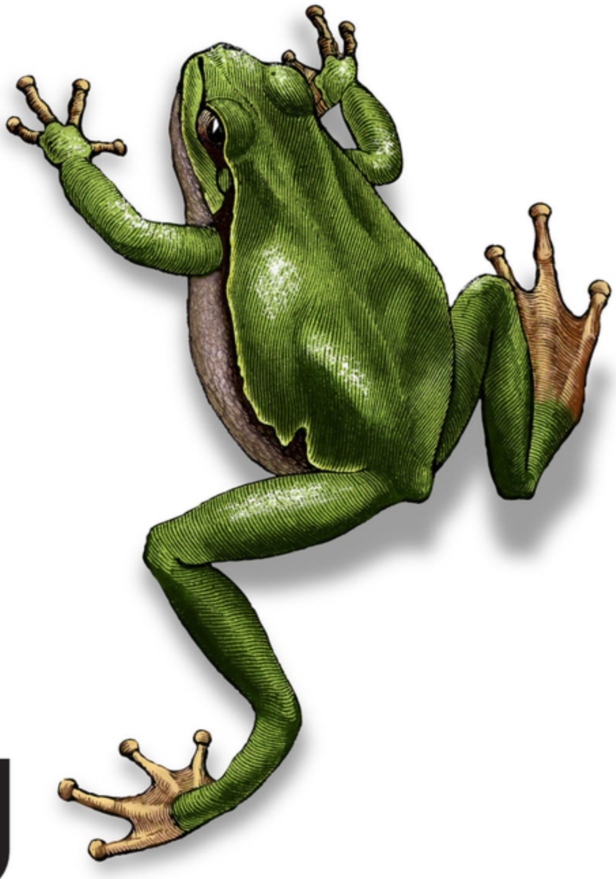
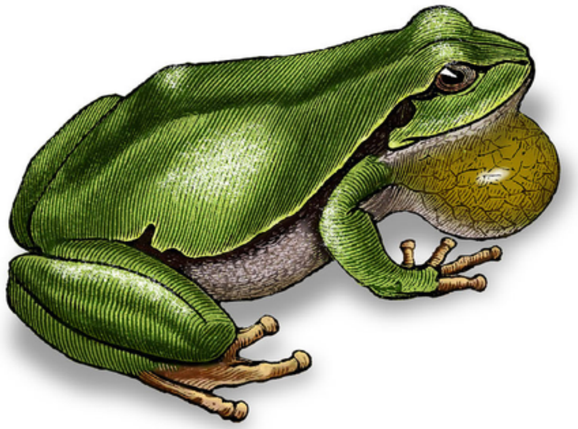
O'REILLY®



Learning LangChain

Building AI and LLM Applications with LangChain and LangGraph



Mayo Oshin &
Nuno Campos

Praise for *Learning LangChain*

With clear explanations and actionable techniques, this is the go-to resource for anyone looking to harness LangChain's power for production-ready generative AI and agents. A must-read for developers aiming to push the boundaries of this platform.

—Tom Taulli, IT consultant and author of *AI-Assisted Programming*

This comprehensive guide on LangChain covers everything from document retrieval and indexing to deploying and monitoring AI agents in production. With engaging examples, intuitive illustrations, and hands-on code, this book made learning LangChain interesting and fun!

—Rajat K. Goel, senior software engineer, IBM

This book is a comprehensive LLM guide covering fundamentals to production, packed with technical insights, practical strategies, and robust AI patterns.

—Gourav Singh Bais, senior data scientist and senior technical content writer, Allianz Services

Prototyping generative AI apps is easy—shipping them is hard. The strategies and tools in Learning LangChain make it possible to turn ideas into modern, production-ready applications.

—James Spiteri, director of product management for security, Elastic

Learning LangChain provides a clear path for transforming how you build AI-powered applications. By breaking down flexible architectures and robust checkpointing, it offers a strong foundation for creating reliable, production-ready AI agents at scale.

—David O'Regan, engineering manager for AI/ML, GitLab

Learning LangChain helped us skip the boilerplate for debugging and monitoring. The many helpful patterns and tooling insights allowed us to move fast and deploy AI apps with confidence.

—Chris Focke, chief AI scientist, AppFolio

Teaching LangChain through clear, actionable examples, this book is a gateway to agentic applications that are as inspiring as Asimov's sci-fi novels.

— Ilya Meyzin, SVP head of data science, Dun & Bradstreet

Learning LangChain

Building AI and LLM Applications with LangChain and LangGraph

Mayo Oshin and Nuno Campos

O'REILLY®

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.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Corbin Collins
- Production Editor: Clare Laylock
- Copyeditor: nSight, Inc.
- Proofreader: Helena Stirling
- Indexer: Judith McConville

- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- February 2025: First Edition

Revision History for the First Edition

- 2024-02-13: First Release

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

978-1-098-16728-8

[LSI]

Preface

On November 30, 2022, San Francisco–based 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. In this book we will be making use of the OpenAI API as the LLM provider we use in the code examples (pricing is listed on its platform). One of the benefits of working with LangChain is that you can follow along with all of these examples using either OpenAI or alternative commercial or open source LLM providers.

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 touch on the concept of *machine learning* (ML). Some computer *algorithms* (imagine a repeatable recipe 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 (such as 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:

- They are trained on much larger amounts of data; training one of these models from scratch would be very costly.
- 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 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 LangChain is all about.

By the end of 2023, competing LLMs emerged, including Anthropic's Claude and Google's Bard (later renamed Gemini), 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 computation or knowledge”. 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 the question into an input that 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, at the time of writing, the largest) library to provide such building blocks and the tooling to reliably combine them into larger applications. Before discussing what it takes to build compelling applications with these new tools, let’s get more familiar with LLMs and LangChain.

Brief Primer on LLMs

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

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

- *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.[1] *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.)
- *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 interpreting the meaning of individual words and sentences, which makes it possible to use them for generating plausible, readable, written text.

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. Researchers have found that LLMs are able to transfer some of their semantic understanding to other languages.[2]

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 spans on average 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*.[3] The transformer architecture enables models to handle sequences of data, such as 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 *pretrained 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. In fact, the only feasible 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, they require you 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*, but not with the more natural *What is the capital of England?*

Instruction-Tuned LLMs

Researchers have made pretrained LLMs easier to use by further training (additional training applied on top of the long and costly training described in the previous section), also known as *fine-tuning* 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 that 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 pretraining datasets, these are manually assembled, so they are by necessity much smaller:

Reinforcement learning from human feedback (RLHF)

Through the use of RLHF 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 earlier question.

Instruction-tuning has been key to broadening the number of people who can build applications with LLMs, as they can now be prompted with *instructions*, often in the form of questions such as, *What is the capital of England?*, as opposed to *The capital of England is*.

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