Build a Retrieval Augmented Generation (RAG) App

One of the most powerful applications enabled by LLMs is sophisticated question-answering (Q&A) chatbots. These are applications that can answer questions about specific source information. These applications use a technique known as Retrieval Augmented Generation, or RAG.

This tutorial will show how to build a simple Q&A application over a text data source. Along the way we'll go over a typical Q&A architecture and highlight additional resources for more advanced Q&A techniques. We'll also see how LangSmith can help us trace and understand our application. LangSmith will become increasingly helpful as our application grows in complexity.

If you're already familiar with basic retrieval, you might also be interested in this high-level overview of different retrieval techinques.

What is RAG?

RAG is a technique for augmenting LLM knowledge with additional data.

LLMs can reason about wide-ranging topics, but their knowledge is limited to the public data up to a specific point in time that they were trained on. If you want to build AI applications that can reason about private data or data introduced after a model's cutoff date, you need to augment the knowledge of the model with the specific information it needs. The process of bringing the appropriate information and inserting it into the model prompt is known as Retrieval Augmented Generation (RAG).

LangChain has a number of components designed to help build Q&A applications, and RAG applications more generally.

Note: Here we focus on Q&A for unstructured data. If you are interested for RAG over structured data, check out our tutorial on doing question/answering over SQL data.

Concepts

A typical RAG application has two main components:

Indexing: a pipeline for ingesting data from a source and indexing it. *This usually happens offline.*

Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model.

The most common full sequence from raw data to answer looks like:

Indexing

1. **Load**: First we need to load our data. This is done with Document Loaders.
2. **Split**: Text splitters break large `Documents` into smaller chunks. This is useful both for indexing data and for passing it in to a model, since large chunks are harder to search over and won't fit in a model's finite context window.
3. **Store**: We need somewhere to store and index our splits, so that they can later be searched over. This is often done using a VectorStore and Embeddings model.

Retrieval and generation

4. **Retrieve**: Given a user input, relevant splits are retrieved from storage using a Retriever.
5. **Generate**: A ChatModel / LLM produces an answer using a prompt that includes the question and the retrieved data

Setup

Jupyter Notebook

This guide (and most of the other guides in the documentation) uses Jupyter notebooks and assumes the reader is as well. Jupyter notebooks are perfect for learning how to work with LLM systems because oftentimes things can go wrong (unexpected output, API down, etc) and going through guides in an interactive environment is a great way to better understand them.

This and other tutorials are perhaps most conveniently run in a Jupyter notebook. See here for instructions on how to install.

Installation

This tutorial requires these langchain dependencies:

Pip Conda

```
%pip install --quiet --upgrade langchain langchain-community langchain-chroma
```

For more details, see our Installation guide.

LangSmith

Many of the applications you build with LangChain will contain multiple steps with multiple invocations of LLM calls. As these applications get more and more complex, it becomes crucial to be able to inspect what exactly is going on inside your chain or agent. The best way to do this is with LangSmith.

After you sign up at the link above, make sure to set your environment variables to start logging traces:

```
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_API_KEY="..."
```

Or, if in a notebook, you can set them with:

```
import getpass
import os

os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_API_KEY"] = getpass.getpass()
```

Preview

In this guide we'll build an app that answers questions about the content of a website. The specific website we will use is the LLM Powered Autonomous Agents blog post by Lilian Weng, which allows us to ask questions about the contents of the post.

We can create a simple indexing pipeline and RAG chain to do this in ~20 lines of code:

OpenAI | Anthropic | Azure | Google | Cohere | NVIDIA | FireworksAI | Gr

```
pip install -qU langchain-openai
```

```
import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass()

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")
```

```
import bs4
from langchain import hub
```

```
from langchain_chroma import Chroma
from langchain_community.document_loaders import WebBaseLoader
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Load, chunk and index the contents of the blog.
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
    bs_kwargs=dict(
        parse_only=bs4.SoupStrainer(
            class_=("post-content", "post-title", "post-header")
        )
    ),
)
docs = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
splits = text_splitter.split_documents(docs)
vectorstore = Chroma.from_documents(documents=splits,
embedding=OpenAIEmbeddings())

# Retrieve and generate using the relevant snippets of the blog.
retriever = vectorstore.as_retriever()
prompt = hub.pull("rlm/rag-prompt")


def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)


rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

rag_chain.invoke("What is Task Decomposition?")
```

API Reference: WebBaseLoader | StrOutputParser | RunnablePassthrough | OpenAIEmbeddings | RecursiveCharacterTextSplitter

```
'Task Decomposition is a process where a complex task is broken down into
smaller, simpler steps or subtasks. This technique is utilized to enhance
model performance on complex tasks by making them more manageable. It can be
done by using language models with simple prompting, task-specific
instructions, or with human inputs.'
```

```
# cleanup
vectorstore.delete_collection()
```

Check out the LangSmith trace.

Detailed walkthrough

Let's go through the above code step-by-step to really understand what's going on.

1. Indexing: Load

We need to first load the blog post contents. We can use DocumentLoaders for this, which are objects that load in data from a source and return a list of Documents. A `Document` is an object with some `page_content` (str) and `metadata` (dict).

In this case we'll use the WebBaseLoader, which uses `urllib` to load HTML from web URLs and `BeautifulSoup` to parse it to text. We can customize the HTML -> text parsing by passing in parameters to the `BeautifulSoup` parser via `bs_kwargs` (see BeautifulSoup docs). In this case only HTML tags with class "post-content", "post-title", or "post-header" are relevant, so we'll remove all others.

```
import bs4
from langchain_community.document_loaders import WebBaseLoader

# Only keep post title, headers, and content from the full HTML.
bs4_strainer = bs4.SoupStrainer(class_=("post-title", "post-header", "post-
content"))
loader = WebBaseLoader(
    web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
```

```
    bs_kwargs={"parse_only": bs4_strainer},
)
docs = loader.load()

len(docs[0].page_content)
```

API Reference: WebBaseLoader

```
43131
```

```
print(docs[0].page_content[:500])
```

```
      LLM Powered Autonomous Agents

Date: June 23, 2023  |  Estimated Reading Time: 31 min  |  Author: Lilian Weng


Building agents with LLM (large language model) as its core controller is a
cool concept. Several proof-of-concepts demos, such as AutoGPT, GPT-Engineer
and BabyAGI, serve as inspiring examples. The potentiality of LLM extends
beyond generating well-written copies, stories, essays and programs; it can be
framed as a powerful general problem solver.
Agent System Overview#
In
```

Go deeper

`DocumentLoader`: Object that loads data from a source as list of `Documents`.

- Docs: Detailed documentation on how to use `DocumentLoaders`.
- Integrations: 160+ integrations to choose from.
- Interface: API reference for the base interface.

2. Indexing: Split

Our loaded document is over 42k characters long. This is too long to fit in the context window of many models. Even for those models that could fit the full post in their context window, models can struggle to find information in very long inputs.

To handle this we'll split the `Document` into chunks for embedding and vector storage. This should help us retrieve only the most relevant bits of the blog post at run time.

In this case we'll split our documents into chunks of 1000 characters with 200 characters of overlap between chunks. The overlap helps mitigate the possibility of separating a statement from important context related to it. We use the RecursiveCharacterTextSplitter, which will recursively split the document using common separators like new lines until each chunk is the appropriate size. This is the recommended text splitter for generic text use cases.

We set `add_start_index=True` so that the character index at which each split Document starts within the initial Document is preserved as metadata attribute "start_index".

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, add_start_index=True
)
all_splits = text_splitter.split_documents(docs)

len(all_splits)
```

API Reference: RecursiveCharacterTextSplitter

```
66
```

```
len(all_splits[0].page_content)
```

```
969
```

```
all_splits[10].metadata
```

```
{'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/',
 'start_index': 7056}
```

Go deeper

`TextSplitter`: Object that splits a list of `Document`s into smaller chunks. Subclass of `DocumentTransformer`s.

- Learn more about splitting text using different methods by reading the how-to docs
- Code (py or js)
- Scientific papers
- Interface: API reference for the base interface.

`DocumentTransformer`: Object that performs a transformation on a list of `Document` objects.

- Docs: Detailed documentation on how to use `DocumentTransformers`
- Integrations
- Interface: API reference for the base interface.

3. Indexing: Store

Now we need to index our 66 text chunks so that we can search over them at runtime. The most common way to do this is to embed the contents of each document split and insert these embeddings into a vector database (or vector store). When we want to search over our splits, we take a text search query, embed it, and perform some sort of “similarity” search to identify the stored splits with the most similar embeddings to our query embedding. The simplest similarity measure is cosine similarity — we measure the cosine of the angle between each pair of embeddings (which are high dimensional vectors).

We can embed and store all of our document splits in a single command using the Chroma vector store and OpenAIEmbeddings model.

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
```

```
vectorstore = Chroma.from_documents(documents=all_splits,
embedding=OpenAIEmbeddings())
```

API Reference: OpenAIEmbeddings

Go deeper

`Embeddings`: Wrapper around a text embedding model, used for converting text to embeddings.

- Docs: Detailed documentation on how to use embeddings.
- Integrations: 30+ integrations to choose from.
- Interface: API reference for the base interface.

`VectorStore`: Wrapper around a vector database, used for storing and querying embeddings.

- Docs: Detailed documentation on how to use vector stores.
- Integrations: 40+ integrations to choose from.
- Interface: API reference for the base interface.

This completes the **Indexing** portion of the pipeline. At this point we have a query-able vector store containing the chunked contents of our blog post. Given a user question, we should ideally be able to return the snippets of the blog post that answer the question.

4. Retrieval and Generation: Retrieve

Now let's write the actual application logic. We want to create a simple application that takes a user question, searches for documents relevant to that question, passes the retrieved documents and initial question to a model, and returns an answer.

First we need to define our logic for searching over documents. LangChain defines a Retriever interface which wraps an index that can return relevant `Documents` given a string query.

The most common type of `Retriever` is the VectorStoreRetriever, which uses the similarity search capabilities of a vector store to facilitate retrieval. Any `VectorStore` can easily be turned into a `Retriever` with `VectorStore.as_retriever()`:

```
retriever = vectorstore.as_retriever(search_type="similarity", search_kwargs={"k": 6})

retrieved_docs = retriever.invoke("What are the approaches to Task Decomposition?")

len(retrieved_docs)
```

```
6
```

```
print(retrieved_docs[0].page_content)
```

```
Tree of Thoughts (Yao et al. 2023) extends CoT by exploring multiple reasoning possibilities at each step. It first decomposes the problem into multiple thought steps and generates multiple thoughts per step, creating a tree structure. The search process can be BFS (breadth-first search) or DFS (depth-first search) with each state evaluated by a classifier (via a prompt) or majority vote.
Task decomposition can be done (1) by LLM with simple prompting like "Steps for XYZ.\n1.", "What are the subgoals for achieving XYZ?", (2) by using task-specific instructions; e.g. "Write a story outline." for writing a novel, or (3) with human inputs.
```

Go deeper

Vector stores are commonly used for retrieval, but there are other ways to do retrieval, too.

`Retriever`: An object that returns `Document`s given a text query

- Docs: Further documentation on the interface and built-in retrieval techniques. Some of which include:
  - `MultiQueryRetriever` generates variants of the input question to improve retrieval hit rate.

- `MultiVectorRetriever` instead generates variants of the embeddings, also in order to improve retrieval hit rate.
- `Max marginal relevance` selects for relevance and diversity among the retrieved documents to avoid passing in duplicate context.
- Documents can be filtered during vector store retrieval using metadata filters, such as with a Self Query Retriever.

- Integrations: Integrations with retrieval services.
- Interface: API reference for the base interface.

5. Retrieval and Generation: Generate

Let's put it all together into a chain that takes a question, retrieves relevant documents, constructs a prompt, passes that to a model, and parses the output.

We'll use the gpt-4o-mini OpenAI chat model, but any LangChain `LLM` or `ChatModel` could be substituted in.

OpenAI | Anthropic | Azure | Google | Cohere | NVIDIA | FireworksAI | Gr

```
pip install -qU langchain-openai
```

```
import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass()

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")
```

We'll use a prompt for RAG that is checked into the LangChain prompt hub (here).

```
from langchain import hub

prompt = hub.pull("rlm/rag-prompt")

example_messages = prompt.invoke(
    {"context": "filler context", "question": "filler question"}
).to_messages()

example_messages
```

```
[HumanMessage(content="You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Use three sentences maximum and keep the answer concise.\nQuestion: filler question \nContext: filler context \nAnswer:")]
```

```
print(example_messages[0].content)
```

```
You are an assistant for question-answering tasks. Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Use three sentences maximum and keep the answer concise.
Question: filler question
Context: filler context
Answer:
```

We'll use the LCEL Runnable protocol to define the chain, allowing us to

- pipe together components and functions in a transparent way
- automatically trace our chain in LangSmith
- get streaming, async, and batched calling out of the box.

Here is the implementation:

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
```

```
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)


rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)

for chunk in rag_chain.stream("What is Task Decomposition?"):
    print(chunk, end="", flush=True)
```

API Reference: StrOutputParser | RunnablePassthrough

```
Task Decomposition is a process where a complex task is broken down into
smaller, more manageable steps or parts. This is often done using techniques
like "Chain of Thought" or "Tree of Thoughts", which instruct a model to
"think step by step" and transform large tasks into multiple simple tasks.
Task decomposition can be prompted in a model, guided by task-specific
instructions, or influenced by human inputs.
```

Let's dissect the LCEL to understand what's going on.

First: each of these components (`retriever`, `prompt`, `llm`, etc.) are instances of Runnable. This means that they implement the same methods--such as sync and async `.invoke`, `.stream`, or `.batch`--which makes them easier to connect together. They can be connected into a RunnableSequence--another Runnable--via the `|` operator.

LangChain will automatically cast certain objects to runnables when met with the `|` operator. Here, `format_docs` is cast to a RunnableLambda, and the dict with `"context"` and `"question"` is cast to a RunnableParallel. The details are less important than the bigger point, which is that each object is a Runnable.

Let's trace how the input question flows through the above runnables.

As we've seen above, the input to `prompt` is expected to be a dict with keys `"context"` and `"question"`. So the first element of this chain builds runnables that will calculate both of

these from the input question:

- `retriever | format_docs` passes the question through the retriever, generating Document objects, and then to `format_docs` to generate strings;
- `RunnablePassthrough()` passes through the input question unchanged.

That is, if you constructed

```
chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
)
```

Then `chain.invoke(question)` would build a formatted prompt, ready for inference. (Note: when developing with LCEL, it can be practical to test with sub-chains like this.)

The last steps of the chain are `llm`, which runs the inference, and `StrOutputParser()`, which just plucks the string content out of the LLM's output message.

You can analyze the individual steps of this chain via its LangSmith trace.

Built-in chains

If preferred, LangChain includes convenience functions that implement the above LCEL. We compose two functions:

- create_stuff_documents_chain specifies how retrieved context is fed into a prompt and LLM. In this case, we will "stuff" the contents into the prompt -- i.e., we will include all retrieved context without any summarization or other processing. It largely implements our above `rag_chain`, with input keys `context` and `input` -- it generates an answer using retrieved context and query.
- create_retrieval_chain adds the retrieval step and propagates the retrieved context through the chain, providing it alongside the final answer. It has input key `input`, and includes `input`, `context`, and `answer` in its output.

```python
from langchain.chains import create_retrieval_chain
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate

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

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),
        ("human", "{input}"),
    ]
)


question_answer_chain = create_stuff_documents_chain(llm, prompt)
rag_chain = create_retrieval_chain(retriever, question_answer_chain)

response = rag_chain.invoke({"input": "What is Task Decomposition?"})
print(response["answer"])
```

API Reference: create_retrieval_chain | create_stuff_documents_chain | ChatPromptTemplate

```
Task Decomposition is a process in which complex tasks are broken down into
smaller and simpler steps. Techniques like Chain of Thought (CoT) and Tree of
Thoughts are used to enhance model performance on these tasks. The CoT method
instructs the model to think step by step, decomposing hard tasks into
manageable ones, while Tree of Thoughts extends CoT by exploring multiple
reasoning possibilities at each step, creating a tree structure of thoughts.
```

Returning sources

Often in Q&A applications it's important to show users the sources that were used to generate the answer. LangChain's built-in `create_retrieval_chain` will propagate retrieved source documents through to the output in the `"context"` key:

```
for document in response["context"]:
    print(document)
    print()
```

```
page_content='Fig. 1. Overview of a LLM-powered autonomous agent
system.\nComponent One: Planning#\nA complicated task usually involves many
steps. An agent needs to know what they are and plan ahead.\nTask
Decomposition#\nChain of thought (CoT; Wei et al. 2022) has become a standard
prompting technique for enhancing model performance on complex tasks. The
model is instructed to “think step by step” to utilize more test-time
computation to decompose hard tasks into smaller and simpler steps. CoT
transforms big tasks into multiple manageable tasks and shed lights into an
interpretation of the model’s thinking process.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/'}

page_content='Fig. 1. Overview of a LLM-powered autonomous agent
system.\nComponent One: Planning#\nA complicated task usually involves many
steps. An agent needs to know what they are and plan ahead.\nTask
Decomposition#\nChain of thought (CoT; Wei et al. 2022) has become a standard
prompting technique for enhancing model performance on complex tasks. The
model is instructed to “think step by step” to utilize more test-time
computation to decompose hard tasks into smaller and simpler steps. CoT
transforms big tasks into multiple manageable tasks and shed lights into an
interpretation of the model’s thinking process.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 1585}

page_content='Tree of Thoughts (Yao et al. 2023) extends CoT by exploring
multiple reasoning possibilities at each step. It first decomposes the problem
into multiple thought steps and generates multiple thoughts per step, creating
a tree structure. The search process can be BFS (breadth-first search) or DFS
(depth-first search) with each state evaluated by a classifier (via a prompt)
or majority vote.\nTask decomposition can be done (1) by LLM with simple
prompting like "Steps for XYZ.\\n1.", "What are the subgoals for achieving
XYZ?", (2) by using task-specific instructions; e.g. "Write a story outline."
for writing a novel, or (3) with human inputs.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 2192}

page_content='Tree of Thoughts (Yao et al. 2023) extends CoT by exploring
multiple reasoning possibilities at each step. It first decomposes the problem
into multiple thought steps and generates multiple thoughts per step, creating
a tree structure. The search process can be BFS (breadth-first search) or DFS
(depth-first search) with each state evaluated by a classifier (via a prompt)
or majority vote.\nTask decomposition can be done (1) by LLM with simple
prompting like "Steps for XYZ.\\n1.", "What are the subgoals for achieving
XYZ?", (2) by using task-specific instructions; e.g. "Write a story outline."
```

```
for writing a novel, or (3) with human inputs.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/'}

page_content='Resources:\n1. Internet access for searches and information
gathering.\n2. Long Term memory management.\n3. GPT-3.5 powered Agents for
delegation of simple tasks.\n4. File output.\n\nPerformance Evaluation:\n1.
Continuously review and analyze your actions to ensure you are performing to
the best of your abilities.\n2. Constructively self-criticize your big-picture
behavior constantly.\n3. Reflect on past decisions and strategies to refine
your approach.\n4. Every command has a cost, so be smart and efficient. Aim to
complete tasks in the least number of steps.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/'}

page_content='Resources:\n1. Internet access for searches and information
gathering.\n2. Long Term memory management.\n3. GPT-3.5 powered Agents for
delegation of simple tasks.\n4. File output.\n\nPerformance Evaluation:\n1.
Continuously review and analyze your actions to ensure you are performing to
the best of your abilities.\n2. Constructively self-criticize your big-picture
behavior constantly.\n3. Reflect on past decisions and strategies to refine
your approach.\n4. Every command has a cost, so be smart and efficient. Aim to
complete tasks in the least number of steps.' metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 29630}
```

Go deeper

Choosing a model

`ChatModel`: An LLM-backed chat model. Takes in a sequence of messages and returns a message.

- Docs
- Integrations: 25+ integrations to choose from.
- Interface: API reference for the base interface.

`LLM`: A text-in-text-out LLM. Takes in a string and returns a string.

- Docs
- Integrations: 75+ integrations to choose from.
- Interface: API reference for the base interface.

See a guide on RAG with locally-running models here.

Customizing the prompt

As shown above, we can load prompts (e.g., this RAG prompt) from the prompt hub. The prompt can also be easily customized:

```
from langchain_core.prompts import PromptTemplate

template = """Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
Use three sentences maximum and keep the answer as concise as possible.
Always say "thanks for asking!" at the end of the answer.

{context}

Question: {question}

Helpful Answer:"""
custom_rag_prompt = PromptTemplate.from_template(template)

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | custom_rag_prompt
    | llm
    | StrOutputParser()
)

rag_chain.invoke("What is Task Decomposition?")
```

API Reference: PromptTemplate

```
'Task decomposition is the process of breaking down a complex task into smaller, more manageable parts. Techniques like Chain of Thought (CoT) and Tree of Thoughts allow an agent to "think step by step" and explore multiple reasoning possibilities, respectively. This process can be executed by a Language Model with simple prompts, task-specific instructions, or human inputs. Thanks for asking!'
```

Check out the LangSmith trace

Next steps

We've covered the steps to build a basic Q&A app over data:

- Loading data with a Document Loader
- Chunking the indexed data with a Text Splitter to make it more easily usable by a model
- Embedding the data and storing the data in a vectorstore
- Retrieving the previously stored chunks in response to incoming questions
- Generating an answer using the retrieved chunks as context

There's plenty of features, integrations, and extensions to explore in each of the above sections. Along from the **Go deeper** sources mentioned above, good next steps include:

- Return sources: Learn how to return source documents
- Streaming: Learn how to stream outputs and intermediate steps
- Add chat history: Learn how to add chat history to your app
- Retrieval conceptual guide: A high-level overview of specific retrieval techniques
- Build a local RAG application: Create an app similar to the one above using all local components

Edit this page

Was this page helpful?

You can also leave detailed feedback on GitHub.