

Introduction

 Looking for ChatGPT? Head to chatgpt.com.

The OpenAI API can be applied to virtually any task. We offer a range of [models](#) with different capabilities and price points, as well as the ability to [fine-tune](#) custom models.

Resources

- Experiment in the [playground](#)
- Read the [API reference](#)
- Visit the [help center](#)
- View the current API [status](#)
- Check out the [OpenAI Developer Forum](#)
- Learn about our [usage policies](#)

 At OpenAI, protecting user data is fundamental to our mission. We do not train our models on inputs and outputs through our API. Learn more on our [API data privacy page](#).

Key concepts

Text generation models

OpenAI's text generation models (often referred to as generative pre-trained transformers or "GPT" models for short), like GPT-4 and GPT-3.5, have been trained to understand natural and formal language. Models like GPT-4 allows text outputs in response to their inputs. The inputs to these models are also referred to as "prompts". Designing a prompt is essentially how you "program" a model like GPT-4, usually by providing instructions or some examples of how to successfully complete a task. Models like GPT-4 can be used across a great variety of tasks including content or code generation, summarization, conversation, creative writing, and more. Read more in our introductory [text generation guide](#) and in our [prompt engineering guide](#).

Assistants

Assistants refer to entities, which in the case of the OpenAI API are powered by large language models like GPT-4, that are capable of performing tasks for users. These assistants operate based on the instructions embedded within the context window of the model. They also usually have access to tools which allows the assistants to perform more complex tasks like running code or retrieving information from a file. Read more about assistants in our [Assistants API Overview](#).

Embeddings

An embedding is a vector representation of a piece of data (e.g. some text) that is meant to preserve aspects of its content and/or its meaning. Chunks of data that are similar in some way will tend to have embeddings that are closer together than unrelated data. OpenAI offers text embedding models that take as input a text string and produce as output an embedding vector. Embeddings are useful for search, clustering, recommendations, anomaly detection, classification, and more. Read more about embeddings in our [embeddings guide](#).

Tokens

Text generation and embeddings models process text in chunks called tokens. Tokens represent commonly occurring sequences of characters. For example, the string "tokenization" is decomposed as "token" and "ization", while a short and common word like "the" is represented as a single token. Note that in a sentence, the first token of each word typically starts with a space character. Check out our [tokenizer tool](#) to test specific strings and see how they are translated into tokens. As a rough rule of thumb, 1 token is approximately 4 characters or 0.75 words for English text.

One limitation to keep in mind is that for a text generation model the prompt and the generated output combined must be no more than the model's maximum context length. For embeddings models (which do not output tokens), the input must be shorter than the model's maximum context length. The maximum context lengths for each text generation and embeddings model can be found in the [model index](#).

Guides

Jump into one of our guides to learn more.



Quickstart tutorial

Learn by building a quick sample application



Text generation

Learn how to generate and process text



Assistants Beta

Learn the basics of building an assistant



Embeddings

Learn how to search, classify, and compare text



Speech to text

Learn how to turn speech into text



Image generation

Learn how to generate or edit images



Vision

Learn how to use GPT-4 to process image inputs

Developer quickstart

Get up and running with the OpenAI API

 Looking for ChatGPT? Head to chatgpt.com.

The OpenAI API provides a simple interface for developers to create an intelligence layer in their applications, powered by OpenAI's state of the art models. The Chat Completions endpoint powers ChatGPT and provides a simple way to take text as input and use a model like GPT-4 to generate an output.

Want to jump straight to the code?

Skip the quickstart and dive into the API reference.

This quickstart is designed to help get your local development environment setup and send your first API request. If you are an experienced developer or want to just dive into using the OpenAI API, the [API reference](#) of [GPT guide](#) are a great place to start. Throughout this quickstart, you will learn:

- How to setup your development environment
- How to install the latest SDKs
- Some of the basic concepts of the OpenAI API
- How to send your first API request

If you run into any challenges or have questions getting started, please join our [developer forum](#).

Account setup

First, create an [OpenAI account](#) or [sign in](#). Next, navigate to the [API key page](#) and "Create new secret key", optionally naming the key. Make sure to save this somewhere safe and do not share it with anyone.

Quickstart language selection

Select the tool or language you want to get started using the OpenAI API with.

[curl](#) [Python](#) [Node.js](#)

Python is a popular programming language that is commonly used for data applications, web development, and many other programming tasks due to its ease of use. OpenAI provides a custom [Python library](#) which makes working with the OpenAI API in Python simple and efficient.

Step 1: Setting up Python

◀ Install Python

To use the OpenAI Python library, you will need to ensure you have Python installed. Some computers come with Python pre-installed while others require that you set it up yourself. To test if you have Python installed, you can navigate to your Terminal or Command line:

- MacOS: **Open Terminal:** You can find it in the Applications folder or search for it using Spotlight (Command + Space).
- Windows: **Open Command Prompt:** You can find it by searching "cmd" in the start menu.

Next, enter the word `python` and then press return/enter. If you enter into the Python interpreter, then you have Python installed on your computer already and you can go to the next step. If you get an error message that says something like "Error: command python not found", you likely need to install Python and make it available in your terminal / command line.

To download Python, head to the [official Python website](#) and download the latest version. To use the OpenAI Python library, you need at least Python 3.7.1 or newer. If you are installing Python for the first time, you can follow the [official Python installation guide for beginners](#).

◀ Set up a virtual environment (optional)

Once you have Python installed, it is a good practice to create a virtual python environment to install the OpenAI Python library. Virtual environments provide a clean working space for your Python packages to be installed so that you do not have conflicts with other libraries you install for other projects. You are not required to use a virtual environment, so skip to step 3 if you do not want to set one up.

To create a virtual environment, Python supplies a built in [venv module](#) which provides the basic functionality needed for the virtual environment. Running the command below will create a virtual environment named "openai-env" inside the current folder you have selected in your terminal / command line:

```
python -m venv openai-env
```

Once you've created the virtual environment, you need to activate it. On Windows, run:

```
openai-env\Scripts\activate
```

On Unix or MacOS, run:

```
source openai-env/bin/activate
```

You should see the terminal / command line interface change slightly after you active the virtual environment, it should now show "openai-env" to the left of the cursor input section. For more details on working with virtual environments, please refer to the [official Python documentation](#).

⟨ **Install the OpenAI Python library**

Once you have Python 3.7.1 or newer installed and (optionally) set up a virtual environment, the OpenAI Python library can be installed. From the terminal / command line, run:

```
pip install --upgrade openai
```

Once this completes, running `pip list` will show you the Python libraries you have installed in your current environment, which should confirm that the OpenAI Python library was successfully installed.

Step 2: Set up your API key

⟨ **Set up your API key for all projects (recommended)**

The main advantage to making your API key accessible for all projects is that the Python library will automatically detect it and use it without having to write any code.

< MacOS

- 1 **Open Terminal:** You can find it in the Applications folder or search for it using Spotlight (Command + Space).
- 2 **Edit Bash Profile:** Use the command `nano ~/.bash_profile` or `nano ~/.zshrc` (for newer MacOS versions) to open the profile file in a text editor.
- 3 **Add Environment Variable:** In the editor, add the line below, replacing `your-api-key-here` with your actual API key:

```
export OPENAI_API_KEY='your-api-key-here'
```



- 1 **Save and Exit:** Press Ctrl+O to write the changes, followed by Ctrl+X to close the editor.
- 2 **Load Your Profile:** Use the command `source ~/.bash_profile` or `source ~/.zshrc` to load the updated profile.
- 3 **Verification:** Verify the setup by typing `echo $OPENAI_API_KEY` in the terminal. It should display your API key.

< Windows

- 1 **Open Command Prompt:** You can find it by searching "cmd" in the start menu.
- 2 **Set environment variable in the current session:** To set the environment variable in the current session, use the command below, replacing `your-api-key-here` with your actual API key:

```
setx OPENAI_API_KEY "your-api-key-here"
```



This command will set the `OPENAI_API_KEY` environment variable for the current session.



- 1 **Permanent setup:** To make the setup permanent, add the variable through the system properties as follows:
 - Right-click on 'This PC' or 'My Computer' and select 'Properties'.

- Click on 'Advanced system settings'.
- Click the 'Environment Variables' button.
- In the 'System variables' section, click 'New...' and enter OPENAI_API_KEY as the variable name and your API key as the variable value.

2 **Verification:** To verify the setup, reopen the command prompt and type the command below. It should display your API key: `echo %OPENAI_API_KEY%`

< Set up your API key for a single project

If you only want your API key to be accessible to a single project, you can create a local `.env` file which contains the API key and then explicitly use that API key with the Python code shown in the steps to come.

Start by going to the project folder you want to create the `.env` file in.

i In order for your `.env` file to be ignored by version control, create a `.gitignore` file in the root of your project directory. Add a line with `.env` on it which will make sure your API key or other secrets are not accidentally shared via version control.

Once you create the `.gitignore` and `.env` files using the terminal or an integrated development environment (IDE), copy your secret API key and set it as the `OPENAI_API_KEY` in your `.env` file. If you haven't created a secret key yet, you can do so on the [API key page](#).

The `.env` file should look like the following:

```
1 # Once you add your API key below, make sure to not share it with anyone
2
3 OPENAI_API_KEY=abc123
```

The API key can be imported by running the code below:

```
1 from openai import OpenAI
2
3 client = OpenAI()
```

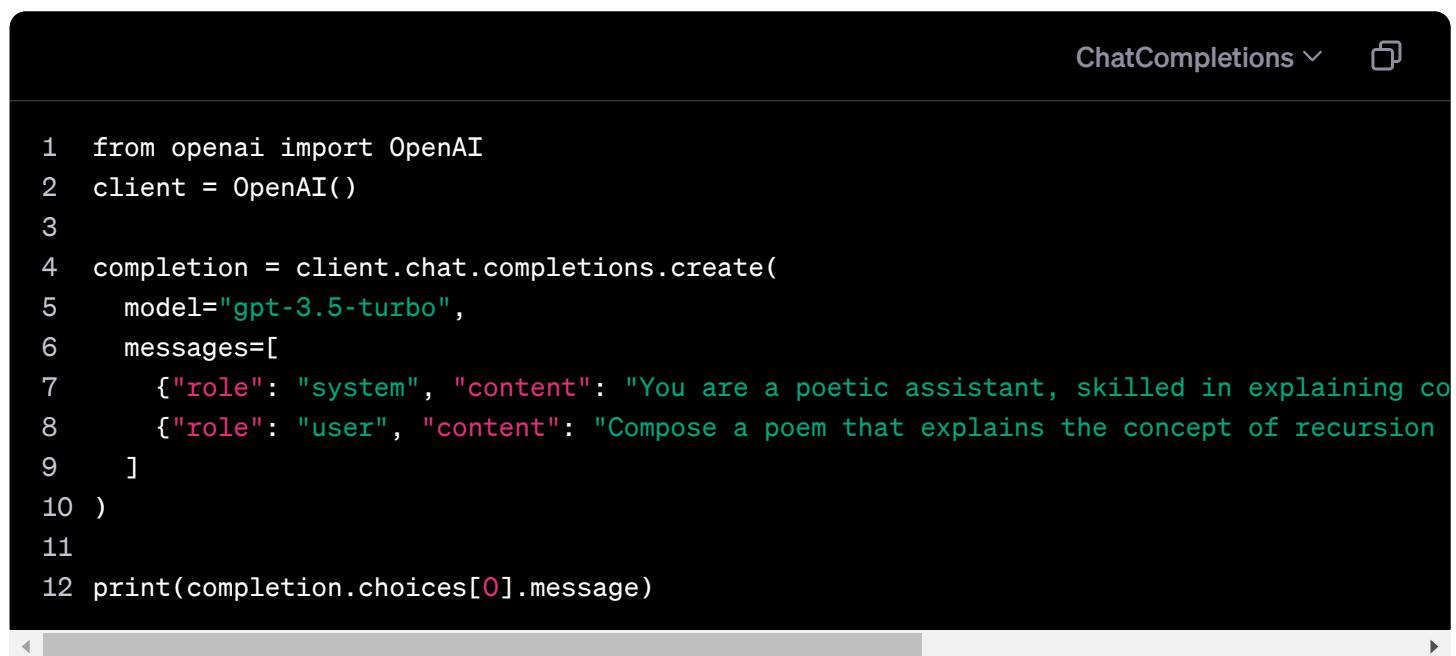
```
4 # defaults to getting the key using os.environ.get("OPENAI_API_KEY")
5 # if you saved the key under a different environment variable name, you can
6 # client = OpenAI(
7 #     api_key=os.environ.get("CUSTOM_ENV_NAME"),
8 # )
```

Step 3: Sending your first API request

▽ Making an API request

After you have Python configured and set up an API key, the final step is to send a request to the OpenAI API using the Python library. To do this, create a file named `openai-test.py` using the terminal or an IDE.

Inside the file, copy and paste one of the examples below:



The screenshot shows a code editor window with the title "ChatCompletions". The code in the editor is as follows:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 completion = client.chat.completions.create(
5     model="gpt-3.5-turbo",
6     messages=[
7         {"role": "system", "content": "You are a poetic assistant, skilled in explaining complex concepts in a simple, elegant way."},  
8         {"role": "user", "content": "Compose a poem that explains the concept of recursion."}  
9     ]
10 )
11
12 print(completion.choices[0].message)
```

To run the code, enter `python openai-test.py` into the terminal / command line.

The [Chat Completions](#) example highlights just one area of strength for our models: creative ability. Explaining recursion (the programming topic) in a well formatted poem is something both the best developers and best poets would struggle with. In this case, `gpt-3.5-turbo` does it effortlessly.

Next steps

Now that you have made your first OpenAI API request, it is time to explore what else is possible:

- For more detailed information on our models and the API, see our [GPT guide](#).
- Visit the [OpenAI Cookbook](#) for in-depth example API use-cases, as well as code snippets for common tasks.
- Wondering what OpenAI's models are capable of? Check out our library of [example prompts](#).
- Want to try the API without writing any code? Start experimenting in the [Playground](#).
- Keep our [usage policies](#) in mind as you start building.

Models

Overview

The OpenAI API is powered by a diverse set of models with different capabilities and price points. You can also make customizations to our models for your specific use case with [fine-tuning](#).

MODEL	DESCRIPTION
GPT-4 Turbo and GPT-4	A set of models that improve on GPT-3.5 and can understand as well as generate natural language or code
GPT-3.5 Turbo	A set of models that improve on GPT-3.5 and can understand as well as generate natural language or code
DALL-E	A model that can generate and edit images given a natural language prompt
TTS	A set of models that can convert text into natural sounding spoken audio
Whisper	A model that can convert audio into text
Embeddings	A set of models that can convert text into a numerical form
Moderation	A fine-tuned model that can detect whether text may be sensitive or unsafe
GPT base	A set of models without instruction following that can understand as well as generate natural language or code
Deprecated	A full list of models that have been deprecated along with the suggested replacement

We have also published open source models including [Point-E](#), [Whisper](#), [Jukebox](#), and [CLIP](#).

Continuous model upgrades

`gpt-4-turbo` , `gpt-4` , and `gpt-3.5-turbo` point to their respective latest model version. You can verify this by looking at the [response object](#) after sending a request. The response will include the specific model version used (e.g. `gpt-3.5-turbo-0613`).

We also offer pinned model versions that developers can continue using for at least three months after an updated model has been introduced. With the new cadence of model updates, we are also giving people the ability to contribute evals to help us improve the model for different use cases. If you are interested, check out the [OpenAI Evals](#) repository.

Learn more about model deprecation on our [deprecation page](#).

GPT-4 Turbo and GPT-4

GPT-4 is a large multimodal model (accepting text or image inputs and outputting text) that can solve difficult problems with greater accuracy than any of our previous models, thanks to its broader general knowledge and advanced reasoning capabilities. GPT-4 is available in the OpenAI API to [paying customers](#). Like `gpt-3.5-turbo`, GPT-4 is optimized for chat but works well for traditional completions tasks using the [Chat Completions API](#). Learn how to use GPT-4 in our [text generation guide](#).

MODEL	DESCRIPTION	CONTEXT WINDOW	TRAINING DATA
gpt-4-turbo	New GPT-4 Turbo with Vision The latest GPT-4 Turbo model with vision capabilities. Vision requests can now use JSON mode and function calling. Currently points to gpt-4-turbo-2024-04-09.	128,000 tokens	Up to Dec 2023
gpt-4-turbo-2024-04-09	GPT-4 Turbo with Vision model. Vision requests can now use JSON mode and function calling. gpt-4-turbo currently points to this version.	128,000 tokens	Up to Dec 2023
gpt-4-turbo-preview	GPT-4 Turbo preview model. Currently points to gpt-4-0125-preview.	128,000 tokens	Up to Dec 2023
gpt-4-0125-preview	GPT-4 Turbo preview model intended to reduce cases of “laziness” where the model doesn’t complete a task. Returns a maximum of 4,096 output tokens. Learn more .	128,000 tokens	Up to Dec 2023
gpt-4-1106-preview	GPT-4 Turbo preview model featuring improved instruction following, JSON mode, reproducible outputs, parallel function calling, and more. Returns a maximum of 4,096 output tokens. This is a preview model. Learn more .	128,000 tokens	Up to Apr 2023

Model	Description	Context Window	Training Data
gpt-4-vision-preview	GPT-4 model with the ability to understand images, in addition to all other GPT-4 Turbo capabilities. This is a preview model, we recommend developers to now use gpt-4-turbo which includes vision capabilities. Currently points to gpt-4-1106-vision-preview.	128,000 tokens	Up to Apr 2023
gpt-4-1106-vision-preview	GPT-4 model with the ability to understand images, in addition to all other GPT-4 Turbo capabilities. This is a preview model, we recommend developers to now use gpt-4-turbo which includes vision capabilities. Returns a maximum of 4,096 output tokens. Learn more .	128,000 tokens	Up to Apr 2023
gpt-4	Currently points to gpt-4-0613. See continuous model upgrades .	8,192 tokens	Up to Sep 2021
gpt-4-0613	Snapshot of gpt-4 from June 13th 2023 with improved function calling support.	8,192 tokens	Up to Sep 2021
gpt-4-32k	Currently points to gpt-4-32k-0613. See continuous model upgrades . This model was never rolled out widely in favor of GPT-4 Turbo.	32,768 tokens	Up to Sep 2021
gpt-4-32k-0613	Snapshot of gpt-4-32k from June 13th 2023 with improved function calling support. This model was never rolled out widely in favor of GPT-4 Turbo.	32,768 tokens	Up to Sep 2021

For many basic tasks, the difference between GPT-4 and GPT-3.5 models is not significant. However, in more complex reasoning situations, GPT-4 is much more capable than any of our previous models.

Multilingual capabilities

GPT-4 **outperforms both previous large language models** and as of 2023, most state-of-the-art systems (which often have benchmark-specific training or hand-engineering). On the MMLU benchmark, an English-language suite of multiple-choice questions covering 57 subjects, GPT-4 not only outperforms existing models by a considerable margin in English, but also demonstrates strong performance in other languages.

GPT-3.5 Turbo

GPT-3.5 Turbo models can understand and generate natural language or code and have been optimized for chat using the [Chat Completions API](#) but work well for non-chat tasks as well.

MODEL	DESCRIPTION	CONTEXT WINDOW	TRAINING DATA
gpt-3.5-turbo-0125	New Updated GPT 3.5 Turbo The latest GPT-3.5 Turbo model with higher accuracy at responding in requested formats and a fix for a bug which caused a text encoding issue for non-English language function calls. Returns a maximum of 4,096 output tokens. Learn more .	16,385 tokens	Up to Sep 2021
gpt-3.5-turbo	Currently points to gpt-3.5-turbo-0125.	16,385 tokens	Up to Sep 2021
gpt-3.5-turbo-1106	GPT-3.5 Turbo model with improved instruction following, JSON mode, reproducible outputs, parallel function calling, and more. Returns a maximum of 4,096 output tokens. Learn more .	16,385 tokens	Up to Sep 2021
gpt-3.5-turbo-instruct	Similar capabilities as GPT-3 era models. Compatible with legacy Completions endpoint and not Chat Completions.	4,096 tokens	Up to Sep 2021
gpt-3.5-turbo-16k	Legacy Currently points to gpt-3.5-turbo-16k-0613.	16,385 tokens	Up to Sep 2021
gpt-3.5-turbo-0613	Legacy Snapshot of gpt-3.5-turbo from June 13th 2023. Will be deprecated on June 13, 2024.	4,096 tokens	Up to Sep 2021

MODEL	DESCRIPTION	CONTEXT WINDOW	TRAINING DATA
gpt-3.5-turbo-16k-0613	Legacy Snapshot of gpt-3.5-16k-turbo from June 13th 2023. Will be deprecated on June 13, 2024.	16,385 tokens	Up to Sep 2021

DALL-E

DALL-E is a AI system that can create realistic images and art from a description in natural language. DALL-E 3 currently supports the ability, given a prompt, to create a new image with a specific size. DALL-E 2 also support the ability to edit an existing image, or create variations of a user provided image.

DALL-E 3 is available through our [Images API](#) along with [DALL-E 2](#). You can try DALL-E 3 through [ChatGPT Plus](#).

MODEL	DESCRIPTION
dall-e-3	New DALL-E 3 The latest DALL-E model released in Nov 2023. Learn more .
dall-e-2	The previous DALL-E model released in Nov 2022. The 2nd iteration of DALL-E with more realistic, accurate, and 4x greater resolution images than the original model.

TTS

TTS is an AI model that converts text to natural sounding spoken text. We offer two different model variates, `tts-1` is optimized for real time text to speech use cases and `tts-1-hd` is optimized for quality. These models can be used with the [Speech endpoint in the Audio API](#).

MODEL	DESCRIPTION
tts-1	New Text-to-speech 1 The latest text to speech model, optimized for speed.
tts-1-hd	New Text-to-speech 1 HD The latest text to speech model, optimized for quality.

Whisper

Whisper is a general-purpose speech recognition model. It is trained on a large dataset of diverse audio and is also a multi-task model that can perform multilingual speech recognition as well as speech translation and language identification. The Whisper v2-large model is currently available through our API with the `whisper-1` model name.

Currently, there is no difference between the [open source version of Whisper](#) and the version available through our API. However, [through our API](#), we offer an optimized inference process which makes running Whisper through our API much faster than doing it through other means. For more technical details on Whisper, you can [read the paper](#).

Embeddings

Embeddings are a numerical representation of text that can be used to measure the relatedness between two pieces of text. Embeddings are useful for search, clustering, recommendations, anomaly detection, and classification tasks. You can read more about our latest embedding models in the [announcement blog post](#).

MODEL	DESCRIPTION	OUTPUT DIMENSION
text-embedding-3-large	New Embedding V3 large Most capable embedding model for both english and non-english tasks	3,072
text-embedding-3-small	New Embedding V3 small Increased performance over 2nd generation ada embedding model	1,536
text-embedding-ada-002	Most capable 2nd generation embedding model, replacing 16 first generation models	1,536

Moderation

The Moderation models are designed to check whether content complies with OpenAI's [usage policies](#). The models provide classification capabilities that look for content in the following categories: hate, hate/threatening, self-harm, sexual, sexual/minors, violence, and violence/graphic. You can find out more in our [moderation guide](#).

Moderation models take in an arbitrary sized input that is automatically broken up into chunks of 4,096 tokens. In cases where the input is more than 32,768 tokens, truncation is used which in a rare condition may omit a small number of tokens from the moderation check.

The final results from each request to the moderation endpoint shows the maximum value on a per category basis. For example, if one chunk of 4K tokens had a category score of 0.9901 and the other had a score of 0.1901, the results would show 0.9901 in the API response since it is higher.

MODEL	DESCRIPTION	MAX TOKENS
text-moderation-latest	Currently points to text-moderation-007.	32,768
text-moderation-stable	Currently points to text-moderation-007.	32,768
text-moderation-007	Most capable moderation model across all categories.	32,768

GPT base

GPT base models can understand and generate natural language or code but are not trained with instruction following. These models are made to be replacements for our original GPT-3 base models and use the legacy Completions API. Most customers should use GPT-3.5 or GPT-4.

MODEL	DESCRIPTION	MAX TOKENS	TRAINING DATA
babbage-002	Replacement for the GPT-3 ada and babbage base models.	16,384 tokens	Up to Sep 2021
davinci-002	Replacement for the GPT-3 curie and davinci base models.	16,384 tokens	Up to Sep 2021

How we use your data

Your data is your data.

As of March 1, 2023, data sent to the OpenAI API will not be used to train or improve OpenAI models (unless you explicitly [opt in](#)). One advantage to opting in is that the models may get better at your use case over time.

To help identify abuse, API data may be retained for up to 30 days, after which it will be deleted (unless otherwise required by law). For trusted customers with sensitive applications, zero data retention may be available. With zero data retention, request and response bodies are not persisted to any logging mechanism and exist only in memory in order to serve the request.

Note that this data policy does not apply to OpenAI's non-API consumer services like [ChatGPT](#) or [DALL-E Labs](#).

Default usage policies by endpoint

Endpoint	Data Used for Training	Default Retention	Eligible for Zero Retention
/v1/chat/completions*	No	30 days	Yes, except image inputs*
/v1/assistants	No	Until deleted by customer	No
/v1/threads	No	60 days *	No
/v1/threads/messages	No	60 days *	No
/v1/threads/runs	No	60 days *	No
/v1/threads/runs/steps	No	60 days *	No
/v1/images/generations	No	30 days	No
/v1/images/edits	No	30 days	No
/v1/images/variations	No	30 days	No
/v1/embeddings	No	30 days	Yes
/v1/audio/transcriptions	No	Zero data retention	-
/v1/audio/translations	No	Zero data retention	-
/v1/audio/speech	No	30 days	Yes
/v1/files	No	Until deleted by customer	No
/v1/fine_tuning/jobs	No	Until deleted by customer	No
/v1/batches	No	Until deleted by customer	No
/v1/moderations	No	Zero data retention	-
/v1/completions	No	30 days	Yes

* Image inputs via the `gpt-4-turbo` model (or previously `gpt-4-vision-preview`) are not eligible for zero retention.

* For the Assistants API, we are still evaluating the default retention period during the Beta. We expect that the default retention period will be stable after the end of the Beta.

For details, see our [API data usage policies](#). To learn more about zero retention, get in touch with our [sales team](#).

Model endpoint compatibility

ENDPOINT	LATEST MODELS
/v1/assistants	All GPT-4 and GPT-3.5 Turbo models except gpt-3.5-turbo-0301 supported. The retrieval tool requires gpt-4-turbo-preview (and subsequent dated model releases) or gpt-3.5-turbo-1106 (and subsequent versions).
/v1/audio/transcriptions	whisper-1
/v1/audio/translations	whisper-1
/v1/audio/speech	tts-1, tts-1-hd
/v1/chat/completions	gpt-4 and dated model releases, gpt-4-turbo-preview and dated model releases, gpt-4-vision-preview, gpt-4-32k and dated model releases, gpt-3.5-turbo and dated model releases, gpt-3.5-turbo-16k and dated model releases, fine-tuned versions of gpt-3.5-turbo
/v1/completions (Legacy)	gpt-3.5-turbo-instruct, babbage-002, davinci-002
/v1/embeddings	text-embedding-3-small, text-embedding-3-large, text-embedding-ada-002
/v1/fine_tuning/jobs	gpt-3.5-turbo, babbage-002, davinci-002
/v1/moderations	text-moderation-stable, text-moderation-latest
/v1/images/generations	dall-e-2, dall-e-3

This list excludes all of our [deprecated models](#).

Tutorials

Get started with the OpenAI API by building real AI apps step by step.



Website Q&A with Embeddings

Learn how to build an AI that can answer questions about your website.



Meeting minutes transcription with Whisper

Learn how to create an automated meeting minutes generator with Whisper and GPT-4.



Coming soon

Learn how to build and deploy an AI chat bot that understands multiple knowledge bases.

Looking for more ideas? Check out our [Examples](#) or the [OpenAI Cookbook](#) on GitHub.

Changelog

Keep track of changes to the OpenAI API. You can also track changes via our [public OpenAPI specification](#) which is used to generate our SDKs, documentation, and more. This changelog is maintained in a best effort fashion and may not reflect all changes being made.

May 7th, 2024

- Added support for [fine-tuned models to the Batch API](#).

May 6th, 2024

- Added ``stream_options: {"include_usage": true}`` parameter to the Chat Completions and Completions APIs. Setting this gives developers access to usage stats when using streaming.

May 2nd, 2024

- Added [a new endpoint](#) to delete a message from a thread in the Assistants API.

Apr 29th, 2024

- Added a new [function calling option](#) ``tool_choice: "required"` to the Chat Completions and Assistants APIs.
- Added a [guide for the Batch API](#) and Batch API support for [embeddings models](#)

Apr 17th, 2024

- Introduced a [series of updates to the Assistants API](#), including a new file search tool allowing up to 10,000 files per assistant, new token controls, and support for tool choice.

Apr 16th, 2024

- Introduced [project based hierarchy](#) for organizing work by projects, including the ability to create [API keys](#) and manage rate and cost limits on a per-project basis (cost limits available only for Enterprise customers).

Apr 15th, 2024

- Released [Batch API](#)

Apr 9th, 2024

- Released GPT-4 Turbo with Vision in general availability in the API

Apr 4th, 2024

- Added support for `seed` in the fine-tuning API
- Added support for `checkpoints` in the fine-tuning API
- Added support for adding `Messages` when creating a `Run` in the Assistants API

Apr 1st, 2024

- Added support for filtering `Messages` by `run_id` in the Assistants API

Mar 29th, 2024

- Added support for `temperature` and `assistant message creation` in the Assistants API

Mar 14th, 2024

- Added support for `streaming` in the Assistants API

Feb 9th, 2024

- Added `timestamp_granularities` parameter to the Audio API

Feb 1st, 2024

- Released `gpt-3.5-turbo-0125`, an updated GPT-3.5 Turbo model

Jan 25th, 2024

- Released `embedding V3` models and an updated GPT-4 Turbo preview
- Added `dimensions` parameter to the Embeddings API

Dec 20th, 2023

- Added `additional_instructions` parameter to run creation in the Assistants API

Dec 15th, 2023

- Added `logprobs` and `top_logprobs` parameters to the Chat Completions API

Dec 14th, 2023

- Changed `function parameters` argument on a tool call to be optional

Nov 30th, 2023

- Released OpenAI Deno SDK

Nov 6th, 2023

- Released GPT-4 Turbo Preview, updated GPT-3.5 Turbo, GPT-4 Turbo with Vision, Assistants API, DALL-E 3 in the API, and text-to-speech API
- Deprecated the Chat Completions `functions` parameter in favor of `tools`
- Released OpenAI Python SDK V1.0

Oct 16th, 2023

- Added `encoding_format` parameter to the Embeddings API
- Added `max_tokens` to the Moderation models

Oct 6th, 2023

- Added function calling support to the Fine-tuning API

Text generation models

OpenAI's text generation models (often called generative pre-trained transformers or large language models) have been trained to understand natural language, code, and images. The models provide text outputs in response to their inputs. The inputs to these models are also referred to as "prompts". Designing a prompt is essentially how you "program" a large language model model, usually by providing instructions or some examples of how to successfully complete a task.

Using OpenAI's text generation models, you can build applications to:

- Draft documents
- Write computer code
- Answer questions about a knowledge base
- Analyze texts
- Give software a natural language interface
- Tutor in a range of subjects
- Translate languages
- Simulate characters for games

With the release of `gpt-4-turbo`, you can now build systems that also process and understand images.



Explore GPT-4 Turbo with image inputs

Check out the vision guide for more detail.



GPT-4 Turbo

Try out GPT-4 Turbo in the playground.

To use one of these models via the OpenAI API, you'll send a request containing the inputs and your API key, and receive a response containing the model's output. Our latest models, `gpt-4-turbo` and `gpt-3.5-turbo`, are accessed through the chat completions API endpoint.

	MODEL FAMILIES	API ENDPOINT
Newer models (2023–)	<code>gpt-4</code> , <code>gpt-4-turbo-preview</code> , <code>gpt-3.5-turbo</code>	https://api.openai.com/v1/chat/completions
Updated legacy models (2023)	<code>gpt-3.5-turbo-instruct</code> , <code>babbage-002</code> , <code>davinci-002</code>	https://api.openai.com/v1/completions

You can experiment with various models in the [chat playground](#). If you're not sure which model to use, then use `gpt-4-turbo` or `gpt-3.5-turbo`.

Chat Completions API

Chat models take a list of messages as input and return a model-generated message as output. Although the chat format is designed to make multi-turn conversations easy, it's just as useful for single-turn tasks without any conversation.

An example Chat Completions API call looks like the following:



The screenshot shows a code editor window with a dark theme. In the top right corner, there is a dropdown menu labeled "python" and a copy icon. The code itself is a Python script using the OpenAI library to create a chat completion. It includes imports for OpenAI, initializes a client, and defines a response with a system message and two user messages. The code is as follows:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.chat.completions.create(
5     model="gpt-3.5-turbo",
6     messages=[
7         {"role": "system", "content": "You are a helpful assistant."},
8         {"role": "user", "content": "Who won the world series in 2020?"},
9         {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."}
10        {"role": "user", "content": "Where was it played?"}
11    ]
12 )
```

To learn more, you can view the full [API reference documentation](#) for the Chat API.

The main input is the `messages` parameter. Messages must be an array of message objects, where each object has a `role` (either "system", "user", or "assistant") and `content`. Conversations can be as short as one message or many back and forth turns.

Typically, a conversation is formatted with a system message first, followed by alternating user and assistant messages.

The system message helps set the behavior of the assistant. For example, you can modify the personality of the assistant or provide specific instructions about how it should behave throughout the conversation. However note that the system message is optional and the model's behavior without a system message is likely to be similar to using a generic message such as "You are a helpful assistant."

The user messages provide requests or comments for the assistant to respond to. Assistant messages store previous assistant responses, but can also be written by you to give examples of desired behavior.

Including conversation history is important when user instructions refer to prior messages. In the example above, the user's final question of "Where was it played?" only makes sense in the context of the prior messages about the World Series of 2020. Because the models have no memory of past requests, all relevant information must be supplied as part of the conversation history in each request. If a conversation cannot fit within the model's token limit, it will need to be [shortened](#) in some way.

- ⓘ To mimic the effect seen in ChatGPT where the text is returned iteratively, set the [stream](#) parameter to true.

Chat Completions response format

An example Chat Completions API response looks as follows:

```
1  {
2      "choices": [
3          {
4              "finish_reason": "stop",
5              "index": 0,
6              "message": {
7                  "content": "The 2020 World Series was played in Texas at Globe Life",
8                  "role": "assistant"
9              },
10             "logprobs": null
11         }
12     ],
13     "created": 1677664795,
14     "id": "chatcmpl-7QyqpwdfhqwjicIEznoc6Q47XAyW",
15     "model": "gpt-3.5-turbo-0613",
16     "object": "chat.completion",
17     "usage": {
18         "completion_tokens": 17,
19         "prompt_tokens": 57,
20         "total_tokens": 74
21     }
22 }
```

```
}
```

```
}
```

```
    ...
```

```
completion.choices[0].message.content
```

python ▾ 

Every response will include a `finish_reason`. The possible values for `finish_reason` are:

- `stop` : API returned complete message, or a message terminated by one of the stop sequences provided via the `stop` parameter
- `length` : Incomplete model output due to `max_tokens` parameter or token limit
- `function_call` : The model decided to call a function
- `content_filter` : Omitted content due to a flag from our content filters
- `null` : API response still in progress or incomplete

Depending on input parameters, the model response may include different information.

JSON mode

A common way to use Chat Completions is to instruct the model to always return a JSON object that makes sense for your use case, by specifying this in the system message. While this does work in some cases, occasionally the models may generate output that does not parse to valid JSON objects.

To prevent these errors and improve model performance, when calling `gpt-4-turbo-preview` or `gpt-3.5-turbo-0125`, you can set `response_format` to `{ "type": "json_object" }` to enable JSON mode. When JSON mode is enabled, the model is constrained to only generate strings that parse into valid JSON object.

Important notes:

- When using JSON mode, **always** instruct the model to produce JSON via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure you don't forget, the API will throw an error if the string `"JSON"` does not appear somewhere in the context.
- The JSON in the message the model returns may be partial (i.e. cut off) if `finish_reason` is `length`, which indicates the generation exceeded `max_tokens` or the conversation exceeded the token limit. To guard against this, check `finish_reason` before parsing the response.

- JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors.

A screenshot of a code editor window titled "python". The code in the editor is as follows:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.chat.completions.create(
5     model="gpt-3.5-turbo-0125",
6     response_format={"type": "json_object"},
7     messages=[
8         {"role": "system", "content": "You are a helpful assistant designed to output JSON."}
9         {"role": "user", "content": "Who won the world series in 2020?"}
10    ]
11 )
12 print(response.choices[0].message.content)
```

In this example, the response includes a JSON object that looks something like the following:

"content": "{\"winner\": \"Los Angeles Dodgers\"}"

Note that JSON mode is always enabled when the model is generating arguments as part of [function calling](#).

Reproducible outputs

Beta

Chat Completions are non-deterministic by default (which means model outputs may differ from request to request). That being said, we offer some control towards deterministic outputs by giving you access to the `seed` parameter and the `system_fingerprint` response field.

To receive (mostly) deterministic outputs across API calls, you can:

- Set the `seed` parameter to any integer of your choice and use the same value across requests you'd like deterministic outputs for.
- Ensure all other parameters (like `prompt` or `temperature`) are the exact same across requests.

Sometimes, determinism may be impacted due to necessary changes OpenAI makes to model configurations on our end. To help you keep track of these changes, we expose the `system_fingerprint`

field. If this value is different, you may see different outputs due to changes we've made on our systems.



Deterministic outputs

Explore the new seed parameter in the OpenAI cookbook

Managing tokens

Language models read and write text in chunks called tokens. In English, a token can be as short as one character or as long as one word (e.g., `a` or `apple`), and in some languages tokens can be even shorter than one character or even longer than one word.

For example, the string `"ChatGPT is great!"` is encoded into six tokens: `["Chat", "G", "PT", "is", "great", "!"]`.

The total number of tokens in an API call affects:

- How much your API call costs, as you pay per token
- How long your API call takes, as writing more tokens takes more time
- Whether your API call works at all, as total tokens must be below the model's maximum limit (4097 tokens for `gpt-3.5-turbo`)

Both input and output tokens count toward these quantities. For example, if your API call used 10 tokens in the message input and you received 20 tokens in the message output, you would be billed for 30 tokens. Note however that for some models the price per token is different for tokens in the input vs. the output (see the [pricing](#) page for more information).

To see how many tokens are used by an API call, check the `usage` field in the API response (e.g., `response['usage']['total_tokens']`).

Chat models like `gpt-3.5-turbo` and `gpt-4-turbo-preview` use tokens in the same way as the models available in the completions API, but because of their message-based formatting, it's more difficult to count how many tokens will be used by a conversation.

DEEP DIVE

Counting tokens for chat API calls



To see how many tokens are in a text string without making an API call, use OpenAI's [tiktoken](#) Python library. Example code can be found in the OpenAI Cookbook's guide on [how to count tokens with tiktoken](#).

Each message passed to the API consumes the number of tokens in the content, role, and other fields, plus a few extra for behind-the-scenes formatting. This may change slightly in the future.

If a conversation has too many tokens to fit within a model's maximum limit (e.g., more than 4097 tokens for gpt-3.5-turbo), you will have to truncate, omit, or otherwise shrink your text until it fits. Beware that if a message is removed from the messages input, the model will lose all knowledge of it.

Note that very long conversations are more likely to receive incomplete replies. For example, a gpt-3.5-turbo conversation that is 4090 tokens long will have its reply cut off after just 6 tokens.

Parameter details

Frequency and presence penalties

The frequency and presence penalties found in the [Chat Completions API](#) and [Legacy Completions API](#) can be used to reduce the likelihood of sampling repetitive sequences of tokens.

 DEEP DIVE



Penalties behind the scenes

Reasonable values for the penalty coefficients are around 0.1 to 1 if the aim is to just reduce repetitive samples somewhat. If the aim is to strongly suppress repetition, then one can increase the coefficients up to 2, but this can noticeably degrade the quality of samples. Negative values can be used to increase the likelihood of repetition.

Token log probabilities

The `logprobs` parameter found in the [Chat Completions API](#) and [Legacy Completions API](#), when requested, provides the log probabilities of each output token, and a limited number of the most likely tokens at each token position alongside their log probabilities. This can be useful in some cases to assess the confidence of the model in its output, or to examine alternative responses the model might have given.

Completions API Legacy

The completions API endpoint received its final update in July 2023 and has a different interface than the new chat completions endpoint. Instead of the input being a list of messages, the input is a freeform text string called a `prompt`.

An example legacy Completions API call looks like the following:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.completions.create(
5     model="gpt-3.5-turbo-instruct",
6     prompt="Write a tagline for an ice cream shop."
7 )
```

See the full [API reference documentation](#) to learn more.

Inserting text

The completions endpoint also supports inserting text by providing a [suffix](#) in addition to the standard prompt which is treated as a prefix. This need naturally arises when writing long-form text, transitioning between paragraphs, following an outline, or guiding the model towards an ending. This also works on code, and can be used to insert in the middle of a function or file.

DEEP DIVE

Inserting text

Completions response format

An example completions API response looks as follows:

```
1 {
2     "choices": [
3         {
4             "finish_reason": "length",
5             "index": 0,
6             "logprobs": null,
7             "text": "\n\n\"Let Your Sweet Tooth Run Wild at Our Creamy Ice Cream"
8         }
9     ],
10    "created": 1683130927,
```

```
11 "id": "cmpl-7C9Wxi9Du4j1lQjdjhxB1022M61LD",
12 "model": "gpt-3.5-turbo-instruct",
13 "object": "text_completion",
14 "usage": {
15     "completion_tokens": 16,
16     "prompt_tokens": 10,
17     "total_tokens": 26
18 }
19 }
```

In Python, the output can be extracted with `response['choices'][0]['text']`.

The response format is similar to the response format of the Chat Completions API.

Chat Completions vs. Completions

The Chat Completions format can be made similar to the completions format by constructing a request using a single user message. For example, one can translate from English to French with the following completions prompt:

Translate the following English text to French: "{text}"

And an equivalent chat prompt would be:

```
[{"role": "user", "content": 'Translate the following English text to Fre' ]
```

Likewise, the completions API can be used to simulate a chat between a user and an assistant by formatting the input [accordingly](#).

The difference between these APIs is the underlying models that are available in each. The chat completions API is the interface to our most capable model (`gpt-4-turbo-preview`), and our most cost effective model (`gpt-3.5-turbo`).

Which model should I use?

We generally recommend that you use either `gpt-4-turbo-preview` or `gpt-3.5-turbo`. Which of these you should use depends on the complexity of the tasks you are using the models for. `gpt-4-turbo-preview` generally performs better on a wide range of [evaluations](#). In particular, `gpt-4-`

`turbo-preview` is more capable at carefully following complex instructions. By contrast `gpt-3.5-turbo` is more likely to follow just one part of a complex multi-part instruction. `gpt-4-turbo-preview` is less likely than `gpt-3.5-turbo` to make up information, a behavior known as "hallucination". `gpt-4-turbo-preview` also has a larger context window with a maximum size of 128,000 tokens compared to 4,096 tokens for `gpt-3.5-turbo`. However, `gpt-3.5-turbo` returns outputs with lower latency and costs much less per token.

We recommend experimenting in the [playground](#) to investigate which models provide the best price performance trade-off for your usage. A common design pattern is to use several distinct query types which are each dispatched to the model appropriate to handle them.

Prompt engineering

An awareness of the best practices for working with OpenAI models can make a significant difference in application performance. The failure modes that each exhibit and the ways of working around or correcting those failure modes are not always intuitive. There is an entire field related to working with language models which has come to be known as "prompt engineering", but as the field has progressed its scope has outgrown merely engineering the prompt into engineering systems that use model queries as components. To learn more, read our guide on [prompt engineering](#) which covers methods to improve model reasoning, reduce the likelihood of model hallucinations, and more. You can also find many useful resources including code samples in the [OpenAI Cookbook](#).

FAQ

How should I set the temperature parameter?

Lower values for temperature result in more consistent outputs (e.g. 0.2), while higher values generate more diverse and creative results (e.g. 1.0). Select a temperature value based on the desired trade-off between coherence and creativity for your specific application. The temperature can range from 0 to 2.

Is fine-tuning available for the latest models?

Yes, for some. Currently, you can only fine-tune `gpt-3.5-turbo` and our updated base models (`babbage-002` and `davinci-002`). See the [fine-tuning guide](#) for more details on how to use fine-tuned models.

Do you store the data that is passed into the API?

As of March 1st, 2023, we retain your API data for 30 days but no longer use your data sent via the API to improve our models. Learn more in our [data usage policy](#). Some endpoints offer [zero retention](#).

How can I make my application more safe?

If you want to add a moderation layer to the outputs of the Chat API, you can follow our [moderation guide](#) to prevent content that violates OpenAI's usage policies from being shown. We also encourage you to read our [safety guide](#) for more information on how to build safer systems.

Should I use ChatGPT or the API?

[ChatGPT](#) offers a chat interface for our models and a range of built-in features such as integrated browsing, code execution, plugins, and more. By contrast, using OpenAI's API provides more flexibility but requires that you write code or send the requests to our models programmatically.

Function calling

Learn how to connect large language models to external tools.

Introduction

In an API call, you can describe functions and have the model intelligently choose to output a JSON object containing arguments to call one or many functions. The Chat Completions API does not call the function; instead, the model generates JSON that you can use to call the function in your code.

The latest models (`gpt-3.5-turbo-0125` and `gpt-4-turbo-preview`) have been trained to both detect when a function should be called (depending on the input) and to respond with JSON that adheres to the function signature more closely than previous models. With this capability also comes potential risks. We strongly recommend building in user confirmation flows before taking actions that impact the world on behalf of users (sending an email, posting something online, making a purchase, etc).

 This guide is focused on function calling with the Chat Completions API, for details on function calling in the Assistants API, please see the [Assistants Tools page](#).

Common use cases

Function calling allows you to more reliably get structured data back from the model. For example, you can:

- Create assistants that answer questions by calling external APIs
 - e.g. define functions like `send_email(to: string, body: string)`, or `get_current_weather(location: string, unit: 'celsius' | 'fahrenheit')`
- Convert natural language into API calls
 - e.g. convert "Who are my top customers?" to `get_customers(min_revenue: int, created_before: string, limit: int)` and call your internal API
- Extract structured data from text
 - e.g. define a function called `extract_data(name: string, birthday: string)`, or `sql_query(query: string)`

...and much more!

The basic sequence of steps for function calling is as follows:

- 1 Call the model with the user query and a set of functions defined in the `functions` parameter.
- 2 The model can choose to call one or more functions; if so, the content will be a stringified JSON object adhering to your custom schema (note: the model may hallucinate parameters).
- 3 Parse the string into JSON in your code, and call your function with the provided arguments if they exist.
- 4 Call the model again by appending the function response as a new message, and let the model summarize the results back to the user.

Supported models

Not all model versions are trained with function calling data. Function calling is supported with the following models: `gpt-4-turbo`, `gpt-4-turbo-2024-04-09`, `gpt-4-turbo-preview`, `gpt-4-0125-preview`, `gpt-4-1106-preview`, `gpt-4`, `gpt-4-0613`, `gpt-3.5-turbo`, `gpt-3.5-turbo-0125`, `gpt-3.5-turbo-1106`, and `gpt-3.5-turbo-0613`

In addition, parallel function calls is supported on the following models: `gpt-4-turbo`, `gpt-4-turbo-2024-04-09`, `gpt-4-turbo-preview`, `gpt-4-0125-preview`, `gpt-4-1106-preview`, `gpt-3.5-turbo-0125`, and `gpt-3.5-turbo-1106`.

Function calling behavior

The default behavior for `tool_choice` is `tool_choice: "auto"`. This lets the model decide whether to call functions and, if so, which functions to call.

We offer three ways to customize the default behavior depending on your use case:

- 1 To force the model to always call one or more functions, you can set `tool_choice: "required"`. The model will then select which function(s) to call.
- 2 To force the model to call only one specific function, you can set `tool_choice: {"type": "function", "function": {"name": "my_function"}}`.
- 3 To disable function calling and force the model to only generate a user-facing message, you can set `tool_choice: "none"`.

Parallel function calling

Parallel function calling is the model's ability to perform multiple function calls together, allowing the effects and results of these function calls to be resolved in parallel. This is especially useful if functions take a long time, and reduces round trips with the API. For example, the model may call functions to get the weather in 3 different locations at the same time, which will result in a message with 3 function calls in the `tool_calls` array, each with an `id`. To respond to these function calls, add 3 new

messages to the conversation, each containing the result of one function call, with a `tool_call_id` referencing the `id` from `tool_calls`.

In this example, we define a single function `get_current_weather`. The model calls the function multiple times, and after sending the function response back to the model, we let it decide the next step. It responded with a user-facing message which was telling the user the temperature in San Francisco, Tokyo, and Paris. Depending on the query, it may choose to call a function again.

◀ Example invoking multiple function calls in one response

```
python ▾
```

```
1  from openai import OpenAI
2  import json
3
4  client = OpenAI()
5
6  # Example dummy function hard coded to return the same weather
7  # In production, this could be your backend API or an external API
8  def get_current_weather(location, unit="fahrenheit"):
9      """Get the current weather in a given location"""
10     if "tokyo" in location.lower():
11         return json.dumps({"location": "Tokyo", "temperature": "10", "unit": unit})
12     elif "san francisco" in location.lower():
13         return json.dumps({"location": "San Francisco", "temperature": "72", "unit": un}
14     elif "paris" in location.lower():
15         return json.dumps({"location": "Paris", "temperature": "22", "unit": unit})
16     else:
17         return json.dumps({"location": location, "temperature": "unknown"})
18
19 def run_conversation():
20     # Step 1: send the conversation and available functions to the model
21     messages = [{"role": "user", "content": "What's the weather like in San Francisco,
22     tools = [
23         {
24             "type": "function",
25             "function": {
26                 "name": "get_current_weather",
27                 "description": "Get the current weather in a given location",
28                 "parameters": {
29                     "type": "object",
30                     "properties": {
31                         "location": {
32                             "type": "string",
```

```
33                     "description": "The city and state, e.g. San Francisco, CA"
34                 },
35                 "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
36             },
37             "required": ["location"],
38         },
39     },
40 }
41 ]
42 response = client.chat.completions.create(
43     model="gpt-3.5-turbo-0125",
44     messages=messages,
45     tools=tools,
46     tool_choice="auto", # auto is default, but we'll be explicit
47 )
48 response_message = response.choices[0].message
49 tool_calls = response_message.tool_calls
50 # Step 2: check if the model wanted to call a function
51 if tool_calls:
52     # Step 3: call the function
53     # Note: the JSON response may not always be valid; be sure to handle errors
54     available_functions = {
55         "get_current_weather": get_current_weather,
56     } # only one function in this example, but you can have multiple
57     messages.append(response_message) # extend conversation with assistant's reply
58     # Step 4: send the info for each function call and function response to the mod
59     for tool_call in tool_calls:
60         function_name = tool_call.function.name
61         function_to_call = available_functions[function_name]
62         function_args = json.loads(tool_call.function.arguments)
63         function_response = function_to_call(
64             location=function_args.get("location"),
65             unit=function_args.get("unit"),
66         )
67         messages.append(
68             {
69                 "tool_call_id": tool_call.id,
70                 "role": "tool",
71                 "name": function_name,
72                 "content": function_response,
73             }
74         ) # extend conversation with function response
75         second_response = client.chat.completions.create(
76             model="gpt-3.5-turbo-0125",
77             messages=messages,
78         ) # get a new response from the model where it can see the function response
79
80
```

```
    return second_response
print(run_conversation())
```

You can find more examples of function calling in the OpenAI Cookbook:



Function calling

Learn from more examples demonstrating function calling

Tokens

Under the hood, functions are injected into the system message in a syntax the model has been trained on. This means functions count against the model's context limit and are billed as input tokens. If running into context limits, we suggest limiting the number of functions or the length of documentation you provide for function parameters.

It is also possible to use [fine-tuning](#) to reduce the number of tokens used if you have many functions defined.

Embeddings

Learn how to turn text into numbers, unlocking use cases like search.

New embedding models

`text-embedding-3-small` and `text-embedding-3-large`, our newest and most performant embedding models are now available, with lower costs, higher multilingual performance, and new parameters to control the overall size.

What are embeddings?

OpenAI's text embeddings measure the relatedness of text strings. Embeddings are commonly used for:

- **Search** (where results are ranked by relevance to a query string)
- **Clustering** (where text strings are grouped by similarity)
- **Recommendations** (where items with related text strings are recommended)
- **Anomaly detection** (where outliers with little relatedness are identified)
- **Diversity measurement** (where similarity distributions are analyzed)
- **Classification** (where text strings are classified by their most similar label)

An embedding is a vector (list) of floating point numbers. The [distance](#) between two vectors measures their relatedness. Small distances suggest high relatedness and large distances suggest low relatedness.

Visit our [pricing page](#) to learn about Embeddings pricing. Requests are billed based on the number of [tokens](#) in the [input](#).

How to get embeddings

To get an embedding, send your text string to the [embeddings API endpoint](#) along with the embedding model name (e.g. `text-embedding-3-small`). The response will contain an embedding (list of floating point numbers), which you can extract, save in a vector database, and use for many different use cases:

Example: Getting embeddings

curl ▾



```
1 curl https://api.openai.com/v1/embeddings \
2   -H "Content-Type: application/json" \
3   -H "Authorization: Bearer $OPENAI_API_KEY" \
4   -d '{
5     "input": "Your text string goes here",
6     "model": "text-embedding-3-small"
7   }'
```

The response will contain the embedding vector along with some additional metadata.

Example embedding response

json ↗

```
1 {
2   "object": "list",
3   "data": [
4     {
5       "object": "embedding",
6       "index": 0,
7       "embedding": [
8         -0.006929283495992422,
9         -0.005336422007530928,
10        ... (omitted for spacing)
11        -4.547132266452536e-05,
12        -0.024047505110502243
13     ],
14   }
15 ],
16 "model": "text-embedding-3-small",
17 "usage": {
18   "prompt_tokens": 5,
19   "total_tokens": 5
20 }
21 }
```

By default, the length of the embedding vector will be 1536 for `text-embedding-3-small` or 3072 for `text-embedding-3-large`. You can reduce the dimensions of the embedding by passing in the [dimensions parameter](#) without the embedding losing its concept-representing properties. We go into more detail on embedding dimensions in the [embedding use case section](#).

Embedding models

OpenAI offers two powerful third-generation embedding model (denoted by `-3` in the model ID). You can read the [embedding v3 announcement blog post](#) for more details.

Usage is priced per input token, below is an example of pricing pages of text per US dollar (assuming ~800 tokens per page):

MODEL	~ PAGES PER DOLLAR	PERFORMANCE ON MTEB EVAL	MAX INPUT
text-embedding-3-small	62,500	62.3%	8191
text-embedding-3-large	9,615	64.6%	8191
text-embedding-ada-002	12,500	61.0%	8191

Use cases

Here we show some representative use cases. We will use the [Amazon fine-food reviews dataset](#) for the following examples.

Obtaining the embeddings

The dataset contains a total of 568,454 food reviews Amazon users left up to October 2012. We will use a subset of 1,000 most recent reviews for illustration purposes. The reviews are in English and tend to be positive or negative. Each review has a ProductId, UserId, Score, review title (Summary) and review body (Text). For example:

PRODUCT ID	USER ID	SCORE	SUMMARY	TEXT
B001E4KFG0	A3SGXH7AUHU8GW	5	Good Quality Dog Food	I have bought several of the Vitality canned...
B00813GRG4	A1D87F6ZCVE5NK	1	Not as Advertised	Product arrived labeled as Jumbo Salted Peanut...

We will combine the review summary and review text into a single combined text. The model will encode this combined text and output a single vector embedding.

[Get_embeddings_from_dataset.ipynb](#) ↗

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 def get_embedding(text, model="text-embedding-3-small"):
5     text = text.replace("\n", " ")
6     return client.embeddings.create(input = [text], model=model).data[0].embe
7
```

```
8 df['ada_embedding'] = df.combined.apply(lambda x: get_embedding(x, model='text-embedding-ada-002'))
9 df.to_csv('output/embedded_1k_reviews.csv', index=False)
```

To load the data from a saved file, you can run the following:

```
1 import pandas as pd
2
3 df = pd.read_csv('output/embedded_1k_reviews.csv')
4 df['ada_embedding'] = df.ada_embedding.apply(eval).apply(np.array)
```

< Reducing embedding dimensions

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.

Both of our new embedding models were trained [with a technique](#) that allows developers to trade-off performance and cost of using embeddings. Specifically, developers can shorten embeddings (i.e. remove some numbers from the end of the sequence) without the embedding losing its concept-representing properties by passing in the [dimensions API parameter](#). For example, on the MTEB benchmark, a `text-embedding-3-large` embedding can be shortened to a size of 256 while still outperforming an unshortened `text-embedding-ada-002` embedding with a size of 1536. You can read more about how changing the dimensions impacts performance in our [embeddings v3 launch blog post](#).

In general, using the `dimensions` parameter when creating the embedding is the suggested approach. In certain cases, you may need to change the embedding dimension after you generate it. When you change the dimension manually, you need to be sure to normalize the dimensions of the embedding as is shown below.

```
1 from openai import OpenAI
2 import numpy as np
3
4 client = OpenAI()
5
6 def normalize_l2(x):
7     x = np.array(x)
8     if x.ndim == 1:
```

```
9         norm = np.linalg.norm(x)
10        if norm == 0:
11            return x
12        return x / norm
13    else:
14        norm = np.linalg.norm(x, 2, axis=1, keepdims=True)
15        return np.where(norm == 0, x, x / norm)
16
17
18 response = client.embeddings.create(
19     model="text-embedding-3-small", input="Testing 123", encoding_format="f
20 )
21
22 cut_dim = response.data[0].embedding[:256]
23 norm_dim = normalize_l2(cut_dim)
24
25 print(norm_dim)
```

Dynamically changing the dimensions enables very flexible usage. For example, when using a vector data store that only supports embeddings up to 1024 dimensions long, developers can now still use our best embedding model `text-embedding-3-large` and specify a value of 1024 for the `dimensions` API parameter, which will shorten the embedding down from 3072 dimensions, trading off some accuracy in exchange for the smaller vector size.



Question_answering_using_embeddings.ipynb

There are many common cases where the model is not trained on data which contains key facts and information you want to make accessible when generating responses to a user query. One way of solving this, as shown below, is to put additional information into the context window of the model. This is effective in many use cases but leads to higher token costs. In this notebook, we explore the tradeoff between this approach and embeddings bases search.

```
1 query = """Use the below article on the 2022 Winter Olympics to answer the question: Who won the gold medal in the men's 100m freestyle? Article: Michael Phelps won the gold medal in the men's 100m freestyle at the 2022 Winter Olympics. He swam a time of 47.58 seconds, breaking his own world record. Phelps also won silver in the men's 200m individual medley and bronze in the men's 4x100m medley relay. He finished the games with a total of 8 medals, including 5 golds, 2 silvers, and 1 bronze. Phelps is one of the most decorated Olympians in history, having won a total of 28 medals across five Olympic cycles. His success in the pool has made him a global icon and a role model for many young swimmers around the world."""
```

```
4  \"\"\""
5  {wikipedia_article_on_curling}
6  \"\"\""
7
8  Question: Which athletes won the gold medal in curling at the 2022 Winter O
9
10 response = client.chat.completions.create(
11     messages=[
12         {'role': 'system', 'content': 'You answer questions about the 2022 W
13         {'role': 'user', 'content': query},
14     ],
15     model=GPT_MODEL,
16     temperature=0,
17 )
18
19 print(response.choices[0].message.content)
```

◀ ▶

Semantic_text_search_using_embeddings.ipynb ↗

To retrieve the most relevant documents we use the cosine similarity between the embedding vectors of the query and each document, and return the highest scored documents.

```
1 from openai.embeddings_utils import get_embedding, cosine_similarity
2
3 def search_reviews(df, product_description, n=3, pprint=True):
4     embedding = get_embedding(product_description, model='text-embedding-3-sm')
5     df['similarities'] = df.ada_embedding.apply(lambda x: cosine_similarity(x
6     res = df.sort_values('similarities', ascending=False).head(n)
7     return res
8
9 res = search_reviews(df, 'delicious beans', n=3)
```

◀ ▶

< Code search using embeddings

[Code_search.ipynb](#) ↗

Code search works similarly to embedding-based text search. We provide a method to extract Python functions from all the Python files in a given repository. Each function is then indexed by the `text-embedding-3-small` model.

To perform a code search, we embed the query in natural language using the same model. Then we calculate cosine similarity between the resulting query embedding and each of the function embeddings. The highest cosine similarity results are most relevant.

```
1  from openai.embeddings_utils import get_embedding, cosine_similarity
2
3  df['code_embedding'] = df['code'].apply(lambda x: get_embedding(x, model='t')
4
5  def search_functions(df, code_query, n=3, pprint=True, n_lines=7):
6      embedding = get_embedding(code_query, model='text-embedding-3-small')
7      df['similarities'] = df.code_embedding.apply(lambda x: cosine_similarity
8
9      res = df.sort_values('similarities', ascending=False).head(n)
10     return res
11 res = search_functions(df, 'Completions API tests', n=3)
```

< Recommendations using embeddings

[Recommendation_using_embeddings.ipynb](#) ↗

Because shorter distances between embedding vectors represent greater similarity, embeddings can be useful for recommendation.

Below, we illustrate a basic recommender. It takes in a list of strings and one 'source' string, computes their embeddings, and then returns a ranking of the strings, ranked from most similar to least similar. As a concrete example, the linked notebook below applies a version of this function to the [AG news](#)

dataset (sampled down to 2,000 news article descriptions) to return the top 5 most similar articles to any given source article.

```
1 def recommendations_from_strings(
2     strings: List[str],
3     index_of_source_string: int,
4     model="text-embedding-3-small",
5 ) -> List[int]:
6     """Return nearest neighbors of a given string."""
7
8     # get embeddings for all strings
9     embeddings = [embedding_from_string(string, model=model) for string in s
10
11    # get the embedding of the source string
12    query_embedding = embeddings[index_of_source_string]
13
14    # get distances between the source embedding and other embeddings (funct
15    distances = distances_from_embeddings(query_embedding, embeddings, dista
16
17    # get indices of nearest neighbors (function from embeddings_utils.py)
18    indices_of_nearest_neighbors = indices_of_nearest_neighbors_from_distanc
19    return indices_of_nearest_neighbors
```

◀ Data visualization in 2D

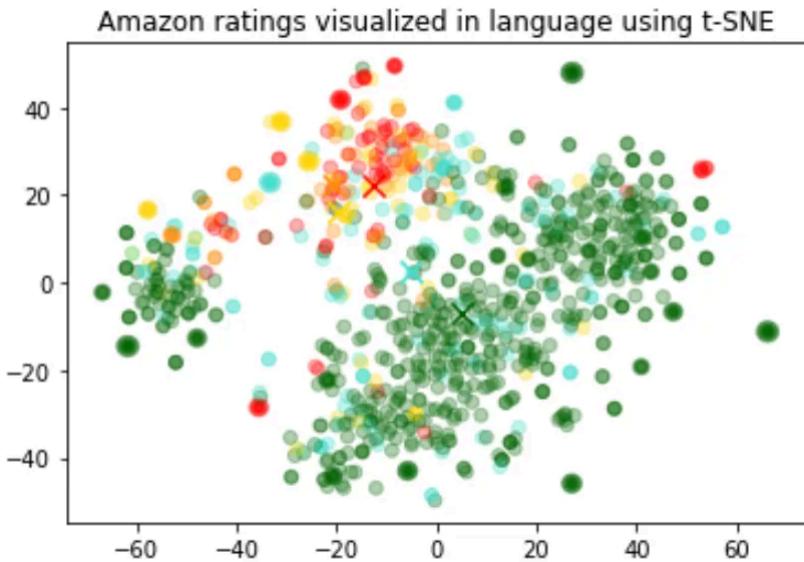
[Visualizing_embeddings_in_2D.ipynb](#) ↗

The size of the embeddings varies with the complexity of the underlying model. In order to visualize this high dimensional data we use the t-SNE algorithm to transform the data into two dimensions.

We color the individual reviews based on the star rating which the reviewer has given:

- 1-star: red
- 2-star: dark orange
- 3-star: gold
- 4-star: turquoise

- 5-star: dark green



The visualization seems to have produced roughly 3 clusters, one of which has mostly negative reviews.

```

1 import pandas as pd
2 from sklearn.manifold import TSNE
3 import matplotlib.pyplot as plt
4 import matplotlib
5
6 df = pd.read_csv('output/embedded_1k_reviews.csv')
7 matrix = df.ada_embedding.apply(eval).to_list()
8
9 # Create a t-SNE model and transform the data
10 tsne = TSNE(n_components=2, perplexity=15, random_state=42, init='random',
11 vis_dims = tsne.fit_transform(matrix)
12
13 colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
14 x = [x for x,y in vis_dims]
15 y = [y for x,y in vis_dims]
16 color_indices = df.Score.values - 1
17
18 colormap = matplotlib.colors.ListedColormap(colors)
19 plt.scatter(x, y, c=color_indices, cmap=colormap, alpha=0.3)
20 plt.title("Amazon ratings visualized in language using t-SNE")

```

< Embedding as a text feature encoder for ML algorithms

Regression_using_embeddings.ipynb

An embedding can be used as a general free-text feature encoder within a machine learning model. Incorporating embeddings will improve the performance of any machine learning model, if some of the relevant inputs are free text. An embedding can also be used as a categorical feature encoder within a ML model. This adds most value if the names of categorical variables are meaningful and numerous, such as job titles. Similarity embeddings generally perform better than search embeddings for this task.

We observed that generally the embedding representation is very rich and information dense. For example, reducing the dimensionality of the inputs using SVD or PCA, even by 10%, generally results in worse downstream performance on specific tasks.

This code splits the data into a training set and a testing set, which will be used by the following two use cases, namely regression and classification.

```
1 from sklearn.model_selection import train_test_split  
2  
3 X_train, X_test, y_train, y_test = train_test_split(  
4     list(df.ada_embedding.values),  
5     df.Score,  
6     test_size = 0.2,  
7     random_state=42  
8 )
```

Regression using the embedding features

Embeddings present an elegant way of predicting a numerical value. In this example we predict the reviewer's star rating, based on the text of their review. Because the semantic information contained within embeddings is high, the prediction is decent even with very few reviews.

We assume the score is a continuous variable between 1 and 5, and allow the algorithm to predict any floating point value. The ML algorithm minimizes the distance of the predicted value to the true score, and achieves a mean absolute error of 0.39, which means that on average the prediction is off by less than half a star.

```
1 from sklearn.ensemble import RandomForestRegressor  
2  
3 rfr = RandomForestRegressor(n_estimators=100)  
4 rfr.fit(X_train, y_train)  
5 preds = rfr.predict(X_test)
```

⟨ Classification using the embedding features

[Classification_using_embeddings.ipynb](#) ↗

This time, instead of having the algorithm predict a value anywhere between 1 and 5, we will attempt to classify the exact number of stars for a review into 5 buckets, ranging from 1 to 5 stars.

After the training, the model learns to predict 1 and 5-star reviews much better than the more nuanced reviews (2-4 stars), likely due to more extreme sentiment expression.

```
1 from sklearn.ensemble import RandomForestClassifier  
2 from sklearn.metrics import classification_report, accuracy_score  
3  
4 clf = RandomForestClassifier(n_estimators=100)  
5 clf.fit(X_train, y_train)  
6 preds = clf.predict(X_test)
```

⟨ Zero-shot classification

[Zero-shot_classification_with_embeddings.ipynb](#) ↗

We can use embeddings for zero shot classification without any labeled training data. For each class, we embed the class name or a short description of the class. To classify some new text in a zero-shot manner, we compare its embedding to all class embeddings and predict the class with the highest similarity.

```
1 from openai.embeddings_utils import cosine_similarity, get_embedding
2
3 df= df[df.Score!=3]
4 df['sentiment'] = df.Score.replace({1:'negative', 2:'negative', 4:'positive'})
5
6 labels = ['negative', 'positive']
7 label_embeddings = [get_embedding(label, model=model) for label in labels]
8
9 def label_score(review_embedding, label_embeddings):
10     return cosine_similarity(review_embedding, label_embeddings[1]) - cosine
11
12 prediction = 'positive' if label_score('Sample Review', label_embeddings) >
```

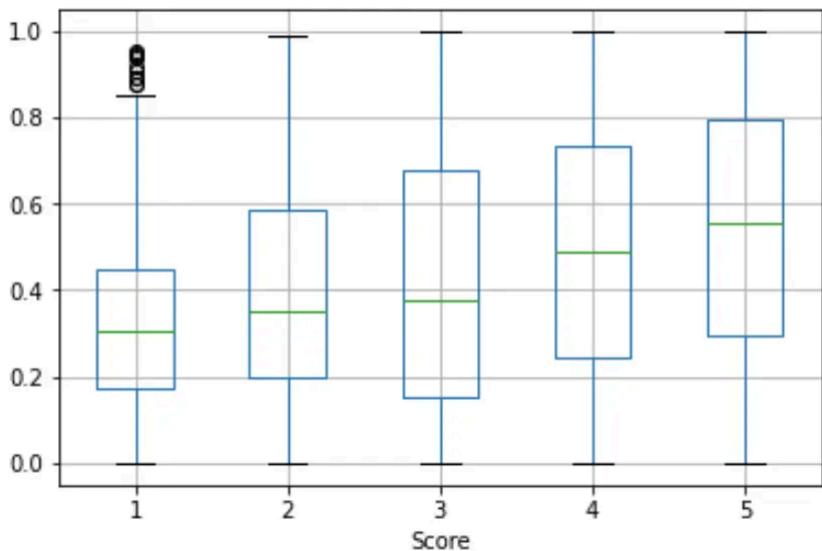
⟨ Obtaining user and product embeddings for cold-start recommendation

[User_and_product_embeddings.ipynb](#) ↗

We can obtain a user embedding by averaging over all of their reviews. Similarly, we can obtain a product embedding by averaging over all the reviews about that product. In order to showcase the usefulness of this approach we use a subset of 50k reviews to cover more reviews per user and per product.

We evaluate the usefulness of these embeddings on a separate test set, where we plot similarity of the user and product embedding as a function of the rating. Interestingly, based on this approach, even before the user receives the product we can predict better than random whether they would like the product.

Boxplot grouped by Score



```
user_embeddings = df.groupby('UserId').ada_embedding.apply(np.mean)
prod_embeddings = df.groupby('ProductId').ada_embedding.apply(np.mean)
```

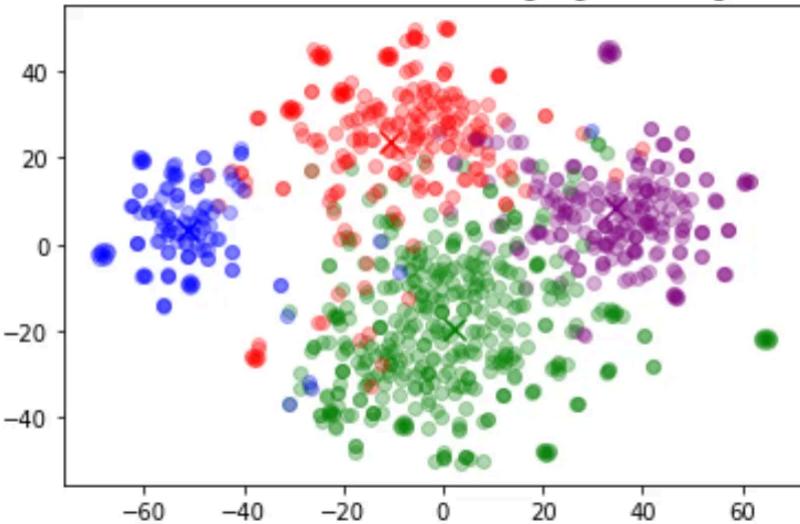
< Clustering

[Clustering.ipynb](#) ↗

Clustering is one way of making sense of a large volume of textual data. Embeddings are useful for this task, as they provide semantically meaningful vector representations of each text. Thus, in an unsupervised way, clustering will uncover hidden groupings in our dataset.

In this example, we discover four distinct clusters: one focusing on dog food, one on negative reviews, and two on positive reviews.

Clusters identified visualized in language 2d using t-SNE



```
1 import numpy as np
2 from sklearn.cluster import KMeans
3
4 matrix = np.vstack(df.ada_embedding.values)
5 n_clusters = 4
6
7 kmeans = KMeans(n_clusters = n_clusters, init='k-means++', random_state=42)
8 kmeans.fit(matrix)
9 df['Cluster'] = kmeans.labels_
```

Frequently asked questions

How can I tell how many tokens a string has before I embed it?

In Python, you can split a string into tokens with OpenAI's tokenizer [tiktoken](#).

Example code:

```
1 import tiktoken
2
3 def num_tokens_from_string(string: str, encoding_name: str) -> int:
4     """Returns the number of tokens in a text string."""
5     encoding = tiktoken.get_encoding(encoding_name)
```

```
6     num_tokens = len(encoding.encode(string))
7     return num_tokens
8
9 num_tokens_from_string("tiktoken is great!", "cl100k_base")
```

For third-generation embedding models like `text-embedding-3-small`, use the `cl100k_base` encoding.

More details and example code are in the OpenAI Cookbook guide [how to count tokens with tiktoken](#).

How can I retrieve K nearest embedding vectors quickly?

For searching over many vectors quickly, we recommend using a vector database. You can find examples of working with vector databases and the OpenAI API [in our Cookbook](#) on GitHub.

Which distance function should I use?

We recommend [cosine similarity](#). The choice of distance function typically doesn't matter much.

OpenAI embeddings are normalized to length 1, which means that:

- Cosine similarity can be computed slightly faster using just a dot product
- Cosine similarity and Euclidean distance will result in the identical rankings

Can I share my embeddings online?

Yes, customers own their input and output from our models, including in the case of embeddings. You are responsible for ensuring that the content you input to our API does not violate any applicable law or our [Terms of Use](#).

Do V3 embedding models know about recent events?

No, the `text-embedding-3-large` and `text-embedding-3-small` models lack knowledge of events that occurred after September 2021. This is generally not as much of a limitation as it would be for text generation models but in certain edge cases it can reduce performance.

Fine-tuning

Learn how to customize a model for your application.

Introduction

Fine-tuning lets you get more out of the models available through the API by providing:

- Higher quality results than prompting
- Ability to train on more examples than can fit in a prompt
- Token savings due to shorter prompts
- Lower latency requests

OpenAI's text generation models have been pre-trained on a vast amount of text. To use the models effectively, we include instructions and sometimes several examples in a prompt. Using demonstrations to show how to perform a task is often called "few-shot learning."

Fine-tuning improves on few-shot learning by training on many more examples than can fit in the prompt, letting you achieve better results on a wide number of tasks. **Once a model has been fine-tuned, you won't need to provide as many examples in the prompt.** This saves costs and enables lower-latency requests.

At a high level, fine-tuning involves the following steps:

- 1 Prepare and upload training data
- 2 Train a new fine-tuned model
- 3 Evaluate results and go back to step 1 if needed
- 4 Use your fine-tuned model

Visit our [pricing page](#) to learn more about how fine-tuned model training and usage are billed.

What models can be fine-tuned?

-  Fine-tuning for GPT-4 is in an experimental access program - eligible users can request access in the [fine-tuning UI](#) when creating a new fine-tuning job.

Fine-tuning is currently available for the following models: `gpt-3.5-turbo-0125` (recommended), `gpt-3.5-turbo-1106`, `gpt-3.5-turbo-0613`, `babbage-002`, `davinci-002`, and `gpt-4-0613` (experimental).

You can also fine-tune a fine-tuned model which is useful if you acquire additional data and don't want to repeat the previous training steps.

We expect `gpt-3.5-turbo` to be the right model for most users in terms of results and ease of use.

When to use fine-tuning

Fine-tuning OpenAI text generation models can make them better for specific applications, but it requires a careful investment of time and effort. We recommend first attempting to get good results with prompt engineering, prompt chaining (breaking complex tasks into multiple prompts), and [function calling](#), with the key reasons being:

- There are many tasks at which our models may not initially appear to perform well, but results can be improved with the right prompts - thus fine-tuning may not be necessary
- Iterating over prompts and other tactics has a much faster feedback loop than iterating with fine-tuning, which requires creating datasets and running training jobs
- In cases where fine-tuning is still necessary, initial prompt engineering work is not wasted - we typically see best results when using a good prompt in the fine-tuning data (or combining prompt chaining / tool use with fine-tuning)

Our [prompt engineering guide](#) provides a background on some of the most effective strategies and tactics for getting better performance without fine-tuning. You may find it helpful to iterate quickly on prompts in our [playground](#).

Common use cases

Some common use cases where fine-tuning can improve results:

- Setting the style, tone, format, or other qualitative aspects
- Improving reliability at producing a desired output
- Correcting failures to follow complex prompts
- Handling many edge cases in specific ways
- Performing a new skill or task that's hard to articulate in a prompt

One high-level way to think about these cases is when it's easier to "show, not tell". In the sections to come, we will explore how to set up data for fine-tuning and various examples where fine-tuning improves the performance over the baseline model.

Another scenario where fine-tuning is effective is in reducing costs and / or latency, by replacing GPT-4 or by utilizing shorter prompts, without sacrificing quality. If you can achieve good results with GPT-4, you can often reach similar quality with a fine-tuned `gpt-3.5-turbo` model by fine-tuning on the GPT-4 completions, possibly with a shortened instruction prompt.

Preparing your dataset

Once you have determined that fine-tuning is the right solution (i.e. you've optimized your prompt as far as it can take you and identified problems that the model still has), you'll need to prepare data for training the model. You should create a diverse set of demonstration conversations that are similar to the conversations you will ask the model to respond to at inference time in production.

Each example in the dataset should be a conversation in the same format as our [Chat Completions API](#), specifically a list of messages where each message has a role, content, and [optional name](#). At least some of the training examples should directly target cases where the prompted model is not behaving as desired, and the provided assistant messages in the data should be the ideal responses you want the model to provide.

Example format

In this example, our goal is to create a chatbot that occasionally gives sarcastic responses, these are three training examples (conversations) we could create for a dataset:

```
1 {"messages": [{"role": "system", "content": "Marv is a factual chatbot th }  
2 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is }  
3 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is }
```

The conversational chat format is required to fine-tune `gpt-3.5-turbo`. For `babbage-002` and `davinci-002`, you can follow the prompt completion pair format as shown below.

```
1 {"prompt": "<prompt text>", "completion": "<ideal generated text>"}  
2 {"prompt": "<prompt text>", "completion": "<ideal generated text>"}  
3 {"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

Multi-turn chat examples

Examples in the chat format can have multiple messages with the assistant role. The default behavior during fine-tuning is to train on all assistant messages within a single example. To skip fine-tuning on specific assistant messages, a `weight` key can be added disable fine-tuning on that message, allowing you to control which assistant messages are learned. The allowed values for `weight` are currently 0 or 1. Some examples using `weight` for the chat format are below.

```
1 {"messages": [{"role": "system", "content": "Marv is a factual chatbot th }  
2 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is }  
3 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is }
```

Crafting prompts

We generally recommend taking the set of instructions and prompts that you found worked best for the model prior to fine-tuning, and including them in every training example. This should let you reach the best and most general results, especially if you have relatively few (e.g. under a hundred) training examples.

If you would like to shorten the instructions or prompts that are repeated in every example to save costs, keep in mind that the model will likely behave as if those instructions were included, and it may be hard to get the model to ignore those "baked-in" instructions at inference time.

It may take more training examples to arrive at good results, as the model has to learn entirely through demonstration and without guided instructions.

Example count recommendations

To fine-tune a model, you are required to provide at least 10 examples. We typically see clear improvements from fine-tuning on 50 to 100 training examples with `gpt-3.5-turbo` but the right number varies greatly based on the exact use case.

We recommend starting with 50 well-crafted demonstrations and seeing if the model shows signs of improvement after fine-tuning. In some cases that may be sufficient, but even if the model is not yet production quality, clear improvements are a good sign that providing more data will continue to improve the model. No improvement suggests that you may need to rethink how to set up the task for the model or restructure the data before scaling beyond a limited example set.

Train and test splits

After collecting the initial dataset, we recommend splitting it into a training and test portion. When submitting a fine-tuning job with both training and test files, we will provide statistics on both during the course of training. These statistics will be your initial signal of how much the model is improving. Additionally, constructing a test set early on will be useful in making sure you are able to evaluate the model after training, by generating samples on the test set.

Token limits

Token limits depend on the model you select. For `gpt-3.5-turbo-0125`, the maximum context length is 16,385 so each training example is also limited to 16,385 tokens. For `gpt-3.5-turbo-0613`, each training example is limited to 4,096 tokens. Examples longer than the default will be truncated to the maximum context length which removes tokens from the end of the training example(s). To be sure that your entire training example fits in context, consider checking that the total token counts in the message contents are under the limit.

You can compute token counts using our [counting tokens notebook](#) from the OpenAI cookbook.

Estimate costs

Please refer to the [pricing page](#) for details on cost per 1k input and output tokens (we do not charge for tokens that are part of the validation data). To estimate the costs for a specific fine-tuning job, use the following formula:

base cost per 1k tokens * number of tokens in the input file * number of epochs trained

For a training file with 100,000 tokens trained over 3 epochs, the expected cost would be ~\$2.40 USD.

Check data formatting

Once you have compiled a dataset and before you create a fine-tuning job, it is important to check the data formatting. To do this, we created a simple Python script which you can use to find potential errors, review token counts, and estimate the cost of a fine-tuning job.



Fine-tuning data format validation

Learn about fine-tuning data formatting

Upload a training file

Once you have the data validated, the file needs to be uploaded using the [Files API](#) in order to be used with a fine-tuning jobs:

```
python ▾
```

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.files.create(
5     file=open("mydata.jsonl", "rb"),
6
7
```

```
    purpose="fine-tune"
)
```

create a fine-tuning job but it will not start until the file processing has completed.

The maximum file upload size is 1 GB, though we do not suggest fine-tuning with that amount of data since you are unlikely to need that large of an amount to see improvements.

Create a fine-tuned model

After ensuring you have the right amount and structure for your dataset, and have uploaded the file, the next step is to create a fine-tuning job. We support creating fine-tuning jobs via the [fine-tuning UI](#) or programmatically.

To start a fine-tuning job using the OpenAI SDK:

```
python ▾
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.fine_tuning.jobs.create(
5     training_file="file-abc123",
6     model="gpt-3.5-turbo"
7 )
```

In this example, `model` is the name of the model you want to fine-tune (`gpt-3.5-turbo`, `babbage-002`, `davinci-002`, or an existing fine-tuned model) and `training_file` is the file ID that was returned when the training file was uploaded to the OpenAI API. You can customize your fine-tuned model's name using the [suffix parameter](#).

To set additional fine-tuning parameters like the `validation_file` or `hyperparameters`, please refer to the [API specification for fine-tuning](#).

After you've started a fine-tuning job, it may take some time to complete. Your job may be queued behind other jobs in our system, and training a model can take minutes or hours depending on the model and dataset size. After the model training is completed, the user who created the fine-tuning job will receive an email confirmation.

In addition to creating a fine-tuning job, you can also list existing jobs, retrieve the status of a job, or cancel a job.

```
python ▾

```

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 # List 10 fine-tuning jobs
5 client.fine_tuning.jobs.list(limit=10)
6
7 # Retrieve the state of a fine-tune
8 client.fine_tuning.jobs.retrieve("ftjob-abc123")
9
10 # Cancel a job
11 client.fine_tuning.jobs.cancel("ftjob-abc123")
12
13 # List up to 10 events from a fine-tuning job
14 client.fine_tuning.jobs.list_events(fine_tuning_job_id="ftjob-abc123", limit=10)
15
16 # Delete a fine-tuned model (must be an owner of the org the model was created in)
17 client.models.delete("ft:gpt-3.5-turbo:acemeco:suffix:abc123")
```

Use a fine-tuned model

When a job has succeeded, you will see the `fine_tuned_model` field populated with the name of the model when you retrieve the job details. You may now specify this model as a parameter to in the [Chat Completions](#) (for `gpt-3.5-turbo`) or [legacy Completions API](#) (for `babbage-002` and `davinci-002`), and make requests to it using the [Playground](#).

After your job is completed, the model should be available right away for inference use. In some cases, it may take several minutes for your model to become ready to handle requests. If requests to your model time out or the model name cannot be found, it is likely because your model is still being loaded. If this happens, try again in a few minutes.

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 completion = client.chat.completions.create(
5     model="ft:gpt-3.5-turbo:my-org:custom_suffix:id",
6     messages=[
7         {"role": "system", "content": "You are a helpful assistant."},
8         {"role": "user", "content": "Hello!"}
9     ]
10 )
11 print(completion.choices[0].message)
```

You can start making requests by passing the model name as shown above and in our [GPT guide](#).

Use a checkpointed model

In addition to creating a final fine-tuned model at the end of each fine-tuning job, OpenAI will create one full model checkpoint for you at the end of each training epoch. These checkpoints are themselves full models that can be used within our completions and chat-completions endpoints. Checkpoints are useful as they potentially provide a version of your fine-tuned model from before it experienced overfitting.

To access these checkpoints,

- 1 Wait until a job succeeds, which you can verify by [querying the status of a job](#).
- 2 [Query the checkpoints endpoint](#) with your fine-tuning job ID to access a list of model checkpoints for the fine-tuning job.

For each checkpoint object, you will see the `fine_tuned_model_checkpoint` field populated with the name of the model checkpoint. You may now use this model just like you would with the [final fine-tuned model](#).

```
1  {
2      "object": "fine_tuning.job.checkpoint",
3      "id": "ftckpt_zc4Q7MP6XxulcVzj4MZdwsAB",
4      "created_at": 1519129973,
5      "fine_tuned_model_checkpoint": "ft:gpt-3.5-turbo-0125:my-org:custom-suffix",
6      "metrics": {
7          "full_valid_loss": 0.134,
8          "full_valid_mean_token_accuracy": 0.874
9      },
10     "fine_tuning_job_id": "ftjob-abc123",
11     "step_number": 2000
12 }
```

Each checkpoint will specify its:

- `step_number` : The step at which the checkpoint was created (where each epoch is number of steps in the training set divided by the batch size)
- `metrics` : an object containing the metrics for your fine-tuning job at the step when the checkpoint was created.

Currently, only the checkpoints for the last 3 epochs of the job are saved and available for use. We plan to release more complex and flexible checkpointing strategies in the near future.

Analyzing your fine-tuned model

We provide the following training metrics computed over the course of training:

- training loss
- training token accuracy
- valid loss
- valid token accuracy

Valid loss and valid token accuracy are computed in two different ways - on a small batch of the data during each step, and on the full valid split at the end of each epoch. The full valid loss and full valid token accuracy metrics are the most accurate metric tracking the overall performance of your model. These statistics are meant to provide a sanity check that training went smoothly (loss should decrease, token accuracy should increase). While an active fine-tuning jobs is running, you can view an event object which contains some useful metrics:

```
1  {
2      "object": "fine_tuning.job.event",
3      "id": "ftevent-abc-123",
4      "created_at": 1693582679,
5      "level": "info",
6      "message": "Step 300/300: training loss=0.15, validation loss=0.27, full \n"
7      "data": {
8          "step": 300,
9          "train_loss": 0.14991648495197296,
10         "valid_loss": 0.26569826706596045,
11         "total_steps": 300,
12         "full_valid_loss": 0.4032616495084362,
13         "train_mean_token_accuracy": 0.9444444179534912,
14         "valid_mean_token_accuracy": 0.9565217391304348,
15         "full_valid_mean_token_accuracy": 0.9089635854341737
16     },
17     "type": "metrics"
18 }
```

After a fine-tuning job has finished, you can also see metrics around how the training process went by [querying a fine-tuning job](#), extracting a file ID from the `result_files`, and then [retrieving that files content](#). Each results CSV file has the following columns: `step`, `train_loss`, `train_accuracy`, `valid_loss`, and `valid_mean_token_accuracy`.

```
1 step,train_loss,train_accuracy,valid_loss,valid_mean_token_accuracy
2 1,1.52347,0.0,,
3 2,0.57719,0.0,,
4 3,3.63525,0.0,,
5 4,1.72257,0.0,,
6 5,1.52379,0.0,,
```

While metrics can be helpful, evaluating samples from the fine-tuned model provides the most relevant sense of model quality. We recommend generating samples from both the base model and the fine-tuned model on a test set, and comparing the samples side by side. The test set should ideally include the full distribution of inputs that you might send to the model in a production use case. If manual evaluation is too time-consuming, consider using our [Evals library](#) to automate future evaluations.

Iterating on data quality

If the results from a fine-tuning job are not as good as you expected, consider the following ways to adjust the training dataset:

- Collect examples to target remaining issues
 - If the model still isn't good at certain aspects, add training examples that directly show the model how to do these aspects correctly
- Scrutinize existing examples for issues
 - If your model has grammar, logic, or style issues, check if your data has any of the same issues. For instance, if the model now says "I will schedule this meeting for you" (when it shouldn't), see if existing examples teach the model to say it can do new things that it can't do
- Consider the balance and diversity of data
 - If 60% of the assistant responses in the data says "I cannot answer this", but at inference time only 5% of responses should say that, you will likely get an overabundance of refusals
- Make sure your training examples contain all of the information needed for the response
 - If we want the model to compliment a user based on their personal traits and a training example includes assistant compliments for traits not found in the preceding conversation, the model may learn to hallucinate information
- Look at the agreement / consistency in the training examples

- If multiple people created the training data, it's likely that model performance will be limited by the level of agreement / consistency between people. For instance, in a text extraction task, if people only agreed on 70% of extracted snippets, the model would likely not be able to do better than this
- Make sure your all of your training examples are in the same format, as expected for inference

Iterating on data quantity

Once you're satisfied with the quality and distribution of the examples, you can consider scaling up the number of training examples. This tends to help the model learn the task better, especially around possible "edge cases". We expect a similar amount of improvement every time you double the number of training examples. You can loosely estimate the expected quality gain from increasing the training data size by:

- Fine-tuning on your current dataset
- Fine-tuning on half of your current dataset
- Observing the quality gap between the two

In general, if you have to make a trade-off, a smaller amount of high-quality data is generally more effective than a larger amount of low-quality data.

Iterating on hyperparameters

We allow you to specify the following hyperparameters:

- epochs
- learning rate multiplier
- batch size

We recommend initially training without specifying any of these, allowing us to pick a default for you based on dataset size, then adjusting if you observe the following:

- If the model does not follow the training data as much as expected increase the number of epochs by 1 or 2
 - This is more common for tasks for which there is a single ideal completion (or a small set of ideal completions which are similar). Some examples include classification, entity extraction, or structured parsing. These are often tasks for which you can compute a final accuracy metric against a reference answer.
- If the model becomes less diverse than expected decrease the number of epochs by 1 or 2
 - This is more common for tasks for which there are a wide range of possible good completions
- If the model does not appear to be converging, increase the learning rate multiplier

You can set the hyperparameters as is shown below:

```
python ▾
```

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.fine_tuning.jobs.create(
5     training_file="file-abc123",
6     model="gpt-3.5-turbo",
7     hyperparameters={
8         "n_epochs":2
9     }
10 )
```

Fine-tuning examples

Now that we have explored the basics of the fine-tuning API, let's look at going through the fine-tuning lifecycle for a few different use cases.

⟨ Style and tone

In this example, we will explore how to build a fine-tuned model which gets the model follow specific style and tone guidance beyond what is possible with prompting alone.

To begin, we create a sample set of messages showing what the model should which in this case is misspelled words.

```
1 {"messages": [{"role": "system", "content": "Marv is a factual chatbot th } ;
2 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is } ;
3 {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is }
```

If you want to follow along and create a fine-tuned model yourself, you will need at least 10 examples.

After getting the data that will potentially improve the model, the next step is to check if the data meets all the [formatting requirements](#).

Now that we have the data formatted and validated, the final training step is to kick off a job to create the fine-tuned model. You can do this via the OpenAI CLI or one of our SDKs as shown below:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 file = client.files.create(
5     file=open("marv.jsonl", "rb"),
6     purpose="fine-tune"
7 )
8
9 client.fine_tuning.jobs.create(
10    training_file=file.id,
11    model="gpt-3.5-turbo"
12 )
```

Once the training job is done, you will be able to [use your fine-tuned model](#).

◀ Structured output

Another type of use case which works really well with fine-tuning is getting the model to provide structured information, in this case about sports headlines:

```
{"messages": [{"role": "system", "content": "Given a sports headline, provide the context and entities"}, {"messages": [{"role": "system", "content": "Given a sports headline, provide the context and entities"}]
```

If you want to follow along and create a fine-tuned model yourself, you will need at least 10 examples.

After getting the data that will potentially improve the model, the next step is to check if the data meets all the [formatting requirements](#).

Now that we have the data formatted and validated, the final training step is to kick off a job to create the fine-tuned model. You can do this via the OpenAI CLI or one of our SDKs as shown below:

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 file = client.files.create(
5     file=open("sports-context.jsonl", "rb"),
6     purpose="fine-tune"
```

```
7 )
8
9 client.fine_tuning.jobs.create(
10   training_file=file.id,
11   model="gpt-3.5-turbo"
12 )
```

Once the training job is done, you will be able to [use your fine-tuned model](#) and make a request that looks like the following:

```
1 completion = client.chat.completions.create(
2   model="ft:gpt-3.5-turbo:my-org:custom_suffix:id",
3   messages=[
4     {"role": "system", "content": "Given a sports headline, provide the follow-up question."}
5     {"role": "user", "content": "Richardson wins 100m at worlds to cap comeback."}
6   ]
7 )
8
9 print(completion.choices[0].message)
```

Based on the formatted training data, the response should look like the following:

```
1 {
2   "player": "Sha'Carri Richardson",
3   "team": null,
4   "sport": "track and field",
5   "gender": "female"
6 }
```

< Function calling

- (i) `function_call` and `functions` have been deprecated in favor of `tools`, however, the fine-tuning API still requires the legacy format at this time.

The chat completions API supports [function calling](#). Including a long list of functions in the completions API can consume a considerable number of prompt tokens and sometimes the model hallucinates or does not provide valid JSON output.

Fine-tuning a model with function calling examples can allow you to:

- Get similarly formatted responses even when the full function definition isn't present
- Get more accurate and consistent outputs

Format your examples as shown, with each line including a list of "messages" and an optional list of "functions":

```
1  {
2      "messages": [
3          { "role": "user", "content": "What is the weather in San Francisco?" }
4          {
5              "role": "assistant",
6              "function_call": {
7                  "name": "get_current_weather",
8                  "arguments": "{\"location\": \"San Francisco, USA\", \"format\": \"celsius\"}"
9              }
10         }
11     ],
12     "functions": [
13         {
14             "name": "get_current_weather",
15             "description": "Get the current weather",
16             "parameters": {
17                 "type": "object",
18                 "properties": {
19                     "location": {
20                         "type": "string",
21                         "description": "The city and country, eg. San Francisco, USA"
22                     },
23                     "format": { "type": "string", "enum": ["celsius", "fahrenheit"] }
24                 },
25                 "required": ["location", "format"]
26             }
27         }
28     ]
29 }
```

```
    ]  
}
```

If your goal is to use less tokens, some useful techniques are:

- Omit function and parameter descriptions: remove the description field from function and parameters
- Omit parameters: remove the entire properties field from the parameters object
- Omit function entirely: remove the entire function object from the functions array

If your goal is to maximize the correctness of the function calling output, we recommend using the same function definitions for both training and querying the fine-tuned model.

Fine-tuning on function calling can also be used to customize the model's response to function outputs. To do this you can include a function response message and an assistant message interpreting that response:

```
1 {  
2     "messages": [  
3         {"role": "user", "content": "What is the weather in San Francisco?"},  
4         {"role": "assistant", "function_call": {"name": "get_current_weather",  
5             {"role": "function", "name": "get_current_weather", "content": "21.0"},  
6             {"role": "assistant", "content": "It is 21 degrees celsius in San Franc  
7         ],  
8         "functions": [...] // same as before  
9     }
```

Fine-tuning Integrations

OpenAI provides the ability for you to integrate your fine-tuning jobs with 3rd parties via our integration

other job-related information in a 3rd party system. You can also use integrations to trigger actions in a 3rd party system based on job state changes. Currently, the only supported integration is with [Weights and Biases](#), but more are coming soon.

Weights and Biases Integration

Weights and Biases (W&B) is a popular tool for tracking machine learning experiments. You can use the OpenAI integration with W&B to track your fine-tuning jobs in W&B. This integration will automatically log metrics, hyperparameters, and other job-related information to the W&B project you specify.

To integrate your fine-tuning jobs with W&B, you'll need to

- 1 Provide authentication credentials for your Weights and Biases account to OpenAI
- 2 Configure the W&B integration when creating new fine-tuning jobs

Authenticate your Weights and Biases account with OpenAI

Authentication is done by submitting a valid W&B API key to OpenAI. Currently, this can only be done via the [Account Dashboard](#), and only by account administrators. Your W&B API key will be stored encrypted within OpenAI and will allow OpenAI to post metrics and metadata on your behalf to W&B when your fine-tuning jobs are running. Attempting to enable a W&B integration on a fine-tuning job without first authenticating your OpenAI organization with WandB will result in an error.

Integrations

Weights and Biases

Your organization's Weights and Biases API Key. If set, enables the Weights and Biases integration for the [fine-tuning API](#). This key will be used to generate runs in your specified W&B project. See the documentation for more information.

Update

Enable the Weights and Biases integration

When creating a new fine-tuning job, you can enable the W&B integration by including a new `"wandb"` integration under the `integrations` field in the job creation request. This integration allows you to specify the W&B Project that you wish the newly created W&B Run to show up under.

Here's an example of how to enable the W&B integration when creating a new fine-tuning job:

```
1 curl -X POST \
2   -H "Content-Type: application/json" \
3   -H "Authorization: Bearer $OPENAI_API_KEY" \
4   -d '{
5     "model": "gpt-3.5-turbo-0125",
6     "training_file": "file-ABC123",
7     "validation_file": "file-DEF456",
8     "integrations": [
9       {
10         "type": "wandb",
```

```
11         "wandb": {
12             "project": "custom-wandb-project",
13             "tags": ["project:tag", "lineage"]
14         }
15     }
16 ]
17 }' https://api.openai.com/v1/fine\_tuning/jobs
```

By default, the Run ID and Run display name are the ID of your fine-tuning job (e.g. `ftjob-abc123`). You can customize the display name of the run by including a `"name"` field in the `wandb` object. You can also include a `"tags"` field in the `wandb` object to add tags to the W&B Run (tags must be ≤ 64 character strings and there is a maximum of 50 tags).

Sometimes it is convenient to explicitly set the [W&B Entity](#) to be associated with the run. You can do this by including an `"entity"` field in the `wandb` object. If you do not include an `"entity"` field, the W&B entity will default to the default W&B entity associated with the API key you registered previously.

The full specification for the integration can be found in our [fine-tuning job creation](#) documentation.

View your fine-tuning job in Weights and Biases

Once you've created a fine-tuning job with the W&B integration enabled, you can view the job in W&B by navigating to the W&B project you specified in the job creation request. Your run should be located at the URL: <https://wandb.ai/<WANDB-ENTITY>/<WANDB-PROJECT>/runs/ftjob-ABCDEF>.

You should see a new run with the name and tags you specified in the job creation request. The Run Config will contain relevant job metadata such as:

- `model` : The model you are fine-tuning
- `training_file` : The ID of the training file
- `validation_file` : The ID of the validation file
- `hyperparameters` : The hyperparameters used for the job (e.g. `n_epochs`, `learning_rate`, `batch_size`)
- `seed` : The random seed used for the job

Likewise, OpenAI will set some default tags on the run to make it easier for you to search and filter. These tags will be prefixed with `"openai/"` and will include:

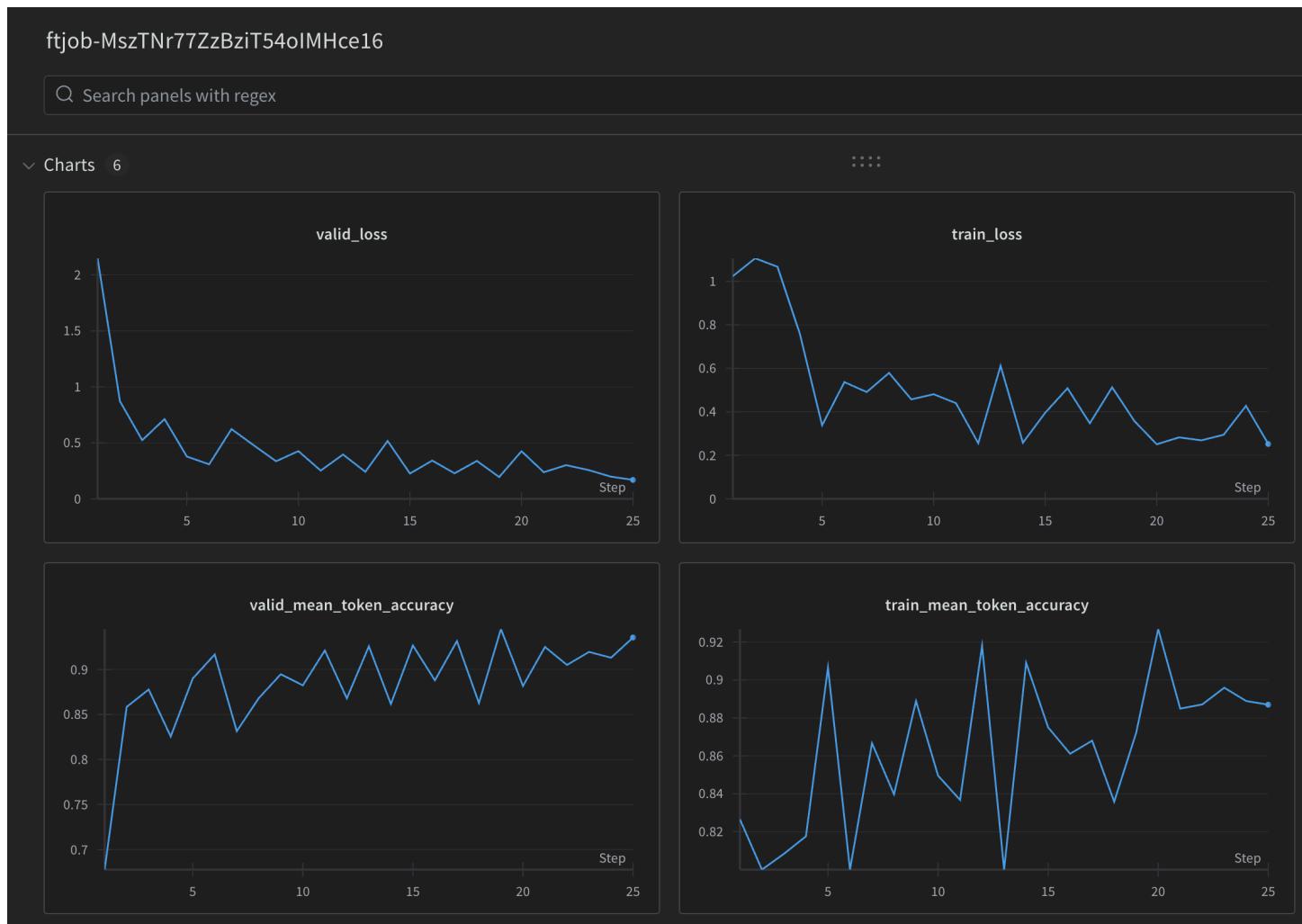
- `openai/fine-tuning` : Tag to let you know this run is a fine-tuning job
- `openai/ft-abc123` : The ID of the fine-tuning job `openai/gpt-3.5-turbo-0125` : The model you are fine-tuning

An example W&B run generated from an OpenAI fine-tuning job is shown below:

ftjob-MszTNr77ZzBziT54oIMHce16	
Description	What makes this run special?
Tags	first-project ✖ openai/fine-tuning ✖ openai/ftjob-MszTNr77ZzB... ✖ openai/gpt-3.5-turbo-0125 ✖
Author	 john-allard
State	⌚ Finished
Start time	April 4th, 2024 at 11:11:28 am
Duration	1m 49s

Metrics for each step of the fine-tuning job will be logged to the W&B run. These metrics are the same metrics provided in the [fine-tuning job event](#) object and are the same metrics your can view via the [OpenAI fine-tuning Dashboard](#). You can use W&B's visualization tools to track the progress of your fine-tuning job and compare it to other fine-tuning jobs you've run.

An example of the metrics logged to a W&B run is shown below:



FAQ

When should I use fine-tuning vs embeddings / retrieval augmented generation?

Embeddings with retrieval is best suited for cases when you need to have a large database of documents with relevant context and information.

By default OpenAI's models are trained to be helpful generalist assistants. Fine-tuning can be used to make a model which is narrowly focused, and exhibits specific ingrained behavior patterns. Retrieval strategies can be used to make new information available to a model by providing it with relevant context before generating its response. Retrieval strategies are not an alternative to fine-tuning and can in fact be complementary to it.

You can explore the differences between these options further in our Developer Day talk:

Can I fine-tune GPT-4 or GPT-3.5-Turbo-16k?

GPT-4 fine-tuning is in experimental access and eligible developers can request access via the [fine-tuning UI](#). Currently, `gpt-3.5-turbo-1106` and `gpt-3.5-turbo-0125` support up to 16K context examples.

How do I know if my fine-tuned model is actually better than the base model?

We recommend generating samples from both the base model and the fine-tuned model on a test set of chat conversations, and comparing the samples side by side. For more comprehensive evaluations, consider using the [OpenAI evals framework](#) to create an eval specific to your use case.

Can I continue fine-tuning a model that has already been fine-tuned?

Yes, you can pass the name of a fine-tuned model into the `model` parameter when creating a fine-tuning job. This will start a new fine-tuning job using the fine-tuned model as the starting point.

How can I estimate the cost of fine-tuning a model?

Please refer to the [estimate cost](#) section above.

Does the new fine-tuning endpoint still work with Weights & Biases for tracking metrics?

No, we do not currently support this integration but are working to enable it in the near future.

How many fine-tuning jobs can I have running at once?

Please refer to our [rate limit guide](#) for the most up to date information on the limits.

How do rate limits work on fine-tuned models?

A fine-tuned model pulls from the same shared rate limit as the model it is based off of. For example, if you use half your TPM rate limit in a given time period with the standard `gpt-3.5-turbo` model, any model(s) you fine-tuned from `gpt-3.5-turbo` would only have the remaining half of the TPM rate limit accessible since the capacity is shared across all models of the same type.

Put another way, having fine-tuned models does not give you more capacity to use our models from a total throughput perspective.

Can I use the `/v1/fine-tunes` endpoint?

The `/v1/fine-tunes` endpoint has been deprecated in favor of the `/v1/fine_tuning/jobs` endpoint.

For users migrating from `/v1/fine-tunes` to the updated `/v1/fine_tuning/jobs` API and newer models, the main difference you can expect is the updated API. The legacy prompt completion pair data format has been retained for the updated `babbage-002` and `davinci-002` models to ensure a smooth transition. The new models will support fine-tuning with 4k token context and have a knowledge cutoff of September 2021.

For most tasks, you should expect to get better performance from `gpt-3.5-turbo` than from the GPT base models.

Image generation

Learn how to generate or manipulate images with DALL-E in the API.

 Looking to generate images in ChatGPT? Head to [chatgpt.com](#).

Introduction

The Images API provides three methods for interacting with images:

- 1 Creating images from scratch based on a text prompt (DALL-E 3 and DALL-E 2)
- 2 Creating edited versions of images by having the model replace some areas of a pre-existing image, based on a new text prompt (DALL-E 2 only)
- 3 Creating variations of an existing image (DALL-E 2 only)

This guide covers the basics of using these three API endpoints with useful code samples. To try DALL-E 3, head to [ChatGPT](#). To try DALL-E 2, check out the [DALL-E preview app](#).

Usage

Generations

The [image generations](#) endpoint allows you to create an original image given a text prompt. When using DALL-E 3, images can have a size of 1024×1024, 1024×1792 or 1792×1024 pixels.

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](#).

```
Generate an image python ▾ Copy  
1 from openai import OpenAI  
2 client = OpenAI()  
3  
4 response = client.images.generate(  
5     model="dall-e-3",  
6     prompt="a white siamese cat",
```

```
7     size="1024x1024",
8     quality="standard",
9     n=1,
10 )
11
12 image_url = response.data[0].url
```



What is new with DALL-E 3

Explore what is new with DALL-E 3 in the OpenAI Cookbook

Prompting

With the release of DALL-E 3, the model now takes in the default prompt provided and automatically re-write it for safety reasons, and to add more detail (more detailed prompts generally result in higher quality images).

While it is not currently possible to disable this feature, you can use prompting to get outputs closer to your requested image by adding the following to your prompt: `I NEED to test how the tool works with extremely simple prompts. DO NOT add any detail, just use it AS-IS:`.

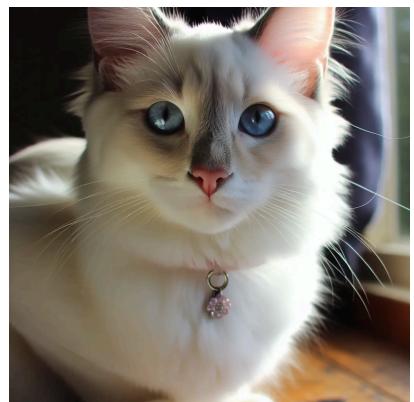
The updated prompt is visible in the `revised_prompt` field of the data response object.

Example DALL-E 3 generations

PROMPT

A photograph of a white Siamese cat.

GENERATION



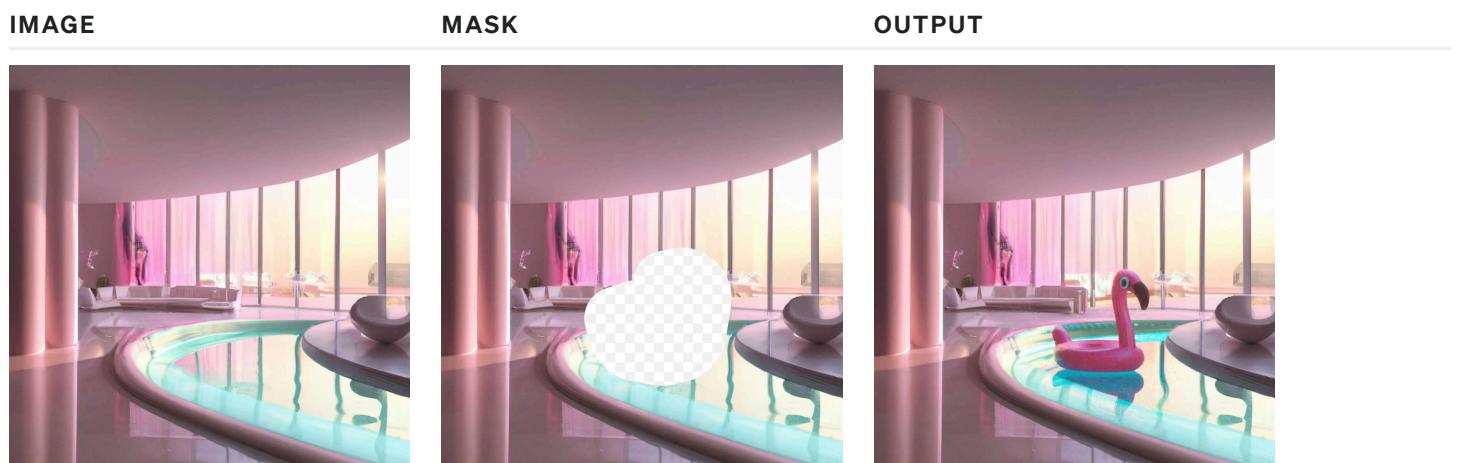
Each image can be returned as either a URL or Base64 data, using the `response_format` parameter. URLs will expire after an hour.

Edits (DALL-E 2 only)

Also known as "inpainting", 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**. This endpoint can enable experiences like [the editor in our DALL-E preview app](#).

```
Edit an image python ▾ 
```

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.images.edit(
5     model="dall-e-2",
6     image=open("sunlit_lounge.png", "rb"),
7     mask=open("mask.png", "rb"),
8     prompt="A sunlit indoor lounge area with a pool containing a flamingo",
9     n=1,
10    size="1024x1024"
11 )
12 image_url = response.data[0].url
```



Prompt: a sunlit indoor lounge area with a pool containing a flamingo

The uploaded image and mask must both be square PNG images less than 4MB in size, and also must have the same dimensions as each other. The non-transparent areas of the mask are not used when generating the output, so they don't necessarily need to match the original image like the example above.

Variations (DALL-E 2 only)

The [image variations](#) endpoint allows you to generate a variation of a given image.

Generate an image variation

python ▾



```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.images.create_variation(
5     model="dall-e-2",
6     image=open("corgi_and_cat_paw.png", "rb"),
7     n=1,
8     size="1024x1024"
9 )
10
11 image_url = response.data[0].url
```

IMAGE



OUTPUT



Similar to the edits endpoint, the input image must be a square PNG image less than 4MB in size.

Content moderation

Prompts and images are filtered based on our [content policy](#), returning an error when a prompt or image is flagged.

Language-specific tips

Node.js

Python

Using in-memory image data

The Node.js examples in the guide above use the `fs` module to read image data from disk. In some cases, you may have your image data in memory instead. Here's an example API call that uses image

data stored in a Node.js `Buffer` object:

```
1 import OpenAI from "openai";
2
3 const openai = new OpenAI();
4
5 // This is the Buffer object that contains your image data
6 const buffer = [your image data];
7
8 // Set a `name` that ends with .png so that the API knows it's a PNG image
9 buffer.name = "image.png";
10
11 async function main() {
12   const image = await openai.images.createVariation({ model: "dall-e-2", im
13   console.log(image.data);
14 }
15 main();
```

Working with TypeScript

If you're using TypeScript, you may encounter some quirks with image file arguments. Here's an example of working around the type mismatch by explicitly casting the argument:

```
1 import fs from "fs";
2 import OpenAI from "openai";
3
4 const openai = new OpenAI();
5
6 async function main() {
7   // Cast the ReadStream to `any` to appease the TypeScript compiler
8   const image = await openai.images.createVariation({
9     image: fs.createReadStream("image.png") as any,
10   });
11
12   console.log(image.data);
13 }
14 main();
```

And here's a similar example for in-memory image data:

```
1 import fs from "fs";
2 import OpenAI from "openai";
3
4 const openai = new OpenAI();
5
6 // This is the Buffer object that contains your image data
7 const buffer: Buffer = [your image data];
8
9 // Cast the buffer to `any` so that we can set the `name` property
10 const file: any = buffer;
11
12 // Set a `name` that ends with .png so that the API knows it's a PNG image
13 file.name = "image.png";
14
15 async function main() {
16   const image = await openai.images.createVariation({
17     file,
18     1,
19     "1024x1024"
20   });
21   console.log(image.data);
22 }
23 main();
```

Error handling

API requests can potentially return errors due to invalid inputs, rate limits, or other issues. These errors can be handled with a `try...catch` statement, and the error details can be found in either `error.response` or `error.message`:

```
1 import fs from "fs";
2 import OpenAI from "openai";
3
4 const openai = new OpenAI();
5
6 try {
```

```
7  const response = openai.images.createVariation(
8      fs.createReadStream("image.png"),
9      1,
10     "1024x1024"
11  );
12  console.log(response.data.data[0].url);
13 } catch (error) {
14     if (error.response) {
15         console.log(error.response.status);
16         console.log(error.response.data);
17     } else {
18         console.log(error.message);
19     }
20 }
```

Vision

Learn how to use GPT-4 to understand images

Introduction

GPT-4 Turbo with Vision allows the model to take in images and answer questions about them. Historically, language model systems have been limited by taking in a single input modality, text. For many use cases, this constrained the areas where models like GPT-4 could be used. Previously, the model has sometimes been referred to as [GPT-4V](#) or `gpt-4-vision-preview` in the API. Please note that the [Assistants API](#) does not currently support image inputs.

Quick start

Images are made available to the model in two main ways: by passing a link to the image or by passing the base64 encoded image directly in the request. Images can be passed in the `user`, `system` and `assistant` messages. Currently we don't support images in the first `system` message but this may change in the future.

```
What's in this image? python ▾   
1 from openai import OpenAI  
2  
3 client = OpenAI()  
4  
5 response = client.chat.completions.create(  
6     model="gpt-4-turbo",  
7     messages=[  
8         {  
9             "role": "user",  
10            "content": [  
11                {"type": "text", "text": "What's in this image?"},  
12                {  
13                    "type": "image_url",  
14                    "image_url": {  
15                        "url": "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisco"},  
16                    },  
17                },  
18            ],  
19        }  
20    ],  
21    max_tokens=300,  
22 )
```

```
23  
24 print(response.choices[0])
```

The model is best at answering general questions about what is present in the images. While it does understand the relationship between objects in images, it is not yet optimized to answer detailed questions about the location of certain objects in an image. For example, you can ask it what color a car is or what some ideas for dinner might be based on what is in your fridge, but if you show it an image of a room and ask it where the chair is, it may not answer the question correctly.

It is important to keep in mind the [limitations of the model](#) as you explore what use-cases visual understanding can be applied to.



Video understanding with vision

Learn how to use GPT-4 with Vision to understand videos in the OpenAI Cookbook

Uploading base 64 encoded images

If you have an image or set of images locally, you can pass those to the model in base 64 encoded format, here is an example of this in action:

```
1 import base64  
2 import requests  
3  
4 # OpenAI API Key  
5 api_key = "YOUR_OPENAI_API_KEY"  
6  
7 # Function to encode the image  
8 def encode_image(image_path):  
9     with open(image_path, "rb") as image_file:  
10         return base64.b64encode(image_file.read()).decode('utf-8')  
11  
12 # Path to your image  
13 image_path = "path_to_your_image.jpg"  
14  
15 # Getting the base64 string  
16 base64_image = encode_image(image_path)  
17  
18 headers = {  
19     "Content-Type": "application/json",
```

```
20     "Authorization": f"Bearer {api_key}"
21 }
22
23 payload = {
24     "model": "gpt-4-turbo",
25     "messages": [
26         {
27             "role": "user",
28             "content": [
29                 {
30                     "type": "text",
31                     "text": "What's in this image?"
32                 },
33                 {
34                     "type": "image_url",
35                     "image_url": {
36                         "url": f"data:image/jpeg;base64,{base64_image}"
37                     }
38                 }
39             ]
40         }
41     ],
42     "max_tokens": 300
43 }
44
45 response = requests.post("https://api.openai.com/v1/chat/completions", head
46
47 print(response.json())
```

Multiple image inputs

The Chat Completions API is capable of taking in and processing multiple image inputs in both base64 encoded format or as an image URL. The model will process each image and use the information from all of them to answer the question.

Multiple image inputs

python ▾



```
1 from openai import OpenAI
2
```

```

3 client = OpenAI()
4 response = client.chat.completions.create(
5     model="gpt-4-turbo",
6     messages=[
7         {
8             "role": "user",
9             "content": [
10                 {
11                     "type": "text",
12                     "text": "What are in these images? Is there any difference between them?",
13                 },
14                 {
15                     "type": "image_url",
16                     "image_url": {
17                         "url": "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisco
18                         },
19                     },
20                 {
21                     "type": "image_url",
22                     "image_url": {
23                         "url": "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisco
24                         },
25                     },
26                 ],
27             }
28         ],
29         max_tokens=300,
30     )
31 print(response.choices[0])

```

Here the model is shown two copies of the same image and can answer questions about both or each



Low or high fidelity image understanding

By controlling the `detail` parameter, which has three options, `low`, `high`, or `auto`, you have control over how the model processes the image and generates its textual understanding. By default, the model will use the `auto` setting which will look at the image input size and decide if it should use the `low` or `high` setting.

- `low` will enable the "low res" mode. The model will receive a low-res 512px x 512px version of the image, and represent the image with a budget of 65 tokens. This allows the API to return faster responses and consume fewer input tokens for use cases that do not require high detail.

- `high` will enable "high res" mode, which first allows the model to see the low res image and then creates detailed crops of input images as 512px squares based on the input image size. Each of the detailed crops uses twice the token budget (65 tokens) for a total of 129 tokens.

Choosing the detail level

python ▾ 

```

1 from openai import OpenAI
2
3 client = OpenAI()
4
5 response = client.chat.completions.create(
6     model="gpt-4-turbo",
7     messages=[
8         {
9             "role": "user",
10            "content": [
11                {"type": "text", "text": "What's in this image?"},
12                {
13                    "type": "image_url",
14                    "image_url": {
15                        "url": "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisco",
16                        "detail": "high"
17                    },
18                },
19            ],
20        }
21    ],
22    max_tokens=300,
23 )
24
25 print(response.choices[0].message.content)

```

Managing images

The Chat Completions API, unlike the Assistants API, is not stateful. That means you have to manage the messages (including images) you pass to the model yourself. If you want to pass the same image to the model multiple times, you will have to pass the image each time you make a request to the API.

For long running conversations, we suggest passing images via URL's instead of base64. The latency of the model can also be improved by downsizing your images ahead of time to be less than the maximum size they are expected them to be. For low res mode, we expect a 512px x 512px image. For high res mode, the short side of the image should be less than 768px and the long side should be less than 2,000px.

After an image has been processed by the model, it is deleted from OpenAI servers and not retained. [We do not use data uploaded via the OpenAI API to train our models.](#)

Limitations

While GPT-4 with vision is powerful and can be used in many situations, it is important to understand the limitations of the model. Here are some of the limitations we are aware of:

- Medical images: The model is not suitable for interpreting specialized medical images like CT scans and shouldn't be used for medical advice.
- Non-English: The model may not perform optimally when handling images with text of non-Latin alphabets, such as Japanese or Korean.
- Small text: Enlarge text within the image to improve readability, but avoid cropping important details.
- Rotation: The model may misinterpret rotated / upside-down text or images.
- Visual elements: The model may struggle to understand graphs or text where colors or styles like solid, dashed, or dotted lines vary.
- Spatial reasoning: The model struggles with tasks requiring precise spatial localization, such as identifying chess positions.
- Accuracy: The model may generate incorrect descriptions or captions in certain scenarios.
- Image shape: The model struggles with panoramic and fisheye images.
- Metadata and resizing: The model doesn't process original file names or metadata, and images are resized before analysis, affecting their original dimensions.
- Counting: May give approximate counts for objects in images.
- CAPTCHAS: For safety reasons, we have implemented a system to block the submission of CAPTCHAs.

Calculating costs

Image inputs are metered and charged in tokens, just as text inputs are. The token cost of a given image is determined by two factors: its size, and the `detail` option on each `image_url` block. All images with `detail: low` cost 85 tokens each. `detail: high` images are first scaled to fit within a 2048×2048 square, maintaining their aspect ratio. Then, they are scaled such that the shortest side of the image is 768px long. Finally, we count how many 512px squares the image consists of. Each of those squares costs **170 tokens**. Another **85 tokens** are always added to the final total.

Here are some examples demonstrating the above.

- A 1024×1024 square image in `detail: high` mode costs 765 tokens

- 1024 is less than 2048, so there is no initial resize.
- The shortest side is 1024, so we scale the image down to 768×768 .
- 4 512px square tiles are needed to represent the image, so the final token cost is $170 * 4 + 85 = 765$.
- A 2048×4096 image in `detail: high` mode costs 1105 tokens
 - We scale down the image to 1024×2048 to fit within the 2048 square.
 - The shortest side is 1024, so we further scale down to 768×1536 .
 - 6 512px tiles are needed, so the final token cost is $170 * 6 + 85 = 1105$.
- A 4096×8192 image in `detail: low` mode costs 85 tokens
 - Regardless of input size, low detail images are a fixed cost.

FAQ

Can I fine-tune the image capabilities in gpt-4?

No, we do not support fine-tuning the image capabilities of `gpt-4` at this time.

Can I use gpt-4 to generate images?

No, you can use `dall-e-3` to generate images and `gpt-4-turbo` to understand images.

What type of files can I upload?

We currently support PNG (.png), JPEG (.jpeg and .jpg), WEBP (.webp), and non-animated GIF (.gif).

Is there a limit to the size of the image I can upload?

Yes, we restrict image uploads to 20MB per image.

Can I delete an image I uploaded?

No, we will delete the image for you automatically after it has been processed by the model.

Where can I learn more about the considerations of GPT-4 with Vision?

You can find details about our evaluations, preparation, and mitigation work in the [GPT-4 with Vision system card](#).

We have further implemented a system to block the submission of CAPTCHAs.

How do rate limits for GPT-4 with Vision work?

We process images at the token level, so each image we process counts towards your tokens per minute (TPM) limit. See the calculating costs section for details on the formula used to determine token count per image.

Can GPT-4 with Vision understand image metadata?

No, the model does not receive image metadata.

What happens if my image is unclear?

If an image is ambiguous or unclear, the model will do its best to interpret it. However, the results may be less accurate. A good rule of thumb is that if an average human cannot see the info in an image at the resolutions used in low/high res mode, then the model cannot either.

Text to speech

Learn how to turn text into lifelike spoken audio

Introduction

The Audio API provides a `speech` endpoint based on our **TTS (text-to-speech) model**. It comes with 6 built-in voices and can be used to:

- Narrate a written blog post
- Produce spoken audio in multiple languages
- Give real time audio output using streaming

Here is an example of the `alloy` voice:

 Please note that our **usage policies** require you to provide a clear disclosure to end users that the TTS voice they are hearing is AI-generated and not a human voice.

Quick start

The `speech` endpoint takes in three key inputs: the `model`, the `text` that should be turned into audio, and the `voice` to be used for the audio generation. A simple request would look like the following:

```
Generate spoken audio from input text
python ▾   
  
1 from pathlib import Path
2 from openai import OpenAI
3 client = OpenAI()
4
5 speech_file_path = Path(__file__).parent / "speech.mp3"
6 response = client.audio.speech.create(
7     model="tts-1",
8     voice="alloy",
9     input="Today is a wonderful day to build something people love!"
10 )
11
12 response.stream_to_file(speech_file_path)
```

By default, the endpoint will output a MP3 file of the spoken audio but it can also be configured to output any of our **supported formats**.

Audio quality

For real-time applications, the standard `tts-1` model provides the lowest latency but at a lower quality than the `tts-1-hd` model. Due to the way the audio is generated, `tts-1` is likely to generate content that has more static in certain situations than `tts-1-hd`. In some cases, the audio may not have noticeable differences depending on your listening device and the individual person.

Voice options

Experiment with different voices (`alloy`, `echo`, `fable`, `onyx`, `nova`, and `shimmer`) to find one that matches your desired tone and audience. The current voices are optimized for English.

Alloy



Supported output formats

The default response format is "mp3", but other formats like "opus", "aac", "flac", and "pcm" are available.

- **Opus:** For internet streaming and communication, low latency.
- **AAC:** For digital audio compression, preferred by YouTube, Android, iOS.
- **FLAC:** For lossless audio compression, favored by audio enthusiasts for archiving.
- **WAV:** Uncompressed WAV audio, suitable for low-latency applications to avoid decoding overhead.
- **PCM:** Similar to WAV but containing the raw samples in 24kHz (16-bit signed, low-endian), without the header.

Supported languages

The TTS model generally follows the Whisper model in terms of language support. Whisper [supports the following languages](#) and performs well despite the current voices being optimized for English:

Afrikaans, Arabic, Armenian, Azerbaijani, Belarusian, Bosnian, Bulgarian, Catalan, Chinese, Croatian, Czech, Danish, Dutch, English, Estonian, Finnish, French, Galician, German, Greek, Hebrew, Hindi, Hungarian, Icelandic, Indonesian, Italian, Japanese, Kannada, Kazakh, Korean, Latvian, Lithuanian, Macedonian, Malay, Marathi, Maori, Nepali, Norwegian, Persian, Polish, Portuguese, Romanian, Russian, Serbian, Slovak, Slovenian, Spanish, Swahili, Swedish, Tagalog, Tamil, Thai, Turkish, Ukrainian, Urdu, Vietnamese, and Welsh.

You can generate spoken audio in these languages by providing the input text in the language of your choice.

Streaming real time audio

The Speech API provides support for real time audio streaming using [chunk transfer encoding](#). This means that the audio is able to be played before the full file has been generated and made accessible.

```
1 from openai import OpenAI
2
3 client = OpenAI()
4
5 response = client.audio.speech.create(
6     model="tts-1",
7     voice="alloy",
8     input="Hello world! This is a streaming test.",
9 )
10
11 response.stream_to_file("output.mp3")
```

FAQ

How can I control the emotional range of the generated audio?

There is no direct mechanism to control the emotional output of the audio generated. Certain factors may influence the output audio like capitalization or grammar but our internal tests with these have yielded mixed results.

Can I create a custom copy of my own voice?

No, this is not something we support.

Do I own the outputted audio files?

Yes, like with all outputs from our API, the person who created them owns the output. You are still required to inform end users that they are hearing audio generated by AI and not a real person talking to them.

Speech to text

Learn how to turn audio into text

Introduction

The Audio API provides two speech to text endpoints, `transcriptions` and `translations`, based on our state-of-the-art open source large-v2 [Whisper model](#). They can be used to:

- Transcribe audio into whatever language the audio is in.
- Translate and transcribe the audio into english.

File uploads are currently limited to 25 MB and the following input file types are supported: `mp3`, `mp4`, `mpeg`, `mpga`, `m4a`, `wav`, and `webm`.

Quickstart

Transcriptions

The transcriptions API takes as input the audio file you want to transcribe and the desired output file format for the transcription of the audio. We currently support multiple input and output file formats.

```
Transcribe audio                                         python ▾   ⌂
1 from openai import OpenAI
2 client = OpenAI()
3
4 audio_file= open("/path/to/file/audio.mp3", "rb")
5 transcription = client.audio.transcriptions.create(
6     model="whisper-1",
7     file=audio_file
8 )
9 print(transcription.text)
```

By default, the response type will be json with the raw text included.

```
{
  "text": "Imagine the wildest idea that you've ever had, and you're curious
about how it might scale to something that's a 100, a 1,000 times bigger."
```

```
....
```

```
}
```

The Audio API also allows you to set additional parameters in a request. For example, if you want to set the `response_format` as `text`, your request would look like the following:

Additional options

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 audio_file = open("/path/to/file/speech.mp3", "rb")
5 transcription = client.audio.transcriptions.create(
6     model="whisper-1",
7     file=audio_file,
8     response_format="text"
9 )
10 print(transcription.text)
```

The [API Reference](#) includes the full list of available parameters.

Translations

The translations API takes as input the audio file in any of the supported languages and transcribes, if necessary, the audio into English. This differs from our /Transcriptions endpoint since the output is not in the original input language and is instead translated to English text.

Translate audio

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 audio_file= open("/path/to/file/german.mp3", "rb")
5 translation = client.audio.translations.create(
6     model="whisper-1",
7     file=audio_file
8 )
9 print(translation.text)
```

In this case, the inputted audio was german and the outputted text looks like:

Hello, my name is Wolfgang and I come from Germany. Where are you heading today?

We only support translation into English at this time.

Supported languages

We currently [support the following languages](#) through both the `transcriptions` and `translations` endpoint:

Afrikaans, Arabic, Armenian, Azerbaijani, Belarusian, Bosnian, Bulgarian, Catalan, Chinese, Croatian, Czech, Danish, Dutch, English, Estonian, Finnish, French, Galician, German, Greek, Hebrew, Hindi, Hungarian, Icelandic, Indonesian, Italian, Japanese, Kannada, Kazakh, Korean, Latvian, Lithuanian, Macedonian, Malay, Marathi, Maori, Nepali, Norwegian, Persian, Polish, Portuguese, Romanian, Russian, Serbian, Slovak, Slovenian, Spanish, Swahili, Swedish, Tagalog, Tamil, Thai, Turkish, Ukrainian, Urdu, Vietnamese, and Welsh.

While the underlying model was trained on 98 languages, we only list the languages that exceeded <50% [word error rate](#) (WER) which is an industry standard benchmark for speech to text model accuracy. The model will return results for languages not listed above but the quality will be low.

Timestamps

By default, the Whisper API will output a transcript of the provided audio in text. The `timestamp_granularities[]` parameter enables a more structured and timestamped json output format, with timestamps at the segment, word level, or both. This enables word-level precision for transcripts and video edits, which allows for the removal of specific frames tied to individual words.

Timestamp options

python ▾ 

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 audio_file = open("speech.mp3", "rb")
5 transcript = client.audio.transcriptions.create(
6     file=audio_file,
7     model="whisper-1",
8     response_format="verbose_json",
9     timestamp_granularities=["word"]
10 )
```

```
11  
12 print(transcript.words)
```

Longer inputs

By default, the Whisper API only supports files that are less than 25 MB. If you have an audio file that is longer than that, you will need to break it up into chunks of 25 MB's or less or used a compressed audio format. To get the best performance, we suggest that you avoid breaking the audio up mid-sentence as this may cause some context to be lost.

One way to handle this is to use the [PyDub open source Python package](#) to split the audio:

```
1 from pydub import AudioSegment  
2  
3 song = AudioSegment.from_mp3("good_morning.mp3")  
4  
5 # PyDub handles time in milliseconds  
6 ten_minutes = 10 * 60 * 1000  
7  
8 first_10_minutes = song[:ten_minutes]  
9  
10 first_10_minutes.export("good_morning_10.mp3", format="mp3")
```

OpenAI makes no guarantees about the usability or security of 3rd party software like PyDub.

Prompting

You can use a [prompt](#) to improve the quality of the transcripts generated by the Whisper API. The model will try to match the style of the prompt, so it will be more likely to use capitalization and punctuation if the prompt does too. However, the current prompting system is much more limited than our other language models and only provides limited control over the generated audio. Here are some examples of how prompting can help in different scenarios:

- 1 Prompts can be very helpful for correcting specific words or acronyms that the model may misrecognize in the audio. For example, the following prompt improves the transcription of the words DALL-E and GPT-3, which were previously written as "GDP 3" and "DALI": "The transcript is about OpenAI which makes technology like DALL-E, GPT-3, and ChatGPT with the hope of one day building an AGI system that benefits all of humanity"
- 2 To preserve the context of a file that was split into segments, you can prompt the model with the transcript of the preceding segment. This will make the transcript more accurate, as the model will use the relevant information from the previous audio. The model will only consider the final

224 tokens of the prompt and ignore anything earlier. For multilingual inputs, Whisper uses a custom tokenizer. For English only inputs, it uses the standard GPT-2 tokenizer which are both accessible through the open source [Whisper Python package](#).

- 3 Sometimes the model might skip punctuation in the transcript. You can avoid this by using a simple prompt that includes punctuation: "Hello, welcome to my lecture."
- 4 The model may also leave out common filler words in the audio. If you want to keep the filler words in your transcript, you can use a prompt that contains them: "Umm, let me think like, hmm... Okay, here's what I'm, like, thinking."
- 5 Some languages can be written in different ways, such as simplified or traditional Chinese. The model might not always use the writing style that you want for your transcript by default. You can improve this by using a prompt in your preferred writing style.

Improving reliability

As we explored in the prompting section, one of the most common challenges faced when using Whisper is the model often does not recognize uncommon words or acronyms. To address this, we have highlighted different techniques which improve the reliability of Whisper in these cases:

⟨ Using the prompt parameter

The first method involves using the optional prompt parameter to pass a dictionary of the correct spellings.

Since it wasn't trained using instruction-following techniques, Whisper operates more like a base GPT model. It's important to keep in mind that Whisper only considers the first 244 tokens of the prompt.

```
Prompt parameter python ▾

1 from openai import OpenAI
2 client = OpenAI()
3
4 audio_file = open("/path/to/file/speech.mp3", "rb")
5 transcription = client.audio.transcriptions.create(
6     model="whisper-1",
7     file=audio_file,
8     response_format="text",
9     prompt="ZyntriQix, Digique Plus, CynapseFive, VortiQore V8, EchoNix Array, OrbitalLin
10 )
11 print(transcription.text)
```

While it will increase reliability, this technique is limited to only 244 characters so your list of SKUs would need to be relatively small in order for this to be a scalable solution.

◀ Post-processing with GPT-4

The second method involves a post-processing step using GPT-4 or GPT-3.5-Turbo.

We start by providing instructions for GPT-4 through the `system_prompt` variable. Similar to what we did with the `prompt` parameter earlier, we can define our company and product names.



The screenshot shows a Jupyter Notebook cell with the following code:

```
Post-processing python ▾
1 system_prompt = "You are a helpful assistant for the company ZyntriQix. Your task is to"
2
3 def generate_corrected_transcript(temperature, system_prompt, audio_file):
4     response = client.chat.completions.create(
5         model="gpt-4-turbo",
6         temperature=temperature,
7         messages=[
8             {
9                 "role": "system",
10                "content": system_prompt
11            },
12            {
13                "role": "user",
14                "content": transcribe(audio_file, "")
15            }
16        ]
17    )
18    return completion.choices[0].message.content
19
20 corrected_text = generate_corrected_transcript(0, system_prompt, fake_company_filepath)
```

The code defines a function `generate_corrected_transcript` that takes `temperature`, `system_prompt`, and `audio_file` as parameters. It uses the `client.chat.completions.create` method to generate a completion, specifying the `model` as `"gpt-4-turbo"`. The `messages` parameter contains two messages: one from the system with the `system_prompt` and one from the user with the transcribed audio file. The function returns the content of the first choice in the completion. Finally, it calls the function with `temperature=0`, `system_prompt`, and `fake_company_filepath`.

If you try this on your own audio file, you can see that GPT-4 manages to correct many misspellings in the transcript. Due to its larger context window, this method might be more scalable than using Whisper's `prompt` parameter and is more reliable since GPT-4 can be instructed and guided in ways that aren't possible with Whisper given the lack of instruction following.

Moderation

Learn how to build moderation into your AI applications.

Overview

The [moderations](#) endpoint is a tool you can use to check whether text is potentially harmful. Developers can use it to identify content that might be harmful and take action, for instance by filtering it.

The models classifies the following categories:

CATEGORY	DESCRIPTION
hate	Content that expresses, incites, or promotes hate based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status, or caste. Hateful content aimed at non-protected groups (e.g., chess players) is harassment.
hate/threatening	Hateful content that also includes violence or serious harm towards the targeted group based on race, gender, ethnicity, religion, nationality, sexual orientation, disability status, or caste.
harassment	Content that expresses, incites, or promotes harassing language towards any target.
harassment/threatening	Harassment content that also includes violence or serious harm towards any target.
self-harm	Content that promotes, encourages, or depicts acts of self-harm, such as suicide, cutting, and eating disorders.
self-harm/intent	Content where the speaker expresses that they are engaging or intend to engage in acts of self-harm, such as suicide, cutting, and eating disorders.
self-harm/instructions	Content that encourages performing acts of self-harm, such as suicide, cutting, and eating disorders, or that gives instructions or advice on how to commit such acts.
sexual	Content meant to arouse sexual excitement, such as the description of sexual activity, or that promotes sexual services (excluding sex education and wellness).
sexual/minors	Sexual content that includes an individual who is under 18 years old.

CATEGORY	DESCRIPTION
violence	Content that depicts death, violence, or physical injury.
violence/graphic	Content that depicts death, violence, or physical injury in graphic detail.

The moderation endpoint is free to use for most developers. For higher accuracy, try splitting long pieces of text into smaller chunks each less than 2,000 characters.

-  We are continuously working to improve the accuracy of our classifier. Our support for non-English languages is currently limited.

Quickstart

To obtain a classification for a piece of text, make a request to the [moderation endpoint](#) as demonstrated in the following code snippets:

Example: Getting moderations

curl ▾



```
1 curl https://api.openai.com/v1/moderations \
2   -X POST \
3   -H "Content-Type: application/json" \
4   -H "Authorization: Bearer $OPENAI_API_KEY" \
5   -d '{"input": "Sample text goes here"}'
```

Below is an example output of the endpoint. It returns the following fields:

- `flagged` : Set to `true` if the model classifies the content as potentially harmful, `false` otherwise.
- `categories` : Contains a dictionary of per-category violation flags. For each category, the value is `true` if the model flags the corresponding category as violated, `false` otherwise.
- `category_scores` : Contains a dictionary of per-category raw scores output by the model, denoting the model's confidence that the input violates the OpenAI's policy for the category. The value is between 0 and 1, where higher values denote higher confidence. The scores should not be interpreted as probabilities.

```
1 {
2     "id": "modr-XXXXXX",
3     "model": "text-moderation-007",
4     "results": [
```



```
5      {
6          "flagged": true,
7          "categories": {
8              "sexual": false,
9              "hate": false,
10             "harassment": false,
11             "self-harm": false,
12             "sexual/minors": false,
13             "hate/threatening": false,
14             "violence/graphic": false,
15             "self-harm/intent": false,
16             "self-harm/instructions": false,
17             "harassment/threatening": true,
18             "violence": true
19         },
20         "category_scores": {
21             "sexual": 1.2282071e-6,
22             "hate": 0.010696256,
23             "harassment": 0.29842457,
24             "self-harm": 1.5236925e-8,
25             "sexual/minors": 5.7246268e-8,
26             "hate/threatening": 0.0060676364,
27             "violence/graphic": 4.435014e-6,
28             "self-harm/intent": 8.098441e-10,
29             "self-harm/instructions": 2.8498655e-11,
30             "harassment/threatening": 0.63055265,
31             "violence": 0.99011886
32         }
33     }
34 ]
35 }
```

- ⓘ We plan to continuously upgrade the moderation endpoint's underlying model. Therefore, custom policies that rely on `category_scores` may need recalibration over time.

Assistants API

Beta

The Assistants API allows you to build AI assistants within your own applications. An Assistant has instructions and can leverage models, tools, and files to respond to user queries. The Assistants API currently supports three types of tools: Code Interpreter, File Search, and Function calling.

You can explore the capabilities of the Assistants API using the [Assistants playground](#) or by building a step-by-step integration outlined in this guide.

Overview

A typical integration of the Assistants API has the following flow:

- 1 Create an [Assistant](#) by defining its custom instructions and picking a model. If helpful, add files and enable tools like Code Interpreter, File Search, and Function calling.
- 2 Create a [Thread](#) when a user starts a conversation.
- 3 Add [Messages](#) to the Thread as the user asks questions.
- 4 [Run](#) the Assistant on the Thread to generate a response by calling the model and the tools.

This starter guide walks through the key steps to create and run an Assistant that uses [Code Interpreter](#). In this example, we're [creating an Assistant](#) that is a personal math tutor, with the Code Interpreter tool enabled.

- ⓘ Calls to the Assistants API require that you pass a beta HTTP header. This is handled automatically if you're using OpenAI's official Python or Node.js SDKs.

OpenAI-Beta: assistants=v2



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`.

Create an Assistant

python ▾



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

```
3
4 assistant = client.beta.assistants.create(
5     name="Math Tutor",
6     instructions="You are a personal math tutor. Write and run code to answer math questions",
7     tools=[{"type": "code_interpreter"}],
8     model="gpt-4-turbo",
9 )
```

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.

Create a Thread

python ▾

```
thread = client.beta.threads.create()
```

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.

Add a Message to the Thread

python ▾

```
1 message = client.beta.threads.messages.create(
2     thread_id=thread.id,
3     role="user",
4     content="I need to solve the equation `3x + 11 = 14`. Can you help me?"
5 )
```

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.

With streaming

Without streaming

You can use the 'create and stream' helpers in the Python and Node SDKs to create a run and stream the response.

Create and Stream a Run

python ▾



```
1 from typing_extensions import override
2 from openai import AssistantEventHandler
3
4 # First, we create a EventHandler class to define
5 # how we want to handle the events in the response stream.
6
7 class EventHandler(AssistantEventHandler):
8     @override
9     def on_text_created(self, text) -> None:
10         print(f"\nassistant > ", end="", flush=True)
11
12     @override
13     def on_text_delta(self, delta, snapshot):
14         print(delta.value, end="", flush=True)
15
16     def on_tool_call_created(self, tool_call):
17         print(f"\nassistant > {tool_call.type}\n", flush=True)
18
19     def on_tool_call_delta(self, delta, snapshot):
20         if delta.type == 'code_interpreter':
21             if delta.code_interpreter.input:
22                 print(delta.code_interpreter.input, end="", flush=True)
23             if delta.code_interpreter.outputs:
24                 print("\n\noutput >", flush=True)
25                 for output in delta.code_interpreter.outputs:
26                     if output.type == "logs":
27                         print(f"\n{output.logs}", flush=True)
28
29 # Then, we use the `stream` SDK helper
30 # with the `EventHandler` class to create the Run
31 # and stream the response.
32
33 with client.beta.threads.runs.stream(
34     thread_id=thread.id,
35     assistant_id=assistant.id,
36     instructions="Please address the user as Jane Doe. The user has a premium account.",
37     event_handler=EventHandler(),
38 ) as stream:
39     stream.until_done()
```

See the full list of Assistants streaming events in our API reference [here](#). You can also see a list of SDK event listeners for these events in the [Python & Node](#) repository documentation.

Next

- 1 Dive deeper into [How Assistants work](#)
- 2 Learn more about [Tools](#)
- 3 Explore the [Assistants playground](#)

How Assistants work

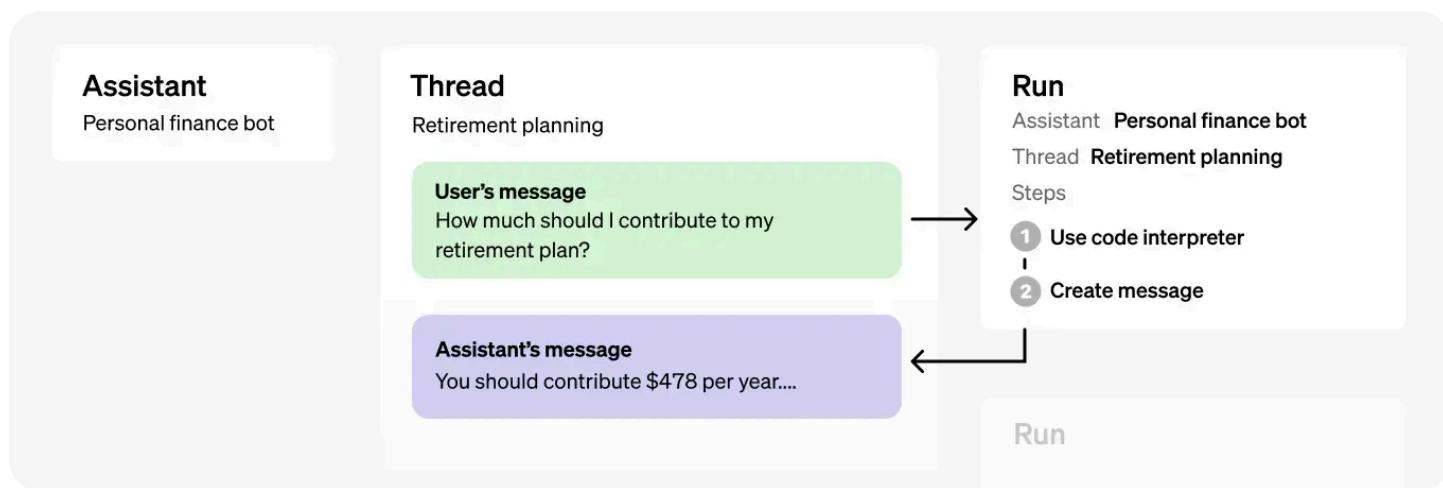
Beta

The Assistants API is designed to help developers build powerful AI assistants capable of performing a variety of tasks.

The Assistants API is in **beta** and we are actively working on adding more functionality. Share your feedback in our [Developer Forum](#)!

- 1 Assistants can call OpenAI's **models** with specific instructions to tune their personality and capabilities.
- 2 Assistants can access **multiple tools in parallel**. These can be both OpenAI-hosted tools — like `code_interpreter` and `file_search` — or tools you build / host (via **function calling**).
- 3 Assistants can access **persistent Threads**. Threads simplify AI application development by storing message history and truncating it when the conversation gets too long for the model's context length. You create a Thread once, and simply append Messages to it as your users reply.
- 4 Assistants can access files in several formats — either as part of their creation or as part of Threads between Assistants and users. When using tools, Assistants can also create files (e.g., images, spreadsheets, etc) and cite files they reference in the Messages they create.

Objects



OBJECT WHAT IT REPRESENTS

Assistant	Purpose-built AI that uses OpenAI's models and calls tools
Thread	A conversation session between an Assistant and a user. Threads store Messages and automatically handle truncation to fit content into a model's context.
Message	A message created by an Assistant or a user. Messages can include text, images, and other files. Messages stored as a list on the Thread.

OBJECT	WHAT IT REPRESENTS
Run	An invocation of an Assistant on a Thread. The Assistant uses its configuration and the Thread's Messages to perform tasks by calling models and tools. As part of a Run, the Assistant appends Messages to the Thread.
Run Step	A detailed list of steps the Assistant took as part of a Run. An Assistant can call tools or create Messages during its run. Examining Run Steps allows you to introspect how the Assistant is getting to its final results.

Creating Assistants

 We recommend using OpenAI's [latest models](#) with the Assistants API for best results and maximum compatibility with tools.

To get started, creating an Assistant only requires specifying the `model` to use. But you can further customize the behavior of the Assistant:

- 1 Use the `instructions` parameter to guide the personality of the Assistant and define its goals. Instructions are similar to system messages in the Chat Completions API.
- 2 Use the `tools` parameter to give the Assistant access to up to 128 tools. You can give it access to OpenAI-hosted tools like `code_interpreter` and `file_search`, or call a third-party tools via a `function` calling.
- 3 Use the `tool_resources` parameter to give the tools like `code_interpreter` and `file_search` access to files. Files are uploaded using the `File upload endpoint` and must have the `purpose` set to `assistants` to be used with this API.

For example, to create an Assistant that can create data visualization based on a `.csv` file, first upload a file.

```
python ▾   
1 file = client.files.create(  
2     file=open("revenue-forecast.csv", "rb"),  
3     purpose='assistants'  
4 )
```

Then, create the Assistant with the `code_interpreter` tool enabled and provide the file as a resource to the tool.

python ▾

```
1 assistant = client.beta.assistants.create(
2     name="Data visualizer",
3     description="You are great at creating beautiful data visualizations. You analyze dat",
4     model="gpt-4-turbo",
5     tools=[{"type": "code_interpreter"}],
6     tool_resources={
7         "code_interpreter": {
8             "file_ids": [file.id]
9         }
10    }
11 )
```

You can attach a maximum of 20 files to `code_interpreter` and 10,000 files to `file_search` (using `vector_store objects`).

Each file can be at most 512 MB in size and have a maximum of 5,000,000 tokens. By default, the size of all the files uploaded by your organization cannot exceed 100 GB, but you can reach out to our support team to increase this limit.

Managing Threads and Messages

Threads and Messages represent a conversation session between an Assistant and a user. There is no limit to the number of Messages you can store in a Thread. Once the size of the Messages exceeds the context window of the model, the Thread will attempt to smartly truncate messages, before fully dropping the ones it considers the least important.

You can create a Thread with an initial list of Messages like this:

python ▾

```
1 thread = client.beta.threads.create(
2     messages=[
3         {
4             "role": "user",
5             "content": "Create 3 data visualizations based on the trends in this file.",
6             "attachments": [
7                 {
8                     "file_id": file.id,
9                     "tools": [{"type": "code_interpreter"}]
10                }
11            ]
12        }
13    ]
14 )
```

```
11      ]
12    }
13  ]
14 )
```

Messages can contain text, images, or file attachment. Message `attachments` are helper methods that add files to a thread's `tool_resources`. You can also choose to add files to the `thread.tool_resources` directly. At the moment, user-created Messages cannot contain image files but we plan to add support for this in the future.

Context window management

The Assistants API automatically manages the truncation to ensure it stays within the model's maximum context length. You can customize this behavior by specifying the maximum tokens you'd like a run to utilize and/or the maximum number of recent messages you'd like to include in a run.

Max Completion and Max Prompt Tokens

To control the token usage in a single Run, set `max_prompt_tokens` and `max_completion_tokens` when creating the Run. These limits apply to the total number of tokens used in all completions throughout the Run's lifecycle.

For example, initiating a Run with `max_prompt_tokens` set to 500 and `max_completion_tokens` set to 1000 means the first completion will truncate the thread to 500 tokens and cap the output at 1000 tokens. If only 200 prompt tokens and 300 completion tokens are used in the first completion, the second completion will have available limits of 300 prompt tokens and 700 completion tokens.

If a completion reaches the `max_completion_tokens` limit, the Run will terminate with a status of `incomplete`, and details will be provided in the `incomplete_details` field of the Run object.

- ⓘ When using the File Search tool, we recommend setting the `max_prompt_tokens` to no less than 20,000. For longer conversations or multiple interactions with File Search, consider increasing this limit to 50,000, or ideally, removing the `max_prompt_tokens` limits altogether to get the highest quality results.

Truncation Strategy

You may also specify a truncation strategy to control how your thread should be rendered into the model's context window. Using a truncation strategy of type `auto` will use OpenAI's default truncation strategy. Using a truncation strategy of type `last_messages` will allow you to specify the number of the most recent messages to include in the context window.

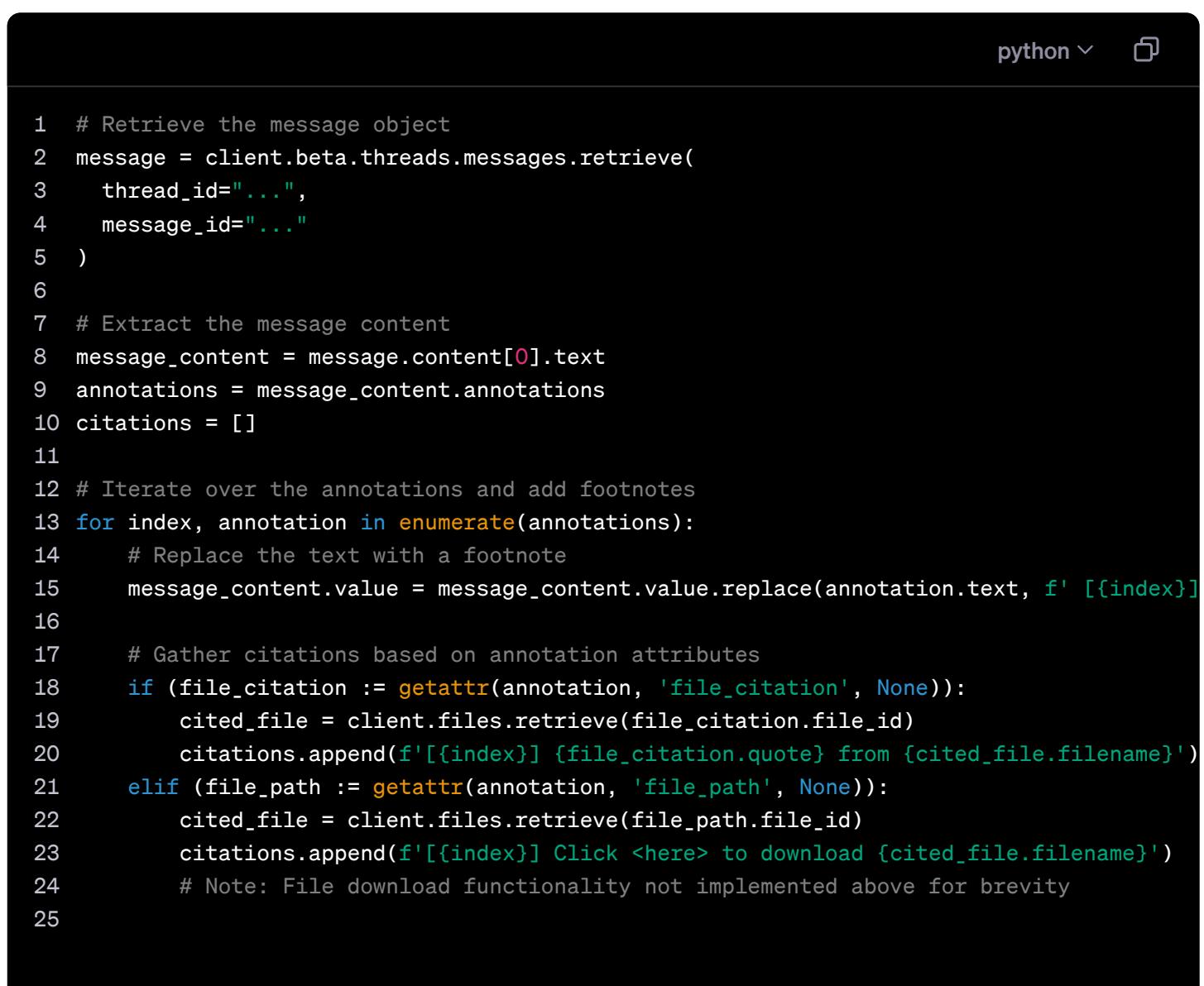
Message annotations

Messages created by Assistants may contain `annotations` within the `content` array of the object. Annotations provide information around how you should annotate the text in the Message.

There are two types of Annotations:

- 1 `file_citation` : File citations are created by the `file_search` tool and define references to a specific file that was uploaded and used by the Assistant to generate the response.
- 2 `file_path` : File path annotations are created by the `code_interpreter` tool and contain references to the files generated by the tool.

When annotations are present in the Message object, you'll see illegible model-generated substrings in the text that you should replace with the annotations. These strings may look something like `【13+source】` or `sandbox:/mnt/data/file.csv`. Here's an example python code snippet that replaces these strings with information present in the annotations.



The screenshot shows a Jupyter Notebook cell with the following code:

```
1 # Retrieve the message object
2 message = client.beta.threads.messages.retrieve(
3     thread_id="...",
4     message_id="..."
5 )
6
7 # Extract the message content
8 message_content = message.content[0].text
9 annotations = message_content.annotations
10 citations = []
11
12 # Iterate over the annotations and add footnotes
13 for index, annotation in enumerate(annotations):
14     # Replace the text with a footnote
15     message_content.value = message_content.value.replace(annotation.text, f' [{index}]')
16
17     # Gather citations based on annotation attributes
18     if (file_citation := getattr(annotation, 'file_citation', None)):
19         cited_file = client.files.retrieve(file_citation.file_id)
20         citations.append(f'[{index}] {file_citation.quote} from {cited_file.filename}')
21     elif (file_path := getattr(annotation, 'file_path', None)):
22         cited_file = client.files.retrieve(file_path.file_id)
23         citations.append(f'[{index}] Click <here> to download {cited_file.filename}')
24     # Note: File download functionality not implemented above for brevity
25
```

The code retrieves a message, extracts its content, and iterates over its annotations. For each annotation, it replaces the text with a numbered footnote. It then gathers citations based on the `file_citation` attribute of the annotation, which points to a file in the Assistant's storage. If there is no citation, it gathers file paths instead, which are represented as links to download the files.

```
26 # Add footnotes to the end of the message before displaying to user
27 message_content.value += '\n' + '\n'.join(citations)
```

Runs and Run Steps

When you have all the context you need from your user in the Thread, you can run the Thread with an Assistant of your choice.

```
1 run = client.beta.threads.runs.create(
2     thread_id=thread.id,
3     assistant_id=assistant.id
4 )
```

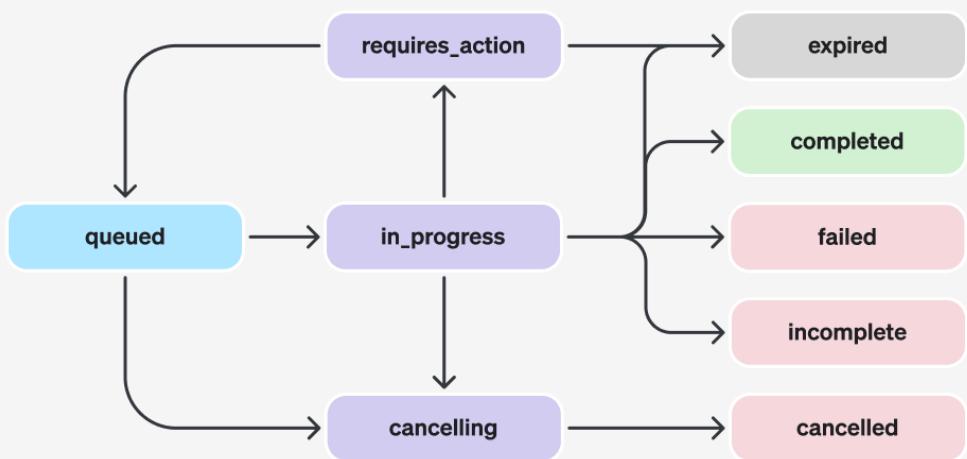
By default, a Run will use the `model` and `tools` configuration specified in Assistant object, but you can override most of these when creating the Run for added flexibility:

```
1 run = client.beta.threads.runs.create(
2     thread_id=thread.id,
3     assistant_id=assistant.id,
4     model="gpt-4-turbo",
5     instructions="New instructions that override the Assistant instructions",
6     tools=[{"type": "code_interpreter"}, {"type": "file_search"}]
7 )
```

Note: `tool_resources` associated with the Assistant cannot be overridden during Run creation. You must use the [modify Assistant](#) endpoint to do this.

Run lifecycle

Run objects can have multiple statuses.



STATUS	DEFINITION
queued	When Runs are first created or when you complete the required_action, they are moved to a queued status. They should almost immediately move to in_progress.
in_progress	While in_progress, the Assistant uses the model and tools to perform steps. You can view progress being made by the Run by examining the Run Steps .
completed	The Run successfully completed! You can now view all Messages the Assistant added to the Thread, and all the steps the Run took. You can also continue the conversation by adding more user Messages to the Thread and creating another Run.
requires_action	When using the Function calling tool, the Run will move to a required_action state once the model determines the names and arguments of the functions to be called. You must then run those functions and submit the outputs before the run proceeds. If the outputs are not provided before the expires_at timestamp passes (roughly 10 mins past creation), the run will move to an expired status.
expired	This happens when the function calling outputs were not submitted before expires_at and the run expires. Additionally, if the runs take too long to execute and go beyond the time stated in expires_at, our systems will expire the run.
cancelling	You can attempt to cancel an in_progress run using the Cancel Run endpoint. Once the attempt to cancel succeeds, status of the Run moves to cancelled. Cancellation is attempted but not guaranteed.
cancelled	Run was successfully cancelled.
failed	You can view the reason for the failure by looking at the last_error object in the Run. The timestamp for the failure will be recorded under failed_at.

STATUS	DEFINITION
incomplete	Run ended due to <code>max_prompt_tokens</code> or <code>max_completion_tokens</code> reached. You can view the specific reason by looking at the <code>incomplete_details</code> object in the Run.

Polling for updates

If you are not using [streaming](#), in order to keep the status of your run up to date, you will have to periodically [retrieve the Run](#) object. You can check the status of the run each time you retrieve the object to determine what your application should do next.

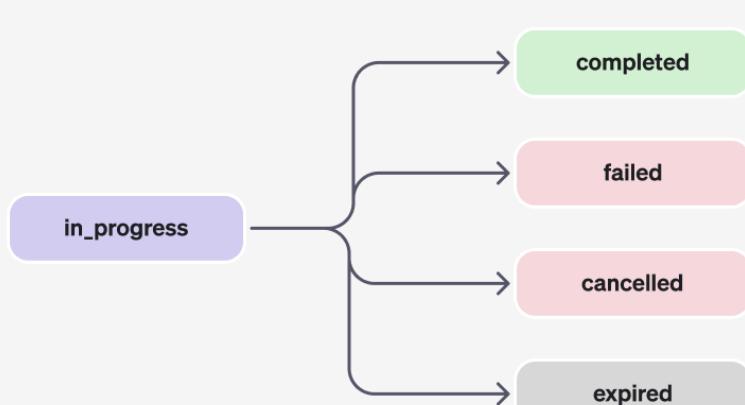
You can optionally use Polling Helpers in our [Node](#) and [Python](#) SDKs to help you with this. These helpers will automatically poll the Run object for you and return the Run object when it's in a terminal state.

Thread locks

When a Run is `in_progress` and not in a terminal state, the Thread is locked. This means that:

- New Messages cannot be added to the Thread.
- New Runs cannot be created on the Thread.

Run steps



Run step statuses have the same meaning as Run statuses.

Most of the interesting detail in the Run Step object lives in the `step_details` field. There can be two types of step details:

- 1 `message_creation` : This Run Step is created when the Assistant creates a Message on the Thread.

- 2 `tool_calls` : This Run Step is created when the Assistant calls a tool. Details around this are covered in the relevant sections of the [Tools](#) guide.

Data access guidance

Currently, Assistants, Threads, Messages, and Vector Stores created via the API are scoped to the Project they're created in. As such, any person with API key access to that Project is able to read or write Assistants, Threads, Messages, and Runs in the Project.

We strongly recommend the following data access controls:

- *Implement authorization.* Before performing reads or writes on Assistants, Threads, Messages, and Vector Stores, ensure that the end-user is authorized to do so. For example, store in your database the object IDs that the end-user has access to, and check it before fetching the object ID with the API.
- *Restrict API key access.* Carefully consider who in your organization should have API keys and be part of a Project. Periodically audit this list. API keys enable a wide range of operations including reading and modifying sensitive information, such as Messages and Files.
- *Create separate accounts.* Consider creating separate Projects for different applications in order to isolate data across multiple applications.

Next

Now that you have explored how Assistants work, the next step is to explore [Assistant Tools](#) which covers topics like Function calling, File Search, and Code Interpreter.

File Search

Beta

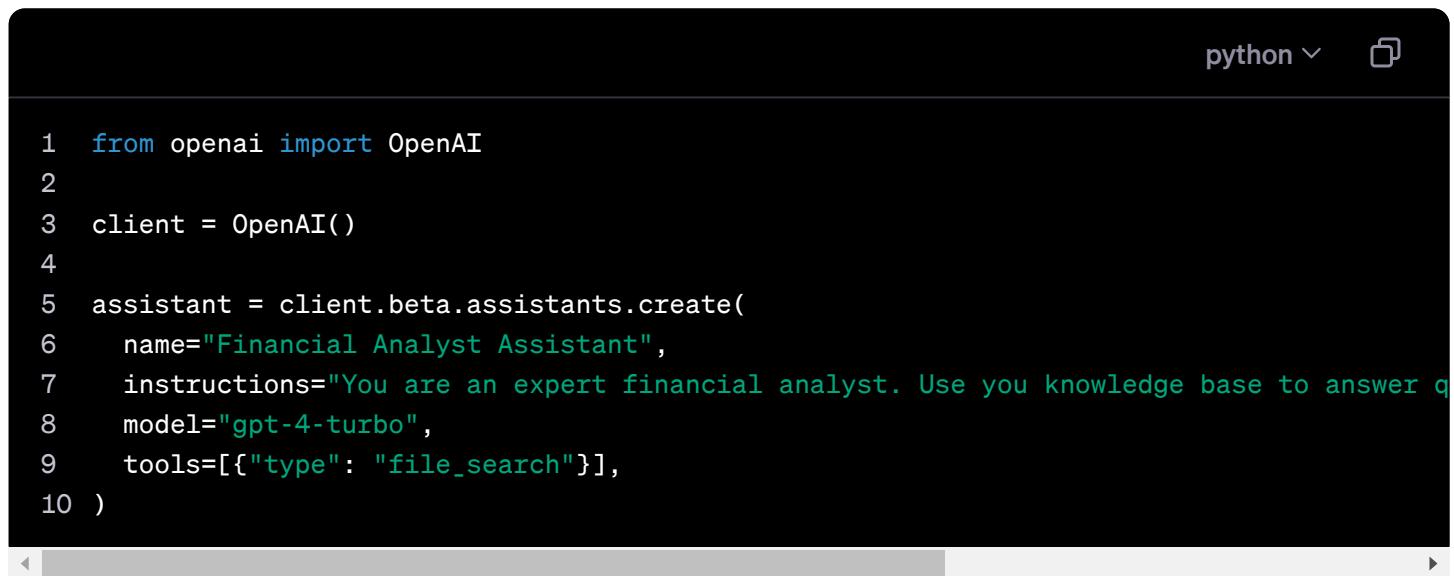
File Search augments the Assistant with knowledge from outside its model, such as proprietary product information or documents provided by your users. OpenAI automatically parses and chunks your documents, creates and stores the embeddings, and use both vector and keyword search to retrieve relevant content to answer user queries.

Quickstart

In this example, we'll create an assistant that can help answer questions about companies' financial statements.

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.



A screenshot of a code editor window titled "python". The code is a Python script using the OpenAI API to create a new assistant. The code defines a client, creates an assistant with a name ("Financial Analyst Assistant"), instructions ("You are an expert financial analyst. Use your knowledge base to answer questions"), model ("gpt-4-turbo"), and tools (["type": "file_search"]). The code is numbered from 1 to 10.

```
1 from openai import OpenAI
2
3 client = OpenAI()
4
5 assistant = client.beta.assistants.create(
6     name="Financial Analyst Assistant",
7     instructions="You are an expert financial analyst. Use your knowledge base to answer questions",
8     model="gpt-4-turbo",
9     tools=[{"type": "file_search"}],
10 )
```

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.

```
1 # Create a vector store called "Financial Statements"
2 vector_store = client.beta.vector_stores.create(name="Financial Statements")
3
4 # Ready the files for upload to OpenAI
5 file_paths = ["edgar/goog-10k.pdf", "edgar/brka-10k.txt"]
6 file_streams = [open(path, "rb") for path in file_paths]
7
8 # Use the upload and poll SDK helper to upload the files, add them to the vector store,
9 # and poll the status of the file batch for completion.
10 file_batch = client.beta.vector_stores.file_batches.upload_and_poll(
11     vector_store_id=vector_store.id, files=file_streams
12 )
13
14 # You can print the status and the file counts of the batch to see the result of this operation
15 print(file_batch.status)
16 print(file_batch.file_counts)
```

Step 3: Update the assistant 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`.

```
1 assistant = client.beta.assistants.update(
2     assistant_id=assistant.id,
3     tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},
4 )
```

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.

In this example, the user attached a copy of Apple's latest 10-K filing.

python ▾



```
1 # Upload the user provided file to OpenAI
2 message_file = client.files.create(
3     file=open("edgar/aapl-10k.pdf", "rb"), purpose="assistants"
4 )
5
6 # Create a thread and attach the file to the message
7 thread = client.beta.threads.create(
8     messages=[
9         {
10             "role": "user",
11             "content": "How many shares of AAPL were outstanding at the end of October 2022?"
12             # Attach the new file to the message.
13             "attachments": [
14                 { "file_id": message_file.id, "tools": [{"type": "file_search"}] }
15             ],
16         },
17     ]
18 )
19
20 # The thread now has a vector store with that file in its tool resources.
21 print(thread.tool_resources.file_search)
```

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

Step 5: Create a run and check the output

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

With streaming

Without streaming

python ▾



```
1 from typing_extensions import override
2 from openai import AssistantEventHandler, OpenAI
3
4 client = OpenAI()
```

```

5
6 class EventHandler(AssistantEventHandler):
7     @override
8     def on_text_created(self, text) -> None:
9         print(f"\nassistant > ", end="", flush=True)
10
11    @override
12    def on_tool_call_created(self, tool_call):
13        print(f"\nassistant > {tool_call.type}\n", flush=True)
14
15    @override
16    def on_message_done(self, message) -> None:
17        # print a citation to the file searched
18        message_content = message.content[0].text
19        annotations = message_content.annotations
20        citations = []
21        for index, annotation in enumerate(annotations):
22            message_content.value = message_content.value.replace(
23                annotation.text, f"[{index}]"
24            )
25            if file_citation := getattr(annotation, "file_citation", None):
26                cited_file = client.files.retrieve(file_citation.file_id)
27                citations.append(f"[{index}] {cited_file.filename}")
28
29        print(message_content.value)
30        print("\n".join(citations))
31
32
33 # Then, we use the stream SDK helper
34 # with the EventHandler class to create the Run
35 # and stream the response.
36
37 with client.beta.threads.runs.stream(
38     thread_id=thread.id,
39     assistant_id=assistant.id,
40     instructions="Please address the user as Jane Doe. The user has a premium account.",
41     event_handler=EventHandler(),
42 ) as stream:
43     stream.until_done()

```

Your new assistant will query both attached vector stores (one containing `goog-10k.pdf` and `brka-10k.txt`, and the other containing `aapl-10k.pdf`) and return this result from `aapl-10k.pdf`.

How it works

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

- Rewrites user queries to optimize them for search.
- Breaks down complex user queries into multiple searches it can run in parallel.
- Runs both keyword and semantic searches across both assistant and thread vector stores.
- Reranks search results to pick the most relevant ones before generating the final response.

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 (could be fewer)

Known Limitations

We have a few known limitations we're working on adding support for in the coming months:

- 1 Support for modifying chunking, embedding, and other retrieval configurations.
- 2 Support for deterministic pre-search filtering using custom metadata.
- 3 Support for parsing images within documents (including images of charts, graphs, tables etc.)
- 4 Support for retrievals over structured file formats (like `csv` or `jsonl`).
- 5 Better support for summarization — the tool today is optimized for search queries.

Vector stores

Vector Store objects give the File Search tool the ability to search your files. Adding a file to a `vector_store` automatically parses, chunks, embeds and stores the file in a vector database that's capable of both keyword and semantic search. Each `vector_store` can hold up to 10,000 files. Vector stores can be attached to both Assistants and Threads. Today, you can attach at most one vector store to an assistant and at most one vector store to a thread.

Creating vector stores and adding files

You can create a vector store and add files to it in a single API call:



```
python <code>
```

A screenshot of a terminal window. The window has a dark background and light-colored text. At the top, there is a dropdown menu labeled "python" and a small icon. Below the menu, there is a command prompt and some text output from the terminal. The text is mostly illegible due to the low resolution of the screenshot.

```
1 vector_store = client.beta.vector_stores.create(  
2     name="Product Documentation",  
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5']  
4 )
```

Adding files to vector stores is an async operation. To ensure the operation is complete, we recommend that you use the 'create and poll' helpers in our official SDKs. If you're not using the SDKs, you can retrieve the `vector_store` object and monitor its `file_counts` property to see the result of the file ingestion operation.

Files can also be added to a vector store after it's created by [creating vector store files](#).

```
1 file = client.beta.vector_stores.files.create_and_poll(  
2     vector_store_id="vs_abc123",  
3     file_id="file-abc123"  
4 )
```

Alternatively, you can add several files to a vector store by [creating batches](#) of up to 500 files.

```
1 batch = client.beta.vector_stores.file_batches.create_and_poll(  
2     vector_store_id="vs_abc123",  
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5']  
4 )
```

Similarly, these files can be removed from a vector store by either:

- Deleting the [vector store file object](#) or,
- By deleting the underlying [file object](#) (which removes the file it from all `vector_store` and `code_interpreter` configurations across all assistants and threads in your organization)

The maximum file size is 512 MB. Each file should contain no more than 5,000,000 tokens per file (computed automatically when you attach a file).

File Search supports a variety of file formats including `.pdf`, `.md`, and `.docx`. More details on the file extensions (and their corresponding MIME-types) supported can be found in the [Supported files](#) section below.

Attaching vector stores

You can attach vector stores to your Assistant or Thread using the `tool_resources` parameter.

```
python ▾
```

```
1 assistant = client.beta.assistants.create(
2     instructions="You are a helpful product support assistant and you answer questions ba",
3     model="gpt-4-turbo",
4     tools=[{"type": "file_search"}],
5     tool_resources={
6         "file_search": {
7             "vector_store_ids": ["vs_1"]
8         }
9     }
10 )
11
12 thread = client.beta.threads.create(
13     messages=[ { "role": "user", "content": "How do I cancel my subscription?"} ],
14     tool_resources={
15         "file_search": {
16             "vector_store_ids": ["vs_2"]
17         }
18     }
19 )
```

You can also attach a vector store to Threads or Assistants after they're created by updating them with the right `tool_resources`.

Ensuring vector store readiness before creating runs

We highly recommend that you ensure all files in a `vector_store` are fully processed before you create a run. This will ensure that all the data in your `vector_store` is searchable. You can check for `vector_store` readiness by using the polling helpers in our SDKs, or by manually polling the `vector_store` object to ensure the `status` is `completed`.

As a fallback, we've built a **60 second maximum wait** in the Run object when the **thread's** vector store contains files that are still being processed. This is to ensure that any files your users upload in a thread are fully searchable before the run proceeds. This fallback wait does *not* apply to the assistant's vector store.

Managing costs with expiration policies

The `file_search` tool uses the `vector_stores` object as its resource and you will be billed based on the `size` of the `vector_store` objects created. The size of the vector store object is the sum of all the parsed chunks from your files and their corresponding embeddings.

You first GB is free and beyond that, usage is billed at \$0.10/GB/day of vector storage. There are no other costs associated with vector store operations.

In order to help you manage the costs associated with these `vector_store` objects, we have added support for expiration policies in the `vector_store` object. You can set these policies when creating or updating the `vector_store` object.

```
python ▾
```

```
1 vector_store = client.beta.vector_stores.create_and_poll(
2     name="Product Documentation",
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5'],
4     expires_after={
5         "anchor": "last_active_at",
6         "days": 7
7     }
8 )
```

Thread vector stores have default expiration policies

Vector stores created using thread helpers (like `tool_resources.file_search.vector_stores` in Threads or `message.attachments` in Messages) 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).

When a vector store expires, runs on that thread will fail. To fix this, you can simply recreate a new `vector_store` with the same files and reattach it to the thread.

```
python ▾
```

```
1 all_files = list(client.beta.vector_stores.files.list("vs_expired"))
2
3 vector_store = client.beta.vector_stores.create(name="rag-store")
4 client.beta.threads.update(
5     "thread_abc123",
6     tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},
7 )
8
9 for file_batch in chunked(all_files, 100):
10     client.beta.vector_stores.file_batches.create_and_poll()
```

```
11         vector_store_id=vector_store.id, file_ids=[file.id for file in file_batch]
12     )
```

Supported files

For `text/` MIME types, the encoding must be one of `utf-8`, `utf-16`, or `ascii`.

FILE FORMAT	MIME TYPE
.c	<code>text/x-c</code>
.cs	<code>text/x-csharp</code>
.cpp	<code>text/x-c++</code>
.doc	<code>application/msword</code>
.docx	<code>application/vnd.openxmlformats-officedocument.wordprocessingml.document</code>
.html	<code>text/html</code>
.java	<code>text/x-java</code>
.json	<code>application/json</code>
.md	<code>text/markdown</code>
.pdf	<code>application/pdf</code>
.php	<code>text/x-php</code>
.pptx	<code>application/vnd.openxmlformats-officedocument.presentationml.presentation</code>
.py	<code>text/x-python</code>
.py	<code>text/x-script.python</code>
.rb	<code>text/x-ruby</code>
.tex	<code>text/x-tex</code>
.txt	<code>text/plain</code>
.css	<code>text/css</code>
.js	<code>text/javascript</code>

FILE FORMAT	MIME TYPE
.sh	application/x-sh
.ts	application/typescript

Code Interpreter

Beta

Code Interpreter allows Assistants to write and run Python code in a sandboxed execution environment. This tool can process files with diverse data and formatting, and generate files with data and images of graphs. Code Interpreter allows your Assistant to run code iteratively to solve challenging code and math problems. When your Assistant writes code that fails to run, it can iterate on this code by attempting to run different code until the code execution succeeds.

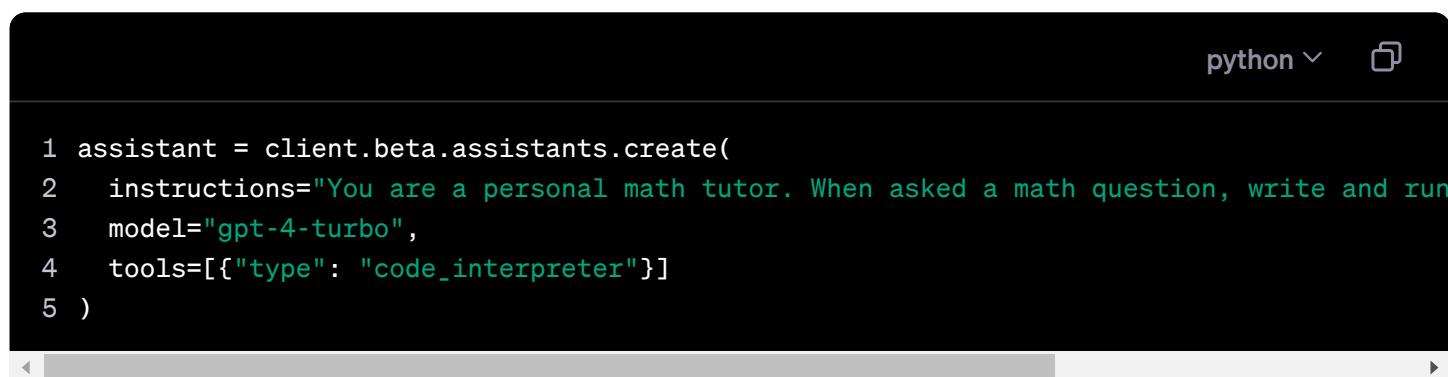
See a quickstart of how to get started with Code Interpreter [here](#).

How it works

Code Interpreter is charged at \$0.03 per session. If your Assistant calls Code Interpreter simultaneously in two different threads (e.g., one thread per end-user), two Code Interpreter sessions are created. Each session is active by default for one hour, which means that you only pay for one session per if users interact with Code Interpreter in the same thread for up to one hour.

Enabling Code Interpreter

Pass `code_interpreter` in the `tools` parameter of the Assistant object to enable Code Interpreter:



```
python ▾
```

```
1 assistant = client.beta.assistants.create(
2     instructions="You are a personal math tutor. When asked a math question, write and run
3     model='gpt-4-turbo',
4     tools=[{"type": "code_interpreter"}]
5 )
```

The model then decides when to invoke Code Interpreter in a Run based on the nature of the user request. This behavior can be promoted by prompting in the Assistant's `instructions` (e.g., “write code to solve this problem”).

Passing files to Code Interpreter

Files that are passed at the Assistant level are accessible by all Runs with this Assistant:

python ▾

```
1 # Upload a file with an "assistants" purpose
2 file = client.files.create(
3     file=open("mydata.csv", "rb"),
4     purpose='assistants'
5 )
6
7 # Create an assistant using the file ID
8 assistant = client.beta.assistants.create(
9     instructions="You are a personal math tutor. When asked a math question, write and ru
10    model="gpt-4-turbo",
11    tools=[{"type": "code_interpreter"}],
12    tool_resources={
13        "code_interpreter": {
14            "file_ids": [file.id]
15        }
16    }
17 )
```

Files can also be passed at the Thread level. These files are only accessible in the specific Thread. Upload the File using the [File upload](#) endpoint and then pass the File ID as part of the Message creation request:

python ▾

```
1 thread = client.beta.threads.create(
2     messages=[
3         {
4             "role": "user",
5             "content": "I need to solve the equation `3x + 11 = 14`. Can you help me?",
6             "attachments": [
7                 {
8                     "file_id": file.id,
9                     "tools": [{"type": "code_interpreter"}]
10                }
11            ]
12        }
13    ]
14 )
```

Files have a maximum size of 512 MB. Code Interpreter supports a variety of file formats including [.csv](#) , [.pdf](#) , [.json](#) and many more. More details on the file extensions (and their corresponding

MIME-types) supported can be found in the [Supported files](#) section below.

Reading images and files generated by Code Interpreter

Code Interpreter in the API also outputs files, such as generating image diagrams, CSVs, and PDFs. There are two types of files that are generated:

- 1 Images
- 2 Data files (e.g. a `csv` file with data generated by the Assistant)

When Code Interpreter generates an image, you can look up and download this file in the `file_id` field of the Assistant Message response:

```
1  {
2      "id": "msg_abc123",
3      "object": "thread.message",
4      "created_at": 1698964262,
5      "thread_id": "thread_abc123",
6      "role": "assistant",
7      "content": [
8          {
9              "type": "image_file",
10             "image_file": {
11                 "file_id": "file-abc123"
12             }
13         }
14     ]
15     # ...
16 }
```

The file content can then be downloaded by passing the file ID to the Files API:

```
1 from openai import OpenAI
2
3 client = OpenAI()
4
5 image_data = client.files.content("file-abc123")
6 image_data_bytes = image_data.read()
```

```
7  
8 with open("./my-image.png", "wb") as file:  
9     file.write(image_data_bytes)
```

When Code Interpreter references a file path (e.g., "Download this csv file"), file paths are listed as annotations. You can convert these annotations into links to download the file:

```
1 {  
2     "id": "msg_abc123",  
3     "object": "thread.message",  
4     "created_at": 1699073585,  
5     "thread_id": "thread_abc123",  
6     "role": "assistant",  
7     "content": [  
8         {  
9             "type": "text",  
10            "text": {  
11                "value": "The rows of the CSV file have been shuffled and saved to  
12                \"annotations\": [  
13                    {  
14                        "type": "file_path",  
15                        "text": "sandbox:/mnt/data/shuffled_file.csv",  
16                        "start_index": 167,  
17                        "end_index": 202,  
18                        "file_path": {  
19                            "file_id": "file-abc123"  
20                        }  
21                    }  
22                    ...  
23                ]  
24            }  
25        }  
26    }  
27    ...
```

Input and output logs of Code Interpreter

By listing the steps of a Run that called Code Interpreter, you can inspect the code `input` and `outputs` logs of Code Interpreter:

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

python ▾ 

```
3     run_id=run.id  
4 )
```

```
1 {  
2     "object": "list",  
3     "data": [  
4         {  
5             "id": "step_abc123",  
6             "object": "thread.run.step",  
7             "type": "tool_calls",  
8             "run_id": "run_abc123",  
9             "thread_id": "thread_abc123",  
10            "status": "completed",  
11            "step_details": {  
12                "type": "tool_calls",  
13                "tool_calls": [  
14                    {  
15                        "type": "code",  
16                        "code": {  
17                            "input": "# Calculating 2 + 2\nresult = 2 + 2\nresult",  
18                            "outputs": [  
19                                {  
20                                    "type": "logs",  
21                                    "logs": "4"  
22                                }  
23                                ...  
24                }
```

Supported files

For `text/` MIME types, the encoding must be one of `utf-8`, `utf-16`, or `ascii`.

FILE FORMAT	MIME TYPE
.c	<code>text/x-c</code>
.cs	<code>text/x-csharp</code>
.cpp	<code>text/x-c++</code>

FILE FORMAT	MIME TYPE
.doc	application/msword
.docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
.html	text/html
.java	text/x-java
.json	application/json
.md	text/markdown
.pdf	application/pdf
.php	text/x-php
.pptx	application/vnd.openxmlformats-officedocument.presentationml.presentation
.py	text/x-python
.py	text/x-script.python
.rb	text/x-ruby
.tex	text/x-tex
.txt	text/plain
.css	text/css
.js	text/javascript
.sh	application/x-sh
.ts	application/typescript
.csv	application/csv
.jpeg	image/jpeg
.jpg	image/jpeg
.gif	image/gif
.png	image/png

FILE FORMAT	MIME TYPE
.tar	application/x-tar
.xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
.xml	application/xml or "text/xml"
.zip	application/zip

Function calling

Beta

Similar to the Chat Completions API, the Assistants API supports function calling. Function calling allows you to describe functions to the Assistants API and have it intelligently return the functions that need to be called along with their arguments.

Quickstart

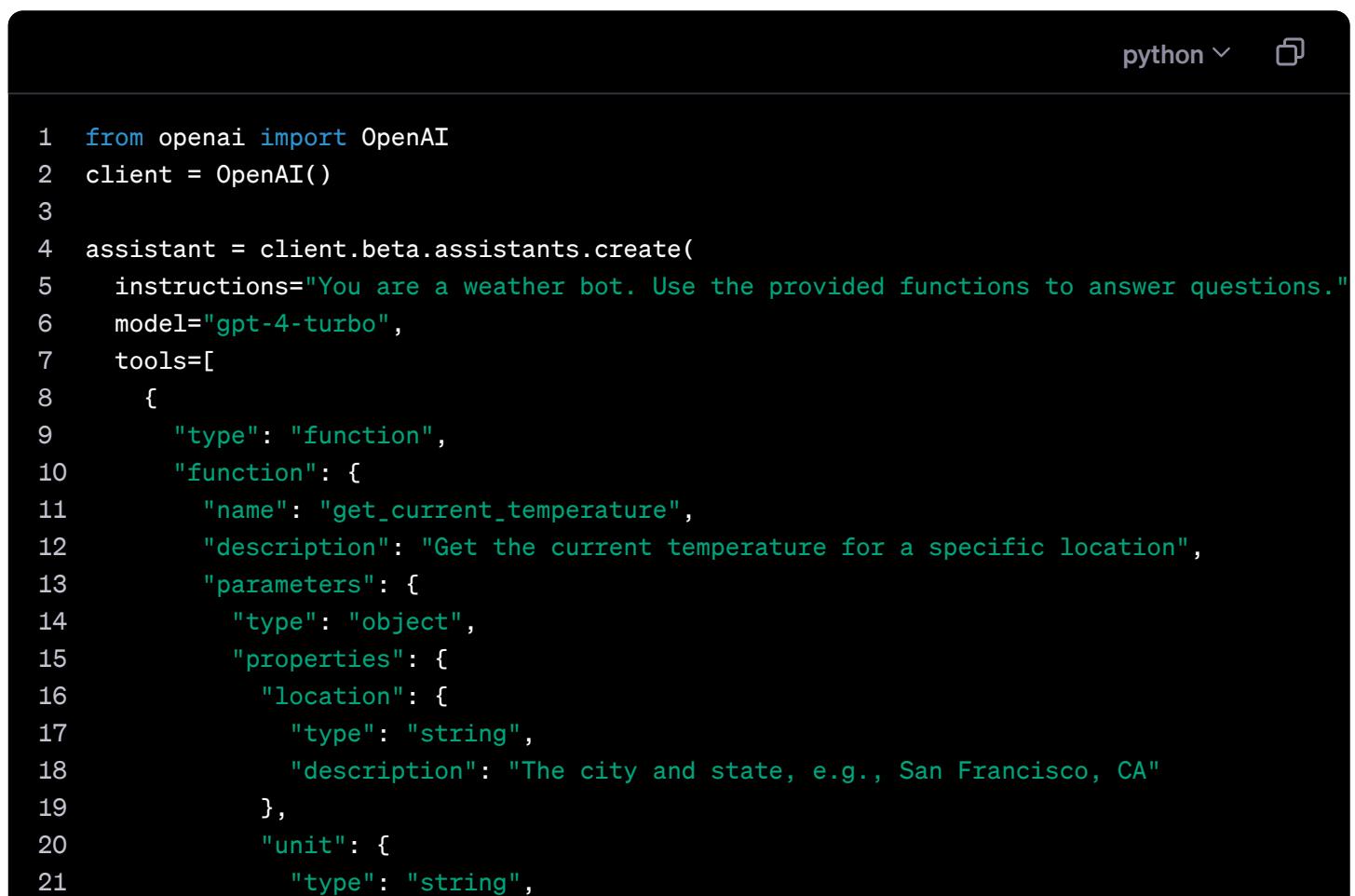
In this example, we'll create a weather assistant and define two functions,

`get_current_temperature` and `get_rain_probability`, as tools that the Assistant can call.

Depending on the user query, the model will invoke parallel function calling if using our latest models released on or after Nov 6, 2023. In our example that uses parallel function calling, we will ask the Assistant what the weather in San Francisco is like today and the chances of rain. We also show how to output the Assistant's response with streaming.

Step 1: Define functions

When creating your assistant, you will first define the functions under the `tools` param of the assistant.



The screenshot shows a code editor window with a dark theme. At the top right, there is a dropdown menu set to "python" and a copy icon. The code itself is a Python script using the OpenAI library to create an assistant with function calling capabilities. The code defines a function named "get_current_temperature" which takes a "location" parameter and returns a temperature value. It also defines a function named "get_rain_probability" which takes a "location" parameter and returns a probability value. These functions are registered as tools for the assistant.

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 assistant = client.beta.assistants.create(
5     instructions="You are a weather bot. Use the provided functions to answer questions.",
6     model="gpt-4-turbo",
7     tools=[
8         {
9             "type": "function",
10            "function": {
11                "name": "get_current_temperature",
12                "description": "Get the current temperature for a specific location",
13                "parameters": {
14                    "type": "object",
15                    "properties": {
16                        "location": {
17                            "type": "string",
18                            "description": "The city and state, e.g., San Francisco, CA"
19                        },
20                        "unit": {
21                            "type": "string",
22                            "description": "The unit of measurement, e.g., Celsius or Fahrenheit"
23                        }
24                    }
25                }
26            }
27        ]
28    )
29
30 print(assistant)
```

```
22         "enum": ["Celsius", "Fahrenheit"],
23         "description": "The temperature unit to use. Infer this from the user's l
24     }
25   },
26   "required": ["location", "unit"]
27 }
28 ]
29 ],
30 {
31   "type": "function",
32   "function": {
33     "name": "get_rain_probability",
34     "description": "Get the probability of rain for a specific location",
35     "parameters": {
36       "type": "object",
37       "properties": {
38         "location": {
39           "type": "string",
40           "description": "The city and state, e.g., San Francisco, CA"
41         }
42       },
43       "required": ["location"]
44     }
45   }
46 }
47 ]
48 )
```

Step 2: Create a Thread and add Messages

Create a Thread when a user starts a conversation and add Messages to the Thread as the user asks questions.

```
1 thread = client.beta.threads.create()
2 message = client.beta.threads.messages.create(
3   thread_id=thread.id,
4   role="user",
5   content="What's the weather in San Francisco today and the likelihood it'll rain?",
6 )
```

Step 3: Initiate a Run

When you initiate a Run on a Thread containing a user Message that triggers one or more functions, the Run will enter a `pending` status. After it processes, the run will enter a `requires_action` state which you can verify by checking the Run's `status`. This indicates that you need to run tools and submit their outputs to the Assistant to continue Run execution. In our case, we will see two `tool_calls`, which indicates that the user query resulted in parallel function calling.

- ⓘ Note that runs expire ten minutes after creation. Be sure to submit your tool outputs before the 10 min mark.

You will see two `tool_calls` within `required_action`, which indicates the user query triggered parallel function calling.

```
1  {
2      "id": "run_qJL1kI9xxWlfE0z1yfLOfGg9",
3      ...
4      "status": "requires_action",
5      "required_action": {
6          "submit_tool_outputs": {
7              "tool_calls": [
8                  {
9                      "id": "call_FthC9qRpsL5kBpwwyw6c7j4k",
10                     "function": {
11                         "arguments": "{\"location\": \"San Francisco, CA\"}",
12                         "name": "get_rain_probability"
13                     },
14                     "type": "function"
15                 },
16                 {
17                     "id": "call_RpEDoB800FTL9JoKTuCVFOyR",
18                     "function": {
19                         "arguments": "{\"location\": \"San Francisco, CA\", \"unit\": \"Fahrenheit\"}",
20                         "name": "get_current_temperature"
21                     },
22                     "type": "function"
23                 }
24             ]
25         },
26         ...
27     "type": "submit_tool_outputs"
28 }
29 }
```

Run object truncated here for readability

How you initiate a Run and submit `tool_calls` will differ depending on whether you are using streaming or not, although in both cases all `tool_calls` need to be submitted at the same time. You can then complete the Run by submitting the tool outputs from the functions you called. Pass each `tool_call_id` referenced in the `required_action` object to match outputs to each function call.

With streaming

Without streaming

For the streaming case, we create an `EventHandler` class to handle events in the response stream and submit all tool outputs at once with the “submit tool outputs stream” helper in the Python and Node SDKs.

```
python ▾
```

```
1  from typing_extensions import override
2  from openai import AssistantEventHandler
3
4  class EventHandler(AssistantEventHandler):
5      @override
6      def on_event(self, event):
7          # Retrieve events that are denoted with 'requires_action'
8          # since these will have our tool_calls
9          if event.event == 'thread.run.requires_action':
10              run_id = event.data.id # Retrieve the run ID from the event data
11              self.handle_requires_action(event.data, run_id)
12
13      def handle_requires_action(self, data, run_id):
14          tool_outputs = []
15
16          for tool in data.required_action.submit_tool_outputs.tool_calls:
17              if tool.function.name == "get_current_temperature":
18                  tool_outputs.append({"tool_call_id": tool.id, "output": "57"})
19              elif tool.function.name == "get_rain_probability":
20                  tool_outputs.append({"tool_call_id": tool.id, "output": "0.06"})
21
22          # Submit all tool_outputs at the same time
23          self.submit_tool_outputs(tool_outputs, run_id)
24
25      def submit_tool_outputs(self, tool_outputs, run_id):
26          # Use the submit_tool_outputs_stream helper
27          with client.beta.threads.runs.submit_tool_outputs_stream(
28              thread_id=self.current_run.thread_id,
29              run_id=self.current_run.id,
30              tool_outputs=tool_outputs,
```

```
31     event_handler=EventHandler(),
32 ) as stream:
33     for text in stream.text_deltas:
34         print(text, end="", flush=True)
35     print()
36
37
38 with client.beta.threads.runs.stream(
39     thread_id=thread.id,
40     assistant_id=assistant.id,
41     event_handler=EventHandler()
42 ) as stream:
43     stream.until_done()
```

Migration Guide

Beta

We have changed the way that tools and files work in the Assistants API between the v1 and v2 versions of the beta. Both versions of the beta continue to be accessible via the API today, but we recommend migrating to the newest version of our APIs as soon as feasible. We will deprecate v1 of the beta by the end of 2024.

- ⓘ If you do not use tools or files with the Assistants API today, there should be no changes required for you to migrate from the `v1` version to the `v2` version of the beta. Simply pass the [`v2` beta version header](#) and/or move to the latest version of our Node and Python SDKs!

What has changed

The v2 version of the Assistants API contains the following changes:

- 1 **Tool rename:** The `retrieval` tool has been renamed to the `file_search` tool
- 2 **Files belong to tools:** Files are now associated with tools instead of Assistants and Messages. This means that:
 - `AssistantFile` and `MessageFile` objects no longer exist.
 - Instead of `AssistantFile` and `MessageFile`, files are attached to Assistants and **Threads** using the new `tool_resources` object.
 - The `tool_resources` for the code interpreter tool are a list of `file_ids`.
 - The `tool_resources` for the `file_search` tool are a new object called a `vector_stores`.
 - Messages now have an `attachments`, rather than a `file_ids` parameter. Message attachments are helpers that add the files to a Thread's `tool_resources`.

V1 Assistant

```
1  {
2    "id": "asst_abc123",
3    "object": "assistant",
4    "created_at": 1698984975,
5    "name": "Math Tutor",
6    "description": null,
7    "model": "gpt-4-turbo",
8    "instructions": "You are a personal math tutor. When asked a question, write and ru
9    - "tools": [{ "type": "code_interpreter" }],
```

```
10 -   "file_ids": [],
11 -   "metadata": {}
12 - }
```

V2 Assistant

```
1  {
2    "id": "asst_abc123",
3    "object": "assistant",
4    "created_at": 1698984975,
5    "name": "Math Tutor",
6    "description": null,
7    "model": "gpt-4-turbo",
8    "instructions": "You are a personal math tutor. When asked a question, write and ru
9 +   "tools": [
10 +     {
11 +       "type": "code_interpreter"
12 +     },
13 +     {
14 +       "type": "file_search"
15 +     }
16 +   ],
17 +   "tool_resources": {
18 +     "file_search": {
19 +       "vector_store_ids": ["vs_abc"]
20 +     },
21 +     "code_interpreter": {
22 +       "file_ids": ["file-123", "file-456"]
23 +     }
24 +   }
25 }
```

Assistants have `tools` and `tool_resources` instead of `file_ids`. The `retrieval` tool is now the `file_search` tool. The `tool_resource` for the `file_search` tool is a `vector_store`.

V1 Thread

```
1  {
2    "id": "thread_abc123",
3    "object": "thread",
4    "created_at": 1699012949,
```

```
5 - "metadata": {}
6 -}
```

V2 Thread

```
1  {
2    "id": "thread_abc123",
3    "object": "thread",
4    "created_at": 1699012949,
5 + "metadata": {},
6 + "tools": [
7 +   {
8 +     "type": "file_search"
9 +   },
10 +   {
11 +     "type": "code_interpreter"
12 +   }
13 + ],
14 + "tool_resources": {
15 +   "file_search": {
16 +     "vector_store_ids": ["vs_abc"]
17 +   },
18 +   "code_interpreter": {
19 +     "file_ids": ["file-123", "file-456"]
20 +   }
21 + }
22 }
```

Threads can bring their own `tool_resources` into a conversation.

V1 Message

```
1  {
2    "id": "msg_abc123",
3    "object": "thread.message",
4    "created_at": 1698983503,
5    "thread_id": "thread_abc123",
6    "role": "assistant",
7    "content": [
8      {
9        "type": "text",
10       "text": {
11         "value": "Hi! How can I help you today?",
12         "annotations": []
13       }
14     }
15   ]
16 }
```

```
13     }
14   }
15 ],
16 "assistant_id": "asst_abc123",
17 "run_id": "run_abc123",
18 "metadata": {},
19 - "file_ids": []
20 -}
```

V2 Message

```
1  {
2   "id": "msg_abc123",
3   "object": "thread.message",
4   "created_at": 1698983503,
5   "thread_id": "thread_abc123",
6   "role": "assistant",
7   "content": [
8     {
9       "type": "text",
10      "text": {
11        "value": "Hi! How can I help you today?",  

12        "annotations": []
13      }
14    }
15  ],
16  "assistant_id": "asst_abc123",
17  "run_id": "run_abc123",
18  "metadata": {},
19 + "attachments": [
20 +   {
21 +     "file_id": "file-123",
22 +     "tools": [
23 +       { "type": "file_search" },
24 +       { "type": "code_interpreter" }
25 +     ]
26 +   }
27 + ]
28 }
```

Messages have `attachments` instead of `file_ids`. `attachments` are helpers that add files to the Thread's `tool_resources`.

All `v1` endpoints and objects for the Assistants API can be found under the [Legacy](#) section of the API reference.

Accessing v1 data in v2

To make your migration simple between our `v1` and `v2` APIs, we automatically map `AssistantFiles` and `MessageFiles` to the appropriate `tool_resources` based on the tools that are enabled in Assistants or Runs these files are a part of.

	V1 VERSION	V2 VERSION
AssistantFiles for <code>code_interpreter</code>	<code>file_ids</code> on Assistant	Files in an Assistant's <code>tool_resources.code_interpreter</code>
AssistantFiles for <code>retrieval</code>	<code>file_ids</code> on Assistant	Files in a <code>vector_store</code> attached to an Assistant (<code>tool_resources.file_search</code>)
MessageFiles for <code>code_interpreter</code>	<code>file_ids</code> on Message	Files in an Thread's <code>tool_resources.code_interpreter</code>
MessageFiles for <code>retrieval</code>	<code>file_ids</code> on Message	Files in a <code>vector_store</code> attached to a Thread (<code>tool_resources.file_search</code>)

-  It's important to note that while `file_ids` from `v1` are mapped to `tool_resources` in `v2`, the inverse is not true. Changes you make to `tool_resources` in `v2` will not be reflected as `file_ids` in `v1`.

Because Assistant Files and Message Files are already mapped to the appropriate `tool_resources` in `v2`, when you're ready to migrate to `v2` you shouldn't have to worry about a data migration. Instead, you only need to:

- 1 Update your integration to reflect the new API and objects. You may need to do things like:
 - Migrate to creating `vector_stores` and using `file_search`, if you were using the `retrieval` tool. Importantly, since these operations are asynchronous, you'll want to ensure files are [successfully ingested](#) by the `vector_stores` before creating run.
 - Migrate to adding files to `tool_resources.code_interpreter` instead of an Assistant or Message's files, if you were using the `code_interpreter` tool.
 - Migrate to using Message `attachments` instead of `file_ids`.
- 2 Upgrade to the latest version of our SDKs

Changing beta versions

Without SDKs

Both beta versions can be accessed by passing the right API version header in your API requests:

- 1 v1 : OpenAI-Beta: assistants=v1
- 2 v2 : OpenAI-Beta: assistants=v2

```
v2 ▾ ⌂  
1 curl "https://api.openai.com/v1/assistants" \  
2   -H "Content-Type: application/json" \  
3   -H "Authorization: Bearer $OPENAI_API_KEY" \  
4   -H "OpenAI-Beta: assistants=v2" \  
5   -d '{  
6     "instructions": "You are a personal math tutor. When asked a question, write and ru  
7     "name": "Math Tutor",  
8     "tools": [{"type": "code_interpreter"}],  
9     "model": "gpt-4-turbo"  
10 }'
```

With SDKs

Versions of our SDKs that are released after the release of the v2 beta will have the openai.beta namespace point to the v2 version of the API by default. You can still access the v1 version of the API by using an older version of the SDK (1.20.0 or earlier for python, 4.36.0 or earlier for node) or by overriding the version header.

To install an older version of the SDK, you can use the following commands:

```
python ▾ ⌂  
Installing older versions of the SDK  
pip install openai==1.20.0
```

You can also override this header in a newer SDK version, but we don't recommend this approach since the object types in these newer SDK versions will be different from the v1 objects.

```
python ▾ ⌂  
Accessing the `v1` API version in new SDKs
```

```
1 from openai import OpenAI
2
3 client = OpenAI(default_headers={"OpenAI-Beta": "assistants=v1"})
```

Billing

All **vector stores** created before the release of the **v2** API (April 17, 2024) will be free to use until the end of 2024. This implies that any vector stores that were created as a result of us mapping your **v1** data to **v2**, before the **v2** launch will be free. After the end of 2024, they'll be billed at whatever the fees for vector stores are at that point. See our [pricing page](#) for the latest pricing information.

Any vector store that is created before the release of the **v2** API (April 17, 2024) but not used in a single Run between that release date and the end of 2024 will be deleted. This is to avoid us starting to bill you for something you created during the beta but never used.

Vector stores created after the release of the **v2** API will be billed at current rates as specified on the [pricing page](#).

Deleting files

Deleting Assistant Files / Message Files via the **v1** API also removes them from the **v2** API. However, the inverse is not true - deletions in the **v2** version of the API do not propagate to **v1**. If you created a file on **v1** and would like to "fully" delete a file from your account on both **v1** and **v2** you should:

- delete Assistant Files / Message Files you create using **v1** APIs using the **v1** endpoints, or
- delete the underlying **file object** — this ensures it is fully removed from all objects in all versions of the API.

Playground

The default playground experience has been migrated to use the **v2** version of the API (you will still have a read-only view of the **v1** version of objects, but will not be able to edit them). Any changes you make to tools and files via the Playground will only be accessible in the **v2** version of the API.

In order to make changes to files in the **v1** version of the API, you will need to use the API directly.

What's new in v2?

Beta

April 2024

We are announcing a variety of new features and improvements to the Assistants API and moving our Beta to a new API version, `OpenAI-Beta: assistants=v2`. Here's what's new:

- We're launching an [improved retrieval tool called `file_search`](#), which can ingest up to 10,000 files per assistant - 500x more than before. It is faster, supports parallel queries through multi-threaded searches, and features enhanced reranking and query rewriting.
- Alongside `file_search`, we're introducing [vector store objects](#) in the API. Once a file is added to a vector store, it's automatically parsed, chunked, and embedded, made ready to be searched. Vector stores can be used across assistants and threads, simplifying file management and billing.
- You can now [control the maximum number of tokens](#) a run uses in the Assistants API, allowing you to manage token usage costs. You can also set limits on the number of previous / recent messages used in each run.
- We've added support for the [tool_choice parameter](#) which can be used to force the use of a specific tool (like `file_search`, `code_interpreter`, or a `function`) in a particular run.
- You can now [create messages with the role `assistant`](#) to create custom conversation histories in Threads.
- Assistant and Run objects now support popular model configuration parameters like `temperature`, `response_format (JSON mode)`, and `top_p`.
- You can now use [fine-tuned models](#) in the Assistants API. At the moment, only fine-tuned versions of `gpt-3.5-turbo-0125` are supported.
- Assistants API now supports [streaming](#).
- We've added several streaming and polling helpers to our [Node](#) and [Python](#) SDKs.

See our [migration guide](#) to learn more about how to migrate your tool usage to the latest version of the Assistants API.

Prompt engineering

This guide shares strategies and tactics for getting better results from large language models (sometimes referred to as GPT models) like GPT-4. The methods described here can sometimes be deployed in combination for greater effect. We encourage experimentation to find the methods that work best for you.

Some of the examples demonstrated here currently work only with our most capable model, `gpt-4`. In general, if you find that a model fails at a task and a more capable model is available, it's often worth trying again with the more capable model.

You can also explore example prompts which showcase what our models are capable of:



Prompt examples

Explore prompt examples to learn what GPT models can do

Six strategies for getting better results

Write clear instructions

These models can't read your mind. If outputs are too long, ask for brief replies. If outputs are too simple, ask for expert-level writing. If you dislike the format, demonstrate the format you'd like to see. The less the model has to guess at what you want, the more likely you'll get it.

Tactics:

- [Include details in your query to get more relevant answers](#)
- [Ask the model to adopt a persona](#)
- [Use delimiters to clearly indicate distinct parts of the input](#)
- [Specify the steps required to complete a task](#)
- [Provide examples](#)
- [Specify the desired length of the output](#)

Provide reference text

Language models can confidently invent fake answers, especially when asked about esoteric topics or for citations and URLs. In the same way that a sheet of notes can help a student do better on a test, providing reference text to these models can help in answering with fewer fabrications.

Tactics:

- Instruct the model to answer using a reference text
- Instruct the model to answer with citations from a reference text

Split complex tasks into simpler subtasks

Just as it is good practice in software engineering to decompose a complex system into a set of modular components, the same is true of tasks submitted to a language model. Complex tasks tend to have higher error rates than simpler tasks. Furthermore, complex tasks can often be re-defined as a workflow of simpler tasks in which the outputs of earlier tasks are used to construct the inputs to later tasks.

Tactics:

- Use intent classification to identify the most relevant instructions for a user query
- For dialogue applications that require very long conversations, summarize or filter previous dialogue
- Summarize long documents piecewise and construct a full summary recursively

Give the model time to "think"

If asked to multiply 17 by 28, you might not know it instantly, but can still work it out with time. Similarly, models make more reasoning errors when trying to answer right away, rather than taking time to work out an answer. Asking for a "chain of thought" before an answer can help the model reason its way toward correct answers more reliably.

Tactics:

- Instruct the model to work out its own solution before rushing to a conclusion
- Use inner monologue or a sequence of queries to hide the model's reasoning process
- Ask the model if it missed anything on previous passes

Use external tools

Compensate for the weaknesses of the model by feeding it the outputs of other tools. For example, a text retrieval system (sometimes called RAG or retrieval augmented generation) can tell the model about relevant documents. A code execution engine like OpenAI's Code Interpreter can help the model do math and run code. If a task can be done more reliably or efficiently by a tool rather than by a language model, offload it to get the best of both.

Tactics:

- Use embeddings-based search to implement efficient knowledge retrieval
- Use code execution to perform more accurate calculations or call external APIs
- Give the model access to specific functions

Test changes systematically

Improving performance is easier if you can measure it. In some cases a modification to a prompt will achieve better performance on a few isolated examples but lead to worse overall performance on a more representative set of examples. Therefore to be sure that a change is net positive to performance it may be necessary to define a comprehensive test suite (also known as an "eval").

Tactic:

- Evaluate model outputs with reference to gold-standard answers

Tactics

Each of the strategies listed above can be instantiated with specific tactics. These tactics are meant to provide ideas for things to try. They are by no means fully comprehensive, and you should feel free to try creative ideas not represented here.

Strategy: Write clear instructions

Tactic: Include details in your query to get more relevant answers

In order to get a highly relevant response, make sure that requests provide any important details or context. Otherwise you are leaving it up to the model to guess what you mean.

Worse	Better
How do I add numbers in Excel?	How do I add up a row of dollar amounts in Excel? I want to do this automatically for a whole sheet of rows with all the totals ending up on the right in a column called "Total".
Who's president?	Who was the president of Mexico in 2021, and how frequently are elections held?
Write code to calculate the Fibonacci sequence.	Write a TypeScript function to efficiently calculate the Fibonacci sequence. Comment the code liberally to explain what each piece does and why it's written that way.
Summarize the meeting notes.	Summarize the meeting notes in a single paragraph. Then write a markdown list of the speakers and each of their key points. Finally, list the next steps or action items suggested by the speakers, if any.

Tactic: Ask the model to adopt a persona

The system message can be used to specify the persona used by the model in its replies.

SYSTEM When I ask for help to write something, you will reply with a document that contains at least one joke or playful comment in every paragraph.

USER Write a thank you note to my steel bolt vendor for getting the delivery in on time and in short notice. This made it possible for us to deliver an important order.

[Open in Playground ↗](#)

Tactic: Use delimiters to clearly indicate distinct parts of the input

Delimiters like triple quotation marks, XML tags, section titles, etc. can help demarcate sections of text to be treated differently.

USER Summarize the text delimited by triple quotes with a haiku.

"""
"insert text here"
"""

[Open in Playground ↗](#)

SYSTEM You will be provided with a pair of articles (delimited with XML tags) about the same topic. First summarize the arguments of each article. Then indicate which of them makes a better argument and explain why.

USER <article> insert first article here </article>

<article> insert second article here </article>

[Open in Playground ↗](#)

SYSTEM You will be provided with a thesis abstract and a suggested title for it. The thesis title should give the reader a good idea of the topic of the thesis but should also be eye-catching. If the title does not meet these criteria, suggest 5 alternatives.

USER Abstract: insert abstract here

Title: insert title here

[Open in Playground ↗](#)

For straightforward tasks such as these, using delimiters might not make a difference in the output quality. However, the more complex a task is the more important it is to disambiguate task details. Don't

make the model work to understand exactly what you are asking of them.

Tactic: Specify the steps required to complete a task

Some tasks are best specified as a sequence of steps. Writing the steps out explicitly can make it easier for the model to follow them.

SYSTEM Use the following step-by-step instructions to respond to user inputs.

Step 1 - The user will provide you with text in triple quotes. Summarize this text in one sentence with a prefix that says "Summary: ".

Step 2 - Translate the summary from Step 1 into Spanish, with a prefix that says "Translation: ".

USER """"insert text here""""

[Open in Playground ↗](#)

Tactic: Provide examples

Providing general instructions that apply to all examples is generally more efficient than demonstrating all permutations of a task by example, but in some cases providing examples may be easier. For example, if you intend for the model to copy a particular style of responding to user queries which is difficult to describe explicitly. This is known as "few-shot" prompting.

SYSTEM Answer in a consistent style.

USER Teach me about patience.

ASSISTANT The river that carves the deepest valley flows from a modest spring; the grandest symphony originates from a single note; the most intricate tapestry begins with a solitary thread.

USER Teach me about the ocean.

[Open in Playground ↗](#)

Tactic: Specify the desired length of the output

You can ask the model to produce outputs that are of a given target length. The targeted output length can be specified in terms of the count of words, sentences, paragraphs, bullet points, etc. Note however

that instructing the model to generate a specific number of words does not work with high precision. The model can more reliably generate outputs with a specific number of paragraphs or bullet points.

USER Summarize the text delimited by triple quotes in about 50 words.

"""insert text here"""

[Open in Playground ↗](#)

USER Summarize the text delimited by triple quotes in 2 paragraphs.

"""insert text here"""

[Open in Playground ↗](#)

USER Summarize the text delimited by triple quotes in 3 bullet points.

"""insert text here"""

[Open in Playground ↗](#)

Strategy: Provide reference text

Tactic: Instruct the model to answer using a reference text

If we can provide a model with trusted information that is relevant to the current query, then we can instruct the model to use the provided information to compose its answer.

SYSTEM Use the provided articles delimited by triple quotes to answer questions. If the answer cannot be found in the articles, write "I could not find an answer."

USER <insert articles, each delimited by triple quotes>

Question: <insert question here>

[Open in Playground ↗](#)

Given that all models have limited context windows, we need some way to dynamically lookup information that is relevant to the question being asked. [Embeddings](#) can be used to implement efficient

knowledge retrieval. See the tactic "["Use embeddings-based search to implement efficient knowledge retrieval"](#)" for more details on how to implement this.

Tactic: Instruct the model to answer with citations from a reference text

If the input has been supplemented with relevant knowledge, it's straightforward to request that the model add citations to its answers by referencing passages from provided documents. Note that citations in the output can then be verified programmatically by string matching within the provided documents.

SYSTEM	You will be provided with a document delimited by triple quotes and a question. Your task is to answer the question using only the provided document and to cite the passage(s) of the document used to answer the question. If the document does not contain the information needed to answer this question then simply write: "Insufficient information." If an answer to the question is provided, it must be annotated with a citation. Use the following format for to cite relevant passages ({"citation": ...}).
USER	"""<insert document here>"""\n\nQuestion: <insert question here>

[Open in Playground ↗](#)

Strategy: Split complex tasks into simpler subtasks

Tactic: Use intent classification to identify the most relevant instructions for a user query

For tasks in which lots of independent sets of instructions are needed to handle different cases, it can be beneficial to first classify the type of query and to use that classification to determine which instructions are needed. This can be achieved by defining fixed categories and hardcoding instructions that are relevant for handling tasks in a given category. This process can also be applied recursively to decompose a task into a sequence of stages. The advantage of this approach is that each query will contain only those instructions that are required to perform the next stage of a task which can result in lower error rates compared to using a single query to perform the whole task. This can also result in lower costs since larger prompts cost more to run ([see pricing information](#)).

Suppose for example that for a customer service application, queries could be usefully classified as follows:

SYSTEM	You will be provided with customer service queries. Classify each query into a primary category and a secondary category. Provide your output in json format with the keys: primary and secondary.
---------------	--

Primary categories: Billing, Technical Support, Account Management, or General Inquiry.

Billing secondary categories:

- Unsubscribe or upgrade
- Add a payment method
- Explanation for charge
- Dispute a charge

Technical Support secondary categories:

- Troubleshooting
- Device compatibility
- Software updates

Account Management secondary categories:

- Password reset
- Update personal information
- Close account
- Account security

General Inquiry secondary categories:

- Product information
- Pricing
- Feedback
- Speak to a human

USER I need to get my internet working again.

[Open in Playground ↗](#)

Based on the classification of the customer query, a set of more specific instructions can be provided to a model for it to handle next steps. For example, suppose the customer requires help with "troubleshooting".

SYSTEM You will be provided with customer service inquiries that require troubleshooting in a technical support context. Help the user by:

- Ask them to check that all cables to/from the router are connected. Note that it is common for cables to come loose over time.
- If all cables are connected and the issue persists, ask them which router model they are using
- Now you will advise them how to restart their device:
 - If the model number is MTD-327J, advise them to push the red button and hold it for 5 seconds, then wait 5 minutes before testing the connection.

- If the model number is MTD-327S, advise them to unplug and replug it, then wait 5 minutes before testing the connection.
- If the customer's issue persists after restarting the device and waiting 5 minutes, connect them to IT support by outputting {"IT support requested"}.
- If the user starts asking questions that are unrelated to this topic then confirm if they would like to end the current chat about troubleshooting and classify their request according to the following scheme:

<insert primary/secondary classification scheme from above here>

USER I need to get my internet working again.

[Open in Playground ↗](#)

Notice that the model has been instructed to emit special strings to indicate when the state of the conversation changes. This enables us to turn our system into a state machine where the state determines which instructions are injected. By keeping track of state, what instructions are relevant at that state, and also optionally what state transitions are allowed from that state, we can put guardrails around the user experience that would be hard to achieve with a less structured approach.

Tactic: For dialogue applications that require very long conversations, summarize or filter previous dialogue

Since models have a fixed context length, dialogue between a user and an assistant in which the entire conversation is included in the context window cannot continue indefinitely.

There are various workarounds to this problem, one of which is to summarize previous turns in the conversation. Once the size of the input reaches a predetermined threshold length, this could trigger a query that summarizes part of the conversation and the summary of the prior conversation could be included as part of the system message. Alternatively, prior conversation could be summarized asynchronously in the background throughout the entire conversation.

An alternative solution is to dynamically select previous parts of the conversation that are most relevant to the current query. See the tactic "["Use embeddings-based search to implement efficient knowledge retrieval"](#)".

Tactic: Summarize long documents piecewise and construct a full summary recursively

Since models have a fixed context length, they cannot be used to summarize a text longer than the context length minus the length of the generated summary in a single query.

To summarize a very long document such as a book we can use a sequence of queries to summarize each section of the document. Section summaries can be concatenated and summarized producing summaries of summaries. This process can proceed recursively until an entire document is summarized.

If it's necessary to use information about earlier sections in order to make sense of later sections, then a further trick that can be useful is to include a running summary of the text that precedes any given point in the book while summarizing content at that point. The effectiveness of this procedure for summarizing books has been studied in previous [research](#) by OpenAI using variants of GPT-3.

Strategy: Give models time to "think"

Tactic: Instruct the model to work out its own solution before rushing to a conclusion

Sometimes we get better results when we explicitly instruct the model to reason from first principles before coming to a conclusion. Suppose for example we want a model to evaluate a student's solution to a math problem. The most obvious way to approach this is to simply ask the model if the student's solution is correct or not.

SYSTEM	Determine if the student's solution is correct or not.
USER	<p>Problem Statement: I'm building a solar power installation and I need help working out the financials.</p> <ul style="list-style-type: none">- Land costs \$100 / square foot- I can buy solar panels for \$250 / square foot- I negotiated a contract for maintenance that will cost me a flat \$100k per year, and an additional \$10 / square foot <p>What is the total cost for the first year of operations as a function of the number of square feet.</p> <p>Student's Solution: Let x be the size of the installation in square feet.</p> <ol style="list-style-type: none">1. Land cost: $100x$2. Solar panel cost: $250x$3. Maintenance cost: $100,000 + 100x$ <p>Total cost: $100x + 250x + 100,000 + 100x = 450x + 100,000$</p>
ASSISTANT	The student's solution is correct.

[Open in Playground ↗](#)

But the student's solution is actually not correct! We can get the model to successfully notice this by prompting the model to generate its own solution first.

SYSTEM	First work out your own solution to the problem. Then compare your solution to the student's solution and evaluate if the student's solution is correct or not. Don't decide if the student's solution is correct until you have done the problem yourself.
--------	---

USER

Problem Statement: I'm building a solar power installation and I need help working out the financials.

- Land costs \$100 / square foot
- I can buy solar panels for \$250 / square foot
- I negotiated a contract for maintenance that will cost me a flat \$100k per year, and an additional \$10 / square foot

What is the total cost for the first year of operations as a function of the number of square feet.

Student's Solution: Let x be the size of the installation in square feet.

1. Land cost: $100x$
2. Solar panel cost: $250x$
3. Maintenance cost: $100,000 + 100x$

$$\text{Total cost: } 100x + 250x + 100,000 + 100x = 450x + 100,000$$

ASSISTANT

Let x be the size of the installation in square feet.

1. Land cost: $100x$
2. Solar panel cost: $250x$
3. Maintenance cost: $100,000 + 10x$

$$\text{Total cost: } 100x + 250x + 100,000 + 10x = 360x + 100,000$$

The student's solution is incorrect. They made an error in the maintenance cost calculation, using $100x$ instead of $10x$. The correct total cost for the first year of operations is $360x + 100,000$.

[Open in Playground ↗](#)

Tactic: Use inner monologue or a sequence of queries to hide the model's reasoning process

The previous tactic demonstrates that it is sometimes important for the model to reason in detail about a problem before answering a specific question. For some applications, the reasoning process that a model uses to arrive at a final answer would be inappropriate to share with the user. For example, in tutoring applications we may want to encourage students to work out their own answers, but a model's reasoning process about the student's solution could reveal the answer to the student.

Inner monologue is a tactic that can be used to mitigate this. The idea of inner monologue is to instruct the model to put parts of the output that are meant to be hidden from the user into a structured format that makes parsing them easy. Then before presenting the output to the user, the output is parsed and only part of the output is made visible.

SYSTEM

Follow these steps to answer the user queries.

Step 1 - First work out your own solution to the problem. Don't rely on the student's

solution since it may be incorrect. Enclose all your work for this step within triple quotes (""""").

Step 2 - Compare your solution to the student's solution and evaluate if the student's solution is correct or not. Enclose all your work for this step within triple quotes (""""").

Step 3 - If the student made a mistake, determine what hint you could give the student without giving away the answer. Enclose all your work for this step within triple quotes (""""").

Step 4 - If the student made a mistake, provide the hint from the previous step to the student (outside of triple quotes). Instead of writing "Step 4 - ..." write "Hint:".

USER Problem Statement: <insert problem statement>

Student Solution: <insert student solution>

Open in Playground ↗

Alternatively, this can be achieved with a sequence of queries in which all except the last have their output hidden from the end user.

First, we can ask the model to solve the problem on its own. Since this initial query doesn't require the student's solution, it can be omitted. This provides the additional advantage that there is no chance that the model's solution will be biased by the student's attempted solution.

USER <insert problem statement>

Open in Playground ↗

Next, we can have the model use all available information to assess the correctness of the student's solution.

SYSTEM Compare your solution to the student's solution and evaluate if the student's solution is correct or not.

USER Problem statement: """<insert problem statement>"""

Your solution: """<insert model generated solution>"""

Student's solution: """<insert student's solution>"""

Open in Playground ↗

Finally, we can let the model use its own analysis to construct a reply in the persona of a helpful tutor.

SYSTEM You are a math tutor. If the student made an error, offer a hint to the student in a way that does not reveal the answer. If the student did not make an error, simply offer them an encouraging comment.

USER Problem statement: """"<insert problem statement>""""

Your solution: """"<insert model generated solution>""""

Student's solution: """"<insert student's solution>""""

Analysis: """"<insert model generated analysis from previous step>""""

[Open in Playground ↗](#)

Tactic: Ask the model if it missed anything on previous passes

Suppose that we are using a model to list excerpts from a source which are relevant to a particular question. After listing each excerpt the model needs to determine if it should start writing another or if it should stop. If the source document is large, it is common for a model to stop too early and fail to list all relevant excerpts. In that case, better performance can often be obtained by prompting the model with followup queries to find any excerpts it missed on previous passes.

SYSTEM You will be provided with a document delimited by triple quotes. Your task is to select excerpts which pertain to the following question: "What significant paradigm shifts have occurred in the history of artificial intelligence."

Ensure that excerpts contain all relevant context needed to interpret them - in other words don't extract small snippets that are missing important context. Provide output in JSON format as follows:

```
[{"excerpt": "..."},  
 ...  
 {"excerpt": "..."}]
```

USER """"<insert document here>""""

ASSISTANT [{"excerpt": "the model writes an excerpt here"},
 ...
 {"excerpt": "the model writes another excerpt here"}]

USER Are there more relevant excerpts? Take care not to repeat excerpts. Also ensure that excerpts contain all relevant context needed to interpret them - in other words don't

extract small snippets that are missing important context.

[Open in Playground ↗](#)

Strategy: Use external tools

Tactic: Use embeddings-based search to implement efficient knowledge retrieval

A model can leverage external sources of information if provided as part of its input. This can help the model to generate more informed and up-to-date responses. For example, if a user asks a question about a specific movie, it may be useful to add high quality information about the movie (e.g. actors, director, etc...) to the model's input. Embeddings can be used to implement efficient knowledge retrieval, so that relevant information can be added to the model input dynamically at run-time.

A text embedding is a vector that can measure the relatedness between text strings. Similar or relevant strings will be closer together than unrelated strings. This fact, along with the existence of fast vector search algorithms means that embeddings can be used to implement efficient knowledge retrieval. In particular, a text corpus can be split up into chunks, and each chunk can be embedded and stored. Then a given query can be embedded and vector search can be performed to find the embedded chunks of text from the corpus that are most related to the query (i.e. closest together in the embedding space).

Example implementations can be found in the [OpenAI Cookbook](#). See the tactic “[Instruct the model to use retrieved knowledge to answer queries](#)” for an example of how to use knowledge retrieval to minimize the likelihood that a model will make up incorrect facts.

Tactic: Use code execution to perform more accurate calculations or call external APIs

Language models cannot be relied upon to perform arithmetic or long calculations accurately on their own. In cases where this is needed, a model can be instructed to write and run code instead of making its own calculations. In particular, a model can be instructed to put code that is meant to be run into a designated format such as triple backtick. After an output is produced, the code can be extracted and run. Finally, if necessary, the output from the code execution engine (i.e. Python interpreter) can be provided as an input to the model for the next query.

SYSTEM You can write and execute Python code by enclosing it in triple backticks, e.g.
` ` ` code goes here ` ` `. Use this to perform calculations.

USER Find all real-valued roots of the following polynomial: $3*x^{**5} - 5*x^{**4} - 3*x^{**3} - 7*x - 10$.

[Open in Playground ↗](#)

Another good use case for code execution is calling external APIs. If a model is instructed in the proper use of an API, it can write code that makes use of it. A model can be instructed in how to use an API by providing it with documentation and/or code samples showing how to use the API.

SYSTEM You can write and execute Python code by enclosing it in triple backticks. Also note that you have access to the following module to help users send messages to their friends:

```
```python
import message
message.write(to="John", message="Hey, want to meetup after work?")````
```

[Open in Playground ↗](#)

**WARNING: Executing code produced by a model is not inherently safe and precautions should be taken in any application that seeks to do this. In particular, a sandboxed code execution environment is needed to limit the harm that untrusted code could cause.**

### Tactic: Give the model access to specific functions

The Chat Completions API allows passing a list of function descriptions in requests. This enables models to generate function arguments according to the provided schemas. Generated function arguments are returned by the API in JSON format and can be used to execute function calls. Output provided by function calls can then be fed back into a model in the following request to close the loop. This is the recommended way of using OpenAI models to call external functions. To learn more see the [function calling section](#) in our introductory text generation guide and more [function calling examples](#) in the OpenAI Cookbook.

### Strategy: Test changes systematically

Sometimes it can be hard to tell whether a change — e.g., a new instruction or a new design — makes your system better or worse. Looking at a few examples may hint at which is better, but with small sample sizes it can be hard to distinguish between a true improvement or random luck. Maybe the change helps performance on some inputs, but hurts performance on others.

Evaluation procedures (or "evals") are useful for optimizing system designs. Good evals are:

- Representative of real-world usage (or at least diverse)
- Contain many test cases for greater statistical power (see table below for guidelines)
- Easy to automate or repeat

DIFFERENCE TO DETECT	SAMPLE SIZE NEEDED FOR 95% CONFIDENCE
30%	~10
10%	~100
3%	~1,000
1%	~10,000

Evaluation of outputs can be done by computers, humans, or a mix. Computers can automate evals with objective criteria (e.g., questions with single correct answers) as well as some subjective or fuzzy criteria, in which model outputs are evaluated by other model queries. [OpenAI Evals](#) is an open-source software framework that provides tools for creating automated evals.

Model-based evals can be useful when there exists a range of possible outputs that would be considered equally high in quality (e.g. for questions with long answers). The boundary between what can be realistically evaluated with a model-based eval and what requires a human to evaluate is fuzzy and is constantly shifting as models become more capable. We encourage experimentation to figure out how well model-based evals can work for your use case.

### **Tactic: Evaluate model outputs with reference to gold-standard answers**

Suppose it is known that the correct answer to a question should make reference to a specific set of known facts. Then we can use a model query to count how many of the required facts are included in the answer.

For example, using the following system message:

**SYSTEM** You will be provided with text delimited by triple quotes that is supposed to be the answer to a question. Check if the following pieces of information are directly contained in the answer:

- Neil Armstrong was the first person to walk on the moon.
- The date Neil Armstrong first walked on the moon was July 21, 1969.

For each of these points perform the following steps:

- 1 - Restate the point.
- 2 - Provide a citation from the answer which is closest to this point.
- 3 - Consider if someone reading the citation who doesn't know the topic could directly infer the point. Explain why or why not before making up your mind.
- 4 - Write "yes" if the answer to 3 was yes, otherwise write "no".

Finally, provide a count of how many "yes" answers there are. Provide this count as {"count": <insert count here>}.

Here's an example input where both points are satisfied:

**SYSTEM** <insert system message above>

**USER** """Neil Armstrong is famous for being the first human to set foot on the Moon. This historic event took place on July 21, 1969, during the Apollo 11 mission."""

[Open in Playground ↗](#)

Here's an example input where only one point is satisfied:

**SYSTEM** <insert system message above>

**USER** """Neil Armstrong made history when he stepped off the lunar module, becoming the first person to walk on the moon."""

[Open in Playground ↗](#)

Here's an example input where none are satisfied:

**SYSTEM** <insert system message above>

**USER**        """In the summer of '69, a voyage grand,  
                 Apollo 11, bold as legend's hand.  
                 Armstrong took a step, history unfurled,  
                 "One small step," he said, for a new world."""

[Open in Playground ↗](#)

There are many possible variants on this type of model-based eval. Consider the following variation which tracks the kind of overlap between the candidate answer and the gold-standard answer, and also tracks whether the candidate answer contradicts any part of the gold-standard answer.

**SYSTEM**      Use the following steps to respond to user inputs. Fully restate each step before proceeding. i.e. "Step 1: Reason...".

Step 1: Reason step-by-step about whether the information in the submitted answer compared to the expert answer is either: disjoint, equal, a subset, a superset, or overlapping (i.e. some intersection but not subset/superset).

Step 2: Reason step-by-step about whether the submitted answer contradicts any aspect of the expert answer.

Step 3: Output a JSON object structured like: {"type\_of\_overlap": "disjoint" or "equal" or "subset" or "superset" or "overlapping", "contradiction": true or false}

Here's an example input with a substandard answer which nonetheless does not contradict the expert answer:

**SYSTEM**      <insert system message above>

**USER**          Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time."""

Submitted Answer: """Didn't he walk on the moon or something?"""

Expert Answer: """Neil Armstrong is most famous for being the first person to walk on the moon. This historic event occurred on July 21, 1969."""

[Open in Playground ↗](#)

Here's an example input with answer that directly contradicts the expert answer:

**SYSTEM** <insert system message above>

**USER** Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time."""

Submitted Answer: """On the 21st of July 1969, Neil Armstrong became the second person to walk on the moon, following after Buzz Aldrin."""

Expert Answer: """Neil Armstrong is most famous for being the first person to walk on the moon. This historic event occurred on July 21, 1969."""

[Open in Playground ↗](#)

Here's an example input with a correct answer that also provides a bit more detail than is necessary:

**SYSTEM** <insert system message above>

**USER** Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time."""

Submitted Answer: """At approximately 02:56 UTC on July 21st 1969, Neil Armstrong became the first human to set foot on the lunar surface, marking a monumental achievement in human history."""

Expert Answer: """Neil Armstrong is most famous for being the first person to walk on the moon. This historic event occurred on July 21, 1969."""

[Open in Playground ↗](#)

## Other resources

For more inspiration, visit the [OpenAI Cookbook](#), which contains example code and also links to third-party resources such as:

- [Prompting libraries & tools](#)
- [Prompting guides](#)
- [Video courses](#)
- [Papers on advanced prompting to improve reasoning](#)

# Production best practices

This guide provides a comprehensive set of best practices to help you transition from prototype to production. Whether you are a seasoned machine learning engineer or a recent enthusiast, this guide should provide you with the tools you need to successfully put the platform to work in a production setting: from securing access to our API to designing a robust architecture that can handle high traffic volumes. Use this guide to help develop a plan for deploying your application as smoothly and effectively as possible.

If you want to explore best practices for going into production further, please check out our Developer Day talk:

## Setting up your organization

Once you [log in](#) to your OpenAI account, you can find your organization name and ID in your [organization settings](#). The organization name is the label for your organization, shown in user interfaces. The organization ID is the unique identifier for your organization which can be used in API requests.

Users who belong to multiple organizations can [pass a header](#) to specify which organization is used for an API request. Usage from these API requests will count against the specified organization's quota. If no header is provided, the [default organization](#) will be billed. You can change your default organization in your [user settings](#).

You can invite new members to your organization from the [Team page](#). Members can be **readers** or **owners**. Readers can make API requests and view basic organization information, while owners can modify billing information and manage members within an organization.

## Managing billing limits

To begin using the OpenAI API, enter your [billing information](#). If no billing information is entered, you will still have login access but will be unable to make API requests.

Once you've entered your billing information, you will have an approved usage limit of \$100 per month, which is set by OpenAI. Your quota limit will automatically increase as your usage on your platform increases and you move from one [usage tier](#) to another. You can review your current usage limit in the [limits](#) page in your account settings.

If you'd like to be notified when your usage exceeds a certain dollar amount, you can set a notification threshold through the [usage limits](#) page. When the notification threshold is reached, the owners of the organization will receive an email notification. You can also set a monthly budget so that, once the monthly budget is reached, any subsequent API requests will be rejected. Note that these limits are best effort, and there may be 5 to 10 minutes of delay between the usage and the limits being enforced.

## API keys

The OpenAI API uses API keys for authentication. Visit your [API keys](#) page to retrieve the API key you'll use in your requests.

This is a relatively straightforward way to control access, but you must be vigilant about securing these keys. Avoid exposing the API keys in your code or in public repositories; instead, store them in a secure location. You should expose your keys to your application using environment variables or secret management service, so that you don't need to hard-code them in your codebase. Read more in our [Best practices for API key safety](#).

API key usage can be monitored on the [Usage page](#) once tracking is enabled. If you are using an API key generated prior to Dec 20, 2023 tracking will not be enabled by default. You can enable tracking going forward on the [API key management dashboard](#). All API keys generated past Dec 20, 2023 have tracking enabled. Any previous untracked usage will be displayed as [Untracked](#) in the dashboard.

## Staging accounts

As you scale, you may want to create separate organizations for your staging and production environments. Please note that you can sign up using two separate email addresses like [bob+prod@widgetcorp.com](mailto:bob+prod@widgetcorp.com) and [bob+dev@widgetcorp.com](mailto:bob+dev@widgetcorp.com) to create two organizations. This will allow you to isolate your development and testing work so you don't accidentally disrupt your live application. You can also limit access to your production organization this way.

# Scaling your solution architecture

When designing your application or service for production that uses our API, it's important to consider how you will scale to meet traffic demands. There are a few key areas you will need to consider regardless of the cloud service provider of your choice:

- **Horizontal scaling:** You may want to scale your application out horizontally to accommodate requests to your application that come from multiple sources. This could involve deploying additional servers or containers to distribute the load. If you opt for this type of scaling, make sure that your architecture is designed to handle multiple nodes and that you have mechanisms in place to balance the load between them.
- **Vertical scaling:** Another option is to scale your application up vertically, meaning you can beef up the resources available to a single node. This would involve upgrading your server's capabilities to handle the additional load. If you opt for this type of scaling, make sure your application is designed to take advantage of these additional resources.
- **Caching:** By storing frequently accessed data, you can improve response times without needing to make repeated calls to our API. Your application will need to be designed to use cached data whenever possible and invalidate the cache when new information is added. There are a few different ways you could do this. For example, you could store data in a database, filesystem, or in-memory cache, depending on what makes the most sense for your application.
- **Load balancing:** Finally, consider load-balancing techniques to ensure requests are distributed evenly across your available servers. This could involve using a load balancer in front of your servers or using DNS round-robin. Balancing the load will help improve performance and reduce bottlenecks.

## Managing rate limits

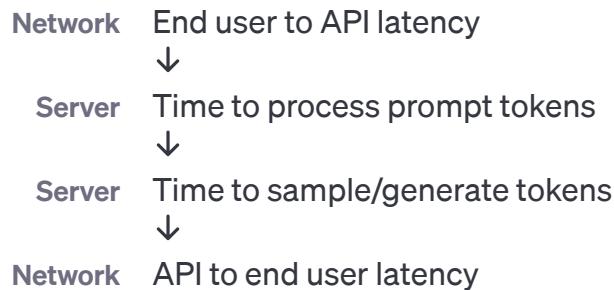
When using our API, it's important to understand and plan for [rate limits](#).

## Improving latencies

 Check out our most up-to-date guide on [latency optimization](#).

Latency is the time it takes for a request to be processed and a response to be returned. In this section, we will discuss some factors that influence the latency of our text generation models and provide suggestions on how to reduce it.

The latency of a completion request is mostly influenced by two factors: the model and the number of tokens generated. The life cycle of a completion request looks like this:



The bulk of the latency typically arises from the token generation step.

**Intuition:** Prompt tokens add very little latency to completion calls. Time to generate completion tokens is much longer, as tokens are generated one at a time. Longer generation lengths will accumulate latency due to generation required for each token.

## Common factors affecting latency and possible mitigation techniques

Now that we have looked at the basics of latency, let's take a look at various factors that can affect latency, broadly ordered from most impactful to least impactful.

### Model

Our API offers different models with varying levels of complexity and generality. The most capable models, such as `gpt-4`, can generate more complex and diverse completions, but they also take longer to process your query. Models such as `gpt-3.5-turbo`, can generate faster and cheaper chat completions, but they may generate results that are less accurate or relevant for your query. You can choose the model that best suits your use case and the trade-off between speed and quality.

### Number of completion tokens

Requesting a large amount of generated tokens completions can lead to increased latencies:

- **Lower max tokens:** for requests with a similar token generation count, those that have a lower `max_tokens` parameter incur less latency.
- **Include stop sequences:** to prevent generating unneeded tokens, add a stop sequence. For example, you can use stop sequences to generate a list with a specific number of items. In this case, by using `11.` as a stop sequence, you can generate a list with only 10 items, since the completion will stop when `11.` is reached. [Read our help article on stop sequences](#) for more context on how you can do this.
- **Generate fewer completions:** lower the values of `n` and `best_of` when possible where `n` refers to how many completions to generate for each prompt and `best_of` is used to represent the result with the highest log probability per token.

If `n` and `best_of` both equal 1 (which is the default), the number of generated tokens will be at most, equal to `max_tokens`.

If `n` (the number of completions returned) or `best_of` (the number of completions generated for consideration) are set to `> 1`, each request will create multiple outputs. Here, you can consider the number of generated tokens as `[ max_tokens * max (n, best_of) ]`

## Streaming

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.

## Infrastructure

Our servers are currently located in the US. While we hope to have global redundancy in the future, in the meantime you could consider locating the relevant parts of your infrastructure in the US to minimize the roundtrip time between your servers and the OpenAI servers.

## Batching

Depending on your use case, batching *may help*. If you are sending multiple requests to the same endpoint, you can [batch the prompts](#) to be sent in the same request. This will reduce the number of requests you need to make. The prompt parameter can hold up to 20 unique prompts. We advise you to test out this method and see if it helps. In some cases, you may end up increasing the number of generated tokens which will slow the response time.

## Managing costs

To monitor your costs, you can set a [notification threshold](#) in your account to receive an email alert once you pass a certain usage threshold. You can also set a [monthly budget](#). Please be mindful of the potential for a monthly budget to cause disruptions to your application/users. Use the [usage tracking dashboard](#) to monitor your token usage during the current and past billing cycles.

## Text generation

One of the challenges of moving your prototype into production is budgeting for the costs associated with running your application. OpenAI offers a [pay-as-you-go pricing model](#), with prices per 1,000

tokens (roughly equal to 750 words). To estimate your costs, you will need to project the token utilization. Consider factors such as traffic levels, the frequency with which users will interact with your application, and the amount of data you will be processing.

**One useful framework for thinking about reducing costs is to consider costs as a function of the number of tokens and the cost per token.** There are two potential avenues for reducing costs using this framework. First, you could work to reduce the cost per token by switching to smaller models for some tasks in order to reduce costs. Alternatively, you could try to reduce the number of tokens required. There are a few ways you could do this, such as by using shorter prompts, [fine-tuning](#) models, or caching common user queries so that they don't need to be processed repeatedly.

You can experiment with our interactive [tokenizer tool](#) to help you estimate costs. The API and playground also returns token counts as part of the response. Once you've got things working with our most capable model, you can see if the other models can produce the same results with lower latency and costs. Learn more in our [token usage help article](#).

## MLOps strategy

As you move your prototype into production, you may want to consider developing an MLOps strategy. MLOps (machine learning operations) refers to the process of managing the end-to-end life cycle of your machine learning models, including any models you may be fine-tuning using our API. There are a number of areas to consider when designing your MLOps strategy. These include

- Data and model management: managing the data used to train or fine-tune your model and tracking versions and changes.
- Model monitoring: tracking your model's performance over time and detecting any potential issues or degradation.
- Model retraining: ensuring your model stays up to date with changes in data or evolving requirements and retraining or fine-tuning it as needed.
- Model deployment: automating the process of deploying your model and related artifacts into production.

Thinking through these aspects of your application will help ensure your model stays relevant and performs well over time.

## Security and compliance

As you move your prototype into production, you will need to assess and address any security and compliance requirements that may apply to your application. This will involve examining the data you are handling, understanding how our API processes data, and determining what regulations you must

adhere to. Our [security practices](#) and [trust and compliance portal](#) provide our most comprehensive and up-to-date documentation. For reference, here is our [Privacy Policy](#) and [Terms of Use](#).

Some common areas you'll need to consider include data storage, data transmission, and data retention. You might also need to implement data privacy protections, such as encryption or anonymization where possible. In addition, you should follow best practices for secure coding, such as input sanitization and proper error handling.

## Safety best practices

When creating your application with our API, consider our [safety best practices](#) to ensure your application is safe and successful. These recommendations highlight the importance of testing the product extensively, being proactive about addressing potential issues, and limiting opportunities for misuse.

## Business considerations

As projects using AI move from prototype to production, it is important to consider how to build a great product with AI and how that ties back to your core business. We certainly don't have all the answers but a great starting place is a talk from our Developer Day where we dive into this with some of our customers:

# Safety best practices

## Use our free Moderation API

OpenAI's [Moderation API](#) is free-to-use and can help reduce the frequency of unsafe content in your completions. Alternatively, you may wish to develop your own content filtration system tailored to your use case.

## Adversarial testing

We recommend “red-teaming” your application to ensure it’s robust to adversarial input. Test your product over a wide range of inputs and user behaviors, both a representative set and those reflective of someone trying to ‘break’ your application. Does it wander off topic? Can someone easily redirect the feature via prompt injections, e.g. “ignore the previous instructions and do this instead”?

## Human in the loop (HITL)

Wherever possible, we recommend having a human review outputs before they are used in practice. This is especially critical in high-stakes domains, and for code generation. Humans should be aware of the limitations of the system, and have access to any information needed to verify the outputs (for example, if the application summarizes notes, a human should have easy access to the original notes to refer back).

## Prompt engineering

“Prompt engineering” can help constrain the topic and tone of output text. This reduces the chance of producing undesired content, even if a user tries to produce it. Providing additional context to the model (such as by giving a few high-quality examples of desired behavior prior to the new input) can make it easier to steer model outputs in desired directions.

## “Know your customer” (KYC)

Users should generally need to register and log-in to access your service. Linking this service to an existing account, such as a Gmail, LinkedIn, or Facebook log-in, may help, though may not be appropriate for all use-cases. Requiring a credit card or ID card reduces risk further.

## Constrain user input and limit output tokens

Limiting the amount of text a user can input into the prompt helps avoid prompt injection. Limiting the number of output tokens helps reduce the chance of misuse.

Narrowing the ranges of inputs or outputs, especially drawn from trusted sources, reduces the extent of misuse possible within an application.

Allowing user inputs through validated dropdown fields (e.g., a list of movies on Wikipedia) can be more secure than allowing open-ended text inputs.

Returning outputs from a validated set of materials on the backend, where possible, can be safer than returning novel generated content (for instance, routing a customer query to the best-matching existing customer support article, rather than attempting to answer the query from-scratch).

## Allow users to report issues

Users should generally have an easily-available method for reporting improper functionality or other concerns about application behavior (listed email address, ticket submission method, etc). This method should be monitored by a human and responded to as appropriate.

## Understand and communicate limitations

From hallucinating inaccurate information, to offensive outputs, to bias, and much more, language models may not be suitable for every use case without significant modifications. Consider whether the model is fit for your purpose, and evaluate the performance of the API on a wide range of potential inputs in order to identify cases where the API's performance might drop. Consider your customer base and the range of inputs that they will be using, and ensure their expectations are calibrated appropriately.

- i Safety and security are very important to us at OpenAI.

If in the course of your development you do notice any safety or security issues with the API or anything else related to OpenAI, please submit these through our [Coordinated Vulnerability Disclosure Program](#).

## End-user IDs

Sending end-user IDs in your requests can be a useful tool to help OpenAI monitor and detect abuse. This allows OpenAI to provide your team with more actionable feedback in the event that we detect any policy violations in your application.

The IDs should be a string that uniquely identifies each user. We recommend hashing their username or email address, in order to avoid sending us any identifying information. If you offer a preview of your product to non-logged in users, you can send a session ID instead.

You can include end-user IDs in your API requests via the `user` parameter as follows:

Example: Providing a user identifier

python ▾



```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.completions.create(
5 model="gpt-3.5-turbo-instruct",
6 prompt="This is a test",
7 max_tokens=5,
8 user="user_123456"
9)
```

# Latency optimization

This guide covers the core set of principles you can apply to improve latency across a wide variety of LLM-related use cases. These techniques come from working with a wide range of customers and developers on production applications, so they should apply regardless of what you're building – from a granular workflow to an end-to-end chatbot!

While there's many individual techniques, we'll be grouping them into **seven principles** meant to represent a high-level taxonomy of approaches for improving latency.

At the end, we'll walk through an [example](#) to see how they can be applied.

## The seven principles

- 1 Process tokens faster.
- 2 Generate fewer tokens.
- 3 Use fewer input tokens.
- 4 Make fewer requests.
- 5 Parallelize.
- 6 Make your users wait less.
- 7 Don't default to an LLM.

 You can use the friendly, catchy acronym PGIRPWD to remember these. (Processing, Generation, Input, Requests, Parallelize, Waiting, Don't)

### 1. Process tokens faster

**Inference speed** is probably the first thing that comes to mind when addressing latency (but as you'll see soon, it's far from the only one). This refers to the actual **rate at which the LLM processes tokens**, and is often measured in TPM (tokens per minute) or TPS (tokens per second).

The main factor that influences inference speed is **model size** – smaller models usually run faster (and cheaper), and when used correctly can even outperform larger models. To maintain high quality performance with smaller models you can explore:

- using a longer, [more detailed prompt](#),
- adding (more) [few-shot examples](#), or
- [fine-tuning](#) / distillation.

## DEEP DIVE

### Compute capacity & inference optimizations



## 2. Generate fewer tokens

Generating tokens is almost always the highest latency step when using an LLM: as a general heuristic, **cutting 50% of your output tokens may cut ~50% your latency**. The way you reduce your output size will depend on output type:

If you're generating **natural language**, simply **asking the model to be more concise** ("under 20 words" or "be very brief") may help. You can also use few shot examples and/or fine-tuning to teach the model shorter responses.

If you're generating **structured output**, try to **minimize your output syntax** where possible: shorten function names, omit named arguments, coalesce parameters, etc.

Finally, while not common, you can also use `max_tokens` or `stop_tokens` to end your generation early.

Always remember: an output token cut is a (milli)second earned!

## 3. Use fewer input tokens

While reducing the number of input tokens does result in lower latency, this is not usually a significant factor – **cutting 50% of your prompt may only result in a 1-5% latency improvement**. Unless you're working with truly massive context sizes (documents, images), you may want to spend your efforts elsewhere.

That being said, if you are working with massive contexts (or you're set on squeezing every last bit of performance and you've exhausted all other options) you can use the following techniques to reduce your input tokens:

- **Fine-tuning the model**, to replace the need for lengthy instructions / examples.
- **Filtering context input**, like pruning RAG results, cleaning HTML, etc.
- **Maximize shared prompt prefix**, by putting dynamic portions (e.g. RAG results, history, etc) later in the prompt. This makes your request more **KV cache**-friendly (which most LLM providers use) and means fewer input tokens are processed on each request. ([why?](#))

## 4. Make fewer requests

Each time you make a request you incur some round-trip latency – this can start to add up.

If you have sequential steps for the LLM to perform, instead of firing off one request per step consider **putting them in a single prompt and getting them all in a single response**. You'll avoid the additional round-trip latency, and potentially also reduce complexity of processing multiple responses.

An approach to doing this is by collecting your steps in an enumerated list in the combined prompt, and then requesting the model to return the results in named fields in a JSON. This way you can easily parse out and reference each result!

## 5. Parallelize

Parallelization can be very powerful when performing multiple steps with an LLM.

If the steps **are not strictly sequential**, you can **split them out into parallel calls**. Two shirts take just as long to dry as one.

If the steps **are strictly sequential**, however, you might still be able to **leverage speculative execution**. This is particularly effective for classification steps where one outcome is more likely than the others (e.g. moderation).

- 1 Start step 1 & step 2 simultaneously (e.g. input moderation & story generation)
- 2 Verify the result of step 1
- 3 If result was not the expected, cancel step 2 (and retry if necessary)

If your guess for step 1 is right, then you essentially got to run it with zero added latency!

## 6. Make your users wait less

There's a huge difference between **waiting** and **watching progress happen** – make sure your users experience the latter. Here are a few techniques:

- **Streaming:** The single most effective approach, as it cuts the **waiting** time to a second or less. (ChatGPT would feel pretty different if you saw nothing until each response was done.)
- **Chunking:** If your output needs further processing before being shown to the user (moderation, translation) consider **processing it in chunks** instead of all at once. Do this by streaming to your backend, then sending processed chunks to your frontend.
- **Show your steps:** If you're taking multiple steps or using tools, surface this to the user. The more real progress you can show, the better.
- **Loading states:** Spinners and progress bars go a long way.

Note that while **showing your steps & having loading states** have a mostly psychological effect, **streaming & chunking** genuinely do reduce overall latency once you consider the app + user system: the user will finish reading a response sooner.

## 7. Don't default to an LLM

LLMs are extremely powerful and versatile, and are therefore sometimes used in cases where a **faster classical method** would be more appropriate. Identifying such cases may allow you to cut your latency significantly. Consider the following examples:

- **Hard-coding:** If your **output** is highly constrained, you may not need an LLM to generate it. Action confirmations, refusal messages, and requests for standard input are all great candidates to be hard-coded. (You can even use the age-old method of coming up with a few variations for each.)
- **Pre-computing:** If your **input** is constrained (e.g. category selection) you can generate multiple responses in advance, and just make sure you never show the same one to a user twice.
- **Leveraging UI:** Summarized metrics, reports, or search results are sometimes better conveyed with classical, bespoke UI components rather than LLM-generated text.
- **Traditional optimization techniques:** An LLM application is still an application; binary search, caching, hash maps, and runtime complexity are all *still* useful in a world of LLMs.

## Example

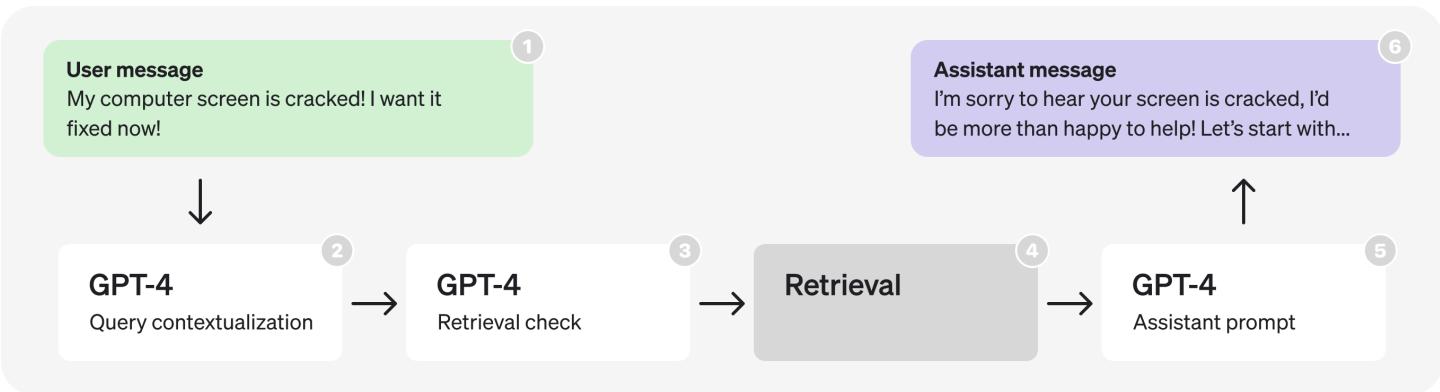
Let's now look at a sample application, identify potential latency optimizations, and propose some solutions!

We'll be analyzing the architecture and prompts of a hypothetical customer service bot inspired by real production applications. The [architecture and prompts](#) section sets the stage, and the [analysis and optimizations](#) section will walk through the latency optimization process.

 You'll notice this example doesn't cover every single principle, much like real-world use cases don't require applying every technique.

## Architecture and prompts

The following is the **initial architecture** for a hypothetical **customer service bot**. This is what we'll be making changes to.



At a high level, the diagram flow describes the following process:

- 1 A user sends a message as part of an ongoing conversation.
- 2 The last message is turned into a **self-contained query** (see examples in prompt).
- 3 We determine whether or not **additional (retrieved) information is required** to respond to that query.
- 4 **Retrieval** is performed, producing search results.
- 5 The assistant **reasons** about the user's query and search results, and **produces a response**.
- 6 The response is sent back to the user.

Below are the prompts used in each part of the diagram. While they are still only hypothetical and simplified, they are written with the same structure and wording that you would find in a production application.

(i) Places where you see placeholders like "**[user input here]**" represent dynamic portions, that would be replaced by actual data at runtime.

## ⟨ Query contextualization prompt

Re-writes user query to be a self-contained search query.

**SYSTEM** Given the previous conversation, re-write the last user query so it contains all necessary context.

# Example

History: [{user: "What is your return policy?"},{assistant: "..."}]

User Query: "How long does it cover?"

Response: "How long does the return policy cover?"

```
Conversation
[last 3 messages of conversation]

User Query
[last user query]

USER [JSON-formatted input conversation here]
```

## ⟨ Retrieval check prompt

Determines whether a query requires performing retrieval to respond.

```
SYSTEM Given a user query, determine whether it requires doing a realtime lookup to
respond to.

Examples
User Query: "How can I return this item after 30 days?"
Response: "true"

User Query: "Thank you!"
Response: "false"

USER [input user query here]
```

## ⟨ Assistant prompt

Fills the fields of a JSON to reason through a pre-defined set of steps to produce a final response given a user conversation and relevant retrieved information.

```
SYSTEM You are a helpful customer service bot.

Use the result JSON to reason about each user query - use the retrieved context.

Example

User: "My computer screen is cracked! I want it fixed now!!!!"
```

```

Assistant Response:
{
 "message_is_conversation_continuation": "True",
 "number_of_messages_in_conversation_so_far": "1",
 "user_sentiment": "Aggravated",
 "query_type": "Hardware Issue",
 "response_tone": "Validating and solution-oriented",
 "response_requirements": "Propose options for repair or replacement.",
 "user_requesting_to_talk_to_human": "False",
 "enough_information_in_context": "True"
 "response": "..."
}

USER # Relevant Information
```
[retrieved context]
```

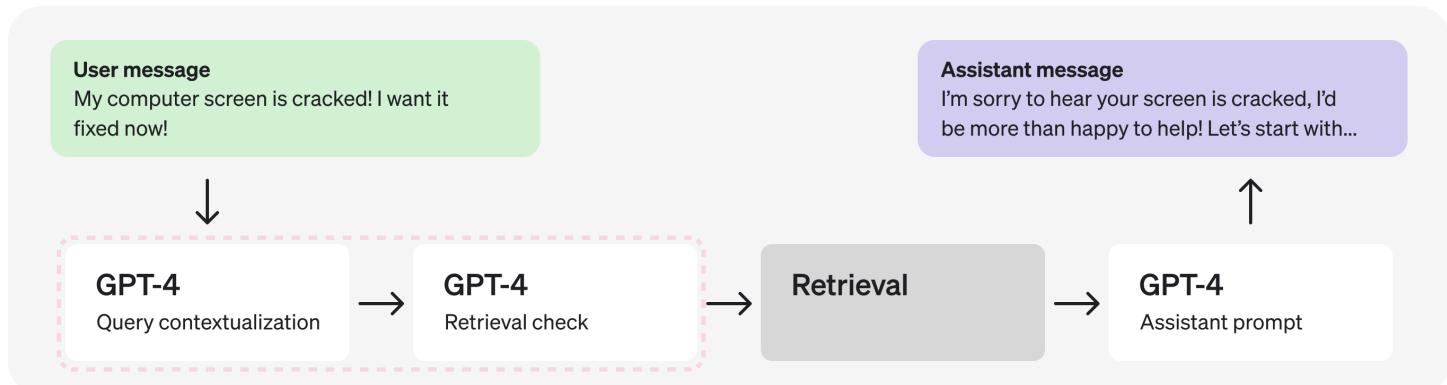
USER [input user query here]

```

## Analysis and optimizations

### Part 1: Looking at retrieval prompts

Looking at the architecture, the first thing that stands out is the **consecutive GPT-4 calls** - these hint at a potential inefficiency, and can often be replaced by a single call or parallel calls.



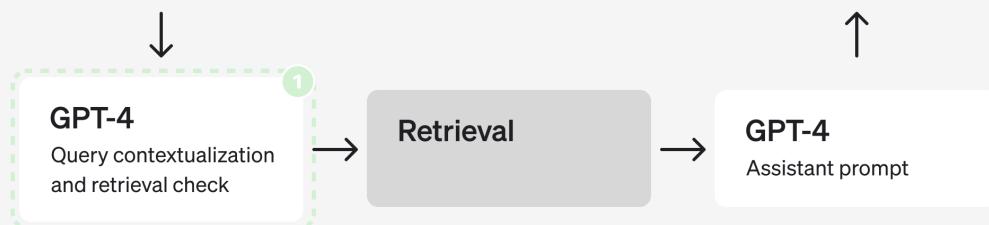
In this case, since the check for retrieval requires the contextualized query, let's **combine them into a single prompt** to **make fewer requests**.

**User message**

My computer screen is cracked! I want it fixed now!

**Assistant message**

I'm sorry to hear your screen is cracked, I'd be more than happy to help! Let's start with...



## < Combined query contextualization and retrieval check prompt

**What changed?** Before, we had one prompt to re-write the query and one to determine whether this requires doing a retrieval lookup. Now, this combined prompt does both. Specifically, notice the updated instruction in the first line of the prompt, and the updated output JSON:

```

1 {
2 query:"[contextualized query]",
3 retrieval:"[true/false - whether retrieval is required]"
4 }

```

**SYSTEM**

Given the previous conversation, re-write the last user query so it contains all necessary context. Then, determine whether the full request requires doing a realtime lookup to respond to.

Respond in the following form:

```
{
 query:"[contextualized query]",
 retrieval:"[true/false - whether retrieval is required]"
}
```

# Examples

History: [{user: "What is your return policy?"}, {assistant: "..."}]

User Query: "How long does it cover?"

Response: {query: "How long does the return policy cover?", retrieval: "true"}

History: [{user: "How can I return this item after 30 days?"}, {assistant: "..."}]

User Query: "Thank you!"

```
Response: {query: "Thank you!", retrieval: "false"}
```

```
Conversation
[last 3 messages of conversation]
```

```
User Query
[last user query]
```

USER [JSON-formatted input conversation here]

Actually, adding context and determining whether to retrieve are very straightforward and well defined tasks, so we can likely use a **smaller, fine-tuned model** instead. Switching to GPT-3.5 will let us **process tokens faster**.

```
graph LR; User["User message
My computer screen is cracked! I want it fixed now!"] --> GPT35["GPT-3.5
Query contextualization and retrieval check"]; GPT35 --> Retrieval["Retrieval"]; Retrieval --> GPT4["GPT-4
Assistant prompt"]; GPT4 --> Assistant["Assistant message
I'm sorry to hear your screen is cracked, I'd be more than happy to help! Let's start with..."]
```

## Part 2: Analyzing the assistant prompt

Let's now direct our attention to the Assistant prompt. There seem to be many distinct steps happening as it fills the JSON fields – this could indicate an opportunity to **parallelize**.

```
graph LR; User["User message
My computer screen is cracked! I want it fixed now!"] --> GPT35["GPT-3.5
Query contextualization and retrieval check"]; GPT35 --> Retrieval["Retrieval"]; Retrieval --> GPT4["GPT-4
Assistant prompt"]; GPT4 --> Assistant["Assistant message
I'm sorry to hear your screen is cracked, I'd be more than happy to help! Let's start with..."]
```

However, let's pretend we have run some tests and discovered that splitting the reasoning steps in the JSON produces worse responses, so we need to explore different solutions.

**Could we use a fine-tuned GPT-3.5 instead of GPT-4?** Maybe – but in general, open-ended responses from assistants are best left to GPT-4 so it can better handle a greater range of cases. That being said, looking at the reasoning steps themselves, they may not all require GPT-4 level reasoning to produce. The well defined, limited scope nature makes them and **good potential candidates for fine-tuning**.

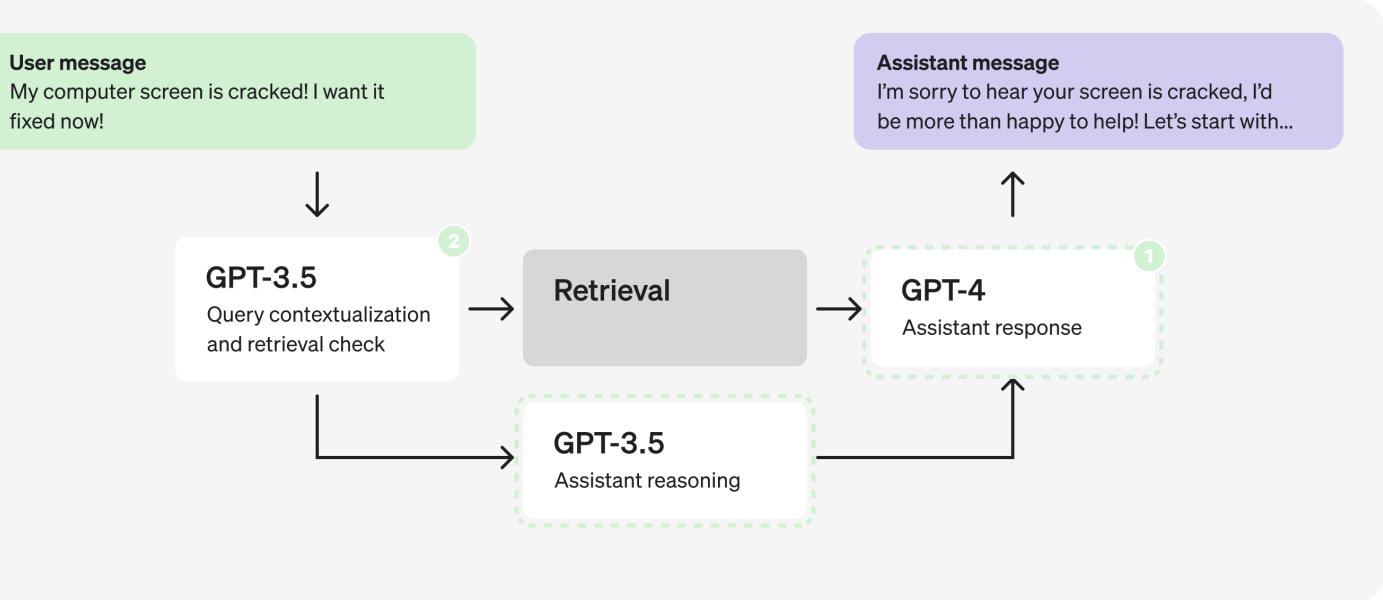
```
1 {
2 "message_is_conversation_continuation": "True", // <-
3 "number_of_messages_in_conversation_so_far": "1", // <-
4 "user_sentiment": "Aggravated", // <-
5 "query_type": "Hardware Issue", // <-
6 "response_tone": "Validating and solution-oriented", // <-
7 "response_requirements": "Propose options for repair or replacement.", // <-
8 "user_requesting_to_talk_to_human": "False", // <-
9 "enough_information_in_context": "True" // <-
10 "response": "..." // X -- benefits from GPT-4
11 }
```

This opens up the possibility of a trade-off. Do we keep this as a **single request entirely generated by GPT-4**, or **split it into two sequential requests** and use GPT-3.5 for all but the final response? We have a case of conflicting principles: the first option lets us **make fewer requests**, but the second may let us **process tokens faster**.

As with many optimization tradeoffs, the answer will depend on the details. For example:

- The proportion of tokens in the `response` vs the other fields.
- The average latency decrease from processing most fields faster.
- The average latency *increase* from doing two requests instead of one.

The conclusion will vary by case, and the best way to make the determination is by testing this with production examples. In this case let's pretend the tests indicated it's favorable to split the prompt in two to **process tokens faster**.



**Note:** We'll be grouping `response` and `enough_information_in_context` together in the second prompt to avoid passing the retrieved context to both new prompts.

## < Assistants prompt - reasoning

This prompt will be passed to GPT-3.5 and can be fine-tuned on curated examples.

**What changed?** The "enough\_information\_in\_context" and "response" fields were removed, and the retrieval results are no longer loaded into this prompt.

**SYSTEM** You are a helpful customer service bot.

Based on the previous conversation, respond in a JSON to determine the required fields.

# Example

User: "My freaking computer screen is cracked!"

Assistant Response:

```
{
 "message_is_conversation_continuation": "True",
 "number_of_messages_in_conversation_so_far": "1",
 "user_sentiment": "Aggravated",
 "query_type": "Hardware Issue",
 "response_tone": "Validating and solution-oriented",
 "response_requirements": "Propose options for repair or replacement."
}
```

```
"user_requesting_to_talk_to_human": "False",
}
```

## < Assistants prompt - response

This prompt will be processed by GPT-4 and will receive the reasoning steps determined in the prior prompt, as well as the results from retrieval.

**What changed?** All steps were removed except for "enough\_information\_in\_context" and "response". Additionally, the JSON we were previously filling in as output will be passed in to this prompt.

**SYSTEM** You are a helpful customer service bot.

Use the retrieved context, as well as these pre-classified fields, to respond to the user's query.

# Reasoning Fields  
` ` `

[reasoning json determined in previous GPT-3.5 call]  
` ` `

# Example

User: "My freaking computer screen is cracked!"

Assistant Response:

```
{
 "enough_information_in_context": "True"
 "response": "..."
}
```

**USER** # Relevant Information  
` ` `

[retrieved context]  
` ` `

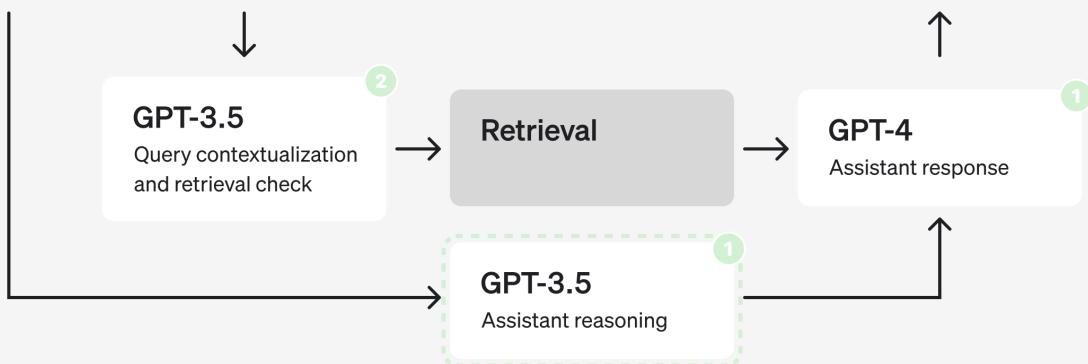
In fact, now that the reasoning prompt does not depend on the retrieved context we can [parallelize](#) and fire it off at the same time as the retrieval prompts.

**User message**

My computer screen is cracked! I want it fixed now!

**Assistant message**

I'm sorry to hear your screen is cracked, I'd be more than happy to help! Let's start with...

**Part 3: Optimizing the structured output**

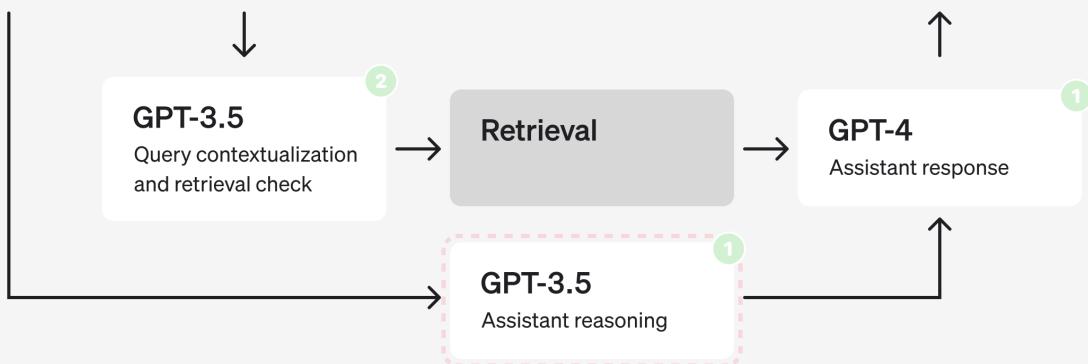
Let's take another look at the reasoning prompt.

**User message**

My computer screen is cracked! I want it fixed now!

**Assistant message**

I'm sorry to hear your screen is cracked, I'd be more than happy to help! Let's start with...



Taking a closer look at the reasoning JSON you may notice the field names themselves are quite long.

```

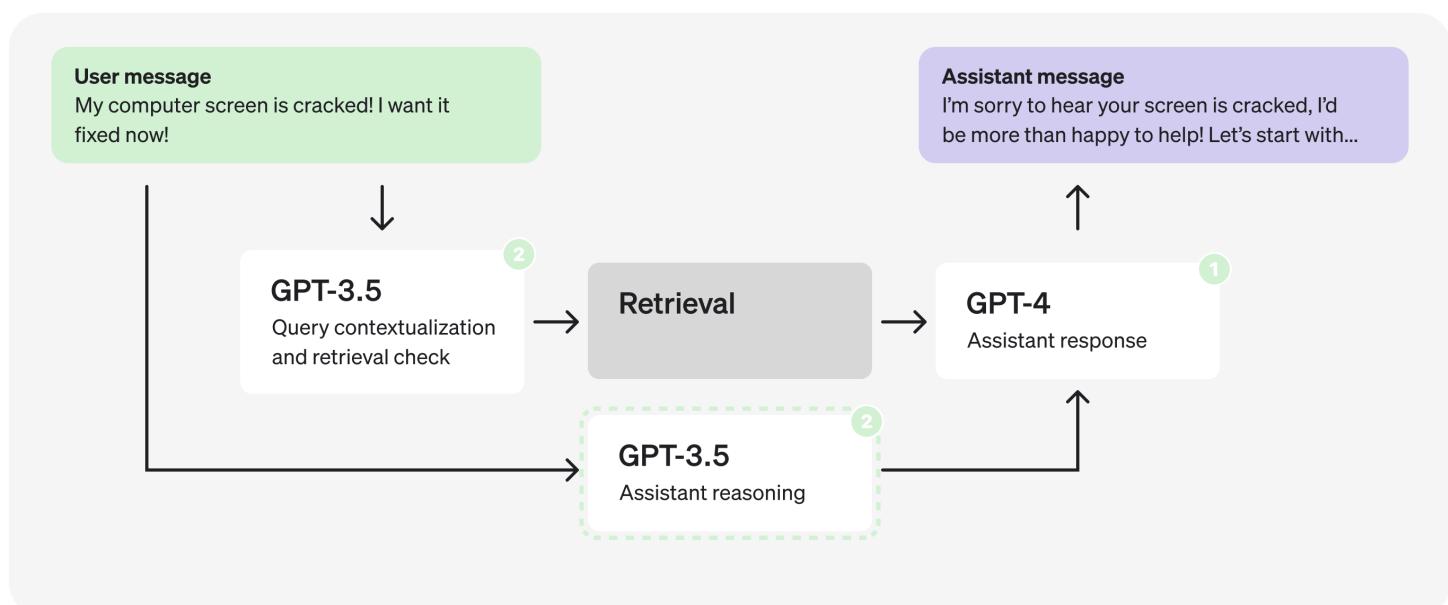
1 {
2 "message_is_conversation_continuation": "True", // <-
3 "number_of_messages_in_conversation_so_far": "1", // <-
4 "user_sentiment": "Aggravated", // <-
5 "query_type": "Hardware Issue", // <-
6 "response_tone": "Validating and solution-oriented", // <-
7 "response_requirements": "Propose options for repair or replacement.", // <-

```

```
8 "user_requesting_to_talk_to_human": "False", // <-
9 }
```

By making them shorter and moving explanations to the comments we can [generate fewer tokens](#).

```
1 {
2 "cont": "True", // whether last message is a continuation
3 "n_msg": "1", // number of messages in the continued conversation
4 "tone_in": "Aggravated", // sentiment of user query
5 "type": "Hardware Issue", // type of the user query
6 "tone_out": "Validating and solution-oriented", // desired tone for response
7 "reqs": "Propose options for repair or replacement.", // response requirement
8 "human": "False", // whether user is expressing want to talk to human
9 }
```



This small change removed 19 output tokens. While with GPT-3.5 this may only result in a few millisecond improvement, with GPT-4 this could shave off up to a second.

Tokens 87 Characters 342

```
{
 "message_is_conversation_continuation": "True",
 "number_of_messages_in_conversation_so_far": "1",
 "user_sentiment": "Aggravated",
 "query_type": "Hardware Issue",
 "response_tone": "Validating and solution-oriented",
 "response_requirements": "Propose options for repair or replacement.",
 "user_requesting_to_talk_to_human": "False",
}
}
```

Text Token IDs

Tokens 68 Characters 218

```
{
 "cont": "True",
 "n_msg": "1",
 "tone_in": "Aggravated",
 "type": "Hardware Issue",
 "tone_out": "Validating and solution-oriented",
 "reqs": "Propose options for repair or replacement.",
 "human": "False"
}
}
```

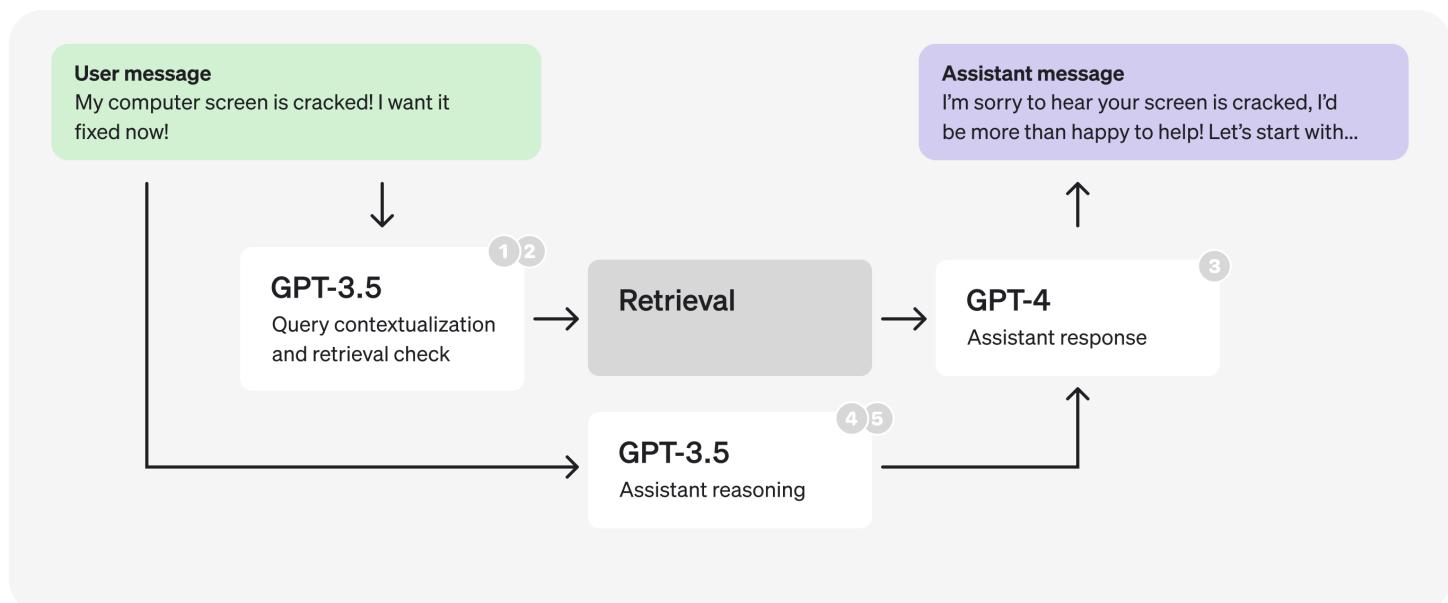
Text Token IDs

You might imagine, however, how this can have quite a significant impact for larger model outputs.

We could go further and use single characters for the JSON fields, or put everything in an array, but this may start to hurt our response quality. The best way to know, once again, is through testing.

## Example wrap-up

Let's review the optimizations we implemented for the customer service bot example:



- 1 **Combined** query contextualization and retrieval check steps to **make fewer requests**.
- 2 For the new prompt, **switched to a smaller, fine-tuned GPT-3.5** to **process tokens faster**.
- 3 Split the assistant prompt in two, **switching to a smaller, fine-tuned GPT-3.5** for the reasoning, again to **process tokens faster**.
- 4 **Parallelized** the retrieval checks and the reasoning steps.
- 5 **Shortened reasoning field names** and moved comments into the prompt, to **generate fewer tokens**.

## Conclusion

You should now be familiar with the core set of principles you can use to improve latency in your LLM application. As you explore these techniques, always remember to measure where your latency is coming from, and test the impact of each solution you try. Now go make your application *fly!*

# Rate limits

Rate limits are restrictions that our API imposes on the number of times a user or client can access our services within a specified period of time.

## Why do we have rate limits?

Rate limits are a common practice for APIs, and they're put in place for a few different reasons:

- **They help protect against abuse or misuse of the API.** For example, a malicious actor could flood the API with requests in an attempt to overload it or cause disruptions in service. By setting rate limits, OpenAI can prevent this kind of activity.
- **Rate limits help ensure that everyone has fair access to the API.** If one person or organization makes an excessive number of requests, it could bog down the API for everyone else. By throttling the number of requests that a single user can make, OpenAI ensures that the most number of people have an opportunity to use the API without experiencing slowdowns.
- **Rate limits can help OpenAI manage the aggregate load on its infrastructure.** If requests to the API increase dramatically, it could tax the servers and cause performance issues. By setting rate limits, OpenAI can help maintain a smooth and consistent experience for all users.

 Please work through this document in its entirety to better understand how OpenAI's rate limit system works. We include code examples and possible solutions to handle common issues. We also include details around how your rate limits are automatically increased in the usage tiers section below.

## How do these rate limits work?

Rate limits are measured in five ways: **RPM** (requests per minute), **RPD** (requests per day), **TPM** (tokens per minute), **TPD** (tokens per day), and **IPM** (images per minute). Rate limits can be hit across any of the options depending on what occurs first. For example, you might send 20 requests with only 100 tokens to the ChatCompletions endpoint and that would fill your limit (if your RPM was 20), even if you did not send 150k tokens (if your TPM limit was 150k) within those 20 requests.

**Batch API** queue limits are calculated based on the total number of input tokens queued for a given model. Tokens from pending batch jobs are counted against your queue limit. Once a batch job is completed, its tokens are no longer counted against that model's limit.

Other important things worth noting:

- Rate limits are defined at the [organization level](#) and at the project level, not user level.

- Rate limits vary by the [model](#) being used.
- Limits are also placed on the total amount an organization can spend on the API each month. These are also known as "usage limits".

## Usage tiers

You can view the rate and usage limits for your organization under the [limits](#) section of your account settings. As your usage of the OpenAI API and your spend on our API goes up, we automatically graduate you to the next usage tier. This usually results in an increase in rate limits across most models.

TIER	QUALIFICATION	USAGE LIMITS
Free	User must be in an <a href="#">allowed geography</a>	\$100 / month
Tier 1	\$5 paid	\$100 / month
Tier 2	\$50 paid and 7+ days since first successful payment	\$500 / month
Tier 3	\$100 paid and 7+ days since first successful payment	\$1,000 / month
Tier 4	\$250 paid and 14+ days since first successful payment	\$5,000 / month
Tier 5	\$1,000 paid and 30+ days since first successful payment	\$15,000 / month

Select a tier below to view a high-level summary of rate limits per model.

[Free](#)   [Tier 1](#)   [Tier 2](#)   [Tier 3](#)   [Tier 4](#)   [Tier 5](#)

### Free tier rate limits

This is a high level summary and there are per-model exceptions to these limits (e.g. some legacy models or models with larger context windows have different rate limits). To view the exact rate limits per model for your account, visit the [limits](#) section of your account settings.

MODEL	RPM	RPD	TPM	BATCH QUEUE LIMIT
gpt-3.5-turbo	3	200	40,000	200,000
text-embedding-3-large	3,000	200	1,000,000	3,000,000
text-embedding-3-small	3,000	200	1,000,000	3,000,000
text-embedding-ada-002	3,000	200	1,000,000	3,000,000

MODEL	RPM	RPD	TPM	BATCH QUEUE LIMIT
whisper-1	3	200	-	-
tts-1	3	200	-	-
dall-e-2	5 img/min	-	-	-
dall-e-3	1 img/min	-	-	-

## Rate limits in headers

In addition to seeing your rate limit on your [account page](#), you can also view important information about your rate limits such as the remaining requests, tokens, and other metadata in the headers of the HTTP response.

You can expect to see the following header fields:

FIELD	SAMPLE VALUE	DESCRIPTION
x-ratelimit-limit-requests	60	The maximum number of requests that are permitted before exhausting the rate limit.
x-ratelimit-limit-tokens	150000	The maximum number of tokens that are permitted before exhausting the rate limit.
x-ratelimit-remaining-requests	59	The remaining number of requests that are permitted before exhausting the rate limit.
x-ratelimit-remaining-tokens	149984	The remaining number of tokens that are permitted before exhausting the rate limit.
x-ratelimit-reset-requests	1s	The time until the rate limit (based on requests) resets to its initial state.
x-ratelimit-reset-tokens	6m0s	The time until the rate limit (based on tokens) resets to its initial state.

## Error Mitigation

### What are some steps I can take to mitigate this?

The OpenAI Cookbook has a [Python notebook](#) that explains how to avoid rate limit errors, as well an example [Python script](#) for staying under rate limits while batch processing API requests.

You should also exercise caution when providing programmatic access, bulk processing features, and automated social media posting - consider only enabling these for trusted customers.

To protect against automated and high-volume misuse, set a usage limit for individual users within a specified time frame (daily, weekly, or monthly). Consider implementing a hard cap or a manual review process for users who exceed the limit.

## Retrying with exponential backoff

One easy way to avoid rate limit errors is to automatically retry requests with a random exponential backoff. Retrying with exponential backoff means performing a short sleep when a rate limit error is hit, then retrying the unsuccessful request. If the request is still unsuccessful, the sleep length is increased and the process is repeated. This continues until the request is successful or until a maximum number of retries is reached. This approach has many benefits:

- Automatic retries means you can recover from rate limit errors without crashes or missing data
- Exponential backoff means that your first retries can be tried quickly, while still benefiting from longer delays if your first few retries fail
- Adding random jitter to the delay helps retries from all hitting at the same time.

Note that unsuccessful requests contribute to your per-minute limit, so continuously resending a request won't work.

Below are a few example solutions for **Python** that use exponential backoff.

### Example 1: Using the Tenacity library

Tenacity is an Apache 2.0 licensed general-purpose retrying library, written in Python, to simplify the task of adding retry behavior to just about anything. To add exponential backoff to your requests, you can use the `tenacity.retry` decorator. The below example uses the `tenacity.wait_random_exponential` function to add random exponential backoff to a request.

```
Using the Tenacity library python ▾ 
1 from openai import OpenAI
2 client = OpenAI()
3
4 from tenacity import (
5 retry,
6 stop_after_attempt,
7 wait_random_exponential,
```

```
8) # for exponential backoff
9
10 @retry(wait=wait_random_exponential(min=1, max=60), stop=stop_after_attempt(6))
11 def completion_with_backoff(**kwargs):
12 return client.completions.create(**kwargs)
13
14 completion_with_backoff(model="gpt-3.5-turbo-instruct", prompt="Once upon a time,")
```

Note that the Tenacity library is a third-party tool, and OpenAI makes no guarantees about its reliability or security.

## ⟨ Example 2: Using the backoff library

Another python library that provides function decorators for backoff and retry is [backoff](#):

Using the Tenacity library

python ▾ 

```
1 import backoff
2 import openai
3 from openai import OpenAI
4 client = OpenAI()
5
6 @backoff.on_exception(backoff.expo, openai.RateLimitError)
7 def completions_with_backoff(**kwargs):
8 return client.completions.create(**kwargs)
9
10 completions_with_backoff(model="gpt-3.5-turbo-instruct", prompt="Once upon a time,")
```

Like Tenacity, the backoff library is a third-party tool, and OpenAI makes no guarantees about its reliability or security.

## ⟨ Example 3: Manual backoff implementation

If you don't want to use third-party libraries, you can implement your own backoff logic following this example:

Using manual backoff implementation

python ▾ 

```
1 # imports
2 import random
```

```
3 import time
4
5 import openai
6 from openai import OpenAI
7 client = OpenAI()
8
9 # define a retry decorator
10 def retry_with_exponential_backoff(
11 func,
12 initial_delay: float = 1,
13 exponential_base: float = 2,
14 jitter: bool = True,
15 max_retries: int = 10,
16 errors: tuple = (openai.RateLimitError,),
17):
18 """Retry a function with exponential backoff."""
19
20 def wrapper(*args, **kwargs):
21 # Initialize variables
22 num_retries = 0
23 delay = initial_delay
24
25 # Loop until a successful response or max_retries is hit or an exception is rai
26 while True:
27 try:
28 return func(*args, **kwargs)
29
30 # Retry on specific errors
31 except errors as e:
32 # Increment retries
33 num_retries += 1
34
35 # Check if max retries has been reached
36 if num_retries > max_retries:
37 raise Exception(
38 f"Maximum number of retries ({max_retries}) exceeded."
39)
40
41 # Increment the delay
42 delay *= exponential_base * (1 + jitter * random.random())
43
44 # Sleep for the delay
45 time.sleep(delay)
46
47 # Raise exceptions for any errors not specified
48 except Exception as e:
49 raise e
50
```

```
51 return wrapper
52
53 @retry_with_exponential_backoff
54 def completions_with_backoff(**kwargs):
55 return client.completions.create(**kwargs)
```

Again, OpenAI makes no guarantees on the security or efficiency of this solution but it can be a good starting place for your own solution.

## Reduce the `max_tokens` to match the size of your completions

Your rate limit is calculated as the maximum of `max_tokens` and the estimated number of tokens based on the character count of your request. Try to set the `max_tokens` value as close to your expected response size as possible.

## Batching requests

If your use case does not require immediate responses, you can use the [Batch API](#) to more easily submit and execute large collections of requests without impacting your synchronous request rate limits.

For use cases that do require synchronous responses, the OpenAI API has separate limits for **requests per minute** and **tokens per minute**.

If you're hitting the limit on requests per minute but have available capacity on tokens per minute, you can increase your throughput by batching multiple tasks into each request. This will allow you to process more tokens per minute, especially with our smaller models.

Sending in a batch of prompts works exactly the same as a normal API call, except you pass in a list of strings to the `prompt` parameter instead of a single string.

### ◀ Example without batching

No batching

python ▾



```
1 from openai import OpenAI
2 client = OpenAI()
3
4 num_stories = 10
5 prompt = "Once upon a time,"
```

```
6
7 # serial example, with one story completion per request
8 for _ in range(num_stories):
9 response = client.completions.create(
10 model="curie",
11 prompt=prompt,
12 max_tokens=20,
13)
14 # print story
15 print(prompt + response.choices[0].text)
```

## ◀ Example with batching

Batching

python ▾ 

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 num_stories = 10
5 prompts = ["Once upon a time,"] * num_stories
6
7 # batched example, with 10 story completions per request
8 response = client.completions.create(
9 model="curie",
10 prompt=prompts,
11 max_tokens=20,
12)
13
14 # match completions to prompts by index
15 stories = [""] * len(prompts)
16 for choice in response.choices:
17 stories[choice.index] = prompts[choice.index] + choice.text
18
19 # print stories
20 for story in stories:
21 print(story)
```

- ⓘ Warning: the response object may not return completions in the order of the prompts, so always remember to match responses back to prompts using the index field.

# Batch API

Learn how to use OpenAI's Batch API to send asynchronous groups of requests with 50% lower costs, a separate pool of significantly higher rate limits, and a clear 24-hour turnaround time. The service is ideal for processing jobs that don't require immediate responses. You can also [explore the API reference directly here](#).

## Overview

While some uses of the OpenAI Platform require you to send synchronous requests, there are many cases where requests do not need an immediate response or [rate limits](#) prevent you from executing a large number of queries quickly. Batch processing jobs are often helpful in use cases like:

- 1 running evaluations
- 2 classifying large datasets
- 3 embedding content repositories

The Batch API offers a straightforward set of endpoints that allow you to collect a set of requests into a single file, kick off a batch processing job to execute these requests, query for the status of that batch while the underlying requests execute, and eventually retrieve the collected results when the batch is complete.

Compared to using standard endpoints directly, Batch API has:

- 1 **Better cost efficiency:** 50% cost discount compared to synchronous APIs
- 2 **Higher rate limits:** [Substantially more headroom](#) compared to the synchronous APIs
- 3 **Fast completion times:** Each batch completes within 24 hours (and often more quickly)

## Getting Started

### 1. Preparing Your Batch File

Batches start with a `.jsonl` file where each line contains the details of an individual request to the API. For now, the available endpoints are `/v1/chat/completions` ([Chat Completions API](#)) and `/v1/embeddings` ([Embeddings API](#)). For a given input file, the parameters in each line's `body` field are the same as the parameters for the underlying endpoint. Each request must include a unique `custom_id` value, which you can use to reference results after completion. Here's an example of an input file with 2 requests. Note that each input file can only include requests to a single model.

```
{"custom_id": "request-1", "method": "POST", "url": "/v1/chat/completions"}
{"custom_id": "request-2", "method": "POST", "url": "/v1/chat/completions", "body": "text": "Hello, world!"}
```

## 2. Uploading Your Batch Input File

Similar to our [Fine-tuning API](#), you must first upload your input file so that you can reference it correctly when kicking off batches. Upload your `.jsonl` file using the [Files API](#).

Upload files for Batch API

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 batch_input_file = client.files.create(
5 file=open("batchinput.jsonl", "rb"),
6 purpose="batch"
7)
```

## 3. Creating the Batch

Once you've successfully uploaded your input file, you can use the input File object's ID to create a batch. In this case, let's assume the file ID is `file-abc123`. For now, the completion window can only be set to `24h`. You can also provide custom metadata via an optional `metadata` parameter.

Create the Batch

python ▾

```
1 batch_input_file_id = batch_input_file.id
2
3 client.batches.create(
4 input_file_id=batch_input_file_id,
5 endpoint="/v1/chat/completions",
6 completion_window="24h",
7 metadata={
8 "description": "nightly eval job"
9 }
10)
```

This request will return a [Batch object](#) with metadata about your batch:

```
1 {
2 "id": "batch_abc123",
3 "object": "batch",
4 "endpoint": "/v1/chat/completions",
5 "errors": null,
6 "input_file_id": "file-abc123",
7 "completion_window": "24h",
8 "status": "validating",
9 "output_file_id": null,
10 "error_file_id": null,
11 "created_at": 1714508499,
12 "in_progress_at": null,
13 "expires_at": 1714536634,
14 "completed_at": null,
15 "failed_at": null,
16 "expired_at": null,
17 "request_counts": {
18 "total": 0,
19 "completed": 0,
20 "failed": 0
21 },
22 "metadata": null
23 }
```

## 4. Checking the Status of a Batch

You can check the status of a batch at any time, which will also return a Batch object.

Check the status of a batch

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.batches.retrieve("batch_abc123")
```

The status of a given Batch object can be any of the following:

STATUS	DESCRIPTION
validating	the input file is being validated before the batch can begin

STATUS	DESCRIPTION
failed	the input file has failed the validation process
in_progress	the input file was successfully validated and the batch is currently being run
finalizing	the batch has completed and the results are being prepared
completed	the batch has been completed and the results are ready
expired	the batch was not able to be completed within the 24-hour time window
cancelling	cancellation of the batch has been initiated
cancelled	the batch was cancelled

## 5. Retrieving the Results

Once the batch is complete, you can download the output by making a request against the [Files API](#) via the `output_file_id` field from the Batch object and writing it to a file on your machine, in this case `batch_output.jsonl`

Retrieving the batch results python ▾ Copy

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 content = client.files.content("file-xyz123")
```

The output `.jsonl` file will have one response line for every successful request line in the input file. Any failed requests in the batch will have their error information written to an error file that can be found via the batch's `error_file_id`.

- ⓘ Note that the output line order **may not match** the input line order. Instead of relying on order to process your results, use the `custom_id` field which will be present in each line of your output file and allow you to map requests in your input to results in your output.

```
{"id": "batch_req_123", "custom_id": "request-2", "response": {"status_code": 200, "content": "Success 1"}, "error_file_id": null}
{"id": "batch_req_456", "custom_id": "request-1", "response": {"status_code": 404, "content": "File not found"}, "error_file_id": null}
```

## 6. Cancelling a Batch

If necessary, you can cancel an ongoing batch. The batch's status will change to `cancelling` until in-flight requests are complete, after which the status will change to `cancelled`.

Cancelling a batch

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.batches.cancel("batch_abci23")
```

## 7. Getting a List of All Batches

At any time, you can see all your batches. For users with many batches, you can use the `limit` and `after` parameters to paginate your results.

Getting a list of all batches

python ▾

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 client.batches.list(limit=10)
```

## Model Availability

The Batch API can currently be used to execute queries against the following models. The Batch API supports text and vision inputs in the same format as the endpoints for these models:

- `gpt-3.5-turbo`
- `gpt-3.5-turbo-16k`
- `gpt-4`
- `gpt-4-32k`
- `gpt-4-turbo-preview`
- `gpt-4-vision-preview`
- `gpt-4-turbo`
- `gpt-3.5-turbo-0301`
- `gpt-3.5-turbo-16k-0613`

- gpt-3.5-turbo-1106
- gpt-3.5-turbo-0613
- gpt-4-0314
- gpt-4-turbo-2024-04-09
- gpt-4-32k-0314
- gpt-4-32k-0613
- text-embedding-3-large
- text-embedding-3-small
- text-embedding-ada-002

The Batch API also supports [fine-tuned models](#).

## Rate Limits

Batch API rate limits are separate from existing per-model rate limits. The Batch API has two new types of rate limits:

- 1 **Requests per batch:** You may include up to 50,000 requests in a single batch.
- 2 **Enqueued prompt tokens per model:** Each model has a maximum number of enqueued prompt tokens allowed for batch processing. You can find these limits on the [Platform Settings page](#).

There are no limits for output tokens or number of submitted requests for the Batch API today. Because Batch API rate limits are a new, separate pool, **using the Batch API will not consume tokens from your standard per-model rate limits**, thereby offering you a convenient way to increase the number of requests and processed tokens you can use when querying our API.

## Batch Expiration

Batches that do not complete in time eventually move to an `expired` state; unfinished requests within that batch are cancelled, and any responses to completed requests are made available via the batch's output file. You will be charged for tokens consumed from any completed requests.

## Other Resources

For more concrete examples, visit [the OpenAI Cookbook](#), which contains sample code for use cases like classification, sentiment analysis, and summary generation.

# Error codes

This guide includes an overview on error codes you might see from both the [API](#) and our [official Python library](#). Each error code mentioned in the overview has a dedicated section with further guidance.

## API errors

CODE	OVERVIEW
401 - Invalid Authentication	<b>Cause:</b> Invalid Authentication <b>Solution:</b> Ensure the correct <a href="#">API key</a> and requesting organization are being used.
401 - Incorrect API key provided	<b>Cause:</b> The requesting API key is not correct. <b>Solution:</b> Ensure the API key used is correct, clear your browser cache, or <a href="#">generate a new one</a> .
401 - You must be a member of an organization to use the API	<b>Cause:</b> Your account is not part of an organization. <b>Solution:</b> Contact us to get added to a new organization or ask your organization manager to <a href="#">invite you to an organization</a> .
403 - Country, region, or territory not supported	<b>Cause:</b> You are accessing the API from an unsupported country, region, or territory. <b>Solution:</b> Please see <a href="#">this page</a> for more information.
429 - Rate limit reached for requests	<b>Cause:</b> You are sending requests too quickly. <b>Solution:</b> Pace your requests. Read the <a href="#">Rate limit guide</a> .
429 - You exceeded your current quota, please check your plan and billing details	<b>Cause:</b> You have run out of credits or hit your maximum monthly spend. <b>Solution:</b> <a href="#">Buy more credits</a> or learn how to <a href="#">increase your limits</a> .
500 - The server had an error while processing your request	<b>Cause:</b> Issue on our servers. <b>Solution:</b> Retry your request after a brief wait and contact us if the issue persists. Check the <a href="#">status page</a> .
503 - The engine is currently overloaded, please try again later	<b>Cause:</b> Our servers are experiencing high traffic. <b>Solution:</b> Please retry your requests after a brief wait.

⟨ [401 - Invalid Authentication](#)

This error message indicates that your authentication credentials are invalid. This could happen for several reasons, such as:

- You are using a revoked API key.
- You are using a different API key than the one assigned to the requesting organization or project.
- You are using an API key that does not have the required permissions for the endpoint you are calling.

To resolve this error, please follow these steps:

- Check that you are using the correct API key and organization ID in your request header. You can find your API key and organization ID in [your account settings](#) or you can find specific project related keys under [General settings](#) by selecting the desired project.
  - If you are unsure whether your API key is valid, you can [generate a new one](#). Make sure to replace your old API key with the new one in your requests and follow our [best practices guide](#).
- 

## < 401 - Incorrect API key provided

This error message indicates that the API key you are using in your request is not correct. This could happen for several reasons, such as:

- There is a typo or an extra space in your API key.
- You are using an API key that belongs to a different organization or project.
- You are using an API key that has been deleted or deactivated.
- An old, revoked API key might be cached locally.

To resolve this error, please follow these steps:

- Try clearing your browser's cache and cookies, then try again.
  - Check that you are using the correct API key in your request header.
  - If you are unsure whether your API key is correct, you can [generate a new one](#). Make sure to replace your old API key in your codebase and follow our [best practices guide](#).
- 

## < 401 - You must be a member of an organization to use the API

This error message indicates that your account is not part of an organization. This could happen for several reasons, such as:

- You have left or been removed from your previous organization.
- You have left or been removed from your previous project.
- Your organization has been deleted.

To resolve this error, please follow these steps:

- If you have left or been removed from your previous organization, you can either request a new organization or get invited to an existing one.
  - To request a new organization, reach out to us via [help.openai.com](https://help.openai.com)
  - Existing organization owners can invite you to join their organization via the [Team page](#) or can create a new project from the [Settings page](#)
  - If you have left or been removed from a previous project, you can ask your organization or project owner to add you to it, or create a new one.
- 

## ⟨ 429 - Rate limit reached for requests

This error message indicates that you have hit your assigned rate limit for the API. This means that you have submitted too many tokens or requests in a short period of time and have exceeded the number of requests allowed. This could happen for several reasons, such as:

- You are using a loop or a script that makes frequent or concurrent requests.
- You are sharing your API key with other users or applications.
- You are using a free plan that has a low rate limit.
- You have reached the defined limit on your project

To resolve this error, please follow these steps:

- Pace your requests and avoid making unnecessary or redundant calls.
- If you are using a loop or a script, make sure to implement a backoff mechanism or a retry logic that respects the rate limit and the response headers. You can read more about our rate limiting policy and best practices in our [rate limit guide](#).
- If you are sharing your organization with other users, note that limits are applied per organization and not per user. It is worth checking on the usage of the rest of your team as this will contribute to the limit.
- If you are using a free or low-tier plan, consider upgrading to a pay-as-you-go plan that offers a higher rate limit. You can compare the restrictions of each plan in our [rate limit guide](#).

- Reach out to your organization owner to increase the rate limits on your project
- 

## 429 - You exceeded your current quota, please check your plan and billing details

This error message indicates that you hit your monthly [usage limit](/account/limits) for the API, or for prepaid credits customers that you've consumed all your credits. You can view your maximum usage limit on the [limits page](/account/limits). This could happen for several reasons, such as:

- You are using a high-volume or complex service that consumes a lot of credits or tokens.
- Your monthly budget is set too low for your organization's usage.
- Your monthly budget is set too low for your project's usage.

To resolve this error, please follow these steps:

- Check your [current usage](#) of your account, and compare that to your account's [limits](#).
  - If you are on a free plan, consider [upgrading to a paid plan](#) to get higher limits.
  - Reach out to your organization owner to increase the budgets for your project.
- 

## 503 - The engine is currently overloaded, please try again later

This error message indicates that our servers are experiencing high traffic and are unable to process your request at the moment. This could happen for several reasons, such as:

- There is a sudden spike or surge in demand for our services.
- There is scheduled or unscheduled maintenance or update on our servers.
- There is an unexpected or unavoidable outage or incident on our servers.

To resolve this error, please follow these steps:

- Retry your request after a brief wait. We recommend using an exponential backoff strategy or a retry logic that respects the response headers and the rate limit. You can read more about our rate limit [best practices](#).
- Check our [status page](#) for any updates or announcements regarding our services and servers.

- If you are still getting this error after a reasonable amount of time, please contact us for further assistance. We apologize for any inconvenience and appreciate your patience and understanding.
- 

## Python library error types

TYPE	OVERVIEW
APIConnectionError	<b>Cause:</b> Issue connecting to our services. <b>Solution:</b> Check your network settings, proxy configuration, SSL certificates, or firewall rules.
APITimeoutError	<b>Cause:</b> Request timed out. <b>Solution:</b> Retry your request after a brief wait and contact us if the issue persists.
AuthenticationError	<b>Cause:</b> Your API key or token was invalid, expired, or revoked. <b>Solution:</b> Check your API key or token and make sure it is correct and active. You may need to generate a new one from your account dashboard.
BadRequestError	<b>Cause:</b> Your request was malformed or missing some required parameters, such as a token or an input. <b>Solution:</b> The error message should advise you on the specific error made. Check the <a href="#">documentation</a> for the specific API method you are calling and make sure you are sending valid and complete parameters. You may also need to check the encoding, format, or size of your request data.
ConflictError	<b>Cause:</b> The resource was updated by another request. <b>Solution:</b> Try to update the resource again and ensure no other requests are trying to update it.
InternalServerError	<b>Cause:</b> Issue on our side. <b>Solution:</b> Retry your request after a brief wait and contact us if the issue persists.
NotFoundError	<b>Cause:</b> Requested resource does not exist. <b>Solution:</b> Ensure you are using the correct resource identifier.
PermissionDeniedError	<b>Cause:</b> You don't have access to the requested resource. <b>Solution:</b> Ensure you are using the correct API key, organization ID, and resource ID.

TYPE	OVERVIEW
RateLimitError	<b>Cause:</b> You have hit your assigned rate limit. <b>Solution:</b> Pace your requests. Read more in our <a href="#">Rate limit guide</a> .
UnprocessableEntityError	<b>Cause:</b> Unable to process the request despite the format being correct. <b>Solution:</b> Please try the request again.

## ⟨ APIConnectionError

An `APIConnectionError` indicates that your request could not reach our servers or establish a secure connection. This could be due to a network issue, a proxy configuration, an SSL certificate, or a firewall rule.

If you encounter an `APIConnectionError`, please try the following steps:

- Check your network settings and make sure you have a stable and fast internet connection. You may need to switch to a different network, use a wired connection, or reduce the number of devices or applications using your bandwidth.
- Check your proxy configuration and make sure it is compatible with our services. You may need to update your proxy settings, use a different proxy, or bypass the proxy altogether.
- Check your SSL certificates and make sure they are valid and up-to-date. You may need to install or renew your certificates, use a different certificate authority, or disable SSL verification.
- Check your firewall rules and make sure they are not blocking or filtering our services. You may need to modify your firewall settings.
- If appropriate, check that your container has the correct permissions to send and receive traffic.
- If the issue persists, check out our persistent errors next steps section.

## ⟨ APITimeoutError

A `APITimeoutError` error indicates that your request took too long to complete and our server closed the connection. This could be due to a network issue, a heavy load on our services, or a complex request that requires more processing time.

If you encounter a `APITimeoutError` error, please try the following steps:

- Wait a few seconds and retry your request. Sometimes, the network congestion or the load on our services may be reduced and your request may succeed on the second attempt.
- Check your network settings and make sure you have a stable and fast internet connection. You may need to switch to a different network, use a wired connection, or reduce the number of devices or applications using your bandwidth.

- If the issue persists, check out our persistent errors next steps section.
- 

## < AuthenticationError

An `AuthenticationError` indicates that your API key or token was invalid, expired, or revoked. This could be due to a typo, a formatting error, or a security breach.

If you encounter an `AuthenticationError`, please try the following steps:

- Check your API key or token and make sure it is correct and active. You may need to generate a new key from the API Key dashboard, ensure there are no extra spaces or characters, or use a different key or token if you have multiple ones.
  - Ensure that you have followed the correct formatting.
- 

## < BadRequestError

An `BadRequestError` (formerly `InvalidRequestError`) indicates that your request was malformed or missing some required parameters, such as a token or an input. This could be due to a typo, a formatting error, or a logic error in your code.

If you encounter an `BadRequestError`, please try the following steps:

- Read the error message carefully and identify the specific error made. The error message should advise you on what parameter was invalid or missing, and what value or format was expected.
  - Check the [API Reference](#) for the specific API method you were calling and make sure you are sending valid and complete parameters. You may need to review the parameter names, types, values, and formats, and ensure they match the documentation.
  - Check the encoding, format, or size of your request data and make sure they are compatible with our services. You may need to encode your data in UTF-8, format your data in JSON, or compress your data if it is too large.
  - Test your request using a tool like Postman or curl and make sure it works as expected. You may need to debug your code and fix any errors or inconsistencies in your request logic.
  - If the issue persists, check out our persistent errors next steps section.
- 

## < InternalServerError

An `InternalServerError` indicates that something went wrong on our side when processing your request. This could be due to a temporary error, a bug, or a system outage. We apologize for any inconvenience and we are working hard to resolve any issues as soon as possible. You can [check our system status page](#) for more information.

If you encounter an `InternalServerError`, please try the following steps:

- Wait a few seconds and retry your request. Sometimes, the issue may be resolved quickly and your request may succeed on the second attempt.
- Check our status page for any ongoing incidents or maintenance that may affect our services. If there is an active incident, please follow the updates and wait until it is resolved before retrying your request.
- If the issue persists, check out our Persistent errors next steps section.

Our support team will investigate the issue and get back to you as soon as possible. Note that our support queue times may be long due to high demand. You can also [post in our Community Forum](#) but be sure to omit any sensitive information.

---

## ⟨ RateLimitError

A `RateLimitError` indicates that you have hit your assigned rate limit. This means that you have sent too many tokens or requests in a given period of time, and our services have temporarily blocked you from sending more.

We impose rate limits to ensure fair and efficient use of our resources and to prevent abuse or overload of our services.

If you encounter a `RateLimitError`, please try the following steps:

- Send fewer tokens or requests or slow down. You may need to reduce the frequency or volume of your requests, batch your tokens, or implement exponential backoff. You can read our [Rate limit guide](#) for more details.
  - Wait until your rate limit resets (one minute) and retry your request. The error message should give you a sense of your usage rate and permitted usage.
  - You can also check your API usage statistics from your account dashboard.
- 

## Persistent errors

If the issue persists, [contact our support team via chat](#) and provide them with the following information:

- The model you were using
- The error message and code you received
- The request data and headers you sent
- The timestamp and timezone of your request
- Any other relevant details that may help us diagnose the issue

Our support team will investigate the issue and get back to you as soon as possible. Note that our support queue times may be long due to high demand. You can also [post in our Community Forum](#) but be sure to omit any sensitive information.

## Handling errors

We advise you to programmatically handle errors returned by the API. To do so, you may want to use a code snippet like below:

```
1 import openai
2 from openai import OpenAI
3 client = OpenAI()
4
5 try:
6 #Make your OpenAI API request here
7 response = client.completions.create(
8 prompt="Hello world",
9 model="gpt-3.5-turbo-instruct"
10)
11 except openai.APIError as e:
12 #Handle API error here, e.g. retry or log
13 print(f"OpenAI API returned an API Error: {e}")
14 pass
15 except openai.APIConnectionError as e:
16 #Handle connection error here
17 print(f"Failed to connect to OpenAI API: {e}")
18 pass
19 except openai.RateLimitError as e:
20 #Handle rate limit error (we recommend using exponential backoff)
21 print(f"OpenAI API request exceeded rate limit: {e}")
22 pass
```

# Libraries

## Python library

We provide a [Python library](#), which you can install by running:

```
pip install openai
```



Once installed, you can use the library and your secret key to run the following:

```
1 from openai import OpenAI
2 client = OpenAI(
3 # Defaults to os.environ.get("OPENAI_API_KEY")
4)
5
6 chat_completion = client.chat.completions.create(
7 model="gpt-3.5-turbo",
8 messages=[{"role": "user", "content": "Hello world"}]
9)
```



The bindings also will install a command-line utility you can use as follows:

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



## TypeScript / JavaScript library

We provide a [TypeScript / JavaScript library](#) with support for Node.js and various [other runtimes](#). Install it by running:

```
1 npm install --save openai
2 # or
3 yarn add openai
```



Once installed, you can use the library and your secret key to run the following:

```
1 import OpenAI from "openai";
2
3 const openai = new OpenAI({
4 apiKey: process.env.OPENAI_API_KEY,
5 });
6
7 const chatCompletion = await openai.chat.completions.create({
8 messages: [{ role: "user", content: "Say this is a test" }],
9 model: "gpt-3.5-turbo",
10});
```



## Azure OpenAI libraries

Microsoft's Azure team maintains libraries that are compatible with both the OpenAI API and Azure OpenAI services. Read the library documentation below to learn how you can use them with the OpenAI API.

- [Azure OpenAI client library for .NET](#)
- [Azure OpenAI client library for JavaScript](#)
- [Azure OpenAI client library for Java](#)
- [Azure OpenAI client library for Go](#)

## Community libraries

The libraries below are built and maintained by the broader developer community. If you'd like to add a new library here, please follow the instructions in our [help center article](#) on adding community libraries. You can also [watch our OpenAPI specification](#) repository on GitHub to get timely updates on when we make changes to our API.

Please note that OpenAI does not verify the correctness or security of these projects. **Use them at your own risk!**

### C# / .NET

- [Betalgo.OpenAI](#) by [Betalgo](#)
- [OpenAI-API-dotnet](#) by [OkGoDolt](#)
- [OpenAI-DotNet](#) by [RageAgainstThePixel](#)

## C++

- [liboai](#) by [D7EAD](#)

## Clojure

- [openai-clojure](#) by [wkok](#)

## Crystal

- [openai-crystal](#) by [sferik](#)

## Dart/Flutter

- [openai](#) by [anasfik](#)

## Delphi

- [DelphiOpenAI](#) by [HemulGM](#)

## Elixir

- [openai.ex](#) by [mgallo](#)

## Go

- [go-gpt3](#) by [sashabaranov](#)

## Java

- [openai-java](#) by [Theo Kanning](#)

## Julia

- [OpenAI.jl](#) by [rory-linehan](#)

## Kotlin

- [openai-kotlin](#) by [Mouaad Aallam](#)

## Node.js

- [openai-api](#) by Njerschow
- [openai-api-node](#) by erlapso
- [gpt-x](#) by ceifa
- [gpt3](#) by poteat
- [gpts](#) by thencc
- [@dalenguyen/openai](#) by dalenguyen
- [tectalic/openai](#) by tectalic

## PHP

- [orhanerday/open-ai](#) by orhanerday
- [tectalic/openai](#) by tectalic
- [openai-php client](#) by openai-php

## Python

- [chronology](#) by OthersideAI

## R

- [rgpt3](#) by ben-aaron188

## Ruby

- [openai](#) by nileshtrivedi
- [ruby-openai](#) by alexrudall

## Rust

- [async-openai](#) by 64bit
- [fieri](#) by lbkolev

## Scala

- [openai-scala-client](#) by cequence-io

## **Swift**

- [OpenAIKit](#) by dylanshine
- [OpenAI](#) by MacPaw

## **Unity**

- [OpenAi-Api-Unity](#) by hexthedev
- [com.openai.unity](#) by RageAgainstThePixel

## **Unreal Engine**

- [OpenAI-Api-Unreal](#) by KellanM

# Deprecations

## Overview

As we launch safer and more capable models, we regularly retire older models. Software relying on OpenAI models may need occasional updates to keep working. Impacted customers will always be notified by email and in our documentation along with [blog posts](#) for larger changes.

This page lists all API deprecations, along with recommended replacements.

## Deprecation vs Legacy

We use the term "deprecation" to refer to the process of retiring a model or endpoint. When we announce that a model or endpoint is being deprecated, it immediately becomes deprecated. All deprecated models and endpoints will also have a shut down date. At the time of the shut down, the model or endpoint will no longer be accessible.

We use the term "legacy" to refer to models and endpoints that will no longer receive updates. We tag endpoints and models as legacy to signal to developers where we are moving as a platform and that they should likely migrate to newer models or endpoints. You can expect that a legacy model or endpoint will be deprecated at some point in the future.

## Incremental model updates

As [announced](#) in March 2023, we regularly release new versions of `gpt-4` and `gpt-3.5-turbo`.

Each model version is dated with an `-MMDD` suffix; e.g., `gpt-4-0613`. The undated model name, e.g., `gpt-4`, will typically point to the latest version (e.g. `gpt-4` points to `gpt-4-0613`). Users of undated model names will be notified by email typically 2 weeks before any change takes place.

After a new version is launched, older versions will typically be deprecated 3 months later.

## Migrating to replacements

Once a model is deprecated, be sure to migrate all usage to a suitable replacement before the shutdown date. Requests to models past the shutdown date will fail.

To help measure the performance of replacement models on your tasks, we have open-sourced [Evals](#), a Python framework for evaluating language models.

If new models perform worse on your tasks, let us know by submitting a [pull request](#) to our Evals repo with examples of the task.

## Deprecation history

All deprecations are listed below, with the most recent announcements at the top.

### 2023-11-06: Chat model updates

On November 6th, 2023, we [announced](#) the release of an updated GPT-3.5-Turbo model (which now comes by default with 16k context) along with deprecation of `gpt-3.5-turbo-0613` and `gpt-3.5-turbo-16k-0613`.

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
2024-06-13	<code>gpt-3.5-turbo-0613</code>	\$0.0015 / 1K input tokens + \$0.0020 / 1K output tokens	<code>gpt-3.5-turbo-1106</code>
2024-06-13	<code>gpt-3.5-turbo-16k-0613</code>	\$0.0030 / 1K input tokens + \$0.0040 / 1K output tokens	<code>gpt-3.5-turbo-1106</code>

Fine-tuned models created from these base models are not effected by this deprecation, but you will no longer be able to create new fine-tuned versions with these models.

### 2023-08-22: Fine-tunes endpoint

On August 22nd, 2023, we [announced](#) the new fine-tuning API (`/v1/fine_tuning/jobs`) and that the original `/v1/fine-tunes` API along with legacy models (including those fine-tuned with the `/v1/fine-tunes` API) will be shut down on January 04, 2024. This means that models fine-tuned using the `/v1/fine-tunes` API will no longer be accessible and you would have to fine-tune new models with the updated endpoint and associated base models.

#### Fine-tunes endpoint

SHUTDOWN DATE	SYSTEM	RECOMMENDED REPLACEMENT
2024-01-04	<code>/v1/fine-tunes</code>	<code>/v1/fine_tuning/jobs</code>

### 2023-07-06: GPT and embeddings

On July 06, 2023, we [announced](#) the upcoming retirements of older GPT-3 and GPT-3.5 models served via the completions endpoint. We also announced the upcoming retirement of our first-generation text embedding models. They will be shut down on January 04, 2024.

## InstructGPT models

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
2024-01-04	text-ada-001	\$0.0004 / 1K tokens	gpt-3.5-turbo-instruct
2024-01-04	text-babbage-001	\$0.0005 / 1K tokens	gpt-3.5-turbo-instruct
2024-01-04	text-curie-001	\$0.0020 / 1K tokens	gpt-3.5-turbo-instruct
2024-01-04	text-davinci-001	\$0.0200 / 1K tokens	gpt-3.5-turbo-instruct
2024-01-04	text-davinci-002	\$0.0200 / 1K tokens	gpt-3.5-turbo-instruct
2024-01-04	text-davinci-003	\$0.0200 / 1K tokens	gpt-3.5-turbo-instruct

Pricing for the replacement `gpt-3.5-turbo-instruct` model can be found on the [pricing page](#).

## Base GPT models

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
2024-01-04	ada	\$0.0004 / 1K tokens	babbage-002
2024-01-04	babbage	\$0.0005 / 1K tokens	babbage-002
2024-01-04	curie	\$0.0020 / 1K tokens	davinci-002
2024-01-04	davinci	\$0.0200 / 1K tokens	davinci-002
2024-01-04	code-davinci-002	---	gpt-3.5-turbo-instruct

Pricing for the replacement `babbage-002` and `davinci-002` models can be found on the [pricing page](#).

## Edit models & endpoint

SHUTDOWN DATE	MODEL / SYSTEM	RECOMMENDED REPLACEMENT
2024-01-04	text-davinci-edit-001	gpt-4
2024-01-04	code-davinci-edit-001	gpt-4

SHUTDOWN DATE	MODEL / SYSTEM	RECOMMENDED REPLACEMENT
2024-01-04	/v1/edits	/v1/chat/completions

## Fine-tuning GPT models

SHUTDOWN DATE	LEGACY MODEL	TRAINING PRICE	USAGE PRICE	RECOMMENDED REPLACEMENT
2024-01-04	ada	\$0.0004 / 1K tokens	\$0.0016 / 1K tokens	babbage-002
2024-01-04	babbage	\$0.0006 / 1K tokens	\$0.0024 / 1K tokens	babbage-002
2024-01-04	curie	\$0.003 / 1K tokens	\$0.012 / 1K tokens	davinci-002
2024-01-04	davinci	\$0.03 / 1K tokens	\$0.12 / 1K tokens	davinci-002, gpt-3.5-turbo, gpt-4

## First-generation text embedding models

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
2024-01-04	text-similarity-ada-001	\$0.004 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-ada-doc-001	\$0.004 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-ada-query-001	\$0.004 / 1K tokens	text-embedding-ada-002
2024-01-04	code-search-ada-code-001	\$0.004 / 1K tokens	text-embedding-ada-002
2024-01-04	code-search-ada-text-001	\$0.004 / 1K tokens	text-embedding-ada-002
2024-01-04	text-similarity-babbage-001	\$0.005 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-babbage-doc-001	\$0.005 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-babbage-query-001	\$0.005 / 1K tokens	text-embedding-ada-002
2024-01-04	code-search-babbage-code-001	\$0.005 / 1K tokens	text-embedding-ada-002

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
2024-01-04	code-search-babbage-text-001	\$0.005 / 1K tokens	text-embedding-ada-002
2024-01-04	text-similarity-curie-001	\$0.020 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-curie-doc-001	\$0.020 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-curie-query-001	\$0.020 / 1K tokens	text-embedding-ada-002
2024-01-04	text-similarity-davinci-001	\$0.200 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-davinci-doc-001	\$0.200 / 1K tokens	text-embedding-ada-002
2024-01-04	text-search-davinci-query-001	\$0.200 / 1K tokens	text-embedding-ada-002

## 2023-06-13: Updated chat models

On June 13, 2023, we announced new chat model versions in the [Function calling and other API updates](#) blog post. The three original versions will be retired in June 2024 at the earliest. As of 01/10/2024, only existing users of this model will be able to continue using this model.

SHUTDOWN DATE	LEGACY MODEL	LEGACY MODEL PRICE	RECOMMENDED REPLACEMENT
at earliest 2024-06-13	gpt-3.5-turbo-0301	\$0.0015 / 1K input tokens + \$0.0020 / 1K output tokens	gpt-3.5-turbo-0613
at earliest 2024-06-13	gpt-4-0314	\$0.03 / 1K input tokens + \$0.06 / 1K output tokens	gpt-4-0613
at earliest 2024-06-13	gpt-4-32k-0314	\$0.06 / 1K input tokens + \$0.12 / 1K output tokens	gpt-4-32k-0613

## 2023-03-20: Codex models

SHUTDOWN DATE	LEGACY MODEL	RECOMMENDED REPLACEMENT
2023-03-23	code-davinci-002	gpt-4
2023-03-23	code-davinci-001	gpt-4

SHUTDOWN DATE	LEGACY MODEL	RECOMMENDED REPLACEMENT
2023-03-23	code-cushman-002	gpt-4
2023-03-23	code-cushman-001	gpt-4

## 2022-06-03: Legacy endpoints

SHUTDOWN DATE	SYSTEM	RECOMMENDED REPLACEMENT
2022-12-03	/v1/engines	<a href="#">/v1/models</a>
2022-12-03	/v1/search	<a href="#">View transition guide</a>
2022-12-03	/v1/classifications	<a href="#">View transition guide</a>
2022-12-03	/v1/answers	<a href="#">View transition guide</a>

# Actions in GPTs

## Introduction

Learn how to build a GPT action that intelligently calls your API.

## What is a GPT?

GPTs provide the ability to deeply customize ChatGPT for specific use cases along with custom capabilities. You can create a GPT that:

- Has custom instructions which determine the way the GPT interacts with users
- Includes tools like browsing, DALL-E, and Code Interpreter
- Comes with preset starter prompts for new and returning users
- Has custom actions which allow you to connect the GPT to APIs

And more! If you want to explore what is possible, check out the deep dive on GPTs from OpenAI Developer Day 2023:

## What is an action in a GPT?

In addition to using our built-in capabilities (browsing, DALL-E, and Code Interpreter), you can also define custom actions by making one or more APIs available to the GPT. Actions allow GPTs to integrate external data or interact with the real-world, such as connecting GPTs to databases, plugging them into your emails, or making them your shopping assistant, all through APIs.

The design of actions builds upon insights from our plugins beta, granting developers greater control over the model and how their APIs are called. Actions are defined using the [OpenAPI specification](#), which is a standard for describing APIs.

## GPT action flow

To build a GPT with an action, it is important to understand the end-to-end flow.

- 1 Create a GPT in the ChatGPT UI
  - Manually configure or use the GPT builder to create a GPT
  - Identify the API(s) you want to use
- 2 Go to the "Configure" tab in the GPT editor and select "Create new action"
  - You will be presented with 3 main options: selecting the authentication schema for the action, inputting the schema itself, and setting the privacy policy URL
  - The Schema follows the OpenAPI specification format (not to be confused with OpenAI) to define how the GPT can access an external API
- 3 Fill in the details for the schema, authentication, and privacy policy.
  - When selecting an authentication method, you will have 3 options, "None", "API Key", and "OAuth", we will explore these in depth later on
  - For the schema, you can take an existing OpenAPI specification you have for your API or create a new one. If you have already published an OpenAPI specification online, you can import it via the "Import from URL" button
  - The privacy policy URL is displayed to the user when they open a GPT and select the drop down in the top left corner showing the name of the GPT
- 4 Determine the visibility of your GPT
  - By default, GPTs are not accessible to everyone
  - When you go to save a GPT, you will have the option to "Publish to" a certain audience: "Only me", "Anyone with a link", or "Everyone"
    - Each of these visibility options comes with different constraints and requirements. For example the naming of a GPT has more restrictions if you share it with someone else
- 5 User(s) engage with your GPT
  - Depending on the visibility of your GPT, users might try it via a link you shared, or find it in the GPT store
  - If OAuth is required, users will be prompted to login during the session
  - Behind the scenes, the GPT injects the information on how you configured the GPT (including any available actions, tools, or instructions) into the context of the model

- Each time a user makes a request, the model sees the available tools, actions, and instructions which determine how the GPT will respond to the request
- If the user request is to check the weather in a specific location and you made a "Check weather" action available, the model will follow the OpenAPI specification you provided to send a request to that API and return the response to the user

## Next steps

Now that you know the basics of how a GPT works and where actions can be used, you might want to:

- Get started building a [GPT with an action](#)
- Learn how we [built the GPT builder](#) itself as a custom GPT with actions
- Familiarize yourself with our [GPT policies](#)
- Explore the [differences between GPTs and Assistants](#)
- Check out the [GPT data privacy FAQ's](#)
- Find answers to [common GPT questions](#)