Open in Colab
Open on GitHub

Build a Retrieval Augmented Generation (RAG) App: Part 1

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 is a multi-part tutorial:

- Part 1 (this guide) introduces RAG and walks through a minimal implementation.
- Part 2 extends the implementation to accommodate conversation-style interactions and multi-step retrieval processes.

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 techniques.

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.

Overview

A typical RAG application has two main components:

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

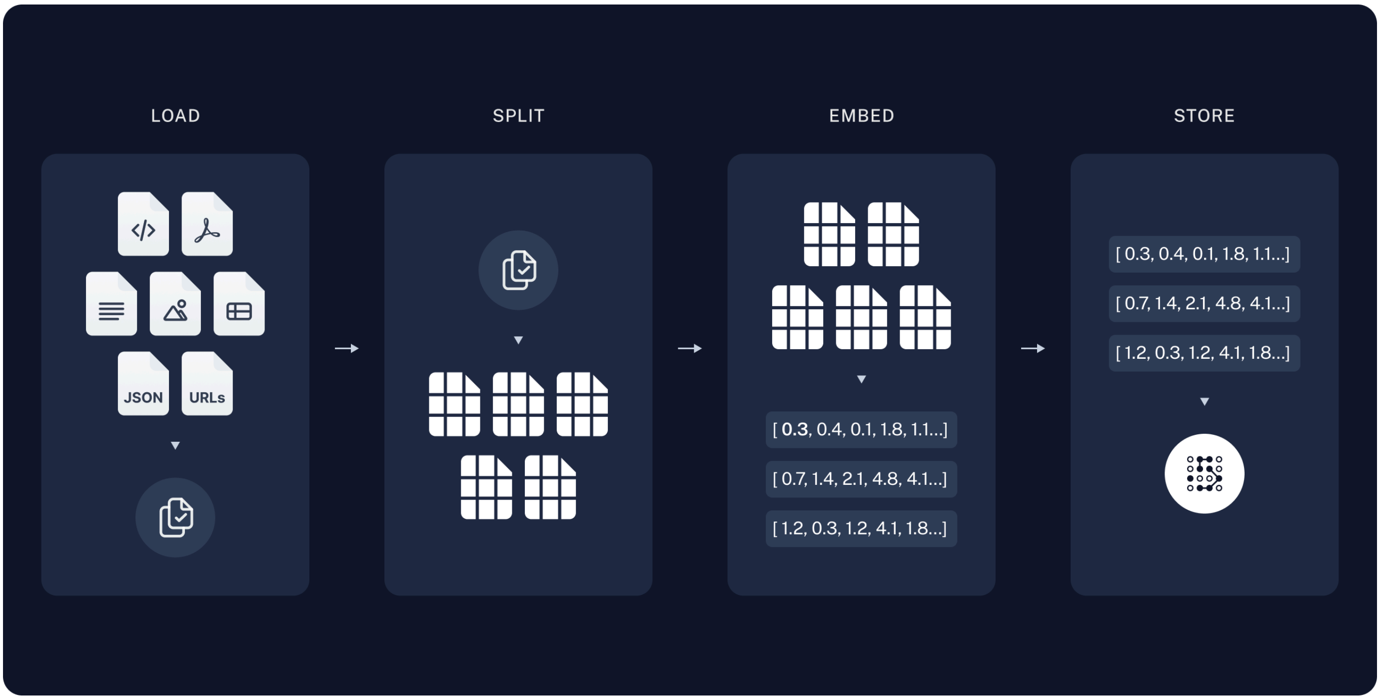
Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model.

Note: the indexing portion of this tutorial will largely follow the semantic search tutorial.

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

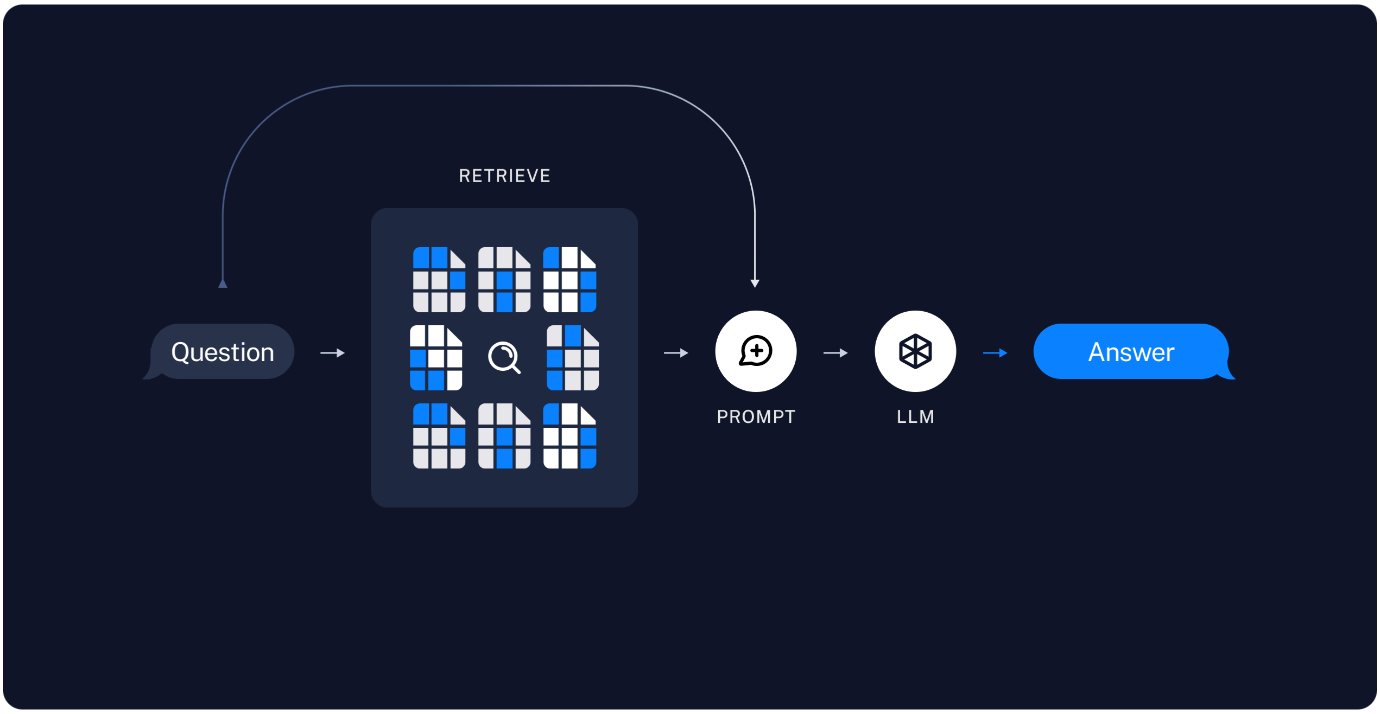
Indexing

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 passing it into a model, as 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 be searched over later. 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 both the question with the retrieved data



Once we've indexed our data, we will use LangGraph as our orchestration framework to implement the retrieval and generation steps.

Setup

Jupyter Notebook

This and other tutorials are perhaps most conveniently run in a Jupyter notebooks. Going through guides in an interactive environment is a great way to better understand them. See here for instructions on how to install.

Installation

This tutorial requires these langchain dependencies:

Pip Conda

```
%pip install --quiet --upgrade langchain-text-splitters langchain-community
langgraph
```

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 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 LANGSMITH_TRACING="true"
export LANGSMITH_API_KEY="..."
```

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

```
import getpass
import os

os.environ["LANGSMITH_TRACING"] = "true"
os.environ["LANGSMITH_API_KEY"] = getpass.getpass()
```

Components

We will need to select three components from LangChain's suite of integrations.

Select chat model: OpenAI ▾

```
pip install -qU "langchain[openai]"
```

```
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
  os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI:
")

from langchain.chat_models import init_chat_model

llm = init_chat_model("gpt-4o-mini", model_provider="openai")
```

Select embeddings model: OpenAI ▾

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

```
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
  os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter API key for OpenAI:
")

from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")
```

Select vector store: In-memory ▾

```
pip install -qU langchain-core
```

```
from langchain_core.vectorstores import InMemoryVectorStore

vector_store = InMemoryVectorStore(embeddings)
```

Preview

In this guide we'll build an app that answers questions about the website's content. 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 ~50 lines of code.

```
import bs4
from langchain import hub
from langchain_community.document_loaders import WebBaseLoader
from langchain_core.documents import Document
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langgraph.graph import START, StateGraph
from typing_extensions import List, TypedDict

# Load and chunk 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)
all_splits = text_splitter.split_documents(docs)

# Index chunks
_ = vector_store.add_documents(documents=all_splits)

# Define prompt for question-answering
prompt = hub.pull("rlm/rag-prompt")


# Define state for application
class State(TypedDict):
    question: str
    context: List[Document]
    answer: str


# Define application steps
def retrieve(state: State):
```

```
    retrieved_docs = vector_store.similarity_search(state["question"])
    return {"context": retrieved_docs}


def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context":
docs_content})
    response = llm.invoke(messages)
    return {"answer": response.content}


# Compile application and test
graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()
```

API Reference: hub | WebBaseLoader | Document | RecursiveCharacterTextSplitter | StateGraph

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

```
Task Decomposition is the process of breaking down a complicated task into
smaller, manageable steps to facilitate easier execution and understanding.
Techniques like Chain of Thought (CoT) and Tree of Thoughts (ToT) guide models
to think step-by-step, allowing them to explore multiple reasoning
possibilities. This method enhances performance on complex tasks and provides
insight into the model's thinking process.
```

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

NOTE

This section is an abbreviated version of the content in the semantic search tutorial. If you're comfortable with document loaders, embeddings, and vector stores, feel free to skip to the next section on retrieval and generation.

Loading documents

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 Document objects.

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 into 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()

assert len(docs) == 1
print(f"Total characters: {len(docs[0].page_content)}")
```

API Reference: WebBaseLoader

```
Total characters: 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.

Splitting documents

Our loaded document is over 42k characters which is too long to fit into 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 parts of the blog post at run time.

As in the semantic search tutorial, we use a 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.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # chunk size (characters)
```

```python
    chunk_overlap=200,  # chunk overlap (characters)
    add_start_index=True,  # track index in original document
)
all_splits = text_splitter.split_documents(docs)

print(f"Split blog post into {len(all_splits)} sub-documents.")
```

API Reference: RecursiveCharacterTextSplitter

```
Split blog post into 66 sub-documents.
```

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.

Storing documents

Now we need to index our 66 text chunks so that we can search over them at runtime. Following the semantic search tutorial, our approach is to embed the contents of each document split and insert these embeddings into a vector store. Given an input query, we can then use vector search to retrieve relevant documents.

We can embed and store all of our document splits in a single command using the vector store and embeddings model selected at the start of the tutorial.

```
document_ids = vector_store.add_documents(documents=all_splits)

print(document_ids[:3])
```

```
['07c18af6-ad58-479a-bfb1-d508033f9c64', '9000bf8e-1993-446f-8d4d-f4e507ba4b8f', 'ba3b5d14-bed9-4f5f-88be-44c88aedc2e6']
```

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.

2. Retrieval and Generation

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.

For generation, we will use the chat model selected at the start of the tutorial.

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

```python
from langchain import hub

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

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

assert len(example_messages) == 1
print(example_messages[0].content)
```

API Reference: hub

```
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: (question goes here)
Context: (context goes here)
Answer:
```

We'll use LangGraph to tie together the retrieval and generation steps into a single application. This will bring a number of benefits:

- We can define our application logic once and automatically support multiple invocation modes, including streaming, async, and batched calls.
- We get streamlined deployments via LangGraph Platform.
- LangSmith will automatically trace the steps of our application together.
- We can easily add key features to our application, including persistence and human-in-the-loop approval, with minimal code changes.

To use LangGraph, we need to define three things:

1. The state of our application;
2. The nodes of our application (i.e., application steps);
3. The "control flow" of our application (e.g., the ordering of the steps).

State:

The **state** of our application controls what data is input to the application, transferred between steps, and output by the application. It is typically a `TypedDict`, but can also be a Pydantic BaseModel.

For a simple RAG application, we can just keep track of the input question, retrieved context, and generated answer:

```
from langchain_core.documents import Document
from typing_extensions import List, TypedDict


class State(TypedDict):
    question: str
    context: List[Document]
    answer: str
```

API Reference: Document

Nodes (application steps)

Let's start with a simple sequence of two steps: retrieval and generation.

```
def retrieve(state: State):
    retrieved_docs = vector_store.similarity_search(state["question"])
    return {"context": retrieved_docs}


def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context":
docs_content})
    response = llm.invoke(messages)
    return {"answer": response.content}
```

Our retrieval step simply runs a similarity search using the input question, and the generation step formats the retrieved context and original question into a prompt for the chat model.

Control flow

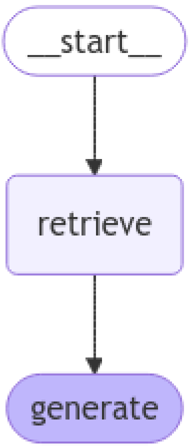
Finally, we compile our application into a single `graph` object. In this case, we are just connecting the retrieval and generation steps into a single sequence.

```
from langgraph.graph import START, StateGraph

graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()
```

API Reference: StateGraph

LangGraph also comes with built-in utilities for visualizing the control flow of your application:

```
from IPython.display import Image, display

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



▸ Do I need to use LangGraph?

Usage

Let's test our application! LangGraph supports multiple invocation modes, including sync, async, and streaming.

Invoke:

```python
result = graph.invoke({"question": "What is Task Decomposition?"})

print(f'Context: {result["context"]}\n\n')
print(f'Answer: {result["answer"]}')
```

```
Context: [Document(id='a42dc78b-8f76-472a-9e25-180508af74f3', metadata=
{'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/',
'start_index': 1585}, 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.'), Document(id='c0e45887-
d0b0-483d-821a-bb5d8316d51d', 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.'), Document(id='4cc7f318-35f5-
440f-a4a4-145b5f0b918d', metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 29630},
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.'), Document(id='f621ade4-9b0d-
471f-a522-44eb5feeba0c', metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 19373},
page_content="(3) Task execution: Expert models execute on the specific tasks
and log results.\nInstruction:\n\nWith the input and the inference results,
the AI assistant needs to describe the process and results. The previous
stages can be formed as - User Input: {{ User Input }}, Task Planning: {{
Tasks }}, Model Selection: {{ Model Assignment }}, Task Execution: {{
Predictions }}. You must first answer the user's request in a straightforward
```

```
manner. Then describe the task process and show your analysis and model
inference results to the user in the first person. If inference results
contain a file path, must tell the user the complete file path.")]


Answer: Task decomposition is a technique used to break down complex tasks
into smaller, manageable steps, allowing for more efficient problem-solving.
This can be achieved through methods like chain of thought prompting or the
tree of thoughts approach, which explores multiple reasoning possibilities at
each step. It can be initiated through simple prompts, task-specific
instructions, or human inputs.
```

Stream steps:

```
for step in graph.stream(
    {"question": "What is Task Decomposition?"}, stream_mode="updates"
):
    print(f"{step}\n\n----------------\n")
```

```
{'retrieve': {'context': [Document(id='a42dc78b-8f76-472a-9e25-180508af74f3',
metadata={'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/',
'start_index': 1585}, 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.'), Document(id='c0e45887-
d0b0-483d-821a-bb5d8316d51d', 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.'), Document(id='4cc7f318-35f5-
440f-a4a4-145b5f0b918d', metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 29630},
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.'), Document(id='f621ade4-9b0d-
471f-a522-44eb5feeba0c', metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 19373},
page_content="(3) Task execution: Expert models execute on the specific tasks
and log results.\nInstruction:\n\nWith the input and the inference results,
the AI assistant needs to describe the process and results. The previous
stages can be formed as - User Input: {{ User Input }}, Task Planning: {{
Tasks }}, Model Selection: {{ Model Assignment }}, Task Execution: {{
Predictions }}. You must first answer the user's request in a straightforward
manner. Then describe the task process and show your analysis and model
inference results to the user in the first person. If inference results
contain a file path, must tell the user the complete file path.")]}}

----------------

{'generate': {'answer': 'Task decomposition is the process of breaking down a
complex task into smaller, more manageable steps. This technique, often
enhanced by methods like Chain of Thought (CoT) or Tree of Thoughts, allows
models to reason through tasks systematically and improves performance by
clarifying the thought process. It can be achieved through simple prompts,
task-specific instructions, or human inputs.'}}

----------------
```

Stream tokens:

```
for message, metadata in graph.stream(
    {"question": "What is Task Decomposition?"}, stream_mode="messages"
):
    print(message.content, end="|")
```

```
|Task| decomposition| is| the| process| of| breaking| down| complex| tasks|
into| smaller|,| more| manageable| steps|.| It| can| be| achieved| through|
techniques| like| Chain| of| Thought| (|Co|T|)| prompting|,| which|
encourages| the| model| to| think| step| by| step|,| or| through| more|
structured| methods| like| the| Tree| of| Thoughts|.| This| approach| not|
```

```
only| simplifies| task| execution| but| also| provides| insights| into| the|
model|'s| reasoning| process|.||
```

TIP

For async invocations, use:

```
result = await graph.ainvoke(...)
```

and

```
async for step in graph.astream(...):
```

Returning sources

Note that by storing the retrieved context in the state of the graph, we recover sources for the model's generated answer in the `"context"` field of the state. See this guide on returning sources for more detail.

Go deeper

Chat models take in a sequence of messages and return a message.

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

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. For example:

```
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)
```

API Reference: PromptTemplate

Query analysis

So far, we are executing the retrieval using the raw input query. However, there are some advantages to allowing a model to generate the query for retrieval purposes. For example:

- In addition to semantic search, we can build in structured filters (e.g., "Find documents since the year 2020.");
- The model can rewrite user queries, which may be multifaceted or include irrelevant language, into more effective search queries.

Query analysis employs models to transform or construct optimized search queries from raw user input. We can easily incorporate a query analysis step into our application. For illustrative purposes, let's add some metadata to the documents in our vector store. We will add some (contrived) sections to the document which we can filter on later.

```
total_documents = len(all_splits)
third = total_documents // 3

for i, document in enumerate(all_splits):
    if i < third:
        document.metadata["section"] = "beginning"
    elif i < 2 * third:
        document.metadata["section"] = "middle"
    else:
        document.metadata["section"] = "end"
```

```
all_splits[0].metadata
```

```
{'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/',
 'start_index': 8,
 'section': 'beginning'}
```

We will need to update the documents in our vector store. We will use a simple InMemoryVectorStore for this, as we will use some of its specific features (i.e., metadata filtering). Refer to the vector store integration documentation for relevant features of your chosen vector store.

```
from langchain_core.vectorstores import InMemoryVectorStore

vector_store = InMemoryVectorStore(embeddings)
_ = vector_store.add_documents(all_splits)
```

API Reference: InMemoryVectorStore

Let's next define a schema for our search query. We will use structured output for this purpose. Here we define a query as containing a string query and a document section (either "beginning", "middle", or "end"), but this can be defined however you like.

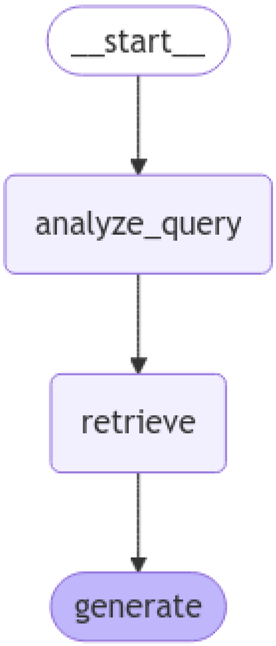
```
from typing import Literal

from typing_extensions import Annotated


class Search(TypedDict):
    """Search query."""

    query: Annotated[str, ..., "Search query to run."]
    section: Annotated[
        Literal["beginning", "middle", "end"],
        ...,
        "Section to query.",
    ]
```

Finally, we add a step to our LangGraph application to generate a query from the user's raw input:

```
class State(TypedDict):
    question: str
    query: Search
    context: List[Document]
    answer: str


def analyze_query(state: State):
    structured_llm = llm.with_structured_output(Search)
    query = structured_llm.invoke(state["question"])
    return {"query": query}


def retrieve(state: State):
    query = state["query"]
    retrieved_docs = vector_store.similarity_search(
        query["query"],
        filter=lambda doc: doc.metadata.get("section") == query["section"],
    )
    return {"context": retrieved_docs}


def generate(state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context":
docs_content})
    response = llm.invoke(messages)
    return {"answer": response.content}


graph_builder = StateGraph(State).add_sequence([analyze_query, retrieve,
generate])
graph_builder.add_edge(START, "analyze_query")
graph = graph_builder.compile()
```

▸ Full Code:

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



We can test our implementation by specifically asking for context from the end of the post. Note that the model includes different information in its answer.

```
for step in graph.stream(
    {"question": "What does the end of the post say about Task 
Decomposition?"},
    stream_mode="updates",
):
    print(f"{step}\n\n----------------\n")
```

```
{'analyze_query': {'query': {'query': 'Task Decomposition', 'section': 
'end'}}}

----------------

{'retrieve': {'context': [Document(id='d6cef137-e1e8-4ddc-91dc-b62bd33c6020', 
metadata={'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/', 
'start_index': 39221, 'section': 'end'}, page_content='Finite context length: 
The restricted context capacity limits the inclusion of historical 
information, detailed instructions, API call context, and responses. The 
design of the system has to work with this limited communication bandwidth, 
while mechanisms like self-reflection to learn from past mistakes would 
benefit a lot from long or infinite context windows. Although vector stores 
and retrieval can provide access to a larger knowledge pool, their 
representation power is not as powerful as full attention.\n\n\nChallenges in 
long-term planning and task decomposition: Planning over a lengthy history and 
effectively exploring the solution space remain challenging. LLMs struggle to 
adjust plans when faced with unexpected errors, making them less robust
```

```
compared to humans who learn from trial and error.'), Document(id='d1834ae1-
eb6a-43d7-a023-08dfa5028799', metadata={'source':
'https://lilianweng.github.io/posts/2023-06-23-agent/', 'start_index': 39086,
'section': 'end'}, page_content='}\n]\nChallenges#\nAfter going through key
ideas and demos of building LLM-centered agents, I start to see a couple
common limitations:'), Document(id='ca7f06e4-2c2e-4788-9a81-2418d82213d9',
metadata={'source': 'https://lilianweng.github.io/posts/2023-06-23-agent/',
'start_index': 32942, 'section': 'end'}, page_content='}\n]\nThen after these
clarification, the agent moved into the code writing mode with a different
system message.\nSystem message:'), Document(id='1fcc2736-30f4-4ef6-90f2-
c64af92118cb', metadata={'source': 'https://lilianweng.github.io/posts/2023-
06-23-agent/', 'start_index': 35127, 'section': 'end'},
page_content='"content": "You will get instructions for code to write.\\nYou
will write a very long answer. Make sure that every detail of the architecture
is, in the end, implemented as code.\\nMake sure that every detail of the
architecture is, in the end, implemented as code.\\n\\nThink step by step and
reason yourself to the right decisions to make sure we get it right.\\nYou
will first lay out the names of the core classes, functions, methods that will
be necessary, as well as a quick comment on their purpose.\\n\\nThen you will
output the content of each file including ALL code.\\nEach file must strictly
follow a markdown code block format, where the following tokens must be
replaced such that\\nFILENAME is the lowercase file name including the file
extension,\\nLANG is the markup code block language for the code\'s language,
and CODE is the code:\\n\\nFILENAME\\n`\`\`LANG\\nCODE\\n`\`\`\\n\\nYou will
start with the \\"entrypoint\\" file, then go to the ones that are imported by
that file, and so on.\\nPlease')]}}

----------------

{'generate': {'answer': 'The end of the post highlights that task
decomposition faces challenges in long-term planning and adapting to
unexpected errors. LLMs struggle with adjusting their plans, making them less
robust compared to humans who learn from trial and error. This indicates a
limitation in effectively exploring the solution space and handling complex
tasks.'}}

----------------
```

In both the streamed steps and the LangSmith trace, we can now observe the structured query that was fed into the retrieval step.

Query Analysis is a rich problem with a wide range of approaches. Refer to the how-to guides for more examples.

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.

In Part 2 of the tutorial, we will extend the implementation here to accommodate conversation-style interactions and multi-step retrieval processes.

Further reading:

- 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

13 comments – *powered by giscus*

Oldest Newest

ksairos Dec 17, 2024

If you're using Chroma during the query analysis, don't forget to change the filter for `vector_store.similarity_search()` function in the `def retrieve(state: State)` node.

Instead of `filter=lambda doc: doc.metadata.get("section") == query["section"]`, which causes a `Expected type 'dict[str, str] | None', got '(doc: Any) -> bool' instead` error, you can use `{"section": query["section"]}`.

↑ 1 👍 2 0 replies

korpog Jan 1

You have to run `pip install beautifulsoup4`, otherwise there will be an import error

↑ 1 👍 3 0 replies

JoshuaMichaelHanson Jan 7

On Windows 11 running Python 3.13.1 needed to change to a pydantic BaseModel to get the analyze_query to work. Keep getting ValueError: no signature found for builtin type <class 'dict'> when running the query = structured_llm.invoke(**state**["question"])

Updated code as follows and works. The answer is slightly different but by adjusting the query was able to come close

from typing import Literal
from pydantic import BaseModel, Field

Define the Search schema using Pydantic instead of TypedDict

class Search(BaseModel):
"""Search query schema."""
query: str = Field(description="Search query to run.")
section: Literal["beginning", "middle", "end"] = Field(
description="Section to query."
)

for step in graph.stream(
{"question": "What are Challenges of task decomposition say at the end?"},
stream_mode="updates",
):
print(f"{step}\n\n----------------\n")

If anyone knows how to make it work with the TypeDict vs BaseModel would be interested in the answer, otherwise hope this helps someone else if they get stuck. Cheers!

↑ 1 👍 1 0 replies

0JCRG0 Jan 14

The documentation is not up-to-date:

Traceback (most recent call last):
File "/Users/juanreyesgarcia/Dev/Python/LLMSandbox/main.py", line 58, in
graph_builder = StateGraph(State).add_sequence([retrieve, generate]) # type: ignore
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'StateGraph' object has no attribute 'add_sequence'

↑ 1 3 replies

JoshuaMichaelHanson Jan 14

Are you sure you have all the correct dependencies installed? Just did the tutorial a week ago and everything worked fine-ish, see other comment.

pip install -U langgraph
pip show langgraph
from langgraph.graph import START, StateGraph

If no error occurs, the package is installed correctly

pip check langgraph

make sure you are using a virtual environment to manage dependencies cleanly

python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate
pip install -U langgraph

0JCRG0 Jan 14

It is probably the version? This is what I have `langgraph==0.2.39`

But this worked:

But this worked:

```
graph_builder = StateGraph(State)
graph_builder.add_node("retrieve", retrieve)
graph_builder.add_node("generate", generate)

graph_builder.add_edge(START, "retrieve")
graph_builder.add_edge("retrieve", "generate")
graph_builder.add_edge("generate", END)
graph = graph_builder.compile()

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

ralphy1976 23 days ago

i just followed the tutorial, i had no issues with the "graph_builder = StateGraph(State).add_sequence([retrieve, generate])" ONCE i remembered to install langgraph....

kgasioro Feb 14

Use of an NVIDIA chat models did not work for me, as returned message indicated model_provider='nvidia' is not supported for init_chat_model() using latest version, even though online documentation states that it is.

↑ 1 0 replies

LiuDiliDili Feb 23

for step in graph.stream(
{"question": "What does the end of the post say about Task Decomposition?"},
stream_mode="updates",
):
print(f"{step}\n\n----------------\n")
NotImplementedError (When following this tutorial, I encountered an error when executing the last piece of code mentioned above.)
My LangChain version as below：
langchain 0.3.19
langchain-community 0.3.18
langchain-core 0.3.37
langchain-text-splitters 0.3.6
langchain-unstructured 0.1.6
langdetect 1.0.9
langgraph 0.2.74
langgraph-checkpoint 2.0.16
langgraph-sdk 0.1.53
langsmith 0.3.8

↑ 1 · 2 replies

LiuDiliDili Feb 23

\langchain\Lib\site-packages\langchain_core\language_models\chat_models.py:1189, in BaseChatModel.bind_tools(self, tools, **kwargs)
1182 def bind_tools(
1183 self,
1184 tools: Sequence[
(...)
1187 **kwargs: Any,
1188) -> Runnable[LanguageModelInput, BaseMessage]:
-> 1189 raise NotImplementedError

LiuDiliDili Feb 24

llm = ChatOpenAI(
temperature=0.95,
model="GLM-4-Air",
openai_api_key="***",
openai_api_base="https://open.bigmodel.cn/api/paas/v4/",
model_kwargs={ "response_format": { "type": "json_object" } }
)
ChatSparkLLM still does not support bind_tools. Using ZhiPuAi in this way is useful. I didn't encountered this mistake again.

mg1094 Mar 16

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")
)
),
)
Error:HTTPSConnectionPool(host='lilianweng.github.io', port=443): Max retries exceeded with url: /posts/2023-06-23-agent/ (Caused by ProtocolError('Connection aborted.', ConnectionResetError(54, 'Connection reset by peer')))

↑ 1 · 0 replies

devo-q-fontaine Apr 7

When running this code:

```
****async for step in app.astream(
    {"contents": [doc.page_content for doc in split_docs]},
    {"recursion_limit": 10},
):
    print(list(step.keys()))****
```

I get the following error:

```
AttributeError                            Traceback (most recent call last)
Cell In[68], line 1
----> 1 async for step in app.astream(
      2     {"contents": [doc.page_content for doc in split_docs]},
      3     {"recursion_limit": 10},
      4 ):
      5     print(list(step.keys()))

File c:\Users\qfontaine\rag_env\Lib\site-packages\langgraph\pregel\__init__.py:2626, in Pregel.as
   2620 # Similarly to Bulk Synchronous Parallel / Pregel model
   2621 # computation proceeds in steps, while there are channel updates
   2622 # channel updates from step N are only visible in step N+1
   2623 # channels are guaranteed to be immutable for the duration of the step,
   2624 # with channel updates applied only at the transition between steps
   2625 while loop.tick(input_keys=self.input_channels):
-> 2626     async for _ in runner.atick(
   2627         loop.tasks.values(),
   2628         timeout=self.step_timeout,
   2629         retry_policy=self.retry_policy,
   2630         get_waiter=get_waiter,
   2631     ):
   2632         # emit output
   2633         for o in output():
   2634             yield o
...
     94             return model_encoding_name
     96 if encoding_name is None:

AttributeError: 'NoneType' object has no attribute 'startswith'
During task with name 'collect_summaries' and id '643a0160-cef7-cc52-4dd8-fc21399daa47'
```

Could I please get some help with this as I am quite lost to what is going wrong.

↑ 1 0 replies

tapegoji Apr 15

Do we have the use Langsmith in this tutorial? seems beyond the point.

↑ 1 👍 1 1 reply

ralphy1976 23 days ago

i found that if you do not use the line: prompt = hub.pull("rlm/rag-prompt")

you won't have a langsmith warning.

You can use your own prompt as shown in the later part of the tutorial.

gopesh97 Apr 18

Earlier there was a conversation retrieval chain, which offered map-reduce llm calls. Is there a tutorial that implements rag with map-reduce and doesnt use langsmith?

↑ 1 0 replies

palladium123 28 days ago

It seems the line 'response = llm.invoke(messages)' do not return the correct class.

When I run this block of code (also in the tutorial), I get the correct type for example_messages:

from langchain import hub

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

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

assert len(example_messages) == 1
print(example_messages[0].content) # returns: <class 'langchain_core.messages.human.HumanMessage'>

But when I run this:

def generate(**state**:State):
docs_content = "\n\n".join(doc.page_content for doc in **state**["context"])
messages = prompt.invoke({"question": **state**["question"], "context": docs_content}).to_messages()
response = llm.invoke(messages)
print(type(response)) # This returns <class 'str>
return {'answer': response.content}

This fits with the error trace I got:

'str' object has no attribute 'content'
File "D:\python\phi4 dictate\RAG_test.py", line 73, in generate
return {"answer": response.content}
^^^^^^^^^^^^^^^^
File "D:\python\phi4 dictate\RAG_test.py", line 80, in
response = graph.invoke({"question": "What is Task Decomposition?"})
^^
AttributeError: 'str' object has no attribute 'content'

Some help would be appreciated.

↑ 1 2 replies

ralphy1976 23 days ago

I had the same problem.

return {"answer": response.content} is not the write syntax.

return {"answer": response} is the write syntax.

to solve it i looked for a similar line of code. luckily there was one 3 lines above: return {"context": retrieved_docs}

after that it worked without any problems.

hope this helps.

ralphy1976 23 days ago

right syntax, not write syntax....

ralphy1976 23 days ago

If you are interested in being able to type your question directly in the terminal, you can do so by using the following:

Before the chatbot template, i wrote: question = input("ask your question: ")

Then, where i would normally type my question in the code, i wrote: response = graph.invoke({"question": question})

so when i run my code, i can ask whatever question i want.

↑ 1 0 replies

clickyman1142 16 days ago

How can the model decide the "section" in the analyze_query function? Does it look into the documents