2. The ChatGPT API

On March 1st 2023, OpenAI introduced a programming interface (API) to ChatGPT[1]. This essentially enabled developers to build their own version of ChatGPT. I had been waiting for this ever since the ChatGPT web app got released in the end of 2022. I knew somehow that OpenAI could not keep this amazing technology just to themselves, and that they had to make it a platform for developers to build their own apps. The availability of APIs to the previous generation of models (GPT3, Dall-E, …) made me hope for the day when I could build on top of this Large Language Model.

But when I started programming with this new chat completion API, I started understanding better how the ChatGPT app was working, especially when it comes to storing the state of the conversation. Spoiler alert: the API is stateless: it doesn't store the conversation, and just passes it entirely each time that a new request is made to the model.

2.1. 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.

2.1.1. 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.

Step 1: Install Python

Ensure that Python (version 3.6 or later) is installed on your system. You can download it from the official Python website[2].

Step 2: Set Up a Virtual Environment

This is a good habit if you don't want to mess up completely your Python dev environment. That being said, this blog 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

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

[1] https://openai.com/blog/introducing-chatgpt-and-whisper-apis
[2] www.python.org

Step 3: Install Packages

- OpenAI-Python[3] is the official library provided by OpenAI for interacting with their GPT models..
- LangChain[4] is useful for chaining language model calls and building complex applications.
- Streamlit[5] is a Python library for creating and sharing beautiful, custom web apps.

```
pip install openai langchain streamlit
```

Step 4: 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).

Step 5: 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:

```python
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)
```

Step 6: Streamlit Basic Setup

- Create a simple Streamlit app to test the setup and save it under `1_3_streamlit_setup.py`

```python
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 chap1/1_3_streamlit_setup.py
```

[3] https://github.com/openai/openai-python
[4] https://www.langchain.com/
[5] https://streamlit.io/

2.1.2. Setup the dev environment

To setup VSCode for a Python project, you will need to install some extensions and configure some settings. Here are the steps to follow:

- 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 Pylance extension from the VSCode marketplace. This will enhance the Python language server with type information, signature help, and more.
- 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 and the linter. Alternatively, you can use the command palette and type "Python: Select Interpreter" or "Python: Select Linter". You can choose any linter that you prefer, such as pylint, flake8, or black.
- Write your Python code and enjoy the features of VSCode. You can use the Run > Start Debugging menu or the F5 key to debug your code. You can also use the Test > Run Tests menu or the command palette to run your tests. You can also use the Code > Format Document menu or the Shift+Alt+F key to format your code.

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 an end-to-end full online dev environment.[6]

2.1.3. Develop and deploy a web app with Streamlit

I get to speak about two of my favorites Python frameworks in one blog: OpenAI and Streamlit.

What is 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.

How does Streamlit work? 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[7]. 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.

I have developed and deployed around 60 apps with Streamlit over the past couple of years. I wrote apps to manage my Spotify playlist, map out the houses in Boston suburbs, track the price of Bitcoin,

[6] https://blog.streamlit.io/edit-inbrowser-with-github-codespaces/
[7] https://docs.streamlit.io/library/get-started/main-concepts#app-model

edit videos from YouTube, play back some chess moves, … So this feels like a natural way to bring to live the content of this blog.

2.2. Getting started with the chat completion API

This is what the first lines of code would 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-3.5-turbo',
                                            messages=m)
response = completion.choices[0].message.content
print(response) # world
```

In order to facilitate the storage of our API key, we can store it as a variable "`OPENAI_API_KEY`" in a file `secrets.toml`. 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)
```

As you can see from the messages that are passed to the chat completion API, the structure expected is a list of dictionaries[8], with each entry containing a role (either system, user or assistant) and a content entry. 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 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. For the first version of GPT-3.5[9], the number of tokens accepted was 4,096 tokens (inputs and outputs tokens are summed). In some cases, it's easier to show the model what you want rather than tell the model what you want. One way to show the model what you want is with faked example messages. This is what is called few-shot prompting.

The message contained in the chat completion object[10] that is returned by the API can be appended to the message list to iterate on the next question to the assistant:

```
{"role": "assistant", "content": "world"}
```

[8] https://cookbook.openai.com/examples/how_to_format_inputs_to_chatgpt_models
[9] https://platform.openai.com/docs/models/gpt-3-5
[10] https://platform.openai.com/docs/api-reference/chat/object

Additional parameters can be passed to the chat completion function[11] (like max number of tokens, number of choices to generate, and stream option).

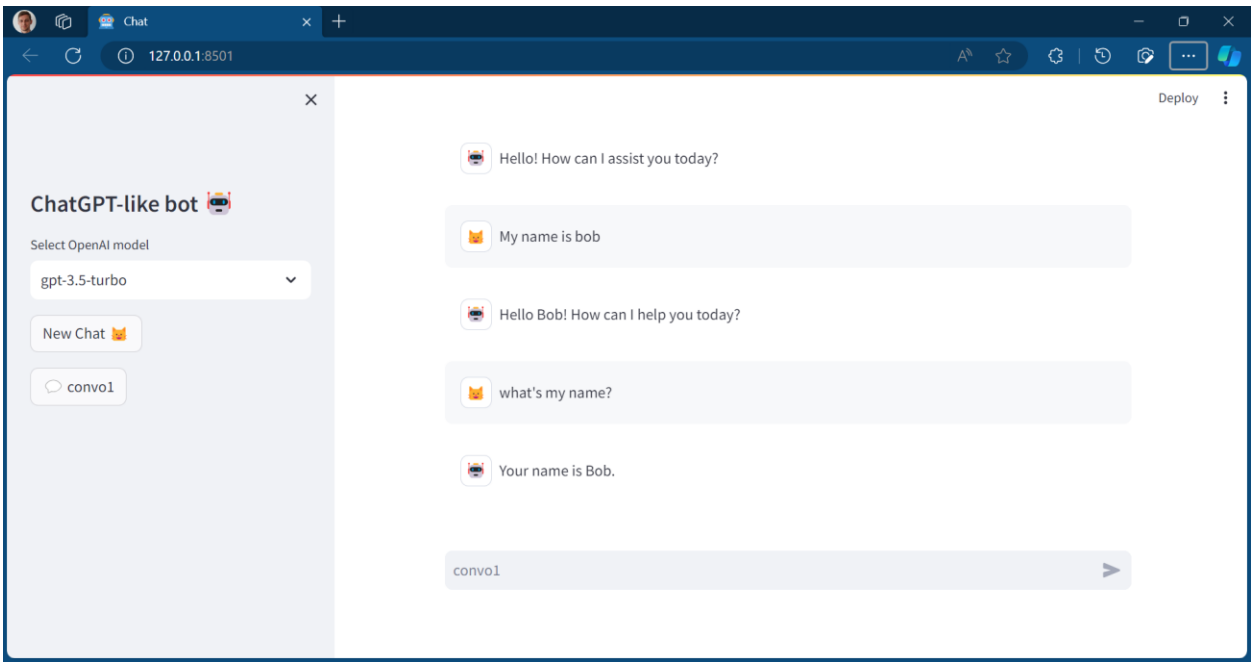
2.3. Prices of the API

The prices[12] have evolved quite a bit since the introduction of the ChatGPT API. As of November 2023:

- GPT 3.5 turbo costs $0.001 / 1k input tokens, $0.002 / 1k output tokens.
  This model was introduced at 1/10 of the price of the davinci legacy GPT 3 model.
- GPT 4 (8k version) costs $0.03 / 1k input tokens, $0.06 / 1k output tokens.
  The 32k version costs twice as much.
- GPT 4 turbo (introduced in November 2023) cost 1/3 of GPT 4.

2.4. Build your first chatbot app

In this chapter you will learn how to build your very first chatbot `chat_app.py`. It will look like this:



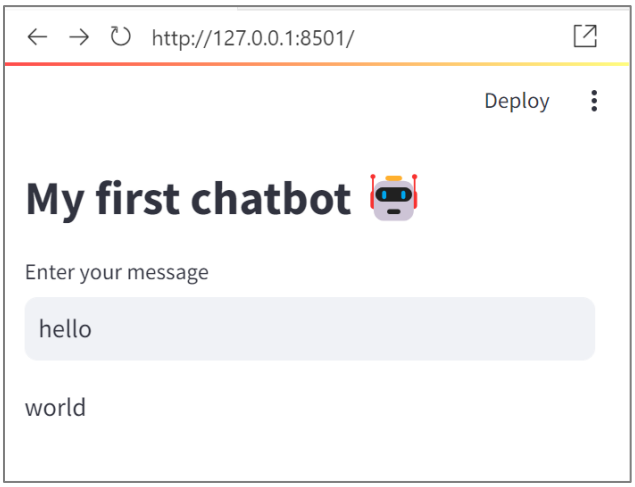
We will breakdown the development into the following steps:

1. Define the graphical components of the app (**text_input**, **button**, **selectbox**)
2. Display the chat conversation in the main area (**chat_message**)
3. Implement various functions (**new_chat**, **save_chat**, **select_chat**, **dumb_chat**) for handling chat functionalities, and save the chat history as json file
4. Handle session states
5. Stream the conversation (**chat_stream**)

[11] https://platform.openai.com/docs/api-reference/chat/create
[12] https://openai.com/pricing

2.4.1. Create a first chat graphical user interface

This part is going to focus more on the Streamlit app building framework.



This is the code for the first chat GUI 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-3.5-turbo',
                                                messages=m)
    response = completion.choices[0].message.content
    st.write(response)
```

The basic elements that are useful for any basic app are the **text_input**[13] and the **write**[14] method.

We can gradually increase the complexity of the app, by adding a **selectbox**[15] to choose the model:

```
models_name = ['gpt-3.5-turbo', 'gpt-4']
selected_model = st.sidebar.selectbox('Select OpenAI model', models_name)
```

Notice that Streamlit provides you with a way to store your OpenAI key as secrets[16].

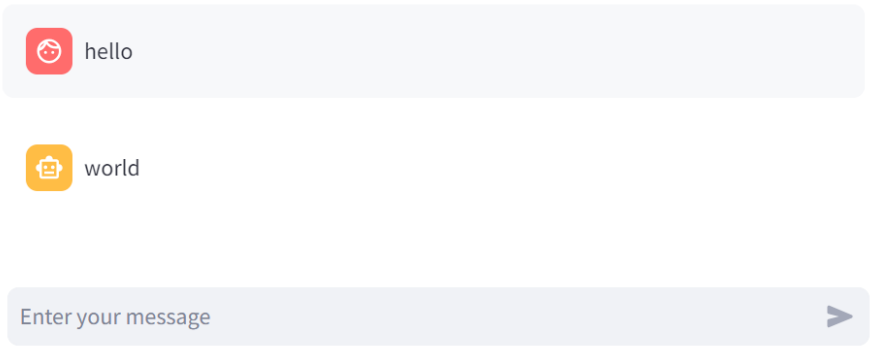
[13] https://docs.streamlit.io/library/api-reference/widgets/st.text_input
[14] https://docs.streamlit.io/library/api-reference/write-magic/st.write
[15] https://docs.streamlit.io/library/api-reference/widgets/st.selectbox
[16] https://docs.streamlit.io/library/advanced-features/secrets-management

2.4.2. Chat elements

Two new methods have been added in 2023 to facilitate the development of chat-based apps: **chat_message**[17] lets you insert a chat message container into the app so you can display messages from the user or the app. **chat_input**[18] lets you display a chat input widget 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.



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

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

[17] https://docs.streamlit.io/library/api-reference/chat/st.chat_message
[18] https://docs.streamlit.io/library/api-reference/chat/st.chat_input

2.4.3. Chat functions

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)
```

In order to start building a chat that has more than one question and one answer (not much of a chat), we will need to save the conversation (and load it back).

First let's start by loading an existing conversation, with a function called **load_chat**
(we can see the conversation dictionary in the sidebar 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)

# Display the response in the Streamlit app
for line in convo:
    st.chat_message(line['role']).write(line['content'])
```

Second 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** (here with a more complex version of the dummy chat, with subsequent display of the messages in the sidebar after each modification):

```
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)
```

2.4.4. Building the conversation

In the previous chapter, we've seen how to save and load a conversation. But we 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 discover the execution model of Streamlit, and how to memorize the elements from the app.

We 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 chapter, we'll see how to make use of Streamlit session states[19] in order to store the conversation.

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
  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})
```

2.4.5. Stream the response

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

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-3.5-turbo'):
  # 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`

3. Chaining and Summarization

In this chapter, you will learn how to chain calls to a Large Language Model (LLM) and use it to summarize texts, such as articles, books, or transcripts. You will implement those methods with the ChatGPT API to build artificially intelligent applications.

3.1. 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.

This gave birth to a new science (or art?) and a buzzword: **prompt engineering**. Essentially, Prompt engineering[1] is the process of creating and refining prompts to get specific responses from AI models. Prompts are questions or instructions that are structured in natural language to describe a task for an AI to perform. I doubt that "Prompt Engineer" will become a real job (but what is a "real" job really…?)

The benefits of chaining are the following:

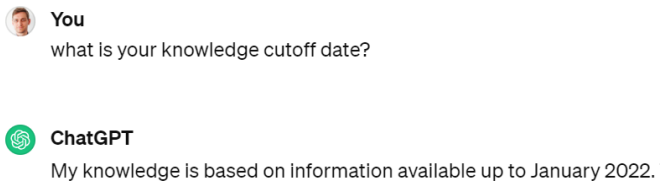
- **Enhanced contextual understanding**: By chaining calls, the LLM can maintain context over a longer interaction, leading to more coherent and relevant responses.
- **Complex task handling**: Chaining allows for the breakdown of complex tasks into simpler sub-tasks, which the model can handle more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

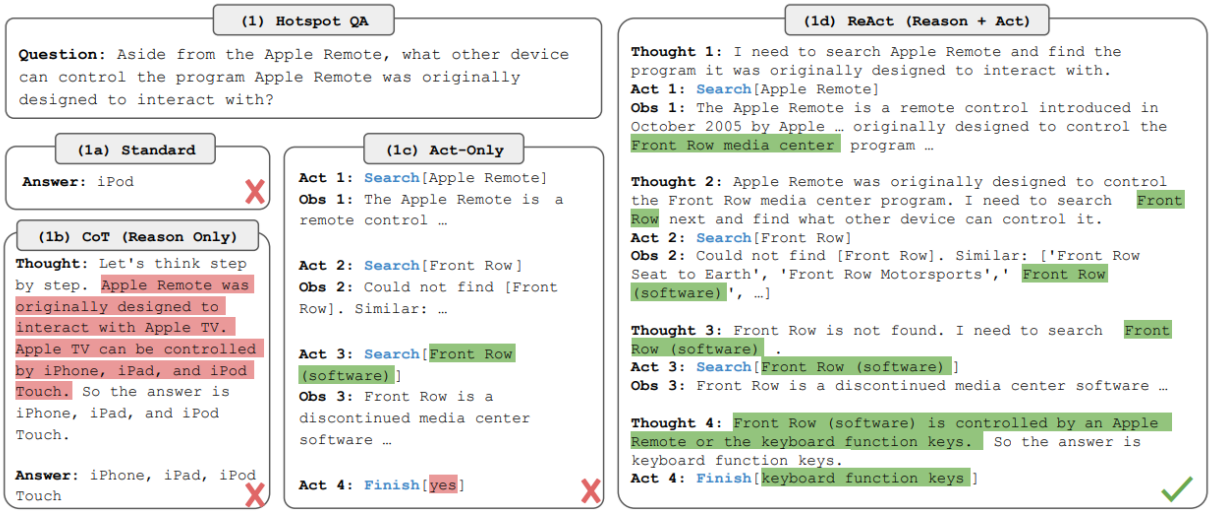
- **Access to Data and Tools**: LLMs do not have access to your data, only the ones they have seen during the training phase. By default they do not have access to search over the internet. As such, it would not make sense to compare ChatGPT with Google search, even if their use cases overlap significantly.

- **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.



[1] https://www.datacamp.com/blog/what-is-prompt-engineering-the-future-of-ai-communication

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**[2].



Instead of directly answering a question, this approach breaks down the process into Reasoning + Acting. It does so by implementing a method called **Chain-of-Thoughts**[3].

A number of frameworks emerged to provide developers with the ability to integrate LLMs into their applications, by implementing those concepts. I'll cover the few main ones in this blog:

- Semantic Kernel
- LangChain
- LlamaIndex
- Haystack

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:

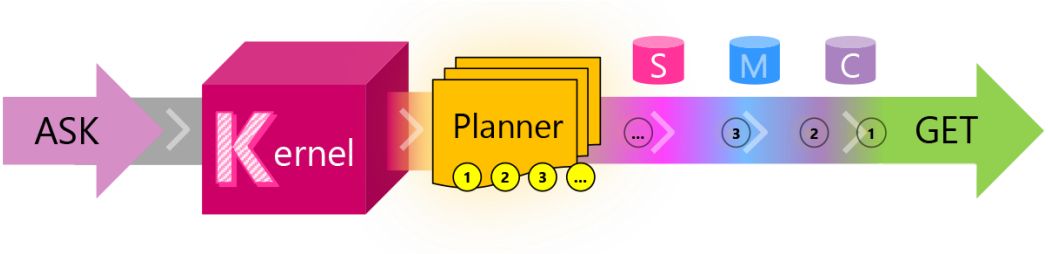
- First, we will look at the basics of chaining to go around the limitations of the LLM context
- Second, we will look at methods such as vector search to extend the LLM context
- Third, we will look at properties that appear when we extend the reasoning of LLMs

3.1.1. 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[4] is a lightweight Software Development Kit (SDK) developed by Microsoft that enables to mix conventional programming languages and Large Language Models with

[2] ReAct paper: https://arxiv.org/abs/2210.03629
[3] Chain-of-Thought: https://blog.research.google/2022/05/language-models-perform-reasoning-via.html
[4] Semantic Kernel: https://learn.microsoft.com/semantic-kernel/

templating, chaining, and planning capabilities. Here is a diagram that explains the workflow triggered by the user asking a question:



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 ...

3.1.2. 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:

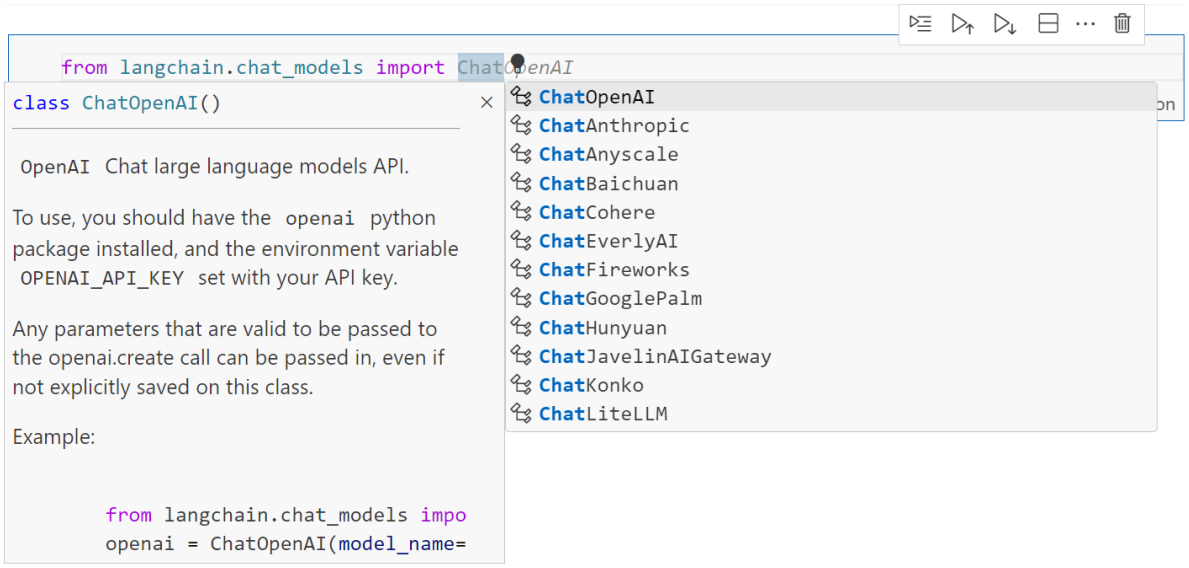
- Extend the length of the context (limited to 4096 tokens)
- Augment generation by retrieving elements not in the context (Retrieval Augmented Generation)
- Add tools such as loading documents, searching the internet or doing basic math

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:



But the use of chat models (like gpt-3.5-turbo) were preferred to those base models like davinci[5]. They've been significantly optimized, and the price is 10 times cheaper[6], so it is best to standardize your code on those 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[7].

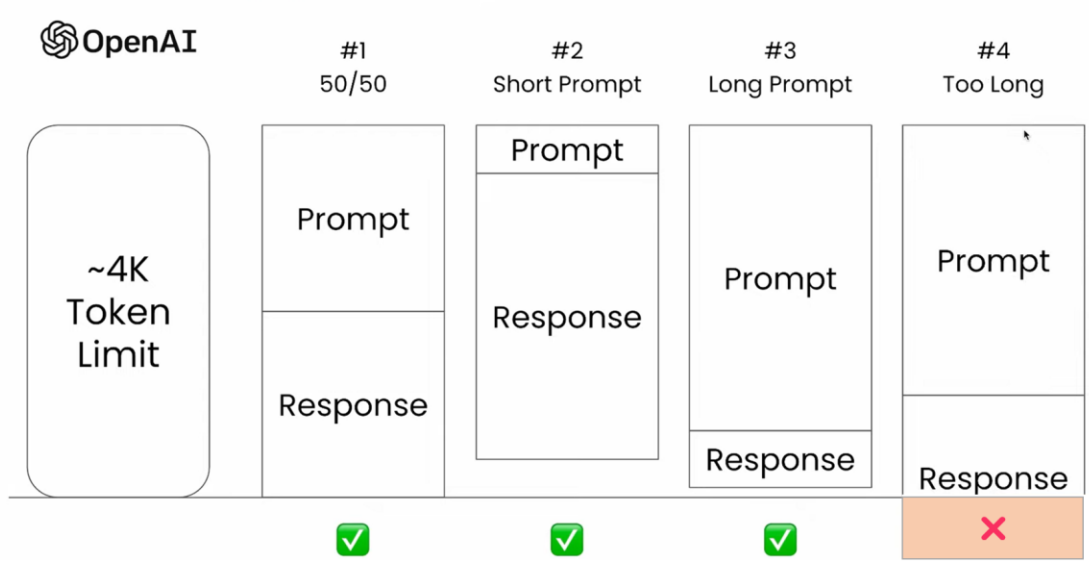
[5] GPT 3 legacy models: https://platform.openai.com/docs/models/gpt-3
[6] GPT 3.5 turbo prices: https://openai.com/pricing#gpt-3-5-turbo
[7] Langchain tutorials:
https://youtube.com/playlist?list=PLqZXAkvF1bPNQER9mLmDbntNfSpzdDIU5
https://github.com/gkamradt/langchain-tutorials

3.2. Summarization

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.

3.2.1. 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.



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). OpenAI provides a tokenizer[8] to help you compute the number of tokens of your text. In what follows, we will use the package tiktoken[9] 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[10] (like the first page of the forewords in Letter format 8.5" x 11"). Which means that 4k tokens can fit 6 pages of text. As the previous diagram suggests, this context length is shared between the prompt and the response.

[8] OpenAI tokenizer page: https://platform.openai.com/tokenizer
[9] Tiktoken package: https://github.com/openai/tiktoken
[10] Words per page: https://wordcounter.net/words-per-page

Let's illustrate this with a basic example of a conversation recorded as WebVTT file[11]:

```
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

num_tokens(convo)
```

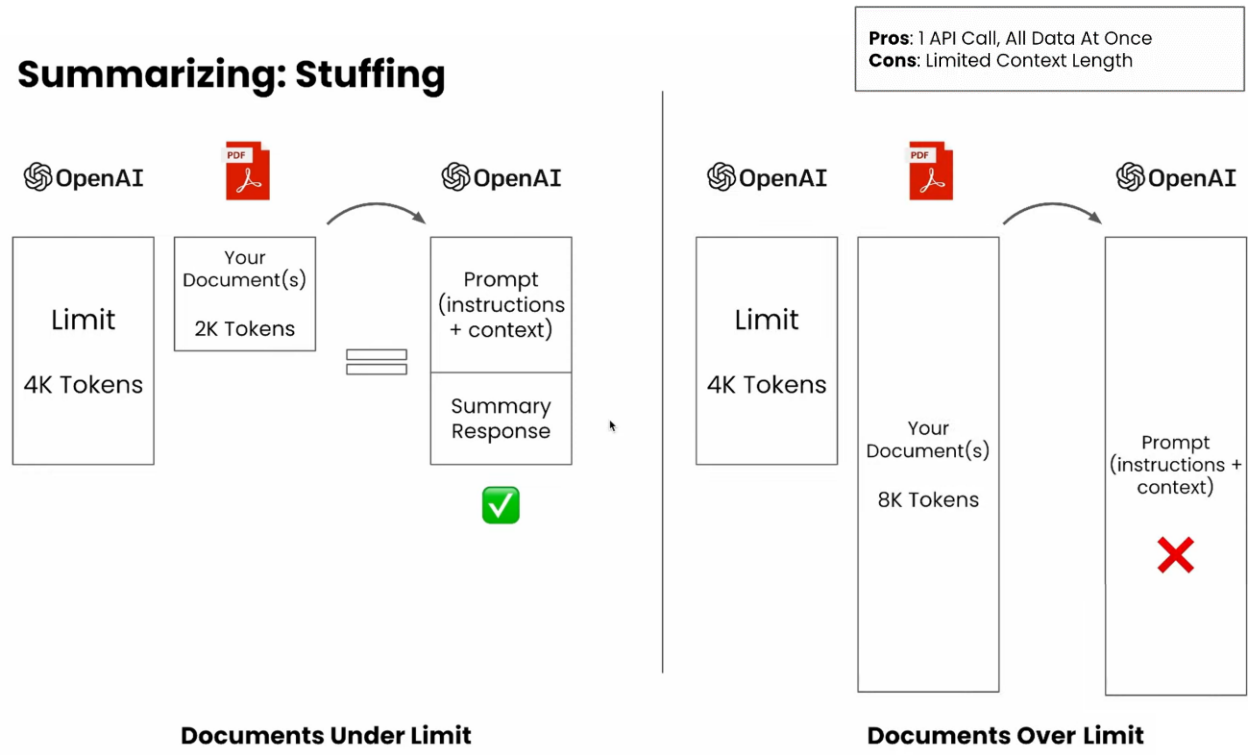
```
150
```

In the next parts, we will go over different chain types to address the limitation of the context length[12].

[11] WebVTT file: https://developer.mozilla.org/en-US/docs/Web/API/WebVTT_API
[12] Workaround OpenAI's Token Limit With Chain Types: https://www.youtube.com/watch?v=f9_BWhCI4Zo

3.2.2. 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.



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)
```

```
'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:

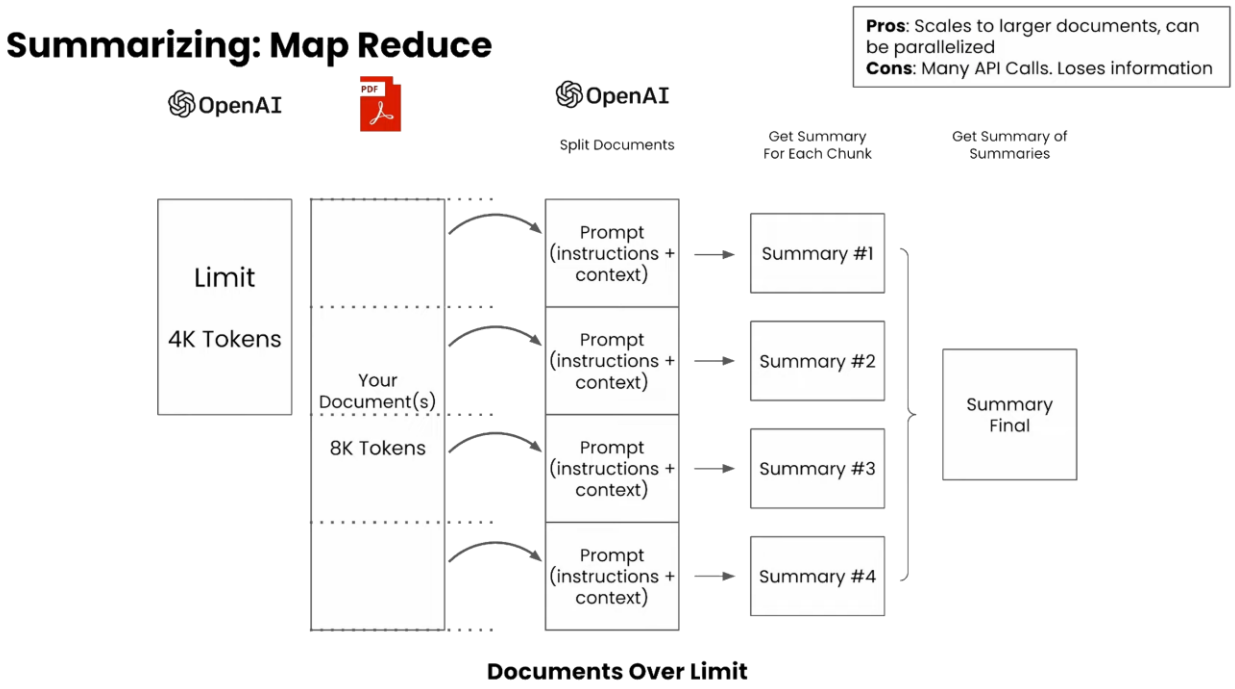
```
page = "https://python.langchain.com/docs/use_cases/summarization"
summarize(page,model = "gpt-3.5-turbo-16k")
```

```
 '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.'
```

3.2.3. 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.



Documents Over Limit

As an example, we will take another source of VTT files: Youtube transcripts[13]. The video we will summarize is the LangChain Cookbook - Beginner Guide To 7 Essential Concepts[14].

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

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

Now comes a new concept that will be important anytime you are breaking down content into chunks: a **text splitter**[15].

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).
- `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.

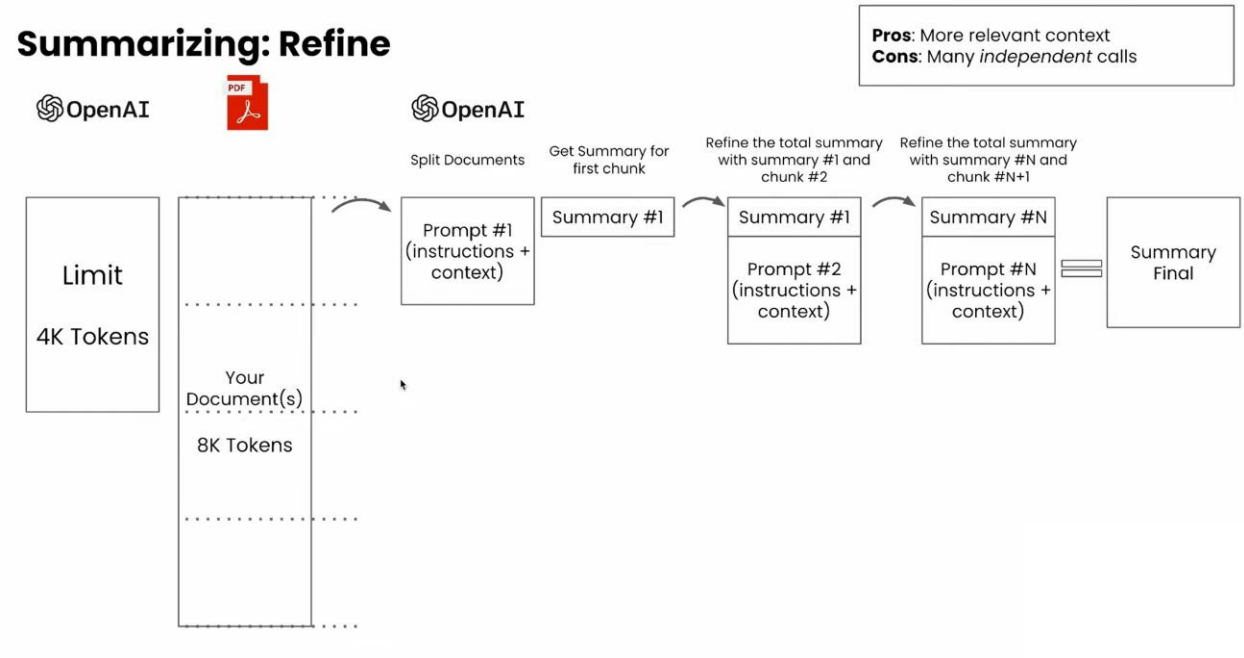
[15] https://python.langchain.com/docs/modules/data_connection/document_transformers/#text-splitters

```
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."
```

3.2.4. 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.



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.'
```

3.2.5. 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[16] (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[17] 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[18] (Nov 2023) and Claude 3 (Mar 2024) propose an impressive 200k tokens context.
- LLaMa[19] from Meta initially came with a 2k tokens context window in February 2023.
- Llama2[20] 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.

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.

---

[16] GPT 4 model: https://platform.openai.com/docs/models/gpt-4
[17] Anthropic Claude 100k context window: https://www.anthropic.com/index/100k-context-windows
[18] Claude 2 & 3: https://www.anthropic.com/news/claude-2-1 - https://www.anthropic.com/news/claude-3-family
[19] LLaMa: https://ai.meta.com/blog/large-language-model-llama-meta-ai/
[20] Llama 2 & 3: https://llama.meta.com/llama2/ - https://llama.meta.com/llama3/

4. Vector search & Question Answering

Another very popular application of the Large Language Models is question answering[1]. Unlike summarization that can be solved with a fixed context, answering questions can be a moving target.

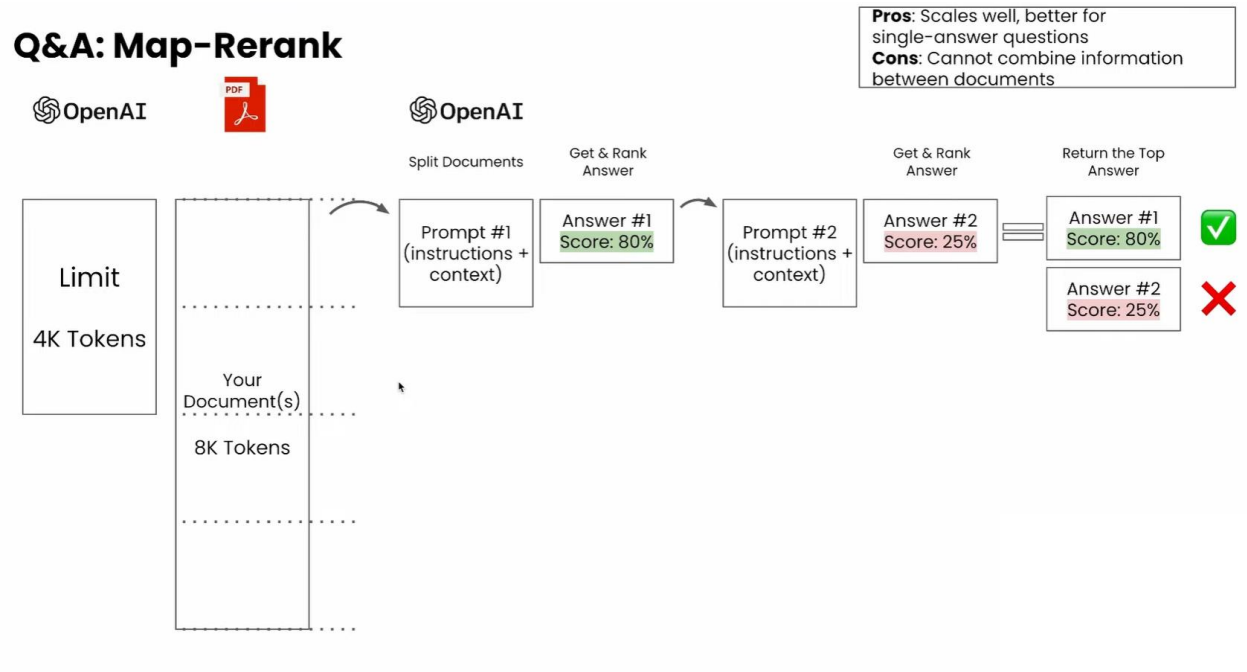
This is why we will investigate methods to extend the context with additional tools such as search, and introduce a special kind of search called similarity search over a vector database of embeddings.

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.

4.1. 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.

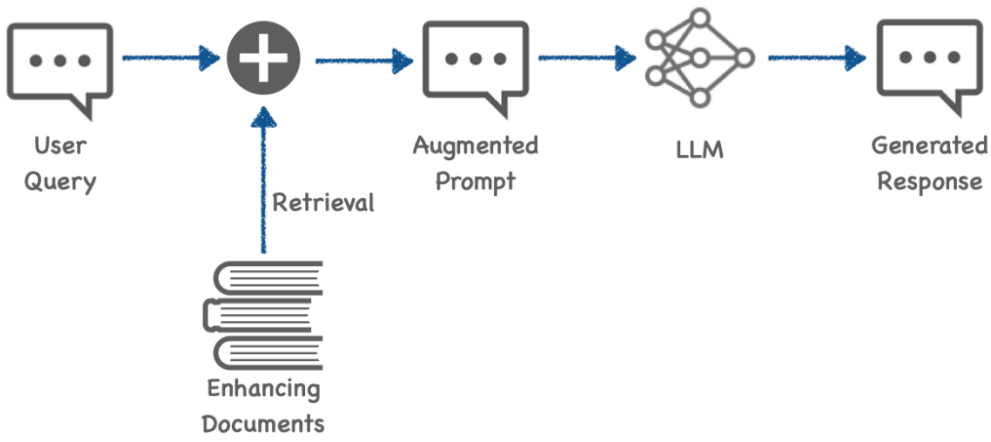
This method, suited for question-answer tasks rather than summarization, involves splitting the document, posing questions to each chunk, and ranking the answers based on confidence scores. It's effective for single-answer questions but does not combine information across chunks.



4.2. 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.

[1] https://python.langchain.com/docs/use_cases/question_answering/



Resources:

- Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
  https://arxiv.org/pdf/2005.11401.pdf
- IBM Blog - What is retrieval-augmented generation?
  https://research.ibm.com/blog/retrieval-augmented-generation-RAG
- Youtube video - What is Retrieval-Augmented Generation (RAG)?
  https://www.youtube.com/watch?v=T-D1OfcDW1M

4.2.1. 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.

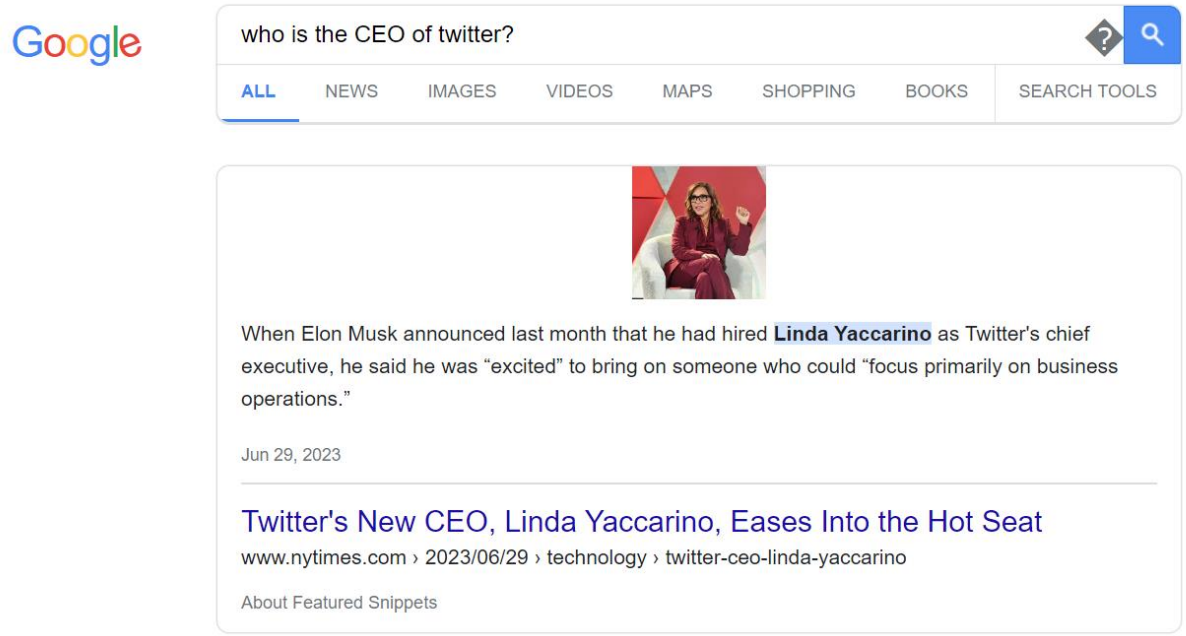
In this chapter, we will implement a very simple search tool that will input information from Google into our ChatGPT request. This way, the chatbot will be able to answer news related questions such as *"Who is the CEO of Twitter?"*.

```
# 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-3.5-turbo",
        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).



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"[2] followed 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()
```

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

```
    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[3], 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

[3] Google Still Uses PageRank. Here's What It Is & How They Use It https://ahrefs.com/blog/google-pagerank/

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[4] (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[5] that impact their effectiveness and user experience.

- **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.

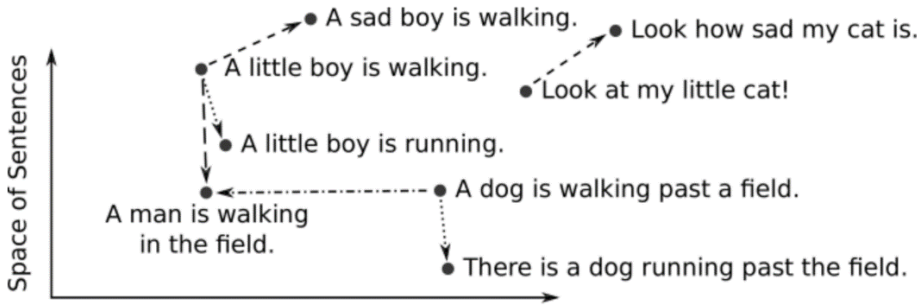
[4] Understanding TF-ID: A Simple Introduction https://monkeylearn.com/blog/what-is-tf-idf/
[5] Vector vs Keyword Search https://www.algolia.com/blog/ai/vector-vs-keyword-search-why-you-should-care/

4.2.2. 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.

- Embeddings: Vector representation of text

Vector embeddings[6] 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.



- Different ways to break down text into numbers

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[7], developed by Google. It uses shallow neural networks to produce word embeddings. It comes in two flavors: Continuous Bag-of-Words (CBOW) and Skip-Gram, each capturing different word relationships.

- Example of embedding service

Let's take a look at an example with the OpenAI's text embeddings[8], on the first paragraph of chapter 1 of A Tale of Two Cities by Charles Dickens[9]

```
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.
```

[6] Vector Embeddings: https://www.pinecone.io/learn/vector-embeddings/ - https://www.pinecone.io/learn/series/nlp/dense-vector-embeddings-nlp/
[7] Word2Vec: https://www.tensorflow.org/text/tutorials/word2vec
[8] OpenAI's text embeddings: https://platform.openai.com/docs/guides/embeddings
[9] A Tale of Two Cities by Charles Dickens: https://www.gutenberg.org/ebooks/98

```
"""
sentences = paragraph.replace("\n"," ").split(", ")
```

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

sentence = sentences[0]

response = client.embeddings.create(
    input=sentence,
    model="text-embedding-3-small"
)

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

```
1536
```

```
[-0.007160207256674765, -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)
```

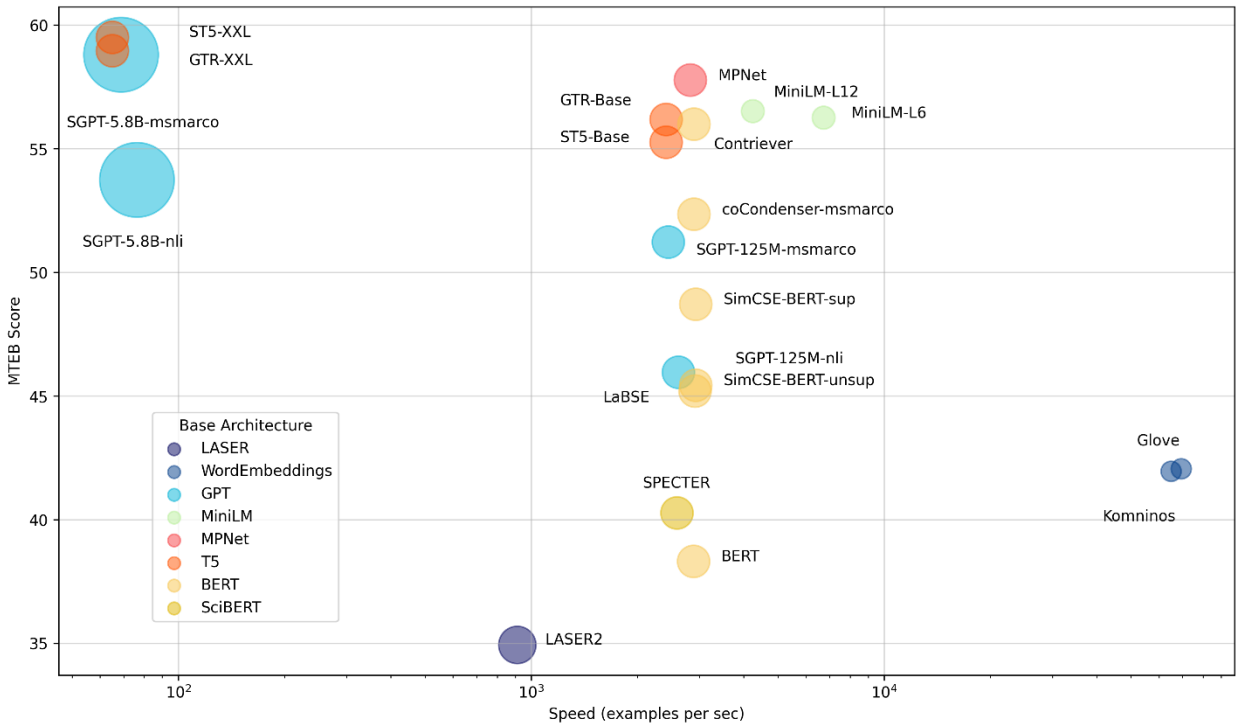
- Trade-offs of different embedding models

Different factors that comes 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)[10] 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):



- Alternative embedding models:

Sentence transformers[11] 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
```

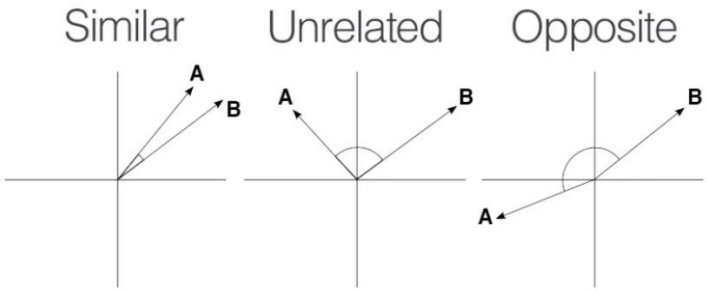
```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2') # or 'all-mpnet-base-v2'

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

```
(384,)
```

[10] Massive Text Embedding Benchmark: https://huggingface.co/blog/mteb
[11] Sentence Transformers: https://sbert.net/

- Calculate semantic similarity[12]

There are different ways to calculate the distance in a high dimensional vector space. One approach is to use trigonometry, with cosine similarity[13] described with the following 3 cases in 2 dimensions:



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

We can simply use provides the `numpy.dot`[14] 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[15].

4.2.3. LlamaIndex: building an index

One of the early frameworks that competed with LangChain nicely to enable question answering was LlamaIndex[16] (initially known as GPTindex). It stood out by the simplicity of its implementation.

[12] Sentence Similarity With Sentence-Transformers in Python https://www.youtube.com/watch?v=Ey81KfQ3PQU
[13] Cosine similarity: https://en.wikipedia.org/wiki/Cosine_similarity
[14] Numpy dot function: https://numpy.org/doc/stable/reference/generated/numpy.dot.html
[15] Vector Databases simply explained! (Embeddings & Indexes) https://www.youtube.com/watch?v=dN0lsF2cvm4
[16] Llama-index: https://docs.llamaindex.ai/en/stable/

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[17]:

- **Step 1: Document loader (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[18]:

```
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()
        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
```

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

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.

4.2.4. 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[19]. 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
```

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

```
    # 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
from langchain_community.document_loaders import PyPDFLoader
from langchain.vectorstores import Chroma
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
chat = ChatOpenAI(model_name='gpt-3.5-turbo')
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)[20], which is known to be insanely performant:

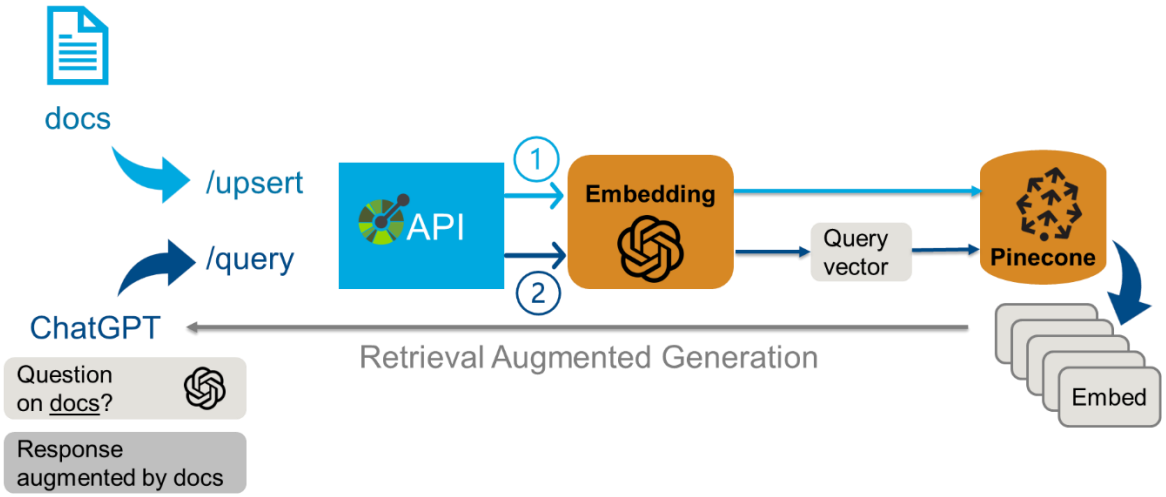
```
pip install faiss-cpu
```

```
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
```

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

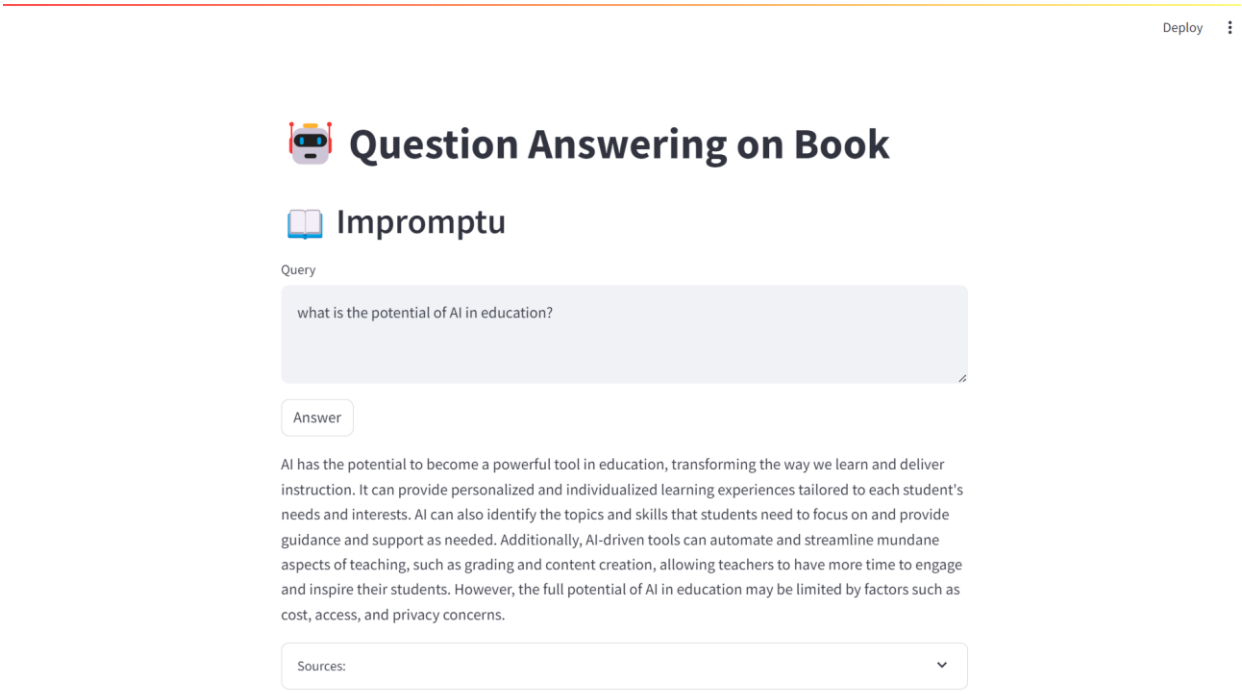
Finally, after trying several vector databases, you can build a production system with Pinecone like this:



As you will see in a future chapter, this application[21] 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.

4.3. 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

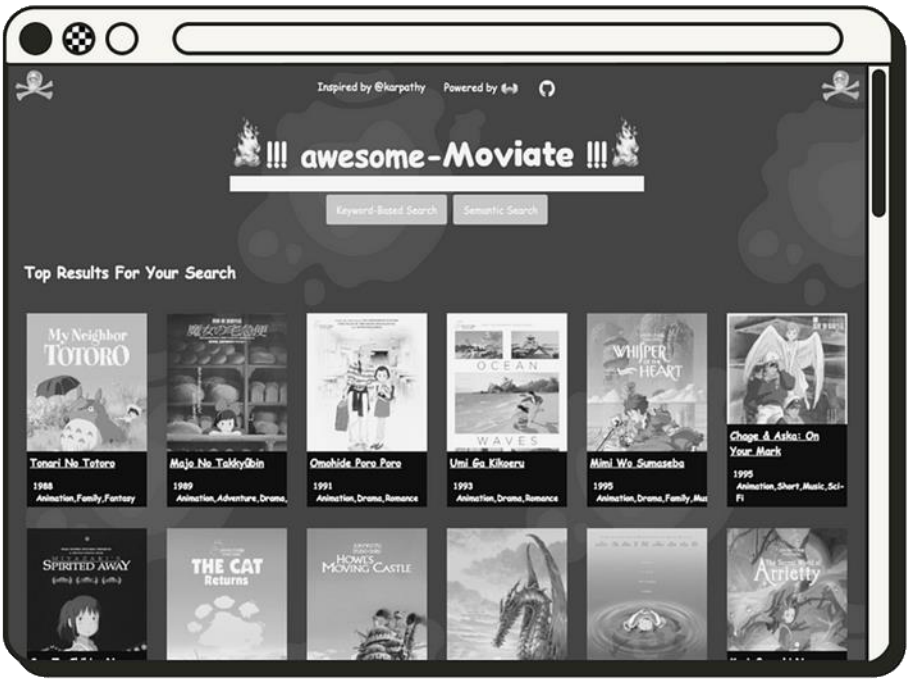


[21] https://github.com/pinecone-io/examples/blob/master/learn/generation/openai/chatgpt/plugins/langchain-docs-plugin.ipynb - https://www.youtube.com/watch?v=hpePPqKxNq8

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.

4.4. Application: Movie search and recommendation engine

In April 2023, Andrej Karpathy[22], 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



Even though Karpathy was too ashamed of the design to share the code, Leonie – developer advocate at Weaviate reimplemented it[23], using a vector database instead of simply storing embeddings as numpy arrays. This is a perfect additional exercise to make sure that you master the notions in this chapter.

[22] https://karpathy.ai/

[23] 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

5. Agents & Tools

Large Language Models are one (big) step in the direction of what is called Artificial General Intelligence (AGI)[1]. 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**.

5.1. 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*[2] from Haystack. Here is the example of a prompt that illustrates how agents operate:

You

You are an agent that has access to tools to perform action.
For every prompt you receive, you will decide which available tool to use from the following list:

- search : Useful for when you need to answer questions about current events.
- python: Useful for when performing computing and trying to solve mathematical problems

Here's some examples of prompts you'll get and the response you should give:

USER: What movie won best picture in 2023?
BOT: search

USER: What is the 10th element of the Fibonacci suite
BOT: python

Question:
What is the square root of 42?

ChatGPT

python

You

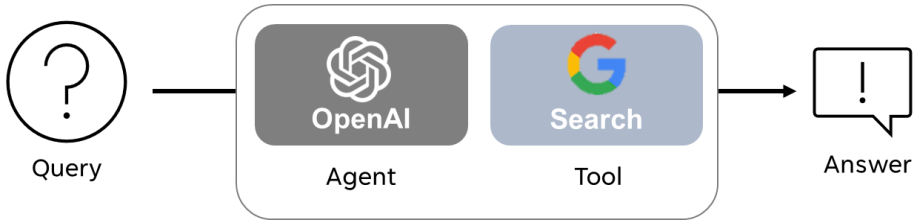
who won the oscar in 2024

ChatGPT

search

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

Here is a simple representation of what is happening here:



5.1.1. 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.

5.1.2. LangChain agents

Here is the same example using LangChain with a search API[3]. Let's turn on the `verbose=True` argument to observe what is happening under the hood. The syntax will evolve, so check the doc.[4]

```
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.
```

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

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'

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.

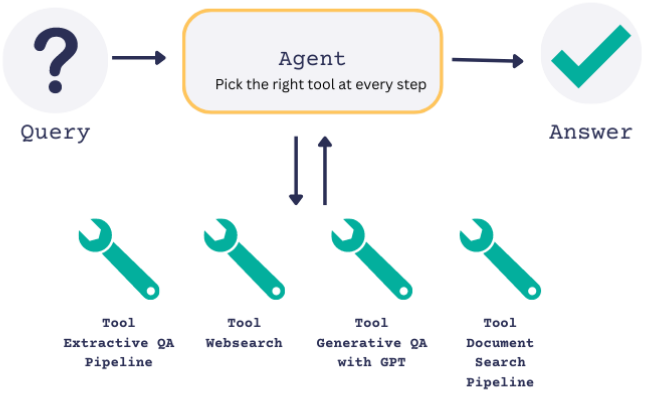
Action:
```
{
  "action": "time",
  "action_input": ""
}
```
Observation: ***2024-05-04***

```
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.

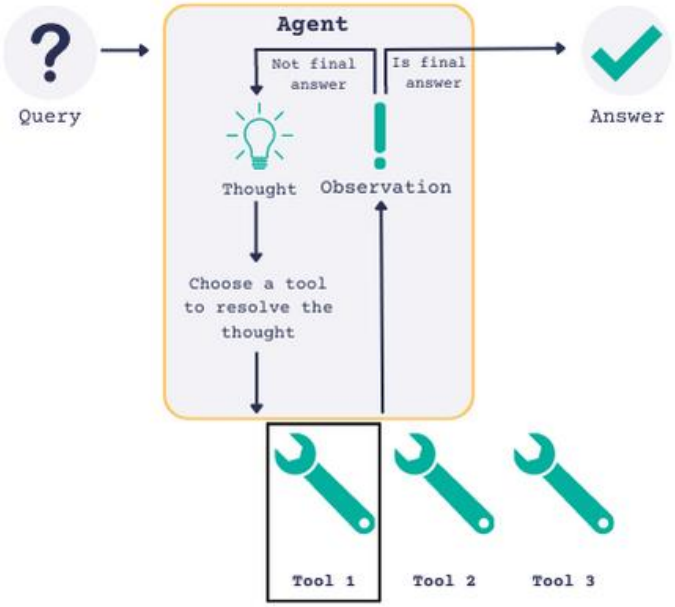
5.1.3. Haystack agents

I'm presenting the Haystack framework because I really like their tutorials and learning materials[5]. Haystack[6] 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**.



[5] https://haystack.deepset.ai/blog/introducing-haystack-agents
[6] https://haystack.deepset.ai/

What is the difference between an Agent and a Pipeline in Haystack? A Pipeline is a one-pass system 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.



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[7] (to not mess up your local dev environment). It uses a HuggingFace dataset[8] 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 really need a framework to build LLM-based applications anymore. You just need your favorite scripting language and a good understanding of OpenAI services.

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

5.2. OpenAI Functions

In July 2023, OpenAI announced Function calling capabilities[9]. 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.

5.2.1. Setup functions list

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

```python
# 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, hence when to use it)
- Parameters (including with ones are required)

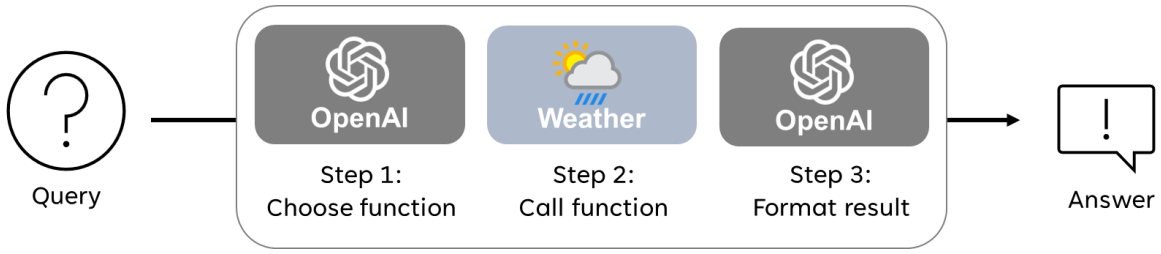
I am saving those attributes into a `get_current_weather.json` file in a folder called functions:

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]}
        },
        "required": ["location"]
    }
}
```

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

5.2.2. Steps to function calling

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




```
# 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-3.5-turbo-0613",
    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
```

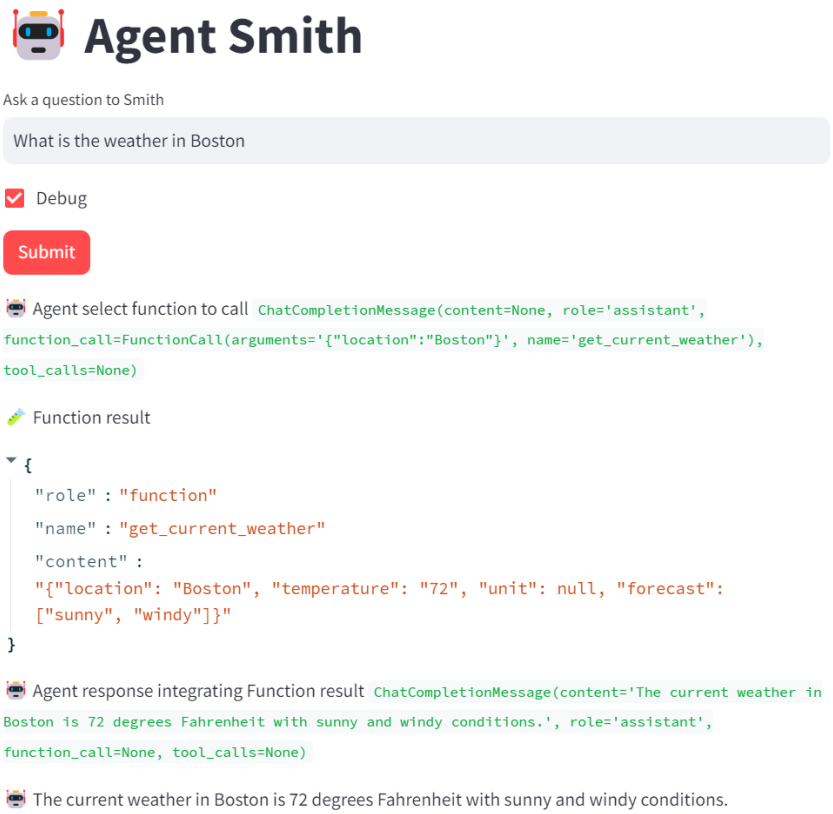
```
'{"location": "Boston, MA", "temperature": "72", "unit": null, "forecast":
["sunny", "windy"]}'
```

```
# 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-3.5-turbo-0613",
    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.'
```

5.2.3. Create an app to speak to the weather agent

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



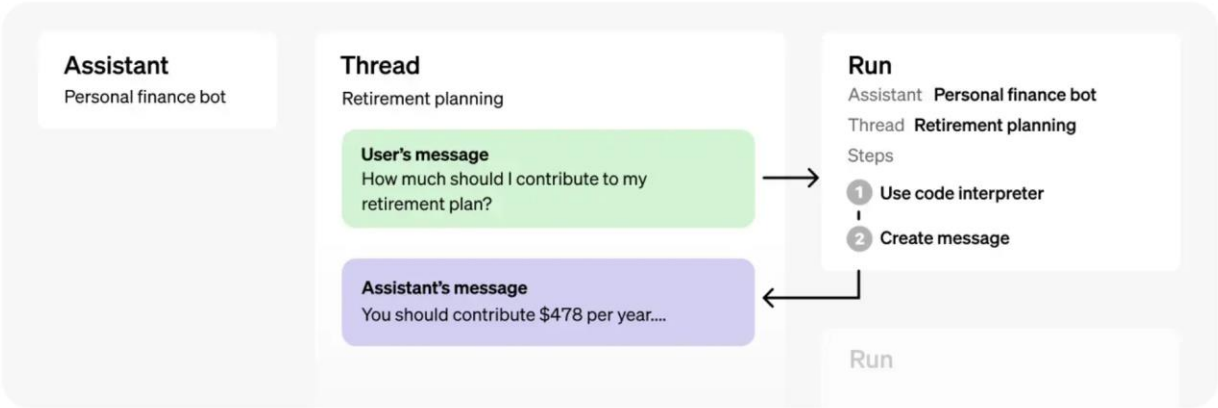
5.3. OpenAI Assistants

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:



Let's break this down in 4 steps with a simple Math Tutor assistant[10]:

- **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])
```

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

```
id: asst_RjI8kOYWuIa8t6DXPlembrrW
Name: Math Tutor
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).

```python
# 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 \(x = 1\).
```

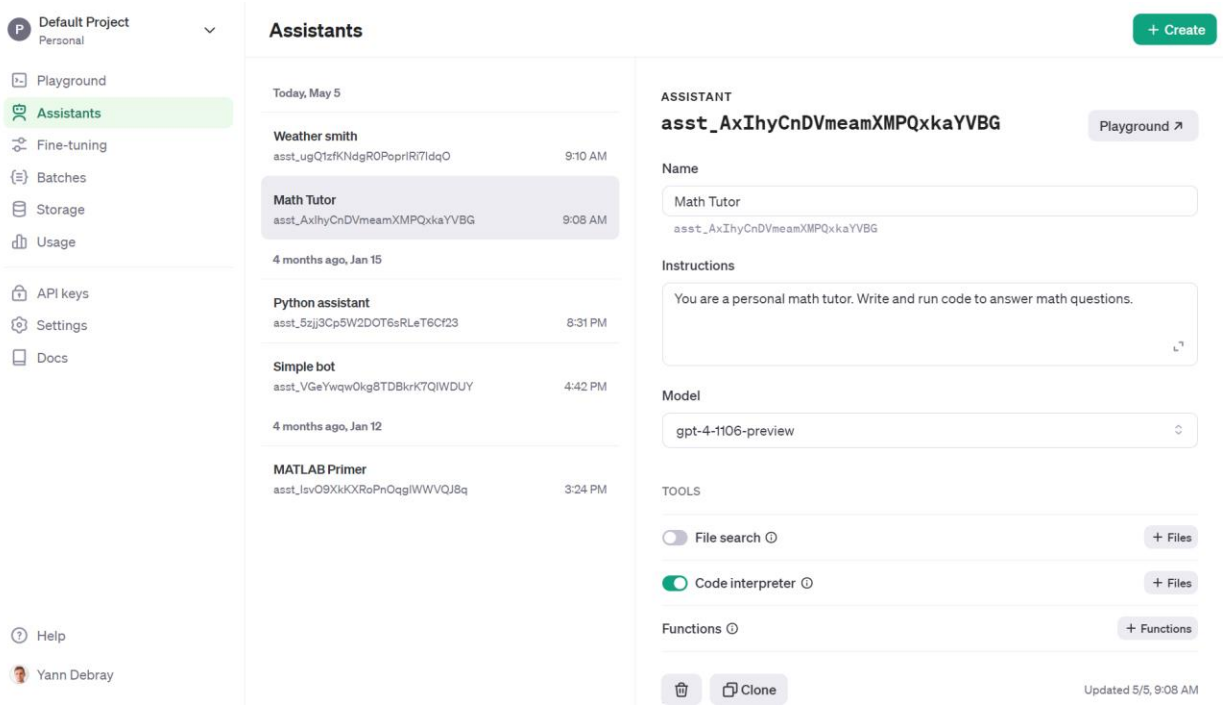
5.3.1. Assistant API

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

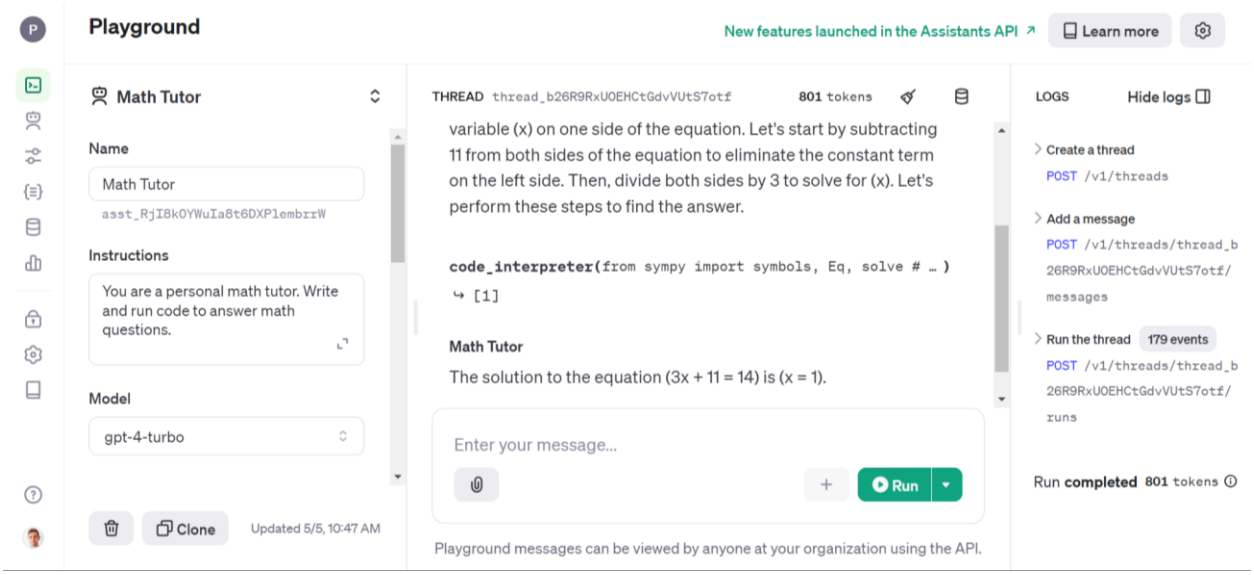
Assistants can be characterized by the tools they have at their disposal. We will look at 3 kinds:

- Code interpreter
- File search
- Function

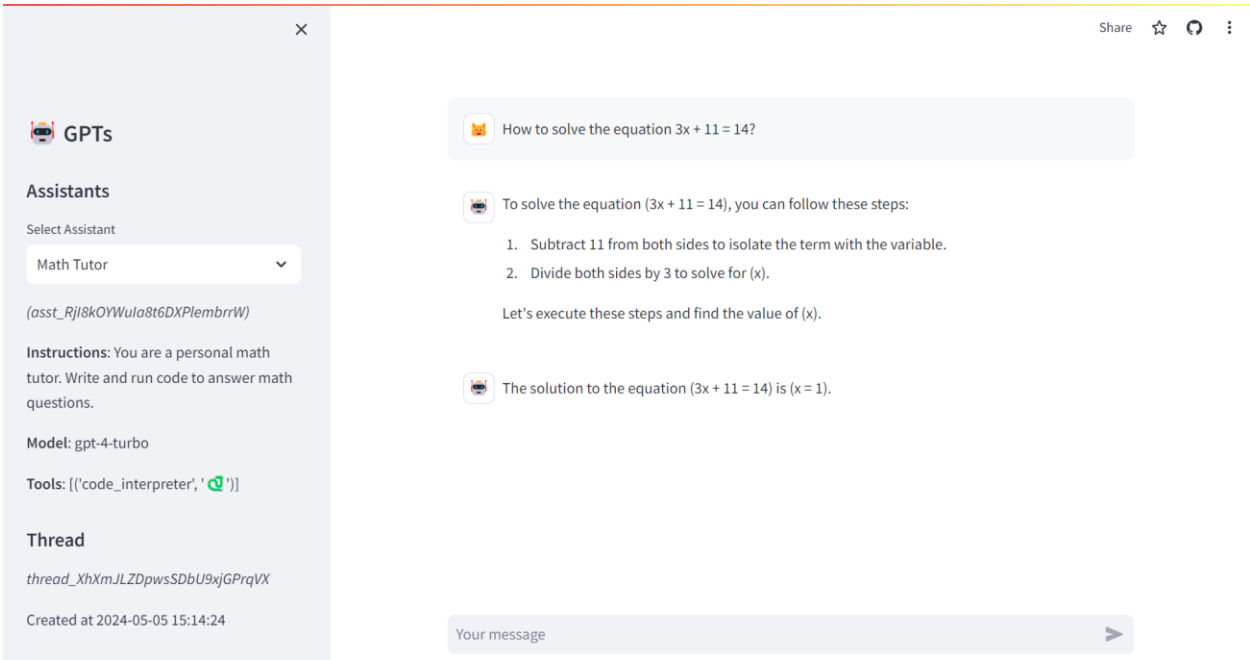
You can access and create assistants in the platform dashboard: https://platform.openai.com/assistants



And you can test the assistants in the playground: https://platform.openai.com/playground/assistants



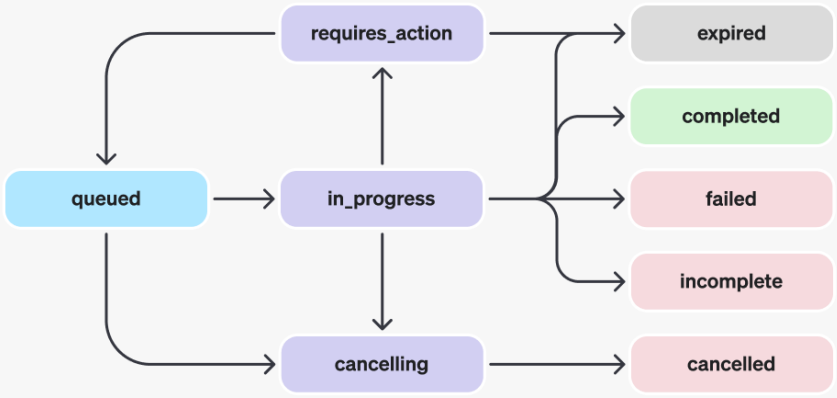
Here is an `assistant_app` that will provide an interface to the assistants:



5.3.2. 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[11]**:** runs will go through several steps and update their status



You may also want to list the Run Steps[12] 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
from sympy import symbols, Eq, solve

x = symbols('x')
equation = Eq(3*x + 11, 14)
solution = solve(equation, x)
```

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

```
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.
------
```

5.3.3. 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[13]. Plugins have since been retired[14] in favor of the GPT store. And OpenAI introduced the assistant API as a way to create your own code interpreter.

How is the Code Interpreter API priced?

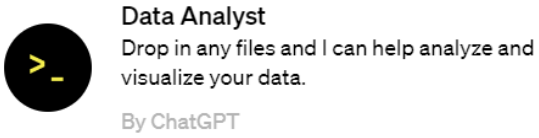
Code Interpreter is priced at $0.03 / session. If your assistant calls Code Interpreter simultaneously in two *different threads*, this would create two Code Interpreter sessions (2 * $0.03). Each session is active by default for one hour, which means that you would only pay this fee once if your user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

Passing files to Code Interpreter: meet the Data Analyst

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[15] endpoint and then pass the File ID as part of the Message creation request:

The Data Analyst GPT is accessible only with a ChatGPT Plus subscription:



But you can create your own with the Code Interpreter tool of the assistant API. (Note: Add to TODO list)

Example with the Titanic dataset[16]:

```
from openai import OpenAI
client = OpenAI()
analyst = client.beta.assistants.create(
  name="Data Analyst",
```

[13] https://openai.com/blog/chatgpt-plugins#code-interpreter
[14] https://help.openai.com/en/articles/8988022-winding-down-the-chatgpt-plugins-beta
[15] https://platform.openai.com/docs/api-reference/files/create
[16] https://www.kaggle.com/c/titanic/data

```
  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,
)
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
```

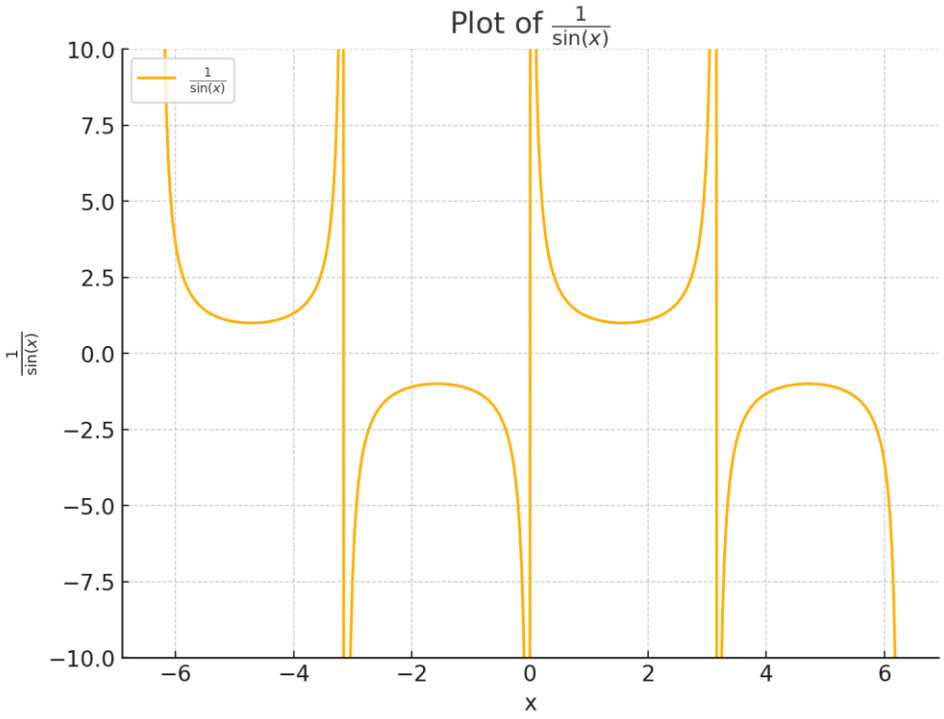
Reading images and files generated by Code Interpreter

```
# 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,assistant_id=a
nalyst_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'}
```

```python
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)
```



5.3.4. File Search

File search (previously called Retrieval in the API V1) implements the notion of RAG (Retrieval Augmented Generation) as a service[17].

What is the File Search tool?

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.

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

What are the restrictions for File upload?

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.

How is the File Search API priced?

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.

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",
```

[17] https://platform.openai.com/docs/assistants/tools/file-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_ulgeT4pn8fzb5wJYebXivJHo'])
```

Vector stores created using message attachements 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.
```

Read more about Vector Store: https://platform.openai.com/docs/assistants/tools/file-search/vector-stores

6. Speech-to-Text and Text-to-Speech

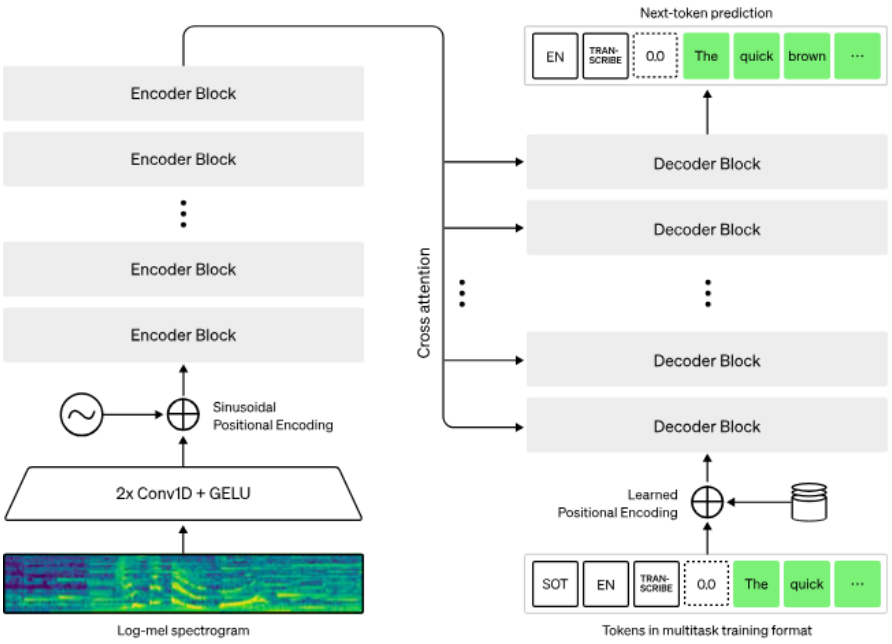
In this chapter, you will learn how to transcribe text from speech (such as Youtube videos) and synthetize speech from text (such as articles).

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[1] the engineering assistant of iron-man, that appeared in the first movie from 2008.

6.1. 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[2], 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.



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

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

A few pure players of Speech AI have specialized in this task, like AssemblyAI[4] and Gladia[5]. But here again, a large chunk of this market is driven by OpenAI, with the release of Whisper[6] 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
```

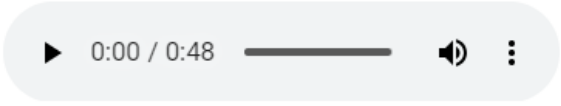
It also requires the command-line tool `ffmpeg`[7] to be installed on your system.

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.

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)
```



[3] https://platform.openai.com/docs/tutorials/meeting-minutes
[4] https://www.assemblyai.com/ - https://www.youtube.com/watch?v=r8KTOBOMm0A
[5] https://www.gladia.io/ - https://techcrunch.com/2023/06/19/gladia-turns-any-audio-into-text-in-near-real-time/
[6] https://openai.com/research/whisper
[7] https://ffmpeg.org/

```
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, ...
```

You can also use Whisper through the transcription API[8] 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. ...'
```

6.2. 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[9]. 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[10] 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",
  input="The quick brown fox jumped over the lazy dog.")
response.stream_to_file(speech_file_path)
```

[8] https://platform.openai.com/docs/guides/speech-to-text
[9] https://elevenlabs.io/
[10] https://platform.openai.com/docs/guides/text-to-speech

```
Audio(speech_file_path)
```

6.3. 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

6.3.1. Parse the Techcrunch RSS feed

RSS[11] 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, newsletters, and website content
- 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"[12], citing that RSS gives more control over content compared to algorithms and trackers from social media sites. At that time, Feedly[13] 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[14] (`pip install feedparser`). I will simply default to the basic xml parsing capabilities in base python

[11] https://en.wikipedia.org/wiki/RSS
[12] https://www.wired.com/story/rss-readers-feedly-inoreader-old-reader/
[13] https://feedly.com/news-reader
[14] https://feedparser.readthedocs.io/en/latest/

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

6.3.2. 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 in the prompt) 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

```python
# 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
```

```python
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

```python
def summarize(text):
    inst = '''Summarize the following article in less than 4000 characters.'''
    completion = openai.chat.completions.create(
        model='gpt-3.5-turbo-1106',
        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)}")
```

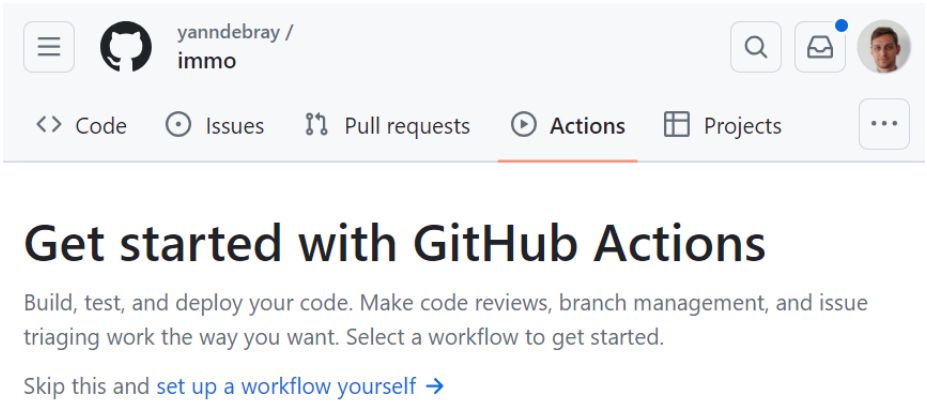
```
Summary: President Joe Biden signed a bill allowing
Characters: 1435
```

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

6.3.3. Schedule a GitHub Action to run daily

GitHub Actions[15] 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.



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

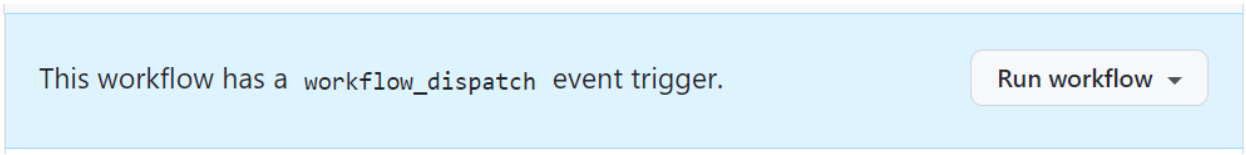
```yaml
name: daily tech podcast
on:
  # Triggers the workflow on a schedule, every day at 00:00 UTC
  schedule:
    - cron: "0 0 * * *"
```

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

```yaml
  # 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]/
env:
  OPENAI_API_KEY: ${{secrets.OPENAI_API_KEY}}
  # ELEVEN_API_KEY: ${{secrets.ELEVEN_API_KEY}}
```

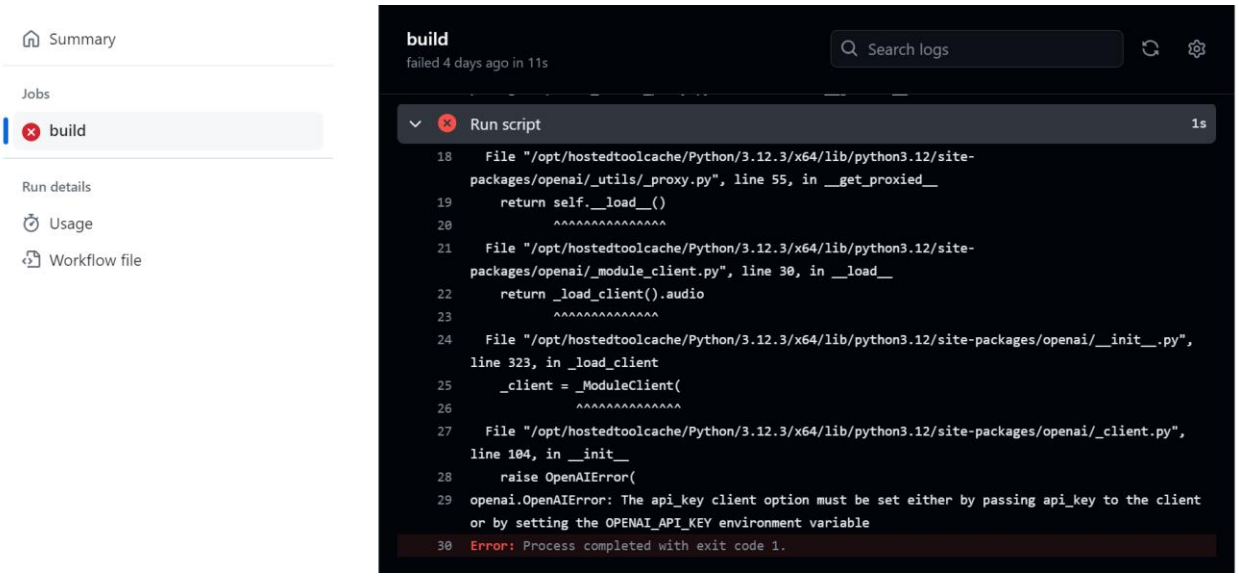
Some explanations of what this is doing:

- Actions use a YAML format[16] to specify the elements of the job to run.
- Actions use the cron syntax[17] to specify the schedule at which to run.
- You can also specify workflow_dispatch to get the following manual trigger in the actions tab:



[16] https://realpython.com/python-yaml/
[17] https://crontab.guru/every-day

- Setup Python and install the necessary dependencies in the runner[18]
- If the run fails, you will get an email. You can analyze the log to see what went south:



- Once the jobs ran according to plan (which happens), you can upload the resulting artifact[19]:

  Point to the file or the entire directory:

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

  `path: path/to/artifact/`

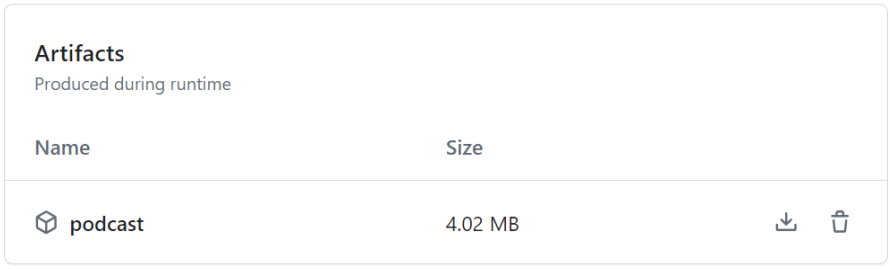
  Insert a wild card in the path, in order to take variations into account:

  `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.

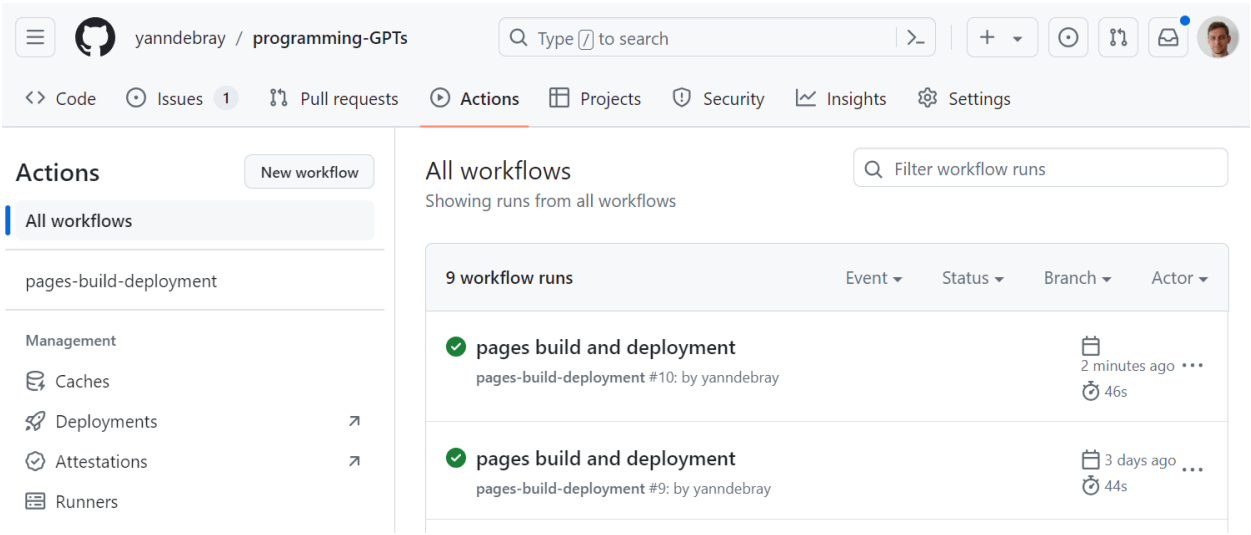
  The artifact will be available in the detail of the run, as a zip file:

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



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[20].



-

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

7. Vision

7.1. 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[1], 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.

```
 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}"}},
         ],
         }
     ],
```

[1] GPT-4 Developer Livestream https://www.youtube.com/watch?v=outcGtbnMuQ

```
    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)
```


```
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.'
```

7.2. Object detection

Object detection[2] 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 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-4V, 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.

7.2.1. Application: detecting cars

Let's take an example from a self-driving dataset from Udacity:

- This project starts with 223GB of open-source Driving Data: https://medium.com/udacity/open-sourcing-223gb-of-mountain-view-driving-data-f6b5593fbfa5

[2] https://www.mathworks.com/discovery/object-detection.html

- The repo has been archived, but you can still access it: https://github.com/udacity/self-driving-car
- Streamlit developed an associated app and hosted organized data on AWS (That's how I found out about this example): https://github.com/streamlit/demo-self-driving/blob/master/streamlit_app.py 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)[3] 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]
```

[3] 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
```

7.2.2. 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/1478019976687231684.jpg"
 image = plt.imread(image_path)

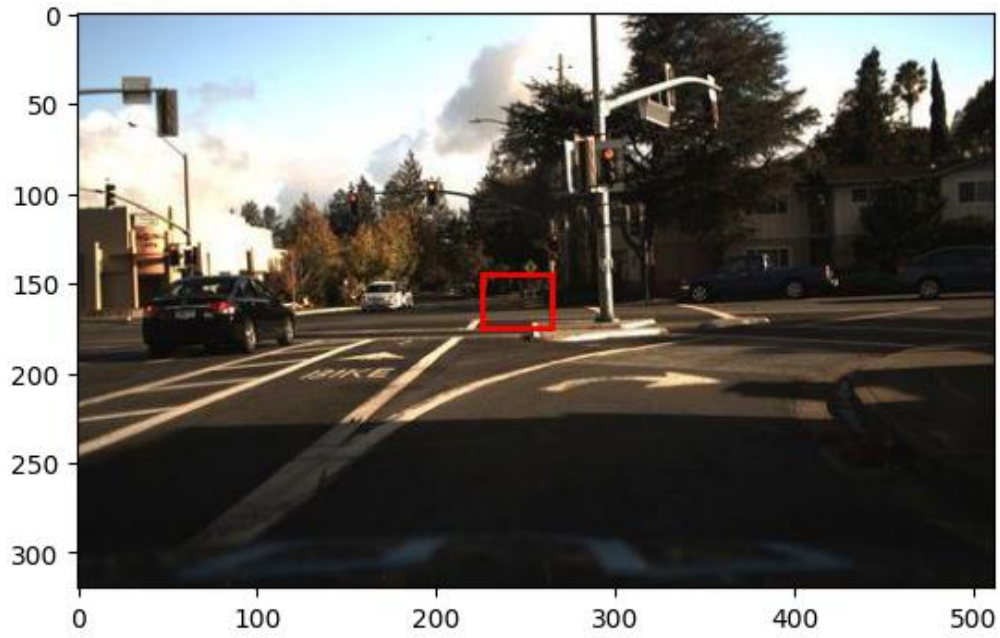
 # Create figure and axes
 fig, ax = plt.subplots()

 # 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()
```



As you can see, this clearly isn't a success yet for object detection and localization.

7.2.3. Traditional Computer Vision

YOLO (You Look Only Once)[4] 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 this repo: https://github.com/pjreddie/darknet

You will have to download the pre-trained weight file here (237 MB): 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()]
```

[4] https://pjreddie.com/darknet/yolo/

```
 # Load image
 image = cv2.imread(image_path)
 height, width, _ = image.shape

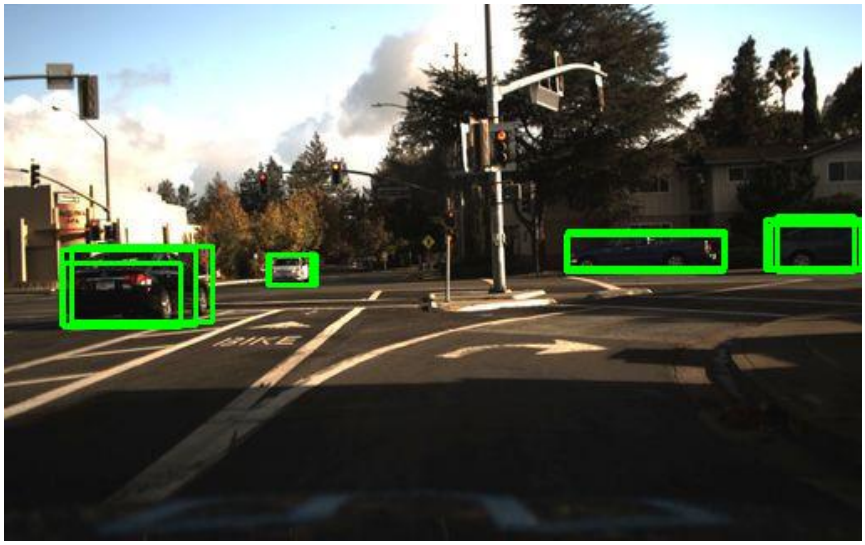
 # Preprocess image
 blob = cv2.dnn.blobFromImage(image, 1/255.0, (416, 416), swapRB=True,
crop=False)

 # 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)
```



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.

7.3. 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-4V 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
```

```
    # get image from clipboard
    from PIL import ImageGrab
    img = ImageGrab.grabclipboard()
    img
✓ 0.0s
```

```
 # 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)
```

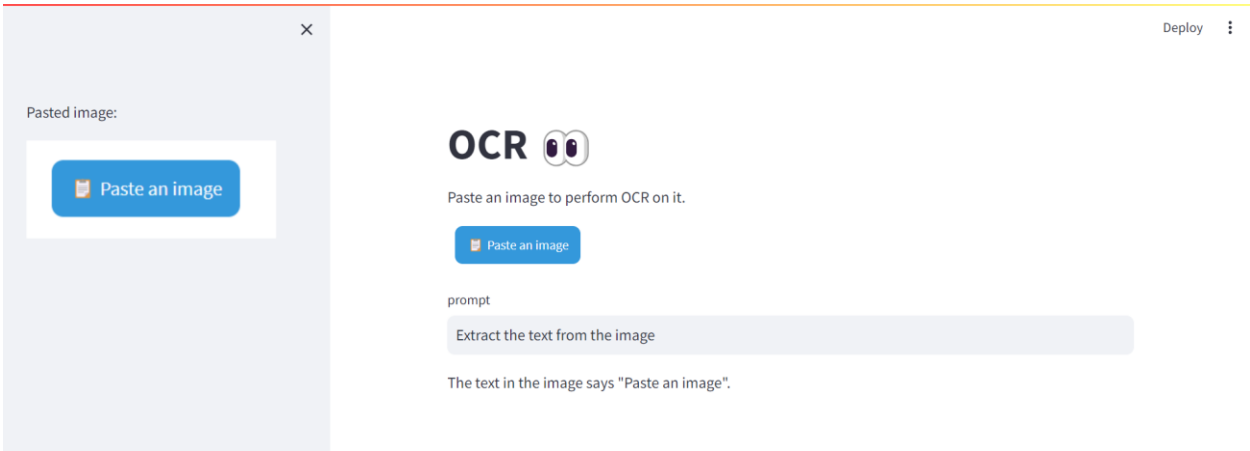
```
# get image from clipboard
from PIL import ImageGrab
img = ImageGrab.grabclipboard()
img
```

Let's implement a simple Streamlit app that will provide us with an OCR assistant, with a paste button[5] getting images from the clipboard:

```
pip install streamlit-paste-button
```

This is what the ocr_assistant.py looks like:

[5] https://github.com/olucaslopes/streamlit-paste-button



7.4. From mock to web UI

One really exciting application of GPT-4V 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}'
    # 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

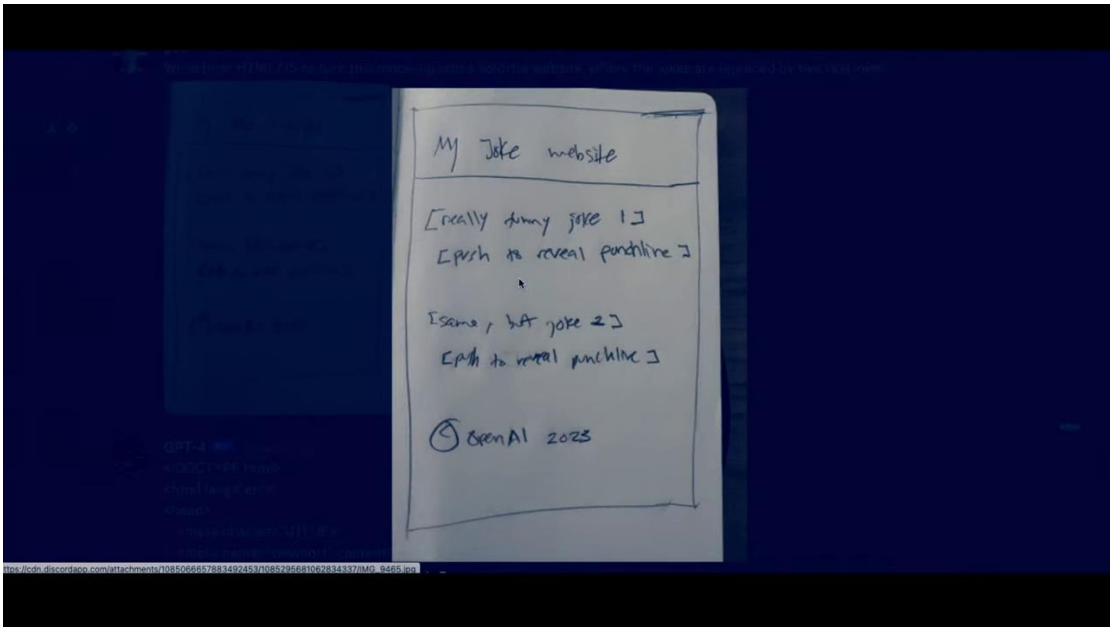
video_id = "outcGtbnMuQ"
video_path = youtube_download(video_id, quality="highest",path="../data/video")
```

```
# 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

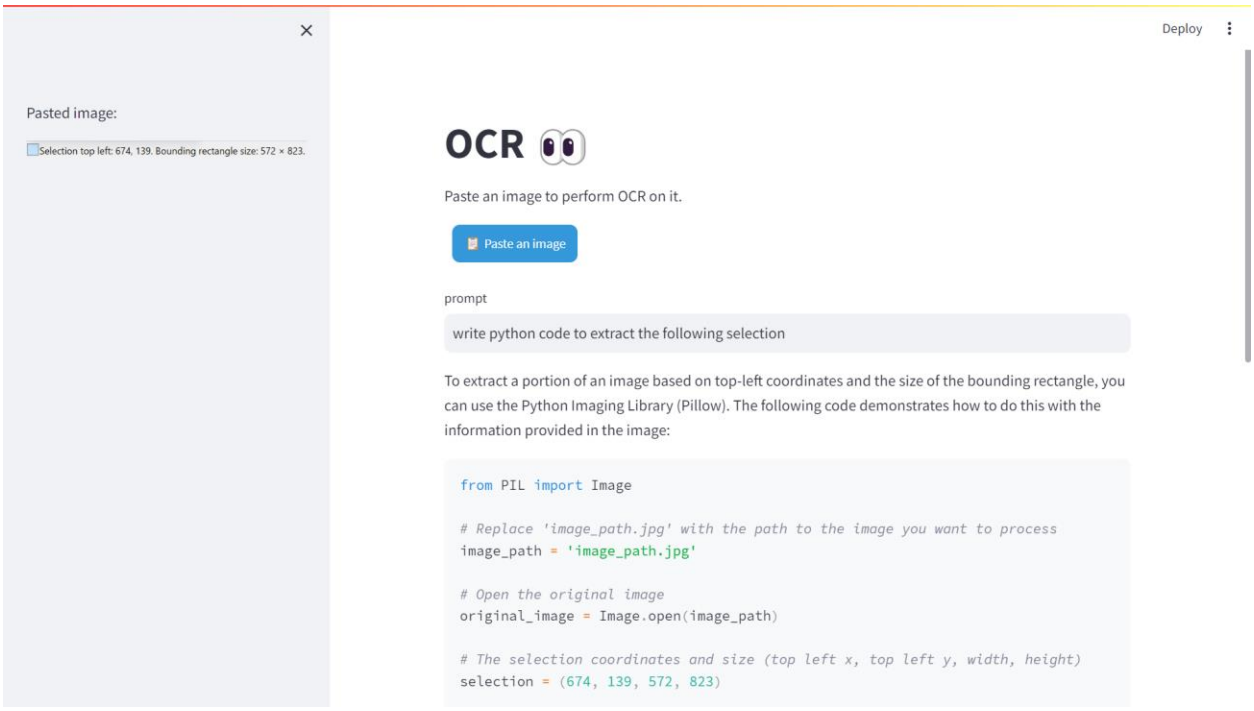
frame_path = extract_video_frame(video_path, time="18:27")
# Display the frame
Image.open(frame_path)
```



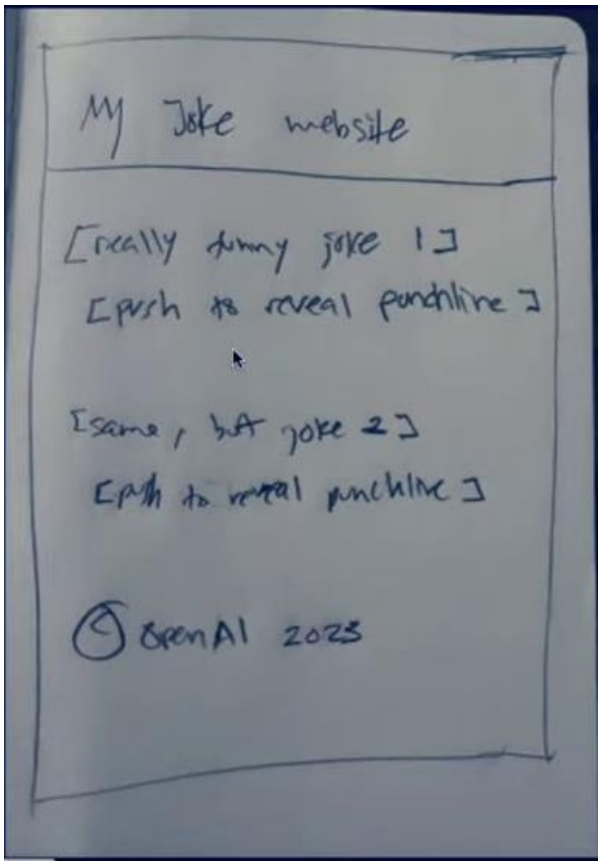
Now we need to crop the image. Let's open up the frame in Paint.net, and grab the area to crop:

Selection top left: 449, 95. Bounding rectangle size: 383 × 552.

We can leverage our OCR assistant to `write the python code to extract the selection`.




```python
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
```

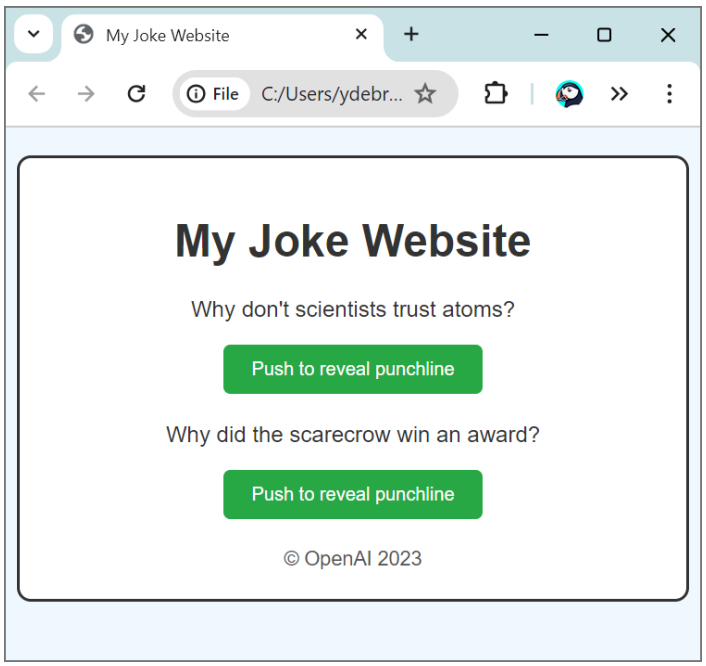


Let's reuse the prompt from the demo:

```
 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



7.5. Video understanding

Let's look at another use case: Processing and narrating a video with GPT's visual capabilities and the TTS API[6]. For this example, I will retrieve a video from Youtube that I created, and add a different voiceover.

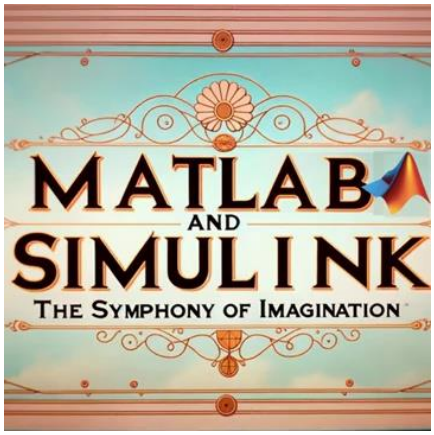
```
from IPython.display import display, Image, Audio
from pytube import YouTube
import cv2, base64, time, openai, os, requests

video_id = "1BaEO836ECY"
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()
```

[6] https://cookbook.openai.com/examples/gpt_with_vision_for_video_understanding

```
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("utf-8"))))
    time.sleep(0.025)
```




```
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."
```

```python
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)

```python
with open("../data/audio/voiceover.mp3", "wb") as f:
    f.write(audio)
```

Let’s use MusicGen[7] 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.

```python
from moviepy.editor import VideoFileClip, AudioFileClip
from moviepy.video.fx.speedx import speedx
```

[7] https://audiocraft.metademolab.com/musicgen.html - https://huggingface.co/facebook/musicgen-large

```
 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
```

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[1] was one of the APIs that could call programmatically a model named DALL-E[2] (already in v2 since august 2022). In parallel other Generative AI models, like the open-source Stable Diffusion[3] or the proprietary Midjourney[4] 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.

To get started you can do this tutorial: https://realpython.com/generate-images-with-dalle-openai-api/

I tried out a few on fun projects like *Creating Viral Videos With AI*[5]. 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
```

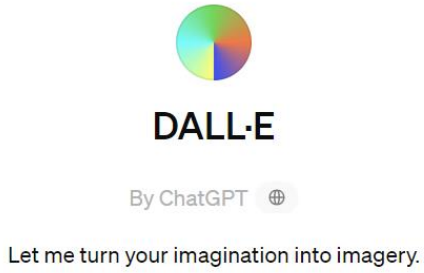
In this chapter we will see how to create a graphical novel on the same topic.

[1] https://platform.openai.com/docs/guides/images
[2] https://openai.com/index/dall-e-2/
[3] https://stability.ai/news/stable-diffusion-public-release
[4] https://www.midjourney.com/
[5] https://www.youtube.com/watch?v=pgdmhwtsntk

The methods investigated are:

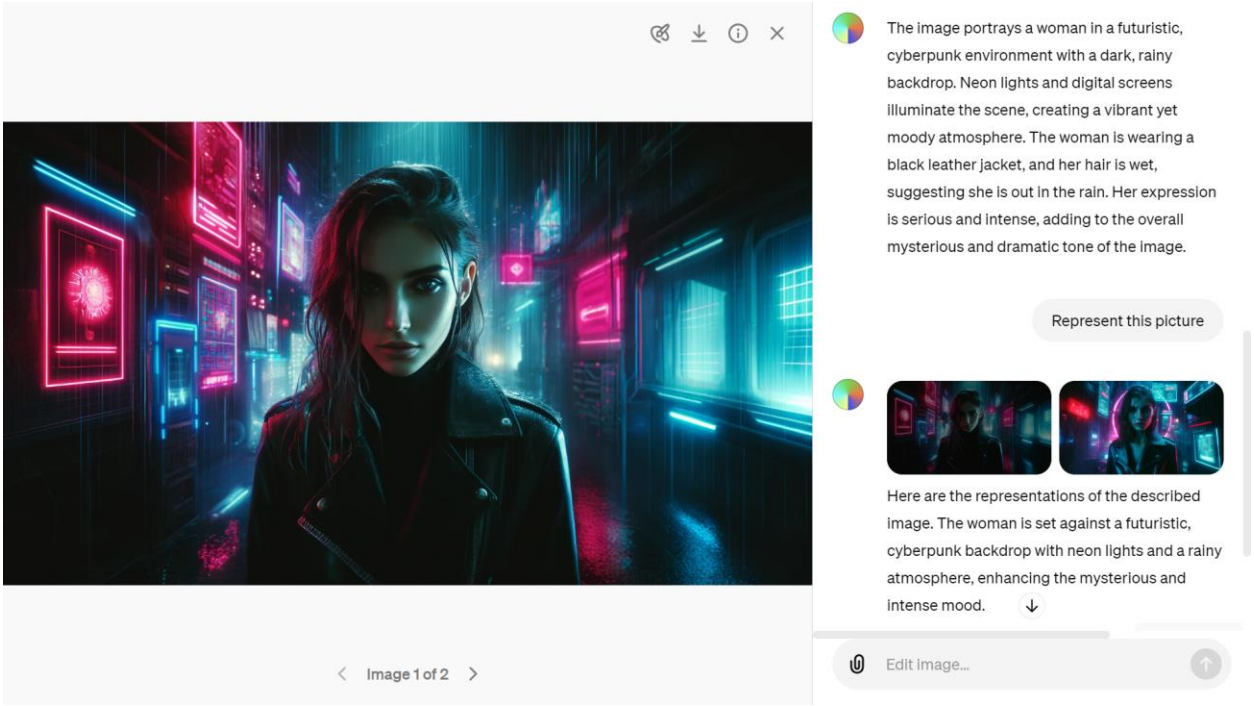
- Generation
- Edits (In- & Outpainting)
- Variation

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.



8.1. 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.



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

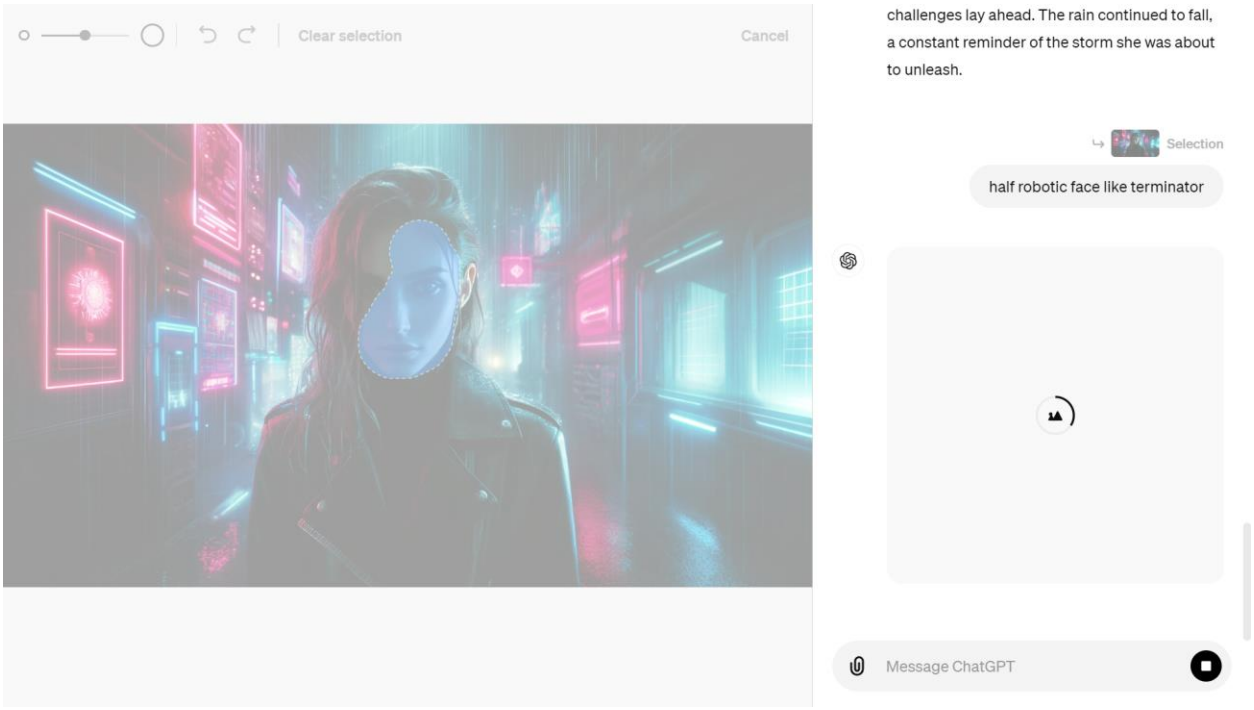
```
from PIL import Image

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
```

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")
```

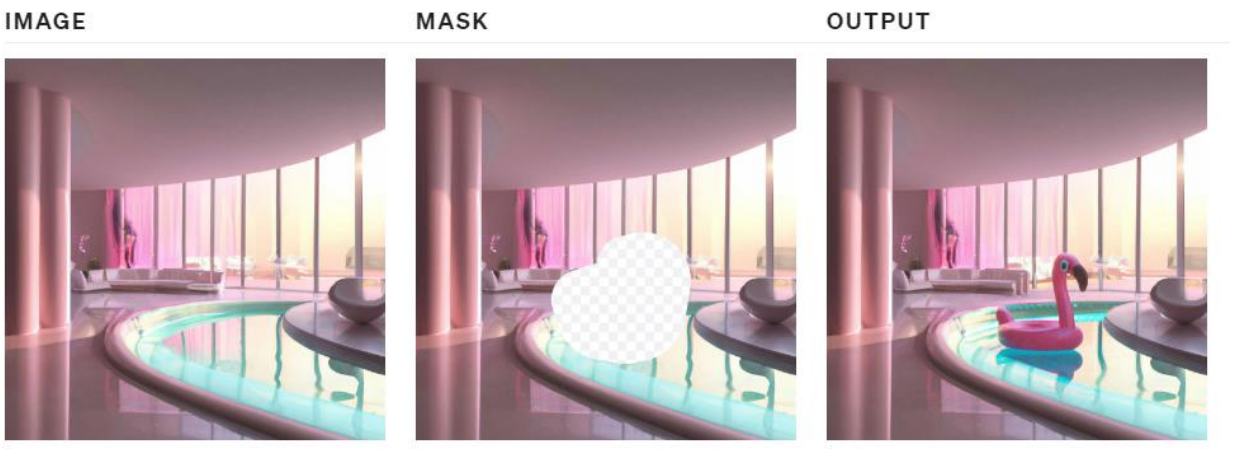
8.2. Edits

Edits can be performed interactively with the ChatGPT app by selecting a region in the image:



“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

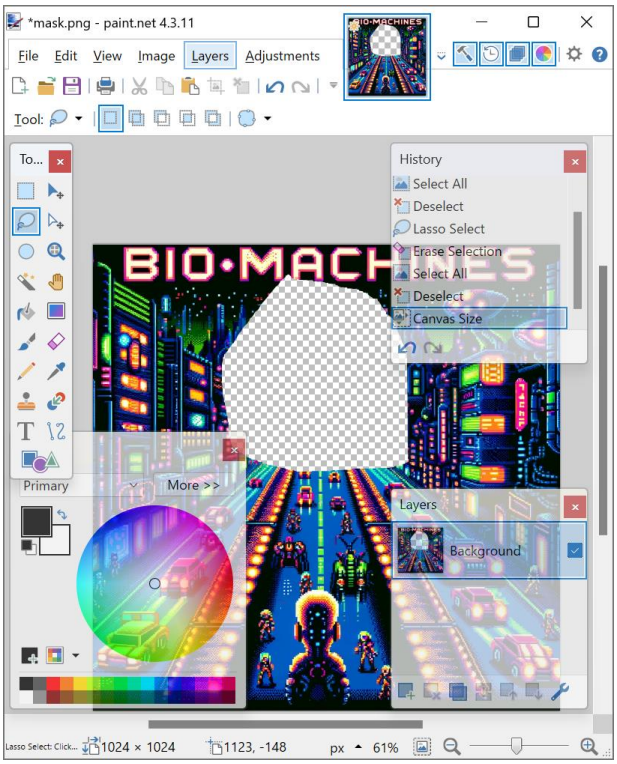
transparent areas of the mask indicate where the image should be edited, and the prompt should describe the full new image, **not just the erased area.**



To perform this operation manually, you can use Paint.net[6] to erase the masked area in the picture.

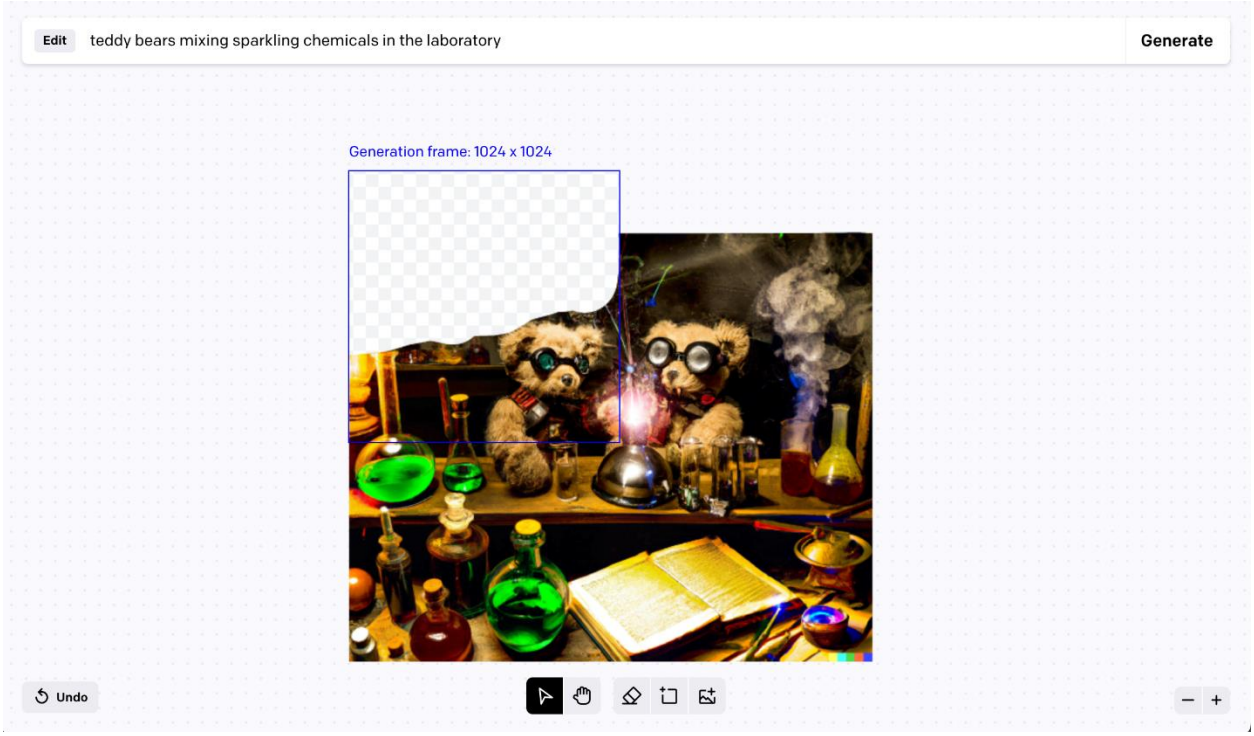
```
response = openai.images.edit(
  model="dall-e-2",
  image=open("biomachines-8bits.png", "rb"),
  mask=open("mask.png", "rb"),
  prompt="""
  Terminator giant skull face in a cyberpunk 8 bits pixelated video game
  """,
  n=1,
  size="1024x1024"
)
image_url = response.data[0].url
image = Image.open(requests.get(image_url, stream = True).raw)
image.save("biomachines_terminator.png")
```

[6] https://www.getpaint.net/

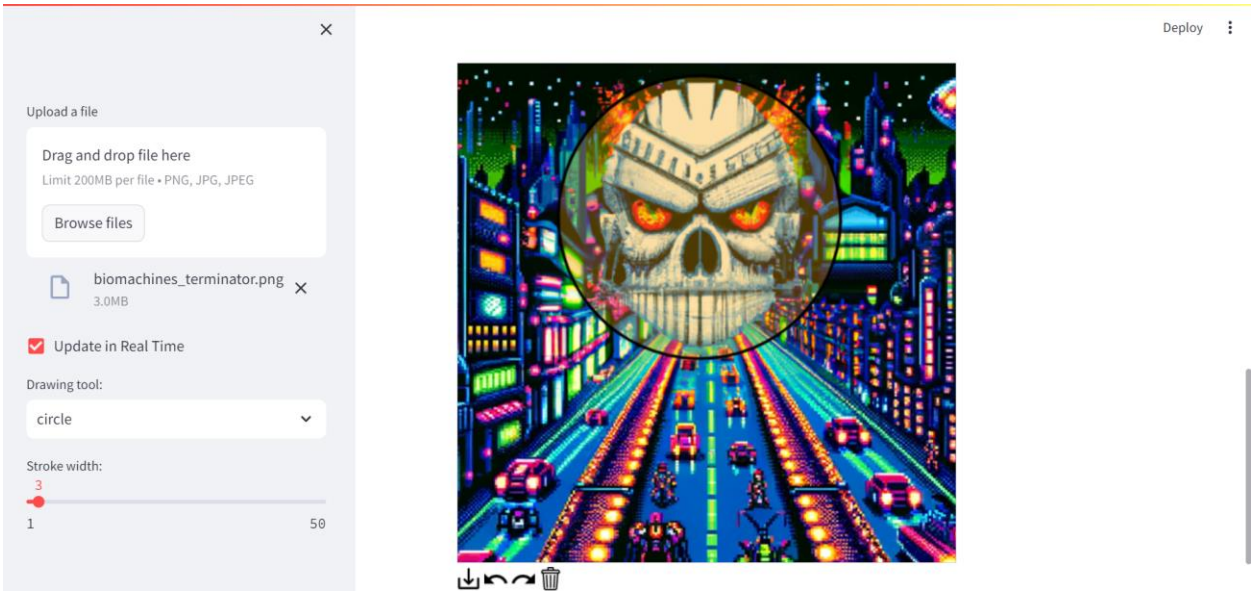


“Outpainting” is following the same principal as “inpainting”, but you only need to shift the image outside of the canvas in the direction that you want to fill.

Before DALL-E got available in the ChatGPT app, you could access it in a lab environment to “outpaint”:



Since this app is no longer available, we can use a Streamlit app with a drawable canvas[7] that will fulfill some of those needs.



8.3. 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.

[7] https://github.com/andfanilo/streamlit-drawable-canvas

8.4. Application: GPT Journey – choose your own adventure

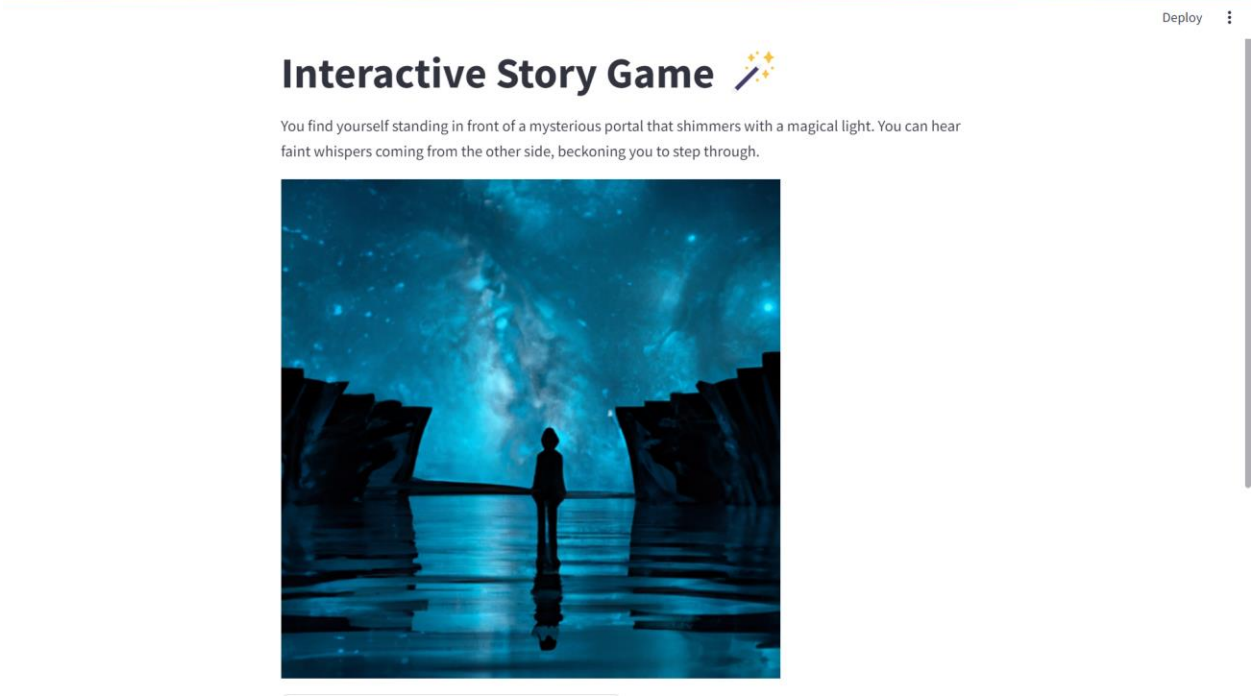
Imagine playing a text-based game where you can not only read the story, but also see the images generated by your choices. That's what GPT Journey offers: a choose your own adventure game powered by artificial intelligence. This concept was developed by the genius Sentdex on his Youtube channel[8]. 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:

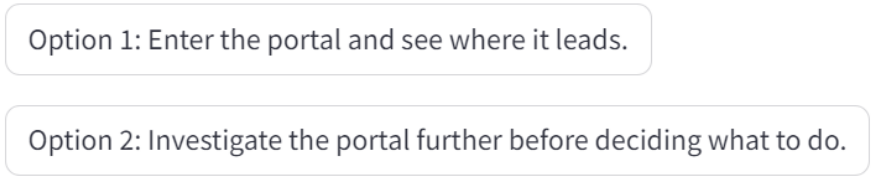
- Use the ChatGPT to create coherent dynamic text responses based on user choices
- Use the Dall-E to create images from text descriptions to produce an engaging game experience
- Build a Streamlit application to host the game online (original version in Flask)

By the end of this section, you will have a fun and creative game that showcases the power of AI for story telling. Let's get started!

[8] 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!