModel, Field
from typing import Literal

GRADE_PROMPT = (
    "You are a grader assessing relevance of a retrieved document to a
    "Here is the retrieved document: \n\n {context} \n\n"
    "Here is the user question: {question} \n"
    "If the document contains keyword(s) or semantic meaning related t
    "Give a binary score 'yes' or 'no' score to indicate whether the d
)


class GradeDocuments(BaseModel):
    """Grade documents using a binary score for relevance check."""

    binary_score: str = Field(
        description="Relevance score: 'yes' if relevant, or 'no' if no
    )


grader_model = init_chat_model("openai:gpt-4o", temperature=0)


def grade_documents(
    state: MessagesState,
) -> Literal["generate_answer", "rewrite_question"]:
    """Determine whether the retrieved documents are relevant to the q
    question = state["messages"][0].content
    context = state["messages"][-1].content

    prompt = GRADE_PROMPT.format(question=question, context=context)
    response = (
        grader_model
        .with_structured_output(GradeDocuments).invoke(
            [{"role": "user", "content": prompt}]
        )
    )
    score = response.binary_score

    if score == "yes":
        return "generate_answer"
    else:
        return "rewrite_question"
```

2. Run this with irrelevant documents in the tool response:

```
from langchain_core.messages import convert_to_messages

input = {
    "messages": convert_to_messages(
        [
            {
                "role": "user",
                "content": "What does Lilian Weng say about types of r
            },
            {
                "role": "assistant",
                "content": "",
                "tool_calls": [
                    {
                        "id": "1",
                        "name": "retrieve_blog_posts",
                        "args": {"query": "types of reward hacking"},
                    }
                ],
            },
            {"role": "tool", "content": "meow", "tool_call_id": "1"},
        ]
    )
}
grade_documents(input)
```

3. Confirm that the relevant documents are classified as such:

```
input = {
    "messages": convert_to_messages(
        [
            {
                "role": "user",
                "content": "What does Lilian Weng say about types of r
            },
            {
                "role": "assistant",
                "content": "",
                "tool_calls": [
                    {
                        "id": "1",
                        "name": "retrieve_blog_posts",
                        "args": {"query": "types of reward hacking"},
                    }
                ],
            },
            {
                "role": "tool",
                "content": "reward hacking can be categorized into two
                "tool_call_id": "1",
            },
        ]
    )
}
grade_documents(input)
```

5. Rewrite question

1. Build the `rewrite_question` node. The retriever tool can return potentially irrelevant documents, which indicates a need to improve the original user question. To do so, we will call the `rewrite_question` node:

```
REWRITE_PROMPT = (
    "Look at the input and try to reason about the underlying semantic
    "Here is the initial question:"
    "\n ------- \n"
    "{question}"
    "\n ------- \n"
    "Formulate an improved question:"
)


def rewrite_question(state: MessagesState):
    """Rewrite the original user question."""
    messages = state["messages"]
    question = messages[0].content
    prompt = REWRITE_PROMPT.format(question=question)
    response = response_model.invoke([{"role": "user", "content": prom
    return {"messages": [{"role": "user", "content": response.content}
```

2. Try it out:

```
input = {
    "messages": convert_to_messages(
        [
            {
                "role": "user",
                "content": "What does Lilian Weng say about types of r
            },
            {
                "role": "assistant",
                "content": "",
                "tool_calls": [
                    {
                        "id": "1",
                        "name": "retrieve_blog_posts",
                        "args": {"query": "types of reward hacking"},
                    }
                ],
            },
            {"role": "tool", "content": "meow", "tool_call_id": "1"},
        ]
    )
}

response = rewrite_question(input)
print(response["messages"][-1]["content"])
```

Output:

```
What are the different types of reward hacking described by Lilian Wen
```

6. Generate an answer

1. Build `generate_answer` node: if we pass the grader checks, we can generate the final answer based on the original question and the retrieved context:

```
GENERATE_PROMPT = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer the quest
    "If you don't know the answer, just say that you don't know. "
    "Use three sentences maximum and keep the answer concise.\n"
    "Question: {question} \n"
    "Context: {context}"
)


def generate_answer(state: MessagesState):
    """Generate an answer."""
    question = state["messages"][0].content
    context = state["messages"][-1].content
    prompt = GENERATE_PROMPT.format(question=question, context=context
    response = response_model.invoke([{"role": "user", "content": prom
    return {"messages": [response]}
```

2. Try it:

```
input = {
    "messages": convert_to_messages(
        [
            {
                "role": "user",
                "content": "What does Lilian Weng say about types of r
            },
            {
                "role": "assistant",
                "content": "",
                "tool_calls": [
                    {
                        "id": "1",
                        "name": "retrieve_blog_posts",
                        "args": {"query": "types of reward hacking"},
                    }
                ],
            },
            {
                "role": "tool",
                "content": "reward hacking can be categorized into two
                "tool_call_id": "1",
            },
        ]
    )
}

response = generate_answer(input)
response["messages"][-1].pretty_print()
```

Output:

```
================================ Ai Message ==================================

Lilian Weng categorizes reward hacking into two types: environment or
```

7. Assemble the graph

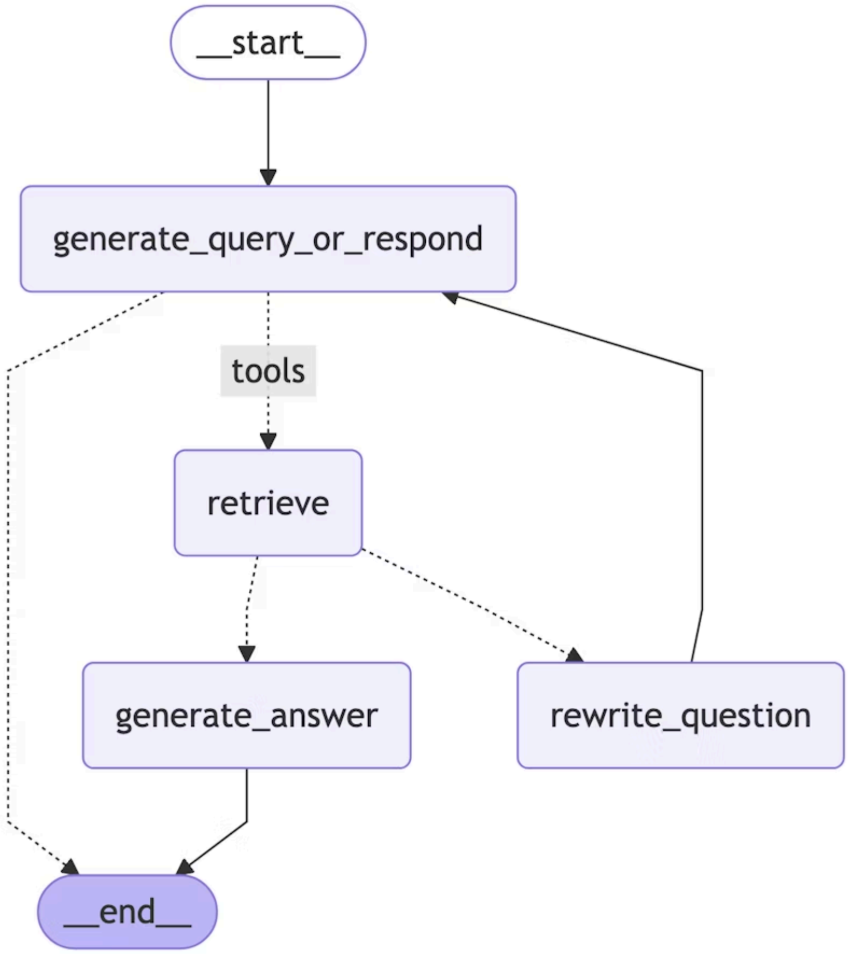
Now we'll assemble all the nodes and edges into a complete graph:

- Start with a `generate_query_or_respond` and determine if we need to call `retriever_tool`
- Route to next step using `tools_condition` :
  - If `generate_query_or_respond` returned `tool_calls` , call `retriever_tool` to retrieve context
  - Otherwise, respond directly to the user
- Grade retrieved document content for relevance to the question (`grade_documents`) and route to next step:
  - If not relevant, rewrite the question using `rewrite_question` and then call `generate_query_or_respond` again
  - If relevant, proceed to `generate_answer` and generate final response using the `ToolMessage` with the retrieved document context

```python
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode, tools_condition

workflow = StateGraph(MessagesState)

# Define the nodes we will cycle between
workflow.add_node(generate_query_or_respond)
workflow.add_node("retrieve", ToolNode([retriever_tool]))
workflow.add_node(rewrite_question)
workflow.add_node(generate_answer)

workflow.add_edge(START, "generate_query_or_respond")

# Decide whether to retrieve
workflow.add_conditional_edges(
    "generate_query_or_respond",
    # Assess LLM decision (call `retriever_tool` tool or respond to th
    tools_condition,
    {
        # Translate the condition outputs to nodes in our graph
        "tools": "retrieve",
        END: END,
    },
)

# Edges taken after the `action` node is called.
workflow.add_conditional_edges(
    "retrieve",
    # Assess agent decision
    grade_documents,
)
workflow.add_edge("generate_answer", END)
workflow.add_edge("rewrite_question", "generate_query_or_respond")

# Compile
graph = workflow.compile()
```

Visualize the graph:

```python
from IPython.display import Image, display

display(Image(graph.get_graph().draw_mermaid_png()))
```



8. Run the agentic RAG

Now let's test the complete graph by running it with a question:

```
for chunk in graph.stream(
    {
        "messages": [
            {
                "role": "user",
                "content": "What does Lilian Weng say about types of r
            }
        ]
    }
):
    for node, update in chunk.items():
        print("Update from node", node)
        update["messages"][-1].pretty_print()
        print("\n\n")
```

Output:

```
Update from node generate_query_or_respond
================================== Ai Message =========================
Tool Calls:
  retrieve_blog_posts (call_NYu2vq4km9nNNEFqJwefWKu1)
 Call ID: call_NYu2vq4km9nNNEFqJwefWKu1
  Args:
    query: types of reward hacking



Update from node retrieve
================================= Tool Message =========================
Name: retrieve_blog_posts

(Note: Some work defines reward tampering as a distinct category of mi
At a high level, reward hacking can be categorized into two types: env

Why does Reward Hacking Exist?#

Pan et al. (2022) investigated reward hacking as a function of agent c

Let's Define Reward Hacking#
Reward shaping in RL is challenging. Reward hacking occurs when an RL



Update from node generate_answer
================================== Ai Message =========================

Lilian Weng categorizes reward hacking into two types: environment or
```

Edit the source of this page on GitHub

Was this page helpful? Yes No