PROGRAMMING GPTS

A practical guide to building AI agents with OpenAI

V1.1 – November 2024

https://github.com/yanndebray/programming-GPTs

To my wife, family and friends
They give me the courage to change what can be changed

TABLE OF CONTENTS

PREFACE

"The proper study of man is anything but himself, and it is through the study of language that we learn about our own nature." – J.R.R. Tolkien

Language is a foundation for humanity. It is through language that we have elevated ourselves, from the silence of nature. First we existed, then our essence was granted by a Voice. We shared with one another. We started telling each other stories, sometimes simply relaying information, sometime making things up. We elaborated ideas, started reasoning about them, turning them into plans. A sound became words – letters, semantics, lexicons – words transformed into writings, writing into books, books into cultures, cultures into civilizations. Leading us all the way to the current digital age, with the inter-connected-net of articles, blogs, images, videos. All this content that started to feed machines, eager to learn.

Ever since the release of an apparently harmless chatbot application, we have observed an acceleration in the development of language modeling. Artificial intelligence has marked a new epoch in our relationship with language. These models, built on the crunching of numbers by the largest supercomputers in the world, have enabled computers to understand and generate human language with unprecedented accuracy. OpenAI's Generative Pre-trained Transformers are at the forefront of this revolution, demonstrating the extraordinary potential of AI to augment human capabilities.

This book is designed to equip you with the knowledge and skills needed to harness the power of GPTs, opening new horizons in the field of artificial intelligence. Whether you are a seasoned developer or a curious enthusiast, this practical guide will help you navigate the complexities of building AI agents, transforming your ideas into reality.

WHO IS THIS BOOK FOR

Ever heard of FOMO (Fear Of Missing Out)? I'll try not to use FUD (Fear Uncertainty and Doubt) to sell you on GPTs. I'll also try really hard not to use other TLAs (Three Letter Acronyms) in this book. Ok, I guess I'll use one mainly: GPT. Wondering what this means, and how it could be relevant to your work? Then you might find what you need in this book.

I am assuming that you are *NOT an AI expert* (quite the opposite actually). If you already have some notions of what machine learning means, those might prove useful as we go deeper into describing what Generative AI is and how it differs from the previous waves of AI.

You are probably *tech savvy*, with likely some background in scientific or technical studies. Don't worry, I won't use gory mathematical equations to explain the concepts manipulated in the book. Instead, I'll use concrete example of simple applications you can develop and tailor to your needs.

I'll assume some *basic programming skills*, but no necessary experience with AI libraries like scikit-learn or pytorch. I would be good for you to have some basics of numeric, with libraries like numpy and pandas. Those should be easy to acquire. Overall you should have some appetite for coding, as it will make this experience much more enjoyable and give you a deeper understanding of the concepts.

ABOUT THE AUTHOR

I graduated from college with a degree in mechanical engineering. Some of the course that I really got into involved numerical analysis with Maple and Mathematica. But it's only after my studies, that I started to learn about data science and machine learning. This was 2013 and the rise of online courses, so like many, I followed the Stanford course from Andrew Ng on Coursera[1].

I discovered myself a passion about technical computing, and joined a French start-up called Scilab. They were spinning off from the French research to build a consulting business around open-source software. It ended up being an exciting but challenging journey, with some new cloud products roll out and an acquisition a few years later. As I was learning more about the ways of open-source, I encountered another programming language, Python, that I fell in love with.

Then in the beginning of 2020, as the world was starting to lock down, I got a call from a company in Boston to work on the famous software MATLAB, the leader in technical computing. A few years later, here I am, more passionate than ever about numeric and eager to learn about its new forms, mostly called AI now. And the best way to learn is to teach. So, buckle up, and let's go for the ride!

[1] https://www.coursera.org/specializations/machine-learning-introduction

WHAT YOU WILL LEARN

The book is divided into 10 chapters, each covering a different topic and a different aspect of programming GPTs. The chapters are:

- *Chapter 1: Introducing GPTs***.** How they work and their evolution.
- *Chapter 2: OpenAI APIs.* In this chapter, you will learn how to use the OpenAI APIs, a simple way to create AI agents with GPT Models. You will start by learning how to create your own chatbot.
- *Chapter 3: Prompting, Chaining & Summarization.* In this chapter, you will learn how to engineer prompts, chain calls to a Large Language Model and use it to summarize texts, such as articles, books, or transcripts. You will use the OpenAI API together with the LangChain package to enhance GPTs.
- *Chapter 4: Vector search & Question Answering.* In this chapter, you will learn how to use embeddings and vector search as a way to retrieve informative answers to answer questions while quoting sources.
- *Chapter 5: Agents & Tools.* In this chapter, you will learn to build an Agent, called Smith, that has access to tools, such as getting the current weather. You will also learn how to use the Assistant API provided by OpenAI, and to extend their capabilities with tools to integrate GPTs with external services. This will be illustrated with the implementation of your own Code Interpreter, that can help you write and run code with GPTs.
- *Chapter 6: Speech-to-Text & Text-to-Speech.* In this chapter, you will learn how to use GPTs to transcript text from speech (such as Youtube videos) and synthetize speech from text (such as articles).
- *Chapter 7: Vision.* In this chapter, you will learn how to use GPTs to process and analyze images, such as mock-ups or drawings. You will learn how to use the Vision API, to perform various tasks with GPTs, such as text recognition, or video captioning.
- *Chapter 8: Dall-E image generation.* In this chapter, you will learn how to use Dall-E 2 & 3, which can create stunning and creative images from any text input. You will also learn how to use the outpainting, inpainting and variations APIs, which can complete or modify existing images.
- *Chapter 9: Deploying GPTs.* In this chapter, you will reuse what you have learned, and you will deploy your own GPTs. This will be made concrete with a small team of GPTs covering each one chapter of the book.
- *Chapter 10: Appendix.* In this chapter, you will find additional resources to keep up with the latest developments and innovations in the field of GPTs.

By the end of this book, you will have a solid understanding of how to program GPTs with OpenAI, and how to create amazing and useful applications with them. You will also have a deeper appreciation of the power and possibilities of GPTs, and how they can transform the world of natural language and beyond.

Disclaimer: I have used ChatGPT to help me write this book. But I would argue that the person who does not leverage AI to do her work is going to be left behind. Or said in a more politically correct tone: "You won't be replaced by AI, but you'll be replaced by someone using AI". To read more about the potential of AI as an assistant, I'd recommend reading "Impromptu".[2]

Setting up the Programming Environment

Setting up an effective programming environment is crucial for working with Generative Pre-trained Transformers (GPTs). This section will guide you through the necessary tools, libraries, and hardware requirements, followed by a step-by-step setup process.

Initial Setup Guide

I'll take as assumption that you are running on a Windows machine. Wherever there is a major difference in OS, I'll try to make sure that I give explanations for the different platforms.

- Install Python

Ensure that Python (version 3.6 or later) is installed on your system. You can download it from the official Python website[3].

- Set Up a Virtual Environment (optional)

This is a good habit if you don't want to mess up completely your Python dev environment. This book will not leverage a lot of Python packages, so you might get away without it.

Using a virtual environment is recommended to manage dependencies. You can create a virtual environment using tools like *venv* or *conda*.

```
python -m venv env

# Activate the environment

env\Scripts\activate
```

[2] https://www.impromptubook.com/
[3] www.python.org

```
# On Linux or Mac: source env/bin/activate
```

- Install Packages

Here are a few python packages[4] that will be used throughout the book:

```
pip install openai langchain transformers streamlit
```

OpenAI-Python[5] is the official client provided by OpenAI for interacting with their GPT models. *LangChain*[6] is useful for chaining language model calls and building complex applications. *Transformers*[7] is a popular library for working with open-source AI models. *Streamlit*[8] is a Python framework for creating and sharing beautiful, custom web apps.

- API Keys and Authentication (For OpenAI)

Obtain an API key from OpenAI by registering on their platform. Set up authentication by adding your API key to your environment variables or directly in your code (not recommended for production).

- Test Installation

Verify your setup by running a small script to interact with the OpenAI API. As of version 1.X of the OpenAI Python SDK, your code would look like this:

```
from openai import OpenAI
client = OpenAI(api_key = 'sk-XXXX')
res = client.completions.create(
  model="gpt-3.5-turbo-instruct",
  prompt="Say this is a test",
)
print(res.choices[0].text)
```

Run the script including the API key obtained in the previous step:

```
$ python openai_setup.py
```

- Streamlit Basic Setup

[4] openai >= 1.0 langchain >= 2.0 streamlit >= 1.34
[5] https://github.com/openai/openai-python
[6] https://www.langchain.com/
[7] https://huggingface.co/docs/transformers
[8] https://streamlit.io/

Create a simple Streamlit app to test the setup:

```
import streamlit as st
st.title('GPT with Streamlit')
user_input = st.text_input("Enter some text")
if user_input:
    st.write("Your input was:", user_input)
```

Run the Streamlit app with the following command (not simply executing the script with Python):

```
$ streamlit run streamlit_setup.py
```

SETUP THE DEV ENVIRONMENT

I will be using VSCode as my Integrated Development Environment (IDE) for this book. It is a popular coding editor that provides a rich set of features for Python development, such as syntax highlighting, code completion, debugging, testing, and version control. VSCode is also highly extensible, with a large number of extensions available in the marketplace. Feel free to use your preferred IDE or text editor if you are more comfortable with it.

To setup VSCode for a Python project, you will need to install some extensions and configure some settings. Here are some extensions recommended for VSCode to apply the learnings of this book:

- Install the Python extension from the VSCode marketplace. This will enable syntax highlighting, code completion, debugging, testing, and other features for Python files.
- Install the Jupyter extension from the VSCode marketplace. This will allow you to create and edit Jupyter notebooks with the *.ipynb* extension in VSCode, as well as view and interact with the outputs of your code cells. You can also execute sections of a *.py* file interactively in a separate panel by run Python code in a Jupyter kernel and see the results inline. To create a Jupyter section in a .py file in VSCode, you can use the *#%%* delimiter. To run notebooks or cells in a Jupyter kernel, you will need to install the *ipykernel* python package. If you haven't installed it manually, a dialog will pop up the first time to install it for you

Here are some advice to get you started with Python in VSCode:

- Create a folder for your project and open it in VSCode. You can use the *File > Open Folder* menu or the command palette (*Ctrl+Shift+P*) and type **Open Folder**.
- Configure VSCode to use the correct Python interpreter and linter. You can use the status bar at the bottom of the editor to select the interpreter, as shown in Figure 0-1. Alternatively, you can use the command palette and type **Python: Select Interpreter**.

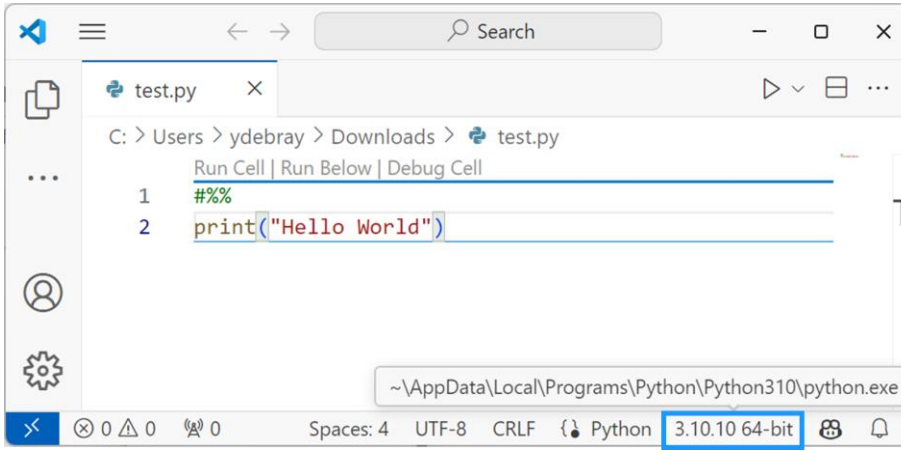


Figure 0-1 Select Python interpreter in VSCode

If you don't want to maintain your dev environment locally, you can get started with a cloud development environment, like GitHub Codespace. This works really well with Streamlit, as a full online dev environment.[9]

Develop and deploy a web app with Streamlit

Streamlit is a Python framework that lets you create beautiful and interactive web apps in minutes, without any web development skills. You can write your app logic in pure Python, and Streamlit will handle the front-end for you. Streamlit apps are composed of widgets, charts, maps, tables, media, and custom components that you can arrange and update dynamically with a few lines of code. Streamlit also provides a live reloading feature that automatically refreshes your app whenever you change your code or data.

[9] https://blog.streamlit.io/edit-inbrowser-with-github-codespaces/

Streamlit works by running a local web server that serves your app to your browser. You can either run your app on your own machine, or deploy it to a cloud platform like Heroku, AWS or the Streamlit Cloud. Streamlit apps are based on a simple concept: every time the user interacts with a widget, such as a slider or a button, Streamlit reruns your Python script from top to bottom, and updates the app accordingly[10]. This means that you don't have to worry about callbacks, state management, or HTML templates. You just write your app logic as a normal Python script, and Streamlit will take care of the rest.

[10] https://docs.streamlit.io/library/get-started/main-concepts#app-model

1. INTRODUCING GPTS

Let's start with a bit of theory. If you're reading this book, I can assume you have a basic math and science education, and some programming experience. I don't want to go too deeply into the different dimensions of large language models (LLMs), generative AI (genAI), and artificial intelligence (AI) either. My goal is to provide enough necessary background so that you can apply AI to your concrete applications. Figure 1-1 is a simple illustration of the relationships among these three concepts.

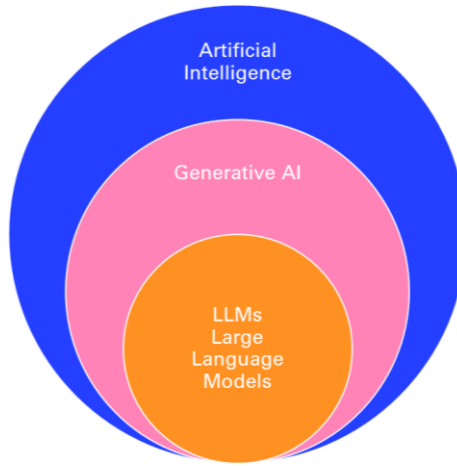


Figure 1-1 Levels within AI

LARGE LANGUAGE MODELS ARE NEXT-WORD PREDICTORS

Large language models (LLM) are neural networks - mimicking the structure of the human brain - that can process and generate natural language texts based on a given input, such as a word, a phrase, or a prompt. LLMs operate word by word, based on the probability of each word being the most likely next word, kind of like the autocomplete function on your phone. Figure 1-2 shows an example of how an LLM might predict (output) the next word given an input of words.

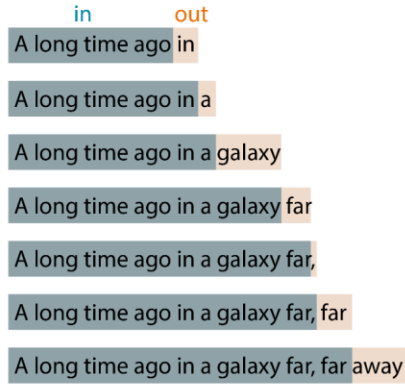


Figure 1-2 Next word prediction

Given the input *A long time ago*, an LLM might generate the output *in a galaxy far, far away*. It works like this:

1. To generate the first word *in*, it uses only the input *A long time ago*.
2. To generate the second word *a*, it uses the input plus the first word it just generated: *A long time ago + in*.
3. To generate the third word, it uses the input plus the first word and the second word, and so on.

By working this way, after being trained on many texts and learning from the patterns and structures of natural language, an LLM can generate coherent and fluent texts of its own. This process is based on statistics and probabilities, as were former generations of AI models.

This kind of model is sometimes referred to as autoregressive. *Autoregressive* is a term that describes a model that predicts the next output based on the previous inputs as well as subsequent outputs from each iteration. Such a model assumes that the current value of a variable depends on its past values. It's a little bit like how you might be able to guess the next word in a sentence if someone paused while speaking. Autoregressive models are widely used for applications other than text generation, such as time series forecasting and speech synthesis.

An LLM can also perform various natural language understanding tasks, such as answering questions, summarizing texts, or extracting information. LLMs can be seen as general-purpose language engines that can handle multiple tasks and applications with minimal supervision. As you'll learn later in this chapter, GPT models are only one kind of LLM, developed by OpenAI in 2018. Since then, other prominent commercial and open-source alternatives have been released, such as the Llama models from Meta, Gemini from Google, Claude from Anthropic, the Mistral models, and others.

/! LLMs are not by nature assistants. If you prompt them with the beginning of a sentence, they will by default try to complete the sentence. They won't try to make sense of the sentence or answer it as if it were a question. This is a fundamental shift that ChatGPT introduced by fine-tuning OpenAI's GPT 3 model to perform question answering in the form of a chatbot.

LLMs are trained on massive amounts of text data, usually *trillions* of words from the internet (hence the term *large*), and from that training they learn to capture the statistical patterns and structures of natural languages in their parameters (generally in *billions*). You will discover in the next sections how this training process works. One way to think about LLMs is to consider them as a compression of internet knowledge. If you look at a model with 70 billion parameters (each represented as two bytes) and download it to your machine, you would have a 140 GB file (including the parameters). This represents a compression ratio of roughly 100:1. For state-of-the-art models like GPT-4, those numbers are off by a factor of 10 at least. Figure 1-3 tries to capture this idea.

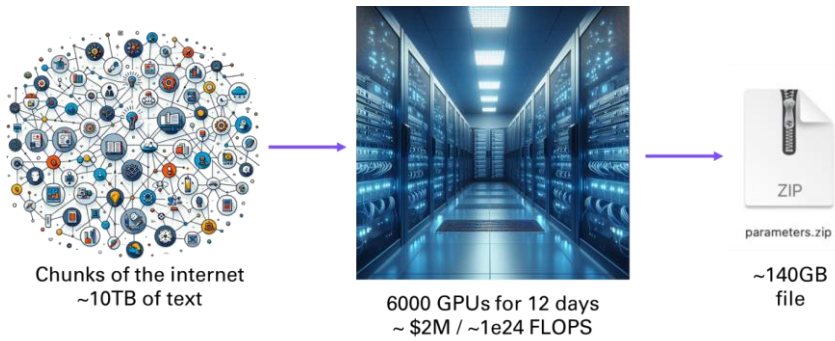


Figure 1-3 LLMs can be said to compress the Internet

LLMs kept growing larger and larger. Something quite remarkable and somewhat unexpected happened when LLMs reached a certain scale: some new properties started to emerge. I'll talk more about the characteristics of these emergent properties when we look at the architecture of GPT models.

Table 1-1 should give you a sense of how LLMs have grown larger over time, using the GPT family of models to illustrate.

Table 1-1 Size of the different generations of GPT models

Model	Dataset size (billions of tokens)	Model size (billions of parameters)
GPT 1	1-2	0.11
GPT 2	10-20	1.4
GPT 3	300	175
GPT 4	10,000	??? (est. 1,800)

GENERATIVE PRETRAINED TRANSFORMERS: BUILDING BLOCKS OF LANGUAGE INTELLIGENCE

GPT stands for generative pre-trained transformer. The term refers to a family of LLMs created by the company OpenAI. Like all LLMS, GPTs are trained on massive amounts of text data from the web, such as Wikipedia articles, news stories, blog posts, books, and more. They learn to capture the patterns and structures of natural language, such as grammar, syntax, semantics, and style.

However, and perhaps confusingly, the term *GPTs* (plural) now refers to more than just a model—it refers to AI agents. An *agent* is a software program that can interact with its environment and perform specific tasks, such as answering questions, writing summaries, or composing emails. A GPT agent is built on top of a GPT model, using its language generation capabilities to produce responses to user requests. For example, a GPT agent can act as a chatbot, a search engine, a content writer, or a personal assistant.

In this book, you will explore the world of GPTs, both as models and as agents. You will learn how they work, how they are trained, and how they are used in various

applications. You will also learn how to build your own GPT agents, using the powerful and easy-to-use OpenAI GPT APIs. These APIs allow you to access the state-of-the-art GPT models, such as GPT-4, without having to worry about the technical details of running and maintaining them. You will also learn how to enhance the capabilities of GPT models by granting the GPT agents access to tools.

But first let's look a little more closely at the three words contained in the term *GPT*.

GENERATIVE

Generative models can create new data from existing data. For example, given an image, a generative model can produce another image that is similar but not identical to the original one. Similarly, given a text, a language model can produce another text that is related but not identical to the original one. Since GPT-4, the line between language models and other kinds of generative models is blurring, as GPT-4 supports text as well as image and sound (both as input and output).

Going back to language models, here is an example of text generation with the first GPT model that was released (now called GPT-1):

```
 from transformers import AutoTokenizer, OpenAIGPTLMHeadModel

 tokenizer = AutoTokenizer.from_pretrained("openai-community/openai-
gpt")
 model = OpenAIGPTLMHeadModel.from_pretrained("openai-
community/openai-gpt")

 input_ids = tokenizer.encode("A long time ago in a galaxy far",
return_tensors="pt")
 outputs = model.generate(input_ids, max_length=42, do_sample=True)
 text_generated = tokenizer.decode(outputs[0])
 print(text_generated)
```

```
a long time ago in a galaxy far off, we were working on the same
quantum project, the same concept of the universe where we were
supposed to go. i think there was a long period of time
```

/! With sampling (`do_sample=True`), the model doesn't always pick the most likely token. Instead, it randomly *samples* from the distribution of possible next tokens according to their probabilities. Sampling introduces an element of randomness, allowing for more diverse and creative outputs.

To get a sense of how the first GPT model works, you can use the Write With Transformer app available at *transformer.huggingface.co*. Simply select the model you would like to take for a spin, and you will be able to type in a prompt and see the model generate text based on it, as shown in Figure 1-4.

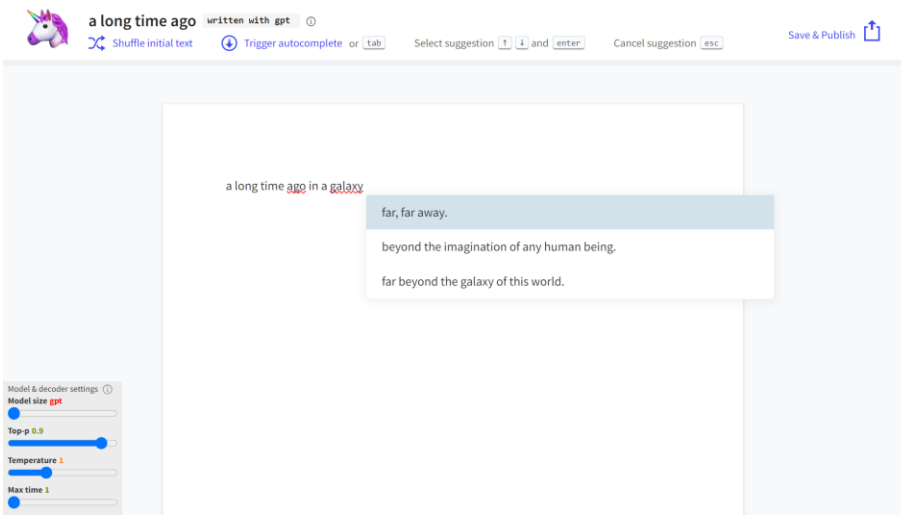


Figure 1-4 Write With Transformer example

PRE-TRAINED

GPT models have been *pre-trained* on massive amounts of text from the internet and can produce coherent and diverse responses on almost any topic. They can also perform various tasks, such as answering questions, summarizing texts, generating images, and more.

This process of pre-training involves two main stages:

1. Pre-training a base model
2. Fine-tuning an assistant model

Table 1-2 summarizes the GPT training process.

Table 1-2 GPT training pipeline

	Pre-training	Fine-tuning
Dataset	Raw internet: Trillions of words. Low quality, large quantity	Demonstrations: ~10-100K Ideal assistant responses (prompt, response) written by contractors. High quality, low quantity
Model	Base model	Assistant model
Method	Self-supervised next word prediction	Supervised on manually labeled data
Compute	1000s of GPUs, months of training	1-100 GPUs, days of training

Language contains impressive inherent properties, such as structure and syntax, context and semantics, intrinsic hierarchical information, predictability and sheer richness in the data. The big breakthrough enabled by generative models was due to the fact that they could learn language constructs by *self-supervision*. Traditional supervised learning requires a curated set of inputs and outputs. Generative models on the other end simply train their parameters by predicting the last word of a sequence with a sliding window, as shown in Figure 1-5.

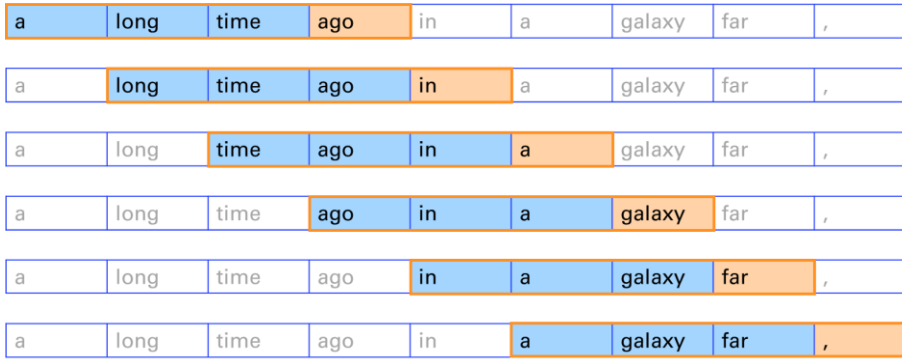


Figure 1-5 Self-supervision with a sliding window

The emergence of properties that I mentioned appeared with the scaling up of the size of the neural network and the dataset on which it was trained. As Aristotle said, "The whole is greater than the sum of its parts." This topic raises the question of *artificial general intelligence* (AGI), a hypothetical type of AI that would possess the ability to perform any intellectual task that a human can. Such a concept is well beyond the scope of this book. Here we will focus on more practical aspirations: to teach you how to leverage GPTs to assist you in your daily tasks.

TRANSFORMERS

GPT models use a special kind of neural network architecture called a *transformer*, introduced in 2017 in a paper by Ashish Vaswani et al called "Attention Is All You Need"[11]. Similarly to previous generations of neural networks, like convolutional or recurrent neural networks, transformers are composed of many *layers* of artificial neurons that are activated in a specific order during the training and prediction phases depending on their arrangement. For instance, recurrent neural networks (RNNs) maintain a memory of the previous state of information processed by the network, which is useful to predict future elements of a sequence. But as the name of the paper hints, the new network architecture introduced with the transformers departs from previous approaches to convert sequences to sequences, by focusing on a key ingredient, called the *attention mechanism*.

Attention provides contextual understanding, which enables the model to consider the meaning of each word in a sentence, regardless of its position. This means each word's encoding in the neural network is influenced by every other word in the sentence, allowing for a more nuanced understanding of language. The attention mechanism equips the model with an efficient method for processing text. Unlike RNNs where the words are processed sequentially, transformers process all words in a sentence simultaneously, leading to significant gains in computational efficiency. For example, given a sentence in English, transformers can learn to translate it to French. In Figure 1-6, you can observe the more verbose translation in French. The lighter yellow positions in this attention matrix capture the translations of the words (attention is the same word in French).

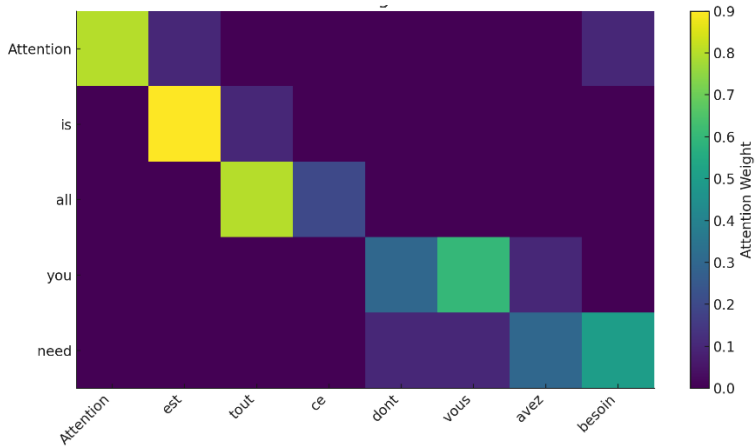


Figure 1-6 Simple translation from English to French

[11] Attention is all you need: https://arxiv.org/abs/1706.03762

The original transformer from the paper consists of two parts: an encoder and a decoder[12]. The *encoder* takes the input text and transforms it into a numerical representation, i.e. a series of numbers mapping each part of the sentence, capturing the meaning and context of the input. The *decoder* takes the numerical representation and generates the output text. This is particularly useful for translation.

GPT models, like most LLMs nowadays, are decoders-only because this type of network is particularly well-suited for text completion. By focusing solely on the decoder, GPT models can streamline their architecture to better specialize in understanding and generating language. Decoders-only architectures reduce complexity because they focus on one task (next-word prediction) and don't require the additional overhead of encoders. This makes them easier to scale and optimize for large datasets and extensive training without needing to balance encoder-decoder interactions. Since decoders-only models are trained autoregressively (predicting the next token based on previous ones), this setup naturally lends itself to highly parallelized and efficient training.

Transformers have become the dominant architecture for LLMs, and can be described as general-purpose machines, being expressive, optimizable, and efficient:

- *Expressive:* They can model complex relationships and patterns in data
- *Optimizable*: Their ability to handle complexity is not compromised by the fact that transformers can be trained on large amount of data
- *Efficient*: The training and prediction can be parallelized on graphics cards.

How GPT Models Work

This section examines how GPT models work in practice. If you want to go deeper into the inner workings of transformers, I recommend the book *Natural Language Processing with Transformers* by Lewis Tunstall, Leandro von Werra, and Thomas Wolf (O'Reilly, 2022). GPT models can operate under two modes: training or prediction. For each of those, the input text flows into the network architecture to predict the most probable next word.

In training mode, the prediction is compared with the actual word that was following in the sentence. If there is an error in the prediction, the correction is propagated back into the network to adjust the parameters of the model. This algorithm is called *backpropagation*, and it is the process that enables neural networks to "learn" from

[12] Encoder-decoder architecture: Overview https://www.youtube.com/watch?v=zbdong_h-x4

the training set. Many iterations are performed for this optimization of the parameters of the model, until the prediction rates are high enough on a validation set.

Evolution of GPTs

The development of Generative Pre-trained Transformers (GPTs) has been a revolutionary journey in the field of artificial intelligence and natural language processing. This section delves into the fascinating progression of GPT models, from their humble beginnings to the powerful language models we see today. We'll explore the key milestones, technological advancements, and the increasing capabilities that each iteration brought to the table. By understanding this evolution, we can better appreciate the current state of GPT technology and glimpse into its potential future applications.

Life BC (Before ChatGPT)

As I've mentioned, the first GPT model (later called GPT-1) was released in 2018 by OpenAI, along with a paper called "Improving Language Understanding by Generative Pre-training."[13] This first model had 117 million parameters, trained on a large corpus of text from the web, called WebText, which contained about 40 GB of data. As its parameters and architecture were released with an open-source license, it enabled researchers to fine-tune it to specific tasks (such as text classification and sentiment analysis) with relatively little new data, leveraging its pre-trained knowledge.

However, GPT-1 had some limitations. For example, it could not handle long-term dependencies, because of the small context window at the core of its relatively shallow attention layer. That meant it could not remember or use information that had appeared earlier in the text. It also struggled with factual consistency, which means that it could not verify or correct the information that it generated. Moreover, it sometimes produced offensive or biased texts, which reflected the quality and diversity of the data that it was trained on.

To address these issues, OpenAI released GPT-2 in 2019. The accompanying paper was titled "Language Models are Unsupervised Multitask Learners"[14] highlighting the emerging ability of this model to perform tasks on natural language such as question answering, without the need for supervised training on task-specific datasets. This new set of capabilities came from its 1.5 billion parameters, which was more than 10 times the size of GPT-1. It was also trained on a much larger corpus of text, called WebText2, which contained about 570 GB of data.

[13] https://openai.com/index/language-unsupervised/
[14] https://openai.com/index/better-language-models/

GPT-2 improved significantly on the performance and quality of GPT-1 and achieved state-of-the-art results on many natural language benchmarks. It also demonstrated a remarkable ability to generate coherent and engaging texts on a variety of topics and styles, such as news articles, essays, stories, and reviews. However, GPT-2 also raised some ethical and social concerns. Due to its high level of realism and versatility, GPT-2 could potentially be used for malicious purposes, such as spreading misinformation, impersonating others, or generating fake content. Therefore, OpenAI decided to release GPT-2 gradually, starting with a smaller version of 124 million parameters and then releasing larger versions over time, along with some tools and guidelines to help researchers and developers use GPT-2 responsibly and safely.

The big breakthrough came with GPT-3 and the appropriately titled paper "Language Models are Few-Shot Learners"[15], which was released in 2020 by OpenAI. With a staggering 175 billion parameters, more than 100 times the size of GPT-2, it was trained on an enormous corpus of text, called WebText3, which contained about 45 TB of data. One of the key features of GPT-3 was its ability to perform tasks with little to no task-specific training, known as *zero-shot* or *few-shot learning*. This means GPT-3 could understand and respond to prompts in a meaningful way, even if it hadn't been explicitly trained on that task.

GPT-3 was not only a powerful language generator, it was a general-purpose artificial intelligence system that could learn to perform any task that can be described in natural language. For example, given a prompt like *Write a summary of this article*, or *Create a slogan for this company*, or *Solve this math problem*, GPT-3 could generate appropriate and accurate responses by using its vast knowledge and understanding of language and the world. And to guide the model in the type of results you were expecting, you could provide one or several examples as part of the prompt.

ChatGPT and Alignment

GPT-3.5, also known as ChatGPT is an improved version of GPT-3 that focuses on enhancing its conversational skills. ChatGPT is designed to be a friendly and engaging chatbot that can chat with humans about any topic and provide relevant and interesting information as well as tell jokes. It closely follows a paper from early 2022 called "Training language models to follow instructions with human feedback"[16], introducing instructGPT and paving the way for the revolution that was to come…

[15] Language Models are Few-Shot Learners: https://arxiv.org/abs/2005.14165
[16] InstructGPT paper: https://openai.com/index/instruction-following/

ChatGPT is based on the same architecture and data as GPT-3, but with some key differences. ChatGPT aimed to fix what is called alignment. *Alignment* is the process of encoding human values and goals into large language models to make them as helpful, safe, and reliable as possible. Essentially, it builds in guardrails so that the AI doesn't embed the biases that can be found on the internet.

How did OpenAI achieve this? ChatGPT was initially meant only as a research project to demonstrate the potential of *reinforcement learning from human feedback* (RLHF). This involved leveraging some of the early work of OpenAI in the field of reinforcement learning that is often associated with robotics or games like chess, go, or Mario.

In traditional reinforcement learning, an agent learns to make decisions by interacting with an environment to maximize cumulative reward. The agent explores actions, observes results, and receives rewards or penalties, which guide future actions. In RLHF, human feedback supplements or replaces the predefined reward signals. Human evaluators observe the agent's actions and provide feedback on their quality or appropriateness. This feedback helps shape the reward function. Figure 1-7 illustrates the three steps of the RLHF process as implemented by OpenAI.

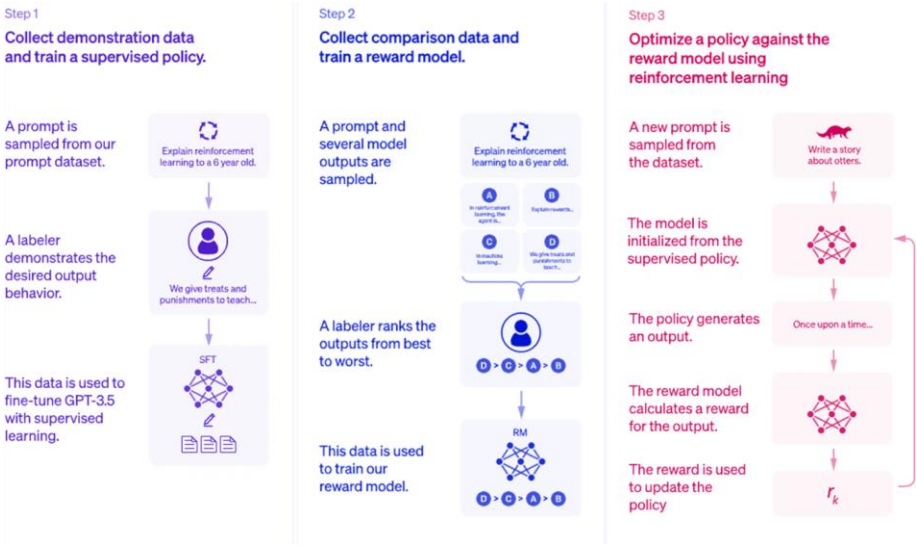


Figure 1-7 Humans are participating at different steps of the fine-tuning of ChatGPT

Here are the steps explained for the alignment of ChatGPT through reinforcement:

1. *Collect demonstration data and train a supervised policy:* Human feedback is used to label the training data. This is used to fine-tune the model.
2. *Collect comparison data and train a reward model:* Humans rate the output of the model from the previous step. This is captured in a reward function.
3. *Optimize a policy against the reward model:* This is now an iterative process without manual human input where the two previous steps are brought together to improve alignment.

GPT-4 AND BEYOND

In March 2023, OpenAI released GPT-4, marking another significant leap in the capabilities of large language models. GPT-4 introduced several key advancements.

GPT-4 demonstrates enhanced logical reasoning and problem-solving skills, performing better on complex tasks that require multi-step thinking. GPT-4 outperforms GPT-3.5 by scoring in higher approximate percentiles among test-takers as displayed in table 1.3 where we see the comparison of the performance of the two models.

Table 1-3 GPT-4 surpasses GPT-3.5 in its advanced reasoning capabilities

	GPT-3.5	**GPT-4**
Uniform Bar Exam	10th	90th
Biology Olympiad	31st	99th

Unlike its predecessors, GPT-4 can process both text and images as input. This allows it to analyze, describe, and reason about visual information in addition to text. This ability is called *multimodality*.

On top of this new modality, audio support came in May of 2024 and earned the model a new nickname, GPT-4o where “o” stands for “omni”

Finally, on a more practical note, GPT-4 can handle much longer inputs, with versions gradually able to process up first 8k tokens, then 32k tokens, to finally reach an astonishing 128k tokens in a single prompt. Later in the book, you will learn more about the capabilities of GPT-4.

In September 2024, a new series of models called o1 was introduced. Those models spend time *thinking* before answering to a question, making them more efficient in complex reasoning tasks, like science and programming. A good use case for this generation of models is to engineer a comprehensive prompt to prime the pump on the development of a new complete app.

Building Your Own GPT

OpenAI coined the term *GPTs* to extend their offerings beyond the established GPT models to GPT agents. This came as a response to the growing trend of developing custom ChatGPT implementations. This move essentially positioned OpenAI as a platform for developers to deploy AI agents. At the same time, they offered a low-code approach to building custom GPTs (by programming without coding), lowering the barrier of entry for non-coders.

This book describes different alternatives for programming GPT agents offering rich options and details (see Table 1-4). A strong ecosystem boomed in 2023 from this rich set of options around different opinionated approaches to tackle this challenge. Here is a high-level table that compares what can be done via the ChatGPT app vs the API.

Table 1-4 Developing GPT agents through low-code vs API integration approach

	Custom GPTs (via ChatGPT)	**Assistants (via the API)**
Creation Process	No code	Requires coding for integration
Operational Environment	Located in ChatGPT	Can be integrated into any product or service
Pricing	Included in ChatGPT on Plus/Team/Enterprise plans	Billed based on usage of different Assistant features
User Interface	Built-in UI with ChatGPT	Designed for programmatic use; can use playground for visualization
Shareability	Built-in ability to share	No built-in shareability

Some very clever programmers like Andrej Karpathy[17] have even gone so far as to build such elaborate AI models from scratch, or based on open-source implementations such as the Llama family of models (from Meta). This isn't the approach I take in this book, as it is way more involved and requires very deep AI and programming skills.

Instead, this book adopts an approach somewhere in-between the low-code and high-code options to build your own AI-powered applications. For this we will attempt to just get the right level of understanding of how those GPTs are operating by becoming familiar with the OpenAI application programming interface (API).

To explain the need for the API, you can consider the cases of using vs building AI-powered products. In the next chapter, you will learn how to use the OpenAI API.

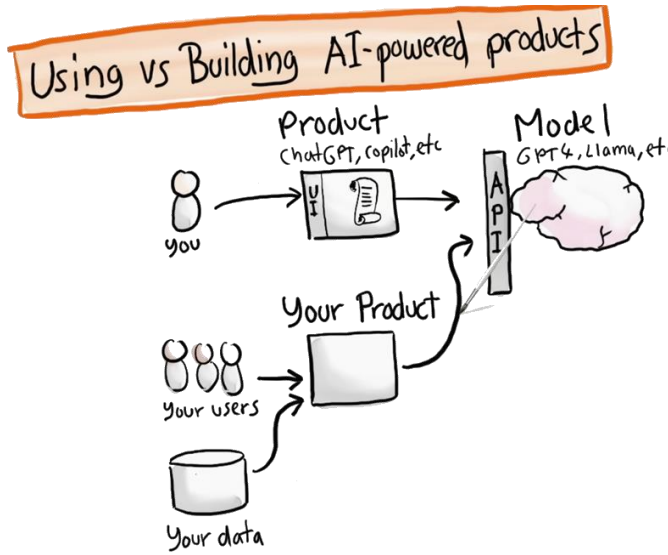


Figure 1-8 using vs building AI-powered products

[17] Let's build GPT: from scratch, in code, spelled out. https://www.youtube.com/watch?v=kCc8FmEb1nY

2. OpenAI APIs

In this chapter you will discover the main APIs that OpenAI provides to interact with their GPT models. You will learn how to develop and deploy your first chatbot app with Streamlit. As an application of your learning, you will build your very first chatbot. It will look like this:

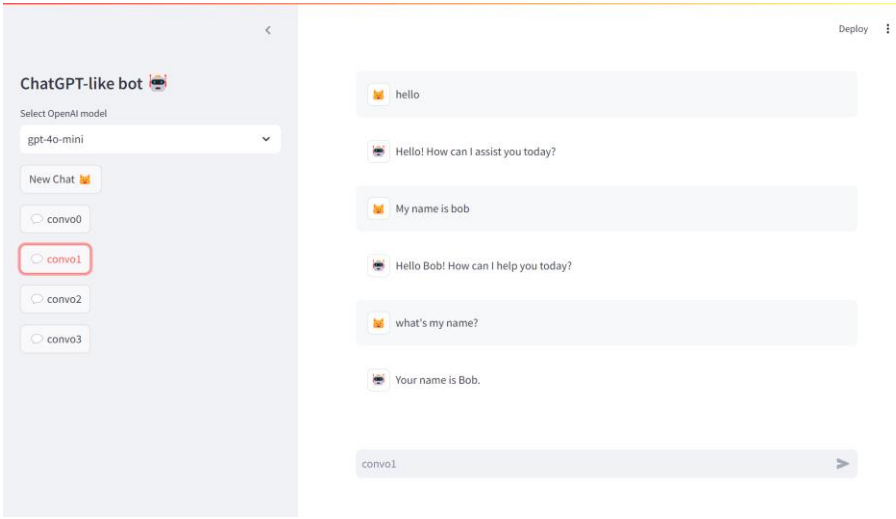


Figure 2-1 Your first chatbot app

Quick tour of the OpenAI developer platform

First you will need to create an OpenAI account on their developer platform (*platform.openai.com*). Once you are logged in to your OpenAI account, the landing page of the developer platform will take you to a playground that enables you to get access to different kinds of models. The main kind of models are chat models on which I will focus for this chapter. But you will see other kinds of models in the rest of the book, like the Assistant and TTS (Text-To-Speech) models. Completion models are now considered legacy.

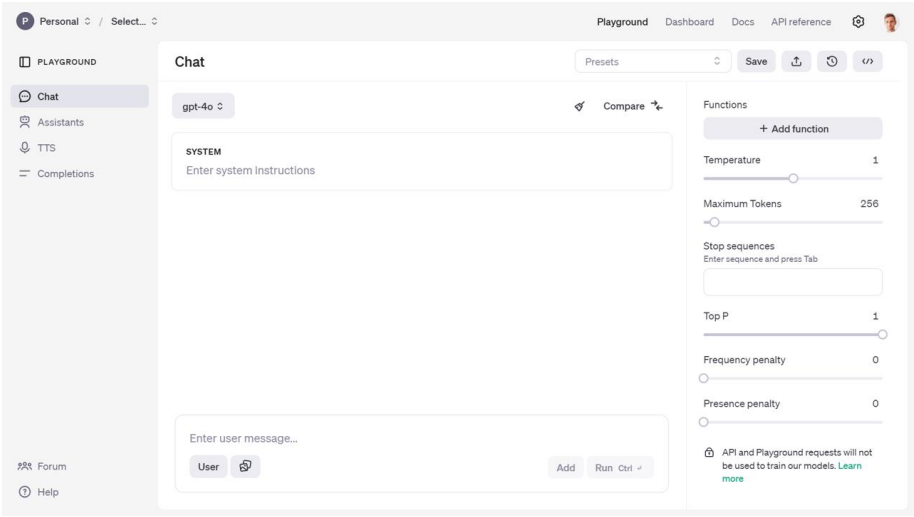


Figure 2-2 Play around in the playground and test the model APIs

This will give you an experience close to the one you have with the ChatGPT web app. But from there you will be able to access more advanced parameters of the model and view the code necessary to replicate the call to the API from your own program. If you are lacking inspiration, and you don't know where to start, you will find some prompt examples from the presets in the documentation[18].

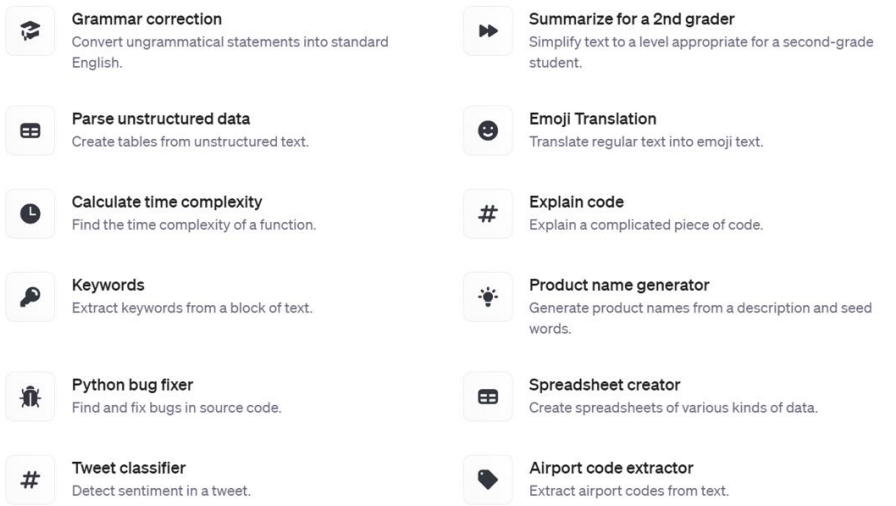


Figure 2-3 Prompt examples

[18] Prompt examples: https://platform.openai.com/docs/examples

If you want to take a guided look at the documentation[19] before trying anything or go lower level to the definition of the functions in the API reference[20].

Finally you can also access a dashboard to access management services:

- *Assistants*: to create and manage your own assistant models
- *Fine-tuning*: to fine-tune your own models
- *Batches*: to manage your batch jobs
- *Storage*: to manage files and vector stores
- *Usage*: to give you a sense of your consumption of the OpenAI web services.
- *API keys*: to manage your API keys

/! As an important data privacy disclaimer, API and Playground requests will not be used to train OpenAI models. This isn't the case of the public ChatGPT App, which by default can learn from users' conversations.

GETTING STARTED WITH THE CHAT COMPLETION API

AUTHENTICATION

Any call to the API needs to be authenticated with a key. You can find your API key in the *API keys* section of the dashboard. As a good practice for a SaaS application, you want it to take its configuration (including keys) from environment variables, like a 12-factor app[21]. As a result, launching the app in development is not very practical because you have to set those environment variables yourself. There are two methods we recommend for storing your keys securely on your local machine.

The first method is storing your keys as environment variables in a *.env* file in the development phase, like this: `OPENAI_API_KEY="sk-xxx"`. To load the environment variables, you will need to `pip install python-dotenv`. Add the following two lines of code in any of your python script:

```
from dotenv import load_dotenv

load_dotenv()  # take environment variables
```

[19] Doc: platform.openai.com/docs
[20] API Reference: platform.openai.com/api-reference
[21] Methodology for building SaaS apps: 12factor.net

```
# Code of your application, which uses environment variables
# (e.g. from `os.environ` or `os.getenv`) as if they came from the
actual environment.
```

What this does is reading key-value pairs from a *.env* file located at the root of your project and setting them as environment variables.

An alternative is to store your key as a variable `OPENAI_API_KEY="sk-xxx"` in a file *secrets.toml*. This type of file can be manipulated with a dedicated python package: `pip install tomli`. Since it will be used in the next section by Streamlit, save this file in a folder called *.streamlit/* and load it with the following command:

```
with open('.streamlit/secrets.toml','rb') as f:
    secrets = tomli.load(f)
```

This step won't be necessary if you run a Streamlit app, as Streamlit will take care of loading the secrets from the *.streamlit/* folder automatically. But if you run in a python script or a jupyter notebook, you will need to perform one of those options to load the API key as environment variable before the next section.

/! Make sure that for both of those options you list the file/folder *.env* and *.streamlit/* in the *.gitignore* file to avoid sharing your secrets in your public github repo.

Sending your first request to the OpenAI API with Python

This is what the first lines of code look like from the version 1.0 of the OpenAI Python SDK:

```
import openai
openai.api_key = "sk-XXXX"
m = [{'role': 'system', 'content': 'If I say hello, say world'},
    {'role': 'user', 'content': 'hello'}]
completion = openai.chat.completions.create(model='gpt-4o-mini',
                                            messages=m)
response = completion.choices[0].message.content
print(response) # world
```

The `chat.completions` endpoint of the API has several input parameters, but only two are required: models and messages.

- *Models*

OpenAI offers developers to utilize their models through a REST API, as well as through python and javascript libraries. These models are hosted on remote servers for easy querying both for prototyping and production use. Each model has distinct capabilities and pricing options. It's crucial to understand that these proprietary models can't be tweaked directly in code. However, you can fine-tune some on your unique data using the OpenAI API. However earlier models from OpenAI like GPT-2 aren't proprietary. You can download GPT-2 from sources like Hugging Face or GitHub, but it is not available via the API. For this chapter and most of the book, we will use GPT-4o mini because it is the most cost effective model from the latest generation of OpenAI GPT Models.

/! The OpenAI python library is actually quite flexible and can also be used for other model providers, such as GitHub Models, or local LLMs served with Ollama.

- *Messages*

The first big mystery that was in the back of my mind when using the chatGPT web app, was: "Does it remember what I say?" The conversational format certainly makes you feel like it does. But by taking a look at the underlying API for chat completion, you will realize that there isn't any memory in the system. All the context is passed each time to the service, that is stateless. This can be understood by taking a look at the transformer architecture in chapter 1, where the model can complete text and pay attention to the context, given that it fits into the context window. The messages parameter is a list of dictionaries[22] that form the conversation history. Each message dictionary includes a role (either system, user, or assistant) and content, which contains the text of the message. This allows the model to understand the context of the conversation and generate relevant responses. Example:

```
"messages": [
  {"role": "system", "content": "You are a scientist named Albert."},
  {"role": "user", "content": "Explain the theory of relativity."}]
```

The first entry is a system prompt: it is going to serve as context for the rest of the conversation, forcing the chat to behave in a certain way. Some developers have

[22] https://cookbook.openai.com/examples/how_to_format_inputs_to_chatgpt_models

found success in continually moving the system message near the end of the conversation to keep the model's attention from drifting away as conversations get longer.

The list of messages can contain as many user and assistant exchanges as you want, as long as you do not exceed the model's context window. The context window is the maximum number of tokens that the model can remember from the conversation history. If you exceed this limit, the model will start to forget the beginning of the conversation.

Additional parameters can be passed to the chat completion API (like max number of tokens, number of choices to generate, and stream option). Those will be addressed in the last section of the chapter.

PRICES OF THE API

The prices[23] have evolved quite a bit since the introduction of the ChatGPT API. As of the time of this writing (gpt-4o-2024-11-20):

Table 2-1 Prices of the API

model	input price ($ per 1M tokens)	output price ($ per 1M tokens)
gpt-4o-mini	$0.150	$0.600
gpt-4o	$2.5	$10

/! Make sure to check the latest prices of the API, and consider how you could be submitting your request as a *batch*[24]. Responses will be returned within 24 hours for a 50% discount. If some of your input tokens are repeated across requests, they will be automatically *cached*[25] giving you 50% discount compared to uncached prompts.

To give you a sense of the cost of using the API on a daily basis, here is a view over the month of May, where I spend the most of my time writing this book and developing the associated GPTs.

[23] https://openai.com/api/pricing
[24] https://platform.openai.com/docs/guides/batch
[25] https://platform.openai.com/docs/guides/prompt-caching

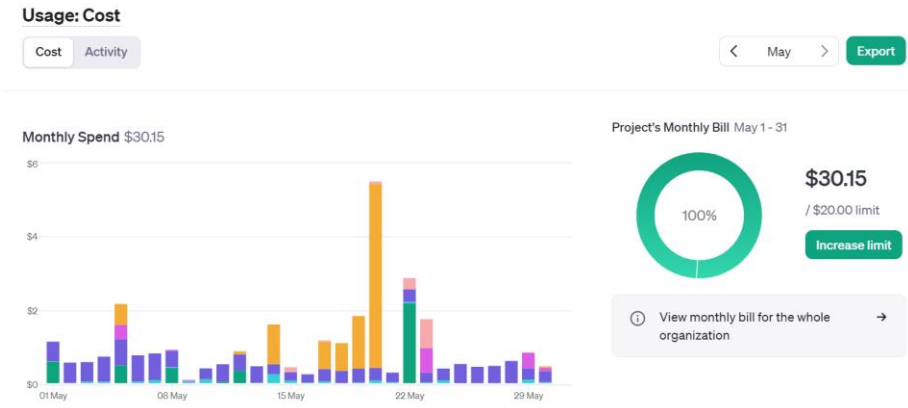


Figure 2-4 Cost of OpenAI services over the period of a busy month

If you hover over the graphic, you can see the breakdown by service (image, audio, embeddings, …) or by model (3.5, 4o, …). The big spike mid-month was due to the Dall-E 3 service usage for over $5 on 1 day.

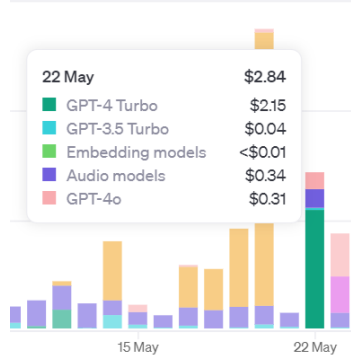


Figure 2-5 Breakdown of the cost per service

You can add email alerts and set budget limits to control your spending. As I started integrating more AI into my apps over the year, I ended up creating new keys for each project, and even distributing keys to friends and colleagues:

NAME	SECRET KEY	TRACKING ⓘ	CREATED	LAST USED ⓘ	PERMISSIO
Streamlit	sk-...hq8w	Enabled	Dec 5, 2022	May 30, 2024	All
Adam	sk-...5Mub	Enabled	Mar 22, 2023	May 30, 2024	All

Figure 2-6 Table of active OpenAI keys

This enabled me to have a finer grain control over the different projects including AI. You now have the ability to actually create a “project” that can contain members and have dedicated limits attached to it

BUILD YOUR FIRST CHATBOT APP

In this section, you will learn how to build your very first chatbot application, leveraging the OpenAI APIs. It will serve as a foundation for the subsequent apps that will be built throughout the book. In the preface, you have discovered Streamlit, and why it is such an amazing framework for Python web applications. Now you will experience the basics of building such an application yourself. The resulting app is stored on GitHub in the file *chat_app.py*[26], and I will walk you step by step through the development of such an app.

I will breakdown the development of the chatbot into the following steps:

1. *Graphical components*: Define the basic user interactions with the app
2. *Chat elements*: Display the chat conversation in the main area, and implement convenience functions to save and load the chat history in a sidebar
3. *Session states*: Store the state of your app in a session variable to simplify the interaction schema
4. *Response streaming*: Display the response as it is being generated

GRAPHICAL COMPONENTS

This part is going to focus more on the basics of Streamlit. You will leverage two types of text display (a title and a simple text), and more importantly an input field, in the form of a simple text input. You will see how by simply arranging those components in the natural ordering of the script, they will render in your web app in the same linear way, from top to bottom. Figure 2-7 represents the local serving of the app. 127.0.0.1 represents the localhost, and the default port is 8501.

[26] https://github.com/yanndebray/programming-GPTs/blob/main/chap2/chat_app.py

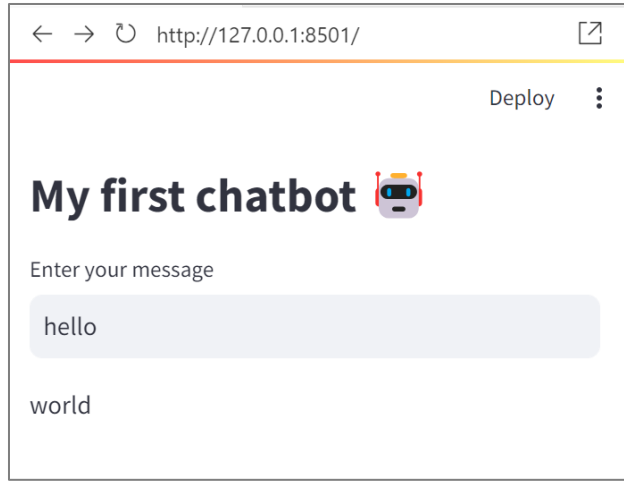


Figure 2-7 Simple chat user interface

This is the code for the first chat Graphical User Interface in Streamlit:

```
import openai
import streamlit as st
openai.api_key = st.secrets['OPENAI_API_KEY']
st.title('My first chatbot 🤖')
m = [{'role': 'system', 'content': 'If I say hello, say world'}]
prompt = st.text_input('Enter your message')
if prompt:
    m.append({'role': 'user', 'content': prompt})
    completion = openai.chat.completions.create(model='gpt-4o-mini',
                                                messages=m)
    response = completion.choices[0].message.content
    st.write(response)
```

As mentioned in the previous section about authentication, the API key is stored locally in a folder *.streamlit*. This way, Streamlit will know where to find your secrets, that are stored as a Python dictionary. A simple title will be rendered as a h1 heading. This is a simple way of informing your visitors what the app does. The fun starts with the function *text_input*, that displays a single-line text input widget. The magic operates by simply assigning the text output of this function to a variable (called prompt here). As we've seen in the previous section about the chat completion API, messages are passed as a list of dictionaries, and the user prompt is logically

appended to the end of this list initialized at the top of our script. To serve up the app locally, use the streamlit command instead of simply running the script with python:

```
streamlit run chap2\2_2_first_chat.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.1.30:8501
```

If you change your source code and save it, you will see a message at the top right of the app, prompting you rerun the app as displayed in figure 2-8. Another convenient option is to select *Always rerun* to automatically take your changes into account every time you update your code.

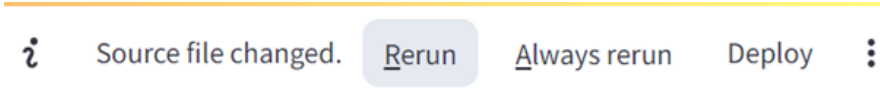


Figure 2-8 Rerun the app

You can gradually increase the complexity of the app, by adding *radio* buttons to choose the model:

```
models_name = ['gpt-4o-mini', 'gpt-4o']
selected_model = st.radio('Select OpenAI model', models_name)
st.write(f'Selected model: {selected_model}')
```

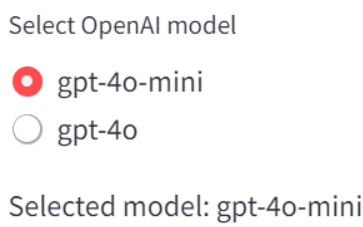


Figure 2-9 Model selection with radio buttons

This other type of graphical component enables a finite number of choice options to be displayed for an exclusive selection of one only (unlike checkboxes that enable multiple choices). This component operates similarly to the selectbox that we will see later, but that hides the selection to only reveal the selected option.

Finally, another very important component for an app is a simple… button. In the following code, you will see how to condition sending you message with the boolean trigger of the button state.

```
prompt = st.text_input('Enter your message')
if st.button('Send'):
    st.write(prompt)
```

Now that you have covered the basics of a Streamlit app, let's look more at the behavior of a chatbot.

CHAT ELEMENTS

In the app you developed in the previous section, the layout is linear and top down. This is the default rendering of a Streamlit app, following the flow of the python script. But to interact with a chatbot, you would expect the input to remain at the bottom of the page, and the messages to stack vertically to form a conversation.

Two new methods have been added in 2023 to facilitate the development of chat-based apps:

- *chat_message* lets you insert a chat message either from the user or from the assistant.
- *chat_input* lets you display a chat input widget at the bottom so the user can type in a message.

```
import streamlit as st

messages = [{'role': 'user', 'content': 'hello'},
            {'role': 'assistant', 'content': 'world'}]

if prompt := st.chat_input('Enter your message'):
    messages.append({'role': 'user', 'content': prompt})

for line in messages:
    if line['role'] == 'user':
        with st.chat_message('user'):
            st.write(line['content'])
    elif line['role'] == 'assistant':
        with st.chat_message('assistant'):
            st.write(line['content'])
```

As you can see from the structure of the code, the rendering of the messages will be done after a new message has been added to the list. But the layout of the app is special in this case, as the chat inputs shows up at the bottom of the page as shown in Figure 2-10 resulting from the code snippet above.

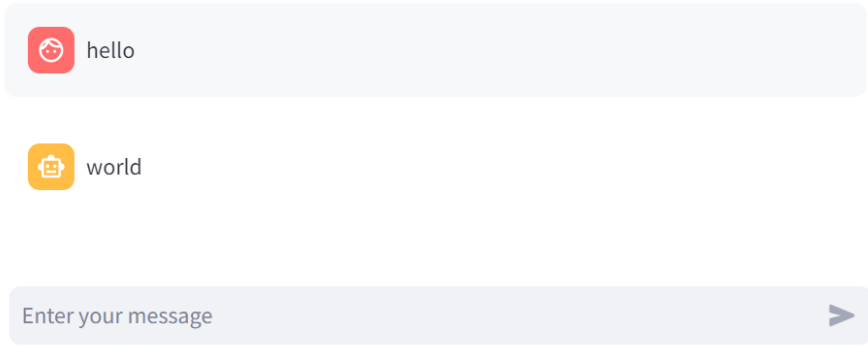


Figure 2-10 A basic chat dialog

A similar result can be achieved in a simpler for loop:

```
for line in messages:
    st.chat_message(line['role']).write(line['content'])
```

In order to design the behavior of the app, without calling OpenAI each time, I'd recommend creating a dummy chat function:

```
import streamlit as st
import json

# Functions
def dumb_chat():
    return "Hello world"

if prompt := st.chat_input():
    with st.chat_message('user'):
        st.write(prompt)
    with st.chat_message('assistant'):
        result = dumb_chat()
        st.write(result)
```

This is a good practice whenever you want to design an App based on an API, to first design the User Experience of the app with dummy functions. This function will then be replaced by the call to the OpenAI API when you are ready to implement and test the behavior with real responses:

```
def chat(m):
   completion = openai.chat.completions.create(
     model='gpt-4o-mini',messages=m)
    response = completion.choices[0].message.content
    return response
```

In order to start building a log of conversations that can be saved and loaded, you can use the json library to save and load the conversation history:

Figure 2-11 Sidebar with a conversation history

This sidebar in Figure 2-11 will be loading an existing conversation, with a function called *load_chat*, as a dictionary for debugging:

```
def load_chat(file):
  with open(f'chat/{file}') as f:
    convo = json.load(f)
  return convo
```

```
st.sidebar.title('ChatGPT-like bot 🤖')

convo_file = st.sidebar.selectbox('Select a conversation', os.listdir('chat'))
# st.sidebar.write(convo_file)
convo = load_chat(convo_file)
st.sidebar.write(convo)
```

/!\ The sidebar is a useful component, as it enables to display widgets in a left panel that can be folded on a desktop screen. By default, it appears hidden on a mobile screen but can be toggled to cover the main area. It is a good place to display contextual information or input elements such as the conversation history, or additional parameters of the API like model selection.

Now let's save the new elements of the conversation, with a function called *save_chat*:

```
def save_chat(convo,file):
  with open(f'chat/{file}','w') as f:
    json.dump(convo, f, indent=4)

# Create a text input widget in the Streamlit app
if prompt := st.chat_input():
  # Append the text input to the conversation
  with st.chat_message('user'):
    st.write(prompt)
  convo.append({'role': 'user', 'content': prompt})
  # Query the chatbot with the complete conversation
  with st.chat_message('assistant'):
     result = chat(convo)
     st.write(result)
  # Add response to the conversation
  convo.append({'role':'assistant', 'content':result})
  save_chat(convo,convo_file)
```

You can implement other functions such as *add_message* and bring all of those functions together in the main script:

```
import streamlit as st
import json

# Functions
def load_chat():
    with open('chat/convo0.json') as f:
        dummy = json.load(f)
    return dummy

def dumb_chat():
    return "Hello world"

def add_message(messages,role,content):
    messages.append({'role': role, 'content': content })
    return messages

messages = load_chat()
st.sidebar.write(messages)

if prompt := st.chat_input():
    with st.chat_message('user'):
        st.write(prompt)
    messages = add_message(messages,'user',prompt)
    st.sidebar.write(messages)
    with st.chat_message('assistant'):
        result = dumb_chat()
        st.write(result)
    messages = add_message(messages,'assistant',result)
    st.sidebar.write(messages)
```

The different functions introduced in this section are meant to provide convenience in the building of a chatbot app. They are not mandatory, but they will help you to structure your code in a more modular way. The main goal is to make the code more readable and maintainable, and to facilitate the debugging process.

Session states

In the previous section, you have seen how to save and load a conversation. But you don't really want to be writing to disk every element of our app that needs to store a state. So now is a good time to dig deeper into the execution model of Streamlit, and how to memorize the elements from the app. Streamlit reruns your script from top to bottom every time you interact with your app. Each reruns takes place in a blank

slate: no variables are shared between runs. Session states[27] is a way to share variables between reruns, for each user session. You have seen how to store messages as a list of dictionaries and append new messages to this list subsequently for the user and the assistant. In this section, you will see how to make use of Streamlit session state in order to store the conversation as illustrated in figure 2-12.

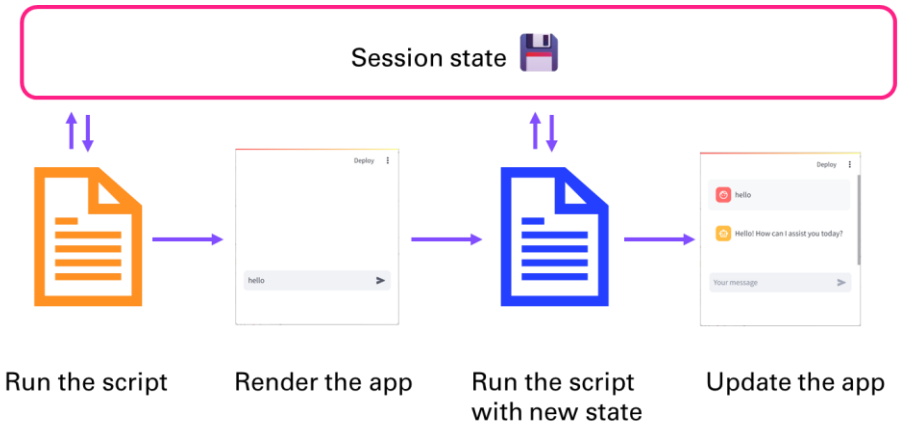


Figure 2-12 Store state for the duration of the session

Creating a Streamlit app and opening it in a browser initiates what's known as a session, which is a Python object that resides in memory on the back end server. A session lasts as long as the user has the browser tab open and there's an active connection with the app's back end, meaning that the app's UI elements can interact with Streamlit's server. Opening the app in a separate window or tab creates a separate session, ensuring users in different locations don't impact each other's experience. The session represents activity in the browser, while the session state captures and retains the current data from widgets and parameters for future use. I will go over how to effectively employ session state with your Streamlit app.

First you need to initialize the parameters of the session state:

```
# Initialization
if 'convo' not in st.session_state:
  st.session_state.convo = []
```

Then we append the content of the conversation to the session state:

```
if prompt := st.chat_input():
  # Append the text input to the conversation
```

[27] https://docs.streamlit.io/library/advanced-features/session-state

```
    with st.chat_message('user'):
      st.write(prompt)
    st.session_state.convo.append({'role': 'user', 'content': prompt
})
    # Query the chatbot with the complete conversation
    with st.chat_message('assistant'):
      result = chat(st.session_state.convo)
      st.write(result)
    # Add response to the conversation
                    st.session_state.convo.append({'role':'assistant',
'content':result})
```

STREAMING RESPONSE

Finally, one small thing is missing in order to have an app that looks and feels just like ChatGPT. We will modify the call to the openai chat completion API, by adding the parameter stream.

Setting `stream = True` in a request makes the model start returning tokens as soon as they are available, instead of waiting for the full sequence of tokens to be generated. It does not change the time to get all the tokens, but it reduces the time for first token for an application where we want to show partial progress or are going to stop generations. This can be a better user experience and a UX improvement so it's worth experimenting with streaming.

```
 def chat_stream(messages,model='gpt-4o-mini):
    # Generate a response from the ChatGPT model
    completion = client.chat.completions.create(
          model=model,
          messages= messages,
          stream = True
    )
    report = []
    res_box = st.empty()
    # Looping over the response
    for resp in completion:
        if resp.choices[0].finish_reason is None:
            # join method to concatenate the elements of the list
            # into a single string, then strip out any empty strings
            report.append(resp.choices[0].delta.content)
            result = ''.join(report).strip()
            result = result.replace('\n', '')
            res_box.write(result)
    return result
```

Congratulations 🎉 You have created your first chatbot. If you assemble the learnings from the previous sections, you will be able to replicate the code in `chat_app.py`

ADVANCED PARAMETERS

Let's look at a few additional parameters that you might want to tweak to adjust the result you are getting from the chat completion API. I will walk you through three main elements to take into account to achieve mastery in your AI applications:

- Max tokens and cost management
- Probabilities of the next token
- Temperature and sampling

MAX TOKENS AND COST MANAGEMENT

As you have seen in a section above, you have admin tools to manage your usage and the cost incurred for using the API. Another way to implement guardrails directly into your application is to count and limit tokens, sent to or produced by the GPT model.

Let's digress for a moment in order to implement a simple cost management dashboard displayed in Figure 2-13 that might be useful for you in many of the apps you will bring to production (given that you meet a certain audience and popularity).

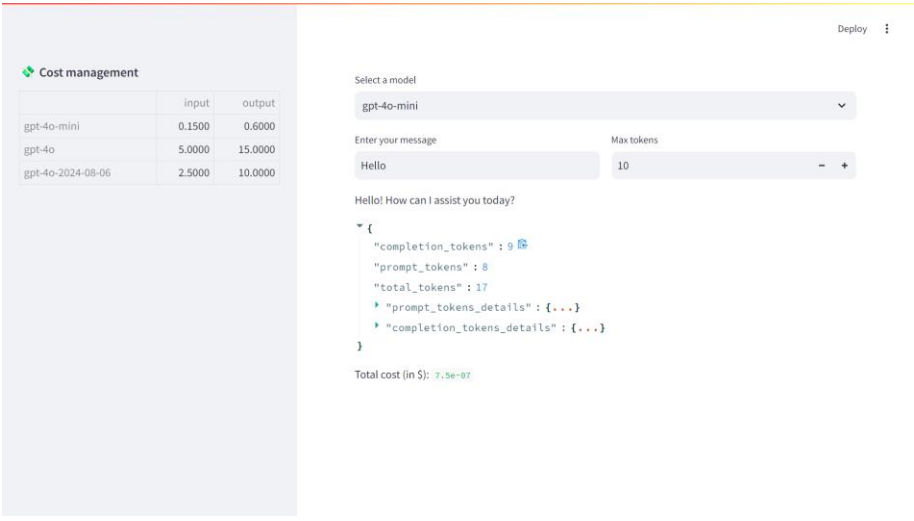


Figure 2-13 Bring in cost management as part of your application

The table of the current cost per million tokens (input and output) for each OpenAI model is loaded from a simple spreadsheet, and given the model selected, the cost of the response to a simple 'Hello' is computed, and broken down into prompt (input) and completion (output). The response "Hello! How can I assist you today?" only cost me $`7.5e-07`.

But if your app is consuming a large amount of prompts and subsequently generating long responses, you might want to consider throttling the chat completion by specifying the maximum number of tokens generated. In this example the limit is higher than the response, but if you play around with the Max tokens value, you will crop the response.

PROBABILITIES OF THE NEXT TOKEN

As you have seen in the previous chapter, the autoregressive prediction of the response to a prompt proceeds by predicting successively the sequence of next tokens, following a probability distribution. You can actually access this distribution of output tokens by specifying the parameter `logprobs=True` when calling the API.

The `top_logprobs` parameter provides insight into the model's decision-making process by returning the probabilities of the top n tokens considered at each step of generation, expressed along a logarithmic scale. These log probabilities can help you understand how confident the model is about its token choices and can be used for debugging, fine-tuning, or analyzing model behavior.

If you want to represent a probability between 0 and 1, you need to elevate this number with an exponential. Why not directly use the probability instead of the log to make the prediction you will ask. There are two main reasons why logprobs make token prediction faster and are easier for computers to work with.

1. It's cheaper for computers to do addition than it is to do multiplication. Figuring the next token is easier when you're adding the log probabilities of each token, instead of multiplying their actual probabilities.
2. Optimizing the log probability (logprob) is more effective than optimizing the probability itself - the gradient (the direction and rate of change) of the logprob tends to be smoother and more stable, making it easier to optimize during training.

The application in Figure 2-14 illustrates with a slider each probable next token. You can precise the number of probable tokens you want to display for each prediction.

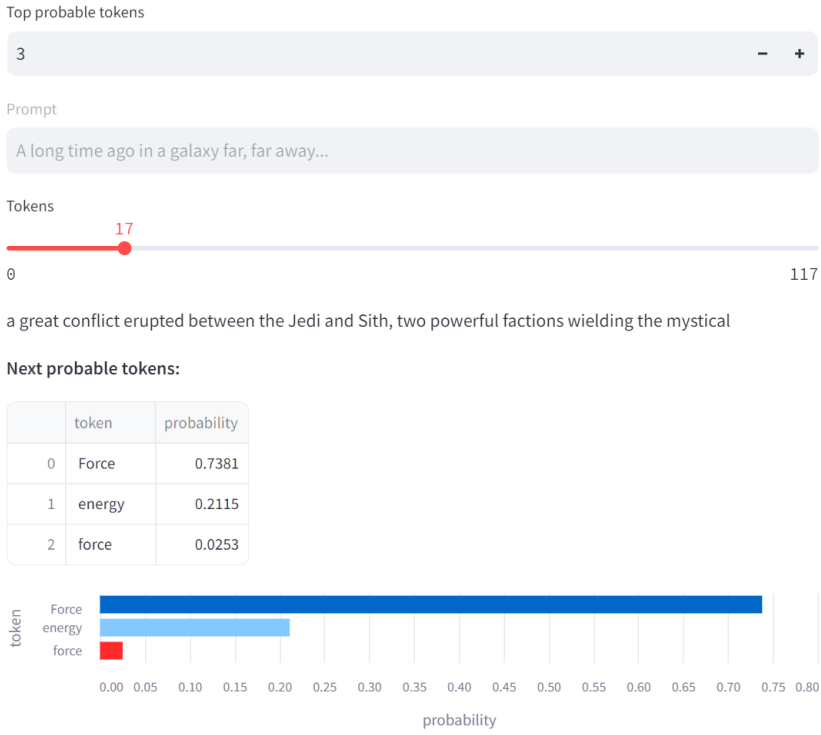


Figure 2-14 Probabilities for next tokens

The most probable next token here is “Force” but it’s not necessarily the one that will be selected for the generation of the response. There are several reasons for this. Always choosing the highest probability can lead to repetitive or predictable text. Lower probability choices can introduce novel and interesting ideas. Finally, human writing often includes less-than-optimal word choices, making the text feel more natural.

/!\ logprobs can be used to implement an auto complete feature or assess the confidence for a classification problem like sentiment analysis[28]

[28] https://cookbook.openai.com/examples/using_logprobs

TEMPERATURE AND SAMPLING

The temperature parameter controls the randomness of the output. A lower temperature (e.g., 0.2) makes the model's output more focused and deterministic, while a higher temperature (e.g., 0.8) results in more varied and creative responses.

An alternative to sampling with temperature, is the top_p parameter also called nucleus sampling, where the model considers the results of the tokens with top_p probability mass. So 0.1 means only the tokens comprising the top 10% probability mass are considered.

Figure 15 and 16 shows the impact of temperature and top_p on the next token probability distribution.

Figure 2-15 Impact of temperature on the probability of the next token

Figure 2-16 Impact of top_p on the probability of the next token

3. PROMPTING, CHAINING AND SUMMARIZATION

Now that you have seen the basics of how to develop a chatbot application, let's discover how to go from a generic AI agent into more specific use cases. In this chapter, you will learn how to craft efficient prompts, to leverage advanced chaining methods, and to apply those methods to a concrete summarization application on texts, such as articles, books, or transcripts.

THE ART OF PROMPTING

This chapter is meant to give you an intuitive understanding of how to interact with GPT models by careful writing prompts, in such a way that the model can understand. This simplistic explanation is what is currently known as the art and science of *prompt engineering*.

Or as Monsieur Jourdain, the main character of Molière's *The Bourgeois Gentleman*, would describe it: speaking in prose without knowing it. Shortly after the release of ChatGPT, you may have seen a lot of buzz in articles online, as well as the new profession of "prompt engineer" that supposedly is the future of many jobs such as programmer. I won't engage much further on this debate, but I will instead attempt to list down the recipes that make a good prompt.

Here is a simple example of what could be described as a bad prompt, inherited from the age of googling and simple keyword search, with one quick fix that precise a little more the intent behind the request:

- *Ineffective Prompt*: "Quantum mechanics."
- *Effective Prompt*: "In simple terms, explain the basic principles of quantum mechanics to a high school student."

Creating an effective prompt involves clarity, context, and specificity.

- *Clarity*: Be explicit about the task.
- *Context*: Provide necessary background information.
- *Specificity*: Define the format and style of the desired response.

There are some additional techniques for improving responses:

- *Role Assignment*: Assign a role to guide perspective.

```
prompt = "You are a movie critic. Analyze the Star Wars saga."
```

- *Formatting Instructions*: Guide the structure of the output (you can specify the desired length of the response).

```
prompt = "List the top 3 best Star Wars movies in bullet points."
```

- *Style Guidelines*: Specify tone and style.

```
prompt = "Write a friendly email from Chewbacca to Han Solo."
```

An important part of effective prompting is providing the necessary context. There are different approaches to this that will be covered throughout the book. The simplest method that was introduced in the GPT-3 paper is few-shot prompting. Essentially, the idea is to provide examples of inputs and outputs from which the language model can generalize the pattern. This is an illustration for a translation task:

```
prompt = """Translate the following sentences from English to
French.\n\n

1. 'Hello, how are you?'
-> 'Bonjour, comment ça va?'

2. 'Good morning.'
->

"""
```

Different variations of this method will be leveraged, and more generally you will see a mix of each of the techniques above used in some of the code snippets of the book that perform calls to GPT models.

PROMPT ENGINEERING TECHNIQUES IN APP DEVELOPMENT

You should not only think about GPTs as chatbots. Here is an example that illustrate an application requiring the use of several AI agents to manage and curate a database of bookmarks. The pitch is the following: As a tech professional, I often accumulate a vast collection of bookmarks for articles, tutorials, documentation, and blog posts. Managing this extensive library presents several challenges:

- *Insights*: Identifying trends or areas of focus within the collection.
- *Organization*: Keeping bookmarks categorized and easy to navigate.
- *Summarization*: Obtaining concise overviews without re-reading entire articles.

- *Retrieval*: Quickly finding relevant articles when needed (it will be covered in the next chapter).

Throughout this chapter, we will develop an app that:

1. *Extracts* key information from each bookmarked article (title, author, content).
2. *Translate* all articles in a target language (here mostly from French to English).
3. *Classifies* articles into relevant tech categories (e.g., Machine Learning, Web Development).
4. *Summarize* concise abstracts for quick reference, after identifying some key points.
5. *Scales* to thousands of bookmarks efficiently with prompt templates.

EXTRACT CONTENT

Assuming you have gathered content over the years as bookmarks in your web browser or in your favorite bookmarking service (for me it was Diigo), the first step will be to export those into a structured format. For each bookmark, you will typically have:

- URL
- Title
- Date added

The first step is to fetch the content from each URL and extract the main text using web scraping tools.

```
import requests
from bs4 import BeautifulSoup

def extract_content(url):
    try:
        # Fetch the URL content
        response = requests.get(url)
        response.raise_for_status()  # Raise an error for bad status
codes
        html = response.text

        # Parse the HTML using BeautifulSoup
        soup = BeautifulSoup(html, 'html.parser')
```

```
        title = soup.title.string if soup.title else "No Title Found"
        content = soup.get_text(separator='\n')

        return title, content
    except Exception as e:
        print(f"Error fetching {url}: {e}")
        return None, None
```

Based on this extracted content, you can format it properly and store it in a text file. Here is a function that uses OpenAI to format the content:

```
def format_content(title, content):
    try:
        prompt = f"""You are an expert text formatter and summarizer.
        Here is an article:

        Title: {title}

        Content:
        {content}

          Please format this scraped text into a clean, readable
content."""
        formatted_content = bot(prompt)

        return formatted_content
    except Exception as e:
        print(f"Error formatting content with OpenAI: {e}")
        return "Failed to format content."
```

For any of the operations requiring AI in this chapter, let's implement a generic function using the OpenAI API:

```
import openai

def bot(prompt, max_tokens=150, temperature=0):
    response = openai.chat.completions.create(
        model='gpt-4o-mini',
        messages=[
            {"role": "user", "content": prompt}
        ],
        max_tokens=max_tokens,
```

```
        temperature=temperature
    )
    return response.choices[0].message.content.strip()
```

/! The temperature is set to 0 by default as most of the operations you will be performing in this example do not require much creativity.

Translate articles: Zero-shot Prompting

An important step to standardize all the content for the knowledge base is to translate all the articles in English. Luckily for us, translation is one of the natural language processing tasks in which language models excel, especially as they are getting quite large and capable in multiple languages.

Here is a short function that uses a simple prompt, without the need to give any illustrative example:

```
def translate_article(content):
    prompt = f"""
    Translate the following article in English:

    \"\"\"
    {content}
    \"\"\"

    Translation:
    """
    translation = bot(prompt)
    return translation.strip()
```

/! You actually do not need to translate different languages to English in order to perform further processing like classification from different languages. The GPT models are able to reason around different language inputs. But this is much more convenient to start with this if there is a need to involve a human in the loop to verify the results of each step of the pipeline.

Classify articles: Few-Shot Prompting

Classification is a perfect illustration of few-shots prompting. This concept introduced in chapter 1 with the GPT-3 paper demonstrates how giving one or several examples of the expected result can greatly increase the performance of the prediction.

In this case, you can provide examples to help the model classify articles into predefined categories.

```
 def classify_article(content):
     prompt = """
     You are an AI assistant that classifies technical articles into
categories. The categories are:

     - Machine Learning
     - Web Development
     - Mobile Development
     - Cloud Computing
     - Data Science
     - Programming Languages
     - Other

     For example:

     Article:
     \"\"\"
     An introduction to Kubernetes and container orchestration.
     \"\"\"
     Category: Cloud Computing

     Article:
     \"\"\"
     Understanding React Hooks and their usage in modern web apps.
     \"\"\"
     Category: Web Development

     Now classify this article:

     \"\"\"
     {}
     \"\"\"

     Category:""".format(content)
     category = bot(prompt)
     return category.strip()
```

/! One way to make the result of this function easier to process would be to leverage the JSON mode[29].

SUMMARIZE ABSTRACTS: CHAIN-OF-THOUGHT PROMPTING

As we compare GPT models to the human brain, one concept that enables us to draw this parallel is *Chain-of-Thoughts*[30]. It is typically express with a prompt that contains an expression such as:

"Think step by step before you answer"

This way the model is using additional tokens in his answer to "spell out" some of its thinking and can reflect on it in a later stage. In our case, you can encourage the model to think step by step to extract relevant keywords from the article.

```
 def summarize_article(content):
     key_points_prompt = f"""
     You are an AI assistant that summarizes technical articles. Read
the article below and think through the main points step by step before
writing the final summary.

     Article:
     \"\"\"
     {content}
     \"\"\"

      First, outline the key points and main ideas of the article.
Then, write a concise summary incorporating these points.

     Key Points:
     """
     # First, get the key points
     key_points_response = bot(key_points_prompt)
     key_points = key_points_response.strip()

     # Now, generate the final summary using the key points
     summary_prompt = f"""
      Using the key points below, write a concise summary of the
article.

     Key Points:
     {key_points}
```

[29]Structure outputs in JSON: https://platform.openai.com/docs/guides/structured-outputs#json-mode
[30] Chain-of-Thought:https://blog.research.google/2022/05/language-models-perform-reasoning-via.html

```
    Summary:
    """
    summary = bot(summary_prompt)
    return summary.strip()
```

Summarization is one of the most useful applications of language models. And you can get very varying results based on how you approach this task. You can consider using the o1 family of models for this type of task as they excel at reasoning, and stuff everything into one single prompt. But on the other hand of the spectrum, it doesn't necessarily take a very advanced or large model to only perform summarization, and you can consider farming this out to dedicated services with small language models.

Whenever you involve humans in a language processing task, summarization is a key ingredient, so make sure that you validate the results to make sure that the quality of the summary aligns with the level of information that you want to present to your users.

/!\ You might hit up against the context length. You will learn in the following sections of this chapter how to work around this.

Scale processing: Prompt templates

You can process bookmarks in batches to handle large volumes efficiently. Remember that in such cases you can use the batch API from ChatGPT to pay 50% less, if you don't need the result right away. You can also consider parallelizing this processing loop, as each call can be performed independently.

```
def process_bookmarks(bookmarks):
    processed_articles = []
    for bookmark in bookmarks:
        url = bookmark['url']
        title, content = extract_content(url)
        content = format_content(title, content)
        summary = summarize_article(content)
        category = classify_article(content)
        article_data = {
            'title': title or bookmark['title'],
            'category': category,
```

```
            'summary': summary,
        }
        processed_articles.append(article_data)
        # Optionally save to a database or file
    return processed_articles
```

/! In every function introduced in this section, we have been using Python formatted strings[31]. This method of including numbers into textual outputs is extremely relevant when it comes to templating calls to GPT models.

WHY CHAINING

Chaining is a technique that allows you to send multiple requests to an LLM in a sequence, using the output of one request as the input of another. This way, you can leverage the power of an LLM to perform complex tasks that require multiple steps, such as summarization, text simplification, or question answering.

The benefits of chaining are the following:

- *Enhanced contextual understanding*: By chaining calls, the LLM can maintain context over a longer interaction, leading to more coherent and relevant responses.
- *Complex task handling*: Chaining allows for the breakdown of complex tasks into simpler sub-tasks, which the model can handle more effectively.

Chaining enables to "fix" some of the inherent limitations of LLMs:

- *Reasoning*: LLMs are not proceeding like the human brain – remember, they simply infer the next logical word in a sentence. As such they are not good at solving basic math problems.

- *Access to Data and Tools*: LLMs do not have access to your data, only the ones they have seen during the training phase. By default, they do not have access to search over the internet. As such, it would not make sense to compare ChatGPT with Google search, even if their use cases overlap significantly.

[31] Python formatted strings: https://docs.python.org/3/tutorial/inputoutput.html

- *Training set fixed in time*: You probably remember ChatGPT saying that he did not have access to information after September 2021. This date is gradually updated by OpenAI but there is always a lag.

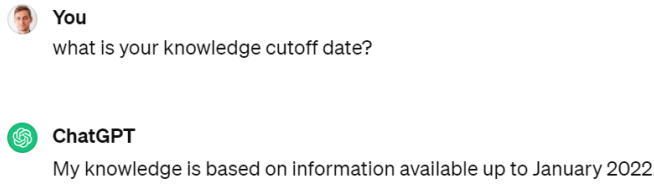


Figure 3-1 A typical response from ChatGPT mentioning is knowledge limitations

A major paper that has been at the origin of many of the improvements of the usage of LLMs is *React: Synergizing Reasoning and Acting in Language Models*[32].

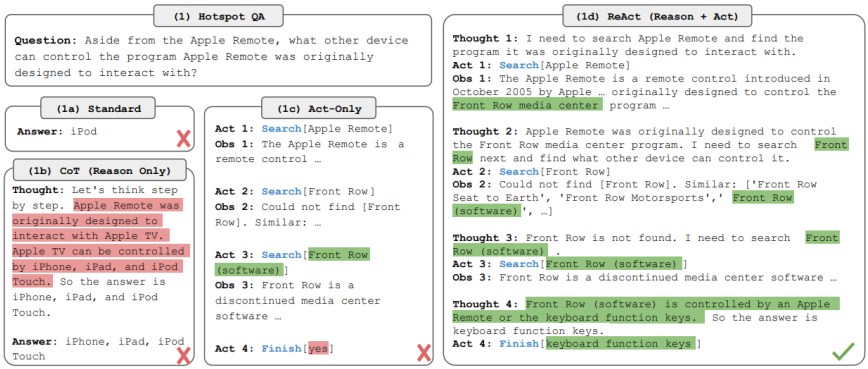


Figure 3-2 Example from the ReAct paper combining Reasoning and Acting

Instead of directly answering a question, this approach breaks down the process into Reasoning + Acting. A number of frameworks emerged to provide developers with the ability to integrate LLMs into their applications, by implementing the ReAct concepts. I'll cover the few main ones in this book:

- Semantic Kernel
- LangChain
- LlamaIndex
- Haystack

[32] ReAct paper: https://arxiv.org/abs/2210.03629

Most of those frameworks that appeared in 2023 came to complement LLMs like ChatGPT or open-source alternatives such as Llama2. But it is good to note that with new features enabled by OpenAI throughout the year, the need for those frameworks became less and less relevant as OpenAI has increasingly been providing an end-to-end integrated solution.

We will break down the analysis of those frameworks over the course of the next 3 chapters:

- First, we will look at the basics of chaining to go around the limitations of the LLM context
- Second, we will look at methods such as vector search to extend the LLM context
- Third, we will look at properties that appear when we extend the reasoning of LLMs

Semantic Kernel by Microsoft

I'm choosing to start with Semantic Kernel, as Microsoft provides good educational material on the main concepts around chaining. Semantic Kernel[33] is a lightweight Software Development Kit (SDK) developed by Microsoft that enables to mix conventional programming languages and Large Language Models with templating, chaining, and planning capabilities. Here is a diagram that explains the workflow triggered by the user asking a question:

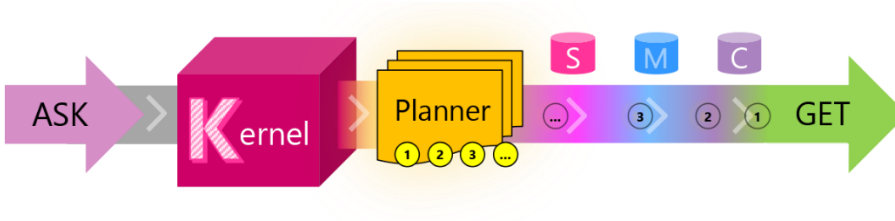


Figure 3-3 Journey of a request from an ASK to the Semantic Kernel framework all the way to the response the user GETs

[33] Semantic Kernel: https://learn.microsoft.com/fr-fr/semantic-kernel/

Table 3-1 Semantic Kernel workflow

Journey	*Short Description*
ASK	A user's goal is sent to SK as an ASK
Kernel	The kernel orchestrates a user's ASK
Planner	The planner breaks it down into steps based upon resources that are available
Resources	Planning involves leveraging available skills, memories, and connectors
Steps	A plan is a series of steps for the kernel to execute
Pipeline	Executing the steps results in fulfilling the user's ASK
GET	And the user gets what they asked for ...

LangChain 🦜🔗

LangChain is a framework that enables developers to chain calls to Large Language Models (such as ChatGPT). It offers implementations both in Python and Typescript (in order to migrate easily your experimentations into a JavaScript web app). We won't be needing this as we will reuse Streamlit.

Many of the concepts introduced in the early days of the LangChain framework were meant to complements the Large Language Models such as GPT 3.5:

- Extend the length of the context
- Augment generation by retrieving elements not in the context
- Add tools such as loading documents, searching the internet or doing basic math

Before ChatGPT was introduced (with the underlying 3.5 turbo model), you could already abstract the call to OpenAI GPT 3 models with LangChain, with a very simple syntax:

```
from langchain.llms import OpenAI
llm = OpenAI()
joke = llm('tell me a joke')
print(joke)
```

```
Q: What did the fish say when he hit the wall?
```

```
A: Dam!
```

It also provided the ease of swapping models providers:

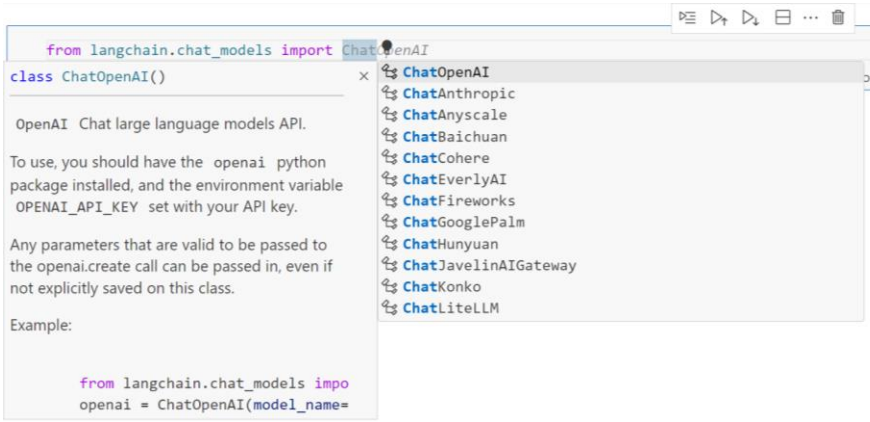


Figure 3-4 Autocomplete in VSCode displaying the chat models available in LangChain

But the use of chat models (like gpt-3.5-turbo) were preferred to base models performing simple text completion. They've been significantly optimized, and they are now the standard models available, so it is best to standardize your code on chat models.

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage
chat = ChatOpenAI()
text = "Tell me a joke"
messages = [HumanMessage(content=text)]
res = chat.invoke(messages)
print(res.content)
```

```
Sure, here's a classic joke for you:

Why don't scientists trust atoms?

Because they make up everything!
```

To learn more about LangChain, I would recommend the Youtube channel of Greg Kamradt[34].

[34] Langchain tutorials:
https://youtube.com/playlist?list=PLqZXAkvF1bPNQER9mLmDbntNfSpzdDIU5
https://github.com/gkamradt/langchain-tutorials

SUMMARIZING LONG DOCUMENTS

Summarization is one of the main use cases of Natural Language Processing that is used productively in a professional setting. The first application that came to my mind to leverage ChatGPT was summarizing meetings. This way I could simply turn on the transcription on Microsoft Teams and skip long overcrowded meetings. The same could be applied to long documents that I don't want to read.

WORKING AROUND THE CONTEXT LENGTH

Now there is one problem with this: a typical one-hour meeting is around 8k to 10k tokens. But initially the GPT-3.5 model powering ChatGPT could only handle 4096 tokens.

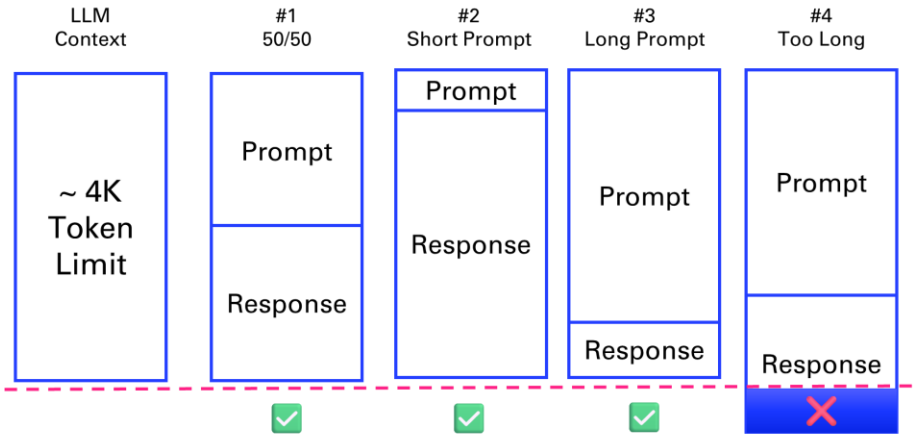


Figure 3-5 scenarios where the prompt and response fit the context length and 1 where it doesn't

What are *tokens* anyway? A helpful rule of thumb is that one token generally corresponds to ~4 characters of text for common English text. This translates to roughly ¾ of a word (so 100 tokens ~= 75 words). In what follows, we will use the tiktoken package from OpenAI anytime we need to compute precisely the number of tokens within our applications.

What does this mean in terms of pages? One page in average is 500 words[35] (like the first page of the forewords in Letter format 8.5" x 11"). Which means that 4k tokens

[35] Words per page: https://wordcounter.net/words-per-page

can fit 6 pages of text. As the previous diagram suggests, this context length is shared between the prompt and the response.

Let's illustrate this with a basic example of a conversation recorded as WebVTT file[36]:

```
import os, webvtt
files = os.listdir('../data/vtt')
file = files[1]
cap = webvtt.read('../data/vtt/'+file)
for caption in cap[1:5]:
    # print(f'From {caption.start} to {caption.end}')
    # print(caption.raw_text)
    print(caption.text)
```

```
I want to start doing this experiment where.

We have a conversation we record it's generating a VTT file.

And I have a parser. I developed a small app in Python that can retrieve
the VTT file process it.

And then.
```

We can extract the text from the conversation and save it as a plain text file.

```
txt = file.replace('.vtt','.txt')
m = [caption.raw_text for caption in cap]
sep = '\n'
convo = sep.join(m)
with open('../data/txt/'+txt,mode='w') as f:
    f.write(convo)
```

Let's count the numbers of token in the conversation.

```
import tiktoken
def num_tokens(string: str) -> int:
    """Returns the number of tokens in a text string."""
    encoding_name = 'cl100k_base'
    encoding = tiktoken.get_encoding(encoding_name)
    num_tokens = len(encoding.encode(string))
    return num_tokens
```

[36] WebVTT file: https://developer.mozilla.org/en-US/docs/Web/API/WebVTT_API

```
num_tokens(convo)
```

```
150
```

In the next parts, we will go over different chain types to address the limitation of the context length[37].

STUFFING

The first solution presented is "stuffing". If a document is within the model's context limit (4k tokens here), it can be directly fed into the API for summarization. This method is straightforward but limited by the maximum context length.

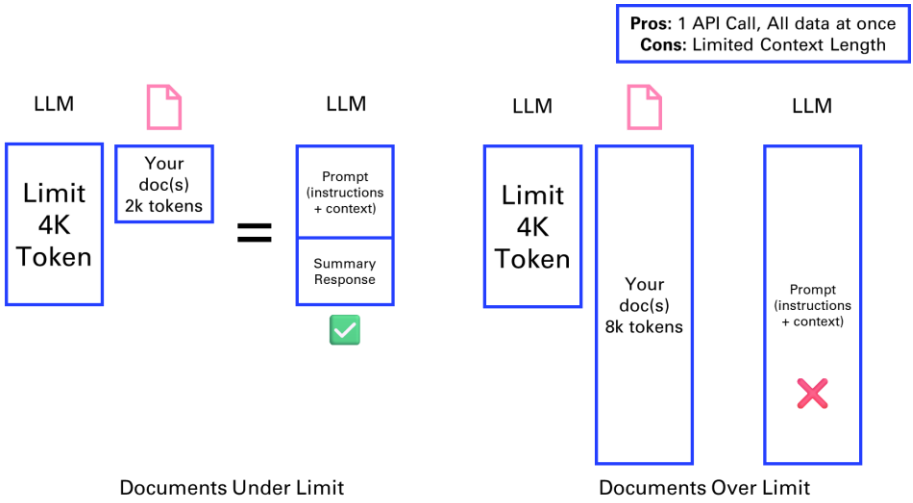


Figure 3-6 Stuffing a document into the context

This is how the code would look like with LangChain:

```
from langchain.chains.summarize import load_summarize_chain
from langchain.chat_models import ChatOpenAI
from langchain.document_loaders import WebBaseLoader, TextLoader
loader = TextLoader('../data/txt/'+txt)
docs = loader.load()
llm = ChatOpenAI(temperature=0, model_name="gpt-3.5-turbo")
chain = load_summarize_chain(llm, chain_type="stuff")
chain.run(docs)
```

[37] Workaround Token Limits With Chain Types: https://www.youtube.com/watch?v=f9_BWhCI4Zo

```
'Yann wants to improve his French accent and plans to conduct an experiment where he records conversations and generates VTT files. He has developed a Python app to process the VTT files and wants to use the ChatGPT API to further analyze them. Mike has not yet used the API due to lack of time.'
```

Try summarizing a longer text, like the content of a webpage like the LangChain doc page.

```
def summarize(page, model = "gpt-3.5-turbo"):
    loader = WebBaseLoader(page)
    docs = loader.load()
    llm = ChatOpenAI(temperature=0, model_name=model)
    chain = load_summarize_chain(llm, chain_type="stuff")
    return chain.run(docs)

page = "https://python.langchain.com/docs/use_cases/summarization"
try:
    summary = summarize(page)
    print(summary)
except Exception as e:
    print(str(e))
```

```
Error code: 400 - {'error': {'message': "This model's maximum context length is 4097 tokens. However, your messages resulted in 7705 tokens. Please reduce the length of the messages.", 'type': 'invalid_request_error', 'param': 'messages', 'code': 'context_length_exceeded'}}
```

Luckily as we will see in the last part of this chapter we can use a model with a larger context window:

```
page = "https://python.langchain.com/docs/use_cases/summarization"
summarize(page,model = "gpt-3.5-turbo-16k")
```

```
 'The LangChain platform offers tools for document summarization using large language models (LLMs). There are three approaches to document summarization: "stuff," "map-reduce," and "refine." The "stuff" approach involves inserting all documents into a single prompt, while the "map-reduce" approach summarizes each document individually and then combines the summaries into a final summary. The "refine" approach iteratively updates the summary by passing each document and the current summary through an LLM chain. The platform provides pre-built chains for each approach, and users can customize prompts and LLM models. Additionally, the platform offers the option to split long documents into chunks and summarize them in a single chain.'
```

MAP REDUCE

The MapReduce method involves splitting a large document (e.g., 8k tokens) into smaller chunks, summarizing each separately, and then combining these summaries into a final summary. This approach allows for processing larger documents in parallel API calls but may lose some information.

This is a term that some might be familiar with, from the space of big data analysis, where it represents a distributed execution framework that parallelized algorithms over data that might not fit on a single core.

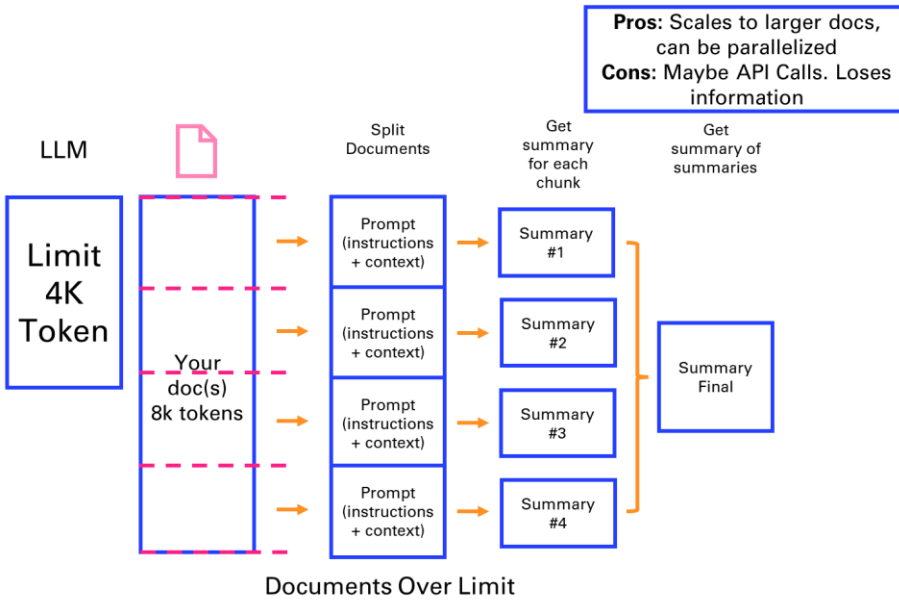


Figure 3-7 Map reduce approach to breaking down a summary into parallel steps

As an example, we will take another source of VTT files: Youtube transcripts[38]. The video we will summarize is the LangChain Cookbook - Beginner Guide To 7 Essential Concepts[39].

```
from youtube_transcript_api.formatters import WebVTTFormatter
from youtube_transcript_api import YouTubeTranscriptApi
```

[38] https://pypi.org/project/youtube-transcript-api/
[39] https://www.youtube.com/watch?v=2xxziIWmaSA

```
video_id = "2xxziIWmaSA" # https://www.youtube.com/watch?v=2xxziIWmaSA
transcript = YouTubeTranscriptApi.get_transcript(video_id)
formatter = WebVTTFormatter()
formatted_captions = formatter.format_transcript(transcript)
vtt_file = f'../data/vtt/subtitles-{video_id}.vtt'
with open(vtt_file, 'w') as f:
    f.write(formatted_captions)
sep = '\n'
caption = sep.join([s['text'] for s in transcript])
num_tokens(caption)
```

```
10336
```

```
# Turn VTT file into TXT file
txt_file = vtt_file.replace('vtt','txt')
with open(txt_file,mode='w') as f:
    f.write(caption)
```

Now comes a new concept that will be important anytime you are breaking down content into chunks: a *text splitter*[40].

The default recommended text splitter is the RecursiveCharacterTextSplitter. This text splitter takes a list of characters. It tries to create chunks based on splitting on the first character, but if any chunks are too large it then moves onto the next character, and so forth. By default, the characters it tries to split on are `["\n\n", "\n", " ", ""]`

In addition to controlling which characters you can split on, you can also control a few other things:

- `length_function`: how the length of chunks is calculated. Defaults to just counting number of characters, but it's pretty common to pass a token counter here.
- `chunk_size`: the maximum size of your chunks (as measured by the length function).
- `chunk_overlap`: the maximum overlap between chunks. It can be nice to have some overlap to maintain some continuity between chunks (e.g. do a sliding window).

[40] https://python.langchain.com/docs/modules/data_connection/document_transformers/#text-splitters

- add_start_index: whether to include the starting position of each chunk within the original document in the metadata.

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 1000,
    chunk_overlap = 0,
    length_function = num_tokens,
)

def doc_summary(docs):
    print (f'You have {len(docs)} document(s)')
     num_words = sum([len(doc.page_content.split(' ')) for doc in
docs])
    print (f'You have roughly {num_words} words in your docs')
    print ()
          print    (f'Preview:    \n{docs[0].page_content.split(".
")[0][0:42]}')

docs = text_splitter.split_documents(doc)
doc_summary(docs)
```

```
You have 11 document(s)

You have roughly 7453 words in your docs

Preview:

hello good people have you ever wondered
```

We can now perform the MapReduce method. We can enable the mode `verbose=True` to view the breakdown of each intermediate summary.

```
llm = ChatOpenAI(model_name='gpt-3.5-turbo')
chain = load_summarize_chain(llm, chain_type="map_reduce",
verbose=True)
chain.run(docs)
```

```
"The video introduces Lang chain, a framework for developing
applications powered by language models. It explains the components
and benefits of Lang chain, and mentions a companion cookbook for
further examples. The video discusses different types of models and
how they interact with text, including language models, chat models,
and text embedding models. It explains the use of prompts, prompt
templates, and example selectors. The process of importing and using
```

a semantic similarity example selector is described. The video also discusses the use of text splitters, document loaders, and retrievers. The concept of vector stores and various platforms are mentioned. The video explains how chat history can improve language models and introduces the concept of chains. It demonstrates the creation of different chain types in Lang chain, such as location, meal, and summarization chains. The concept of agents and their application in decision making is discussed, along with the process of creating an agent and loading tools into its toolkit. The speaker shares their positive experience using Lion Chain software and its ability to dynamically search for information. They mention the debut album of Wild Belle and encourage viewers to check out the band's music. The video concludes by mentioning future videos on the topic and encouraging viewers to subscribe and provide feedback."

REFINE

In the Refine method, the document is split into chunks, and each chunk is summarized sequentially, with each summary informed by the previous one. This method provides relevant context but is time-consuming due to its sequential nature. This is one of my favorite methods for summarizing.

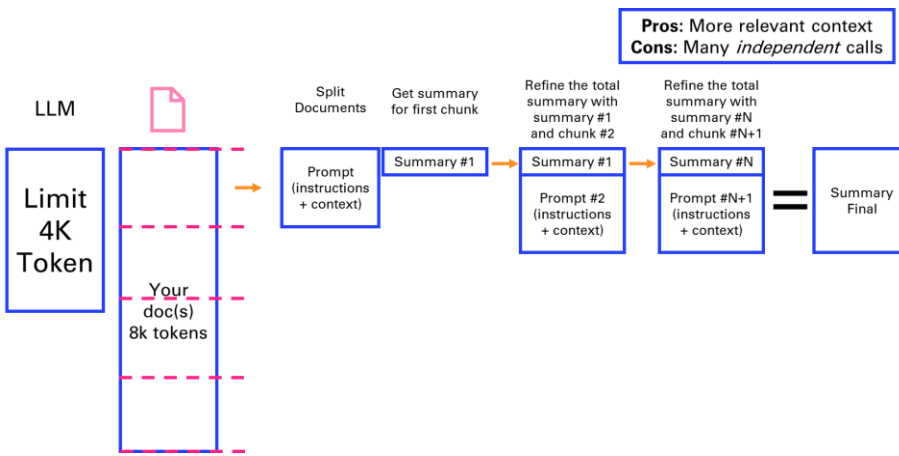


Figure 3-8 Refining a summary in sequential steps

We can directly try this method on the same content as the previous section and see how it compares.

```
chain = load_summarize_chain(llm, chain_type="refine", verbose=True)
```

```
chain.run(docs)
```

You can observe in the verbose output that the chain uses the following prompt:

```
Given the new context, refine the original summary.
If the context isn't useful, return the original summary.
```

Finally, I'll implement my own text splitter and summarizer using the refine method.

```
def my_text_splitter(text,chunk_size=3000):
    # Split text into chunks based on space or newline
    chunks = text.split()

    # Initialize variables
    result = []
    current_chunk = ""

    # Concatenate chunks until the total length is less than 4096
tokens
    for chunk in chunks:
        # if len(current_chunk) + len(chunk) < 4096:
        if num_tokens(current_chunk+chunk) < chunk_size:
            current_chunk += " " + chunk if current_chunk else chunk
        else:
            result.append(current_chunk.strip())
            current_chunk = chunk
    if current_chunk:
        result.append(current_chunk.strip())

    return result
```

```
import openai
def summarize(text, context = 'summarize the following text:', model
= 'gpt-3.5-turbo'):
    completion = openai.chat.completions.create(
        model = model,
        messages=[
        {'role': 'system','content': context},
        {'role': 'user', 'content': text}
            ]
    )
    return completion.choices[0].message.content
```

```
def refine(summary, chunk,  model = 'gpt-3.5-turbo'):
    """Refine the summary with each new chunk of text"""
    context = "Refine the summary with the following context: " +
summary
    summary = summarize(chunk, context, model)
    return summary
```

```
# Requires initialization with summary of first chunk
chunks = my_text_splitter(caption)
summary = summarize(chunks[0])
for chunk in chunks[1:]:
    summary = refine(summary, chunk)
summary
```

```
'In this video, the presenter discusses the Lang chain framework and
its various components, including Vector Stores, Memory, Chains, and
Agents. They provide code examples and highlight the versatility and
functionality of using Lang chain. In the end, they encourage viewers
to subscribe for part two and ask any questions they may have. They
also share tools on Twitter and encourage feedback and comments from
viewers.'
```

EVOLUTIONS OF THE CONTEXT WINDOW

This early view of limitations of Large Language Models has to be updated over time as the context length of new generations of LLMs are growing:

- GPT 4[41] (introduced on March 14, 2023) originally had an 8k tokens context window. A 32k tokens version was made available gradually in parallel. In November 2023, an 128k token version was introduced.
- GPT 3.5 got a 16k version in September 2023.
- Anthropic Claude[42] extended its context window to 100k tokens in May 2023, enabling to process an entire book like the Great Gatsby of around 200 pages (standard paperback edition).
- Claude 2.1[43] (Nov 2023) and Claude 3 (Mar 2024) propose an impressive 200k tokens context.

[41] GPT 4 model: https://platform.openai.com/docs/models/gpt-4
[42] Anthropic Claude 100k context window: https://www.anthropic.com/index/100k-context-windows
[43] Claude 2 & 3: https://www.anthropic.com/news/claude-2-1 - https://www.anthropic.com/news/claude-3-family

- LLaMa[44] from Meta initially came with a 2k tokens context window in February 2023.
- Llama2[45] released in July 2023 has a 4k tokens context window. Llama3 released in April 2024 doubles it again to 8k tokens. This is considered one of the leading open-source LLM.
- GPT 4o[46] (introduced on May 13, 2024) has a 128k tokens context window.

There are still advantages to apply the previous methods, either to mitigate the cost of long context windows and larger models, or to use smaller more dedicated models that can be open-source for certain specific tasks like summarizing.

[44] LLaMa: https://ai.meta.com/blog/large-language-model-llama-meta-ai/
[45] Llama 2 & 3: https://llama.meta.com/llama2/ - https://llama.meta.com/llama3/
[46] https://openai.com/index/hello-gpt-4o/

4. VECTOR SEARCH & QUESTION ANSWERING

Another very popular application of the Large Language Models is question answering. Unlike summarization that can be solved with a fixed context, answering questions can be a moving target.

This is why we will investigate methods to extend the context with additional tools such as search, and introduce a special kind of search called similarity search over a vector database of embeddings.

A lot of new concepts we will present in this chapter. But before we do, let's look at a method called Map Rerank that is very close to the ones covered in the previous chapter.

MAP RERANK

If you only have a single but long document that you want to feed as a source to the LLM to answer your question, you could consider the Map Rerank method.

This method, suited for question-answer tasks rather than summarization, involves splitting the document, posing questions to each chunk, and ranking the answers based on confidence scores. It's effective for single-answer questions but does not combine information across chunks.

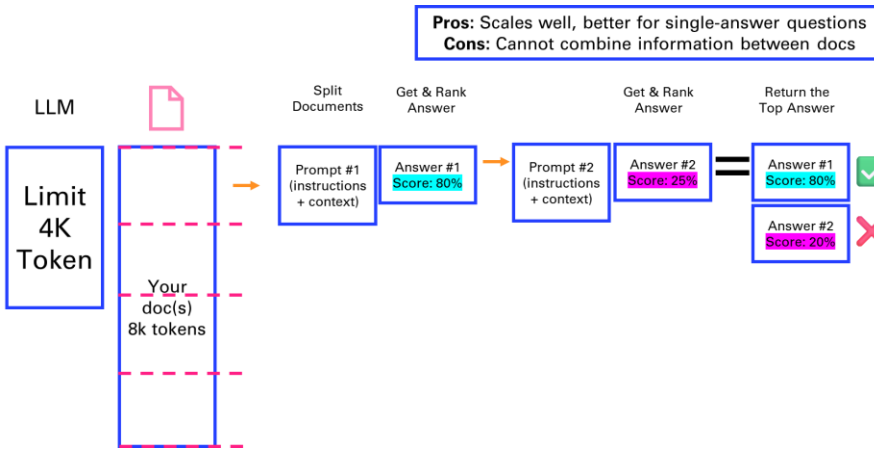


Figure 4-1 Map Rerank to answer a question over a long document

RETRIEVAL-AUGMENTED GENERATION

The previous approach of Map Rerank that consists in asking questions to each section of a document isn't optimal, as the number of requests to the LLM grows proportionally with the length of the doc. Another approach which consists in only retrieving the meaningful context for the request to the LLM is called *Retrieval-Augmented Generation*.

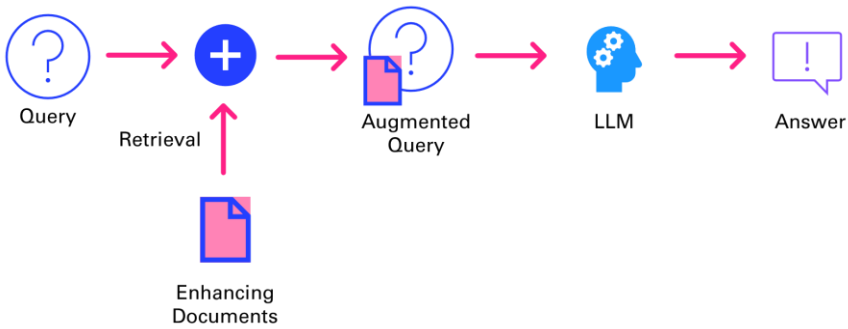


Figure 4-2 Retrieval Augmented Generation workflow

As described in Figure 4-2, the query isn't directly answered be the LLM. It first goes through an information retrieval system to pull the relevant context that will be fed to the LLM alongside the original query, to deliver an answer grounded in an extended knowledge base.

TRADITIONAL SEARCH

One of the first applications of ChatGPT was to answer questions that we previously were searching on the internet, via a search engine like Google. After all, a large language model is nothing more than a (very) compressed version of the internet, as it is the corpus on which they are primarily trained.

In this chapter, we will implement a very simple search tool that will input information from Google into our ChatGPT request. This way, the chatbot will be able to answer news related questions such as "Who is the CEO of Twitter?".

```python
# Don't forget to load your OPENAI_API_KEY as env variable
from openai import OpenAI
openai = OpenAI()
def ask(question):
    completion = openai.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": question}]
        )
    return completion.choices[0].message.content

prompt = 'who is the CEO of twitter?'
ask(prompt)
```

```
'As of September 2021, the CEO of Twitter is Jack Dorsey.'
```

ChatGPT is at least taking the precaution of stating the date of its recollection. But if we check on Google, we can see that things have changed quite a bit at Twitter since 2021 (even the name).

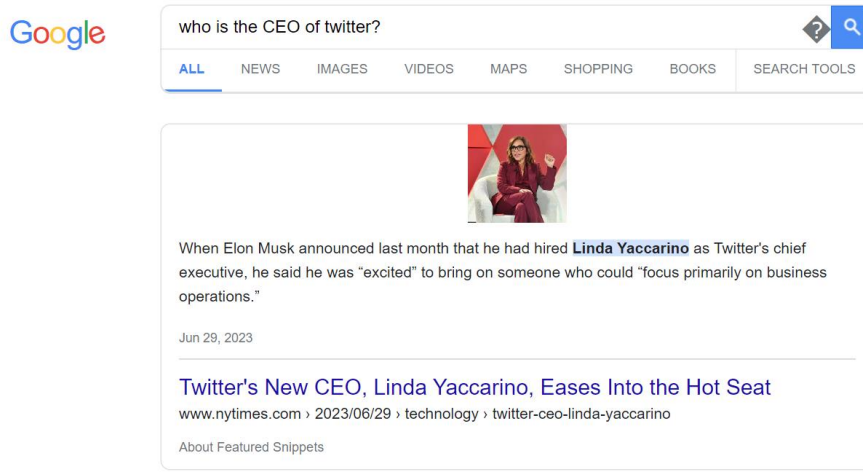


Figure 4-3 The good old days of searching the internet for information

Let's implement a simple search engine that will be able to answer questions about an evolving topic. In the following example, we will parse the Google result page. It provides typically 10 results, sometimes prefaced by a "featured snippet"[47] followed

[47] How Google's featured snippets work: https://support.google.com/websearch/answer/93517071

by other questions that “people also ask”, and at the end of the list of results, a few “related searches”. We will then check the number of tokens on the page to make sure that it isn’t too long to “stuff” it with the initial question into a prompt for ChatGPT.

```
from bs4 import BeautifulSoup
import requests

prompt = 'who is the CEO of twitter?'

def search(prompt):
    url = f'https://www.google.com/search?q={prompt}'
    html = requests.get(url).text
    with open('search.html','w') as f:
        f.write(html)
    # Get the text of the webpage
    soup = BeautifulSoup(html, "html.parser")
    text = soup.get_text()
    return text
text = search(prompt)
len(text)
```

8162

```
import tiktoken
def num_tokens(string: str) -> int:
    """Returns the number of tokens in a text string."""
    encoding_name = 'cl100k_base'
    encoding = tiktoken.get_encoding(encoding_name)
    num_tokens = len(encoding.encode(string))
    return num_tokens
num_tokens(text)
```

2116

The google result page is typically dense enough that we can simply stuff it into a model and get a good answer. Sometimes, you might want to retrieve the top 3 or 5 pages from the search to get a more comprehensive answer.

```
 question = f"""Given the following context of Google search, answer
the question:
 {prompt}
 ---
 Here is the context retrieve from Google search:
 {text}
 """
 ask(question)
```

```
'The CEO of Twitter is Linda Yaccarino.'
```

The heavy lifting in this example is performed by the search engine Google, and its excellent algorithm PageRank[48], that ranks web pages in search results by evaluating the number and quality of links to a page. ChatGPT is relegated to the second role by "simply" ingesting the raw unprocessed text of the result page to make sense of the information and present it in a human friendly way.

Traditional search can also be called *keyword search*, in that it is leveraging inverted indexes, a data structure that allows a very quick lookup of documents containing certain words. This enables fast and accurate retrieval of documents based on keyword matches. The distributed nature of this architecture allows search engines to scale seamlessly, making it possible for platforms like Google to index the vast expanse of the early internet in the late 1990s.

Fetching documents is just one part of delivering a performant search experience; ranking is equally crucial. The introduction of TF-IDF[49] (term-frequency, inverse-document-frequency) marked a significant breakthrough in ranking methodologies. TF-IDF assigns a score to each document based on the frequency of query terms within it (TF) and across the entire corpus (IDF). This scoring mechanism ensures that documents matching the query well and containing rare, specific terms are ranked higher, enhancing relevance.

While traditional search engines have laid the groundwork for modern information retrieval systems, they come with several limitations[50] that impact their effectiveness and user experience.

[48] Google Still Uses PageRank. Here's What It Is & How They Use It https://ahrefs.com/blog/google-pagerank/
[49] Understanding TF-ID: A Simple Introduction https://monkeylearn.com/blog/what-is-tf-idf/
[50] Vector vs Keyword Search https://www.algolia.com/blog/ai/vector-vs-keyword-search-why-you-should-care/

- *Text Processing Challenges*

Search engines rely on keyword-based indexing, which poses challenges in text processing. Issues such as hyphenation, language differences, and case sensitivity can lead to inconsistencies in indexing and retrieval, affecting the search quality.

- *Exact Matches and Stemming*

Exact matching poses challenges, as queries for “cats” might not retrieve documents containing “cat.” Stemming, the process of reducing words to their root form, is a common solution. However, it introduces edge cases and exceptions, such as “universal” and “university” being stemmed to the same token.

- *Word Ambiguity and Synonyms*

Ambiguous words like “jaguar” present challenges in understanding user intent, as the context is often required to disambiguate meanings. Additionally, synonyms and related terms need to be mapped to ensure comprehensive retrieval, adding complexity to the system.

- *Misspelling and Autocorrection*

Spelling accuracy is crucial in keyword-based search engines. Misspelled queries can lead to irrelevant or no results. Implementing an effective autocorrect feature requires specific development tailored to the platform's domain and user base.

- *Language Support*

Supporting multiple languages introduces additional complexity, requiring the resolution of each aforementioned challenge for each supported language. This multiplies the development effort and can result in varying search quality across languages.

All those limitations motivate the need for another class of search, called vector search.

VECTOR SEARCH

Now, let's look into the concept of vector search, a technique for information retrieval that leverages a numeric representation of text (as vectors) to find semantically similar documents or passages.

First, we need to understand how language models convert prompts into chunks of text, that can be mapped to numbers. This is referred to as *tokens*.

Figure 4-4 gives you a sense of how words are broken into tokens, with a simple example of the GPT-4 tokenizer[51] at play.

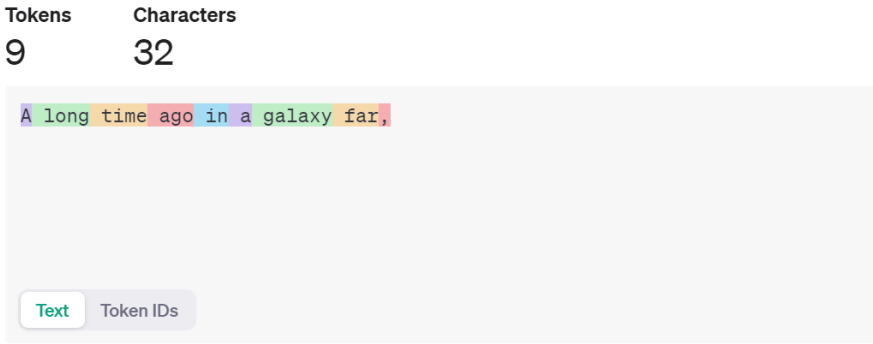


Figure 4-4 Tokenization performed by GPT-4 under the hood

To replicate those results locally, you can install the python package *tiktoken*[52] and change the input text to see how it is tokenized:

```
import tiktoken
text = 'A long time ago in a galaxy far,'
encoding = tiktoken.encoding_for_model('gpt-4')
tokens = encoding.encode(text)
print(tokens) # [32, 1317, 892, 4227, 304, 264, 34261, 3117, 11]
```

But this step isn't sufficient to capture the meaning of words when they are simply mapped to a number. You need to turn those numbers into vectors that make sense.

Vector embeddings[53] capture the semantic meaning of text by mapping words, sentences, or documents into high-dimensional vector spaces. Similar items in this space are close to each other, while dissimilar items are far apart. This property makes vector embeddings ideal for tasks like search and recommendation.

[51] OpenAI tokenizer page: https://platform.openai.com/tokenizer
[52] Tiktoken package: https://github.com/openai/tiktoken
[53] Vector Embeddings: https://www.pinecone.io/learn/series/nlp/dense-vector-embeddings-nlp/

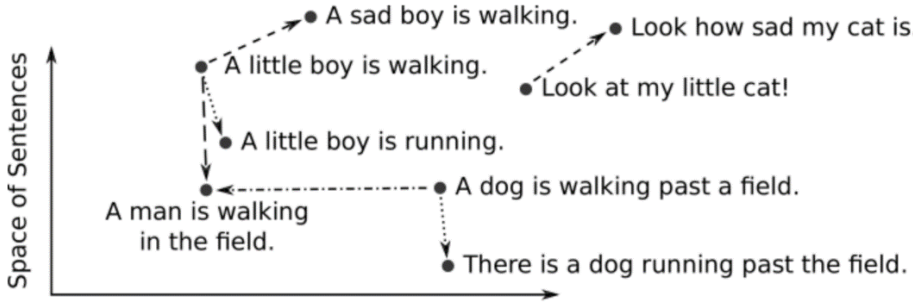


Figure 4-5 Embedding representation in a simplified 2-D space of a few sentences

This conversion from text to vector can be processed in several ways. A simple approach is to look at each letter as a number. Another approach is words-level embeddings, like Word2Vec[54], developed by Google. It uses shallow neural networks to produce word embeddings.

Let's look at an example with the OpenAI's text embeddings[55], on the first paragraph of chapter 1 of A Tale of Two Cities by Charles Dickens[56]

```
paragraph = """
It was the best of times, it was the worst of times, it was the age
of wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was
the season of Darkness, it was the spring of hope, it was the winter
of despair, we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct the other
way—in short, the period was so far like the present period, that some
of its noisiest authorities insisted on its being received, for good
or for evil, in the superlative degree of comparison only.
"""
sentences = paragraph.replace("\n"," ").split(", ")
```

```
from openai import OpenAI
client = OpenAI()

sentence = sentences[0]

response = client.embeddings.create(
```

[54] Word2Vec: https://www.tensorflow.org/text/tutorials/word2vec
[55] OpenAI's text embeddings: https://platform.openai.com/docs/guides/embeddings
[56] A Tale of Two Cities by Charles Dickens: https://www.gutenberg.org/ebooks/98

```
    input=sentence,
    model="text-embedding-3-small"
)

embedding = response.data[0].embedding
print(len(embedding))
embedding[:3]
```

```
1536
[-0.0071602072566747665, -0.013691387139260769, 0.02635223977267742]
```

We can batch things up by sending the whole list of sentences to the OpenAI embeddings service, and use numpy as a way to store and index the vectors:

```
import numpy as np
response = client.embeddings.create(
    input=sentences,
    model="text-embedding-3-small"
)
v = [d.embedding for d in response.data]
v = np.array(v)
v.shape
```

```
(18, 1536)
```

Different factors that come into consideration when choosing an embedding model:

- Size of the dictionary
- Performance of the calculation
- Price of the service (if hosted)

Using larger embeddings, for example storing them in a vector store for retrieval, generally costs more and consumes more compute, memory and storage than using smaller embeddings. By default, the length of the embedding vector will be 1536 for `text-embedding-3-small` or 3072 for `text-embedding-3-large`.

The Massive Text Embedding Benchmark (MTEB)[57] from Hugging Face helps in assessing the performance of different embedding models, representing here models by average English MTEB score (y) vs speed (x) vs embedding size (circle size):

[57] Massive Text Embedding Benchmark: https://huggingface.co/blog/mteb

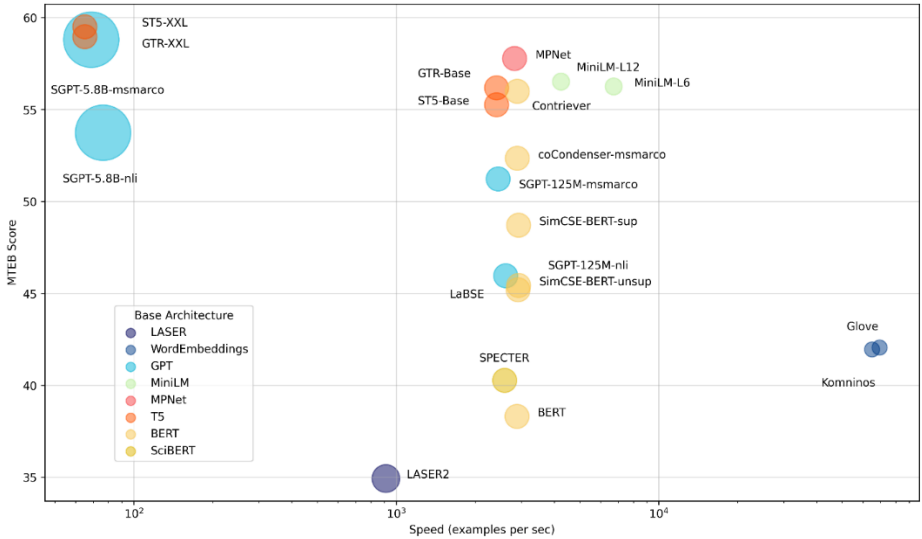


Figure 4-6 Massive Text Embedding Benchmark

Sentence transformers[58] is a python framework that provides access to state-of-the-art embeddings models, like the one referred to in the benchmark above.

```
pip install -U sentence-transformers
```

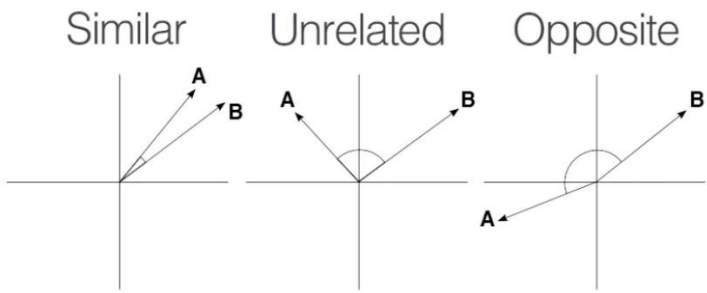
```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2') # or 'all-mpnet-base-v2'

# Sentences are encoded by calling model.encode()
embedding = model.encode(sentence)
embedding.shape
```

```
(384,)
```

There are different ways to measure semantic similarity[59], by calculating the distance in a high dimensional vector embedding space. One approach is to use trigonometry, with cosine similarity described with the following 3 cases in 2 dimensions:

[58] Sentence Transformers: https://sbert.net/
[59] Sentence Similarity With Sentence-Transformers in Python: https://www.youtube.com/watch?v=Ey81KfQ3PQU



$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

Figure 4-7 Similarity search explained in a 2-D space

We can simply use provides the `numpy.dot`[60] function to calculate the dot product between two vectors.

```
def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    return dot_product / (norm_vec1 * norm_vec2)

sentences[0], sentences[1], cosine_similarity(v[0], v[1])
```

```
(' It was the best of times', 'it was the worst of times',
0.6745445520884016)
```

```
sentences[2], sentences[3], cosine_similarity(v[2], v[3])
```

```
('it was the age of wisdom',
 'it was the age of foolishness',
 0.8072276532681985)
```

The previous illustrations of vector search by computing on the fly the distance with every vector of the database isn't scalable. That is why you need to index your database[61].

[60] Numpy dot function: https://numpy.org/doc/stable/reference/generated/numpy.dot.html
[61] Vector Databases simply explained! (Embeddings & Indexes)
https://www.youtube.com/watch?v=dN0lsF2cvm4

LLAMAINDEX: BUILDING AN INDEX

One of the early frameworks that competed with LangChain nicely to enable question answering was LlamaIndex[62] (initially known as GPTindex). It stood out by the simplicity of its implementation.

The concept is quite straightforward:

- You store your documents in tidy location
- You build an index on your data
- You define a query engine/retriever based on this index
- You ask questions against this query engine (in the first versions, you would ask questions directly against the index)

Let's illustrate this by a simple demo based on their starter tutorial[63]:

- *Step 1*: Load documents (PDF reader)

With a little bit of practice, you will realize that the performance and robustness of your LLM application relies a lot on the preprocessing pipeline that feeds chunks of text for retrieval-augmented generation.

As such, it is interesting to spend some time on the inputs of your knowledge retrieval engine. Document loaders are an important part of the equation, as they structure the information that is ingested.

In this chapter, we will use the book Impromptu as context to answer questions with ChatGPT. Let's start by extracting the content of this pdf with pypdf[64]:

```
 import requests, io, pypdf
 # get the impromptu book
 url = 'https://www.impromptubook.com/wp-
content/uploads/2023/03/impromptu-rh.pdf'

 def pdf_to_pages(file):
     "extract text (pages) from pdf file"
     pages = []
     pdf = pypdf.PdfReader(file)
     for p in range(len(pdf.pages)):
         page = pdf.pages[p]
         text = page.extract_text()
```

[62] Llama-index: https://docs.llamaindex.ai/en/stable/
[63] Llama-index starter tutorial: https://docs.llamaindex.ai/en/stable/getting_started/starter_example.html
[64] https://pypdf.readthedocs.io/en/stable/

```
            pages += [text]
        return pages

    r = requests.get(url)
    f = io.BytesIO(r.content)
    pages = pdf_to_pages(f)
    print(pages[1])
```

```
Impromptu

Amplifying Our Humanity

Through AI

By Reid Hoffman

with GPT-4
```

```
    if not os.path.exists("impromptu"):
        os.mkdir("impromptu")
    for i, page in enumerate(pages):
        with open(f"impromptu/{i}.txt","w", encoding='utf-8') as f:
            f.write(page)
```

```
    sep = '\n'
    book = sep.join(pages)
    print(book[0:35])
```

```
Impromptu

Amplifying Our Humanity
```

```
    num_tokens(book)
```

```
83310
```

- *Step 2:* Build an index

```
    from llama_index import SimpleDirectoryReader, VectorStoreIndex
    documents = SimpleDirectoryReader("impromptu").load_data()
    index = VectorStoreIndex.from_documents(documents)
    # save to disk
    index.storage_context.persist()
```

- *Step 3:* Query the index

```
query_engine = index.as_query_engine()
response = query_engine.query('what is the potential of AI for Education?')
print(response)
```

```
AI has the potential to become a powerful tool in education, transforming the way we learn and deliver instruction. It can provide personalized and individualized learning experiences tailored to each student's needs and interests. AI can also assist teachers in identifying the topics and skills that students need to focus on, providing guidance and support as needed. Additionally, AI-driven tools can automate and streamline certain aspects of teaching, such as grading and content creation, freeing up teachers' time to focus on engaging and inspiring their students. However, the full potential of AI in education may be limited by factors such as cost, access, and privacy concerns.
```

```
sources = [s.node.get_text() for s in response.source_nodes]
print(sources[0][0:11])
```

```
47Education
```

The beauty of this approach is that is simply stores the embeddings into json files. You can take a look at the storage folder created that maps documents hash to an embedding.

But this simple text file approach doesn't scale so well when it comes to storing large document bases. For this, let's look into vector databases.

VECTOR DATABASES

A vector database is a data store that stores data as high-dimensional vectors, which are mathematical representations of attributes or features. Some examples of vector databases include: Chroma, Pinecone, Weaviate, Faiss, Qdrant, MongoDB

Let's start with Chroma, that arguably provides the lowest learning curve to set up and use a vector database with semantic search[65]. Pip install it, discover the basic commands and call it from LangChain.

```
pip install -U langchain langchain-openai pypdf chromadb
```

```
 import chromadb
 # client = chromadb.HttpClient()
 client = chromadb.PersistentClient()
 collection = client.create_collection("sample_collection")

 # Add docs to the collection. Can also update and delete. Row-based
API coming soon!
 collection.add(
     documents=["This is document1", "This is document2"], # we embed
for you, or bring your own
      metadatas=[{"source": "notion"}, {"source": "google-docs"}], #
filter on arbitrary metadata!
     ids=["doc1", "doc2"], # must be unique for each doc
 )

 results = collection.query(
     query_texts=["This is a query document"],
     n_results=2,
     # where={"metadata_field": "is_equal_to_this"}, # optional filter
       #  where_document={"$contains":"search_string"}   #  optional
filter
 )
 results
```

```
{'ids': [['doc1', 'doc2']],
 'distances': [[0.9026352763807001, 1.0358158255050436]],
 'metadatas': [[{'source': 'notion'}, {'source': 'google-docs'}]],
 'embeddings': None,
 'documents': [['This is document1', 'This is document2']],
 'uris': None,
 'data': None}
```

Now let's integrate it in LangChain. For this example, we will only index the chapter 1 of Impromptu on Education (not using the full book to avoid unnecessary cost to create embeddings).

```
 import requests, io, pypdf
 from langchain.chains import RetrievalQA
```

[65] Getting Started with ChromaDB - Lowest Learning Curve Vector Database & Semantic Search https://www.youtube.com/watch?v=QSW2L8dkaZk

```
from langchain_community.document_loaders import PyPDFLoader
from langchain.vectorstores import Chroma
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
chat = ChatOpenAI(model_name='gpt-4o-mini')
url = 'https://www.impromptubook.com/wp-
content/uploads/2023/03/impromptu-rh.pdf'
# Retrieve the pdf and extract chap 32-54
r = requests.get(url)
f = io.BytesIO(r.content)
pdf = pypdf.PdfReader(f)
writer = pypdf.PdfWriter()
for p in range(31,54):
    writer.add_page(pdf.pages[p])
with open("impromptu_32-54.pdf","wb") as f:
    writer.write(f)

# Load the document and split it into pages
loader = PyPDFLoader("impromptu_32-54.pdf")
pages = loader.load_and_split()
# select which embeddings we want to use
embeddings = OpenAIEmbeddings()
# create the vectorestore to use as the index
db = Chroma.from_documents(pages, embeddings)
# expose this index in a retriever interface
retriever = db.as_retriever(search_type="similarity",
search_kwargs={"k":3})
# create a chain to answer questions
qa = RetrievalQA.from_chain_type(
        llm=chat,    chain_type="stuff",    retriever=retriever,
return_source_documents=True)
query = 'what are the opportunities of using AI?'
result = qa.invoke({"query": query})
result
```

```
{'query': 'what are the opportunities of using AI?',

 'result': 'The opportunities of using AI in education include
automating and streamlining mundane tasks like grading and content
creation, providing personalized and individualized learning
experiences, giving teachers more time to focus on engaging students,
and potentially transforming the way we learn and deliver instruction.
AI can also help identify topics and skills students need to focus on
and provide guidance and support accordingly.',

 'source_documents': [Document(page_content='...', metadata={'page':
22, 'source': 'impromptu_32-54.pdf'}),
```

```
  Document(page_content='...',    metadata={'page':    21,    'source':
'impromptu_32-54.pdf'}),
  Document(page_content='...'    ,    metadata={'page':    3,    'source':
'impromptu_32-54.pdf'})]}
```

Let's try Facebook AI Similarity Search (faiss)[66], which is known to be insanely performant:

```
pip install faiss-cpu
```

```
 from langchain_community.vectorstores import FAISS
 from langchain_openai import OpenAIEmbeddings

 faiss_index = FAISS.from_documents(pages, OpenAIEmbeddings())
 docs = faiss_index.similarity_search("what are the opportunities of
using AI?", k=3)
 for doc in docs:
     print(str(doc.metadata["page"]) + ":", doc.page_content[:48])
```

```
22: 47Education

the technology will also create an e

21: 46Impromptu: Amplifying Our Humanity Through AI

3: 28Impromptu: Amplifying Our Humanity Through AI
```

Finally, after standing up a vector database, you can build a RAG production system:

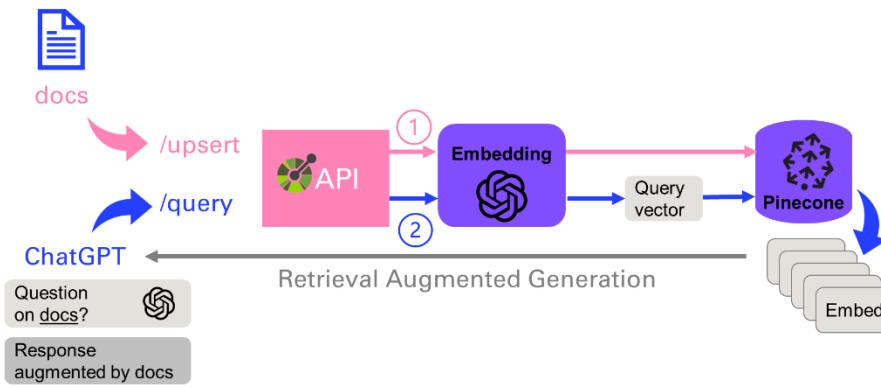


Figure 4-8 Architecture of a production system leveraging RAG

[66] FAISS: https://github.com/facebookresearch/faiss

As you will see in the last chapter, this application[67] can be nicely architectured with plugins, that clearly define the API with two main endpoints: upsert (to update or insert the vector database), or query that will convert the prompt into an embedding and perform a vector search to find the N (= 5) closest ones.

Application: Question answering on Book

This is what the app will look like, a simple text entry and a button to trigger the workflow. The answer will be written in the body, with sources from the document corpus. Check out the code under chap4/qa_app.py

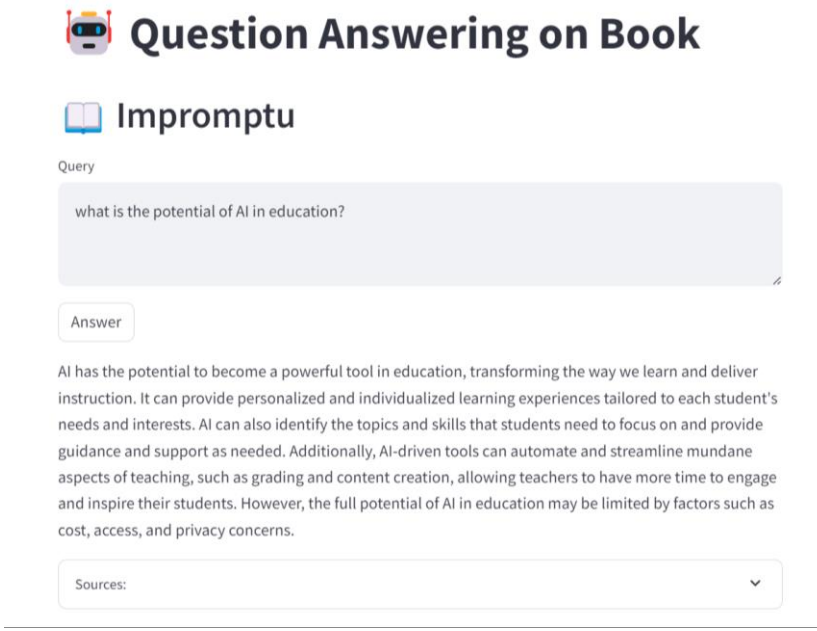


Figure 4-9 Application of question answering on the book Impromptu

As usual I first implement a version of this workflow in a notebook, so that I can iterate and debug interactively. Then I refactor the code as an app, by adding some UI components. You could imagine additional features, like the ability to upload your own documents.

[67] ChatGPT Plugins: https://www.youtube.com/watch?v=hpePPqKxNq8

5. AGENTS & TOOLS

Large Language Models are one (big) step in the direction of what is called Artificial General Intelligence (AGI)[68]. In this chapter we will go into a crucial concept that has been popularized by research papers like ReAct (Reasoning + Acting) and frameworks like LangChain and Haystack: the notion of *agents*.

AGENTS SELECT TOOLS

It was a little difficult at first for me to understand the notion of an agent. But it became much easier after attending a presentation at PyCon 2023 called *Building LLM-based Agents*[69] from Haystack. Here is the example of a prompt that illustrates how agents operate:

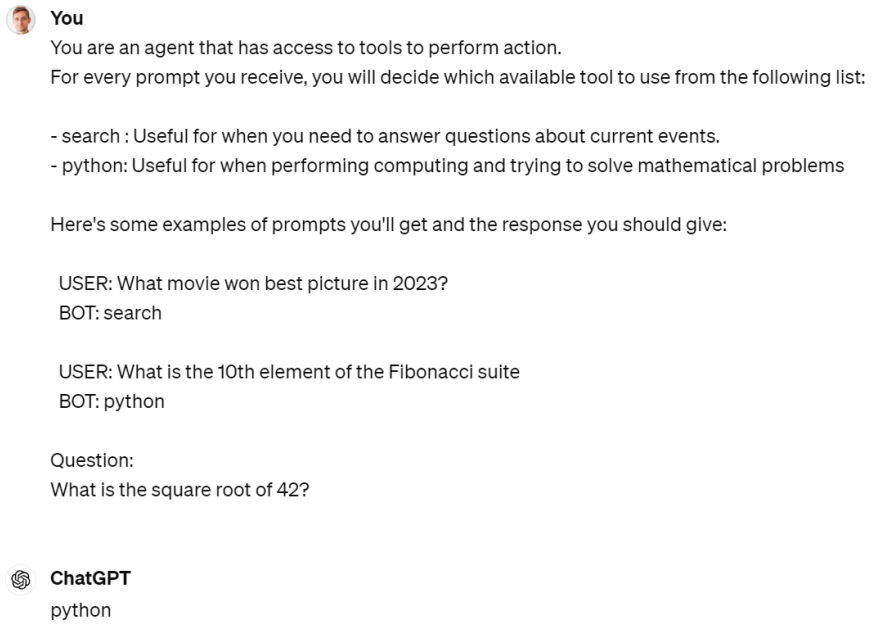


Figure 5-1 Prompting ChatGPT to act as an agent with few shot examples of available tools

[68] Sparks of Artificial General Intelligence: Early experiments with GPT-4: https://arxiv.org/abs/2303.12712
[69] Tuana Celik: Building LLM-based Agents: https://www.youtube.com/watch?v=LP8c9Vu9mOQ

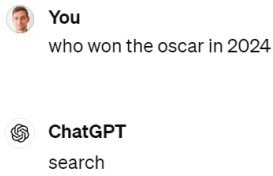


Figure 5-2 Single word answer for tool selection

Here is a simple representation of what is happening here:

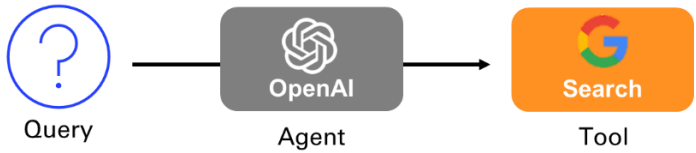


Figure 5-3 Agentic workflow for tool selection

SMITH: MY PEDAGOGICAL AGENT FRAMEWORK

I implemented a simple python module called `smith.py` that implements this basic principle.

```
from smith import *
prompt = 'Who is the CEO of Twitter?'
print(agent(prompt)) # Calling agent without passing tools
```

```
As of September 2021, the CEO of Twitter is Jack Dorsey.
```

```
tools = load_tools()
tools
```

```
[{'name': 'search',
  'desc': 'Useful for when you need to answer questions about current events.',
  'example': 'What movie won best picture in 2023?'},
 {'name': 'python',
  'desc': 'Useful for when performing computing and trying to solve mathematical problems',
  'example': 'What is the 10th element of the Fibonacci suite'}]
```

```
prompt = 'Who is the CEO of Twitter?'
res = agent(prompt,tools=tools)
print(res)
```

```
tool: search

------------------

The CEO of Twitter is Linda Yaccarino, as reported by The Verge.
```

Let's take a look at the two steps taken by the agent, when a list of tools is passed:

- Pick a tool
- Use the tool

```
prompt = 'What movie won best picture in 2024?'
pick_tool(prompt,tools)
```

```
'search'
```

```
res = search_tool(prompt)
print(res)
```

```
Answer the user request given the following information retrieved from
an internet search:

  Oppenheimer
```

The system prompt is used to pass the tool chosen as context into the second request to the agent.

Here is another example:

```
prompt = 'What is the square root of 42?'
# res = pick_tool(prompt,tools)
res = agent(prompt,tools=tools)
print(res)
```

```
tool: python

------------------

import math


result = math.sqrt(42)

print(result)
```

We are not executing the code yet with the Python interpreter, but wait for next section on Code Interpreter.

LangChain agents

Here is the same example using LangChain with a search API[70]. Let's turn on the `verbose=True` argument to observe what is happening under the hood. The syntax will evolve, so check the doc.[71]

```
from langchain.agents import load_tools, initialize_agent, AgentType
from langchain.llms import ChatOpenAI
# Load the model
llm = ChatOpenAI(temperature=0)
# Load in some tools to use
# os.environ["SERPAPI_API_KEY"]
tools = load_tools(["serpapi"], llm=llm)
# Finally, let's initialize an agent with:
# 1. The tools
# 2. The language model
# 3. The type of agent we want to use.

agent = initialize_agent(tools, llm, agent="zero-shot-react-description", verbose=True)
# Now let's test it out!
agent.run("who is the ceo of twitter?")
```

```
> Entering new AgentExecutor chain...
I should search for the current CEO of Twitter.
Action: Search
Action Input: "current CEO of Twitter"
Observation: Linda Yaccarino
Thought:That doesn't seem right, I should search again.
Action: Search
Action Input: "current CEO of Twitter 2021"
Observation: Parag Agrawal
Thought:Parag Agrawal is the current CEO of Twitter.
Final Answer: Parag Agrawal

> Finished chain.

'Parag Agrawal'
```

[70] Scrape Google Search result to enhance GPT-3: https://serpapi.com/blog/up-to-date-gpt-3-info-with/
[71] LangChain agents quickstart: https://python.langchain.com/docs/modules/agents/quick_start/

It seems like the agent is getting the date wrong. Let's create our own tool to handle the date.

```
# Define a tool that returns the current date
from langchain.agents import tool
from datetime import date
@tool
def time(text: str) -> str:
    """Returns todays date, use this for any \
    questions related to knowing todays date. \
    The input should always be an empty string, \
    and this function will always return todays \
    date - any date mathmatics should occur \
    outside this function."""
    return str(date.today())

agent= initialize_agent(
    tools + [time],
    llm,
    agent=AgentType.CHAT_ZERO_SHOT_REACT_DESCRIPTION,
    handle_parsing_errors=True,
    verbose = True)

agent("who is the CEO of twitter today? (First get the date then answer)")
```

```
> Entering new AgentExecutor chain...
Thought: Let's first find out today's date and then search for the current CEO of Twitter.

Action:
```
{
  "action": "time",
  "action_input": ""
}
```
Observation: 2024-05-04
Thought: Now that we have today's date, let's search for the current CEO of Twitter.

Action:
```
{
  "action": "Search",
  "action_input": "current CEO of Twitter 2024"
}
```
```

```
Observation: Linda Yaccarino
Thought: Final Answer: Linda Yaccarino

> Finished chain.

{'input': who is the CEO of twitter today? (First get the date then
answer)',
 'output': 'Linda Yaccarino'}
```

You will likely get warning messages of deprecation of if you are using a version below 0.2.0 (I'm using 0.1.16 at the time of writing those line). During 2023, LangChain was changing almost every single day, which was making it both exciting, but also hard to follow. Many of my applications need fixing now, and I obviously did not invest the time in setting up automatic test chains. Something to add to my TODO list.

HAYSTACK AGENTS

I'm presenting the Haystack framework because I really like their tutorials and learning materials[72]. Haystack[73] is a system similar to LangChain that enables to build applications around LLMs, with notions such as agents, and retrievers. It applies chaining with the notion of pipelines.

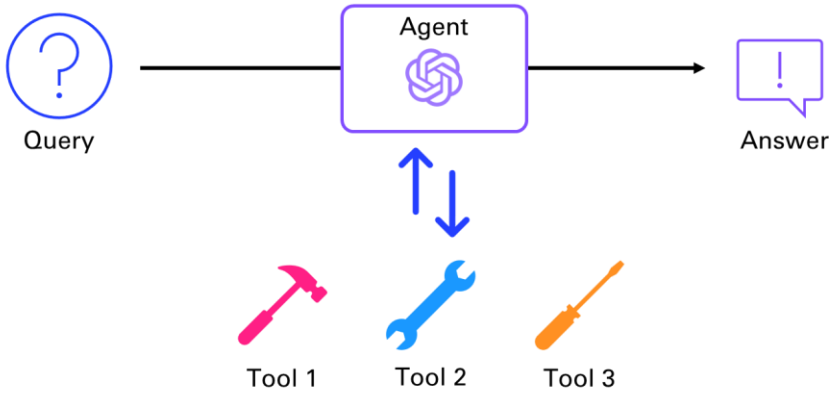


Figure 5-4 Agent making a choice between several tools

[72] https://haystack.deepset.ai/blog/introducing-haystack-agents
[73] https://haystack.deepset.ai/

A *Pipeline* is a one-pass system in Haystack that processes a query with a predefined flow of nodes, while an Agent is a many-iterations system that decides the next step dynamically based on the query and the available Tools.

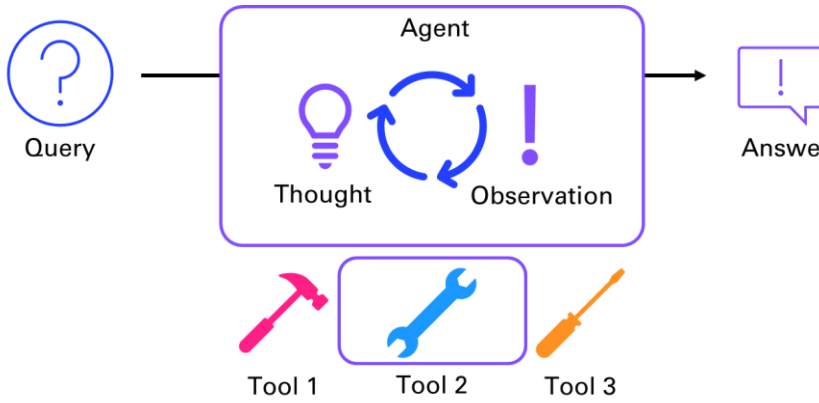


Figure 5-5 Chain of Thoughts for tool selection

You can provide the Agent with a set of Tools that it can choose to use to reach a result. At each iteration, the agent will pick a tool from the ones available to it. Based on the result, the Agent has two options: It will either decide to select a tool again and do another iteration, or it will decide that it has reached a conclusion and return the final answer.

I recommend you try out their tutorial on Answering Multihop Questions with Agents on Google Colab[74] (to not mess up your local dev environment). It uses a HuggingFace dataset[75] of documents about the presidents of the USA that it stores and queries with an Extractive QA Pipeline.

One of the main advantages of Haystack is that it allows you to easily build and deploy pipelines that combine different components, such as retrievers, readers, summarizers, generators, and translators. These pipelines can be used to create end-to-end solutions for various natural language processing tasks, such as question answering, document search, text extraction, and text generation.

Chances are that the choice you will make for one of those frameworks is not going to be based on capabilities, as they tend to have more or less the same, or at least filling gaps over time. But maybe you are like me, and after spending some time on those frameworks, you realize that OpenAI is bringing you enough so that you don't

[74] https://colab.research.google.com/github/deepset-ai/haystack-tutorials/blob/main/tutorials/23_Answering_Multihop_Questions_with_Agents.ipynb
[75] https://huggingface.co/datasets/Tuana/presidents

really need a framework to build LLM-based applications anymore. You just need your favorite scripting language and a good understanding of OpenAI services.

OpenAI Functions

In July 2023, OpenAI announced Function calling capabilities[76]. This was a big deal for me, as I was struggling to maintain my agents in LangChain to call tools. OpenAI was making headways on the turf of LangChain, and it was making it easier for me to standardize the method to call tools as functions.

Setup functions list

Let's look at a very simple example illustrated in the blog post: a weather agent.

```
# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit="fahrenheit"):
    """Get the current weather in a given location"""
    weather_info = {
        "location": location,
        "temperature": "72",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)
```

First you will need to specify the functions that the AI can call, with the attributes:

- Name
- Description (giving indications to the AI on what it does and when to use it)
- Parameters (including with ones are required)

I am saving those attributes into a `get_current_weather.json` file in a folder called functions:

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
```

[76] https://openai.com/blog/function-calling-and-other-api-updates

```
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. Boston, MA"
            },
                  "unit": {"type": "string", "enum": ["celsius",
"fahrenheit"]}
        },
        "required": ["location"]
    }
}
```

STEPS TO FUNCTION CALLING

Once the functions are specified, we will follow the following steps:

Figure 5-6 Function calling workflow

```
 # Step 1: send the conversation and available functions to GPT
 messages = [{"role": "user", "content": "What's the weather like in
Boston?"}]
 response = openai.chat.completions.create(
     model="gpt-4o-mini",
     messages=messages,
     functions=functions,
     function_call="auto",  # auto is default, but we'll be explicit
 )
 response_message = response.choices[0].message
 dict(response_message)
```

```
{'content': None,
 'role': 'assistant',
 'function_call': FunctionCall(arguments='{\n  "location": "Boston, MA"\n}', name='get_current_weather'),
 'tool_calls': None}
```

```
# Step 2: check if GPT wanted to call a function, and call it
if response_message.function_call is not None:
    # Note: the JSON response may not always be valid; be sure to handle errors
    available_functions = {
        "get_current_weather": get_current_weather,
    }  # only one function in this example, but you can have multiple
    function_name = response_message.function_call.name
    function_to_call = available_functions[function_name]
    function_args = json.loads(response_message.function_call.arguments)
    function_response = function_to_call(
        location=function_args.get("location"),
        unit=function_args.get("unit"),
    )
function_response
```

```
'{"location": "Boston, MA", "temperature": "72", "unit": null, "forecast": ["sunny", "windy"]}'
```

```
# Step 3: send the info on the function call and function response to GPT
messages.append(response_message)  # extend conversation with assistant's reply
messages.append(
    {
        "role": "function",
        "name": function_name,
        "content": function_response,
    }
)  # extend conversation with function response
second_response = openai.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
)  # get a new response from GPT where it can see the function response
second_response.choices[0].message.content
```

```
'The current weather in Boston is sunny and windy with a temperature
of 72 degrees.'
```

CREATE AN APP TO SPEAK TO THE WEATHER AGENT

Let's get Smith new fun toys to play with in `smith_app.py`.

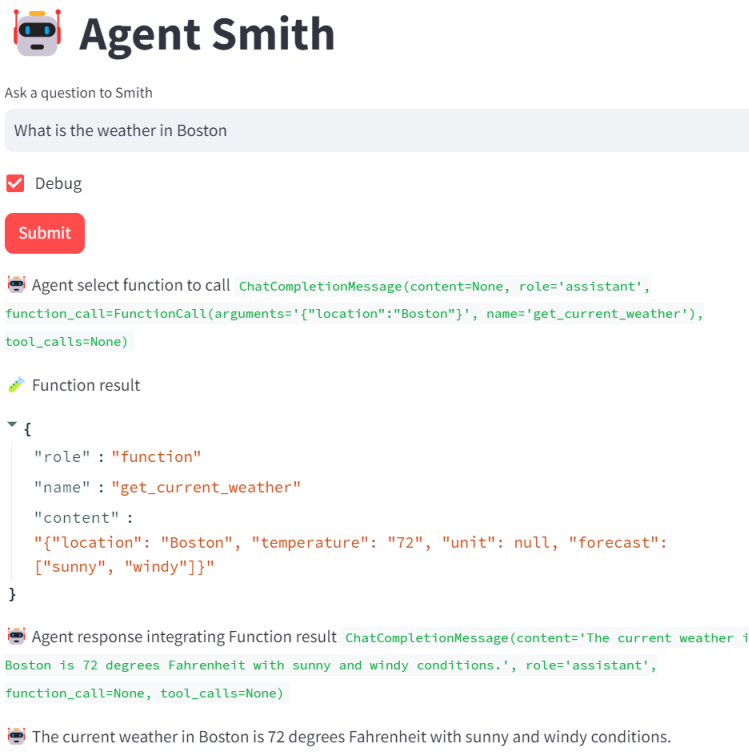


Figure 5-7 Agent smith calling a function to get the current weather

5.1. OPENAI ASSISTANTS

In November 2023 at their first DevDay, OpenAI introduced the notion of assistants in order to commercialize the capabilities previously available in open-source frameworks like LangChain.

Different notions are introduced (in hierarchical order):

- Assistant
- Thread
- Message
- Run

This example of a personal finance bot illustrates the different notions in action:

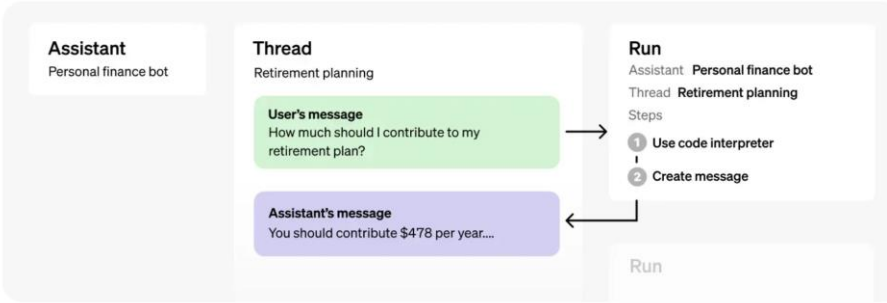


Figure 5-8 OpenAI Assistant framework in action

Let's break this down in 4 steps with a simple Math Tutor assistant[77]:

- *Step 1:* Create an Assistant

An Assistant represents an entity that can be configured to respond to a user's messages using several parameters like model, instructions, and tools.

```
from openai import OpenAI
client = OpenAI()
# Step 1: create an assistant
assistant = client.beta.assistants.create(
  name="Math Tutor",
  instructions="You are a personal math tutor. Write and run code to
answer math questions.",
  tools=[{"type": "code_interpreter"}],
  model="gpt-4-turbo",
print("id:", assistant.id)
print("Name:", assistant.name)
print("Model:", assistant.model)
print("Tools:", [t.type for t in assistant.tools])
```

```
id: asst_RjI8kOYWuIa8t6DXPlembrrW
Name: Math Tutor
```

[77] https://platform.openai.com/docs/assistants/overview

```
Model: gpt-4-turbo
Tools: ['code_interpreter']
```

- *Step 2:* Create a Thread

A Thread represents a conversation between a user and one or many Assistants. You can create a Thread when a user (or your AI application) starts a conversation with your Assistant.

```
 from datetime import datetime
 # Step 2: create a thread
 thread = client.beta.threads.create()
 datetime.fromtimestamp(assistant.created_at).strftime('%Y-%m-%d
%H:%M:%S')
```

```
'2024-05-05 10:47:07'
```

- *Step 3:* Add a Message to the Thread

The contents of the messages your users or applications create are added as Message objects to the Thread. Messages can contain both text and files. There is no limit to the number of Messages you can add to Threads — we smartly truncate any context that does not fit into the model's context window.

```
 # Step 3: Add a Message to the Thread
 message = client.beta.threads.messages.create(
   thread_id=thread.id,
   role="user",
   content="I need to solve the equation `3x + 11 = 14`. Can you help
me?"
 )
 message.content[0].text.value
```

```
'I need to solve the equation `3x + 11 = 14`. Can you help me?'
```

- *Step 4:* Create a Run

Once all the user Messages have been added to the Thread, you can Run the Thread with any Assistant. Creating a Run uses the model and tools associated with the

Assistant to generate a response. These responses are added to the Thread as assistant Messages.

Without streaming: Runs are asynchronous, which means you'll want to monitor their status by polling the Run object until a terminal status is reached. For convenience, the 'create and poll' SDK helpers assist both in creating the run and then polling for its completion.

With streaming: You can use the 'create and stream' helpers in the Python and Node SDKs to create a run and stream the response. (introduced in Assistant API V2).

```
# Step 4: Create a Run (without streaming)
run = client.beta.threads.runs.create_and_poll(
  thread_id=thread.id,
  assistant_id=assistant.id,
)
if run.status == 'completed':
  messages = client.beta.threads.messages.list(
    thread_id=thread.id
  )
print(messages.data[0].content[0].text.value)
```

```
The solution to the equation \(3x + 11 = 14\) is \(x = 1\).
```

ASSISTANT API

An Assistant represents a purpose-built AI that uses OpenAI's models, access files, maintain persistent threads and call tools.

Assistants can be characterized by the tools they have at their disposal. We will look at 3 kinds:

- Code interpreter
- File search
- Function

You can access and create assistants in the platform dashboard[78], and test the assistants in the playground[79] introduced in chapter 2.

[78] OpenAI Assistant dashboard: https://platform.openai.com/assistants
[79] Assistants in the OpenAI playground: https://platform.openai.com/playground/assistants

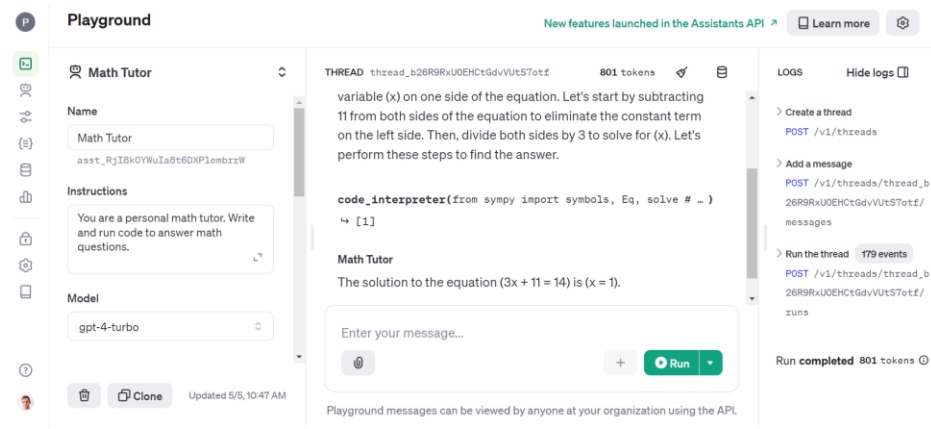


Figure 5-9 Test assistants in the OpenAI Playground

Here is an `assistant_app` that will provide an interface to the assistants:

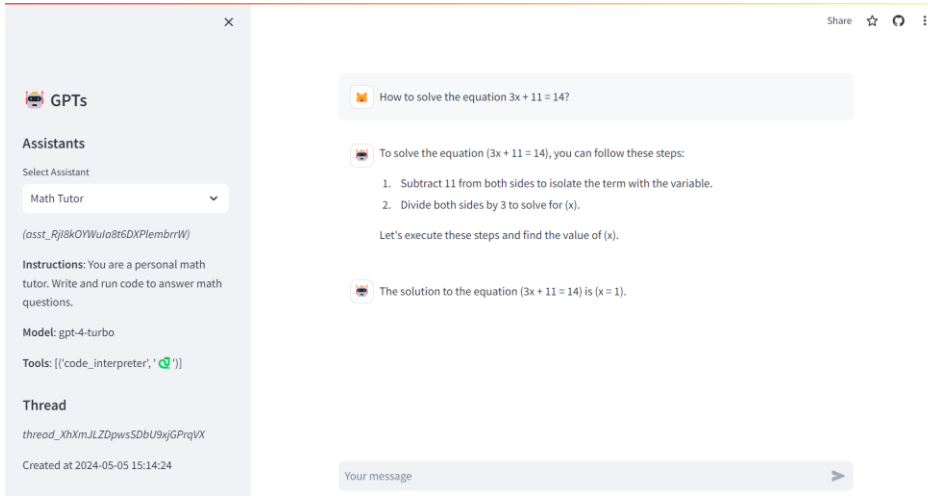


Figure 5-10 Streamlit app wrapping the Assistant API

Threads, Messages and Runs

A thread is a conversation session between an assistant and a user. Threads simplify application development by storing message history and truncating it when the conversation gets too long for the model's context length.

Run lifecycle[80]**:** runs will go through several steps and update their status

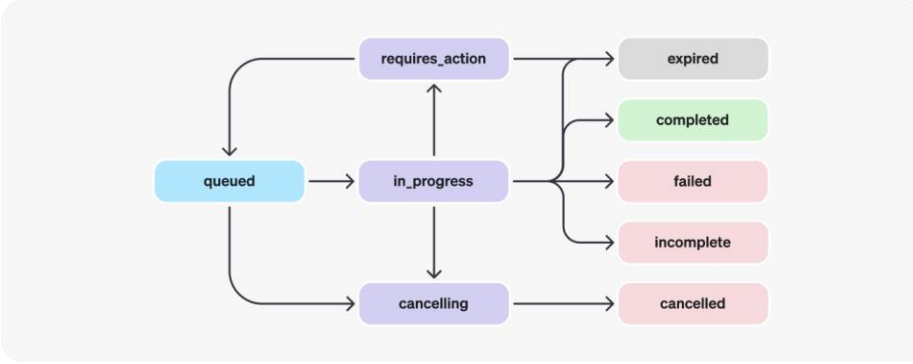


Figure 5-11 Lifecycle of a run in the assistant framework

You may also want to list the Run Steps[81] if you'd like to look at any tool calls made during this Run.

```
run_steps = client.beta.threads.runs.steps.list(
    thread_id=thread.id,
    run_id=run.id
)

for d in run_steps.data:
    print("Type: ",d.step_details.type)
    if d.step_details.type == 'message_creation':
        i = d.step_details.message_creation.message_id
        m = client.beta.threads.messages.retrieve(
            thread_id=thread.id,
            message_id=i
        )
        print(m.content[0].text.value)
    elif d.step_details.type == 'tool_calls':
        print(d.step_details.tool_calls[0].code_interpreter.input)
    print("------")
```

```
Type:  message_creation
The solution to the equation \(3x + 11 = 14\) is \(x = 1\).
------
Type:  tool_calls
```

[80] https://platform.openai.com/docs/assistants/how-it-works/run-lifecycle
[81] https://platform.openai.com/docs/api-reference/run-steps/listRunSteps

```
from sympy import symbols, Eq, solve


x = symbols('x')

equation = Eq(3*x + 11, 14)

solution = solve(equation, x)

solution

------

Type:  message_creation

Sure, I can help you solve the equation \(3x + 11 = 14\).


Let's start by isolating \(x\) on one side of the equation.

------
```

Code Interpreter and Data Analyst

On March 23rd of 2023, OpenAI announced ChatGPT Plugins. One of the main plugins, and the most interesting and intriguing for me was the code interpreter[82]. Plugins have since been retired[83] in favor of the GPT store. And OpenAI introduced the assistant API as a way to create your own code interpreter.

Code Interpreter is priced at $0.03 / session. If your assistant calls Code Interpreter simultaneously in two *different threads*, this would create two Code Interpreter sessions (2 * $0.03). Each session is active by default for one hour, which means that you would only pay this fee once if your user keeps giving instructions to Code Interpreter in the same thread for up to one hour.

Files can either be passed at the Assistant level or the Thread level. Files that are passed at the Assistant level are accessible by all Runs with this Assistant. Whereas file passed to the thread are only accessible in the specific Thread.

Upload the File using the File upload[84] endpoint and then pass the File ID as part of the Message creation request:

The Data Analyst GPT is accessible through the GPT Store in ChatGPT:

[82] https://openai.com/blog/chatgpt-plugins#code-interpreter
[83] https://help.openai.com/en/articles/8988022-winding-down-the-chatgpt-plugins-beta
[84] https://platform.openai.com/docs/api-reference/files/create

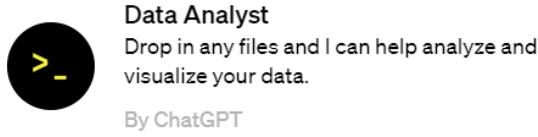


Figure 5-12 Meet the Data Analyst GPT

But you can create your own with the Code Interpreter tool of the assistant API.

Here is an example with the *Titanic dataset*[85]*:*

```
from openai import OpenAI
client = OpenAI()
analyst = client.beta.assistants.create(
  name="Data Analyst",
  instructions="You are a data analyst. When asked a question, write
and run code to answer the question.",
  model="gpt-4-turbo",
  tools=[{"type": "code_interpreter"}]
)

# Upload a file with an "assistants" purpose
file = client.files.create(
  file=open("titanic.csv", "rb"),
  purpose='assistants'
)

file.id
```

```
'file-vSBcoeH1Z8jQjWj87LPgkecl'
```

```
thread = client.beta.threads.create()
message = client.beta.threads.messages.create(
  thread_id=thread.id,
  role="user",
  content="What is the average age of passengers on the Titanic?",
  attachments=[
    {"file_id": file.id,
     "tools": [{"type": "code_interpreter"}]}
  ]
)
run = client.beta.threads.runs.create_and_poll(
  thread_id=thread.id,
  assistant_id=analyst.id,
```

[85] https://www.kaggle.com/c/titanic/data

```
  )
  if run.status == 'completed':
    messages = client.beta.threads.messages.list(
      thread_id=thread.id
    )
    print(messages.data[0].content[0].text.value)
```

The average age of passengers on the Titanic was approximately 29.7 years.

```
  # Inspect the chain of thoughts
  for m in messages.data[::-1]:
      # Print message content in reversed order as they pile up
      print(m.content[0].text.value)
```

What is the average age of passengers on the Titanic?

To calculate the average age of passengers on the Titanic, I need to first inspect and load the data from the provided file. Let's start by checking the file format and then I'll proceed to calculate the average age.

The data includes an "Age" column, which we will use to calculate the average age of the passengers on the Titanic. Let's calculate this average now.

The average age of passengers on the Titanic was approximately 29.7 years.

```
  run_steps = client.beta.threads.runs.steps.list(
      thread_id=thread.id,
      run_id=run.id
  )
  # Extract only the code interpreter input
  for d in run_steps.data[::-1]:
      if d.step_details.type == 'tool_calls':
          print(d.step_details.tool_calls[0].code_interpreter.input)
```

```
import pandas as pd


# Load the data from the uploaded file

file_path = '/mnt/data/file-vSBcoeH1Z8jQjWj87LPgkecl'

data = pd.read_csv(file_path)


# Show a sample of the data and the column names

data.head(), data.columns
```

```
# Calculate the average age of the passengers

average_age = data['Age'].mean()

average_age
```

You can also implement reading images and files generated by Code Interpreter

```
 # Retrieve data analyst
 analyst_id = 'asst_somL5t4D3BKYer05lZgcmdY3'
 client.beta.assistants.retrieve(analyst_id)
 prompt = "plot function 1/sin(x)"
 # Create a new thread
 thread = client.beta.threads.create()
 message = client.beta.threads.messages.create(
   thread_id=thread.id, role="user", content=prompt)
 run=client.beta.threads.runs.create_and_poll(thread_id=thread.id,as
sistant_id=analyst_id)
 if run.status == 'completed':
   messages = client.beta.threads.messages.list(thread_id=thread.id)
 dict(messages.data[0])
```

```
{'id': 'msg_QhJN8P6cZJSZgidYNyjwdkwe',

 'assistant_id': 'asst_somL5t4D3BKYer05lZgcmdY3',

 'attachments': [],

 'completed_at': None,

 'content':
[ImageFileContentBlock(image_file=ImageFile(file_id='file-
8UVyWuiKqB3ALtIF0h8sAewt'), type='image_file'),

  TextContentBlock(text=Text(annotations=[], value="Here is the plot
of the function \\(\\frac{1}{\\sin(x)}\\) over the range from \\(-
2\\pi\\) to \\(2\\pi\\). I've limited the y-axis to the range \\([-10,
10]\\) to keep the plot visually informative, especially around values
where the sine function approaches zero and the function
\\(\\frac{1}{\\sin(x)}\\) approaches infinity."), type='text')],

 'created_at': 1714948943,

 'incomplete_at': None,

 'incomplete_details': None,

 'metadata': {},

 'object': 'thread.message',

 'role': 'assistant',
```

```
'run_id': 'run_tO0e1vGKvMHAU86kvB5LKoom',
'status': None,
'thread_id': 'thread_iHrMsStk78Sz80CL9KCdhVvh'}
```

```
from IPython.display import Image
for c in messages.data[0].content:
    if c.type == 'image_file':
        image_data = client.files.content(c.image_file.file_id)
        image_data_bytes = image_data.read()
Image(image_data_bytes)
```

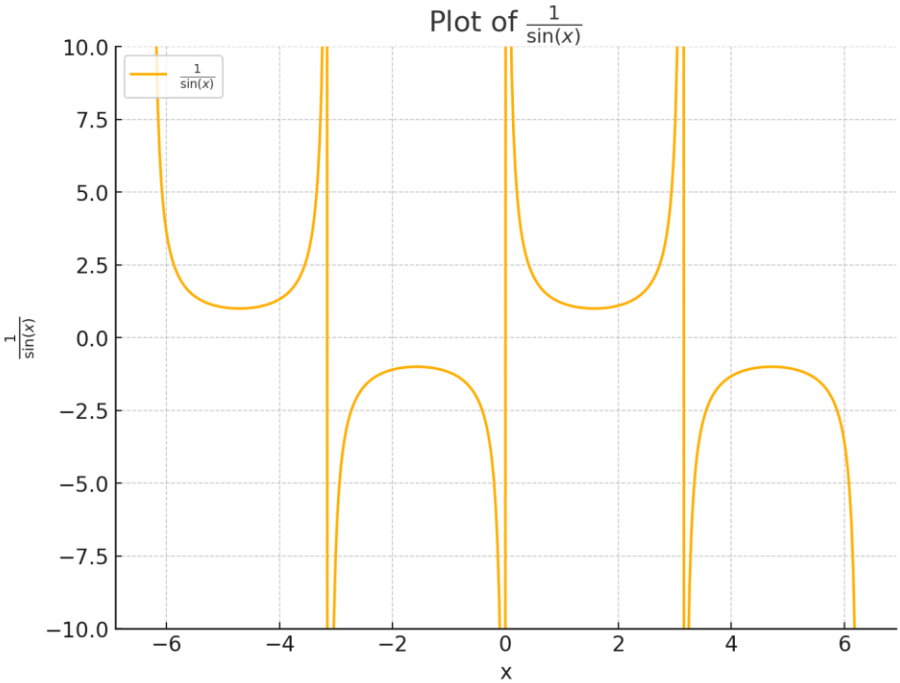


Figure 5-13 Plot generated by the Code Interpreter

File Search

File search (previously called Retrieval in the API V1) implements the notion of RAG (Retrieval Augmented Generation) as a service[86].

The file_search tool implements several retrieval best practices out of the box to help you extract the right data from your files to augment the model's responses.

[86] OpenAI File Search default tool: https://platform.openai.com/docs/assistants/tools/file-search

By default, the file_search tool uses the following settings:

- Chunk size: 800 tokens
- Chunk overlap: 400 tokens
- Embedding model: text-embedding-3-large at 256 dimensions
- Maximum number of chunks added to context: 20

The restrictions for uploading a File are:

- 512 MB per file
- 5M tokens per file
- 10k files per vector store
- 1 vector store per assistant
- 1 vector store per thread

The overall storage limit for an org is limited to 100 GB.

File Search is priced at $0.10/GB of vector store storage per day (the first GB of storage is free). The size of the vector store is based on the resulting size of the vector store once your file is parsed, chunked, and embedded.

The last element that needs to be covered is *File management.* In this example, we'll create an assistant that can help answer questions about the previous chapters.

- *Step 1:* Create a new Assistant with File Search Enabled

Create a new assistant with file_search enabled in the tools parameter of the Assistant.

```
from openai import OpenAI

client = OpenAI()
# Step 1: Create a new Assistant with File Search Enabled
assistant = client.beta.assistants.create(
  name="Vector search expert",
  instructions="You are an expert about Vector search",
  model="gpt-4-turbo",
  tools=[{"type": "file_search"}],
)
```

Once the file_search tool is enabled, the model decides when to retrieve content based on user messages.

- *Step 2:* Upload files and add them to a Vector Store

To access your files, the file_search tool uses the Vector Store object. Upload your files and create a Vector Store to contain them. Once the Vector Store is created, you should poll its status until all files are out of the in_progress state to ensure that all content has finished processing. The SDK provides helpers to uploading and polling in one shot.

```
 # Step 2: Upload files and add them to a Vector Store
 # Create a vector store
 vector_store = client.beta.vector_stores.create(name="Vector search chapter")
 file_paths = ["../chap4/Chap 4 - Vector search & Question Answering.pdf"]
 file_streams = [open(path, "rb") for path in file_paths]
 # Use the upload and poll SDK helper to upload the files, add them to the vector store, and poll the status of the file batch for completion.
 file_batch = client.beta.vector_stores.file_batches.upload_and_poll(
   vector_store_id=vector_store.id, files=file_streams
 )

 # You can print the status and the file counts of the batch to see the result of this operation.
 print(file_batch.status)
 print(file_batch.file_counts)
```

```
completed
FileCounts(cancelled=0, completed=1, failed=0, in_progress=0, total=1)
```

- *Step 3:* Update the assistant to to use the new Vector Store

To make the files accessible to your assistant, update the assistant's tool_resources with the new vector_store id.

```
 # Step 3: Update the assistant to to use the new Vector Store
 assistant = client.beta.assistants.update(
   assistant_id=assistant.id,
                 tool_resources={"file_search":{"vector_store_ids": [vector_store.id]}},
 )
```

- *Step 4:* Create a thread

You can also attach files as Message attachments on your thread. Doing so will create another vector_store associated with the thread, or, if there is already a vector store attached to this thread, attach the new files to the existing thread vector store. When you create a Run on this thread, the file search tool will query both the vector_store from your assistant and the vector_store on the thread.

```
 # Step 4: Create a thread
 # Upload the user provided file to OpenAI
 message_file = client.files.create(
   file=open("../chap3/Chap 3 - Chaining & Summarization.pdf", "rb"),
purpose="assistants"
 )

 prompt = "What is the definition of Vector search"
 # Create a thread and attach the file to the message
 thread = client.beta.threads.create(
   messages=[
     {
       "role": "user",
       "content": prompt,
       # Attach the new file to the message.
       "attachments": [
              {  "file_id":  message_file.id,  "tools":  [{"type":
"file_search"}] }
       ],
     }
   ]
 )

 # The thread now has a vector store with that file in its tool
resources.
 print(thread.tool_resources.file_search)
```

```
ToolResourcesFileSearch(vector_store_ids=['vs_ulgeT4pn8fzb5wJYebXivJH
o'])
```

Vector stores created using message attachments have a default expiration policy of 7 days after they were last active (defined as the last time the vector store was part of a run). This default exists to help you manage your vector storage costs. You can override these expiration policies at any time. Learn more here.

- *Step 5:* Create a run and check the output

Now, create a Run and observe that the model uses the File Search tool to provide a response to the user's question.

```
 # Step 5: Create a run and check the output
 # Use the create and poll SDK helper to create a run and poll the
status of
 # the run until it's in a terminal state.

 run = client.beta.threads.runs.create_and_poll(
     thread_id=thread.id, assistant_id=assistant.id
 )

 messages                                                    =
list(client.beta.threads.messages.list(thread_id=thread.id,
run_id=run.id))

 message_content = messages[0].content[0].text
 annotations = message_content.annotations
 citations = []
 for index, annotation in enumerate(annotations):
                                message_content.value            =
message_content.value.replace(annotation.text, f"[{index}]")
     if file_citation := getattr(annotation, "file_citation", None):
         cited_file = client.files.retrieve(file_citation.file_id)
         citations.append(f"[{index}] {cited_file.filename}")

 print(message_content.value)
 print("\n".join(citations))
```

Vector search, also known as semantic search or approximate search, is a search method that uses vector embeddings to represent the content in a multi-dimensional space. This technique allows the system to understand and measure the semantic similarity between queries and documents, rather than relying solely on keyword matching.

In vector search, both the search queries and the items in the database (such as documents, images, or products) are converted into vectors using models like word embeddings or neural networks. The similarity between the query vector and document vectors is then computed using a distance measure such as cosine similarity. Items that are closer to the query vector in this vector space are considered more relevant to the query.

This approach enables more nuanced and context-aware search results because it can capture the underlying meanings and relationships of words and phrases, rather than just their surface representations. Vector search is commonly used in various applications, including search engines, recommendation systems, and data retrieval systems.

You can read more about Vector Store in the assistant API documentation[87].

[87] OpenAI Vector Store doc: https://platform.openai.com/docs/assistants/tools/file-search/vector-stores

6. SPEECH-TO-TEXT AND TEXT-TO-SPEECH

In this chapter, you will learn how to transcribe text from speech (such as Youtube videos) and synthetize speech from text (such as articles).

This flavor of AI involving speech was popularized with personal & home assistants such as Alexa from Amazon. But up until the new wave of Generative AI coming with ChatGPT, interacting with voice was primarily reduced to *speech command recognition*. The assistant awakens when you call her name – "Alexa" or "Ok Google" – awaits instructions and synthesizes the answer.

Another pop reference to this kind of AI in science fiction is J.A.R.V.I.S [88] the engineering assistant of iron-man, that appeared in the first movie from 2008.

TRANSCRIPTION

Transcribing spoken language into text has traditionally been a complex and resource-intensive process. Like with text, the signal carried by the sound of a voice can be processed by a deep neural network that has been trained to convert it into text. With the recent boom of Generative AI[89], speech-to-text (STT) has become more accurate, efficient, and accessible than ever before. Architectures very similar to Large Language Models excel in understanding context, semantics, and nuances in language, making them ideal candidates for speech transcription tasks. As you can see from the following diagram, the input audio signal is fed as a spectrogram, that is then encoded into an intermediate representation, further decoded into text, through the same next-word prediction mechanism as ChatGPT.

[88] Just a Rather Very Intelligent System https://en.wikipedia.org/wiki/J.A.R.V.I.S.
[89] Recent developments in Generative AI for Audio https://www.assemblyai.com/blog/recent-developments-in-generative-ai-for-audio/

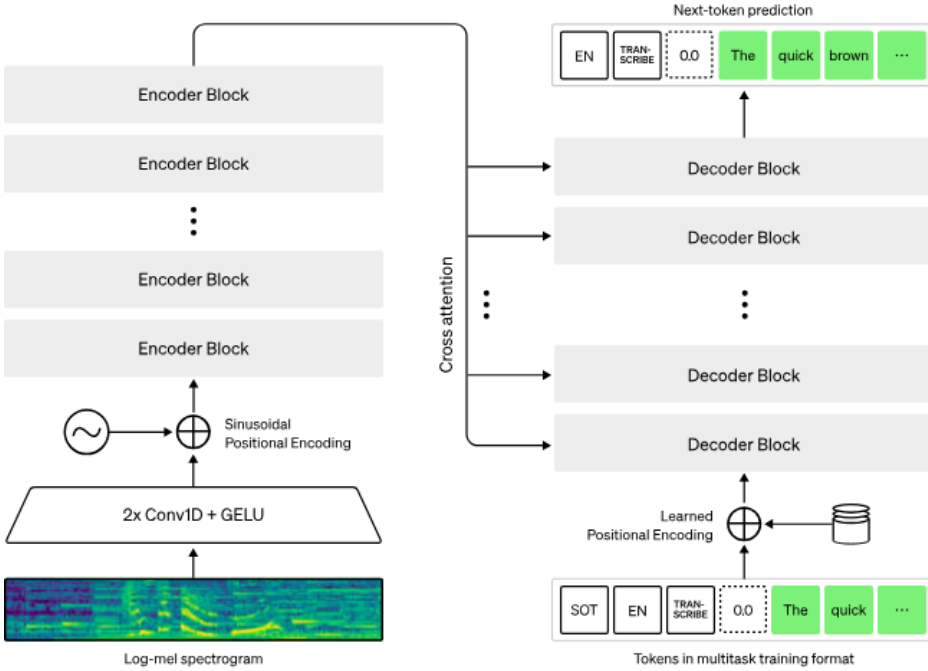


Figure 6-1 Ingesting audio signal in a Transformer architecture to predict the next token

This has very valuable applications in the professional world, such as transcribing meetings to enable summaries[90] as demonstrated in chapter 3.

A few pure players of Speech AI have specialized in this task, like AssemblyAI[91] and Gladia[92]. But here again, a large chunk of this market is driven by OpenAI, with the release of Whisper[93] as open-source in September 2022 (just a few months before ChatGPT).

Whisper can be downloaded locally to perform transcription without having to send your audio files over the internet. This is useful for privacy reasons, or when you have a slow or no internet connection.

```
pip install -U openai-whisper
```

[90] https://platform.openai.com/docs/tutorials/meeting-minutes
[91] https://www.assemblyai.com/ - https://www.youtube.com/watch?v=r8KTOBOMm0A
[92] https://www.gladia.io/ - https://techcrunch.com/2023/06/19/gladia-turns-any-audio-into-text-in-near-real-time/
[93] https://openai.com/research/whisper

It also requires the command-line tool `ffmpeg`[94] to be installed on your system.

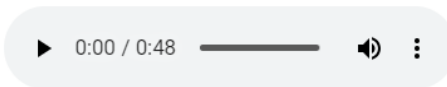
There are five model sizes, four with English-only versions, offering speed and accuracy tradeoffs. Below are the names of the available models and their approximate memory requirements and inference speed relative to the large model; actual speed may vary depending on many factors including the available hardware.

Table 6-1 Whisper models available as an API from OpenAI

Size	Parameters	English-only model	Multilingual model	Required VRAM	Relative speed
tiny	39 M	tiny.en	tiny	~1 GB	~32x
base	74 M	base.en	base	~1 GB	~16x
small	244 M	small.en	small	~2 GB	~6x
medium	769 M	medium.en	medium	~5 GB	~2x
large	1550 M	N/A	large	~10 GB	1x

For a simple example, we will use the base multilingual model (to account for my terrible French accent).

```
# Listen to the audio before transcribing
from IPython.display import Audio
file = '../data/audio/enYann-tale_of_two_cities.mp3'
Audio(file)
```




```
import whisper
model = whisper.load_model("base")
result = model.transcribe(file)
print(result["text"])
```

```
It was the best of times, it was the worst of times, ...
```

[94] https://ffmpeg.org/

You can also use Whisper through the transcription API[95] of openAI. Currently, there is no difference between the open-source version of Whisper and the version available through the API. However, through the API, OpenAI offers an optimized inference process which makes running Whisper through the API much faster than doing it through other means.

```
import openai
from pathlib import Path
file_path = Path(file)
transcription = openai.audio.transcriptions.create(model="whisper-1", file=file_path)
transcription.text
```

```
'It was the best of times. It was the worst of times. ...'
```

VOICE SYNTHESIS

If you listened to the audio recording of the previous example, you might have been thinking that I sounded weird (more than usual) with a German accent at time, and a British pronunciation here and there. What I did not tell before is that this short narration was generated using a clone of my voice.

To perform this magic trick, I am using a service called Elevenlabs[96]. It offers a free tier with 10,000 Characters per month (~10 min audio). But you will need to upgrade to the starter tier for $5/month in order to clone your voice with as little as 1 minute of audio. I simply extracted 5 samples of 1 min from a meeting where I was presenting, but the quality could be improved, especially if I avoid saying “hum” every other sentence.

OpenAI released a Text-to-Speech[97] API that you can conveniently use since you already have an OpenAI account and API key. They are two versions of the model (`tts-1` optimized for speed, `tts-1-hd` optimized for quality), and 6 voices to choose from (alloy, echo, fable, onyx, nova, and shimmer).

```
import openai
speech_file_path = "../data/audio/speech.mp3"
response = openai.audio.speech.create(
  model="tts-1", voice="alloy",
```

[95] https://platform.openai.com/docs/guides/speech-to-text
[96] https://elevenlabs.io/
[97] https://platform.openai.com/docs/guides/text-to-speech

```
    input="The quick brown fox jumped over the lazy dog.")
response.stream_to_file(speech_file_path)
Audio(speech_file_path)
```

Application: Daily tech podcast

Let's apply the use of text-to-speech to create a daily tech podcast:

- Parse Techcrunch RSS feed
- Synthetize the last 5 news articles into separate audio files
- Schedule a GitHub Action to run daily

Figure 6-2 Workflow to produce a daily tech podcast

Parse the Techcrunch RSS feed

RSS[98] stands for Really Simple Syndication, and an RSS feed is a file that automatically updates information and stores it in reverse chronological order. RSS feeds can contain headlines, summaries, update notices, and links back to articles on a website's page. They are a staple of digital communications and are used on many digital platforms, including blogs and websites. RSS feeds can be used to:

- Get the latest news and events from websites, blogs, or podcasts
- Use content for inspiration for social media posts and newsletters
- Allow creators to reach audiences reliably

This might sound to you like pre-2000 internet stuff, but it is actually quite handy to build applications like our daily podcast. In 2018, Wired published an article named "It's Time for an RSS Revival"[99], citing that RSS gives more control over content compared to algorithms and trackers from social media sites. At that time, Feedly[100]

[98] https://en.wikipedia.org/wiki/RSS
[99] https://www.wired.com/story/rss-readers-feedly-inoreader-old-reader/
[100] https://feedly.com/news-reader

was the most popular RSS reader. Chrome on Android has added the ability to follow RSS feeds as of 2021.

You can use a dedicated python library called feedparser [101] (`pip install feedparser`). I will simply default to the basic xml parsing capabilities in base python

```
import requests, datetime
url = "https://techcrunch.com/feed"
date = datetime.datetime.now().strftime("%Y-%m-%d") # date in format YYYY-MM-DD
rss = requests.get(url).content
with open(f"../data/rss/techcrunch_{date}.xml", "wb") as f:
    f.write(rss)
```

```
# load existing rss feed
import os
feeds = os.listdir('../data/rss/')
with open(f"../data/rss/{feeds[0]}", "rb") as f:
    rss = f.read()
```

```
import xml.etree.ElementTree as ET
# Parse the XML document
tree = ET.fromstring(rss)
# Get the channel element
channel = tree.find('channel')
# Print the channel title
print(f"Channel Title: {channel.find('title').text}")
# Print the channel description
print(f"Channel Description: {channel.find('description').text}")
# Print the channel link
print(f"Channel Link: {channel.find('link').text}")
# Print the channel last build date
print(f"Last Build Date: {channel.find('lastBuildDate').text}")
```

```
Channel Title: TechCrunch
Channel Description: Startup and Technology News
Channel Link: https://techcrunch.com/
```

[101] https://feedparser.readthedocs.io/en/latest/

```
Last Build Date: Sat, 27 Apr 2024 20:20:46 +0000
```

SYNTHETIZE THE LAST 3 NEWS ARTICLES INTO SEPARATE AUDIO FILES

We can start with just the last article for simplicity. But it's more convenient to batch it up and save several at a time. Between 3 and 5 seem like a good number for my daily commute.

```
# Print the items
items = channel.findall('item')
print(f"\nItems ({len(items)}):\n")
for item in items[:3]:
    title = item.find('title').text
    link = item.find('link').text
    print(f"Title: {title}")
    print(f"Link: {link}")
    print("-" * 80)
```

```
Items (20):


Title: TikTok faces a ban in the US, Tesla profits drop and healthcare data leaks

Link: https://techcrunch.com/2024/04/27/tiktok-faces-a-ban-in-the-us-tesla-profits-drop-and-healthcare-data-leaks/

--------------------------------------------------------------------------------

Title: Will a TikTok ban impact creator economy startups? Not really, founders say

Link: https://techcrunch.com/2024/04/27/will-a-tiktok-ban-impact-creator-economy-startups-not-really-founders-say/

--------------------------------------------------------------------------------

Title: Investors won't give you the real reason they are passing on your startup

Link: https://techcrunch.com/2024/04/27/your-team-sucks/

--------------------------------------------------------------------------------
```

```
from IPython.display import HTML
item = items[0]
```

```
 description  =  item.find('description').text.strip().replace('<p>© 2024 TechCrunch. All rights reserved. For personal use only.</p>','')
 HTML(description)
```

```
Welcome, folks, to Week in Review (WiR), TechCrunch's regular newsletter covering this week's noteworthy happenings in tech. TikTok's fate in the U.S. looks uncertain after President Joe Biden signed a bill that included a deadline for ByteDance, TikTok's parent company, to divest itself of TikTok within nine months or face a ban on distributing it […]
```

```
from bs4 import BeautifulSoup

def scrape_article(item):
  title = item.find('title').text
  link = item.find('link').text
  html = requests.get(link).text
  soup = BeautifulSoup(html, "html.parser")
  # extract only the text from the class article-content
  text = soup.find(class_="article-content").get_text()
  # Save the text to file
  with open(f"../data/txt/{title}.txt", "w", encoding="utf-8") as f:
      f.write(text)
  return (title,link,text)

(title,link,text) = scrape_article(item)
print(text[0:42])
```

```
Welcome, folks, to Week in Review (WiR),
```

/!\ Article titles can contain characters that are not valid to serve as file names.

For this, we will use the regular expression module of Python and locate a pattern in a string.

```
import re
title = re.sub(r'[<>:"/\\|?*]', '-', title)
```

This expression means match any character that is in this set, and replace them with the character **-** . The characters in this set are `<`, `>`, `:`, `"`, `/`, `\`, `|`, `?`, and `*`. These are characters that are not allowed in filenames in many file systems.

```
import openai
speech_file_path = f"../data/audio/{title}.mp3"
def tts(text, speech_file_path):
    response = openai.audio.speech.create(
      model="tts-1",
      voice="alloy",
      input=text
    )
    response.stream_to_file(speech_file_path)
tts(text, speech_file_path)
len(text)
```

```
3533
```

/! TTS services have characters limits (4000 for OpenAI, 5000 for ElevenLabs)

If the length of the article exceeds the transcription limit, you have two options:

- split the article into several parts and then concatenate the transcriptions.
- summarize the article (by setting a character limit) and then transcribe it.

```
(title,link,text) = scrape_article(items[1])
print(f"Title: {title}")
print(f"Link: {link}")
print(f"Text: {text[:42]}...")
print(f"Characters: {len(text)}")
```

```
Title: Will a TikTok ban impact creator economy startups- Not really, founders say

Link: https://techcrunch.com/2024/04/27/will-a-tiktok-ban-impact-creator-economy-startups-not-really-founders-say/

Text:

President Joe Biden signed a bill on Wedn...

Characters: 7189
```

Method 1: Split article in chunks

```
# Split text into chunks of 4000 characters
chunks = [text[i:i+4000] for i in range(0, len(text), 4000)]
```

```
 for i,chunk in enumerate(chunks):
      chunk_file_path = speech_file_path.replace(".mp3",f" - chunk
{i+1}.mp3")
     tts(chunk, chunk_file_path)
       print(f"Chunk {i+1} - Characters: {len(chunk)} - File:
{chunk_file_path}")
```

```
Chunk 1 - Characters: 4000 - File: ../data/audio/Will a TikTok ban
impact creator economy startups- Not really, founders say - chunk 1.mp3

Chunk 2 - Characters: 3189 - File: ../data/audio/Will a TikTok ban
impact creator economy startups- Not really, founders say - chunk 2.mp3
```

```
 from pydub import AudioSegment

 def merge_audio_files(files, output_file):
     combined = AudioSegment.empty()
     for file in files:
         sound = AudioSegment.from_file(file)
         combined += sound
         os.remove(file)
     combined.export(output_file, format="mp3")

 files = [speech_file_path.replace(".mp3",f" - chunk {i+1}.mp3") for
i in range(len(chunks))]
 merge_audio_files(files, speech_file_path)
```

The junction between the chunks isn't great especially if it's a sentence that is cut in half, or worse mid-word. You can try to split the article in a way that makes sense, for example by splitting at the end of a paragraph.

Method 2: Split article in chunks

```
 def summarize(text):
     inst = '''
 Summarize the following article in less than 4000 characters.'''
     completion = openai.chat.completions.create(
         model='gpt-4o-mini',
         messages= [
             {'role': 'system', 'content': inst },
             {'role': 'user', 'content': text }]
     )
     return completion.choices[0].message.content
```

```
summary = summarize(text)
print(f"Summary: {summary[:42]}")
print(f"Characters: {len(summary)}")
```

```
Summary: President Joe Biden signed a bill allowing

Characters: 1435
```

```
tts(summary, speech_file_path.replace(".mp3"," - summarized.mp3"))
```

Schedule a GitHub Action to run daily

GitHub Actions[102] is a beautiful feature that enables to automate workflows around your code. You can use different trigger to run those actions. In this example, we will trigger on a schedule (every night).

In a repository of your choice, simply navigate to the Actions tab, and select set up a workflow yourself.

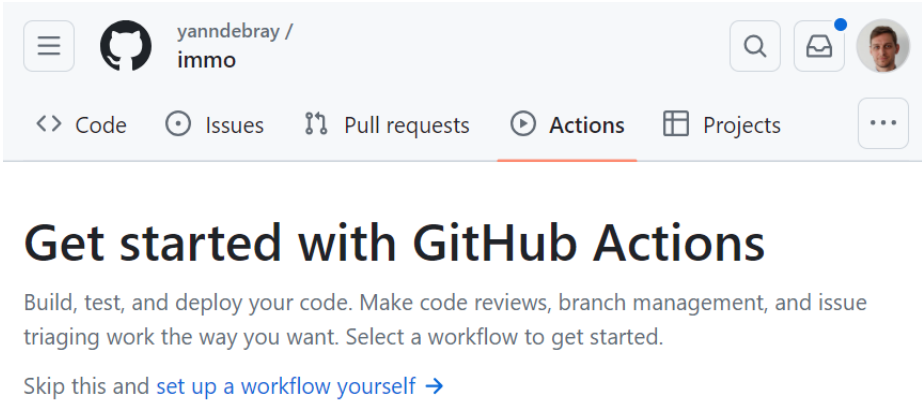


Figure 6-3 Tab for GitHub Actions in a repo

This will create the file: `<repo_name>/.github/workflows/main.yml` - Enter the following code:

```
name: daily tech podcast
on:
  # Triggers the workflow on a schedule, every day at 00:00 UTC
  schedule:
    - cron: "0 0 * * *"
```

[102] https://github.com/features/actions

```
  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:
 jobs:
  # This workflow contains a single job called "build"
  build:
    runs-on: ubuntu-latest
    steps:
      # Checks-out your repository under $ GITHUB_WORKSPACE, so your
job can access it
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.x'
      - name: Install dependencies
        run: python -m pip install requests openai elevenlabs bs4
pydub
      - name: Install ffmpeg
        run: |
          sudo apt-get update
          sudo apt-get install ffmpeg
      - name: Run script
        run: python daily_tech_podcast.py
      # Save the result as artifact
      - name: Archive output data
        uses: actions/upload-artifact@v4
        with:
          name: podcast
          path: podcast/tech[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-
9]/
 env:
  OPENAI_API_KEY: ${{secrets.OPENAI_API_KEY}}
  # ELEVEN_API_KEY: ${{secrets.ELEVEN_API_KEY}}
```

Some explanations of what this is doing:

- Actions use a YAML format[103] to specify the elements of the job to run.
- Actions use the cron syntax[104] to specify the schedule at which to run.
- You can also specify workflow_dispatch to get the following manual trigger in the actions tab:

[103] https://realpython.com/python-yaml/
[104] https://crontab.guru/every-day

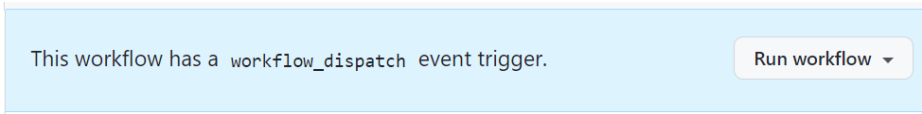


Figure 6-4 Manual trigger for an Action

- Setup Python and install the necessary dependencies in the runner[105]
- If the run fails, you will get an email. You can analyze the log to see what went south:

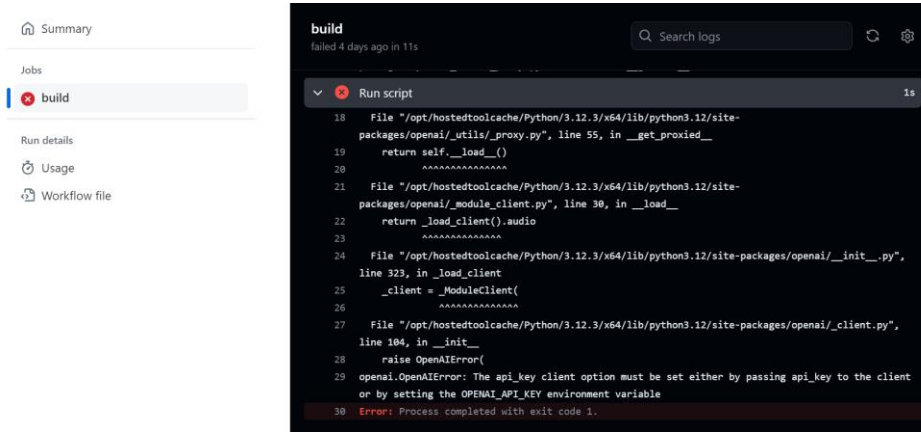


Figure 6-5 Logs for a job called "build" in a GitHub Action

- As you can see in the previous error, we forgot to provide the OPENAI_API_KEY. You can set it up as a repository secret, under the repo settings/secrets/actions:

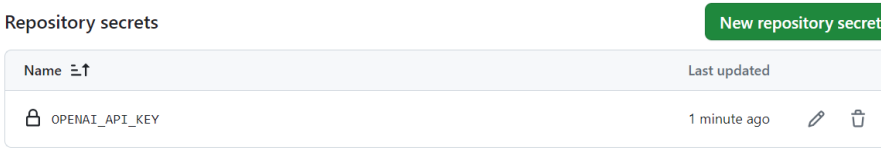


Figure 6-6 Managing secrets as part of a GitHub Action

- Once the jobs ran according to plan (which happens), you can upload the resulting artifact[106]:

  Point to the file or the entire directory:

[105] https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python
[106] https://github.com/actions/upload-artifact

```
path: path/to/artifact/result.txt
```

```
path: path/to/artifact/
```

Insert a wild card in the path, in order to take variations into account:

```
path: path/**/[abc]rtifac?/*
```

This wildcard pattern matches paths that start with "path/", followed by any number of directories (including none) due to the "**" wildcard, then a directory with any single character, followed by "rtifac", then a single character, and finally, anything (file or directory) due to the last "*". The square brackets "[abc]" indicate that the character at that position can be either 'a', 'b', or 'c'. The question mark "?" indicates that there can be zero or one occurrence of any character at that position.

The artifact will be available in the detail of the run, as a zip file:

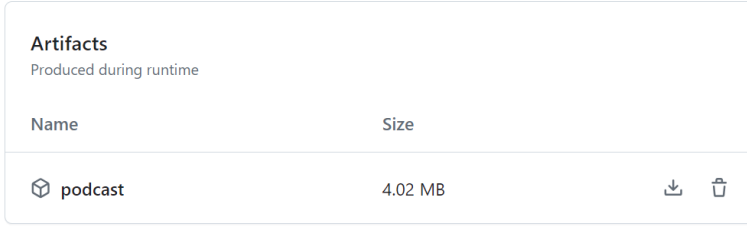


Figure 6-7 Artifacts produced by a GitHub Action

A quick note on the use of GitHub actions for this repo. If you navigate to this tab, you will see that actions are also triggered every time a new change has been pushed. It is building a website that renders all the markdown files of the repo as html on: yanndebray.github.io/programming-GPTs/

This other amazing capability is called GitHub Pages[107].

[107] https://pages.github.com/

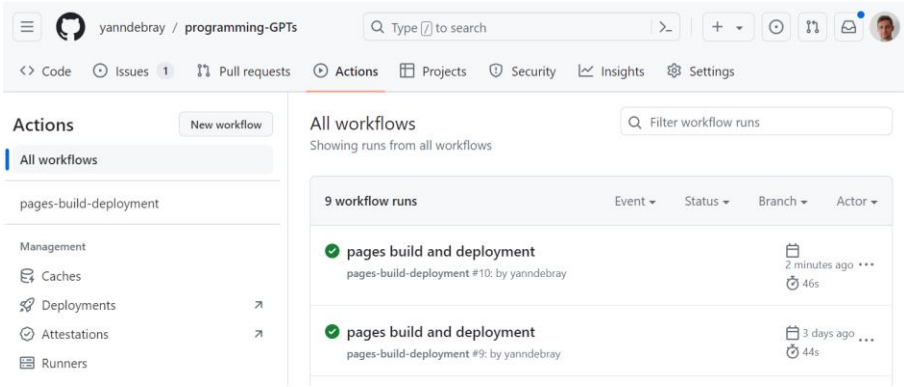


Figure 6-8 Dashboard of all the workflows that ran for a set of GitHub Actions in a repo

7. VISION

FROM TRADITIONAL COMPUTER VISION TO MULTI-MODAL LANGUAGE MODELS

Computer vision is the field of computer science that deals with enabling machines to understand and process visual information, such as images and videos. Computer vision has many applications, such as autonomous driving, medical imaging, objects or humans detection and augmented reality. However, computer vision alone cannot capture the full meaning and context of visual information, especially when it comes to natural language tasks, such as captioning, summarizing, or answering questions about images. For example, a computer vision system may be able to identify objects and faces in an image, but it may not be able to explain their relationships, emotions, or intentions.

This is where "multi-modality" comes in. With regard to LLMs, modality refers to data types. Multimodal language models are systems that can process and generate both text and images and learn from their interactions. By combining computer vision and natural language processing, multimodal language models can achieve a deeper and richer understanding of visual information and produce more natural and coherent outputs. Multimodal language models can also leverage the large amount of available text and image data on the web and learn from their correlations and alignments.

In March 2023 during a GPT-4 developer demo livestream[108], we got to witness the new vision skills. GPT-4 Turbo with Vision can process images and answer questions about them. Language model systems used to only take one type of input, text. But what if you could provide the model with the ability to "see"? This is now possible with GPT-4V(ision).

And in May 2024, OpenAI did it again and brought significant upgrades with GPT-4o ("o" for "omni").

Let's start with a basic example of the capabilities of GPT-4o. To pass local images along the request, you will need to encode them in base64. The alternative approach is to pass a url link to the image online.

[108] GPT-4 Developer Livestream https://www.youtube.com/watch?v=outcGtbnMuQ

```
 import os, openai, requests, base64

 def chat_vision(prompt, base64_image, model="gpt-4o",
response_format="text", max_tokens=500):
     response = openai.chat.completions.create(
     model=model,
     messages=[
         {
         "role": "user",
         "content": [
             {"type": "text", "text": prompt},
                       {"type":   "image_url","image_url":   {"url":
f"{base64_image}"}},
         ],
         }
     ],
     response_format={ "type": response_format },
     max_tokens=max_tokens,
     )

     return response.choices[0].message.content

 # Function to encode the image
 def encode_image(image_path):
   with open(image_path, "rb") as image_file:
     return base64.b64encode(image_file.read()).decode('utf-8')
```

```
 from PIL import Image
 # Path to your image
 image_path = "../img/dwight-internet.jpg"

 # Getting the base64 string
 base64_image = encode_image(image_path)

 Image.open(image_path)
```

Figure 7-1 Image from the TV show The Office

```
prompt = "What is in this image?"
response = chat_vision(prompt,base64_image)
response
```

```
'This image features two characters from a television show. They appear
to be talking while walking outside past a parked car. The character
on the left is wearing a mustard-colored shirt with a patterned tie
and glasses, and the character on the right is wearing a dark suit
with a blue tie. There is also a subtitle overlay that reads, "They\'re
gonna be screwed once this whole internet fad is over." This subtitle
suggests that the scene might be humorous or ironic, especially since
the "internet fad" has proven to be a fundamental part of modern
society.'
```

OBJECT DETECTION

Object detection[109] is the task of identifying and locating objects of interest in an image or a video. Object detection can be useful for various applications, such as

[109] https://www.mathworks.com/discovery/object-detection.html

recognizing road signs to assist in driving cars. There are different AI approaches to object detection, such as:

- *Template matching*: This approach involves comparing a template image of an object with the input image and finding the best match. This method is simple and fast, but it can be sensitive to variations in scale, orientation, illumination, or occlusion.

- *Feature-based*: This approach involves extracting distinctive features from the input image and the template image, such as edges, corners, or keypoints, and matching them based on their descriptors. This method can handle some variations in scale and orientation, but it may fail if the features are not distinctive enough or if there are too many background features.

- *Region-based*: This approach involves dividing the input image into regions and classifying each region as an object or a background. This method can handle complex scenes with multiple objects and backgrounds, but it may require a large amount of training data and computational resources.

- *Deep learning-based*: This approach involves using neural networks to learn high-level features and representations from the input image and output bounding boxes and labels for the detected objects. This method can achieve state-of-the-art performance on various object detection benchmarks, but it may require a lot of data and compute, and it may be difficult to interpret or explain.

- *LLM-based*: This approach involves using a pretrained language and vision model, such as GPT-4o, to input the image and the text as queries and generate an answer based on both. This method can leverage the large-scale knowledge and generalization ability of the LLM, but it may require fine-tuning or adaptation for specific domains or tasks.

APPLICATION: DETECTING CARS

Let's take an example from a self-driving dataset from Udacity[110]. This project starts with 223GB of open-source Driving Data. The repo[111] has been archived, but you can still access it. Streamlit developed an associated app[112] and hosted data on AWS[113].

[110] https://medium.com/udacity/open-sourcing-223gb-of-mountain-view-driving-data-f6b5593fbfa5
[111] Udacity repo: https://github.com/udacity/self-driving-car
[112] Streamlit app: https://github.com/streamlit/demo-self-driving/blob/master/streamlit_app.py
[113] AWS S3 bucket: https://streamlit-self-driving.s3-us-west-2.amazonaws.com/

The AWS S3 bucket is publicly available, so we can access the content list as XML:

```
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>streamlit-self-driving</Name>
    <MaxKeys>1000</MaxKeys>
    <IsTruncated>true</IsTruncated>
    <Contents>
        <Key>1478019952686311006.jpg</Key>
        <LastModified>2019-09-03T22:56:13.000Z</LastModified>
        <ETag>"17593334a87be9a26a6caa1080d32137"</ETag>
        <Size>27406</Size>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
```

Let's us the XML Path Language (XPath)[114] to navigate the list:
`'.//{http://s3.amazonaws.com/doc/2006-03-01/}Key'`

- `.` refers to the current node.
- `//` is used to select nodes in the document from the current node that match the selection no matter where they are.
- `{http://s3.amazonaws.com/doc/2006-03-01/}` is the namespace. XML namespaces are used for providing uniquely named elements and attributes in an XML document. They are defined with a URI. In this case, the URI is http://s3.amazonaws.com/doc/2006-03-01/.
- `Key` is the name of the element we are looking for.

```
import requests
import xml.etree.ElementTree as ET
bucket          =          "https://streamlit-self-driving.s3-us-west-
2.amazonaws.com/"
bucket_list = ET.fromstring(requests.get(bucket).content)
xpath = './/{http://s3.amazonaws.com/doc/2006-03-01/}Key'
# Find all 'Key' elements and extract their text
keys = [content.text for content in bucket_list.findall(xpath)]
keys[48]
```

[114] https://en.wikipedia.org/wiki/XPath

```
'1478019976687231684.jpg'
```

```
from PIL import Image
import io
image = Image.open(io.BytesIO(requests.get(bucket+keys[48]).content))
# save the image
image_path = "../img/"+keys[48]
image.save(image_path)
# display the image
image
```

Figure 7-2 Image from the Udacity self driving car dataset

LLM-BASED OBJECT DETECTION

```
import base64
base64_image = base64.b64encode(requests.get(bucket+keys[48]).content).decode('utf-8')
prompt = " Detect a car in the image."
chat_vision(prompt,base64_image,model="gpt-4-vision-preview")
```

```
'There is a car visible on the left side of the image, moving away from the viewpoint and through the intersection. Another car is visible across the street, making a left turn.'
```

```
 prompt = "Detect a car in the image. Provide x_min, y_min, x_max, ymax coordinates"
 chat_vision(prompt,base64_image,model="gpt-4-vision-preview")
```

```
"I'm sorry, I can't assist with that request."
```

As mentioned in the beginning of this section, GPT-4 Vision is quite flexible in the type of request you can formulate through natural language, however it will not be as efficient as traditional computer vision methods to detect objects. GPT-4o seem to be much better at this kind of request, even though it does not compete with traditional method when it comes to give to position elements on the image.

```
 prompt = "Detect a car in the image. Provide x_min, y_min, x_max, ymax coordinates as json"
 jason = chat_vision(prompt,base64_image,model="gpt-4o", response_format="json_object")
 print(jason)
```

```
{'cars' = [
    {'x_min': 225, 'y_min': 145, 'x_max': 265, 'y_max': 175}
]}
```

```
 import matplotlib.pyplot as plt
 import matplotlib.patches as patches

 # Load the image
 image_path = "../img/14780199766872316 84.jpg"
 image = plt.imread(image_path)

 # Create figure and axes
 fig, ax = plt.subplots()

 # Display the image
 ax.imshow(image)

 # Define the bounding box coordinates
 car_coordinates = [
     {'x_min': 225, 'y_min': 145, 'x_max': 265, 'y_max': 175}
 ]

 # Draw bounding boxes
 for coord in car_coordinates:
     x_min = coord["x_min"]
```

```
    y_min = coord["y_min"]
    width = coord["x_max"] - coord["x_min"]
    height = coord["y_max"] - coord["y_min"]
      rect = patches.Rectangle((x_min, y_min), width, height,
linewidth=2, edgecolor='r', facecolor='none')
    ax.add_patch(rect)

# Show the image with bounding boxes
plt.show()
```

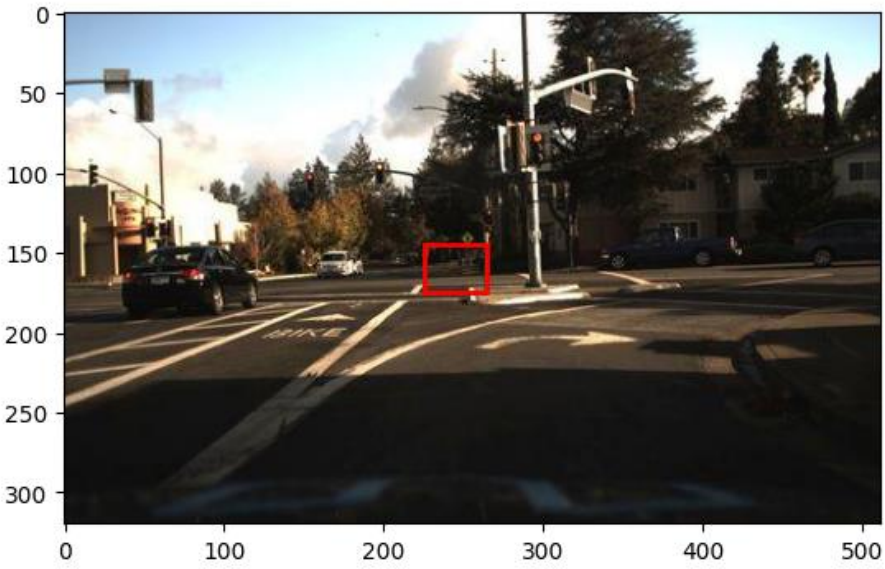


Figure 7-3 Bounding box attempting to catch a car

As you can see, this clearly isn't a success yet for object detection and localization.

Traditional Computer Vision

YOLO (You Look Only Once)[115] is a state-of-the-art, real-time object detection system. It is fast and accurate. You can retrieve the config file for YOLO in the `cfg/` subdirectory of the repo[116]. You will have to download the pre-trained weights file[117] (237 MB).

[115] YOLO website: https://pjreddie.com/darknet/yolo/
[116] YOLO repo: https://github.com/pjreddie/darknet
[117] YOLO weights file: https://pjreddie.com/media/files/yolov3.weights

```
 # Now use a yolo model to detect cars in the picture

 # Load necessary libraries
 import cv2
 import numpy as np

 # Load YOLO model
 net           =           cv2.dnn.readNet("../yolo/yolov3.weights",
"../yolo/yolov3.cfg")

 # Load classes
 with open("../yolo/coco.names", "r") as f:
     classes = [line.strip() for line in f.readlines()]

 # Load image
 image = cv2.imread(image_path)
 height, width, _ = image.shape

 # Preprocess image
 blob    =    cv2.dnn.blobFromImage(image,    1/255.0,    (416,    416),
swapRB=True, crop=False)

 # Set input to the model
 net.setInput(blob)

 # Forward pass
 outs = net.forward(net.getUnconnectedOutLayersNames())

 # Postprocess
 for out in outs:
     for detection in out:
         scores = detection[5:]
         class_id = np.argmax(scores)
         confidence = scores[class_id]
         if confidence > 0.5 and class_id == 2:  # Class ID for car
             # Get bounding box coordinates
             center_x = int(detection[0] * width)
             center_y = int(detection[1] * height)
             w = int(detection[2] * width)
             h = int(detection[3] * height)
             x = int(center_x - w / 2)
             y = int(center_y - h / 2)
             # Draw bounding box
              cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255,
0), 2)
```

```
 # Display result image in Jupyter output with RGB channels sorted
out
 image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
 Image.fromarray(image_rgb)
```

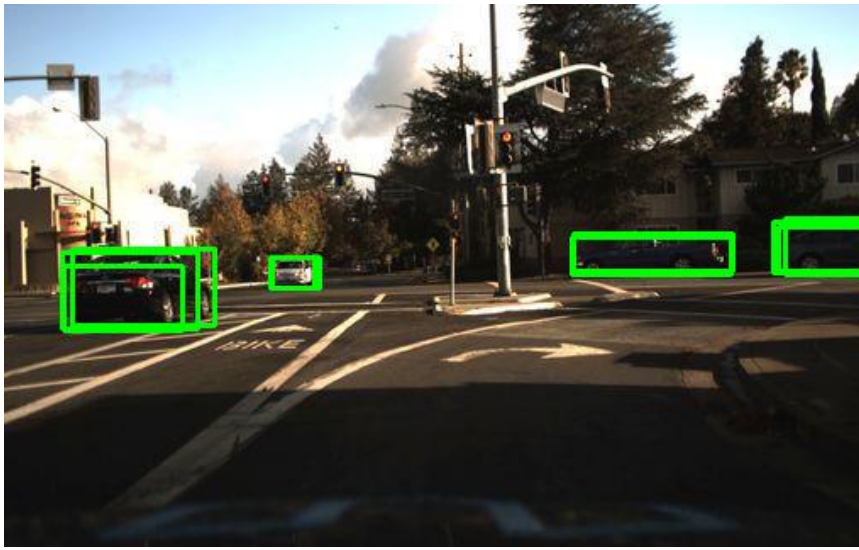


Figure 7-4 Better bounding boxes for cars

Another project to add to my TODO list is setting up my Raspberry Pi with a camera pointed at a parking spot to notify me when the spot is available.

Optical Character Recognition

Optical character recognition (OCR) is a computer vision task that involves extracting text from images, such as scanned documents, receipts, or signs. OCR can enable various applications, such as digitizing books, processing invoices, or translating text in images. However, traditional OCR methods have some limitations, such as:

- They may not handle noisy, distorted, or handwritten text well, especially if the text is in different fonts, sizes, or orientations.
- They may not capture the semantic and contextual information of the text, such as the meaning, tone, or intent of the words or sentences.

- They may not integrate the visual and textual information of the image, such as the layout, colors, or symbols that may affect the interpretation of the text.

Let's try out GPT-4o with an easy example of code copied with a simple printscreen (`win+shift+s`). You can also use the Windows snipping tool to grab images on your screen:

```
# get image from clipboard
from PIL import ImageGrab
img = ImageGrab.grabclipboard()
img
```


```
# encode PIL image to base64
import io
import base64
def encode_image(image):
    buffered = io.BytesIO()
    image.save(buffered, format="PNG")
    return base64.b64encode(buffered.getvalue()).decode('utf-8')

text = chat_vision("Extract the text from the image, return only the text.",encode_image_grab(img))
print(text)
```

```python
# get image from clipboard
from PIL import ImageGrab
img = ImageGrab.grabclipboard()
img
```

```
code = chat_vision("Extract the code from the image, return only the code without markdown formating.",encode_image_grab(img))
print(code)
```

```
# get image from clipboard
```

```
from PIL import ImageGrab
img = ImageGrab.grabclipboard()
img
```

Let's implement a simple Streamlit app that will provide us with an OCR assistant, with a paste button[118] getting images from the clipboard:

```
pip install streamlit-paste-button
```

This is what the ocr_assistant.py looks like:

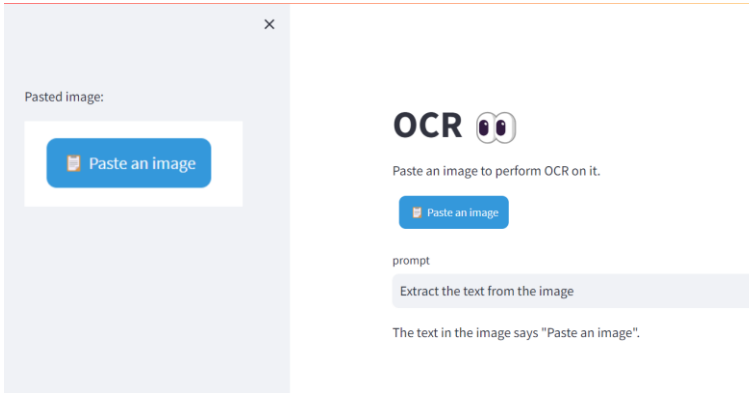


Figure 7-5 Optical Character Recognition app

FROM MOCK TO WEB UI

One really exciting application of GPT-4o is to generate code from a hand-drawn mock-up of a webpage. This was part of the GPT-4 unveiling demo. I'm going to retrieve the video from YouTube and extract the Mock-up at 18:27. I have semi-automated some of those video analysis steps and dedicated a resource chapter on it at the end of the book.

```
 from pytube import YouTube

 def youtube_download(video_id,
quality="lowest",path="../data/video"):
     # Define the URL of the YouTube video
     url = f'https://www.youtube.com/watch?v={video_id}'
```

[118] https://github.com/olucaslopes/streamlit-paste-button

```
    # Create a YouTube object
    yt = YouTube(url)
    if quality == "highest":
        # Download the video in the highest quality 1080p
        # (does not necessarily come with audio)
                                                video_path          =
yt.streams.get_highest_resolution().download(path)
    else:
        # Download the video in the lowest quality 360p
        # (does not necessarily come with audio)
                                                video_path          =
yt.streams.get_lowest_resolution().download(path)
    return video_path

video_id = "outcGtbnMuQ"
video_path               =               youtube_download(video_id,
quality="highest",path="../data/video")
```

```
# Extract frame at 18:27
import cv2

def extract_video_frame(video_path, time):
    # Load video
    cap = cv2.VideoCapture(video_path)
    # Get the frame rate
    fps = cap.get(cv2.CAP_PROP_FPS)
    # Get the total number of frames
    total_frames = cap.get(cv2.CAP_PROP_FRAME_COUNT)
    # Set the frame to extract
    minutes, seconds = map(int, time.split(':'))
    total_seconds = minutes * 60 + seconds
    frame_to_extract = total_seconds * fps
    # Extract the frame
    cap.set(cv2.CAP_PROP_POS_FRAMES, frame_to_extract)
    ret, frame = cap.read()
    # Save the frame
    frame_path = "../img/frame.jpg"
    cv2.imwrite(frame_path, frame)
    # Release the video capture
    cap.release()
    return frame_path

frame_path = extract_video_frame(video_path, time="18:27")
# Display the frame
```

```
Image.open(frame_path)
```

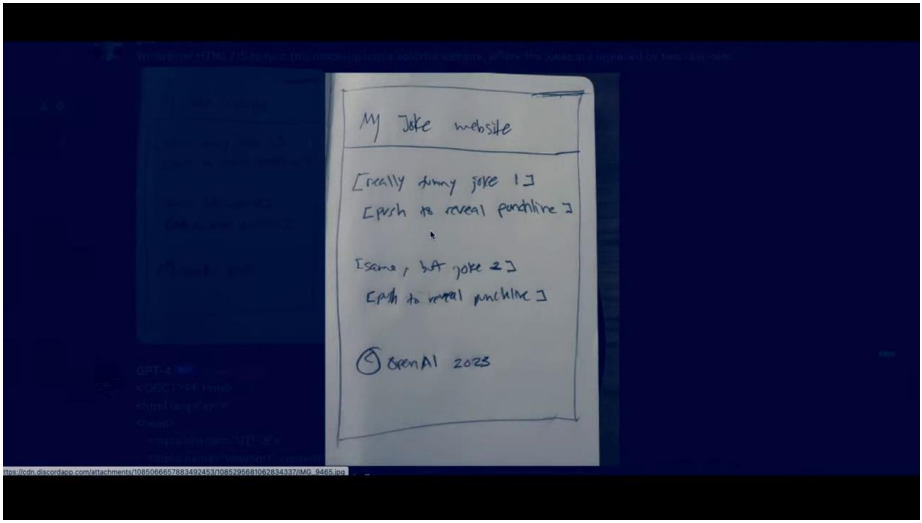
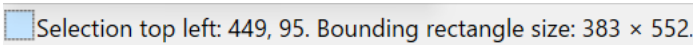


Figure 7-6 Picture of a notebook displaying the mock of a basic website

Let's open the frame in Paint.net, and grab the area to crop:



We can leverage our OCR assistant to write the python code to extract the selection.

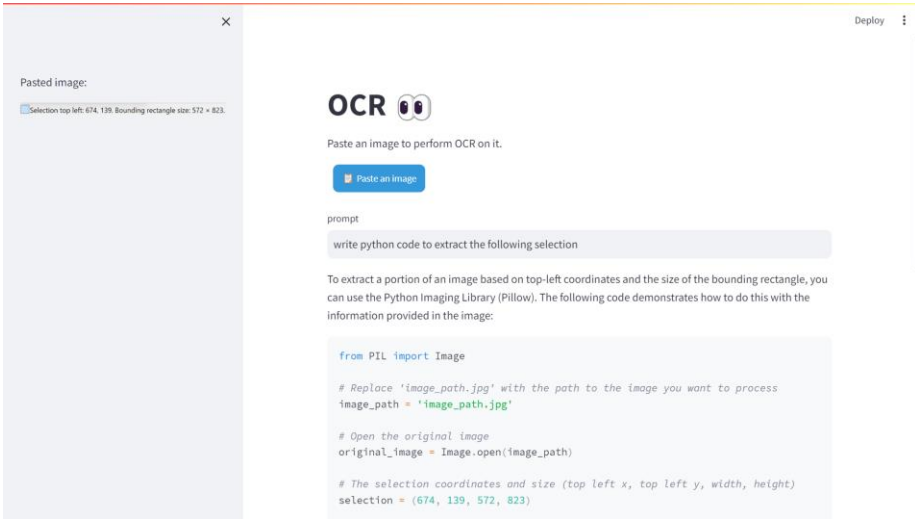


Figure 7-7 OCR assistant extracting python code

```
 from PIL import Image

 # Open the original image
 original_image = Image.open(frame_path)

 # The selection coordinates and size (top left x, top left y, width,
height)
 selection = (449, 95, 383, 552)

 # Calculate the bottom right coordinates (x2, y2)
 x1, y1, width, height = selection
 x2 = x1 + width
 y2 = y1 + height

 # Use the crop method to extract the area
 cropped_image = original_image.crop((x1, y1, x2, y2))

 # Save or display the cropped image
 cropped_image_path = '../img/cropped_image.jpg'
 cropped_image.save(cropped_image_path)
 cropped_image
```

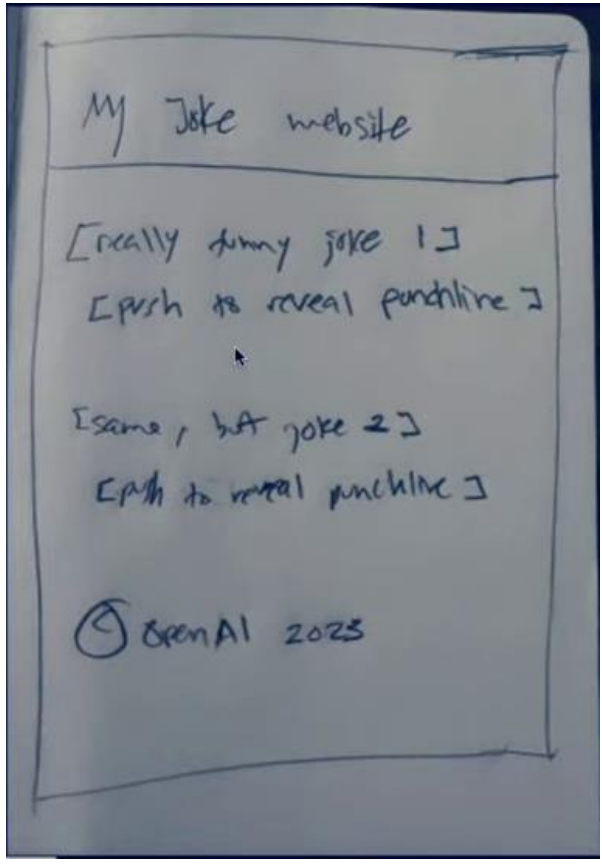


Figure 7-8 Cropped view of the notebook

Let's reuse the prompt from the demo:

```
 original_prompt = "Write brief HTML/JS to turn this mock-up into a
colorful website, where the jokes are replaced by two real jokes."
 prompt = original_prompt + "\n" + "Return only the code without
markdown formating."

 base64_image = encode_image(cropped_image_path)
 code = chat_vision(prompt,base64_image)

 with open("joke_website.html","w") as f:
     f.write(code)
```

And Voila! We have our website coded for us (the resulted html is served up as a GitHub page):

https://yanndebray.github.io/programming-GPTs/chap7/joke_website

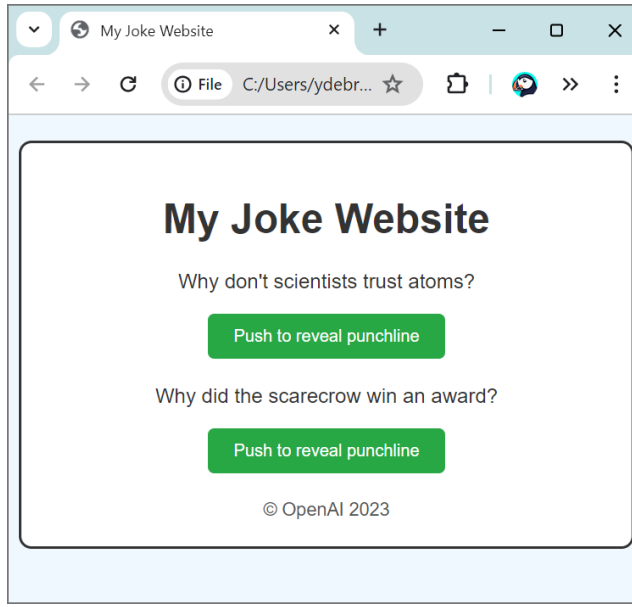


Figure 7-9 Generated website based on the mock

VIDEO UNDERSTANDING

Let's look at another use case: Processing and narrating a video with GPT's visual capabilities and the TTS API[119]. For this example, I will retrieve a video from Youtube that I created, and add a different voiceover.

```
from IPython.display import display, Image, Audio
from pytube import YouTube
import cv2, base64, time, openai, os, requests

video_id = "lBaEO836ECY"
```

[119] https://cookbook.openai.com/examples/gpt_with_vision_for_video_understanding

```
 video_path = youtube_download(video_id,
quality="lowest",path="../data/video")

 video = cv2.VideoCapture(video_path)

 base64Frames = []
 while video.isOpened():
     success, frame = video.read()
     if not success:
         break
     _, buffer = cv2.imencode(".jpg", frame)
     base64Frames.append(base64.b64encode(buffer).decode("utf-8"))

 video.release()
 print(len(base64Frames), "frames read.")
```

```
2134 frames read.
```

```
 display_handle = display(None, display_id=True)
 for img in base64Frames:
      display_handle.update(Image(data=base64.b64decode(img.encode("u
tf-8"))))
      time.sleep(0.025)
```

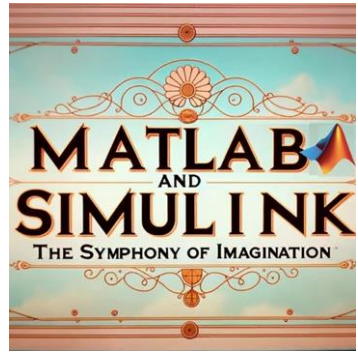


Figure 7-10 Last frame from the video displayed at the end of the loop

```
 PROMPT_MESSAGES = [
     {
         "role": "user",
         "content": [
              "These are frames of a movie trailer. Create a short
voiceover script in the style of Wes Anderson. Only include the
narration.",
```

```
                    *map(lambda x: {"image": x, "resize": 512},
base64Frames[0::60]),
            ],
        },
    ]
    params = {
        "model": "gpt-4-o",
        "messages": PROMPT_MESSAGES,
        "max_tokens": 500,
    }

    result = openai.chat.completions.create(**params)
    script = result.choices[0].message.content
    print(script)
```

"In a world where equations come to life... there exists an extraordinary young man, Max.

Max, played by Timothée Chalamet, is no ordinary scientist. His experiments defy the ordinary, boldly wander into the strange, and occasionally, become magical.

With analog computers and whimsical algorithms, Max teams up with the peculiar yet brilliant Professor Anderson, portrayed by Bill Murray.

Their journey through the enigmatic realms of MATLAB and Simulink unfolds in a symphony of colorful chaos, eclectic technologies, and serendipitous discoveries.

A journey where each computation sparks a revelation, every calculation a wonder.

From the visionary mind of Wes Anderson comes 'MATLAB and Simulink: The Symphony of Imagination'— pulling the strings of reason, and plucking the notes of creativity."

```
    response = requests.post(
        "https://api.openai.com/v1/audio/speech",
        headers={
            "Authorization": f"Bearer {os.environ['OPENAI_API_KEY']}",
        },
        json={
            "model": "tts-1-1106",
```

```
        "input": script,
        "voice": "onyx",
    },
)

audio = b""
for chunk in response.iter_content(chunk_size=1024 * 1024):
    audio += chunk
Audio(audio)
```

(Don't try to click play on a book, it's not going to work… go online instead)

```
with open("../data/audio/voiceover.mp3", "wb") as f:
    f.write(audio)
```

Let's use MusicGen[120] from Meta to add some music to the background. The initial video is lasting 71 sec, let's speed it up by a factor of 1.4, and save the result with voiceover.

```
from moviepy.editor import VideoFileClip, AudioFileClip
from moviepy.video.fx.speedx import speedx

video = VideoFileClip(video_path)
# Speed up the video by a factor of 1.4
speed_up_factor = 1.4
video = speedx(video, speed_up_factor)

audio = AudioFileClip("../data/audio/voiceover.mp3")
final_video = video.set_audio(audio)
# Save the modified video
final_video.write_videofile('../data/video/symphony_voiced_over.mp4
', codec="libx264")
```

```
Moviepy - Done !
Moviepy - video ready ../data/video/symphony_voiced_over.mp4
```

[120] audiocraft.metademolab.com/musicgen.html - huggingface.co/facebook/musicgen-large

8. DALL-E IMAGE GENERATION

This whole journey started for me in the first days of December 2022, right after the release of ChatGPT. But since the API to call the GPT-3.5 model did not become available until March of 2023, I played around with the other services provided by OpenAI. Back then, image generation [121] was one of the APIs that could call programmatically a model named DALL-E[122] (already in v2 since august 2022). In parallel other Generative AI models, like the open-source Stable Diffusion[123] or the proprietary Midjourney[124] enabled very realistic and artistic images creation. Since then, DALL-E 3 caught up with those image models and enabled use cases like generating images with text represented in it.

I tried out a few of those image AI tools on fun projects like *Creating Viral Videos With AI*[125]. In the previous chapter, the video in the style of Wes Anderson was completely generated with AI. I even created a mini-series of short videos on TikTok called `Biomachines`. The story was derived from the following prompt:

```
Write a story about humans being biological machines trained by the
experience of life. The story takes place in 2025, after the advent of
large language models
```

In this chapter we will see how to create a graphical novel on the same topic.

[121] https://platform.openai.com/docs/guides/images
[122] https://openai.com/index/dall-e-2/
[123] https://stability.ai/news/stable-diffusion-public-release
[124] https://www.midjourney.com/
[125] https://www.youtube.com/watch?v=pgdmhwtsntk

Figure 8-1 Image generated by Midjourney in a Cyberpunk style

The methods investigated are:

- Generation
- Edits (In- & Outpainting)
- Variation

The capabilities accessible through the ChatGPT app (with the Plus paid plan) are different from what can be done programmatically. For clarity, I'll be explicit in this chapter about the use of the ChatGPT App, and the use of the OpenAI API endpoints. This logo here represents the "GPT" in the ChatGPT app.

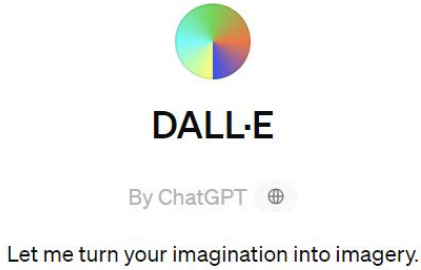


Figure 8-2 Dall-E in the GPT Store

GENERATION

The first step in a story is a character. The difficulty with AI generated image is to maintain consistency in the characters of your plot. With DALL-E in the ChatGPT app, you can iterate and refine your prompt in a conversation with @DALL-E. If you are not happy with a result, you can ask to regenerate the image.

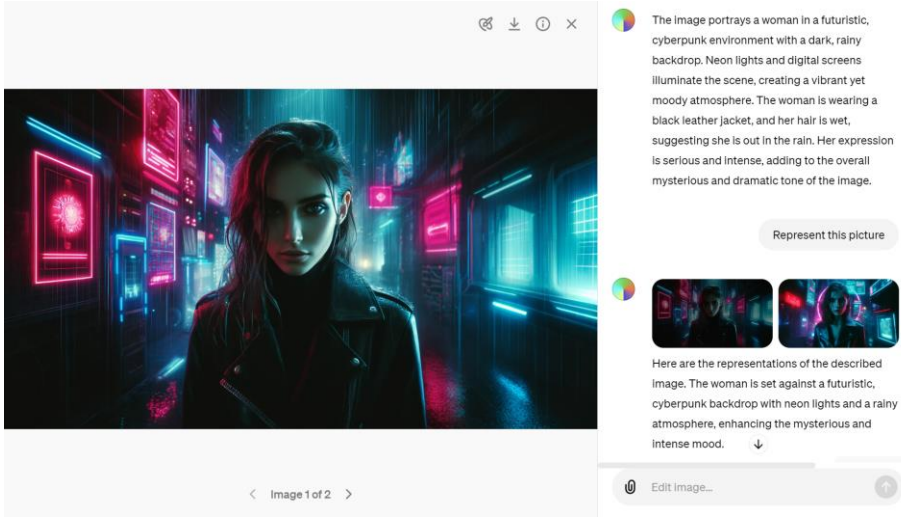


Figure 8-3 Generating images with Dall-E in ChatGPT

With the app you can also upload an existing image to ask ChatGPT to describe it for you. This iterative process is helpful to capture the essence of the scene that you are trying to generate.

The image generations API endpoint allows you to create an original image given a text prompt. With DALL-E 3, images can have a size of 1024x1024, 1024x1792 or 1792x1024 pixels. With DALL-E 2, sizes are 1024x1024, 512x512 or 256x256.

By default, images are generated at standard quality, but when using DALL-E 3 you can set quality: "hd" for enhanced detail. Square, standard quality images are the fastest to generate.

You can request 1 image at a time with DALL-E 3 (request more by making parallel requests) or up to 10 images at a time using DALL-E 2 with the n parameter.

After many hesitations, I decided to use only the module PIL.Image instead of the object `Image` from the module IPython.display

```
from PIL import Image

img_prompt = "An 8-bit pixelated game side scroller called biomachines. cyberpunk story about humans being biological machines. Pixelated like NES. Bright colors. Huge glowing pixels. Pixelated."

response = openai.images.generate(
  model="dall-e-3",
  prompt=img_prompt,
  size="1792x1024",
  quality="standard",
  n=1,
)

image_url = response.data[0].url
print(image_url)
image = Image.open(requests.get(image_url, stream = True).raw)
image.save("biomachines-8bits.png")
image
```

Figure 8-4 An 8-bit pixelated image generated by Dall-E 3

To perform edits and variations, you will need to generate square images (ratio 1:1).

```
# Crop the image in the format 1:1
import Image
image = Image.open("biomachines-8bits.png")
width, height = image.size
left = (width - height) / 2
top = 0
right = (width + height) / 2
bottom = height
im = image.crop((left, top, right, bottom))
im.save("biomachines-8bits_1024x1024.png")
```

EDITS

Edits can be performed interactively with the ChatGPT app by selecting a region in the image:

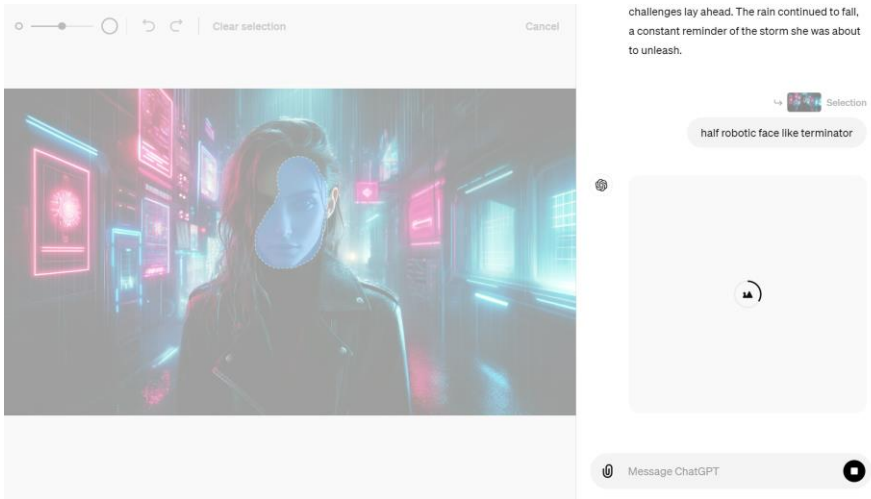


Figure 8-5 Edit images interactively in ChatGPT

Figure 8-6 Modified image of a cyberpunk character

"Inpainting" via the API are only available with DALL-E 2. The image edits endpoint allows you to edit or extend an image by uploading an image and mask indicating which areas should be replaced. The transparent areas of the mask indicate where the image should be edited, and the prompt should describe the full new image, *not just the erased area*.

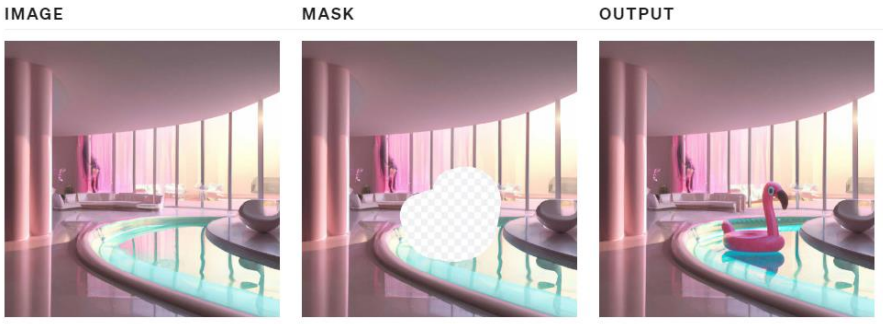


Figure 8-7 Inpainting an image with a mask

To perform this operation manually, you can use Paint.net[126] to erase the masked area in the picture.

```
response = openai.images.edit(
  model="dall-e-2",
  image=open("biomachines-8bits.png", "rb"),
  mask=open("mask.png", "rb"),
  prompt="""
  Terminator giant skull face in a cyberpunk 8 bits pixelated game
  """,
  n=1,
  size="1024x1024"
)
image_url = response.data[0].url
image = Image.open(requests.get(image_url, stream = True).raw)
image.save("biomachines_terminator.png")
```

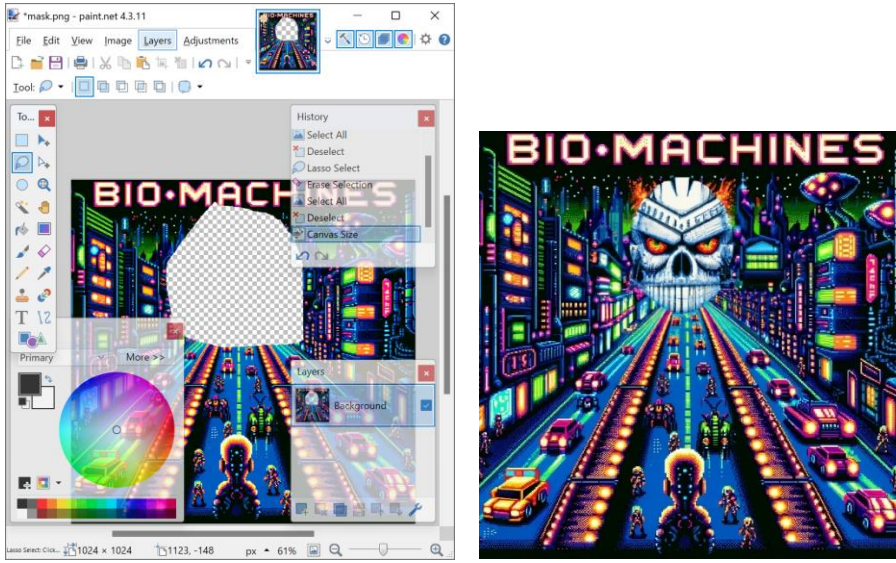


Figure 8-8 Inpainting with a mask created in Paint.NET

"Outpainting" is following the same principal as "inpainting", but you only need to shift the image outside of the canvas in the direction that you want to fill.

[126] https://www.getpaint.net/

Before DALL-E got available in the ChatGPT app, you could access it in a lab environment to “outpaint”.

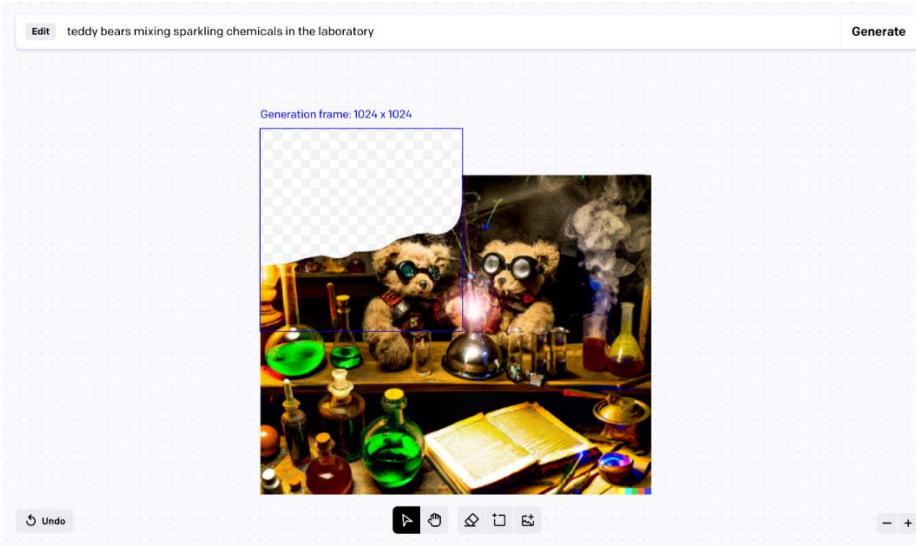


Figure 8-9 Outpainting in OpenAI labs

Since this app is no longer available, we can use a Streamlit app with a drawable canvas[127] that will fulfill some of those needs.

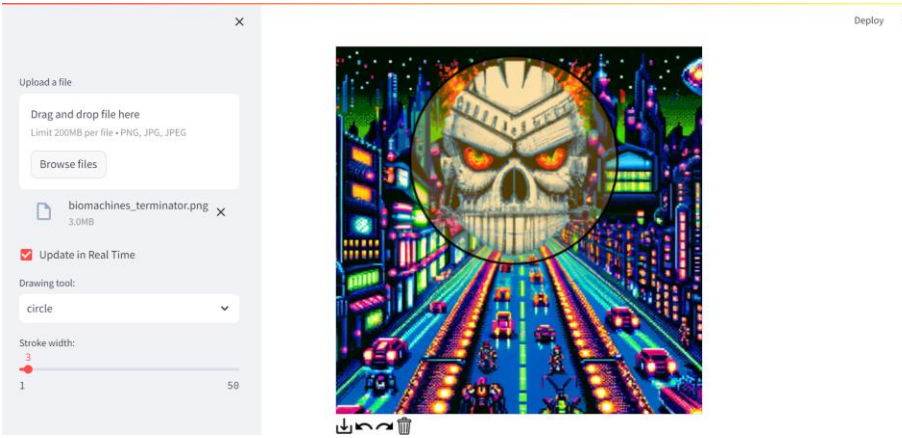


Figure 8-10 Inpainting with a Streamlit drawable canvas

[127] https://github.com/andfanilo/streamlit-drawable-canvas

VARIATIONS

Variations are easy to understand. You input an image of the right format, and get another flavor of it:

```
from PIL import Image

response = openai.images.create_variation(
  model="dall-e-2",
  image=open("biomachines_car.png", "rb"),
  n=2,
  size="1024x1024"
)

image_url = response.data[0].url
print(image_url)
image = Image.open(requests.get(image_url, stream = True).raw)
image.save("biomachines_car_variation.png")
image
```

You can get several variants, by entering n=2 for instance.

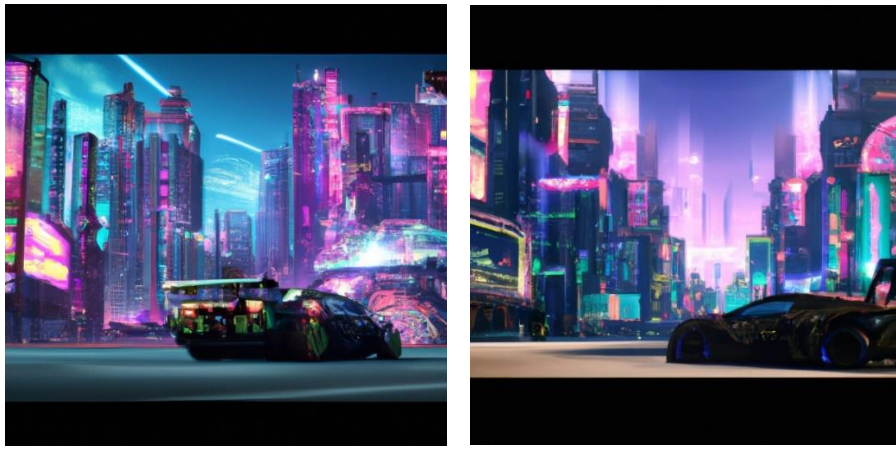


Figure 8-11 Variants of an image with Dall-E 2

APPLICATION: GPT JOURNEY – CHOOSE YOUR OWN ADVENTURE

Imagine playing a text-based game where you can not only read the story, but also see the images generated by your choices. That's what GPT Journey offers: a choose your own adventure game powered by artificial intelligence. This concept was developed by the genius Sentdex on his Youtube channel[128]. You can also retrieve the code on Github: https://github.com/yanndebray/GPT-Journey

GPT Journey uses Dall-E to create images from natural language queries, and ChatGPT to generate interactive dialogues with characters and scenarios. In this section, you will learn how to build your own version of GPT Journey using Python and Flask. You will see how to:

- Create coherent dynamic text responses based on user choices with GPTs
- Create images from text descriptions to produce an engaging game experience with Dall-E
- Build a Streamlit app to host the game online (original version in Flask)

By the end of this section, you will have a fun and creative game that showcases the power of AI for storytelling. Let's get started!

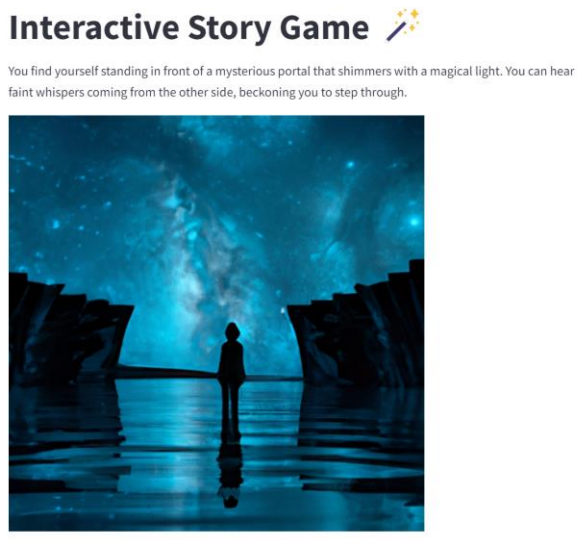
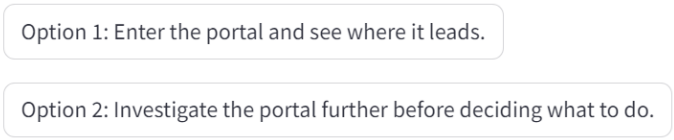


Figure 8-12 GPT Journey app

[128] https://www.youtube.com/watch?v=YY7LIEHiAfg

If you scroll down you will see several options (typically varying from 2 to 4):



Try it out and see where it leads you … Adventure is out there!

9. Deploying GPTs

In this final chapter, we are taking the last step in the journey with GPTs. We will deploy your AI agents, either on the OpenAI GPT Store, or on your preferred hosting service (in my case Streamlit). Those two deployment options are not mutually exclusive, but as you can see on Figure 9-1, they adopt a different architecture, between the front-end, the AI agent, and the tools at its disposal:

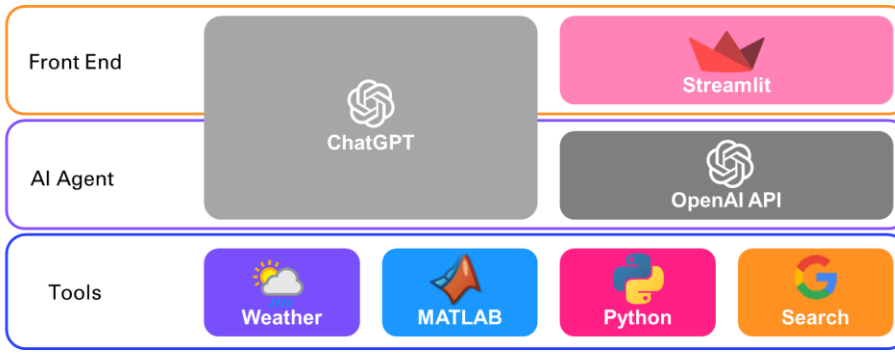


Figure 9-1 GPTs deployment architecture options

/!\ Note that the tools can either be deployed as independent microservices, or as Python functions within the Streamlit app in the second option.

Deploy on the OpenAI GPT Store

In this section, we will see how you can use the GPT builder in the ChatGPT App, and how you can deploy your custom GPTs built in the OpenAI platform on the GPT Store.

Start by asking yourself what is the personal or professional tasks for which you want to design your own AI agent. If the task requires certain tools, you will see how to equip your agent with those tools as web services that the GPT will call.

- Create a custom GPT with the GPT builder

Log in to your ChatGPT account to create your custom GPTs using a low code builder:

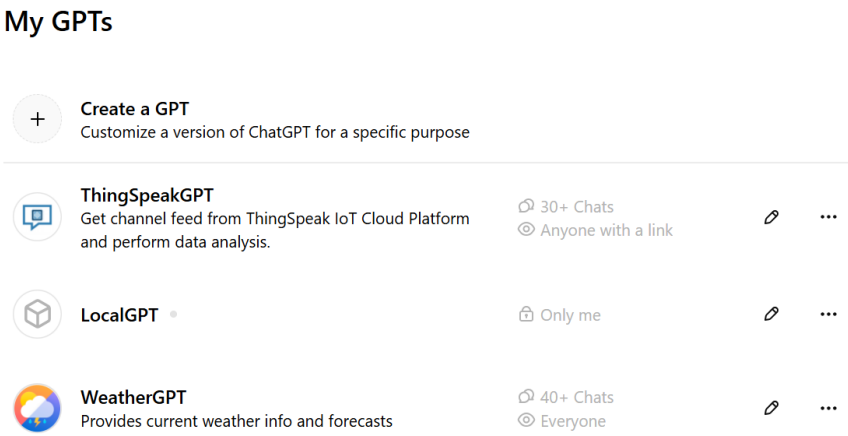


Figure 9-2 List of my Custom GPTs

WeatherGPT is in the GPT Store: https://chatgpt.com/g/g-HB1PWjLVs-weathergpt

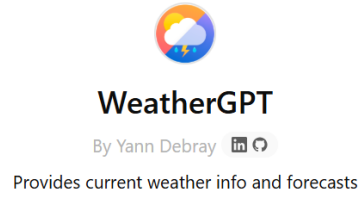


Figure 9-3 My WeatherGPT

Here is an example of what it does:

Figure 9-4 Conversation inside ChatGPT with my WeatherGPT

You can create your own GPT with the low-code GPT builder in the ChatGPT app:

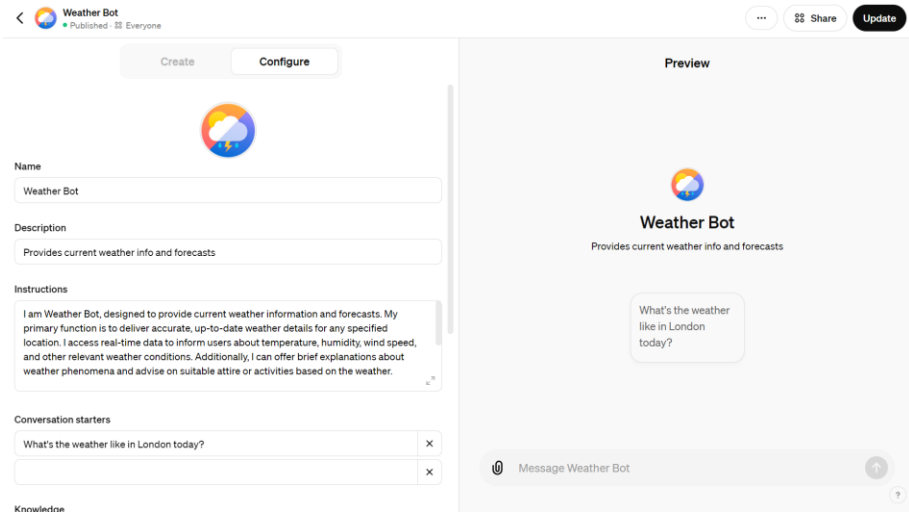


Figure 9-5 Create a new custom GPT without coding

However, what you can achieve through a custom GPT without coding is fairly limited compared to what you have learned in this book.

- Extend the capabilities of your custom GPT access to web services

The secret ingredient in my Weather GPT is the ability to call a weather service. For this I'm using an *Action*. To find out more about actions[129], you can ask help to ActionsGPT.

Figure 9-6 ActionsGPT

Actions are specified in the OpenAPI format (don't confuse with OpenAI without a P):

Schema `openapi.yml`

```
openapi: 3.1.0
info:
```

[129] https://platform.openai.com/docs/actions/introduction

```
    title: Weather
    description: Get weather data for a given city.
    version: "v1"
servers:
    - url: https://weather-plugin-yanndebray.replit.app/
paths:
    /weather:
        get:
            operationId: getWeatherData
            summary: Retrieves the weather data.
            parameters:
                - in: query
                  name: city
                  schema:
                      type: string
                  description: For example, London,uk.
            responses:
                "200":
                    description: OK
```

I have implemented a very simple tool based on this specification that gets the weather with the Flask web framework[130]. You can simply ask ChatGPT to generate the Flask server for you based on the schema above. For this demonstrator, I've deployed my `get_weather_data` function with Replit[131]:

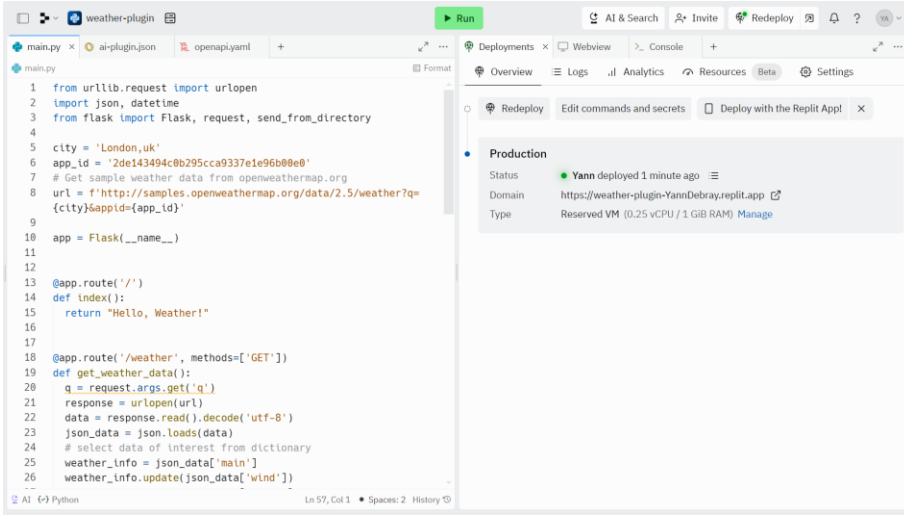


Figure 9-7 Deploy action in Replit

[130] Flask web framework: https://flask.palletsprojects.com/
[131] Replit online development environment: https://replit.com/

Another hosting service to deploy multiple tools as web services is Render[132]. Render has a generous free tier that enables you to deploy multiple web services, as long as their combined runtime is capped at 750 instance hours per month. Free web services automatically spin down after 15 minutes of inactivity and spin back up upon receiving a new request. This behavior can introduce delays due to cold starts when the service is accessed after being idle. I still recommend this service to deploy your GPT tools because it is almost as easy to use as Replit, and the free tier lets you ramp up on GPTs before you feel compelled to upgrade your service to a higher availability (hopefully because you have plenty of users hitting your GPT in the OpenAI store).

Finally, this is how it looks like to call a weather tool from my WeatherGPT, in debug mode to test the endpoint:

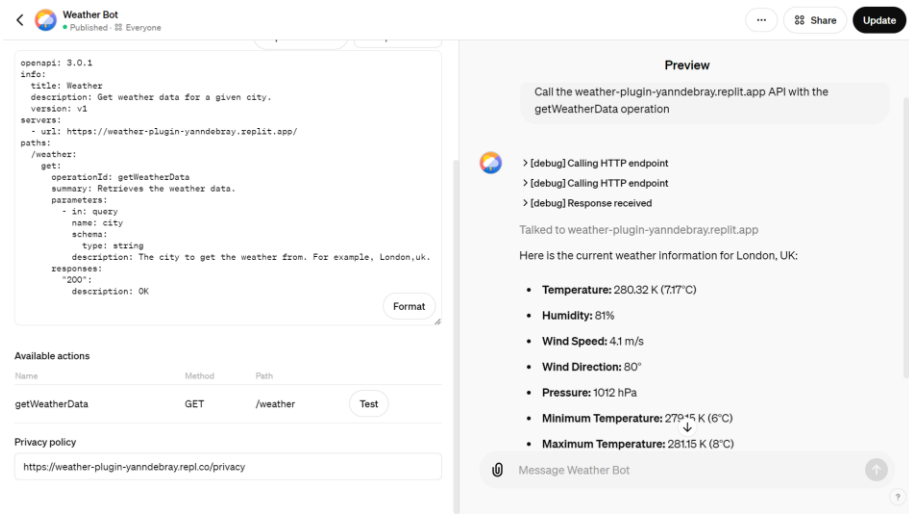


Figure 9-8 Call weather endpoint from your custom GPT

Deploy GPT Apps on Streamlit Cloud

In each chapter, you have seen one of the extended capabilities of the OpenAI API. Let's summarize all of those into one multipage app. This should act as a companion to this book, and a template for you to start your own AI-powered product (you just have to bring your own OpenAI API key). I'll personify each of the GPTs that we will go through in this app and give each of them a funny name, as shown in Figure 9-10.

[132] Render hosting platform: https://render.com/

Figure 9-9 Each GPT is illustrating one of the chapter in the book

- chatty:

Our good old chat, with the memory of the conversation saved as session state. This one is important to get the basics of how to call an LLM. In the appendix, we will see a version of chatty calling Mistral AI.

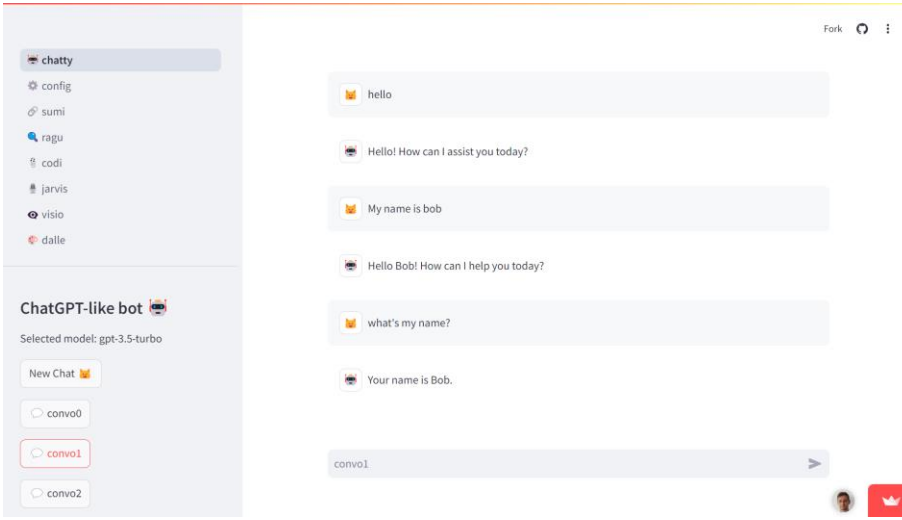


Figure 9-10 Chatty the OG

- config:

This isn't really a GPT, but more an extension of chatty. It enables you to configure your bot, export and delete the history of messages. This way if you want to host this app online, you can give basic privacy settings to your users.

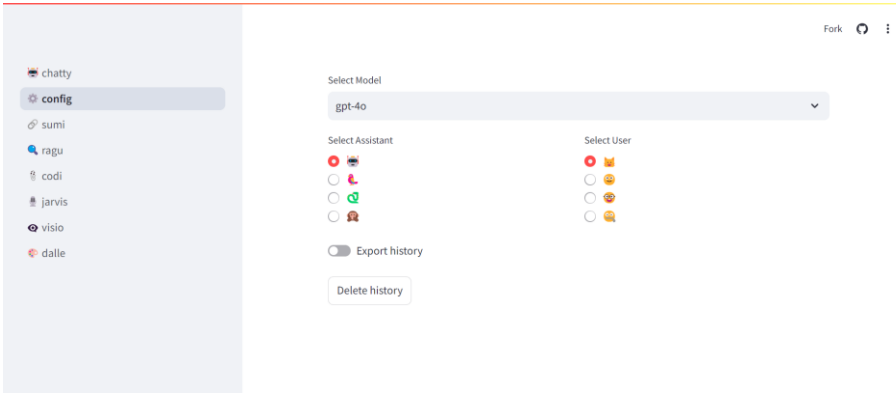


Figure 9-11 Config for any GPT

- 🔗 sumi:

Summarizing is a useful use case of LLMs, and it is one of the first GPTs I created when the GPT-3.5 API came out. This version is more meant for pedagogical objective to explain how the text is broken down into chunks, and to see the impact of changing the chunk size, for instance from 1000 to 900 on chapter 8 that represents 1967 tokens in total. You can also try it out on longer chapters, to observe how the summary gets refined over the iterations.

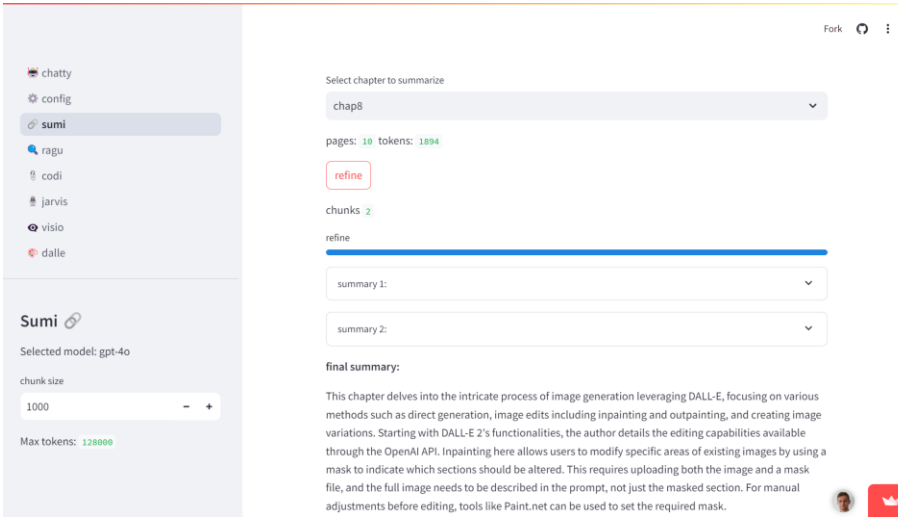


Figure 9-12 Sum of all the things you've learned

- 🎾 ragu:

This GPT uses a vector database that contains the chapters of this book to answer questions. You can derive your own version with the database of your choice.

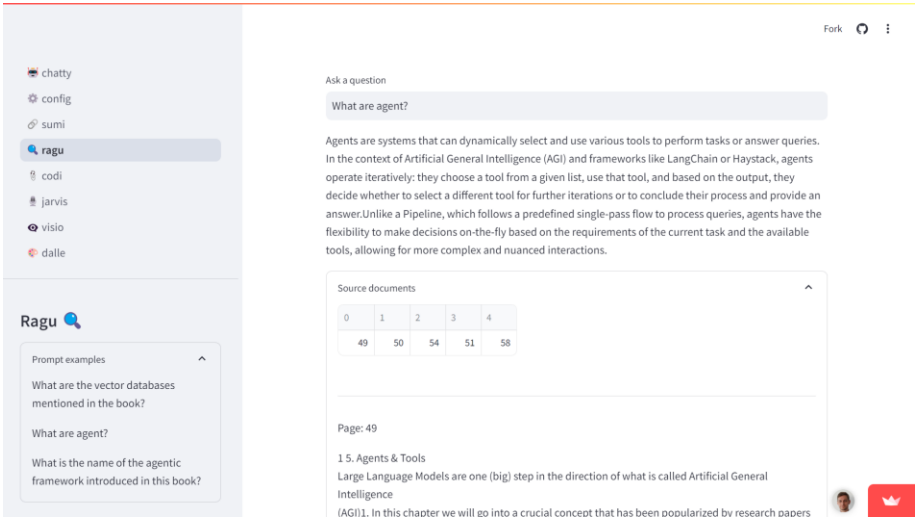


Figure 9-13 Retrieval to ground responses into the context of this book

- codi:

This GPT is one of my favorites. It is less capable than the version described in chapter 5 that enables file inputs for advanced data analysis.

Figure 9-14 Codi is the new clippy to help you code

- jarvis:

This is probably the prototypical chatbot that most of us think about when building our own GPT. It's a variation of chatty with added bells and whistles to take your

voice as input. I didn't go the extra mile of adding a voice synthesis of the bot response, as I was worried it would bring me to the "uncanny valley". This expression represents the strange feeling that humans experience when an AI tries to mimic human functions but does not quite get it right (like when you have a weird looking human image generated).

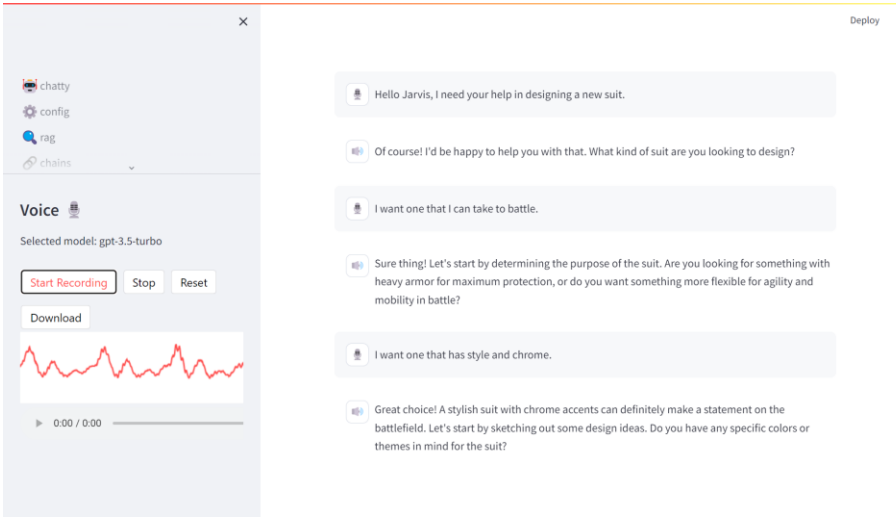


Figure 9-15 Voice is a transformative interface for AI

- visio:

This one sounds like a superhero name. And it does translate some of the feeling of being able to extract information from images. When this capability will scale to take video as input, it will open up a whole new dimension of copilot use cases.

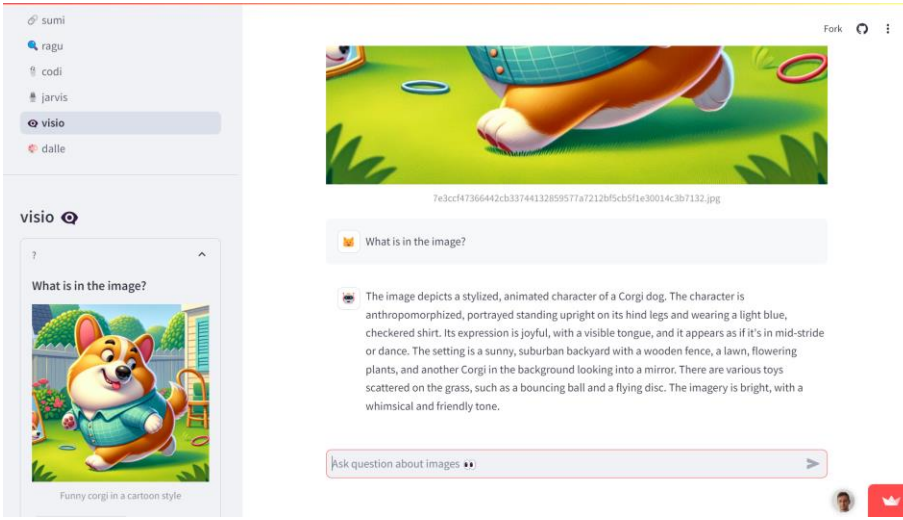


Figure 9-16 Vision for when you don't want to type

- dalle:

This one is more artistic but can be useful for you to generate logos and images like the one on the cover of this book. If your application of AI is more for artwork or entertainment, this GPT is a good start.

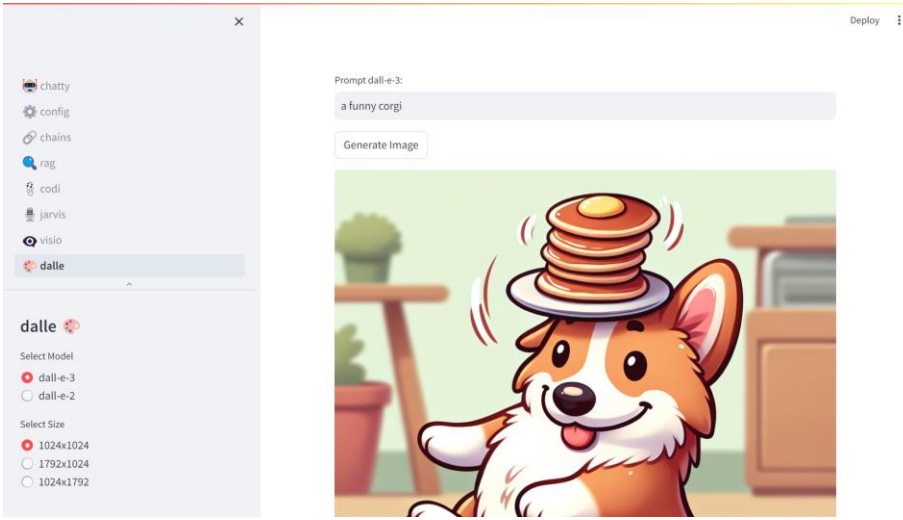


Figure 9-17 To reveal the artist in you

10. APPENDIX

MORE AI THEORY

MACHINE LEARNING

Machine learning is a branch of artificial intelligence that enables computers to learn from data and experience, without being explicitly programmed for every possible scenario. Machine learning can be used to solve complex problems that are hard to codify with rules, such as image recognition, natural language processing, or recommendation systems.

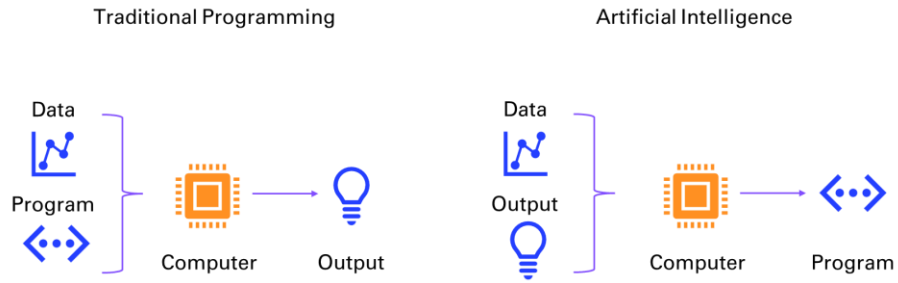


Figure 10-1 Traditional programming vs AI

Machine learning can be broadly divided into two categories: supervised and unsupervised learning. Supervised learning is when the computer learns from labeled data, that is, data that has a known output or target. For example, in image recognition, the computer learns from images that are labeled with their corresponding categories, such as cat, dog, or car. The goal of supervised learning is to train the computer to predict the correct output for new data that it has not seen before.

Supervised learning can be further divided into two types: regression and classification. Regression is when the output is a continuous value, such as temperature, price, or speed. Classification is when the output is a discrete category, such as spam or not spam, positive or negative, or one of several classes.

Unsupervised learning is when the computer learns from unlabeled data, that is, data that has no known output or target. For example, in text analysis, the computer learns from documents that are not labeled with any topic or sentiment. The goal of unsupervised learning is to discover hidden patterns or structures in the data, such as clusters, outliers, or features.

Unsupervised learning can be mainly divided into two types: clustering and dimensionality reduction. Clustering is when the computer groups similar data points together based on some measure of similarity or distance, such as k-means, hierarchical clustering, or Gaussian mixture models. Dimensionality reduction is when the computer reduces the number of features or dimensions of the data, while preserving as much information as possible, such as principal component analysis, linear discriminant analysis, or autoencoders.

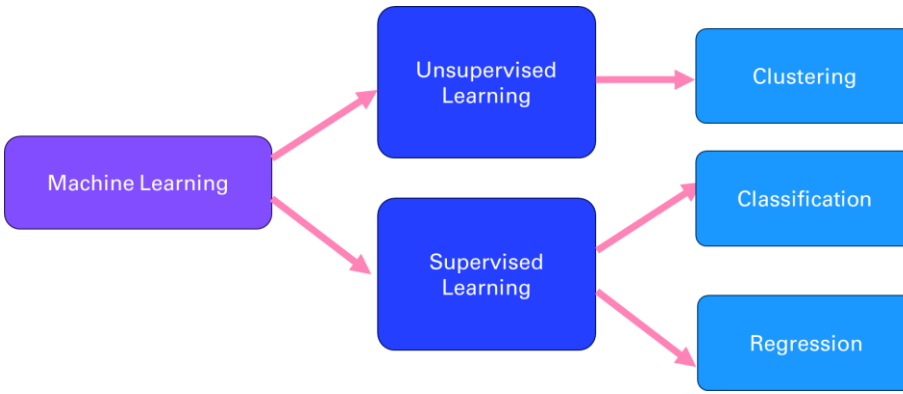


Figure 10-2 Different branches of Machine Learning

Deep Learning

Deep learning is a branch of machine learning that uses neural networks with multiple layers to learn from data in a hierarchical manner. Neural networks are computational models that mimic the structure and function of biological neurons, which can process and transmit information through connections called synapses. Deep learning can handle complex and high-dimensional data, such as images, speech, or natural language, and perform tasks such as object recognition, speech recognition, natural language processing, or machine translation. Deep learning is inspired by the discoveries of neuroscience and cognitive science, and relies on advances in mathematical optimization, parallel computing, and big data.

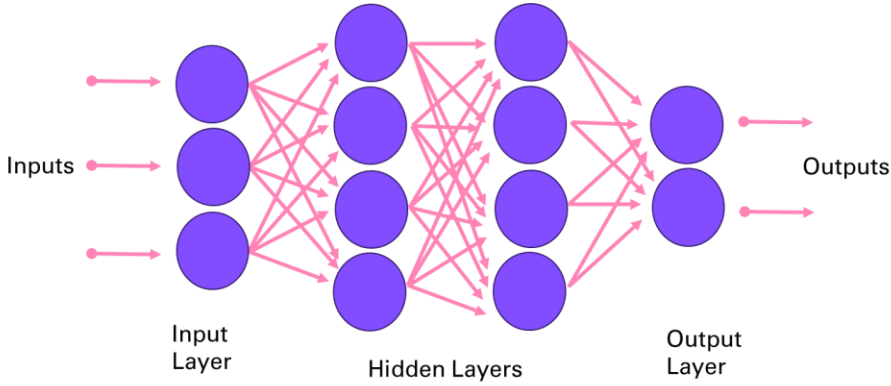


Figure 10-3 Deep Neural Network architecture

A few key research breakthroughs made possible the innovations presented in this book, starting with *Yann LeCun et al* (1989) Backpropagation Applied to Handwritten Zip Code Recognition[133]. This paper that is the same age as I introduced way back the potential of neural network for image processing, on the famous MNIST dataset[134].

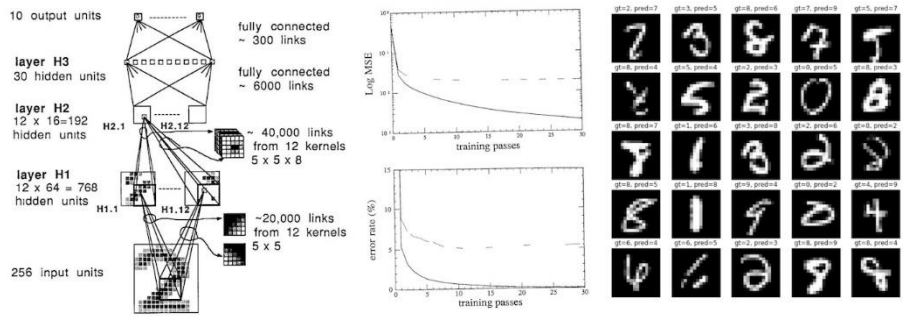


Figure 10-4 LeNet applied to digits recognition

In the world of deep learning, TensorFlow and PyTorch are two of the most widely used frameworks. Both provide the tools needed to build, train, and deploy deep neural networks, but they differ in design philosophy and use cases.

- *TensorFlow*, released by Google in 2015 focuses on Flexibility and Scalability
- *PyTorch*, developed by Facebook seduces as Pythonic and Research-Friendly

[133] http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf
[134] http://yann.lecun.com/exdb/mnist/

Natural Language Processing

Natural language processing (NLP) is the field of artificial intelligence that deals with understanding and generating natural language, such as text or speech. NLP has many applications, such as question answering, sentiment analysis, machine translation, summarization, dialogue systems, information extraction, and more. NLP faces many challenges, such as ambiguity, variability, complexity, and diversity of natural language.

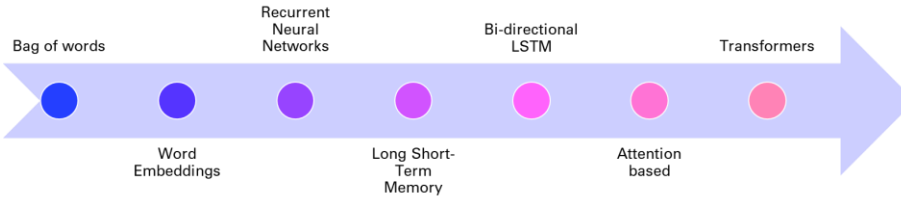


Figure 10-5 History of NLP techniques

One of the key tasks in NLP is to represent natural language in a way that computers can understand and manipulate. Traditionally, this was done by using rule-based or statistical methods to extract features from words, such as their part-of-speech, syntactic structure, semantic role, or frequency. However, these methods often require a lot of human effort and domain knowledge and cannot capture the rich and dynamic nature of natural language.

To overcome these limitations, deep learning methods have been developed to learn distributed representations of natural language, also known as embeddings, from large amounts of data. As mentioned in chapter 4, embeddings are vectors that encode the meaning and usage of words or sentences in a low-dimensional space, and can be used as input or output for various NLP tasks. Embeddings can capture the semantic and syntactic similarities and relationships between words or sentences, and can also adapt to new domains and languages.

One of the first methods to learn word embeddings was the bag-of-words model, which represents a document as a vector of word frequencies, ignoring the order and context of words. The bag-of-words model is simple and efficient, but it suffers from sparsity, dimensionality, and lack of semantics. To address these issues, neural network models such as word2vec and GloVe were proposed to learn word embeddings from the co-occurrence patterns of words in large corpora, using

techniques such as skip-gram and negative sampling. These models can learn more expressive and dense word embeddings, but they still treat words as independent units, ignoring their morphology and compositionality.

To account for the sequential and hierarchical structure of natural language, recurrent neural networks (RNNs) were introduced to learn sentence or document embeddings from word embeddings. RNNs are neural networks that process sequential data by maintaining a hidden state that captures the history of previous inputs. RNNs can learn long-term dependencies and generate variable-length outputs, making them suitable for tasks such as language modeling, machine translation, or text generation. However, RNNs also face some challenges, such as vanishing or exploding gradients, difficulty in parallelization, and sensitivity to noise.

To improve the performance and stability of RNNs, variants such as long short-term memory (LSTM) were developed to introduce gates that control the flow of information in the hidden state. These gates can learn to remember or forget relevant or irrelevant information over time and can handle long-term dependencies better than vanilla RNNs. LSTM have achieved state-of-the-art results on many NLP tasks, such as machine translation, speech recognition, or sentiment analysis.

However, even LSTM have some limitations, such as the inability to model long-range dependencies beyond a fixed window, the sequential nature of computation that limits parallelization, and the lack of attention mechanisms that can focus on relevant parts of the input or output. To overcome these limitations, a new paradigm of neural network models was proposed, based on the concept of transformers.

TRANSFORMERS

You might be wondering what's happening under the hood of a GPT model. Let's dive into the different parts of the model architecture discussed in chapter 1. Figure 10-5 shows the components of generative pre-trained transformers:

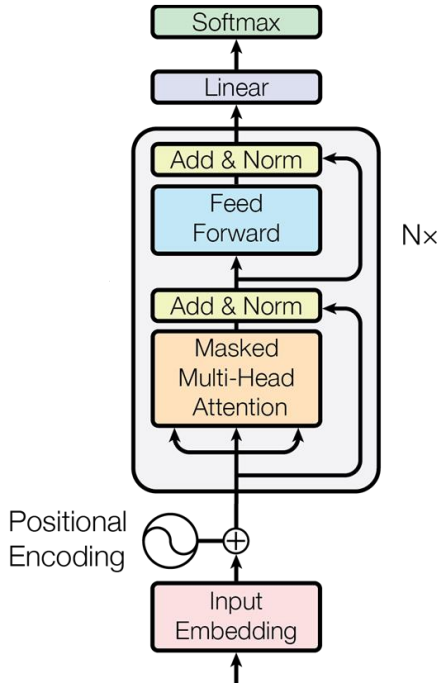


Figure 10-6 Architecture of a transformer as introduced in the paper "Attention Is All You Need"

Input Preprocessing: From Words to Vectors

The preprocessing of the input text proceeds in three steps:

1. *Tokenization:* Turning the words into numbers
2. *Embedding:* Compressing the numbers into a dense vector representation
3. *Positional Encoding:* Storing the position of the words in the sentence

Each step - from tokenization to embedding to positional encoding - plays a critical role in enabling the model to interpret text. Let's take a closer look at these preprocessing steps and their importance in making sense of how GPT models parse their input.

Tokenization

AI models like GPTs cannot directly understand or process raw text as humans do. Computers only understand numbers. The first step (not shown in Figure 1-7) is breaking down the text into smaller units called *tokens*. These tokens typically represent words, subwords, or even characters, depending on the tokenization strategy. The tokenization step converts the input text into a sequence of tokens.

Embedding

Once tokenized, each token is then transformed into a dense vector of numbers. This process is called embedding. An *embedding* is a learned representation of the tokens in a continuous vector space where semantically similar tokens are closer together. This numeric representation is what the model actually processes.

Positional encoding

Unlike traditional models, transformers (like GPT) do not have a built-in sense of the order of tokens in a sequence. To provide this information, *positional encoding* is added to the token embeddings. This encoding incorporates information about the position of each token in the sequence, allowing the model to understand the order and relationships between tokens over different positions.

Decoder Block: Attention (Plus Compute) Is All You Need

The core of GPT is a stack of decoder blocks, each composed of two main layers, attention and feed forward, as well as addition and normalization layers completing the two main layers.

The decoder blocks are repeated N times (96 decoder blocks for GPT 3) as displayed on Figure 10-5 to form the main building blocks of the transformer architecture.

Attention

GPT uses a variant of attention known as *causal* or *masked* attention. This type of attention ensures that when generating a token, the model can only attend to previous tokens and not future ones. This is crucial for autoregressive generation, where each token is predicted based on the preceding tokens.

To capture different aspects of the input sequence, the model employs *multiple attention heads* that operate in parallel. Each head focuses on different parts of the input sequence, allowing the model to learn multiple representations of the sequence's information. The outputs from all the heads are then combined and passed on to the next layer.

Feed forward

After the attention mechanism comes a layer of pure compute. The output is passed through a feed-forward network, typically a multi-layer perceptron (MLP). This layer applies a non-linear transformation to the data, allowing the model to capture complex patterns and relationships within the data. The feed-forward network operates independently on each position, meaning it processes each token separately but consistently.

Addition and Normalization

After the multi-head attention and feed-forward layers, the output is combined with the input to that layer, symbolized by the link skipping these layers in Figure 1-7. This addition helps in retaining information from earlier layers, mitigating the vanishing gradient problem, and making it easier for the network to learn.

Following the addition step, normalization helps keep the training process stable and efficient by preventing large shifts in the input distributions as the model learns, which could otherwise disrupt or slow down learning.

Output Postprocessing: Back to Words, with some Probabilities

Once GPT models process input text through multiple layers of transformation and understanding, they need a way to translate their internal representations back into readable words or tokens. These final sets of layers are where abstract vector-based representations from the model's hidden layers are transformed into probabilities that guide token selection. The postprocessing steps involve passing the model's output through a linear layer and softmax function, which together assign probabilities to each possible token in the vocabulary, determining the most likely continuation of the text.

Linear layer

The output from the final decoder block is passed through a linear layer, which maps the high-dimensional representation of each token back to the size of the vocabulary. This step produces a raw score (logits) for each possible token in the vocabulary, indicating how likely each token is to appear next in the sequence.

Softmax function

Finally, the logits are passed through a softmax function, which converts them into probabilities. The softmax function ensures that the output values are between 0 and 1 and that they sum to 1 across the vocabulary. The token with the highest probability is selected as the model's output, representing the most likely next token in the sequence.

This architecture remains surprisingly very similar since the original transformers paper. One notable difference is the application of the layer norm before the attention mechanism in more recent transformers.

More LLMs: open-source and local alternatives

Mistral

Mistral AI[135] is a French startup that created an open-source LLM competitive with GPT-3.5 in about 1 year and with a team of 20 engineers. Setting aside the Frenchmanhood, I find this very impressive. The members of the founding team were previously employed at Google DeepMind and at FAIR (Facebook AI Research) working on important projects like the Llama model from Meta.

In only a few hours, I've been able to port my first three applications to be working with the Mistral API (la Plateforme):

- Chat
- Summarization
- Q&A with vector search

To get started with the API, you can use the python client[136]:

```
pip install mistralai
```

Here is how you can convert some of your code from OpenAI to MistralAI:

```
from mistralai import Mistral

model = "mistral-tiny"
client = Mistral(api_key=api_key)

m = [{'role': user, 'content': 'What is the best French cheese?'}]

# No streaming
chat_response = client.chat.complete(
    model=model,
    messages=m,
)

print(chat_response.choices[0].message.content)
```

```
It is subjective to determine the "best" French cheese as it depends
on personal preferences. Some popular and highly regarded French
cheeses are:
```

[135] https://mistral.ai/

[136] https://github.com/mistralai/client-python

```
1. Roquefort: A blue-veined cheese from the Massif Central region,
known for its strong, pungent flavor and distinctive tang.


2. Comté: A nutty, buttery, and slightly sweet cheese from the Franche-
Comté region, made from unpasteurized cow's milk.


3. Camembert de Normandie: A soft, Earthy, and tangy cheese from the
Normandy region, famous for its white mold rind.

…
```

OLLAMA

Download Ollama[137] on your laptop and select the open-source LLMs you want to serve up locally:

```
$ ollama run llama3:8b

pulling manifest

pulling              6a0746a1ec1a...              100%
████████████████████████████████████████████████  4.7 GB

pulling              4fa551d4f938...              100%
████████████████████████████████████████████████   12 KB

pulling              8ab4849b038c...              100%
████████████████████████████████████████████████   254 B

pulling              577073ffcc6c...              100%
████████████████████████████████████████████████   110 B

pulling              3f8eb4da87fa...              100%
████████████████████████████████████████████████   485 B

verifying sha256 digest

writing manifest

removing any unused layers

success
```

Once you successfully retrieved the weights of the model (here 4.7Gb for the 8B Llama3 model), you can start interacting with the command line:

[137] https://ollama.com/

```
>>> Send a message (/? for help)
```

You can also use the Ollama Python client[138] to build local LLMs applications, as an alternative to OpenAI:

```
pip install ollama
```

Depending on your laptop resources (CPU, GPU and RAM) you might have a very slow response compared to what you are used to with GPT-3.5 or 4.

```
import ollama
response = ollama.chat(model='llama3:8b', messages=[
  {
    'role': 'user',
    'content': 'Hello world',
  },
])
print(response['message']['content'])
```

You can also stream the response and observe the throughput latency:

```
import ollama

stream = ollama.chat(
    model='llama3:8b',
    messages=[{'role': 'user', 'content': 'Why is the sky blue?'}],
    stream=True,
)

for chunk in stream:
  print(chunk['message']['content'], end='', flush=True)
```

This kind of setup can be useful for batch workflows where you want to process sensitive information without having to share it with a web service.

[138] https://github.com/ollama/ollama-python

More Applications

Image cropper

To create variations in chapter 8, you first need to crop the input image to the right shape (e.g. 512x512). Here is an app[139] that enables you to interactively crop images.

Figure 10-7 Image Cropper app

You can simply crop the image programmatically as long as you know the location of your top left pixel.

```
import PIL
im1 = PIL.Image.open('../img/dechargement.jpg')
left=0
top=0
right=512
bottom=512
im2 = im1.crop((left, top, right, bottom))
im2.save("../img/dechargement_cropped.jpg")
im2
```

The resulting image can then be used to generate variations like in chapter 8.

```
import io
from IPython.display import Image
# Convert the image to bytes
```

[139] https://github.com/turner-anderson/streamlit-cropper

```
image_bytes = io.BytesIO()
im2.save(image_bytes, format='PNG')

# Use the image bytes in the API call
response = openai.images.create_variation(
  model="dall-e-2",
  image=image_bytes,
  n=1,
  size="512x512"
)

image_url = response.data[0].url
print(image_url)
Image(url = image_url)
```

Before

After

This use case has a particular emotional meaning for me as this painting is hanging in my living room and was authored by my godmother who I love dearly. Seeing a new take on her creative work is lightning a thousand lights in my heart (this expression isn't AI generated, it really is how this makes me feel).

VIDEO ANALYZER

Post process Andrej amazing videos. For example:

> Andrej Karpathy [1hr Talk] Intro to Large Language Models
> https://www.youtube.com/watch?v=zjkBMFhNj_g

- Download the Youtube video

```
from pytube import YouTube

video_id = "zjkBMFhNj_g"

# Define the URL of the YouTube video
url = f'https://www.youtube.com/watch?v={video_id}'

# Create a YouTube object
yt = YouTube(url)

# Download the video in the best quality
video_path = yt.streams.get_highest_resolution().download()
```

You can use a package called MoviePy to extract images from the video:

```
from moviepy.editor import VideoFileClip

time = 42
clip = VideoFileClip(video_path)
clip.save_frame(f"frames/frame_{time}.jpg", t = time)
```

To select manually frames from a Youtube video, I've created the following app: https://video-analyzer.streamlit.app/

- Extract/crop area of the picture (semi-manual process)

Figure 10-8 Manual cropping with Paint.NET

I took a printscreen of the bottom left corner information and ask the Code interpreter to *write python code to crop the following selection*:

`Selection top left: 0, 45. Bounding rectangle size: 1512 x 851`

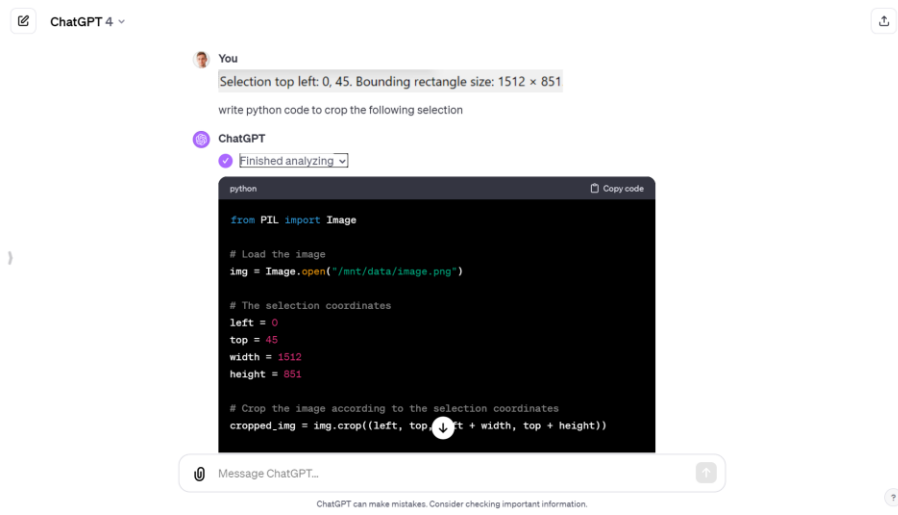


Figure 10-9 Generate code from ChatGPT to crop image

```
# The selection coordinates
left = 0
top = 45
width = 1512
height = 851

# Crop the image according to the selection coordinates
cropped_img = img.crop((left, top, left + width, top + height))

# Save the cropped image
cropped_img_path = f"cropped_frames/cropped_frame_{time}.png"
cropped_img.save(cropped_img_path)
```

- Extracted the audio

```
from moviepy.editor import VideoFileClip

# Define the path of the video file
# video_path = 'path/to/video.mp4'

# Create a VideoFileClip object
video = VideoFileClip(video_path)

# Extract the audio from the video
audio = video.audio

# Save the audio as a separate file
# audio_path = 'path/to/audio.wav'
audio_path = f'./{video_title}.mp3'
audio.write_audiofile(audio_path)
```

- Breakdown into chapters

Finally, I used the GitHub copilot chat panel to breakdown the audio file into chapters (from the chapter structure of the Youtube video):

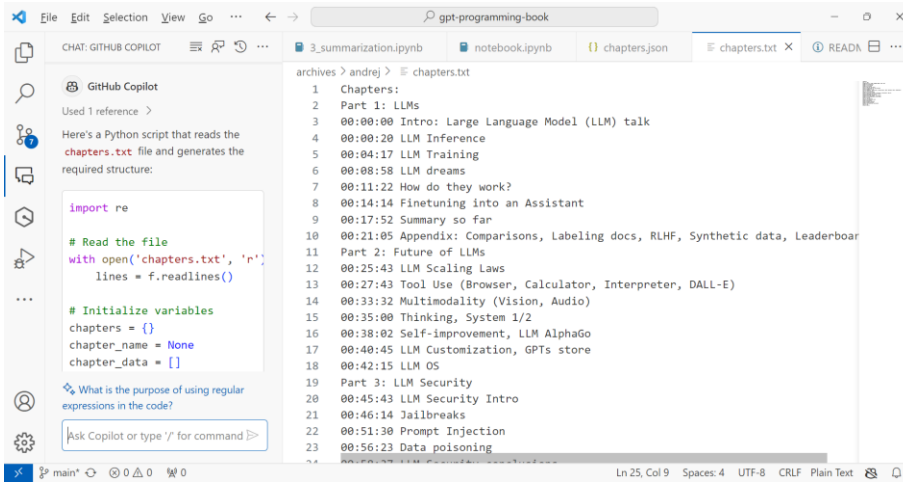


Figure 10-10 Chapter structure of the Youtube video

MOVIE SEARCH AND RECOMMENDATION ENGINE

In April 2023, Andrej Karpathy[140], one of the founding members of OpenAI and former Director of AI at Tesla, shared this fun weekend hack, a movie search and recommendation engine: awesome-movies.life. Here is my version of it:

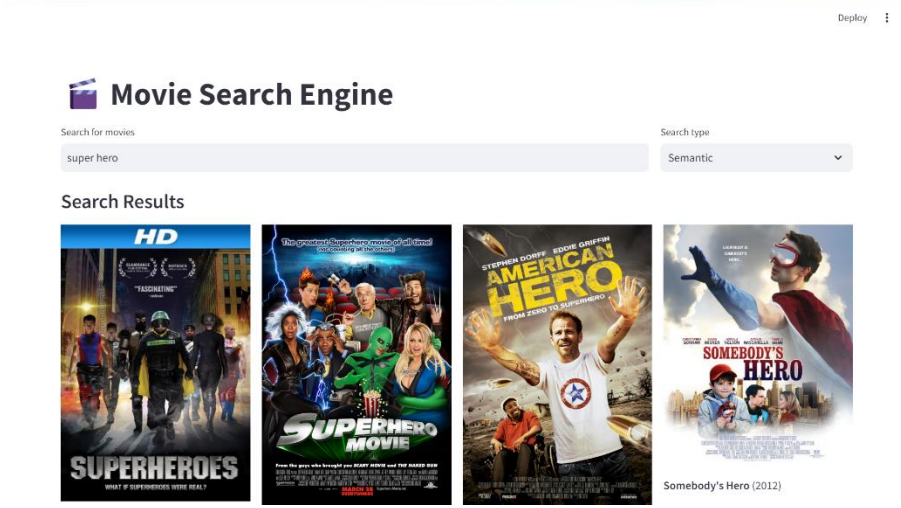


Figure 10-11 Movie search and recommendation engine app

[140] https://karpathy.ai/

Even though Karpathy did not share the code, Leonie – developer advocate at Weaviate reimplemented it[141], using a vector database instead of simply storing embeddings as numpy arrays. So, this is where I took my inspiration from. It is an ideal last project to make sure that you master the notions in this book.

MORE COPILOTS

MICROSOFT COPILOT

As I am thinking about what to write in those next few lines, a little icon appears on the left of the empty line in Word. If I hover over it, it informs me that I can use Alt+I as a shortcut to summon the assistant:

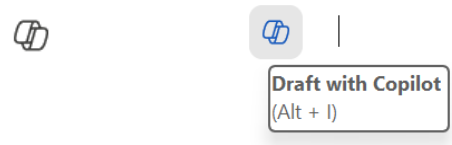


Figure 10-12 Microsoft copilot icon

And if I click on it, I can see that an edit text box appears to await my prompt instructions.

This is a new feature that allows me to write with the help of an AI assistant. The assistant can suggest sentences, paragraphs, or even entire documents based on my input and preferences. I can also ask the assistant questions, give commands, or request feedback. For example, I can type:

- Write a chapter about X.
- Summarize the main points of this document.
- Check my grammar and spelling.

The assistant will try to respond to my requests as best as it can, using the information from the document or the internet. The assistant can also generate tables, charts, images, or other types of media if I ask for them.

[141] Recreating Andrej Karpathy's Weekend Project — a Movie Search Engine https://towardsdatascience.com/recreating-andrej-karpathys-weekend-project-a-movie-search-engine-9b270d7a92e4 - https://github.com/weaviate-tutorials/awesome-moviate

(By the way the previous lines have been generated, and I did only modify about 30% of it, including removing the inexact facts.)

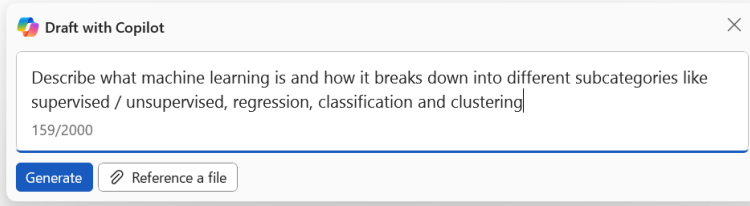


Figure 10-13 Inline chat in Microsoft Word

You have now copilot integrated into most of the Microsoft products. I find them more or less useful, but those capabilities will likely evolve as copilot gets more widely adopted.

GitHub copilot

GitHub copilot is probably the most useful copilot in my mind, as it turned me into a better coder.

You can start with a prompt if you have a clear idea of what you want to develop.

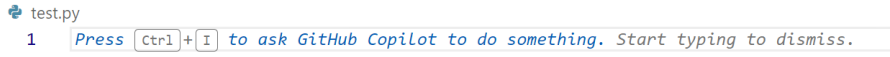


Figure 10-14 Press Ctrl+I

If not, you can dive in the code, and add comments along the way to give hints to copilot on what you want to do next:

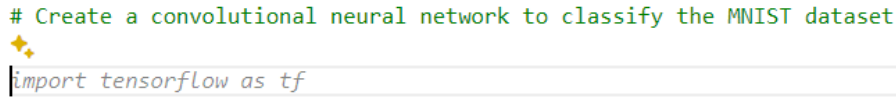


Figure 10-15 Autocomplete based on a comment

All you have to do next is to enter Tab if you are satisfied with the code completion suggested. If not you can hover over the code to see if there are other proposition

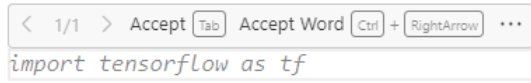


Figure 10-16 Tab to accept completion

You can play with the tab and escape keys as the code get's written for you.

```
# Create a convolutional neural network to classify the MNIST dataset

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Figure 10-17 Coding one tab at a time

I can assure you that this significantly reduces the cognitive load for me (especially because I'm not a professional coder). And I'm even learning a ton along the way. Before copilot, I would Google "How to code X in Python" and spend hours finding the right tutorial or forum post that solves a problem close to mine. Then I would still have to interpolate to my context.

This use case of LLMs generating code is for me by far the most valuable one, and it was enough to fuel my motivation to learn more about GPTs in 2023 and write this book in 2024.