**2. LangChain: Your Swiss Army Knife**

Aarushi Kansal[**1**](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Aff2)

(1)

Melbourne, Australia

In Chapter [1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_1_Chapter.xhtml), you were introduced to the various types of generative models available, the most popular architectures, and how they work. Now, this chapter introduces you to LangChain, your Swiss Army knife to building robust applications on top of LLMs and other models. As you build applications beyond just making API calls, you’re going to need various components to connect a model to your own data, to external data, and services, and that’s what LangChain helps you with. A standard, modular way for you to essentially plug and play with models and various integrations.

As you go through this chapter, you’ll be introduced to a few concepts you might not be intimately familiar with – don’t worry, as I go along, I’ll explain these concepts, and as you go through further chapters, you’ll use these concepts in increasingly complex ways – which will help you further understand. Basically, this is the approach:

* Introduce the concept and theory.
* Learn by getting your hands dirty.

**The Whats and Whys**

LangChain is not only one of my favorite frameworks for building AI-powered applications but also quickly becoming an industry standard. This framework provides engineers with a modularized, standard interface to plug different models (open and closed source), with various data sources and API integrations.

You can think of it like playing with an assorted bunch of building blocks to create almost anything you can imagine.

The main components LangChain is composed of are the following:

* LLMs
* Retrieval
* Memory
* Chains
* Tools
* Agents

By combining these concepts, you can create end-to-end LLM-powered apps that go beyond just a simple API call to OpenAI. You can chain calls, can allow your model to have access to various tools (e.g., Google search APIs), and finally, can use your LLM’s reasoning abilities to decide which tools to use for particular tasks (this concept is agents).

In the next two chapters, you’ll get to dive into each block or component. As I mentioned earlier, you’re going to learn by doing, so the next few sections are broken down by use cases.

In the first use case, you’re going to build an app to chat to your company or organization’s Slack – you know, for when you have to look up certain information or messages? Instead of keyword searches and then scrolling through messages – why not chat to your archive? This use case will cover LLMs, retrieval, and memory.

In the next use case, you’re going to build an agent that plans your day for you based on your mood, the weather, and your past preferences. This will cover chains, tools, and agents.

Let’s move on to the first one, a chatbot.

**Chatbot**

From Chapter [1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_1_Chapter.xhtml), you understand transformer models, and they’re essentially predicting the next word/s. That’s great for plenty of tasks like translation and text generation, but the thing that’s really useful is the ability of an LLM to hold a conversation with you. That’s the great thing about ChatGPT; it’s been trained on *so* much data, and then on top of that, they’ve built a chatbot, so it’s kind of like being able to chat with **everything**. You’re going to learn how to build your own, smaller version for your personal data like notes, text messages, or Slack messages – without training or even fine-tuning.

Think about the ingredients in a human conversation for a second:

* All participants need to be able to speak the same language.
* Let’s assume English for now, and we have that from most LLMs.
* The participants have to be able to remember what’s happened in the past during this conversation.
* And access to knowledge in some way (in our case, knowledge of your Slack messages).

The last two points are referring to two concepts, which form the basis of *a lot* of LLM-powered applications you’ll build:

* Memory
* Retrieval

Let’s discuss both and then you’ll start to build your chatbot.

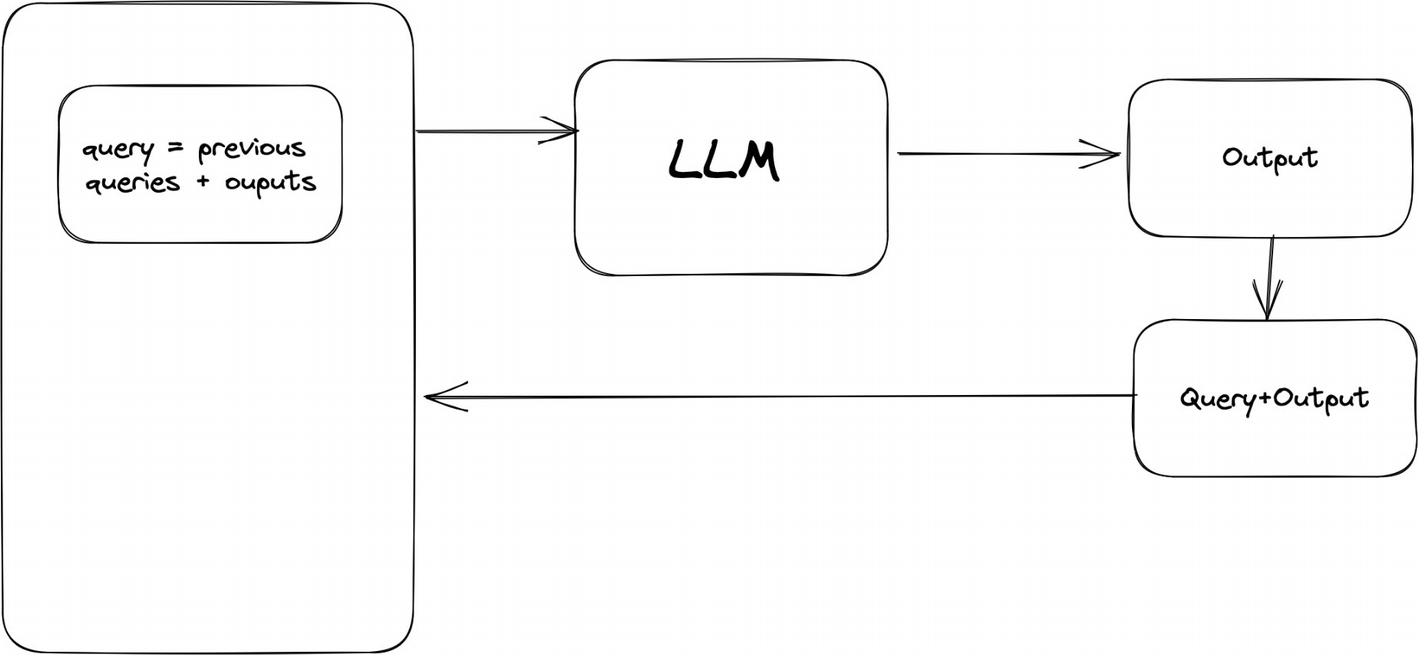
**Memory**

With LLMs, by default, they have no concept of history or memory. Every query or call to an LLM is stateless, meaning they answer every question as if it’s the first time it’s been asked. And the model doesn’t take into account your past interactions.

And that’s the role this concept of memory comes into play – aptly named, it’s a way to give an LLM remembering capabilities so you can hold a conversation with the model.

At this point, maybe you’ve already started thinking about how to start giving any LLM a memory.

One way would be to simply capture each query + LLM response and send that back into your LLM on the next query. As you can see in Figure [2-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Fig1) – you make a call to the LLM, it responds, and you parse the response, format it, and then send the response + your next question back as part of the context. You would keep repeating this pattern (until you run out of context length).



***Figure 2-1***

Example of a simple way of just chaining queries and outputs for “memory”

LangChain actually provides different chains to replicate this behavior in a reusable way – so you just have to decide on the type of memory you want to use, your input prompts, and some tunable parameters. Let’s take a look with some real code.

**Types of Memory**

First, if you haven’t already, go ahead and install LangChain (you can find installation instructions on the LangChain documentation page). For these examples, I’m going to use OpenAI, but you can use any other LLM – LangChain has integrations with the vast majority out there.

LangChain provides a number of types of memories for you to leverage; we’ll focus on the four basic ones in this chapter and some of the more complex ones in later chapters.

First up is ConversationBufferMemory, which serves as a flexible memory buffer for chat conversations. It allows you to access the chat history in two formats:

1. 1.

As a string (buffer\_as\_str)

1. 2.

As a list of message objects (buffer\_as\_messages)

The class provides a load\_memory\_variables method that returns the chat history based on a chosen format. This output can then be used as context to your LLM, thereby providing it info on the previous parts of the conversation.

Let’s take a look at a small example of how memory is represented:

from langchain.memory import ConversationBufferMemory

from langchain.llms import OpenAI

llm = OpenAI()

memory = ConversationBufferMemory()

memory.save\_context({"input": "What is the capital of the UK"}, {"output": "London"})

print(memory.load\_memory\_variables({}))

**Output:**

{'history': 'Human: What is the capital of the UK\nAI: London'}

Here what you’re doing is adding some history into your memory buffer. Right now, nothing is being passed into an LLM, but you can see the output of what would be passed to the LLM.

The prompt would include the “history,” the “Human,” and “AI” conversation – thereby giving the LLM context into the conversation.

In other words, ConversationBufferMemory is a simple way of representing historical context as a string that can be parsed and passed into a prompt.

Notice that the ConversationBufferMemory automatically formatted your input and output into the format of Human and AI conversation. This is the default, but you can change it using these variables:

human\_prefix and ai\_prefix.

For example:

memory = ConversationBufferMemory(human\_prefix="Aarushi", ai\_prefix="Hermione")

memory.save\_context({"input": "What is the capital of the UK"}, {"output": "London"})

**Output:**

**{'history': 'Aarushi: What is the capital of the UK\nHermione: London'}**

How might this look with an LLM attached?

1. 1)

Format the output so the LLM understands what this whole “history” thing is.

1. 2)

Pass that as a prompt + your next query.

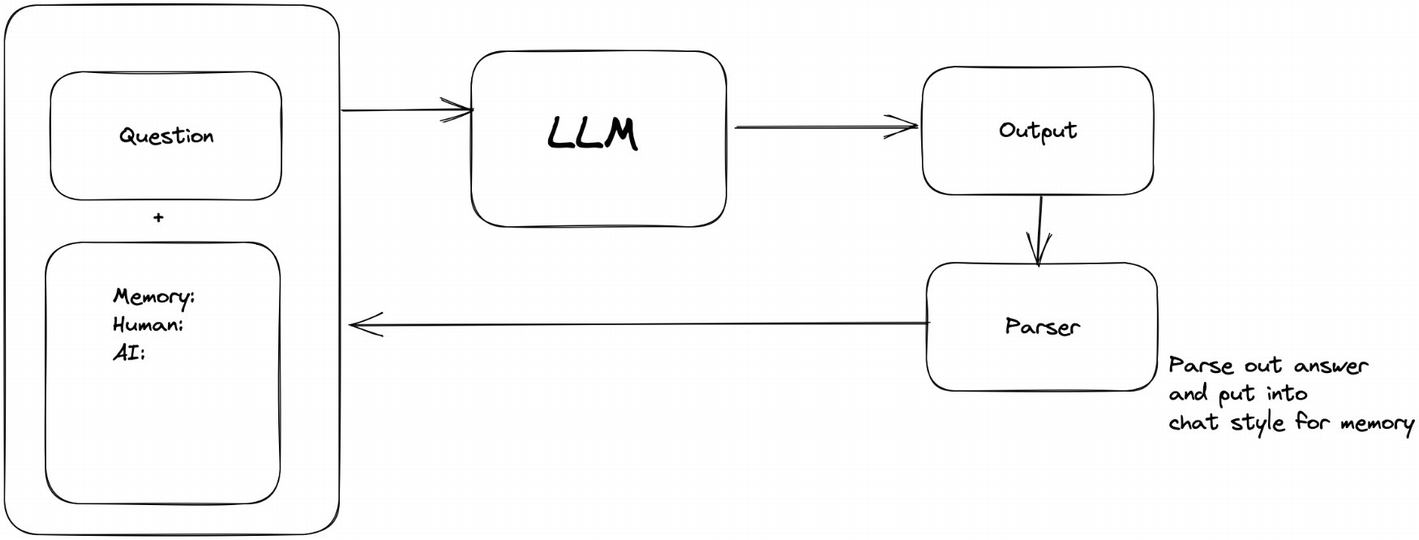
1. 3)

Parse the response into your “history.”

1. 4)

Rinse and repeat.

This process is shown in Figure [2-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Fig2)



***Figure 2-2***

Overall architecture of using memory in the prompt of an LLM

Luckily, LangChain also has a built-in chain for just that. I’ll go into chains later on, but let’s take a quick look at how that would work.

from langchain.llms import OpenAI

from langchain.chains import ConversationChain

llm = OpenAI(temperature=0)

conversation = ConversationChain(

   llm=llm,

   verbose=True,

   memory=ConversationBufferMemory(human\_prefix="Aarushi", ai\_prefix="Hermione")

)

conversation.predict(input="What is the largest city in the UK by population?")

conversation.predict(input="And second?")

conversation.predict(input="What about in Germany?")

# just to run one more time

conversation.predict(input="")

Okay, so similar to before, you initialize your ConversationBufferMemory, with the prefix you want (omit for defaults). Then all you do is add your questions – take note here, I’ve specifically kept the second and third questions brief with minimal context so you can see how it gleans context from the conversation memory.

When you run this, you should get a final output similar to the following:

The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:

Aarushi: What is the largest city in the UK by population?

Hermione: The largest city in the UK by population is London, with a population of 8.9 million people.

Aarushi: And second?

Hermione: The second largest city in the UK by population is Birmingham, with a population of 1.1 million people.

Aarushi: What about in Germany?

Hermione: The largest city in Germany by population is Berlin, with a population of 3.7 million people. The second largest city in Germany by population is Hamburg, with a population of 1.8 million people.

Basically, this chain abstracts away all the logic of parsing and updating the prompt from you so you just choose the memory type and related configuration.

As you use LLMs more and more, you’ll start to notice two things:

1. 1)

Generally, LLMs have a maximum context length – meaning you can only really send a prompt of a certain size.

1. 2)

The larger your history or prompt, LLMs tend to start ignoring or missing older pieces of information.

So really, you want to send less of your history or maybe a condensed version of it – and once again, LangChain allows you to do just that with more types of memory, specifically ConversationBufferWindowMemory, ConversationSummaryMemory, ConversationSummaryBufferMemory, and ConversationTokenBufferMemory.

ConversationBufferWindowMemory – It is a variant of ConversationBuffer; it also keeps a history of your interactions, but only up to *k* number. This is a number you can decide on for your own needs – in my experience, I have found that for most use cases, a larger number is actually detrimental, and the model ends up hallucinating more often than not. I would recommend you experiment and find a balance of short but informational queries combined with a smaller window (k).

ConversationSummaryMemory – As the name suggests, this is a type of memory that condenses down your conversation into a summary that can be passed into your LLM.

Let’s take a quick look at some code:

conversation\_with\_summary = ConversationChain(

   llm=llm,

   memory=ConversationSummaryMemory(llm=llm, human\_prefix="Aarushi", ai\_prefix="Hermione"),

   verbose=True

)

conversation\_with\_summary.predict(input="How are you Hermione?")

conversation\_with\_summary.predict(input="What is the third planet from the sun?")

conversation\_with\_summary.predict(input="second?")

conversation\_with\_summary.predict(input="fifth?")

conversation\_with\_summary.predict(input="")

**Output:**

**The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.**

**Current conversation:**

**Aarushi asked how Hermione was doing and Hermione replied that she was doing great and asked how Aarushi was doing. Aarushi then asked what the third and fifth planets from the sun were, to which Hermione replied that the third was Earth, the only planet known to have life and the only planet in our solar system with liquid water on its surface, and the fifth was Jupiter, the largest planet in our solar system made up mostly of hydrogen and helium with a strong magnetic field and home to the Great Red Spot.**

Here you’ll notice instead of the conversation, with each participant, it’s a summary of it – this kind of memory is great for particularly long conversations, especially where you need the LLM to understand the overall gist rather than each individual nuance of the conversation. For example, if you were chatting over internal HR documents, looking for general time-off policies – the LLM just needs the summary and the large points, not each bit of detailed policy.

Next up is ConversationSummaryBufferMemory, which combines both summaries and a buffer – meaning instead of only keeping a summary of previous interactions, it keeps interactions in a buffer as well as a summary. It means it keeps more recent interactions in a buffer and older ones as a summary (once the buffer hits a certain token length that you can tune).

Finally, ConversationTokenBufferMemory is similar to ConversationBufferWindowMemory but instead maintains a buffer of x tokens length rather than x number of interactions length.

So now we’ve covered some basic types of memory (there’s more, but I want to save those for later chapters). At this point, with even just the simple code snippets shown previously, you have yourself a chatbot that remembers previous interactions and can hold a conversation with you based just on the knowledge it’s been trained on, which is a lot, but it’s not going to be your personal or company data (unless it’s public). And we want to build a chatbot on your *own* information.

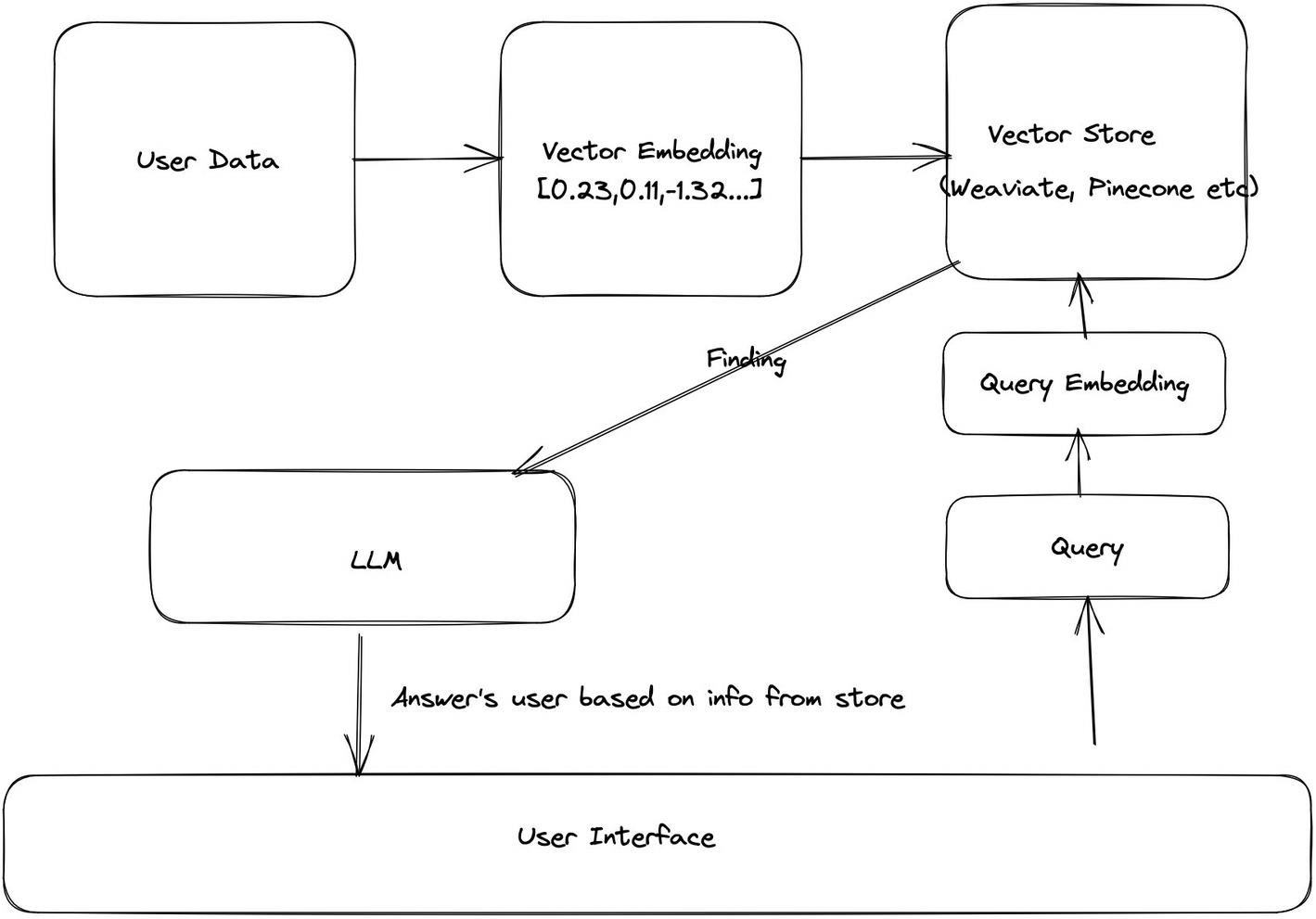
To do this, you could fine-tune your own model, and there are definitely use cases and reasons to fine-tune a model. But in our use case, that can be expensive and time consuming, and most importantly your own info such as Slack messages is going to change way too fast for you to be able to fine-tune fast enough. Think about the speed at which we message each other on Slack or any other messaging app. Luckily, there’s been a rise of a new standard practice, called Retrieval Augmented Generation (RAG), to help give LLMs more knowledge without fine-tuning.

**Retrieval**

Retrieval is just one way of giving an LLM more specific, niche knowledge. It involves fetching data from some external source and passing it into your chosen LLM. Retrieval can be done any way – such as making an API call, reading a static file, reading a SQL DB, etc. This end-to-end flow of fetching info, passing it into an LLM, and the LLM generating a response is known in the industry as Retrieval Augmented Generation (RAG).

**Diving into RAG**

Since we’re generally dealing with natural language and often unstructured messy data, the most popular (for good reason) storage system is a vector store. This essentially involves taking all of your niche data, creating a vector embedding, and storing in a vector database of your choice, as shown in Figure [2-3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Fig3).



***Figure 2-3***

Retrieval Augmented Generation with a vector store

Let’s walk through each component in detail, starting with embeddings.

**Embeddings Explained**

In layman’s terms, embeddings are a way to represent everything that’s not numbers (e.g., text, audio, images, etc.) as numbers. All ML models are basically math equations in some form, complex math but still just math – and so they don’t actually understand or perceive words or images or anything else the way humans can with our five senses; they only understand numbers. That’s why to deal with language, we need to convert words and sentences into a numerical representation that ML models can understand. The term “embeddings” is a general term for taking one type of data and representing it in numbers. Embeddings come in different types, such as graph embeddings, tensor embeddings, and many more.

In the context of LLMs, we’re talking about vector embeddings, and you’ll see vector embeddings and embeddings used interchangeably.

**Vector Embeddings**

As the name suggests, vector embeddings are a specific type of embedding where the representation is in the form of a vector. This means that the data, regardless of its original form, is translated into a fixed-length list of numbers.

Vector embeddings are commonly used in natural language processing (like Word2Vec or GloVe), where words or phrases are represented as vectors.

One of the major benefits of vectors is the ability to have vectors in high-dimensional spaces, which means that a vast number of features or aspects of language can be captured. Each dimension can potentially represent some facet of meaning, allowing the vector to encapsulate a rich set of semantic information.

And because vectors are basically long lists of numbers, we can do mathematical computations on data that normally wouldn’t be possible. For example:

* vector(Germany) - vector(Berlin) + vector(France) = vector(Paris)

This shows that the difference between a country and its capital can be consistently represented in the vector space. So by knowing the capital of Germany and applying this relationship to France, we can deduce the capital of France.

Because we can do computations like this, another benefit of vector embeddings now is that we can use these geometric relationships between vectors to model semantic or functional relationships. For instance, in word embeddings, the vector difference between “dolphin” and “ocean” might be similar to the difference between “camel” and “desert,” reflecting habitat relationships.

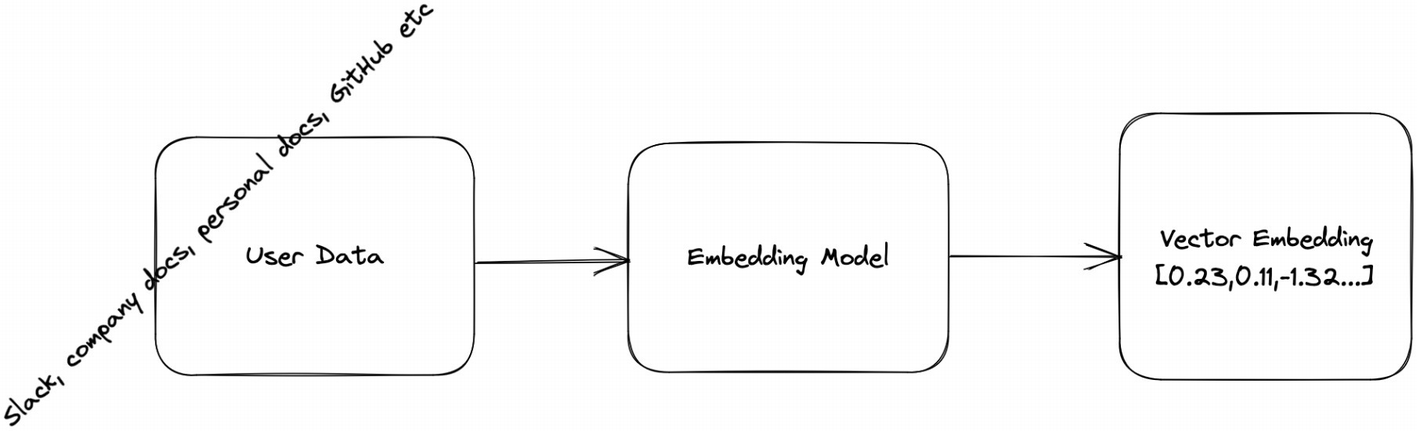
Basically, we use vector embeddings as a way to model intricate and complex semantic meaning and relationships between words and text.

Now, modelling these semantic relationships isn’t a trivial task – you need an embedding model that understands these complex relationships and can take your raw words or sentences and create vector embeddings. Luckily there are quite a few embedding models available:

* OpenAI’s embedding model, for example, text-embedding-ada-002
* Cohere’s embedding model
* Open source embedding models (<https://huggingface.co/BAAI/bge-large-en>)
* And a lot more

LangChain also integrates with the vast majority, and you can have a look here: <https://python.langchain.com/docs/integrations/text_embedding/>.

So in an RAG application, you’ll have one step that involves taking your raw data, inputting it into an embedding model, and then getting a vector embedding out of it, as shown in Figure [2-4](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Fig4).



***Figure 2-4***

Steps for taking unstructured, raw data and converting to an embedding

Now that you have these embeddings – you need somewhere to store them, somewhere to search from so your LLM has an external data source. And that’s where vector stores come into play.

**Vector Stores Explained**

Vector stores or vector DBs have been around long before generative AI became so mainstream – they were used in areas such as information retrieval, recommendation systems, and even molecular biology.

Now with modern embedding models and the rise of LLMs, there’s also been a rise not just in popularity of vector stores but also new, more modern DBs available specifically designed to fit in with modern generative AI models.

Here’s a non-exhaustive list for you:

* Weaviate
* Pinecone
* Chroma
* Qdrant
* Traditional DBs that have started supporting vector embeddings

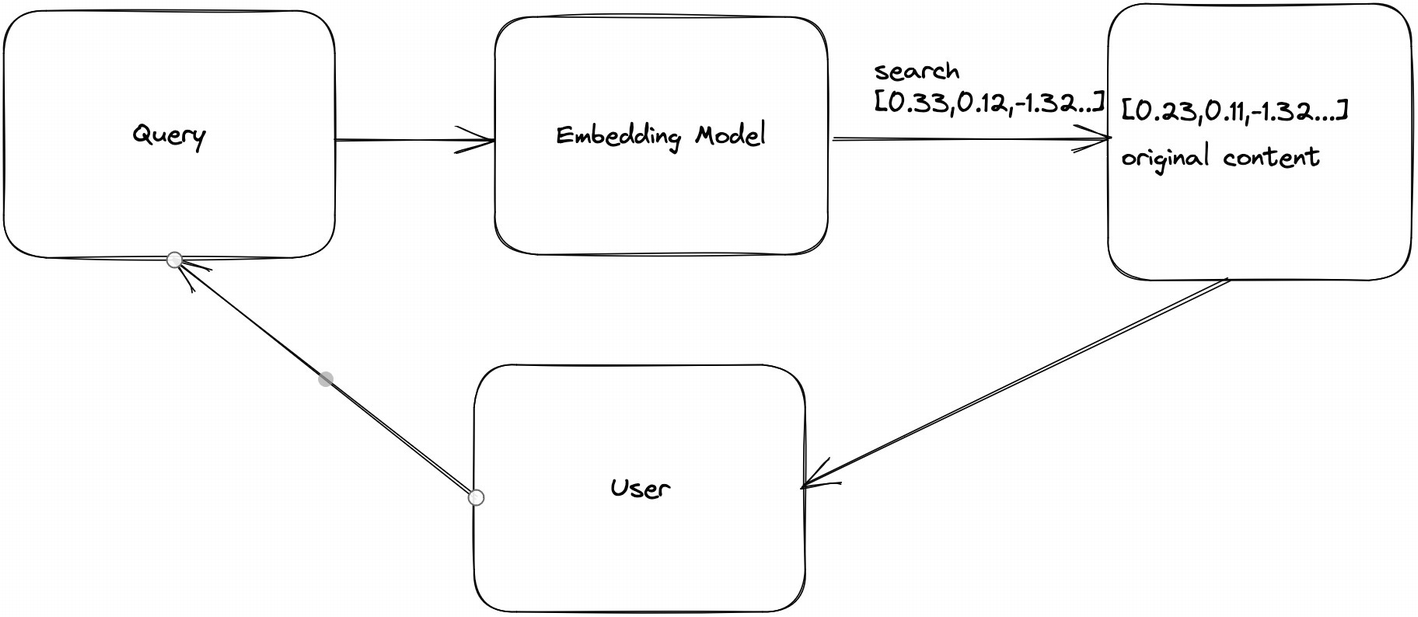
If you want a more detailed list, check out <https://python.langchain.com/docs/integrations/vectorstores>.

So how do these DBs actually work and what’s so special about them compared to existing SQL and NoSQL DBs?

Vector DBs are specifically designed to store high dimensional data like embeddings and allow for fast querying and lookups. They have the capabilities of traditional databases, while being optimized to handle the complexity of vector embeddings.

When you create + store an embedding, a reference to the original data is also stored. Then, when you make a query to the DB, the query is first converted to an embedding (using the same model), and this embedding is used to find the most similar content and return it to you.

Steps are shown in Figure [2-5](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml#Fig5).



***Figure 2-5***

Steps for converting a user query into a searchable query in a vector DB and returning the answer

Unlike traditional databases that are optimized to search for exact values – a number, a string, or some other single dimensional, exact value – vector databases are optimized to search for vectors (high dimensional data) that are most *similar*, not exact to another query vector.

To do this, vector databases store your data in structures that allow for fast querying – called indexes. These indexes are created using a variety of algorithms, which we won’t go into detail but are listed if you want to read more about them:

* Random Projection
* Product Quantization
* Locality-Sensitive Hashing
* Hierarchical Navigable Small World

When the query comes in, these DBs make use of various algorithms to do an Approximate Nearest Neighbor (ANN) search to get the most similar matches based on some similarity metrics, such as cosine similarity, dot product, Euclidean distance, or Hamming distance.

And there you have it, theory on how you get an LLM hooked up to your own data.

**What Else Is RAG Good For?**

Hallucination – that’s probably a word you’ve heard *a lot* when discussing or critiquing LLMs or LLM systems. Hallucination basically refers to the LLM’s tendency to just “make up” information. This can happen for reasons such as not having the information and being trained on incorrect information (remember, these systems are trained on public data, and there’s a lot of misinformation out there).

By using RAG, you can help reduce this tendency by having your system fetch the correct information and including it in the context for when your LLM crafts a response.

Let’s move on to some actual code.

**The App**

Okay, so now we’re going to put all of these pieces together and build a chatbot over your Slack messages (or any other data source really).

**Prerequisites**

* Python 3
* Latest LangChain version
* A vector database
* I’m going to use Weaviate’s 14-day free hosted one, but you can choose one of your choice. The code generally remains the same.
* Slack messages
* I’ve created a new workspace with some “fake” messages – you can do the same or if you are able to use real ones, use those.
* An LLM
* I’m using OpenAI’s GPT-4.

**Loading Your Data**

Since we’re dealing with unstructured data here, the first thing you’re going to need to do is load your Slack messages in a form that can then be turned into vector embeddings.

Go ahead and export your Slack messages as a zip file from {your\_slack\_domain}.slack.com/services/export.

Once you have that, let’s load it as seen in the following code snippet:

LOCAL\_ZIPFILE = "gen\_ai\_co\_slack.zip"  # Paste the local paty to your Slack zip file here.

loader = SlackDirectoryLoader(LOCAL\_ZIPFILE)

docs = loader.load()

Basically, you’ll most likely be dealing with unstructured data, so LangChain provides you with a lot of different types of “loaders” that take data of one type (e.g., Slack, epub, logs from Datadog, Excel, GitHub, and many more) and turn them into structured data. For example, the SlackDirectoryLoader takes the json files exported from Slack and converts them into a list of documents. This Document structure just stores text and its associated metadata.

For example, the json files end up looking something like this:

Document(page\_content='<@U05SQ9E71EF> has joined the channel', metadata={'source': 'q4-planning - U05SQ9E71EF - 1694993629.080429', 'channel': 'q4-planning', 'timestamp': '1694993629.080429', 'user': 'U05SQ9E71EF'})

where page\_content is the messages and the other fields are associated metadata. Other loaders work in different ways – but the end result is always the same, unstructured data converted into structured.

**Transforming Your Structured Data**

Now, you have your data loaded, but you still need to transform it before you can create the embeddings – this means transforming your data into smaller chunks before creating an embedding. This is because you want to be able to fit meaningful parts of your data within the context window of your model when querying and adding answers as context. This is where LangChain has document transformers that you’re going to use. The default you’re using here is the RecursiveCharacterTextSplitter, which tries to split on certain characters – by default on \n\n, \, “ ”, and “”.

In this snippet, I’ve chosen a chunk size of 500 and an overlap of 40 – the overlap ensures continuity between chunks.

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=40)

documents = text\_splitter.split\_documents(docs)

You can check out the other document transformers here: <https://python.langchain.com/docs/integrations/document_transformers/>.

Now, choosing a chunk size can almost be a bit of an art form. Remember, the size of your chunk influences your embeddings.

If you go with shorter chunks (think words, or sentences), then your embeddings will lose the wider context in the paragraph – the embedding will narrow down on the specific meaning of the word or sentence.

If you go with longer chunks – you’ll get the broader context, but that could add confusion and actually cause the embedding to lose the more specific or nuanced meanings you might need.

As such, you need to really take into account:

What kind of documents are you dealing with? For example, Slack messages are usually quite short, so you can easily go for a shorter chunk size. Books and scientific articles are a different story – they’re longer and often you need the wider context; in this case, I would consider the next questions.

What is your use case? Will you or your users be asking very specific, nuanced questions? Short or long queries? Vague queries? For example, if your application is more of a very specific Q&A application, I would go through the documents and get a feel of how long do I as a human being need to read to get the right answer and based on that choose my chunk size.

A lot of these questions can be answered through you experimenting with sizes.

**Embeddings and Storage**

Next is creating your actual embeddings out of these chunks and actually storing them somewhere. Again, LangChain provides an abstraction layer to various embedding models and vector stores. This means you can just plug in any one you have access to.

In the next example, I’m going to use OpenAI for embeddings and Weaviate for storage, but since it’s a plug-and-play concept, you can replace it with one of your choices and the overall code doesn’t need to change drastically.

Check out all the embedding integrations here: <https://python.langchain.com/docs/integrations/text_embedding/openai>.

embeddings = OpenAIEmbeddings()

db = Weaviate.from\_documents(documents, embeddings, weaviate\_url=WEAVIATE\_URL, by\_text=False)

And your code is as simple as this; choose the embedding integration and the vector store and pass your chunked docs and embedding model and you get a vector store populated with your embeddings, that you can now query.

For example, in my setup, I can run this query:

query = "What is the work from anywhere policy?"

docs = db.similarity\_search(query)

and get a response like this:

Exciting News! We’re officially launching our \*Work from Anywhere (WFA)\* policy. Starting next month, you’ll have the flexibility to choose your work location, be it from home, a café, or any place that boosts your productivity.

So you can see my query was turned into an embedding; a similarity search took place; the resultant content was returned to me as is.

Okay so, maybe now you’re wondering how to choose an embedding model.

Here are some of my considerations:

* Cost: Hosted ones like OpenAI can be expensive.
* Latency: Hosted ones are quite new currently and often don’t provide SLAs, so expect unexpected latencies.
* Quality: This one’s tricky because it’s unlikely you’re going to be able to test out *all* the models closed and open source out there. I use the Massive Text Embedding Benchmark (MTEB) leaderboard as a good source (<https://huggingface.co/spaces/mteb/leaderboard>).

**Memory**

And finally, now that you can retrieve data, you also want to hook up a memory component.

By now, you know how to instantiate memory, so go ahead and do that. The new step you’re going to do is include a so-called “retriever,” which is just the vector store from which your app can fetch. It’s a VectorStoreRetriever object, which has functions on it to allow it to actually query the store. Instantiate it as shown here:

llm = OpenAI(temperature=0)

memory = ConversationSummaryMemory(llm=llm, memory\_key="chat\_history", return\_messages=True)

ret = db.as\_retriever()

Now, previously we used a ConversationChain, in this case, we’re going to use a different chain that can handle a retriever, called a ConversationalRetrievalChain.

This chain is similar to the chain you’ve used previously, except it includes one extra step internally, when you ask questions. It actually passes your question directly to the vector store and returns the stored documents.

Essentially, it’s an abstraction on this call:

docs = db.similarity\_search(query)

In the following code snippet, you’ll see how to set up your chain – now when you run this, you’ll see the same memory + summarization combination you saw previously.

qa = ConversationalRetrievalChain.from\_llm(llm, retriever=ret, memory=memory)

qa("What is the work from anywhere policy?")

qa("are there any in office days required?")

qa("any coworking?")

And there you have it, a conversational app across your Slack messages.

**What’s Next?**

Okay, so far you’ve built the seedlings of a conversational chatbot across your Slack messages. Next you’re going to learn in depth about chains and agents. This will help take your app to the next step – moving from purely a Python script to something slightly more interactive.

**Summary**

In this chapter, you were introduced to LangChain and you learned how to use the two most basic building blocks: memory and retrieval, which allow you to create Retrieval Augmented Generation applications. This is a great start to giving your LLMs external knowledge, without having to spend time and effort on fine-tuning. On top of this, you can keep updating your vector store with new information, much faster than you could fine-tune with new information.

In the next chapter, you’ll take what you’ve learned about memory and RAG one step further and create an agent using LangChain.