

Programming Large Language Models with Azure Open AI

Conversational programming and
prompt engineering with LLMs



Francesco Esposito

Professional



FREE SAMPLE CHAPTER |



Programming Large Language Models with Azure Open AI: Conversational programming and prompt engineering with LLMs

Francesco Esposito

Programming Large Language Models with Azure Open AI: Conversational programming and prompt engineering with LLMs

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2024 by Francesco Esposito.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-828037-6

ISBN-10: 0-13-828037-1

Library of Congress Control Number: 2024931423

\$PrintCode

Trademarks

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corp-sales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor-in-Chief

Brett Bartow

Executive Editor

Loretta Yates

Associate Editor

Shourav Bose

Development Editor

Kate Shoup

Managing Editor

Sandra Schroeder

Senior Project Editor

Tracey Croom

Copy Editor

Dan Foster

Indexer

Timothy Wright

Proofreader

Donna E. Mulder

Technical Editor

Dino Esposito

Editorial Assistant

Cindy Teeters

Cover Designer

Twist Creative, Seattle

Compositor

codeMantra

Graphics

codeMantra

Dedication

A I.

Perché non dedicarti un libro sarebbe stato un sacrilegio.

Contents at a Glance

	<i>Introduction</i>	<i>xiii</i>
CHAPTER 1	The genesis and an analysis of large language models	1
CHAPTER 2	Core prompt learning techniques	25
CHAPTER 3	Engineering advanced learning prompts	51
CHAPTER 4	Mastering language frameworks	79
CHAPTER 5	Security, privacy, and accuracy concerns	131
CHAPTER 6	Building a personal assistant	159
CHAPTER 7	Chat with your data	181
CHAPTER 8	Conversational UI	203
	<i>Appendix: Inner functioning of LLMs</i>	<i>217</i>
	<i>Index</i>	<i>234</i>

Contents

<i>Acknowledgments</i>	<i>xi</i>
<i>Introduction</i>	<i>xiii</i>

Chapter 1 The genesis and an analysis of large language models 1

LLMs at a glance	1
History of LLMs	2
Functioning basics	5
Business use cases	14
Facts of conversational programming	14
The emerging power of natural language	15
LLM topology	16
Future perspective	18
Summary	23

Chapter 2 Core prompt learning techniques 25

What is prompt engineering?	25
Prompts at a glance	25
Alternative ways to alter output	28
Setting up for code execution	31
Basic techniques	34
Zero-shot scenarios	35
Few-shot scenarios	37
Chain-of-thought scenarios	41
Fundamental use cases	44
Chatbots	44
Translating	47
LLM limitations	48
Summary	49

Chapter 3	Engineering advanced learning prompts	51
	What's beyond prompt engineering?	51
	Combining pieces	52
	Fine-tuning	54
	Function calling	56
	Homemade-style	57
	OpenAI-style	60
	Talking to (separated) data	64
	Connecting data to LLMs	64
	Embeddings	65
	Vector store	69
	Retrieval augmented generation	72
	Summary	76
Chapter 4	Mastering language frameworks	79
	The need for an orchestrator	79
	Cross-framework concepts	80
	Points to consider	86
	LangChain	88
	Models, prompt templates, and chains	88
	Agents	96
	Data connection	104
	Microsoft Semantic Kernel	109
	Plug-ins	111
	Data and planners	116
	Microsoft Guidance	121
	Configuration	121
	Main features	123
	Summary	129

Chapter 5	Security, privacy, and accuracy concerns	131
	Overview	131
	Responsible AI.....	131
	Red teaming	132
	Abuse and content filtering	133
	Hallucination and performances.....	134
	Bias and fairness.....	135
	Security and privacy	135
	Security.....	135
	Privacy.....	140
	Evaluation and content filtering	144
	Evaluation.....	144
	Content filtering.....	148
	Summary	157
Chapter 6	Building a personal assistant	159
	Overview of the chatbot web application	159
	Scope.....	160
	Tech stack.....	160
	The project.....	161
	Setting up the LLM	161
	Setting up the project.....	163
	Integrating the LLM.....	165
	Possible extensions	178
	Summary	179
Chapter 7	Chat with your data	181
	Overview	181
	Scope.....	181
	Tech stack.....	182
	What is Streamlit?	182
	A brief introduction to Streamlit.....	183

Main UI features	183
Pros and cons in production	185
The project	186
Setting up the project and base UI	187
Data preparation	189
LLM integration	193
Progressing further	198
Retrieval augmented generation versus fine-tuning	198
Possible extensions	200
Summary	201
Chapter 8 Conversational UI	203
Overview	203
Scope	204
Tech stack	205
The project	206
Minimal API setup	206
OpenAPI	208
LLM integration	210
Possible extensions	215
Summary	216
<i>Appendix: Inner functioning of LLMs</i>	217
<i>Index</i>	234

Acknowledgments

In the spring of 2023, when I told my dad how cool Azure OpenAI was becoming, his reply was kind of a shock: “Why don’t you write a book about it?” He said it so naturally that it hit me as if he really thought I could do it. In fact, he added, “Are you up for it?” Then there was no need to say more. Loretta Yates at Microsoft Press enthusiastically accepted my proposal, and the story of this book began in June 2023.

AI has been a hot topic for the better part of a decade, but the emergence of new-generation large language models (LLMs) has propelled it into the mainstream. The increasing number of people using them translates to more ideas, more opportunities, and new developments. And this makes all the difference.

Hence, the book you hold in your hands can’t be the ultimate and definitive guide to AI and LLMs because the speed at which AI and LLMs evolve is impressive and because—by design—every book is an act of approximation, a snapshot of knowledge taken at a specific moment in time. Approximation inevitably leads to some form of dissatisfaction, and dissatisfaction leads us to take on new challenges. In this regard, I wish for myself decades of dissatisfaction. And a few more years of being on the stage presenting books written for a prestigious publisher—it does wonders for my ego.

First, I feel somewhat indebted to all my first dates since May because they had to endure monologues lasting at least 30 minutes on LLMs and some weird new approach to transformers.

True thanks are a private matter, but publicly I want to thank Martina first, who cowrote the appendix with me and always knows what to say to make me better. My gratitude to her is keeping a promise she knows. Thank you, Martina, for being an extraordinary human being.

To Gianfranco, who taught me the importance of discussing and expressing, even loudly, when something doesn’t please us, and taught me to always ask, because the worst thing that can happen is hearing a no. Every time I engage in a discussion, I will think of you.

I also want to thank Matteo, Luciano, Gabriele, Filippo, Daniele, Riccardo, Marco, Jacopo, Simone, Francesco, and Alessia, who worked with me and supported me during my (hopefully not too frequent) crises. I also have warm thoughts for Alessandro, Antonino, Sara, Andrea, and Cristian who tolerated me whenever we weren’t like 25-year-old youngsters because I had to study and work on this book.

To Mom and Michela, who put up with me before the book and probably will continue after. To my grandmas. To Giorgio, Gaetano, Vito, and Roberto for helping me to grow every day. To Elio, who taught me how to dress and see myself in more colors.

As for my dad, Dino, he never stops teaching me new things—for example, how to get paid for doing things you would just love to do, like being the technical editor of this book. Thank you, both as a father and as an editor. You bring to my mind a song you well know: “Figlio, figlio, figlio.”

Beyond Loretta, if this book came to life, it was also because of the hard work of Shourav, Kate, and Dan. Thank you for your patience and for trusting me so much.

This book is my best until the next one!

Introduction

This is my third book on artificial intelligence (AI), and the first I wrote on my own, without the collaboration of a coauthor. The sequence in which my three books have been published reflects my own learning path, motivated by a genuine thirst to understand AI for far more than mere business considerations. The first book, published in 2020, introduced the mathematical concepts behind machine learning (ML) that make it possible to classify data and make timely predictions. The second book, which focused on the Microsoft ML.NET framework, was about concrete applications—in other words, how to make fancy algorithms work effectively on amounts of data hiding their complexity behind the charts and tables of a familiar web front end.

Then came ChatGPT.

The technology behind astonishing applications like ChatGPT is called a *large language model (LLM)*, and LLMs are the subject of this third book. LLMs add a crucial capability to AI: the ability to generate content in addition to classifying and predicting. LLMs represent a paradigm shift, raising the bar of communication between humans and computers and opening the floodgates to new applications that for decades we could only dream of.

And for decades, we did dream of these applications. Literature and movies presented various supercomputers capable of crunching any sort of data to produce human-intelligible results. An extremely popular example was HAL 9000—the computer that governed the spaceship *Discovery* in the movie *2001: A Space Odyssey* (1968). Another famous one was JARVIS (Just A Rather Very Intelligent System), the computer that served Tony Stark’s home assistant in *Iron Man* and other movies in the Marvel Comics universe.

Often, all that the human characters in such books and movies do is simply “load data into the machine,” whether in the form of paper documents, digital files, or media content. Next, the machine autonomously figures out the content, learns from it, and communicates back to humans using natural language. But of course, those supercomputers were conceived by authors; they were only science fiction. Today, with LLMs, it is possible to devise and build concrete applications that not only make human–computer interaction smooth and natural, but also turn the old dream of simply “loading data into the machine” into a dazzling reality.

This book shows you how to build software applications using the same type of engine that fuels ChatGPT to autonomously communicate with users and orchestrate business tasks driven by plain textual prompts. No more, no less—and as easy and striking as it sounds!

Who should read this book

Software architects, lead developers, and individuals with a background in programming—particularly those familiar with languages like Python and possibly C# (for ASP.NET Core)—will find the content in this book accessible and valuable. In the vast realm of software professionals who might find the book useful, I'd call out those who have an interest in ML, especially in the context of LLMs. I'd also list cloud and IT professionals with an interest in using cloud services (specifically Microsoft Azure) or in sophisticated, real-world applications of human-like language in software. While this book focuses primarily on the services available on the Microsoft Azure platform, the concepts covered are easily applicable to analogous platforms. At the end of the day, using an LLM involves little more than calling a bunch of API endpoints, and, by design, APIs are completely independent of the underlying platform.

In summary, this book caters to a diverse audience, including programmers, ML enthusiasts, cloud-computing professionals, and those interested in natural language processing, with a specific emphasis on leveraging Azure services to program LLMs.

Assumptions

To fully grasp the value of a programming book on LLMs, there are a couple of prerequisites, including proficiency in foundational programming concepts and a familiarity with ML fundamentals. Beyond these, a working knowledge of relevant programming languages and frameworks, such as Python and possibly ASP.NET Core, is helpful, as is an appreciation for the significance of classic natural language processing in the context of business domains. Overall, a blend of programming expertise, ML awareness, and linguistic understanding is recommended for a comprehensive grasp of the book's content.

This book might not be for you if...

This book might not be for you if you're just seeking a reference book to find out in detail how to use a particular pattern or framework. Although the book discusses advanced aspects of popular frameworks (for example, LangChain and Semantic Kernel) and APIs (such as OpenAI and Azure OpenAI), it does not qualify as a programming reference on any of these. The focus of the book is on using LLMs to build useful applications in the business domains where LLMs really fit well.

Organization of this book

This book explores the practical application of existing LLMs in developing versatile business domain applications. In essence, an LLM is an ML model trained on extensive text data, enabling it to comprehend and generate human-like language. To convey knowledge about these models, this book focuses on three key aspects:

- The first three chapters delve into scenarios for which an LLM is effective and introduce essential tools for crafting sophisticated solutions. These chapters provide insights into conversational programming and prompting as a new, advanced, yet structured, approach to coding.
- The next two chapters emphasize patterns, frameworks, and techniques for unlocking the potential of conversational programming. This involves using natural language in code to define workflows, with the LLM-based application orchestrating existing APIs.
- The final three chapters present concrete, end-to-end demo examples featuring Python and ASP.NET Core. These demos showcase progressively advanced interactions between logic, data, and existing business processes. In the first demo, you learn how to take text from an email and craft a fitting draft for a reply. In the second demo, you apply a retrieval augmented generation (RAG) pattern to formulate responses to questions based on document content. Finally, in the third demo, you learn how to build a hotel booking application with a chatbot that uses a conversational interface to ascertain the user's needs (dates, room preferences, budget) and seamlessly places (or denies) reservations according to the underlying system's state, without using fixed user interface elements or formatted data input controls.

Downloads: notebooks and samples

Python and Polyglot notebooks containing the code featured in the initial part of the book, as well as the complete codebases for the examples tackled in the latter part of the book, can be accessed on GitHub at:

<https://github.com/Youbiquitous/programming-llm>

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

MicrosoftPressStore.com/LLMAzureAI/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *<http://support.microsoft.com>*.

Stay in touch

Let's keep the conversation going! We're on X / Twitter: *<http://twitter.com/MicrosoftPress>*.

Core prompt learning techniques

Prompt learning techniques play a crucial role in so-called “conversational programming,” the new paradigm of AI and software development that is now taking off. These techniques involve the strategic design of prompts, which are then used to draw out desired responses from large language models (LLMs).

Prompt engineering is the creative sum of all these techniques. It provides developers with the tools to guide, customize, and optimize the behavior of language models in conversational programming scenarios. Resulting prompts are in fact instrumental in guiding and tailoring responses to business needs, improving language understanding, and managing context.

Prompts are not magic, though. Quite the reverse. Getting them down is more a matter of trial and error than pure wizardry. Hence, at some point, you may end up with prompts that only partially address the very specific domain requests. This is where the need for fine-tuning emerges.

What is prompt engineering?

As a developer, you use prompts as instructional input for the LLM. Prompts convey your intent and guide the model toward generating appropriate and contextually relevant responses that fulfill specific business needs. Prompts act as cues that inform the model about the desired outcome, the context in which it should operate, and the type of response expected. More technically, the prompt is the point from which the LLM begins to predict and then output new tokens.

Prompts at a glance

Let’s try some prompts with a particular LLM—specifically, GPT-3.5-turbo. Be aware, though, that LLMs are not deterministic tools, meaning that the response they give for the same input may be different every time.



Note Although LLMs are commonly described as non-deterministic, “seed” mode is now becoming more popular—in other words, seeding the model instead of sampling for a fully reproducible output.

A very basic prompt

The hello-world of prompt engineering—easily testable online on Bing Chat, ChatGPT, or something similar—can be as simple as what’s shown here:

During the week I

This prompt might result in something like the following output:

During the week, I typically follow a structured routine.

Overall, the answer makes sense: The model tries to provide a continuation of the string, given the understood context.

Let’s try something a bit more specific:

Complete the following sentence, as if you were Shakespeare.
During the week I

The subsequent output might be similar to:

During the week, I doth engage in myriad tasks and endeavors, as the sun traverseth the sky with unwavering pace.

So far so good.

A more complex prompt

One relatively complex prompt might be the following:

'Unacceptable risk AI systems are systems considered a threat to people and will be banned. They include:

-Cognitive behavioral manipulation of people or specific vulnerable groups: for example, voice-activated toys that encourage dangerous behavior in children

-Social scoring: classifying people based on behavior, socio-economic status, or personal characteristics

-Real-time and remote biometric identification systems, such as facial recognition

Some exceptions may be allowed: For instance, "post" remote biometric identification systems where identification occurs after a significant delay will be allowed to prosecute serious crimes but only after court approval.'

Given the above, extract only the forbidden AI applications and output them as json.

The model might now output the following JSON string:

```
{
  "Forbidden AI Applications": [
    {
      "Application": "Cognitive behavioral manipulation of people or specific vulnerable groups",
      "Example": "Voice-activated toys that encourage dangerous behavior in children"
    },
    {
      "Application": "Social scoring",
      "Example": "Classifying on behavior, socio-economic status or personal characteristics"
    }
  ]
}
```

```
{
  "Application": "Real-time and remote biometric identification systems",
  "Example": "Facial recognition"
}
]
```

Encouraged by these first experiments, let's try to outline some general rules for prompts.

General rules for prompts

A prompt can include context, instructions, input data, and optionally the structure of the desired output (also in the form of explicit examples). Depending on the task, you might need all four pieces or only a couple of them—most likely, instructions and input data.

Designing a prompt is an iterative process. Not surprisingly, the first reply you get from a model might be quite unreasonable. Don't give up; just try again, but be more precise in what you provide, whether it's plain instructions, input data, or context.

Two key points for a good prompt are specificity and descriptiveness.

- *Specificity* means designing prompts to leave as little room for interpretation as possible. By providing explicit instructions and restricting the operational space, developers can guide the language model to generate more accurate and desired outputs.
- *Descriptiveness* plays a significant role in effective prompt engineering. By using analogies and vivid descriptions, developers can provide clear instructions to the model. Analogies serve as valuable tools for conveying complex tasks and concepts, enabling the model to grasp the desired output with improved context and understanding.

General tips for prompting

A more technical tip is to use delimiters to clearly indicate distinct parts of the prompt. This helps the model focus on the relevant parts of the prompt. Usually, backticks or backslashes work well. For instance:

Extract sentiment from the following text delimited by triple backticks: '''Great choice!'''

When the first attempt fails, two simple design strategies might help:

- Doubling down on instructions is useful to reinforce clarity and consistency in the model's responses. Repetition techniques, such as providing instructions both before and after the primary content or using instruction-cue combinations, strengthen the model's understanding of the task at hand.
- Changing the order of the information presented to the model. The order of information presented to the language model is significant. Whether instructions precede the content (summarize the following) or follow it (summarize the preceding) can lead to different

results. Additionally, the order of few-shot examples (which will be covered shortly) can also introduce variations in the model's behavior. This concept is known as recency bias.

One last thing to consider is an exit strategy for the model in case it fails to respond adequately. The prompt should instruct the model with an alternative path—in other words, an out. For instance, when asking a question about some documents, including a directive such as `write 'not found' if you can't find the answer within the document or check if the conditions are satisfied before answering` allows the model to gracefully handle situations in which the desired information is unavailable. This helps to avoid the generation of false or inaccurate responses.

Alternative ways to alter output

When aiming to align the output of an LLM more closely with the desired outcome, there are several options to consider. One approach involves modifying the prompt itself, following best practices and iteratively improving results. Another involves working with inner parameters (also called hyperparameters) of the model.

Beyond the purely prompt-based conversational approach, there are a few screws to tighten—comparable to the old-but-gold hyperparameters in the classic machine learning approach. These include the number of tokens, temperature, top_p (or nucleus) sampling, frequency penalties, presence penalties, and stop sequences.

Temperature versus top_p

Temperature (T) is a parameter that influences the level of creativity (or “randomness”) in the text generated by an LLM. The usual range of acceptable values is 0 to 2, but it depends on the specific model. When the temperature value is high (say, 0.8), the output becomes more diverse and imaginative. Conversely, a lower temperature (say, 0.1), makes the output more focused and deterministic.

Temperature affects the probability distribution of potential tokens at each step of the generation process. In practice, when choosing the next token, a model with a temperature of 0 will always choose the most probable one, while a model with a higher temperature will choose a token more or less randomly. A temperature of 0, therefore, would make the model entirely deterministic.



Note As discussed in Chapter 1, the temperature parameter works on the LLM's last layer, being a parameter of the softmax function.

An alternative technique called top_p sampling (or nucleus sampling) is also useful for altering the default behavior of the LLM when generating the next token. With top_p sampling, instead of considering all possible tokens, the LLM focuses only on a subset of tokens (known as the nucleus) whose cumulative probability mass adds up to a specified threshold called top_p.

With top_p, the range of acceptable values is from 0 to 1. If, for instance, if the top_p value is set to 0.3, the language model will consider only tokens that contribute to the top 30% of the probability mass for the next token. This means lower top_p values limit the vocabulary of the model.

Both temperature and top_p sampling serve as powerful tools for controlling an LLM's behavior, enabling different levels of creativity and control. They can be used independently or in combination, although it is generally recommended to alter only one at a time.

Adjusting these parameters makes them versatile for a wide range of applications. For instance, for a creative-writing task, a temperature of 1.3 (while also playing around with top_p) could be good, while for a code-generation task, 0.4 (again, while also experimenting with top_p) would be better.

Consider this prompt:

Rephrase the following text:

```
<<<When aiming to align the output of a large language model (LLM) more closely with the desired outcome, there are several options to consider. One approach involves modifying the prompt itself, while another involves working with hyperparameters of the model>>>
```

With a temperature of 2, the prompt returns something like the following (which is grammatically incorrect):

```
When attempting to have a large language model (LLM) output that reflects the desired outcome, there are numerous alternatives to ponder. adapting the prompt itself is called one approach that credits sustainability. Grounding on hyperparameters programs appertains settings amidst the indicators simultaneously participates confidentiality impact later goals aren abbreviated instructions. Otherwise instructions brings alternative ramification which doesn omit lengthy resolution reaching goals which aren susceptible to evaluations constantly... likelihood advancement adjustment exemplark.
```

With a temperature of 0, it returns the following:

```
To achieve better alignment between the output of a language model (LLM) and the desired outcome, there are various options to explore. One option is to adjust the prompt, while another involves tweaking the model's hyperparameters.
```

In the middle, with a temperature of 0.8, the result looks like this:

```
To achieve a greater degree of alignment between the output of a language model (LLM) and the intended result, there are various approaches to consider. One tactic involves adjusting the prompt directly, whereas an alternative involves manipulating the model's hyperparameters.
```

Frequency and presence penalties

Another set of parameters is the frequency and presence penalty. These add a penalty when calculating probability of the next token. This results in a recalculation of each probability, which ultimately affects which token is chosen.

The frequency penalty is applied to tokens that have already been mentioned in the preceding text (including the prompt). It is scaled based on the number of times the token has appeared. For example, a token that has appeared five times receives a higher penalty, reducing its likelihood of appearing again, than a token that has appeared only once. The presence penalty, on the other hand, applies a penalty to tokens regardless of their frequency. Once a token has appeared at least once before, it will be subject to the penalty. The range of acceptable values for both is from -2 to 2.

These parameter settings are valuable for eliminating (or promoting, in the case of negative values) repetitive elements from generated outputs. For instance, consider this prompt:

Rephrase the following text:

```
<<<When aiming to align the output of a large language model (LLM) more closely with the desired outcome, there are several options to consider. One approach involves modifying the prompt itself, while another involves working with hyperparameters of the model>>>
```

With a frequency penalty of 2, it returns something like:

To enhance the accuracy of a large language model's (LLM) output to meet the desired result, there are various strategies to explore. One method involves adjusting the prompt itself, whereas another entails manipulating the model's hyperparameters.

While with a frequency penalty of 0, it returns something like:

There are various options to consider when attempting to better align the output of a language model (LLM) with the desired outcome. One option is to modify the prompt, while another is to adjust the model's hyperparameters.

Max tokens and stop sequences

The max tokens parameter specifies the maximum number of tokens that can be generated by the model, while the stop sequence parameter instructs the language model to halt the generation of further content. Stop sequences are in fact an additional mechanism for controlling the length of the model's output.



Note The model is limited by its inner structure. For instance, GPT-4 is limited to a max number of 32,768 tokens, including the entire conversation and prompts, while GPT-4-turbo has a context window of 128k tokens.

Consider the following prompt:

```
Paris is the capital of
```

The model will likely generate France. If a full stop (.) is designated as the stop sequence, the model will cease generating text when it reaches the end of the first sentence, regardless of the specified token limit.

A more complex example can be built with a few-shot approach, which uses a pair of angled brackets (<< ... >>) on each end of a sentiment. Considering the following prompt:

Extract sentiment from the following tweets:

```
Tweet: I love this match!
```

```
Sentiment: <<positive>>
```

```
Tweet: Not sure I completely agree with you
```

```
Sentiment: <<neutral>>
```

```
Tweet: Amazing movie!!!
```

```
Sentiment:
```

Including the angled brackets instructs the model to stop generating tokens after extracting the sentiment.

By using stop sequences strategically within prompts, developers can ensure that the model generates text up to a specific point, preventing it from producing unnecessary or undesired information. This technique proves particularly useful in scenarios where precise and limited-length responses are desired, such as when generating short summaries or single-sentence outputs.

Setting up for code execution

Now that you've learned the basic theoretical background of prompting, let's bridge the gap between theory and practical implementation. This section transitions from discussing the intricacies of prompt engineering to the hands-on aspect of writing code. By translating insights into executable instructions, you'll explore the tangible outcomes of prompt manipulation.

In this section, you'll focus on OpenAI models, like GPT-4, GPT-3.5-turbo, and their predecessors. (Other chapters might use different models.) For these examples, .NET and C# will be used mainly, but Python will also be used at some point.

Getting access to OpenAI APIs

To access OpenAI APIs, there are multiple options available. You can leverage the REST APIs from OpenAI or Azure OpenAI, the Azure OpenAI .NET or Python SDK, or the OpenAI Python package.

In general, Azure OpenAI Services enable Azure customers to use those advanced language AI models, while still benefiting from the security and enterprise features offered by Microsoft Azure, such as private networking, regional availability, and responsible AI content filtering.

At first, directly accessing OpenAI could be the easiest choice. However, when it comes to enterprise implementations, Azure OpenAI is the more suitable option due to its alignment with the Azure platform and its enterprise-grade features.

To get started with Azure OpenAI, your Azure subscription must include access to Azure OpenAI, and you must set up an Azure OpenAI Service resource with a deployed model.

If you choose to use OpenAI directly, you can create an API key on the developer site (<https://platform.openai.com/>).

In terms of technical differences, OpenAI uses the `model` keyword argument to specify the desired model, whereas Azure OpenAI employs the `deployment_id` keyword argument to identify the specific model deployment to use.

Chat Completion API versus Completion API

OpenAI APIs offer two different approaches for generating responses from language models: the Chat Completion API and the Completion API. Both are available in two modes: a standard form, which returns the complete output once ready, and a streaming version, which streams the response token by token.

The Chat Completion API is designed for chat-like interactions, where message history is concatenated with the latest user message in JSON format, allowing for controlled completions. In contrast, the Completion API provides completions for a single prompt and takes a single string as input.

The back-end models used for the two APIs differ:

- The Chat Completion API supports GPT-4-turbo, GPT-4, GPT-4-0314, GPT-4-32k, GPT-4-32k-0314, GPT-3.5-turbo, and GPT-3.5-turbo-0301.
- The Completion API includes older (but still good for some use cases) models, such as text-davinci-003, text-davinci-002, text-curie-001, text-babbage-001, and text-ada-001.

One advantage of the Chat Completion API is the role selection feature, which enables users to assign roles to different entities in the conversation, such as user, assistant, and, most importantly, system. The first system message provides the model with the main context and instructions “set in stone.” This helps in maintaining consistent context throughout the interaction. Moreover, the system message helps set the behavior of the assistant. For example, you can modify the personality or tone of the assistant or give specific instructions on how it should respond. Additionally, the Chat Completion API allows for longer conversational context to be appended, enabling a more dynamic conversation flow. In contrast, the Completion API does not include the role selection or conversation formatting features. It takes a single prompt as input and generates a response accordingly.

Both APIs provide `finish_reasons` in the response to indicate the completion status. Possible `finish_reasons` values include `stop` (complete message or a message terminated by a stop sequence), `length` (incomplete output due to token limits), `function_call` (model calling a function), `content_filter` (omitted content due to content filters), and `null` (response still in progress).

Although OpenAI recommends the Chat Completion API for most use cases, the raw Completion API sometimes offers more potential for creative structuring of requests, allowing users to construct their own JSON format or other formats. The JSON output can be forced in the Chat Completion API by using the JSON mode with the `response_format` parameter set to `json_object`.

To summarize, the Chat Completion API is a higher-level API that generates an internal prompt and calls some lower-level API and is suited for chat-like interactions with role selection and conversation formatting. In contrast, the Completion API is focused on generating completions for individual prompts.

It’s worth mentioning that the two APIs are to some extent interchangeable. That is, a user can force the format of a Chat Completion response to reflect the format of a Completion response by constructing a request using a single user message. For instance, one can translate from English to Italian with the following Completion prompt:

```
Translate the following English text to Italian: "{input}"
```

An equivalent Chat Completion prompt would be:

```
[{"role": "user", "content": "Translate the following English text to Italian: \"{input}\""}]
```

Similarly, a user can use the Completion API to mimic a conversation between a user and an assistant by appropriately formatting the input.

Setting things up in C#

You can now set things up to use Azure OpenAI API in Visual Studio Code through interactive .NET notebooks, which you will find in the source code that comes with this book. The model used is GPT-3.5-turbo. You set up the necessary NuGet package—in this case, `Azure.AI.OpenAI`—with the following line:

```
#r "nuget: Azure.AI.OpenAI, 1.0.0-beta.12"
```

Then, moving on with the C# code:

```
using System;
using Azure.AI.OpenAI;
var AOAI_ENDPOINT = Environment.GetEnvironmentVariable("AOAI_ENDPOINT");
var AOAI_KEY = Environment.GetEnvironmentVariable("AOAI_KEY");
var AOAI_DEPLOYMENTID = Environment.GetEnvironmentVariable("AOAI_DEPLOYMENTID");
var AOAI_chat_DEPLOYMENTID = Environment.GetEnvironmentVariable("AOAI_chat_DEPLOYMENTID");
var endpoint = new Uri(AOAI_ENDPOINT);
var credentials = new Azure.AzureKeyCredential(AOAI_KEY);
var openAIClient = new OpenAIClient(endpoint, credentials);
var completionOptions = new ChatCompletionsOptions
{
    DeploymentName=AOAI_DEPLOYMENTID,
    MaxTokens=500,
    Temperature=0.7f,
    FrequencyPenalty=0f,
    PresencePenalty=0f,
    NucleusSamplingFactor=1,
    StopSequences={}
};

var prompt =
    @"rephrase the following text: <<<When aiming to align the output of a language model (LLM)
more closely with the desired outcome, there are several options to consider. One approach
involves modifying the prompt itself, while another involves working with hyperparameters of the
model>>>";

completionOptions.Messages.Add(new ChatRequestUserMessage (prompt));
var response = await openAIClient.GetChatCompletionsAsync(completionOptions);
var completions = response.Value;
completions.Choices[0].Message.Content.Display();
```

After running this code, one possible output displayed in the notebook is as follows:

There are various ways to bring the output of a language model (LLM) closer to the intended result. One method is to adjust the prompt, while another involves tweaking the model's hyperparameters.

Note that the previous code uses the Chat Completion version of the API. A similar result could have been obtained through the following code, which uses the Completion API and an older model:

```
var completionOptions = new CompletionsOptions
{
    DeploymentName=AOAI_DEPLOYMENTID,
    Prompts={prompt},
    MaxTokens=500,
    Temperature=0.2f,
    FrequencyPenalty=0.0f,
    PresencePenalty=0.0f, NucleusSamplingFactor=1,
    StopSequences={"."}
};
Completions response = await openAIClient.GetCompletionsAsync(completionOptions);
response.Choices.First().Text.Display();
```

Setting things up in Python

If you prefer working with Python, put the following equivalent code in a Jupyter Notebook:

```
import os
import openai
from openai import AzureOpenAI
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

client = AzureOpenAI(
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_KEY"),
    openai.api_version="2023-09-01-preview"
)
deployment_name=os.getenv("AOAI_DEPLOYMENTID")
context = [ {'role':'user', 'content':"rephrase the following text: 'When aiming to align the
output of a language model (LLM) more closely with the desired outcome, there are several
options to consider: one approach involves modifying the prompt itself, while another involves
working with hyperparameters of the model.'"} ]
response = client.chat.completions.create(
    model=deployment_name,
    messages=context,
    temperature=0.7)
response.choices[0].message["content"]
```

This is based on OpenAI Python SDK v.1.6.0, which can be installed via `pip install openai`.

Basic techniques

Prompt engineering involves understanding the fundamental behavior of LLMs to construct prompts effectively. Prompts consist of different components: instructions, primary content, examples, cues, and supporting content (also known as additional context or knowledge). Instructions guide the model on what to do, while primary content is the main text being processed. Examples provide desired behavior demonstrations, while cues act as a jumpstart for the model's output. Supporting content

provides additional information to influence the output, such as knowledge to search for before answering. By strategically combining these elements, you can design prompts that elicit the desired responses from the model.

This section covers basic techniques for mastering the art of prompting.

Zero-shot scenarios

Whenever a task, assigned to a model through a prompt, is given without any specific example of the desired output, it's called *zero-shot prompting*. Basic scenarios might include:

- **Proper text completion** For example, writing an email or a medical record
- **Topic extraction** For example, to classify customers' emails
- **Translations and sentiment analysis** For example, to label as positive/negative a tweet or to translate users' reviews to the same language
- **Style-impersonation** For example, Shakespeare, Hemingway, or any other notorious personality the model may have been trained on.

The basic theory

Given the huge amount of data that LLMs are trained on, these models can perform an increasing number of "zero-shot" tasks. In fact, the model already knows how to act in those cases. It has incorporated the ability to perform these tasks during its considerable training. However, the more clarity expressed in the prompt, the more effective and accurate the output will be.



Note Clear prompts might not be short. In many situations, longer prompts provide more clarity and context.

A few examples

A basic example of a zero-shot prompt might look like this:

```
Extract sentiment from the following text delimited by triple backticks.  
'''Language models have revolutionized the way we interact with technology, empowering us to  
generate creative content, explore new ideas, and enhance our communication. LLMs offer immense  
potential for unlocking innovation and improving various aspects of our lives, opening up  
exciting possibilities for the future.'''
```

An output would be:

The sentiment of the provided text is positive.

Trying topic extractions, it would be:

```
Determine at most three topics that are being discussed in the following text, delimited by  
triple backticks.
```

```
'''Language models have revolutionized the way we interact with technology, empowering us to generate creative content, explore new ideas, and enhance our communication. LLMs offer immense potential for unlocking innovation and improving various aspects of our lives, opening up exciting possibilities for the future.'''
```

Here's a possible resulting output:

```
The three topics discussed in the text are:
```

```
The impact of language models on technology interaction: Language models have revolutionized the way we interact with technology, enabling us to generate creative content, explore new ideas, and enhance communication.
```

```
The potential of LLMs for innovation: LLMs offer immense potential for unlocking innovation and improving various aspects of our lives. They open up exciting possibilities for the future.
```

```
Benefits and improvements in various aspects of life: The text highlights how LLMs can enhance communication, generate creative content, and explore new ideas, thereby improving different aspects of our lives.
```

Iterative refining

Prompt engineering is a matter of refining. Trying to improve the preceding result, you might want to explicitly list the sentiment the model should output and to limit the output to the sentiment only. For example, a slightly improved prompt might look like the following:

```
Extract sentiment (positive, neutral, negative, unknown) from the following text, delimited by triple backticks.
```

```
'''Language models have revolutionized the way we interact with technology, empowering us to generate creative content, explore new ideas, and enhance our communication. LLMs offer immense potential for unlocking innovation and improving various aspects of our lives, opening up exciting possibilities for the future.'''
```

```
Return only one word indicating the sentiment.
```

This would result in the following output:

```
Positive
```

Likewise, regarding the topic extraction, you might want only one or two words per topic, each separated by commas:

```
Determine at most three topics that are being discussed in the following text, delimited by triple backticks.
```

```
Format the response as a list of at most 2 words, separated by commas.
```

```
'''Language models have revolutionized the way we interact with technology, empowering us to generate creative content, explore new ideas, and enhance our communication. LLMs offer immense potential for unlocking innovation and improving various aspects of our lives, opening up exciting possibilities for the future.'''
```

The result would look like:

```
Language models, Interaction with technology, LLM potential.
```

Few-shot scenarios

Zero-shot capabilities are impressive but face important limitations when tackling complex tasks. This is where few-shot prompting comes in handy. Few-shot prompting allows for in-context learning by providing demonstrations within the prompt to guide the model's performance.

A few-shot prompt consists of several examples, or *shots*, which condition the model to generate responses in subsequent instances. While a single example may suffice for basic tasks, more challenging scenarios call for increasing numbers of demonstrations.

When using the Chat Completion API, few-shot learning examples can be included in the system message or, more often, in the messages array as user/assistant interactions following the initial system message.



Note Few-shot prompting is useful if the accuracy of the response is too low. (Measuring accuracy in an LLM context is covered later in the book.)

The basic theory

The concept of few-shot (or in-context) learning emerged as an alternative to fine-tuning models on task-specific datasets. Fine-tuning requires the availability of a base model. OpenAI's available base models are GPT-3.5-turbo, davinci, curie, babbage, and ada, but not the latest GPT-4 and GPT-4-turbo models. Fine-tuning also requires a lot of well-formatted and validated data. In this context, developed as LLM sizes grew significantly, few-shot learning offers advantages over fine-tuning, reducing data requirements and mitigating the risk of overfitting, typical of any machine learning solution.

This approach focuses on priming the model for inference within specific conversations or contexts. It has demonstrated competitive performance compared to fine-tuned models in tasks like translation, question answering, word unscrambling, and sentence construction. However, the inner workings of in-context learning and the contributions of different aspects of shots to task performance remain less understood.

Recent research has shown that ground truth demonstrations are not essential, as randomly replacing correct labels has minimal impact on classification and multiple-choice tasks. Instead, other aspects of demonstrations, such as the label space, input text distribution, and sequence format, play crucial roles in driving performance. For instance, the two following prompts for sentiment analysis—the first with correct labels, and the second with completely wrong labels—offer similar performance.

```
Tweet: "I hate it when I have no wifi"  
Sentiment: Negative  
Tweet: "Loved that movie"  
Sentiment: Positive  
Tweet: "Great car!!!"  
Sentiment: Positive
```

```
Tweet: {new tweet}  
Sentiment:
```

And:

```
Tweet: "I hate it when I have no wifi"  
Sentiment: Positive  
Tweet: "Loved that movie"  
Sentiment: Negative  
Tweet: "Great car!!!"  
Sentiment: Negative
```

```
Tweet: {new tweet}  
Sentiment:
```

In-context learning may struggle with tasks that lack precaptured input-label correspondence. This suggests that the intrinsic ability to perform a task is obtained during training, with demonstrations (or shots) primarily serving as a task locator.

A few examples

One of the most famous examples of the efficiency of few-shot learning prompts is one taken from a paper by Brown et al. (2020), where the task is to correctly use a new word in a sentence:

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is: We were traveling in Africa and we saw these very cute whatpus. To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

The model would correctly output something like:

```
We were so excited that we couldn't help but farduddle when our favorite team won the  
championship.
```

A very good use case for few-shot learning is writing something in a given style, such as code documentation or social media posts. Whenever there is a pattern or a format, and explaining it is more complex than showing it, it's worth trying few-shot prompting. For instance, the following prompt would generate a consistent documentation:

```
/// <summary>  
/// Returns detailed information about a specific event  
/// </summary>  
/// <param name="org">Name of the organization</param>  
/// <param name="year">Year of the event</param>  
/// <param name="tournamentId">Internal ID of the event</param>  
/// <returns>Detailed info about the event</returns>  
/// <remarks>Sample events to try are: 2892, 2331, 1028. Try also 8392.  
/// If any of the parameters are invalid, the endpoint returns 404.</remarks>  
public IActionResult Tournament(string org, int? year, string eventId)  
  
public IActionResult Team(string org, string teamId)
```

Based on the model version and the API (Completion or Chat Completion), the system might tend to explain the code and the documentation instead of producing new documentation for the presented

method (`IActionResult Team`). That's why a better and more precise prompt would be the following, which includes specific instructions:

```
Your task is to generate documentation in a consistent style.
/// <summary>
/// Returns detailed information about a specific event
/// </summary>
/// <param name="org">Name of the organization</param>
/// <param name="year">Year of the event</param>
/// <param name="tournamentId">Internal ID of the event</param>
/// <returns>Detailed info about the event</returns>
/// <remarks>Sample events to try are: 2892, 2331, 1028. Try also 8392.
/// If any of the parameters are invalid, the endpoint returns 404.</remarks>
public IActionResult Tournament(string org, int? year, string eventId)

public IActionResult Team(string org, string teamId)
```

Iterative refining

The last prompt is likely to generate a similar response:

```
/// <summary>
/// Returns detailed information about a specific event.
/// </summary>
/// <param name="org">The name of the organization.</param>
/// <param name="year">The year of the event.</param>
/// <param name="tournamentId">The internal ID of the event.</param>
/// <returns>Detailed information about the event.</returns>
/// <remarks>
/// Sample event IDs to try are: 2892, 2331, 1028. You can also try 8392.
/// If any of the parameters are invalid, the endpoint returns a 404 error.
/// </remarks>
public IActionResult Event(string org, int? year, string eventId) {
// Implementation code here
}
/// <summary>
/// Retrieves information about a specific team.
/// </summary>
/// <param name="org">The name of the organization.</param>
/// <param name="teamId">The ID of the team.</param>
/// <returns>Information about the team.</returns>
public IActionResult Team(string org, string teamId) {
// Implementation code here
}
}
```

The `Event` method is an action that returns detailed information about a specific event. It takes three parameters: `org` (representing the name of the organization), `year` (representing the year of the event, nullable), and `eventId` (representing the internal ID of the event). This method should be called to retrieve detailed information about an event.

The `Team` method is another action that retrieves information about a specific team. It takes two parameters: `org` (representing the name of the organization) and `teamId` (representing the ID of the team). This method should be called to fetch information about a team.

As is easily observable from the previous examples, all these generative models want to write text. This is reasonable because they were trained to predict the next token. It is very likely, then, that a good prompt needs to incorporate something to restrict the output.

Following up with the last example, a better prompt might be:

Your task is to generate code comments in a consistent style with the reference. The response must include only the comments for the new method, without any additional explanation or comment.

Style reference:

```
/// <summary>
/// Returns detailed information about a specific event
/// </summary>
/// <param name="org">Name of the organization</param>
/// <param name="year">Year of the event</param>
/// <param name="tournamentId">Internal ID of the event</param>
/// <returns>Detailed info about the event</returns>
/// <remarks>Sample events to try are: 2892, 2331, 1028. Try also 8392.
/// If any of the parameters are invalid, the endpoint returns 404.</remarks>
public IActionResult Tournament(string org, int? year, string eventId)
```

New method:

```
public IActionResult Team(string org, string teamId)
```

This might still output some additional explanation:

```
/// <summary>
/// Returns detailed information about a specific team.
/// </summary>
/// <param name="org">Name of the organization</param>
/// <param name="teamId">ID of the team</param>
/// <returns>Detailed info about the team</returns>
```

Please note that this is the code comment for the new Team method. It retrieves detailed information about a specific team. The method takes two parameters: org, which represents the name of the organization, and teamId, which is the ID of the team. The method returns detailed information about the team.

At this point, to prevent the model from producing additional text, you might need a different strategy: asking the model to check if certain conditions on the output are satisfied. This can be done by appending this line to the former prompt:

```
Check if the output contains additional text and, if so, remove it.
```

Asking the model to check if certain conditions in the input are satisfied is a very useful technique. It can also be exploited for more standard tasks, such as form or json/xml/html validation.

In this case, you also tried to validate the output text. This is more of a trick than a technique because the model doesn't really produce the full output to be validated. Still, it works as a guardrail. A better way to achieve the same result would have been to add one more API call with the former prompt or, as explored later in book, involving a framework like Microsoft Guidance or Guardrails AI.

Considering this, it's important to stress that these models work better when they are told what they need to do instead of what they must avoid.

Chain-of-thought scenarios

While standard few-shot prompting is effective for many tasks, it is not without limitations—particularly when it comes to more intricate reasoning tasks, such as mathematical and logical problems, as well as tasks that require the execution of multiple sequential steps.



Note Later models such as GPT-4 perform noticeably better on logical problems, even with simple non-optimized prompts.

When few-shot prompting proves insufficient, it may indicate the need for fine-tuning models (if these are an option, which they aren't for GPT-4 and GPT-4-turbo) or exploring advanced prompting techniques. One such technique is chain-of-thought (CoT) prompting. You use CoT prompting to track down all the steps (thoughts) performed by the model to draw the solution.

As presented in the work of Wei et al. (2022), this technique gives the model time to think, enhancing reasoning abilities by incorporating intermediate reasoning steps. When used in conjunction with few-shot prompting, it leads to improved performance on intricate tasks that demand prior reasoning for accurate responses.



Note The effectiveness of CoT prompting is observed primarily when employed with models consisting of approximately 100 billion parameters. Smaller models tend to generate incoherent chains of thought, resulting in lower accuracy compared to standard prompting. The performance improvements achieved through CoT prompting generally scale with the size of the model.

The basic theory

Anyone young enough to remember their days as a student will know that during exams, the brain stops functioning. Most of the time, one tries to answer the professor's question, almost guessing the solution, without really thinking about it. LLMs do the same thing. Sometimes they continue the sentence without thinking; they just make a guess.

The basic idea behind CoT is to instruct the model to take time to think, allowing it to reconstruct the reasoning steps during response generation. This simple idea significantly enhances the model's ability to perform complex tasks. This makes sense when you consider how language models produce responses. They attempt to predict the next token, so the more intermediate "reasoning" tokens they generate, the more the model needs to establish coherence among them and, ideally, provide a correct response.



Note By only asking the model for the final answer, you leave limited room for the model to verify the coherence between the question (prompt) and its response; in contrast, explicitly outlining all the steps helps the model find the logical thread.

A few examples

Following the professor-student example, the first two attempts to improve the output of the model might be the classical “make sure the answer is correct” or “let’s think step by step” approach. For instance, consider the following easy problem:

I bought 20 pens, gave 12 pens to my son, 3 to my daughter, 1 to my mom. I then bought 4 more pens and lost 1.

Output the number of pens I have now.

Regarding logical problems, temperature and model version definitely matter. Lower temperature will keep things more deterministic, while the latest models will have better accuracy, because they have been trained on a larger dataset for logical problems.

With a temperature of 0.2, a GPT-3.5-turbo model will provide an answer like the following, whereas a GPT-4 model will probably say, “You currently have 7 pens”:

I have 8 pens now.

Explanation: Initially, I had 20 pens. I gave away $12 + 3 + 1 = 16$ pens. So I had $20 - 16 = 4$ pens left. Then I bought 4 more pens, so I had $4 + 4 = 8$ pens. But I lost 1 pen, so I have $8 - 1 = 7$ pens. Therefore, the output is 8 pens.



Note Older models would have probably responded with the number only, without any reasoning. Newer models have been reinforced through step-by-step prompting.

The answer is wrong. Worse, the answer is incoherent. It says 8, while it correctly calculates 7.

Let’s try the following prompt to see if it provides more reasonable output:

I bought 20 pens, gave 12 pens to my son, 3 to my daughter, 1 to my mom. I then bought 4 more pens and lost 1.

Output the number of pens I have now and make sure the answer is correct and coherent. Make sure the answer and the explanation are correct and coherent before answering.

Here, the output is as follows:

After buying 20 pens, I had 20 pens. After giving 12 pens to my son, I had 8 pens. After giving 3 pens to my daughter, I had 5 pens. After giving 1 pen to my mom, I had 4 pens. After buying 4 more pens, I had 8 pens. After losing 1 pen, I had 7 pens.

Therefore, I have 7 pens now.

That’s correct.

An equivalent prompt might have been:

I bought 20 pens, gave 12 pens to my son, 3 to my daughter, 1 to my mom. I then bought 4 more pens and lost 1.

Output the number of pens I have now. Let's think it step by step.

And the output:

1. Start with 20 pens.
2. Gave 12 pens to my son, so I have $20 - 12 = 8$ pens left.
3. Gave 3 pens to my daughter, so I have $8 - 3 = 5$ pens left.
4. Gave 1 pen to my mom, so I have $5 - 1 = 4$ pens left.
5. Bought 4 more pens, so I have $4 + 4 = 8$ pens now.
6. Lost 1 pen, so I have $8 - 1 = 7$ pens now.

Therefore, I have 7 pens now.

This is called a zero-shot chain-of-thought prompt; the magic sentence, “Let’s think it step by step,” forces the model to take time to think and to write down all the steps.

At this point, to get the final answer, you could ask the model to produce a structured output or make one more API call with a simple prompt like, “Extract only the final answer from this text”:

I bought 20 pens, gave 12 pens to my son, 3 to my daughter, 1 to my mom. I then bought 4 more pens and lost 1.

Output the number of pens I have now. Let's think it step by step. Output a json with: explanation (string) and result (int).

The result would look like:

```
{"explanation": "Initially, I had 20 pens. After giving 12 to my son, I had 8 left. Then, I gave 3 to my daughter, leaving me with 5. Giving 1 to my mom left me with 4 pens. Buying 4 more pens gave me a total of 8 pens. Unfortunately, I lost 1 pen, leaving me with a final total of 7 pens.", "result": 7}
```

Possible extensions

Combining the few-shot technique with the chain-of-thought approach can give the model some examples of step-by-step reasoning to emulate. This is called *few-shot chain-of-thought*. For instance:

Which is the more convenient way to reach the destination, balancing costs and time?

Option 1: Take a 20-minute walk, then a 15-minute bus ride (2 dollars), and finally a 5-minute taxi ride (15 dollars).

Option 2: Take a 30-minute bike ride, then a 10-minute subway ride (2 dollars), and finally a 5-minute walk.

Option 1 will take $20 + 15 + 5 = 40$ minutes. Option 1 will cost 17 dollars.

Option 2 will take $30 + 10 + 5 = 45$ minutes. Option 2 will cost 2 dollars.

Since Option 1 takes 40 minutes and Option 2 takes 45 minutes, Option 1 is quicker, but Option 2 is cheaper by far. Option 2 is better.

Which is the better way to get to the office?

Option 1: 40 minutes train (5 dollars), 15 mins walk

Option 2: 10-minutes taxi ride (15 dollars), 10-minutes subway (2 dollars), 2-mins walk

An extension of this basic prompting technique is Auto-CoT. This basically leverages the few-shot CoT approach, using a prompt to generate more samples (shots) of reasoning, which are then concatenated into a final prompt. Essentially, the idea is to auto-generate a few-shot CoT prompt.

Beyond chain-of-thought prompting, there is one more sophisticated idea: tree of thoughts. This technique can be implemented in essentially two ways. The first is through a single prompt, like the following:

```
Consider a scenario where three experts approach this question.
Each expert will contribute one step of their thought process and share it with the group.
Subsequently, all experts will proceed to the next step.
If any expert realizes they have made a mistake at any stage, they will exit the process.
The question is the following: {question}
```

A more sophisticated approach to tree of thoughts requires writing some more code, with different prompts running (maybe also with different temperatures) and producing reasoning paths. These paths are then evaluated by another model instance with a scoring/voting prompt, which excludes wrong ones. At the end, a certain mechanism votes (for coherence or majority) for the correct answer.

A few more emerging but relatively easy-to-implement prompting techniques are analogical prompting (by Google DeepMind), which asks the model to recall a similar problem before solving the current one; and step-back prompting, which prompts the model to step back from the specific instance and contemplate the general principle at hand.

Fundamental use cases

Having explored some more intricate techniques, it's time to shift the focus to practical applications. In this section, you'll delve into fundamental use cases where these techniques come to life, demonstrating their effectiveness in real-world scenarios. Some of these use cases will be expanded in later chapters, including chatbots, summarization and expansion, coding helpers, and universal translators.

Chatbots

Chatbots have been around for years, but until the advent of the latest language models, they were mostly perceived as a waste of time by users who had to interact with them. However, these new models are now capable of understanding even when the user makes mistakes or writes poorly, and they respond coherently to the assigned task. Previously, the thought of people who used chatbots was almost always, "Let me talk to a human; this bot doesn't understand." Soon, however, I expect we will reach something like the opposite: "Let me talk to a chatbot; this human doesn't understand."

System messages

With chatbots, system messages, also known as *metaprompts*, can be used to guide the model's behavior. A metaprompt defines the general guidelines to be followed. Still, while using these templates and guidelines, it remains essential to validate the responses generated by the models.

A good system prompt should define the model's profile, capabilities, and limitations for the specific scenario. This involves:

- Specifying how the model should complete tasks and whether it can use additional tools

- Clearly outlining the scope and limitations of the model's performance, including instructions for off-topic or irrelevant prompts
- Determining the desired posture and tone for the model's responses
- Defining the output format, including language, syntax, and any formatting preferences
- Providing examples to demonstrate the model's intended behavior, considering difficult use cases and CoT reasoning
- Establishing additional behavioral guardrails by identifying, prioritizing, and addressing potential harms

Collecting information

Suppose you want to build a booking chatbot for a hotel brand group. A reasonable system prompt might look something like this:

You are a HotelBot, an automated service to collect hotel bookings within a hotel brand group, in different cities.

You first greet the customer, then collect the booking, asking the name of the customer, the city the customer wants to book, room type and additional services.

You wait to collect the entire booking, then summarize it and check for a final time if the customer wants to add anything else.

You ask for arrival date, departure date, and calculate the number of nights. You ask for a passport number. Make sure to clarify all options and extras to uniquely identify the item from the pricing list.

You respond in a short, very conversational friendly style. Available cities: Rome, Lisbon, Bucharest.

The hotel rooms are:
 single 150.00 per night
 double 250 per night
 suite 350 per night

Extra services:
 parking 20.00 per day,
 late checkout 100.00
 airport transfer 50.00
 SPA 30.00 per day

Consider that the previous prompt is only a piece of a broader application. After the system message is launched, the application should ask the user to start an interaction; then, a proper conversation between the user and chatbot should begin.

For a console application, this is the basic code to incorporate to start such an interaction:

```
var chatCompletionsOptions = new ChatCompletionsOptions
{
    DeploymentName = AOAI_chat_DEPLOYMENTID
    Messages =
```

```

        {
            new ChatRequestSystemMessage(systemPrompt),
            new ChatRequestUserMessage("Introduce yourself"),
        }
    };
    while (true)
    {
        Console.WriteLine();
        Console.Write("HotelBot: ");
        var chatCompletionsResponse = await openAIClient.GetChatCompletionsAsync(chatCompletions
Options);
        var chatMessage = chatCompletionsResponse.Value.Choices[0].Message;
        Console.Write(chatMessage.Content);
        chatCompletionsOptions.Messages.Add(new ChatRequestAssistantMessage(chatMessage.
Content));
        Console.WriteLine();
        Console.Write("Enter a message: ");
        var userMessage = Console.ReadLine();
        chatCompletionsOptions.Messages.Add(new ChatRequestUserMessage(userMessage));
    }
}

```



Note When dealing with web apps, you must also consider the UI of the chat.

Summarization and transformation

Now that you have a prompt to collect a hotel booking, the hotel booking system will likely need to save it—calling an API or directly saving the information in a database. But all it has is unstructured natural language, coming from the conversation between the customer and the bot. A prompt to summarize and convert to structured data is needed:

Return a json summary of the previous booking. Itemize the price for each item.

The json fields should be

- 1) name,
- 2) passport,
- 3) city,
- 4) room type with total price,
- 5) list of extras including total price,
- 6) arrival date,
- 7) departure date,
- 8) total days
- 9) total price of rooms and extras (calculated as the sum of the total room price and extra price).

Return only the json, without introduction or final sentences.

Simulating a conversation with the HotelBot, a json like the following would be generated from the previous prompt:

```

{"name":"Francesco Esposito","passport":"XXCONTOS0123","city":"Lisbon","room_type":{"single":15
0.00},"extras":{"parking":{"price_per_day":20.00,"total_price":40.00}},"arrival_date":"2023-06-
28","departure_date":"2023-06-30","total_days":2,"total_price":340.00}

```

Expanding

At some point, you might need to handle the inverse problem: generating a natural language summary from a structured JSON. The prompt to handle such a case could be something like:

Return a text summary from the following json, using a friendly style. Write at most two sentences.

```
{"name": "Francesco Esposito", "passport": "XXCONTOS0123", "city": "Lisbon", "room_type": {"single": 150.00}, "extras": {"parking": {"price_per_day": 20.00, "total_price": 40.00}}, "arrival_date": "2023-06-28", "departure_date": "2023-06-30", "total_days": 2, "total_price": 340.00}
```

This would result in a reasonable output:

Francesco Esposito will be staying in Lisbon from June 28th to June 30th. He has booked a single room for \$150.00 per night, and the total price including parking is \$340.00 for 2 days.

Translating

Thanks to pretraining, one task that LLMs excel at is translating from a multitude of different languages—not just natural human languages, but also programming languages.

From natural language to SQL

One famous example taken directly from OpenAI references is the following prompt:

```
### Postgres SQL tables, with their properties:
#
# Employee(id, name, department_id)
# Department(id, name, address)
# Salary_Payments(id, employee_id, amount, date)
#
### A query to list the names of the departments that employed more than 10 employees in the
last 3 months

SELECT
```

This prompt is a classic example of a plain completion (so, Completion API). The last part (SELECT) acts as cue, which is the jumpstart for the output.

In a broader sense, within the context of Chat Completion API, the system prompt could involve providing the database schema and asking the user which information to extract, which can then be translated into an SQL query. This type of prompt generates a query that the user should execute on the database only after assessing the risks. There are other tools to interact directly with the database through agents using the LangChain framework, discussed later in this book. These tools, of course, come with risks; they provide direct access to the data layer and should be evaluated on a case-by-case basis.

Universal translator

Let's consider a messaging app in which each user selects their primary language. They write in that language, and if necessary, a middleware translates their messages into the language of the other user. At the end, each user will read and write using their own language.

The translator middleware could be a model instance with a similar prompt:

Translate the following text from {user1Language} to {user2Language}:

<<<{message1}>>>

A full schema of the interactions would be:

1. User 1 selects its preferred language {user1Language}.
2. User 2 selects its preferred language {user2Language}.
3. One sends a message to the other. Let's suppose User1 writes a message {message1} in {user1Language}.
4. The middleware translates {message1} in {user1Language} to {message1-translated} in {user2Language}.
5. User 2 sees {message1-translated} in its own language.
6. User 2 writes a message {message2} in {user2Language}.
7. The middleware performs the same job and sends the message to User1.
8. And so on....

LLM limitations

So far, this chapter has focused on the positive aspects of LLMs. But LLMs have limitations in several areas:

- LLMs struggle with accurate source citations due to their lack of internet access and limited memory. Consequently, they may generate sources that appear reliable but are incorrect (this is called *hallucination*). Strategies like search-augmented LLMs can help address this issue.
- LLMs tend to produce biased responses, occasionally exhibiting sexist, racist, or homophobic language, even with safeguards in place. Care should be taken when using LLMs in consumer-facing applications and research to avoid biased results.
- LLMs often generate false information when faced with questions on which they have not been trained, confidently providing incorrect answers or hallucinating responses.
- Without additional prompting strategies, LLMs generally perform poorly in math, struggling with both simple and complex math problems.

It is important to be aware of these limitations. You should also be wary of prompt hacking, where users manipulate LLMs to generate desired content. All these security concerns are addressed later in this book.

Summary

This chapter explored various basic aspects of prompt engineering in the context of LLMs. It covered common practices and alternative methods for altering output, including playing with hyperparameters. In addition, it discussed accessing OpenAI APIs and setting things up in C# and Python.

Next, the chapter delved into basic prompting techniques, including zero-shot and few-shot scenarios, iterative refining, chain-of-thought, time to think, and possible extensions. It also examined basic use cases such as booking chatbots for collecting information, summarization, and transformation, along with the concept of a universal translator.

Finally, the chapter discussed limitations of LLMs, including generating incorrect citations, producing biased responses, returning false information, and performing poorly in math.

Subsequent chapters focus on more advanced prompting techniques to take advantage of additional LLM capabilities and, later, third-party tools.

Index

A

- abstraction, 11
- abuse filtering, 133–134
- acceleration, 125
- accessing, OpenAI API, 31
- adjusting the prompt, 29
- adoption, LLM (large language model), 21
- adversarial training, 4
- agent, 53, 81
 - building, 127–128
 - LangChain, 96, 97
 - log, 53–54
 - ReAct, 101
 - security, 137–138
 - semi-autonomous, 80
- agent_scratchpad, 99
- AGI (artificial general intelligence), 20, 22–23
- AI
 - beginnings, 2
 - conventional, 2
 - engineer, 15
 - generative, 1, 4
 - NLP (natural language processing), 3
 - predictive, 3–4
 - singularity, 19
- AirBot Solutions, 93
- algorithm
 - ANN (artificial neural network), 70
 - KNN (k-nearest neighbor), 67, 70
 - MMR (maximum marginal relevance), 71
 - tokenization, 8
- Amazon Comprehend, 144
- analogical prompting, 44
- angled bracket (<<.>>), 30–31
- ANN (artificial neural network), 70
- API, 86. *See also* framework/s
- Assistants, 80
 - Chat Completion, function calling, 62
 - Minimal, 205–208
 - OpenAI
 - REST, 159
- app, translation, 48
- application, LLM-based, 18
- architecture, transformer, 7, 9
- Assistants API, 80
- attack mitigation strategies, 138–139
- Auto-CoT prompting, 43
- automaton, 2
- autoregressive language modeling, 5–6
- Azure AI Content Safety, 133
- Azure Machine Learning, 89

- Azure OpenAI, 8. *See also* customer care chatbot assistant, building
 - abuse monitoring, 133–134
 - content filtering, 134
 - Create Customized Model wizard, 56
 - environment variables, 89
 - topology, 16–17
- Azure Prompt Flow, 145

B

- Back, Adam, 109
- basic prompt, 26
- batch size hyperparameter, 56
- BERT (Bidirectional Encoder Representation from Transformers), 3, 5
- bias, 145
 - logit, 149–150
 - training data, 135
- big tech, 19
- Bing search engine, 66
- BPE (byte-pair encoding), 8
- building, 159, 161
 - corporate chatbot, 181, 186
 - customer care chatbot assistant
 - hotel reservation chatbot, 203–204
- business use cases, LLM (large language model), 14

C

- C#
 - RAG (retrieval augmented generation), 74
 - setting things up, 33–34
- caching, 87, 125
- callbacks, LangChain, 95–96
- canonical form, 153–154
- chain/s, 52–53, 81
 - agent, 53–54, 81
 - building, 81
 - content moderation, 151
 - debugging, 87–88
 - evaluation, 147–148
 - LangChain, 88
 - memory, 82, 93–94
- Chat Completion API, 32–33, 61, 62
- chatbot/s, 44, 56–57
 - corporate, building, 181, 186
 - customer care, building
 - fine-tuning, 54

- hotel reservation, building, 203–204
- system message, 44–45
- ChatGPT, 19
- ChatML (Chat Markup Language), 137
- ChatRequestMessage class, 173
- ChatStreaming method, 175–176
- chunking, 68–69
- Church, Alonzo, 2
- class
 - ChatRequestMessage, 173
 - ConversationBufferMemory, 94
 - ConversationSummaryMemory, 87, 94
 - Embeddings, 106
 - eval, 147
 - helper, 171–172
- classifier, 150–151
- CLI (command-line interface), OpenAI, 55
- CLM (causal language modeling), 6, 11
- CNN (convolutional neural network), 4
- code, 159. *See also* corporate chatbot, building; customer care chatbot assistant, building; hotel reservation chatbot, building
 - C#, setting things up, 33–34
 - function calling, refining the code, 60
 - homemade function calling, 57–60
 - injection, 136–137
 - prompt template, 90
 - Python, setting things up, 34
 - RAG (retrieval augmented generation), 109
 - RAIL spec file, 152
 - splitting, 105–106
- Colang, 153
 - canonical form, 153–154
 - dialog flow, 154
 - scripting, 155
- collecting information, 45–46
- command/s
 - docker-compose, 85
 - pip install doctran, 106
- Completion API, 32–33
- completion call, 52
- complex prompt, 26–27
- compression, 108, 156, 192
- consent, for LLM data, 141
- content filtering, 133–134, 148–149
 - guardrailing, 151
 - LLM-based, 150–151
 - logit bias, 149–150
 - using a classifier, 150–151
- content moderation, 148–149
- conventional AI, 2
- conversational
 - memory, 82–83
 - programming, 1, 14
 - UI, 14
- ConversationBufferMemory class, 94
- ConversationSummaryMemory class, 87, 94
- corporate chatbot, building, 181, 186
 - data preparation, 189–190
 - improving results, 196–197
 - integrating the LLM, managing history, 194
 - LLM interaction, 195–196
 - possible extensions, 200
 - rewording, 191–193
 - scope, 181–182
 - setting up the project and base UI, 187–189
 - tech stack, 182
- cosine similarity function, 67

- cost, framework, 87
- CoT (chain-of-thought) prompting, 41
 - Auto-, 43
 - basic theory, 41
 - examples, 42–43
- Create Customized Model wizard, Azure OpenAI, 56
- customer care chatbot assistant, building
 - base UI, 164–165
 - choosing a deployment model, 162–163
 - integrating the LLM
 - possible extensions, 178
 - project setup, 163–164
 - scope, 160
 - setting up the LLM, 161–163
 - SSE (server-sent event), 173–174
 - tech stack, 160–161
 - workflow, 160

D

- data. *See also* privacy; security
 - bias, 135
 - collection, 142
 - connecting to LLMs, 64–65
 - de-identification, 142
 - leakage, 142
 - protection, 140
 - publicly available, 140
 - retriever, 83–84
 - talking to, 64
- database
 - relational, 69
 - vector store, 69
- data-driven approaches, NLP (natural language processing), 3
- dataset, preparation, 55
- debugging, 87–88
- deep fake, 136, 137
- deep learning, 3, 4
- deep neural network, 4
- dependency injection, 165–167
- deployment model, choosing for a customer care chatbot assistant, 162–163
- descriptiveness, prompt, 27
- detection, PII, 143–144
- development, regulatory compliance, 140–141
- dialog flow, 153, 154
- differential privacy, 143
- discriminator, 4
- doctran, 106
- document/s
 - adding to VolatileMemoryStore, 83–84
 - vector store, 191–192

E

- embeddings, 9, 65, 71
 - dimensionality, 65
 - LangChain, 106
 - potential issues, 68–69
 - semantic search, 66–67
 - use cases, 67–68
 - vector store, 69
 - word, 65
- encoder/decoder, 7

encryption

- encryption, 143
- eval, 147
- evaluating models, 134–135
- evaluation, 144–145
 - chain, 147–148
 - human-based, 146
 - hybrid, 148
 - LLM-based, 146–148
 - metric-based, 146
- EventSource method, 174
- ExampleSelector, 90–91
- extensions
 - corporate chatbot, 200
 - customer care chatbot assistant, 178
 - hotel reservation chatbot, 215–216

F

- federated learning, 143
- few-shot prompting, 14, 37, 90–91
 - checking for satisfied conditions, 40
 - examples, 38–39
 - iterative refining, 39–40
- file-defined function, calling, 114
- fine-tuning, 12, 37, 54–55
 - constraints, 54
 - dataset preparation, 55
 - hyperparameters, 56
 - iterative, 36
 - versus retrieval augmented generation, 198–199
- framework/s, 79. *See also* chain/s
 - cost, 87
 - debugging, 87–88
 - Evals, 135
 - Guidance, 80, 121
 - LangChain, 57, 79, 80, 88
 - orchestration, need for, 79–80
 - ReAct, 97–98
 - SK (Semantic Kernel), 109–110
 - token consumption tracking, 84–86
 - VolatileMemoryStore, adding documents, 83–84
- frequency penalty, 29–30
- FunctionCallingStepwisePlanner, 205–206, 208
- function/s, 91–93
 - call, 1, 56–57
 - cosine similarity, 67
 - definition, 61
 - plug-in, 111
 - semantic, 117, 215
 - SK (Semantic Kernel), 112–114
 - VolatileMemoryStore, 82, 83–84

G

- GAN (generative adversarial network), 4
- generative AI, 4. *See also* LLM (large language model)
 - LLM (large language model), 4–5, 6
- Georgetown-IBM experiment, 3
- get_format_instructions method, 94
- Google, 5
- GPT (Generative Pretrained Transformer), 3, 5, 20, 23
- GPT-3/GPT-4, 1
 - embedding layers, 9

- topology, 16–17
- GptConversationalEngine method, 170–171
- grounding, 65, 72. *See also* RAG (retrieval augmented generation)
- guardrailing, 151
 - Guardrails AI, 151–152
 - NVIDIA NeMo Guardrails, 153–156
- Guidance, 79, 80, 121
 - acceleration, 125
 - basic usage, 122
 - building an agent, 127–128
 - installation, 121
 - main features, 123–124
 - models, 121
 - structuring output and role-based chat, 125–127
 - syntax, 122–123
 - template, 125
 - token healing, 124–125

H

- hallucination, 11, 134, 156–157
- handlebars planner, 119
- HandleStreamingMessage method, 175
- Hashcash, 109
- history
 - LLM (large language model), 2
 - NLP (natural language processing), 3
- HMM (hidden Markov model), 3
- homemade function calling, 57–60
- horizontal big tech, 19
- hotel reservation chatbot, building, 203–204
 - integrating the LLM, 210
 - LLM interaction, 212–214
 - Minimal API setup, 206–208
 - OpenAPI, 208–209
 - possible extensions, 215–216
 - scope, 204–205
 - semantic functions, 215
 - tech stack, 205
- Hugging Face, 17
- human-based evaluation, 146
- human-in-the-loop, 64
- Humanloop, 145
- hybrid evaluation, 148
- hyperparameters, 28, 56

I

- IBM, Candide system, 3
- indexing, 66, 70
- indirect injection, 136–137
- inference, 11, 55
- information-retrieval systems, 199
- inline configuration, SK (Semantic Kernel), 112–113
- input
 - installation, Guidance, 121
- instantiation, prompt template, 80–81
- instructions, 27, 34–35
- “intelligent machinery”, 2
- iterative refining
 - few-shot prompting, 39–40
 - zero-shot prompting, 36

J-K

jailbreaking, 136
JSONL (JSON Lines), dataset preparation, 55

Karpathy, Andrej, 14
KNN (k-nearest neighbor) algorithm, 67, 70
KV (key/value) cache, 125

L

labels, 37–38

LangChain, 57, 79, 80, 88. *See also* corporate chatbot, building

- agent, 96, 97
- callbacks, 95–96
- chain, 88
- chat models, 89
- conversational memory, handling, 82–83
- data providers, 83
- doctran, 106
- Embeddings module, 106
- ExampleSelector, 90–91
- loaders, 105
- metadata filtering, 108
- MMR (maximum marginal relevance) algorithm, 71
- model support, 89
- modules, 88
- parsing output, 94–95
- prompt
- results caching, 87
- text completion models, 89
- text splitters, 105–106
- token consumption tracking, 84
- toolkits, 96
- tracing server, 85–86
- vector store, 106–107

LangSmith, 145

LCEL (LangChain Expression Language), 81, 91, 92–93. *See also* LangChain

learning

- deep, 3, 4
- federated, 143
- few-shot, 37
- prompt, 25
- rate multiplier hyperparameter, 56
- reinforcement, 10–11
- self-supervised, 4
- semi-supervised, 4
- supervised, 4
- unsupervised, 4

LeCun, Yann, 5–6

Leibniz, Gottfried, 2

LLM (large language model), 1–2, 4–5, 6. *See also* agent; framework/s; OpenAI; prompt/ing

- abuse filtering, 133–134
- autoregressive language modeling, 5–6
- autoregressive prediction, 11
- BERT (Bidirectional Encoder Representation from Transformers), 5
- business use cases, 14
- chain, 52–53. *See also* chain
- ChatGPT, 19
- CLM (causal language modeling), 6
- completion call, 52

- connecting to data, 64–65. *See also* embeddings; semantic search
- content filtering, 133–134, 148–149
- contextualized response, 64
- current developments, 19–20
- customer care chatbot assistant
- embeddings, 9
- evaluation, 134–135, 144–145
- fine-tuning, 12, 54–55
- function calling, 56–57
- future of, 20–21
- hallucination, 11, 134, 156–157
- history, 2
- inference, 11
- inherent limitations, 21–22
- limitations, 48–49
- memory, 82–83
- MLM (masked language modeling), 6
- plug-in, 12
- privacy, 140, 142
- as-a-product, 15–16
- prompt/ing, 2, 25
- RAG (retrieval augmented generation), 72, 73
- red team testing, 132–133
- responsible use, 131–132
- results caching, 87
- security, 135–137. *See also* security
- seed mode, 25
- self-reflection, 12
- Seq2Seq (sequence-to-sequence), 6, 7
- speed of adoption, 21
- stack, 18
- stuff approach, 74
- tokens and tokenization, 7–8
- topology, 16
- training
- transformer architecture, 5, 7, 10
- translation, from natural language to SQL, 47
- word embedding, 5
- Word2Vec model, 5
- zero-shot prompting, iterative refining, 36

loader, LangChain, 105

logging

- agent, 53–54
- token consumption, 84–86

logit bias, 149–150

LSTM (long short-term memory), 4

M

max tokens, 30–31

measuring, similarity, 67

memory. *See also* vector store

- chain, 93–94
- long short-term, 4
- ReAct, 102–104
- retriever, 83–84
- short-term, 82–83
- SK (Semantic Kernel), 116–117

metadata

- filtering, 71, 108
- querying, 108

metaprompt, 44–45

method

- ChatStreaming, 175–176
- EventSource, 174
- get_format_instructions, 94

method

- GptConversationalEngine, 170–171
- HandleStreamingMessage, 175
- parse, 95
- parse_without_prompt, 95
- TextMemorySkill, 117
- Translate, 170
- metric-based evaluation, 146
- Microsoft, 19
 - Guidance. *See* Guidance
 - Presidio, 144
 - Responsible AI, 132
- Minimal API, hotel reservation chatbot, 205–208
- ML (machine learning), 3
 - classifier, 150–151
 - embeddings, 65
 - multimodal model, 13
- MLM (masked language modeling), 6
- MMR (maximum marginal relevance) algorithm, 71, 107
- model/s, 89. *See also* LLM (large language model)
 - chat, 89
 - embedding, 65
 - evaluating, 134–135
 - fine-tuning, 54–55
 - Guidance, 121
 - multimodal, 13
 - reward, 10
 - small language, 19
 - text completion, 89
- moderation, 148–149
- module
 - Embeddings, 106
 - LangChain, 88
- MRKL (Modular Reasoning, Knowledge and Language), 119
- multimodal model, 13
- MultiQueryRetriever, 196

N

- natural language, 15
 - embeddings, dimensionality, 65
 - as presentation layer, 15
 - prompt engineering, 15–16
 - RAG (retrieval augmented generation), 72, 73
 - translation to SQL, 47
- Natural Language to SQL (NL₂SQL), 118
- NeMo. *See* NVIDIA NeMo Guardrails
- network, generative adversarial, 4
- neural database, 67
- neural network, 3, 4, 67
 - convolutional, 4
 - deep, 4
 - recurrent, 4
- n-gram, 3
- NIST (National Institute of Standards and Technology), AI Risk Management Framework, 132
- NLG (natural language generation), 2
- NLP (natural language processing), 2, 3
 - BERT (Bidirectional Encoder Representation from Transformers), 3
 - Candide system, 3
 - data-driven approaches, 3
 - deep learning, 3
 - GPT (Generative Pretrained Transformer), 3
 - history, 3

- NLU (natural language understanding), 2
- number of epochs hyperparameter, 56
- NVIDIA NeMo Guardrails, 153–156

O

- obfuscation, 136
- OpenAI, 8
 - API
 - Assistants API, 80
 - CLI (command-line interface), 55
 - DALL-E, 13
 - Evals framework, 147
 - function calling, 60–63
 - GPT series, 5, 15, 16–17
 - Python SDK v.1.6.0, 34
 - OpenAPI, 208–209
- orchestration, need for, 79–80
- output
 - monitoring, 139
 - multimodal, 13
 - parsing, 94–95
 - structuring, 125–127

P

- PALChain, 91–92
- parse method, 95
- parse_without_prompt method, 95
- payload splitting, 136
- Penn Treebank, 3
- perceptron, 4
- performance, 108
 - evaluation, 144–145
 - prompt, 37–38
- PII
 - detection, 143–144
 - regulatory compliance, 140–141
- pip install doctran command, 106
- planner, 116, 119–120. *See also* agent
- plug-in, 111, 138
 - core, 115–116
 - LLM (large language model), 12
 - OpenAI, 111
 - TextMemorySkill, 117
- predictive AI, 3–4
- preparation, dataset, 55
- presence penalty, 29–30
- privacy, 140, 142
 - differential, 143
 - regulations, 140–141
 - remediation strategies, 143–144
 - at rest, 142
 - retrieval augmented generation, 199
 - in transit, 141–142
- programming, conversational, 1, 14
- prompt/ing, 2, 25
 - adjusting, 29
 - analogical, 44
 - basic, 26
 - chain-of-thought, 41
 - collecting information, 45–46
 - complex, 26–27

- descriptiveness, 27
 - engineering, 6, 10–11, 15–16, 25, 51–52
 - few-shot, 37, 90–91
 - frequency penalty, 29–30
 - general rules and tips, 27–28
 - hyperparameters, 28
 - injection, 136–137
 - instructions, 27, 34–35
 - iterative refining, 36
 - leaking, 136
 - learning, 25
 - logging, 138
 - loss weight hyperparameter, 56
 - max tokens, 30–31
 - meta, 44–45
 - order of information, 27–28
 - performance, 37–38
 - presence penalty, 29–30
 - ReAct, 99, 103–104
 - reusability, 87–88
 - SK (Semantic Kernel), 112–114
 - specificity, 27
 - stop sequence, 30–31
 - summarization and transformation, 46
 - supporting content, 34–35
 - temperature, 28, 42
 - template, 58, 80–81, 88, 90
 - top_p sampling, 28–29
 - tree of thoughts, 44
 - zero-shot, 35, 102
 - zero-shot chain-of-thought, 43
- proof-of-work, 109
- publicly available data, 140
- Python, 86. *See also* Streamlit
 - data providers, 83
 - LangChain, 57
 - setting things up, 34

Q-R

- quantization, 11
- query, parameterized, 139
- RA-DIT (Retrieval-Augmented Dual Instruction Tuning), 199
- RAG (retrieval augmented generation), 65, 72, 73, 109
 - C# code, 74
 - versus fine-tuning, 198–199
 - handling follow-up questions, 76
 - issues and mitigations, 76
 - privacy issues, 199
 - proof-of-work, 109
 - read-retrieve-read, 76
 - refining, 74–76
 - workflow, 72
- ranking, 66
- ReAct, 97–98
 - agent, 101
 - chain, 100
 - custom tools, 99–100
 - editing the base prompt, 101
 - memory, 102–104
- read-retrieve-read, 76
- reasoning, 97–99
- red teaming, 132–133
- regulatory compliance, PII and, 140–141
- relational database, 69

- Responsible AI, 131–132
- REST API, 159
- results caching, 87
- retriever, 83–84
- reusability, prompt, 87–88
- reward modeling, 10
- RLHF (reinforcement learning from human feedback), 10–11
- RNN (recurrent neural network), 4
- role-based chat, Guidance, 125–127
- rules, prompt, 27

S

- script, Colang, 155
- search
 - ranking, 66
 - semantic, 66–67
- security, 135–136. *See also* guardrail; privacy
 - agent, 137–138
 - attack mitigation strategies, 138–139
 - content filtering, 133–134
 - function calling, 64
 - prompt injection, 136–137
 - red team testing, 132–133
- seed mode, 25
- self-attention processing, 7
- SelfQueryRetriever, 197
- self-reflection, 12
- self-supervised learning, 4
- Semantic Kernel, 82
- semantic search, 66–67
 - chunking, 68–69
 - measuring similarity, 67
 - neural database, 67
 - potential issues, 68–69
 - TF-IDF (term frequency-inverse document frequency), 66–67
 - use cases, 67–68
- semi-autonomous agent, 80
- semi-supervised learning, 4
- Seq2Seq (sequence-to-sequence), 6, 7
- SFT (supervised fine-tuning), 10
- short-term memory, 82–83
- shot, 37
- similarity, measuring, 67
- singularity, 19
- SK (Semantic Kernel), 109–110. *See also* hotel reservation
- chatbot, building
 - inline configuration, 112–113
 - kernel configuration, 111–112
 - memory, 116–117
 - native functions, 114–115
 - OpenAI plug-in, 111
 - planner, 119–120
 - semantic function, 117
 - semantic or prompt functions, 112–114
 - SQL, 117–118
 - Stepwise Planner, 205–206
 - telemetry, 85
 - token consumption tracking, 84–86
 - unstructured data ingestion, 118–119
- Skyflow Data Privacy Vault, 144
- SLM (small language model), 19
- Software 3.0, 14
- specificity, prompt, 27
- speech recognition, 3

SQL
 accessing within SK, 117–118
 translating natural language to, 47
 SSE (server-sent event), 173–174, 177
 stack, LLM, 18. *See also* tech stack
 statistical language model, 3
 stop sequence, 30–31
 Streamlit, 182–183. *See also* corporate chatbot, building
 chatbot, building
 pros and cons in production, 185–186
 UI, 183–185
 stuff approach, 74
 supervised learning, 4
 supporting content, 34–35
 SVM (support vector machine), 67
 syntax
 Colang, 153
 Guidance, 122–123
 system message, 44–45, 122–123

T

“talking to data”, 64
 TaskRabbit, 137
 tech stack
 corporate chatbot, 182
 customer care chatbot assistant, 160–161
 technological singularity, 19
 telemetry, SK (Semantic Kernel), 85
 temperature, 28, 42
 template
 chain, 81
 Guidance, 125
 prompt, 58, 80–81, 88, 90
 testing, red team, 132–133
 text completion model, 89
 text splitters, 105–106
 TextMemorySkill method, 117
 TF-IDF (term frequency-inverse document frequency), 66–67
 token/ization, 7–8
 healing, 124–125
 logit bias, 149–150
 tracing and logging, 84–86
 training and, 10
 toolkit, LangChain, 96
 tools, 57, 81, 116. *See also* function, calling
 top_p sampling, 28–29
 topology, GPT-3/GPT-4, 16
 tracing server, LangChain, 85–86
 training
 adversarial, 4
 bias, 135
 initial training on crawl data, 10
 RLHF (reinforcement learning from human feedback), 10–11
 SFT (supervised fine-tuning), 10

transformer architecture, 7, 9
 LLM (large language model), 5
 from natural language to SQL, 47
 reward modeling, 10
 Translate method, 170
 translation
 app, 48
 chatbot user message, 168–172
 doctran, 106
 natural language to SQL, 47
 tree of thoughts, 44
 Turing, Alan, “Computing Machinery and Intelligence”, 2

U

UI (user interface)
 conversational, 14
 customer care chatbot, 164–165
 Streamlit, 183–185
 undesirable content, 148–149
 unstructured data ingestion, SK (Semantic Kernel), 118–119
 unsupervised learning, 4
 use cases, LLM (large language model), 14

V

vector store, 69, 108–109
 basic approach, 70
 commercial and open-source solutions, 70–71
 corporate chatbot, 190–191
 improving retrieval, 71–72
 LangChain, 106–107
 vertical big tech, 19
 virtualization, 136
 VolatileMemoryStore, 82, 83–84
 von Neumann, John, 22

W

web server, tracing, 85–86
 word embeddings, 65
 Word2Vec model, 5, 9
 WordNet, 3

X-Y-Z

zero-shot prompting, 35, 102
 basic theory, 35
 chain-of-thought, 43
 examples, 35–36
 iterative refining, 36