# [GPT models](https://platform.openai.com/docs/guides/gpt/gpt-models)

Looking for ChatGPT? Head to [chat.openai.com](https://chat.openai.com/).

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

Using GPTs, you can build applications to:

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

...and much more!

To use a GPT model 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 and gpt-3.5-turbo, are accessed through the chat completions API endpoint. Currently, only the older legacy models are available via the completions API endpoint.

|  | **MODEL FAMILIES** | **API ENDPOINT** |
| --- | --- | --- |
| Newer models (2023–) | gpt-4, gpt-3.5-turbo | <https://api.openai.com/v1/chat/completions> |
| Legacy models (2020–2022) | text-davinci-003, text-davinci-002, davinci, curie, babbage, ada | <https://api.openai.com/v1/completions> |

You can experiment with GPTs in the [playground](https://platform.openai.com/playground?mode=chat). If you’re not sure which model to use, then use gpt-3.5-turbo or gpt-4.

## [Chat Completions API](https://platform.openai.com/docs/guides/gpt/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 API call looks as follows:

1

2

3

4

5

6

7

8

9

10

11

import openai

openai.ChatCompletion.create(

model="gpt-3.5-turbo",

messages=[

{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": "Who won the world series in 2020?"},

{"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."},

{"role": "user", "content": "Where was it played?"}

]

)

See the full API reference documentation [here](https://platform.openai.com/docs/api-reference/chat).

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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-for-dialogue-applications-that-require-very-long-conversations-summarize-or-filter-previous-dialogue) in some way.

To mimic the effect seen in ChatGPT where the text is returned iteratively, set the [stream](https://platform.openai.com/docs/api-reference/chat/create#chat/create-stream) parameter to true.

### [Chat completions response format](https://platform.openai.com/docs/guides/gpt/chat-completions-response-format)

An example chat completions API response looks as follows:

{

"choices": [

{

"finish\_reason": "stop",

"index": 0,

"message": {

"content": "The 2020 World Series was played in Texas at Globe Life Field in Arlington.",

"role": "assistant"

}

}

],

"created": 1677664795,

"id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",

"model": "gpt-3.5-turbo-0613",

"object": "chat.completion",

"usage": {

"completion\_tokens": 17,

"prompt\_tokens": 57,

"total\_tokens": 74

}

}

In Python, the assistant’s reply can be extracted with response['choices'][0]['message']['content'].

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](https://platform.openai.com/docs/api-reference/chat/create#chat/create-stop) parameter
* length: Incomplete model output due to [max\_tokens](https://platform.openai.com/docs/api-reference/chat/create#chat/create-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 (like providing functions as shown below), the model response may include different information.

## [Function calling](https://platform.openai.com/docs/guides/gpt/function-calling)

In an API call, you can describe functions to gpt-3.5-turbo-0613 and gpt-4-0613, and have the model intelligently choose to output a JSON object containing arguments to call those 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-0613 and gpt-4-0613) have been fine-tuned to both detect when a function should to be called (depending on the input) and to respond with JSON that adheres to the function signature. 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).

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.

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

* Create chatbots that answer questions by calling external APIs (e.g. like ChatGPT Plugins)
  + 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](https://platform.openai.com/docs/api-reference/chat/create#chat/create-functions).
2. The model can choose to call a function; if so, the content will be a stringified JSON object adhering to your custom schema (note: the model may generate invalid JSON or 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.

You can see these steps in action through the example below:

import openai

import json

# Example dummy function hard coded to return the same weather

# In production, this could be your backend API or an external API

def get\_current\_weather(location, unit="fahrenheit"):

"""Get the current weather in a given location"""

weather\_info = {

"location": location,

"temperature": "72",

"unit": unit,

"forecast": ["sunny", "windy"],

}

return json.dumps(weather\_info)

def run\_conversation():

# Step 1: send the conversation and available functions to GPT

messages = [{"role": "user", "content": "What's the weather like in Boston?"}]

functions = [

{

"name": "get\_current\_weather",

"description": "Get the current weather in a given location",

"parameters": {

"type": "object",

"properties": {

"location": {

"type": "string",

"description": "The city and state, e.g. San Francisco, CA",

},

"unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},

},

"required": ["location"],

},

}

]

response = openai.ChatCompletion.create(

model="gpt-3.5-turbo-0613",

messages=messages,

functions=functions,

function\_call="auto", # auto is default, but we'll be explicit

)

response\_message = response["choices"][0]["message"]

# Step 2: check if GPT wanted to call a function

if response\_message.get("function\_call"):

# Step 3: call the function

# Note: the JSON response may not always be valid; be sure to handle errors

available\_functions = {

"get\_current\_weather": get\_current\_weather,

} # only one function in this example, but you can have multiple

function\_name = response\_message["function\_call"]["name"]

fuction\_to\_call = available\_functions[function\_name]

function\_args = json.loads(response\_message["function\_call"]["arguments"])

function\_response = fuction\_to\_call(

location=function\_args.get("location"),

unit=function\_args.get("unit"),

)

# Step 4: send the info on the function call and function response to GPT

messages.append(response\_message) # extend conversation with assistant's reply

messages.append(

{

"role": "function",

"name": function\_name,

"content": function\_response,

}

) # extend conversation with function response

second\_response = openai.ChatCompletion.create(

model="gpt-3.5-turbo-0613",

messages=messages,

) # get a new response from GPT where it can see the function response

return second\_response

print(run\_conversation())

Hallucinated outputs in function calls can often be mitigated with a system message. For example, if you find that a model is generating function calls with functions that weren't provided to it, try using a system message that says: "Only use the functions you have been provided with."

In the example above, we sent the function response back to the model and let it decide the next step. It responded with a user-facing message which was telling the user the temperature in Boston, but depending on the query, it may choose to call a function again.

For example, if you ask the model “Find the weather in Boston this weekend, book dinner for two on Saturday, and update my calendar” and provide the corresponding functions for these queries, it may choose to call them back to back and only at the end create a user-facing message.

If you want to force the model to call a specific function you can do so by setting function\_call: {"name": "<insert-function-name>"}. You can also force the model to generate a user-facing message by setting function\_call: "none". Note that the default behavior (function\_call: "auto") is for the model to decide on its own whether to call a function and if so which function to call.

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

**[Function calling](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb" \t "_blank)**

[Learn from more examples demonstrating function calling](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb" \t "_blank)

## [Completions API](https://platform.openai.com/docs/guides/gpt/completions-api)

## [Legacy](https://platform.openai.com/docs/guides/gpt/completions-api)

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 API call looks as follows:

import openai

response = openai.Completion.create(

model="text-davinci-003",

prompt="Write a tagline for an ice cream shop."

)

See the full [API reference documentation](https://platform.openai.com/docs/api-reference/completions) to learn more.

#### [Token log probabilities](https://platform.openai.com/docs/guides/gpt/token-log-probabilities)

The completions API can provide a limited number of log probabilities associated with the most likely tokens for each output token. This feature is controlled by using the [logprobs](https://platform.openai.com/docs/api-reference/completions/create#completions/create-logprobs) field. This can be useful in some cases to assess the confidence of the model in its output.

#### [Inserting text](https://platform.openai.com/docs/guides/gpt/inserting-text)

The completions endpoint also supports inserting text by providing a [suffix](https://platform.openai.com/docs/api-reference/completions/create#completions/create-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](https://platform.openai.com/docs/guides/gpt/completions-response-format)

An example completions API response looks as follows:

{

"choices": [

{

"finish\_reason": "length",

"index": 0,

"logprobs": null,

"text": "\n\n\"Let Your Sweet Tooth Run Wild at Our Creamy Ice Cream Shack"

}

],

"created": 1683130927,

"id": "cmpl-7C9Wxi9Du4j1lQjdjhxBlO22M61LD",

"model": "text-davinci-003",

"object": "text\_completion",

"usage": {

"completion\_tokens": 16,

"prompt\_tokens": 10,

"total\_tokens": 26

}

}

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 but also includes the optional field logprobs.

## [Chat Completions vs. Completions](https://platform.openai.com/docs/guides/gpt/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 French: "{text}"'}]

Likewise, the completions API can be used to simulate a chat between a user and an assistant by formatting the input [accordingly](https://platform.openai.com/playground/p/default-chat?model=text-davinci-003).

The difference between these APIs derives mainly from the underlying GPT models that are available in each. The chat completions API is the interface to our most capable model (gpt-4), and our most cost effective model (gpt-3.5-turbo). For reference, gpt-3.5-turbo performs at a similar capability level to text-davinci-003 but at 10% the price per token! See pricing details [here](https://openai.com/pricing).

#### [Which model should I use?](https://platform.openai.com/docs/guides/gpt/which-model-should-i-use)

We generally recommend that you use either gpt-4 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 generally performs better on a wide range of [evaluations](https://arxiv.org/abs/2303.08774). In particular, gpt-4 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 is less likely than gpt-3.5-turbo to make up information, a behavior known as "hallucination". gpt-4 also has a larger context window with a maximum size of 8,192 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](https://platform.openai.com/playground?mode=chat) 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.

### [GPT best practices](https://platform.openai.com/docs/guides/gpt/gpt-best-practices)

An awareness of the best practices for working with GPTs can make a significant difference in application performance. The failure modes that GPTs exhibit and the ways of working around or correcting those failure modes are not always intuitive. There is a skill to working with GPTs 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 [GPT best practices](https://platform.openai.com/docs/guides/gpt-best-practices) 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](https://github.com/openai/openai-cookbook).

## [Managing tokens](https://platform.openai.com/docs/guides/gpt/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 (4096 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](https://openai.com/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 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](https://github.com/openai/tiktoken) Python library. Example code can be found in the OpenAI Cookbook’s guide on [how to count tokens with tiktoken](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb).

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 4096 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.

## [FAQ](https://platform.openai.com/docs/guides/gpt/faq)

### [Why are model outputs inconsistent?](https://platform.openai.com/docs/guides/gpt/why-are-model-outputs-inconsistent)

The API is non-deterministic by default. This means that you might get a slightly different completion every time you call it, even if your prompt stays the same. Setting temperature to 0 will make the outputs mostly deterministic, but a small amount of variability will remain.

### [How should I set the temperature parameter?](https://platform.openai.com/docs/guides/gpt/how-should-i-set-the-temperature-parameter)

Lower values for temperature result in more consistent outputs, while higher values generate more diverse and creative results. Select a temperature value based on the desired trade-off between coherence and creativity for your specific application.

### [Is fine-tuning available for the latest models?](https://platform.openai.com/docs/guides/gpt/is-fine-tuning-available-for-the-latest-models)

No. Currently, you can only fine-tune base GPT-3 models (davinci, curie, babbage, and ada). See the [fine-tuning guide](https://platform.openai.com/docs/guides/fine-tuning) for more details on how to use fine-tuned models.

### [Do you store the data that is passed into the API?](https://platform.openai.com/docs/guides/gpt/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](https://openai.com/policies/usage-policies). Some endpoints offer [zero retention](https://platform.openai.com/docs/models/default-usage-policies-by-endpoint).

### [How can I make my application more safe?](https://platform.openai.com/docs/guides/gpt/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](https://platform.openai.com/docs/guides/moderation) to prevent content that violates OpenAI’s usage policies from being shown.

### [Should I use ChatGPT or the API?](https://platform.openai.com/docs/guides/gpt/should-i-use-chatgpt-or-the-api)

[ChatGPT](https://chat.openai.com/) offers a chat interface to the models in the OpenAI API 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.

# [GPT best practices](https://platform.openai.com/docs/guides/gpt-best-practices/gpt-best-practices)

This guide shares strategies and tactics for getting better results from GPTs. 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. If you don't yet have access to gpt-4 consider joining the [waitlist](https://openai.com/waitlist/gpt-4-api). In general, if you find that a GPT model fails at a task and a more capable model is available, it's often worth trying again with the more capable model.

## [Six strategies for getting better results](https://platform.openai.com/docs/guides/gpt-best-practices/six-strategies-for-getting-better-results)

### [Write clear instructions](https://platform.openai.com/docs/guides/gpt-best-practices/write-clear-instructions)

GPTs 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 GPTs have to guess at what you want, the more likely you’ll get it.

Tactics:

* [Include details in your query to get more relevant answers](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-include-details-in-your-query-to-get-more-relevant-answers)
* [Ask the model to adopt a persona](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-ask-the-model-to-adopt-a-persona)
* [Use delimiters to clearly indicate distinct parts of the input](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-use-delimiters-to-clearly-indicate-distinct-parts-of-the-input)
* [Specify the steps required to complete a task](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-specify-the-steps-required-to-complete-a-task)
* [Provide examples](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-provide-examples)
* [Specify the desired length of the output](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-specify-the-desired-length-of-the-output)

### [Provide reference text](https://platform.openai.com/docs/guides/gpt-best-practices/provide-reference-text)

GPTs 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 GPTs can help in answering with fewer fabrications.

Tactics:

* [Instruct the model to answer using a reference text](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-instruct-the-model-to-answer-using-a-reference-text)
* [Instruct the model to answer with citations from a reference text](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-instruct-the-model-to-answer-with-citations-from-a-reference-text)

### [Split complex tasks into simpler subtasks](https://platform.openai.com/docs/guides/gpt-best-practices/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 GPTs. 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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-for-dialogue-applications-that-require-very-long-conversations-summarize-or-filter-previous-dialogue)
* [Summarize long documents piecewise and construct a full summary recursively](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-summarize-long-documents-piecewise-and-construct-a-full-summary-recursively)

### [Give GPTs time to "think"](https://platform.openai.com/docs/guides/gpt-best-practices/give-gpts-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, GPTs make more reasoning errors when trying to answer right away, rather than taking time to work out an answer. Asking for a chain of reasoning before an answer can help GPTs reason their way toward correct answers more reliably.

Tactics:

* [Instruct the model to work out its own solution before rushing to a conclusion](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-ask-the-model-if-it-missed-anything-on-previous-passes)

### [Use external tools](https://platform.openai.com/docs/guides/gpt-best-practices/use-external-tools)

Compensate for the weaknesses of GPTs by feeding them the outputs of other tools. For example, a text retrieval system can tell GPTs about relevant documents. A code execution engine can help GPTs do math and run code. If a task can be done more reliably or efficiently by a tool rather than by a GPT, offload it to get the best of both.

Tactics:

* [Use embeddings-based search to implement efficient knowledge retrieval](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-use-embeddings-based-search-to-implement-efficient-knowledge-retrieval)
* [Use code execution to perform more accurate calculations or call external APIs](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-use-code-execution-to-perform-more-accurate-calculations-or-call-external-apis)
* [Give the model access to specific functions](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-give-the-model-access-to-specific-functions)

### [Test changes systematically](https://platform.openai.com/docs/guides/gpt-best-practices/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 an as an "eval").

Tactic:

* [Evaluate model outputs with reference to gold-standard answers](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-evaluate-model-outputs-with-reference-to-gold-standard-answers)

## [Tactics](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/docs/guides/gpt-best-practices/strategy-write-clear-instructions)

#### [Tactic: Include details in your query to get more relevant answers](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-playful-thank-you-note)

#### [Tactic: Use delimiters to clearly indicate distinct parts of the input](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-delimiters-1)

**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](https://platform.openai.com/playground/p/default-delimiters-2)

**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](https://platform.openai.com/playground/p/default-delimiters-3)

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 GPTs work to understand exactly what you are asking of them.

#### [Tactic: Specify the steps required to complete a task](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-step-by-step-summarize-and-translate)

#### [Tactic: Provide examples](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-chat-few-shot)

#### [Tactic: Specify the desired length of the output](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-summarize-text-50-words)

**USER**

Summarize the text delimited by triple quotes in 2 paragraphs. """insert text here"""

[Open in Playground](https://platform.openai.com/playground/p/default-summarize-text-2-paragraphs)

**USER**

Summarize the text delimited by triple quotes in 3 bullet points. """insert text here"""

[Open in Playground](https://platform.openai.com/playground/p/default-summarize-text-3-bullet-points)

### [Strategy: Provide reference text](https://platform.openai.com/docs/guides/gpt-best-practices/strategy-provide-reference-text)

#### [Tactic: Instruct the model to answer using a reference text](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-answer-from-retrieved-documents)

Given that GPTs have limited context windows, in order to apply this tactic we need some way to dynamically lookup information that is relevant to the question being asked. [Embeddings](https://platform.openai.com/docs/guides/embeddings/what-are-embeddings) can be used to implement efficient knowledge retrieval. See the tactic ["Use embeddings-based search to implement efficient knowledge retrieval"](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/docs/guides/gpt-best-practices/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>""" Question: <insert question here>

[Open in Playground](https://platform.openai.com/playground/p/default-answer-with-citation)

### [Strategy: Split complex tasks into simpler subtasks](https://platform.openai.com/docs/guides/gpt-best-practices/strategy-split-complex-tasks-into-simpler-subtasks)

#### [Tactic: Use intent classification to identify the most relevant instructions for a user query](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://openai.com/pricing)).

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](https://platform.openai.com/playground/p/default-decomposition-by-intent-classification-1)

Based on the classification of the customer query, a set of more specific instructions can be provided to a GPT model 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](https://platform.openai.com/playground/p/default-decomposition-by-intent-classification-2)

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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-for-dialogue-applications-that-require-very-long-conversations-summarize-or-filter-previous-dialogue)

Since GPTs 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"](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-use-embeddings-based-search-to-implement-efficient-knowledge-retrieval).

#### [Tactic: Summarize long documents piecewise and construct a full summary recursively](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-summarize-long-documents-piecewise-and-construct-a-full-summary-recursively)

Since GPTs 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](https://openai.com/research/summarizing-books) by OpenAI using variants of GPT-3.

### [Strategy: Give GPTs time to "think"](https://platform.openai.com/docs/guides/gpt-best-practices/strategy-give-gpts-time-to-think)

#### [Tactic: Instruct the model to work out its own solution before rushing to a conclusion](https://platform.openai.com/docs/guides/gpt-best-practices/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**

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 Total cost: 100x + 250x + 100,000 + 100x = 450x + 100,000

**ASSISTANT**

The student's solution is correct.

[Open in Playground](https://platform.openai.com/playground/p/default-rushing-to-a-conclusion)

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 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 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](https://platform.openai.com/playground/p/default-avoid-rushing-to-a-conclusion)

#### [Tactic: Use inner monologue or a sequence of queries to hide the model's reasoning process](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-inner-monologue)

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](https://platform.openai.com/playground/p/default-query-sequence-1)

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](https://platform.openai.com/playground/p/default-query-sequence-2)

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](https://platform.openai.com/playground/p/default-query-sequence-3)

#### [Tactic: Ask the model if it missed anything on previous passes](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-2nd-pass)

### [Strategy: Use external tools](https://platform.openai.com/docs/guides/gpt-best-practices/strategy-use-external-tools)

#### [Tactic: Use embeddings-based search to implement efficient knowledge retrieval](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://github.com/openai/openai-cookbook/blob/main/examples/vector_databases/Using_vector_databases_for_embeddings_search.ipynb). See the tactic [“Instruct the model to use retrieved knowledge to answer queries”](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/docs/guides/gpt-best-practices/tactic-use-code-execution-to-perform-more-accurate-calculations-or-call-external-apis)

GPTs 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 backtics. 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](https://platform.openai.com/playground/p/default-code-execution)

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](https://platform.openai.com/playground/p/default-code-api)

**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](https://platform.openai.com/docs/guides/gpt-best-practices/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 GPT models to call external functions. To learn more see the [function calling section](https://platform.openai.com/docs/guides/gpt/function-calling) in our introductory GPT guide and more [function calling examples](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_call_functions_with_chat_models.ipynb) in the OpenAI Cookbook.

### [Strategy: Test changes systematically](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://github.com/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](https://platform.openai.com/docs/guides/gpt-best-practices/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](https://platform.openai.com/playground/p/default-model-based-eval-1)

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](https://platform.openai.com/playground/p/default-model-based-eval-2)

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](https://platform.openai.com/playground/p/default-model-based-eval-3)

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](https://platform.openai.com/playground/p/default-model-based-eval-4)

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](https://platform.openai.com/playground/p/default-model-based-eval-5)

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](https://platform.openai.com/playground/p/default-model-based-eval-6)

## [Other resources](https://platform.openai.com/docs/guides/gpt-best-practices/other-resources)

For more inspiration, visit the [OpenAI Cookbook](https://github.com/openai/openai-cookbook), which contains example code and also links to third-party resources such as:

* [Prompting libraries & tools](https://github.com/openai/openai-cookbook#prompting-libraries--tools)
* [Prompting guides](https://github.com/openai/openai-cookbook#prompting-guides)
* [Video courses](https://github.com/openai/openai-cookbook#video-courses)
* [Papers on advanced prompting to improve reasoning](https://github.com/openai/openai-cookbook#papers-on-advanced-prompting-to-improve-reasoning)

# [Fine-tuning](https://platform.openai.com/docs/guides/fine-tuning/fine-tuning)

Learn how to customize a model for your application.

## [Introduction](https://platform.openai.com/docs/guides/fine-tuning/introduction)

On July 6, 2023, we [announced](https://openai.com/blog/gpt-4-api-general-availability) the deprecation of ada, babbage, curie and davinci models. These models, including fine-tuned versions, will be turned off on January 4, 2024. We are actively working on enabling fine-tuning for upgraded base GPT-3 models as well as GPT-3.5 Turbo and GPT-4, we recommend waiting for those new options to be available rather than fine-tuning based off of the soon to be deprecated models.

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

1. Higher quality results than prompt design
2. Ability to train on more examples than can fit in a prompt
3. Token savings due to shorter prompts
4. Lower latency requests

GPT-3 has been pre-trained on a vast amount of text from the open internet. When given a prompt with just a few examples, it can often intuit what task you are trying to perform and generate a plausible completion. This 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 examples in the prompt anymore.** 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. Use your fine-tuned model

Visit our [pricing page](https://openai.com/api/pricing) to learn more about how fine-tuned model training and usage are billed.

## [What models can be fine-tuned?](https://platform.openai.com/docs/guides/fine-tuning/what-models-can-be-fine-tuned)

We are working on safely enabling fine-tuning for GPT-4 and GPT-3.5 Turbo and expect this feature to be available later this year.

Fine-tuning is currently only available for the following base models: davinci, curie, babbage, and ada. These are the original models that do not have any instruction following training (like text-davinci-003 does for example). You are also able to [continue fine-tuning a fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/continue-fine-tuning-from-a-fine-tuned-model) to add additional data without having to start from scratch.

## [Installation](https://platform.openai.com/docs/guides/fine-tuning/installation)

We recommend using our OpenAI command-line interface (CLI). To install this, run

pip install --upgrade openai

(The following instructions work for version **0.9.4** and up. Additionally, the OpenAI CLI requires python 3.)

Set your OPENAI\_API\_KEY environment variable by adding the following line into your shell initialization script (e.g. .bashrc, zshrc, etc.) or running it in the command line before the fine-tuning command:

export OPENAI\_API\_KEY="<OPENAI\_API\_KEY>"

## [Prepare training data](https://platform.openai.com/docs/guides/fine-tuning/prepare-training-data)

Training data is how you teach GPT-3 what you'd like it to say.

Your data must be a [JSONL](https://jsonlines.org/) document, where each line is a prompt-completion pair corresponding to a training example. You can use our [CLI data preparation tool](https://platform.openai.com/docs/guides/fine-tuning/cli-data-preparation-tool) to easily convert your data into this file format.

1

2

3

4

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

{"prompt": "<prompt text>", "completion": "<ideal generated text>"}

...

Designing your prompts and completions for fine-tuning is different from designing your prompts for use with our base models (Davinci, Curie, Babbage, Ada). In particular, while prompts for base models often consist of multiple examples ("few-shot learning"), for fine-tuning, each training example generally consists of a single input example and its associated output, without the need to give detailed instructions or include multiple examples in the same prompt.

For more detailed guidance on how to prepare training data for various tasks, please refer to our [preparing your dataset](https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset) best practices.

The more training examples you have, the better. We recommend having at least a couple hundred examples. In general, we've found that each doubling of the dataset size leads to a linear increase in model quality.

### [CLI data preparation tool](https://platform.openai.com/docs/guides/fine-tuning/cli-data-preparation-tool)

We developed a tool which validates, gives suggestions and reformats your data:

openai tools fine\_tunes.prepare\_data -f <LOCAL\_FILE>

This tool accepts different formats, with the only requirement that they contain a prompt and a completion column/key. You can pass a **CSV, TSV, XLSX, JSON** or **JSONL** file, and it will save the output into a JSONL file ready for fine-tuning, after guiding you through the process of suggested changes.

## [Create a fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/create-a-fine-tuned-model)

The following assumes you've already prepared training data following the [above instructions](https://platform.openai.com/docs/guides/fine-tuning/prepare-training-data).

Start your fine-tuning job using the OpenAI CLI:

openai api fine\_tunes.create -t <TRAIN\_FILE\_ID\_OR\_PATH> -m <BASE\_MODEL>

Where BASE\_MODEL is the name of the base model you're starting from (ada, babbage, curie, or davinci). You can customize your fine-tuned model's name using the [suffix parameter](https://platform.openai.com/docs/guides/fine-tuning/customize-your-model-name).

Running the above command does several things:

1. Uploads the file using the [files API](https://platform.openai.com/docs/api-reference/files) (or uses an already-uploaded file)
2. Creates a fine-tune job
3. Streams events until the job is done (this often takes minutes, but can take hours if there are many jobs in the queue or your dataset is large)

Every fine-tuning job starts from a base model, which defaults to curie. The choice of model influences both the performance of the model and the cost of running your fine-tuned model. Your model can be one of: ada, babbage, curie, or davinci. Visit our [pricing page](https://openai.com/api/pricing/#faq-fine-tuning-pricing-calculation) for details on fine-tune rates.

After you've started a fine-tune job, it may take some time to complete. Your job may be queued behind other jobs on our system, and training our model can take minutes or hours depending on the model and dataset size. If the event stream is interrupted for any reason, you can resume it by running:

openai api fine\_tunes.follow -i <YOUR\_FINE\_TUNE\_JOB\_ID>

When the job is done, it should display the name of the fine-tuned model.

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

1

2

3

4

5

6

7

8

9

10

# List all created fine-tunes

openai api fine\_tunes.list

# Retrieve the state of a fine-tune. The resulting object includes

# job status (which can be one of pending, running, succeeded, or failed)

# and other information

openai api fine\_tunes.get -i <YOUR\_FINE\_TUNE\_JOB\_ID>

# Cancel a job

openai api fine\_tunes.cancel -i <YOUR\_FINE\_TUNE\_JOB\_ID>

## [Use a fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/use-a-fine-tuned-model)

When a job has succeeded, the fine\_tuned\_model field will be populated with the name of the model. You may now specify this model as a parameter to our [Completions API](https://platform.openai.com/docs/api-reference/completions), and make requests to it using the [Playground](https://platform.openai.com/playground).

After your job first completes, it may take several minutes for your model to become ready to handle requests. If completion requests to your model time out, it is likely because your model is still being loaded. If this happens, try again in a few minutes.

You can start making requests by passing the model name as the model parameter of a completion request:

OpenAI CLI:

openai api completions.create -m <FINE\_TUNED\_MODEL> -p <YOUR\_PROMPT>

cURL:

1

2

3

4

curl https://api.openai.com/v1/completions \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-H "Content-Type: application/json" \

-d '{"prompt": YOUR\_PROMPT, "model": FINE\_TUNED\_MODEL}'

Python:

1

2

3

4

import openai

openai.Completion.create(

model=FINE\_TUNED\_MODEL,

prompt=YOUR\_PROMPT)

Node.js:

1

2

3

4

const response = await openai.createCompletion({

model: FINE\_TUNED\_MODEL

prompt: YOUR\_PROMPT,

});

You may continue to use all the other [Completions](https://platform.openai.com/docs/api-reference/completions) parameters like temperature, frequency\_penalty, presence\_penalty, etc, on these requests to fine-tuned models.

## [Delete a fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/delete-a-fine-tuned-model)

To delete a fine-tuned model, you must be designated an "owner" within your organization.

OpenAI CLI:

openai api models.delete -i <FINE\_TUNED\_MODEL>

cURL:

curl -X "DELETE" https://api.openai.com/v1/models/<FINE\_TUNED\_MODEL> \

-H "Authorization: Bearer $OPENAI\_API\_KEY"

Python:

import openai

openai.Model.delete(FINE\_TUNED\_MODEL)

# [Preparing your dataset](https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset)

Fine-tuning is a powerful technique to create a new model that's specific to your use case. **Before fine-tuning your model, we strongly recommend reading these best practices and**[**specific guidelines**](https://platform.openai.com/docs/guides/fine-tuning/specific-guidelines)**for your use case below.**

## [Data formatting](https://platform.openai.com/docs/guides/fine-tuning/data-formatting)

To fine-tune a model, you'll need a set of training examples that each consist of a single input ("prompt") and its associated output ("completion"). This is notably different from using our base models, where you might input detailed instructions or multiple examples in a single prompt.

* Each prompt should end with a fixed separator to inform the model when the prompt ends and the completion begins. A simple separator which generally works well is \n\n###\n\n. The separator should not appear elsewhere in any prompt.
* Each completion should start with a whitespace due to our [tokenization](https://platform.openai.com/tokenizer), which tokenizes most words with a preceding whitespace.
* Each completion should end with a fixed stop sequence to inform the model when the completion ends. A stop sequence could be \n, ###, or any other token that does not appear in any completion.
* For inference, you should format your prompts in the same way as you did when creating the training dataset, including the same separator. Also specify the same stop sequence to properly truncate the completion.

## [General best practices](https://platform.openai.com/docs/guides/fine-tuning/general-best-practices)

Fine-tuning performs better with more high-quality examples. To fine-tune a model that performs better than using a high-quality prompt with our base models, you should provide at least a few hundred high-quality examples, ideally vetted by human experts. From there, performance tends to linearly increase with every doubling of the number of examples. Increasing the number of examples is usually the best and most reliable way of improving performance.

Classifiers are the easiest models to get started with. For classification problems we suggest using ada, which generally tends to perform only very slightly worse than more capable models once fine-tuned, whilst being significantly faster and cheaper.

If you are fine-tuning on a pre-existing dataset rather than writing prompts from scratch, be sure to manually review your data for offensive or inaccurate content if possible, or review as many random samples of the dataset as possible if it is large.

## [Specific guidelines](https://platform.openai.com/docs/guides/fine-tuning/specific-guidelines)

Fine-tuning can solve a variety of problems, and the optimal way to use it may depend on your specific use case. Below, we've listed the most common use cases for fine-tuning and corresponding guidelines.

* [Classification](https://platform.openai.com/docs/guides/fine-tuning/classification)
  + [Is the model making untrue statements?](https://platform.openai.com/docs/guides/fine-tuning/case-study-is-the-model-making-untrue-statements)
  + [Sentiment analysis](https://platform.openai.com/docs/guides/fine-tuning/case-study-sentiment-analysis)
  + [Categorization for email triage](https://platform.openai.com/docs/guides/fine-tuning/case-study-categorization-for-email-triage)
* [Conditional generation](https://platform.openai.com/docs/guides/fine-tuning/conditional-generation)
  + [Write an engaging ad based on a Wikipedia article](https://platform.openai.com/docs/guides/fine-tuning/case-study-write-an-engaging-ad-based-on-a-wikipedia-article)
  + [Entity extraction](https://platform.openai.com/docs/guides/fine-tuning/case-study-entity-extraction)
  + [Customer support chatbot](https://platform.openai.com/docs/guides/fine-tuning/case-study-customer-support-chatbot)
  + [Product description based on a technical list of properties](https://platform.openai.com/docs/guides/fine-tuning/case-study-product-description-based-on-a-technical-list-of-properties)

### [Classification](https://platform.openai.com/docs/guides/fine-tuning/classification)

In classification problems, each input in the prompt should be classified into one of the predefined classes. For this type of problem, we recommend:

* Use a separator at the end of the prompt, e.g. \n\n###\n\n. Remember to also append this separator when you eventually make requests to your model.
* Choose classes that map to a single [token](https://platform.openai.com/tokenizer). At inference time, specify max\_tokens=1 since you only need the first token for classification.
* Ensure that the prompt + completion doesn't exceed 2048 tokens, including the separator
* Aim for at least ~100 examples per class
* To get class log probabilities you can specify logprobs=5 (for 5 classes) when using your model
* Ensure that the dataset used for finetuning is very similar in structure and type of task as what the model will be used for

#### [Case study: Is the model making untrue statements?](https://platform.openai.com/docs/guides/fine-tuning/case-study-is-the-model-making-untrue-statements)

Let's say you'd like to ensure that the text of the ads on your website mention the correct product and company. In other words, you want to ensure the model isn't making things up. You may want to fine-tune a classifier which filters out incorrect ads.

The dataset might look something like the following:

{"prompt":"Company: BHFF insurance\nProduct: allround insurance\nAd:One stop shop for all your insurance needs!\nSupported:", "completion":" yes"}

{"prompt":"Company: Loft conversion specialists\nProduct: -\nAd:Straight teeth in weeks!\nSupported:", "completion":" no"}

In the example above, we used a structured input containing the name of the company, the product, and the associated ad. As a separator we used \nSupported: which clearly separated the prompt from the completion. With a sufficient number of examples, the separator doesn't make much of a difference (usually less than 0.4%) as long as it doesn't appear within the prompt or the completion.

For this use case we fine-tuned an ada model since it will be faster and cheaper, and the performance will be comparable to larger models because it is a classification task.

Now we can query our model by making a Completion request.

1

2

3

4

5

6

7

8

curl https://api.openai.com/v1/completions \

-H "Content-Type: application/json" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-d '{

"prompt": "Company: Reliable accountants Ltd\nProduct: Personal Tax help\nAd:Best advice in town!\nSupported:",

"max\_tokens": 1,

"model": "YOUR\_FINE\_TUNED\_MODEL\_NAME"

}'

Which will return either yes or no.

#### [Case study: Sentiment analysis](https://platform.openai.com/docs/guides/fine-tuning/case-study-sentiment-analysis)

Let's say you'd like to get a degree to which a particular tweet is positive or negative. The dataset might look something like the following:

{"prompt":"Overjoyed with the new iPhone! ->", "completion":" positive"}

{"prompt":"@lakers disappoint for a third straight night https://t.co/38EFe43 ->", "completion":" negative"}

Once the model is fine-tuned, you can get back the log probabilities for the first completion token by setting logprobs=2 on the completion request. The higher the probability for positive class, the higher the relative sentiment.

Now we can query our model by making a Completion request.

curl https://api.openai.com/v1/completions \

-H "Content-Type: application/json" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-d '{

"prompt": "https://t.co/f93xEd2 Excited to share my latest blog post! ->",

"max\_tokens": 1,

"model": "YOUR\_FINE\_TUNED\_MODEL\_NAME"

}'

Which will return:

{

"id": "cmpl-COMPLETION\_ID",

"object": "text\_completion",

"created": 1589498378,

"model": "YOUR\_FINE\_TUNED\_MODEL\_NAME",

"choices": [

{

"logprobs": {

"text\_offset": [19],

"token\_logprobs": [-0.03597255],

"tokens": [" positive"],

"top\_logprobs": [

{

" negative": -4.9785037,

" positive": -0.03597255

}

]

},

"text": " positive",

"index": 0,

"finish\_reason": "length"

}

]

}

#### [Case study: Categorization for Email triage](https://platform.openai.com/docs/guides/fine-tuning/case-study-categorization-for-email-triage)

Let's say you'd like to categorize incoming email into one of a large number of predefined categories. For classification into a large number of categories, we recommend you convert those categories into numbers, which will work well up to ~500 categories. We've observed that adding a space before the number sometimes slightly helps the performance, due to tokenization. You may want to structure your training data as follows:

{

"prompt": "Subject: <email\_subject>\nFrom:<customer\_name>\nDate:<date>\nContent:<email\_body>\n\n###\n\n",

"completion": " <numerical\_category>"

}

For example:

{

"prompt": "Subject: Update my address\nFrom:Joe Doe\nTo:support@ourcompany.com\nDate:2021-06-03\nContent:Hi,\nI would like to update my billing address to match my delivery address.\n\nPlease let me know once done.\n\nThanks,\nJoe\n\n###\n\n",

"completion": " 4"

}

In the example above we used an incoming email capped at 2043 tokens as input. (This allows for a 4 token separator and a one token completion, summing up to 2048.) As a separator we used \n\n###\n\n and we removed any occurrence of ### within the email.

### [Conditional generation](https://platform.openai.com/docs/guides/fine-tuning/conditional-generation)

Conditional generation is a problem where the content needs to be generated given some kind of input. This includes paraphrasing, summarizing, entity extraction, product description writing given specifications, chatbots and many others. For this type of problem we recommend:

* Use a separator at the end of the prompt, e.g. \n\n###\n\n. Remember to also append this separator when you eventually make requests to your model.
* Use an ending token at the end of the completion, e.g. END
* Remember to add the ending token as a stop sequence during inference, e.g. stop=[" END"]
* Aim for at least ~500 examples
* Ensure that the prompt + completion doesn't exceed 2048 tokens, including the separator
* Ensure the examples are of high quality and follow the same desired format
* Ensure that the dataset used for finetuning is very similar in structure and type of task as what the model will be used for
* Using Lower learning rate and only 1-2 epochs tends to work better for these use cases

#### [Case study: Write an engaging ad based on a Wikipedia article](https://platform.openai.com/docs/guides/fine-tuning/case-study-write-an-engaging-ad-based-on-a-wikipedia-article)

This is a generative use case so you would want to ensure that the samples you provide are of the highest quality, as the fine-tuned model will try to imitate the style (and mistakes) of the given examples. A good starting point is around 500 examples. A sample dataset might look like this:

{

"prompt": "<Product Name>\n<Wikipedia description>\n\n###\n\n",

"completion": " <engaging ad> END"

}

For example:

{

"prompt": "Samsung Galaxy Feel\nThe Samsung Galaxy Feel is an Android smartphone developed by Samsung Electronics exclusively for the Japanese market. The phone was released in June 2017 and was sold by NTT Docomo. It runs on Android 7.0 (Nougat), has a 4.7 inch display, and a 3000 mAh battery.\nSoftware\nSamsung Galaxy Feel runs on Android 7.0 (Nougat), but can be later updated to Android 8.0 (Oreo).\nHardware\nSamsung Galaxy Feel has a 4.7 inch Super AMOLED HD display, 16 MP back facing and 5 MP front facing cameras. It has a 3000 mAh battery, a 1.6 GHz Octa-Core ARM Cortex-A53 CPU, and an ARM Mali-T830 MP1 700 MHz GPU. It comes with 32GB of internal storage, expandable to 256GB via microSD. Aside from its software and hardware specifications, Samsung also introduced a unique a hole in the phone's shell to accommodate the Japanese perceived penchant for personalizing their mobile phones. The Galaxy Feel's battery was also touted as a major selling point since the market favors handsets with longer battery life. The device is also waterproof and supports 1seg digital broadcasts using an antenna that is sold separately.\n\n###\n\n",

"completion": "Looking for a smartphone that can do it all? Look no further than Samsung Galaxy Feel! With a slim and sleek design, our latest smartphone features high-quality picture and video capabilities, as well as an award winning battery life. END"

}

Here we used a multi line separator, as Wikipedia articles contain multiple paragraphs and headings. We also used a simple end token, to ensure that the model knows when the completion should finish.

#### [Case study: Entity extraction](https://platform.openai.com/docs/guides/fine-tuning/case-study-entity-extraction)

This is similar to a language transformation task. To improve the performance, it is best to either sort different extracted entities alphabetically or in the same order as they appear in the original text. This will help the model to keep track of all the entities which need to be generated in order. The dataset could look as follows:

{

"prompt": "<any text, for example news article>\n\n###\n\n",

"completion": " <list of entities, separated by a newline> END"

}

For example:

{

"prompt": "Portugal will be removed from the UK's green travel list from Tuesday, amid rising coronavirus cases and concern over a \"Nepal mutation of the so-called Indian variant\". It will join the amber list, meaning holidaymakers should not visit and returnees must isolate for 10 days...\n\n###\n\n",

"completion": " Portugal\nUK\nNepal mutation\nIndian variant END"

}

A multi-line separator works best, as the text will likely contain multiple lines. Ideally there will be a high diversity of the types of input prompts (news articles, Wikipedia pages, tweets, legal documents), which reflect the likely texts which will be encountered when extracting entities.

#### [Case study: Customer support chatbot](https://platform.openai.com/docs/guides/fine-tuning/case-study-customer-support-chatbot)

A chatbot will normally contain relevant context about the conversation (order details), summary of the conversation so far as well as most recent messages. For this use case the same past conversation can generate multiple rows in the dataset, each time with a slightly different context, for every agent generation as a completion. This use case will require a few thousand examples, as it will likely deal with different types of requests, and customer issues. To ensure the performance is of high quality we recommend vetting the conversation samples to ensure the quality of agent messages. The summary can be generated with a separate text transformation fine tuned model. The dataset could look as follows:

{"prompt":"Summary: <summary of the interaction so far>\n\nSpecific information:<for example order details in natural language>\n\n###\n\nCustomer: <message1>\nAgent: <response1>\nCustomer: <message2>\nAgent:", "completion":" <response2>\n"}

{"prompt":"Summary: <summary of the interaction so far>\n\nSpecific information:<for example order details in natural language>\n\n###\n\nCustomer: <message1>\nAgent: <response1>\nCustomer: <message2>\nAgent: <response2>\nCustomer: <message3>\nAgent:", "completion":" <response3>\n"}

Here we purposefully separated different types of input information, but maintained Customer Agent dialog in the same format between a prompt and a completion. All the completions should only be by the agent, and we can use \n as a stop sequence when doing inference.

#### [Case study: Product description based on a technical list of properties](https://platform.openai.com/docs/guides/fine-tuning/case-study-product-description-based-on-a-technical-list-of-properties)

Here it is important to convert the input data into a natural language, which will likely lead to superior performance. For example, the following format:

{

"prompt": "Item=handbag, Color=army\_green, price=$99, size=S->",

"completion": " This stylish small green handbag will add a unique touch to your look, without costing you a fortune."

}

Won't work as well as:

{

"prompt": "Item is a handbag. Colour is army green. Price is midrange. Size is small.->",

"completion": " This stylish small green handbag will add a unique touch to your look, without costing you a fortune."

}

For high performance ensure that the completions were based on the description provided. If external content is often consulted, then adding such content in an automated way would improve the performance. If the description is based on images, it may help to use an algorithm to extract a textual description of the image. Since completions are only one sentence long, we can use . as the stop sequence during inference.

# [Advanced usage](https://platform.openai.com/docs/guides/fine-tuning/advanced-usage)

## [Customize your model name](https://platform.openai.com/docs/guides/fine-tuning/customize-your-model-name)

You can add a suffix of up to 40 characters to your fine-tuned model name using the [suffix](https://platform.openai.com/docs/api-reference/fine-tunes/create#fine-tunes/create-suffix) parameter.

OpenAI CLI:

openai api fine\_tunes.create -t test.jsonl -m ada --suffix "custom model name"

The resulting name would be:

ada:ft-your-org:custom-model-name-2022-02-15-04-21-04

## [Analyzing your fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/analyzing-your-fine-tuned-model)

We attach a result file to each job once it has been completed. This results file ID will be listed when you retrieve a fine-tune, and also when you look at the events on a fine-tune. You can download these files:

OpenAI CLI:

openai api fine\_tunes.results -i <YOUR\_FINE\_TUNE\_JOB\_ID>

CURL:

curl https://api.openai.com/v1/files/$RESULTS\_FILE\_ID/content \

-H "Authorization: Bearer $OPENAI\_API\_KEY" > results.csv

The \_results.csv file contains a row for each training step, where a step refers to one forward and backward pass on a batch of data. In addition to the step number, each row contains the following fields corresponding to that step:

* **elapsed\_tokens**: the number of tokens the model has seen so far (including repeats)
* **elapsed\_examples**: the number of examples the model has seen so far (including repeats), where one example is one element in your batch. For example, if batch\_size = 4, each step will increase elapsed\_examples by 4.
* **training\_loss**: loss on the training batch
* **training\_sequence\_accuracy**: the percentage of **completions** in the training batch for which the model's predicted tokens matched the true completion tokens exactly. For example, with a batch\_size of 3, if your data contains the completions [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 2/3 = 0.67
* **training\_token\_accuracy**: the percentage of **tokens** in the training batch that were correctly predicted by the model. For example, with a batch\_size of 3, if your data contains the completions [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 5/6 = 0.83

### [Classification specific metrics](https://platform.openai.com/docs/guides/fine-tuning/classification-specific-metrics)

We also provide the option of generating additional classification-specific metrics in the results file, such as accuracy and weighted F1 score. These metrics are periodically calculated against the full validation set and at the end of fine-tuning. You will see them as additional columns in your results file.

To enable this, set the parameter --compute\_classification\_metrics. Additionally, you must provide a validation file, and set either the classification\_n\_classes parameter, for multiclass classification, or classification\_positive\_class, for binary classification.

OpenAI CLI:

# For multiclass classification

openai api fine\_tunes.create \

-t <TRAIN\_FILE\_ID\_OR\_PATH> \

-v <VALIDATION\_FILE\_OR\_PATH> \

-m <MODEL> \

--compute\_classification\_metrics \

--classification\_n\_classes <N\_CLASSES>

# For binary classification

openai api fine\_tunes.create \

-t <TRAIN\_FILE\_ID\_OR\_PATH> \

-v <VALIDATION\_FILE\_OR\_PATH> \

-m <MODEL> \

--compute\_classification\_metrics \

--classification\_n\_classes 2 \

--classification\_positive\_class <POSITIVE\_CLASS\_FROM\_DATASET>

The following metrics will be displayed in your [results file](https://platform.openai.com/docs/guides/fine-tuning/analyzing-your-fine-tuned-model) if you set --compute\_classification\_metrics:

##### For multiclass classification

* **classification/accuracy**: accuracy
* **classification/weighted\_f1\_score**: weighted F-1 score

##### For binary classification

The following metrics are based on a classification threshold of 0.5 (i.e. when the probability is > 0.5, an example is classified as belonging to the positive class.)

* **classification/accuracy**
* **classification/precision**
* **classification/recall**
* **classification/f{beta}**
* **classification/auroc** - AUROC
* **classification/auprc** - AUPRC

Note that these evaluations assume that you are using text labels for classes that tokenize down to a single token, as described above. If these conditions do not hold, the numbers you get will likely be wrong.

### [Validation](https://platform.openai.com/docs/guides/fine-tuning/validation)

You can reserve some of your data for validation. A validation file has exactly the same format as a train file, and your train and validation data should be mutually exclusive.

If you include a validation file when creating your fine-tune job, the generated results file will include evaluations on how well the fine-tuned model performs against your validation data at periodic intervals during training.

OpenAI CLI:

openai api fine\_tunes.create -t <TRAIN\_FILE\_ID\_OR\_PATH> \

-v <VALIDATION\_FILE\_ID\_OR\_PATH> \

-m <MODEL>

If you provided a validation file, we periodically calculate metrics on batches of validation data during training time. You will see the following additional metrics in your results file:

* **validation\_loss**: loss on the validation batch
* **validation\_sequence\_accuracy**: the percentage of completions in the validation batch for which the model's predicted tokens matched the true completion tokens exactly. For example, with a batch\_size of 3, if your data contains the completion [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 2/3 = 0.67
* **validation\_token\_accuracy**: the percentage of tokens in the validation batch that were correctly predicted by the model. For example, with a batch\_size of 3, if your data contains the completion [[1, 2], [0, 5], [4, 2]] and the model predicted [[1, 1], [0, 5], [4, 2]], this accuracy will be 5/6 = 0.83

## [Hyperparameters](https://platform.openai.com/docs/guides/fine-tuning/hyperparameters)

We've picked default hyperparameters that work well across a range of use cases. The only required parameter is the training file.

That said, tweaking the hyperparameters used for fine-tuning can often lead to a model that produces higher quality output. In particular, you may want to configure the following:

* model: The name of the base model to fine-tune. You can select one of "ada", "babbage", "curie", or "davinci". To learn more about these models, see the [Models](https://platform.openai.com/docs/models) documentation.
* n\_epochs - defaults to 4. The number of epochs to train the model for. An epoch refers to one full cycle through the training dataset.
* batch\_size - defaults to ~0.2% of the number of examples in the training set, capped at 256. The batch size is the number of training examples used to train a single forward and backward pass. In general, we've found that larger batch sizes tend to work better for larger datasets.
* learning\_rate\_multiplier - defaults to 0.05, 0.1, or 0.2 depending on final batch\_size. The fine-tuning learning rate is the original learning rate used for pretraining multiplied by this multiplier. We recommend experimenting with values in the range 0.02 to 0.2 to see what produces the best results. Empirically, we've found that larger learning rates often perform better with larger batch sizes.
* compute\_classification\_metrics - defaults to False. If True, for fine-tuning for classification tasks, computes classification-specific metrics (accuracy, F-1 score, etc) on the validation set at the end of every epoch.

To configure these additional hyperparameters, pass them in via command line flags on the OpenAI CLI, for example:

openai api fine\_tunes.create \

-t file-JD89ePi5KMsB3Tayeli5ovfW \

-m ada \

--n\_epochs 1

## [Continue fine-tuning from a fine-tuned model](https://platform.openai.com/docs/guides/fine-tuning/continue-fine-tuning-from-a-fine-tuned-model)

If you have already fine-tuned a model for your task and now have additional training data that you would like to incorporate, you can continue fine-tuning from the model. This creates a model that has learned from all of the training data without having to re-train from scratch.

To do this, pass in the fine-tuned model name when creating a new fine-tuning job (e.g. -m curie:ft-<org>-<date>). Other training parameters do not have to be changed, however if your new training data is much smaller than your previous training data, you may find it useful to reduce learning\_rate\_multiplier by a factor of 2 to 4.

# [Weights & Biases](https://platform.openai.com/docs/guides/fine-tuning/weights-biases)

You can sync your fine-tunes with [Weights & Biases](https://wandb.me/openai-docs) to track experiments, models, and datasets.

To get started, you will need a [Weights & Biases](https://wandb.me/openai-docs) account and a paid OpenAI plan. To make sure you are using the lastest version of openai and wandb, run:

pip install --upgrade openai wandb

To sync your fine-tunes with Weights & Biases, run:

openai wandb sync

You can read the [Weights & Biases documentation](https://wandb.me/openai-docs) for more information on this integration.

# [Example notebooks](https://platform.openai.com/docs/guides/fine-tuning/example-notebooks)

**Classification**

[**finetuning-classification.ipynb**](https://github.com/openai/openai-cookbook/blob/main/examples/Fine-tuned_classification.ipynb)

This notebook will demonstrate how to fine-tune a model that can classify whether a piece of input text is related to Baseball or Hockey. We will perform this task in four steps in the [notebook](https://github.com/openai/openai-cookbook/blob/main/examples/Fine-tuned_classification.ipynb):

1. **Data exploration** will give an overview of the data source and what an example looks like
2. **Data preparation** will turn our data source into a jsonl file that can be used for fine-tuning
3. **Fine-tuning** will kick off the fine-tuning job and explain the resulting model's performance
4. **Using the model** will demonstrate making requests to the fine-tuned model to get predictions.

Collapse‍

**Question answering**

[**olympics-1-collect-data.ipynb**](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-1-collect-data.ipynb)[**olympics-2-create-qa.ipynb**](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-2-create-qa.ipynb)[**olympics-3-train-qa.ipynb**](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-3-train-qa.ipynb)

The idea of this project is to create a question answering model, based on a few paragraphs of provided text. Base GPT-3 models do a good job at answering questions when the answer is contained within the paragraph, however if the answer isn't contained, the base models tend to try their best to answer anyway, often leading to confabulated answers.

To create a model which answers questions only if there is sufficient context for doing so, we first create a dataset of questions and answers based on paragraphs of text. In order to train the model to answer only when the answer is present, we also add adversarial examples, where the question doesn't match the context. In those cases, we ask the model to output "No sufficient context for answering the question".

We will perform this task in three notebooks:

1. [The first notebook](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-1-collect-data.ipynb) focuses on collecting recent data, which GPT-3 didn't see during it's pre-training. We picked the topic of Olympic Games 2020 (which actually took place in the summer of 2021), and downloaded 713 unique pages. We organized the dataset by individual sections, which will serve as context for asking and answering the questions.
2. [The second notebook](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-2-create-qa.ipynb) will utilize Davinci-instruct to ask a few questions based on a Wikipedia section, as well as answer those questions, based on that section.
3. [The third notebook](https://github.com/openai/openai-cookbook/blob/main/examples/fine-tuned_qa/olympics-3-train-qa.ipynb) will utilize the dataset of context, question and answer pairs to additionally create adversarial questions and context pairs, where the question was not generated on that context. In those cases the model will be prompted to answer "No sufficient context for answering the question". We will also train a discriminator model, which predicts whether the question can be answered based on the context or not.

# [Embeddings](https://platform.openai.com/docs/guides/embeddings/embeddings)

## [What are embeddings?](https://platform.openai.com/docs/guides/embeddings/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](https://platform.openai.com/docs/guides/embeddings/which-distance-function-should-i-use) between two vectors measures their relatedness. Small distances suggest high relatedness and large distances suggest low relatedness.

Visit our [pricing page](https://openai.com/api/pricing/) to learn about Embeddings pricing. Requests are billed based on the number of [tokens](https://platform.openai.com/tokenizer) in the [input](https://platform.openai.com/docs/api-reference/embeddings/create#embeddings/create-input) sent.

**To see embeddings in action, check out our code samples**

* Classification
* Topic clustering
* Search
* Recommendations

[Browse Samples‍](https://platform.openai.com/docs/guides/embeddings/use-cases)

## [How to get embeddings](https://platform.openai.com/docs/guides/embeddings/how-to-get-embeddings)

To get an embedding, send your text string to the [embeddings API endpoint](https://platform.openai.com/docs/api-reference/embeddings) along with a choice of embedding model ID (e.g., text-embedding-ada-002). The response will contain an embedding, which you can extract, save, and use.

Example requests:

Example: Getting embeddings

python



Copy‍

response = openai.Embedding.create(

input="Your text string goes here",

model="text-embedding-ada-002"

)

embeddings = response['data'][0]['embedding']

Example response:

{

"data": [

{

"embedding": [

-0.006929283495992422,

-0.005336422007530928,

...

-4.547132266452536e-05,

-0.024047505110502243

],

"index": 0,

"object": "embedding"

}

],

"model": "text-embedding-ada-002",

"object": "list",

"usage": {

"prompt\_tokens": 5,

"total\_tokens": 5

}

}

See more Python code examples in the [OpenAI Cookbook](https://github.com/openai/openai-cookbook/).

When using OpenAI embeddings, please keep in mind their [limitations and risks](https://platform.openai.com/docs/guides/embeddings/limitations-risks).

## [Embedding models](https://platform.openai.com/docs/guides/embeddings/embedding-models)

OpenAI offers one second-generation embedding model (denoted by -002 in the model ID) and 16 first-generation models (denoted by -001 in the model ID).

We recommend using text-embedding-ada-002 for nearly all use cases. It’s better, cheaper, and simpler to use. Read the [blog post announcement](https://openai.com/blog/new-and-improved-embedding-model).

| **MODEL GENERATION** | **TOKENIZER** | **MAX INPUT TOKENS** | **KNOWLEDGE CUTOFF** |
| --- | --- | --- | --- |
| V2 | cl100k\_base | 8191 | Sep 2021 |
| V1 | GPT-2/GPT-3 | 2046 | Aug 2020 |

Usage is priced per input token, at a rate of $0.0004 per 1000 tokens, or about ~3,000 pages per US dollar (assuming ~800 tokens per page):

| **MODEL** | **ROUGH PAGES PER DOLLAR** | **EXAMPLE PERFORMANCE ON**[**BEIR**](https://paperswithcode.com/sota/zero-shot-text-search-on-beir)**SEARCH EVAL** |
| --- | --- | --- |
| text-embedding-ada-002 | 3000 | 53.9 |
| \*-davinci-\*-001 | 6 | 52.8 |
| \*-curie-\*-001 | 60 | 50.9 |
| \*-babbage-\*-001 | 240 | 50.4 |
| \*-ada-\*-001 | 300 | 49.0 |

### [Second-generation models](https://platform.openai.com/docs/guides/embeddings/second-generation-models)

| **MODEL NAME** | **TOKENIZER** | **MAX INPUT TOKENS** | **OUTPUT DIMENSIONS** |
| --- | --- | --- | --- |
| text-embedding-ada-002 | cl100k\_base | 8191 | 1536 |

**First-generation models (not recommended)**

## [Use cases](https://platform.openai.com/docs/guides/embeddings/use-cases)

Here we show some representative use cases. We will use the [Amazon fine-food reviews dataset](https://www.kaggle.com/snap/amazon-fine-food-reviews) for the following examples.

### [Obtaining the embeddings](https://platform.openai.com/docs/guides/embeddings/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.

[**Obtain\_dataset.ipynb**](https://github.com/openai/openai-cookbook/blob/main/examples/Obtain_dataset.ipynb)

def get\_embedding(text, model="text-embedding-ada-002"):

text = text.replace("\n", " ")

return openai.Embedding.create(input = [text], model=model)['data'][0]['embedding']

df['ada\_embedding'] = df.combined.apply(lambda x: get\_embedding(x, model='text-embedding-ada-002'))

df.to\_csv('output/embedded\_1k\_reviews.csv', index=False)

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

import pandas as pd

df = pd.read\_csv('output/embedded\_1k\_reviews.csv')

df['ada\_embedding'] = df.ada\_embedding.apply(eval).apply(np.array)

**Data visualization in 2D**

**Embedding as a text feature encoder for ML algorithms**

**Classification using the embedding features**

**Zero-shot classification**

**Obtaining user and product embeddings for cold-start recommendation**

**Clustering**

**Text search using embeddings**

**Code search using embeddings**

**Recommendations using embeddings**

## [Limitations & risks](https://platform.openai.com/docs/guides/embeddings/limitations-risks)

Our embedding models may be unreliable or pose social risks in certain cases, and may cause harm in the absence of mitigations.

#### [Social bias](https://platform.openai.com/docs/guides/embeddings/social-bias)

**Limitation**: The models encode social biases, e.g. via stereotypes or negative sentiment towards certain groups.

We found evidence of bias in our models via running the SEAT ([May et al, 2019](https://arxiv.org/abs/1903.10561)) and the Winogender ([Rudinger et al, 2018](https://arxiv.org/abs/1804.09301)) benchmarks. Together, these benchmarks consist of 7 tests that measure whether models contain implicit biases when applied to gendered names, regional names, and some stereotypes.

For example, we found that our models more strongly associate (a) European American names with positive sentiment, when compared to African American names, and (b) negative stereotypes with black women.

These benchmarks are limited in several ways: (a) they may not generalize to your particular use case, and (b) they only test for a very small slice of possible social bias.

**These tests are preliminary, and we recommend running tests for your specific use cases.** These results should be taken as evidence of the existence of the phenomenon, not a definitive characterization of it for your use case. Please see our [usage policies](https://openai.com/policies/usage-policies) for more details and guidance.

Please [contact our support team via chat](https://help.openai.com/en/) if you have any questions; we are happy to advise on this.

#### [Blindness to recent events](https://platform.openai.com/docs/guides/embeddings/blindness-to-recent-events)

**Limitation**: Models lack knowledge of events that occurred after August 2020.

Our models are trained on datasets that contain some information about real world events up until 8/2020. If you rely on the models representing recent events, then they may not perform well.

## [Frequently asked questions](https://platform.openai.com/docs/guides/embeddings/frequently-asked-questions)

### [How can I tell how many tokens a string has before I embed it?](https://platform.openai.com/docs/guides/embeddings/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](https://github.com/openai/tiktoken).

Example code:

import tiktoken

def num\_tokens\_from\_string(string: str, encoding\_name: str) -> int:

"""Returns the number of tokens in a text string."""

encoding = tiktoken.get\_encoding(encoding\_name)

num\_tokens = len(encoding.encode(string))

return num\_tokens

num\_tokens\_from\_string("tiktoken is great!", "cl100k\_base")

For second-generation embedding models like text-embedding-ada-002, use the cl100k\_base encoding.

More details and example code are in the OpenAI Cookbook guide [how to count tokens with tiktoken](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb).

### [How can I retrieve K nearest embedding vectors quickly?](https://platform.openai.com/docs/guides/embeddings/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](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases) on GitHub.

Vector database options include:

* [Chroma](https://github.com/chroma-core/chroma), an open-source embeddings store
* [Milvus](https://github.com/openai/openai-cookbook/blob/main/examples/vector_databases/Using_vector_databases_for_embeddings_search.ipynb), a vector database built for scalable similarity search
* [Pinecone](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases/pinecone), a fully managed vector database
* [Qdrant](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases/qdrant), a vector search engine
* [Redis](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases/redis) as a vector database
* [Typesense](https://typesense.org/docs/0.24.0/api/vector-search.html), fast open source vector search
* [Weaviate](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases/weaviate), an open-source vector search engine
* [Zilliz](https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases/zilliz), data infrastructure, powered by Milvus

### [Which distance function should I use?](https://platform.openai.com/docs/guides/embeddings/which-distance-function-should-i-use)

We recommend [cosine similarity](https://en.wikipedia.org/wiki/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?](https://platform.openai.com/docs/guides/embeddings/can-i-share-my-embeddings-online)

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](https://openai.com/policies/terms-of-use).

**[Speech to text](https://platform.openai.com/docs/guides/speech-to-text/speech-to-text)**

Learn how to turn audio into text

**[Introduction](https://platform.openai.com/docs/guides/speech-to-text/introduction)**

The speech to text API provides two endpoints, transcriptions and translations, based on our state-of-the-art open source large-v2 [Whisper model](https://openai.com/blog/whisper/). 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](https://platform.openai.com/docs/guides/speech-to-text/quickstart)**

**[Transcriptions](https://platform.openai.com/docs/guides/speech-to-text/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



Copy‍

# Note: you need to be using OpenAI Python v0.27.0 for the code below to work

import openai

audio\_file= open("/path/to/file/audio.mp3", "rb")

transcript = openai.Audio.transcribe("whisper-1", audio\_file)

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. .... }

To set additional parameters in a request, you can add more --form lines with the relevant options. For example, if you want to set the output format as text, you would add the following line:

...

--form file=@openai.mp3 \

--form model=whisper-1 \

--form response\_format=text

**[Translations](https://platform.openai.com/docs/guides/speech-to-text/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



Copy‍

# Note: you need to be using OpenAI Python v0.27.0 for the code below to work

import openai

audio\_file= open("/path/to/file/german.mp3", "rb")

transcript = openai.Audio.translate("whisper-1", audio\_file)

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](https://platform.openai.com/docs/guides/speech-to-text/supported-languages)**

We currently [support the following languages](https://github.com/openai/whisper#available-models-and-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](https://en.wikipedia.org/wiki/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.

**[Longer inputs](https://platform.openai.com/docs/guides/speech-to-text/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](https://github.com/jiaaro/pydub) to split the audio:

from pydub import AudioSegment

song = AudioSegment.from\_mp3("good\_morning.mp3")

# PyDub handles time in milliseconds

ten\_minutes = 10 \* 60 \* 1000

first\_10\_minutes = song[:ten\_minutes]

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](https://platform.openai.com/docs/guides/speech-to-text/prompting)**

You can use a [prompt](https://platform.openai.com/docs/api-reference/audio/create#audio/create-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 often misrecognizes 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](https://github.com/openai/whisper/blob/main/whisper/tokenizer.py#L361).
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](https://platform.openai.com/docs/guides/speech-to-text/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.

transcribe(filepath, prompt="ZyntriQix, Digique Plus, CynapseFive, VortiQore V8, EchoNix Array, OrbitalLink Seven, DigiFractal Matrix, PULSE, RAPT, B.R.I.C.K., Q.U.A.R.T.Z., F.L.I.N.T.")

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.

Collapse‍

**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.

system\_prompt = "You are a helpful assistant for the company ZyntriQix. Your task is to correct any spelling discrepancies in the transcribed text. Make sure that the names of the following products are spelled correctly: ZyntriQix, Digique Plus, CynapseFive, VortiQore V8, EchoNix Array, OrbitalLink Seven, DigiFractal Matrix, PULSE, RAPT, B.R.I.C.K., Q.U.A.R.T.Z., F.L.I.N.T. Only add necessary punctuation such as periods, commas, and capitalization, and use only the context provided."

def generate\_corrected\_transcript(temperature, system\_prompt, audio\_file):

response = openai.ChatCompletion.create(

model="gpt-4",

temperature=temperature,

messages=[

{

"role": "system",

"content": system\_prompt

},

{

"role": "user",

"content": transcribe(audio\_file, "")

}

]

)

return response['choices'][0]['message']['content']

corrected\_text = generate\_corrected\_transcript(0, system\_prompt, 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.

Collapse‍

**[Moderation](https://platform.openai.com/docs/guides/moderation/moderation)**

**[Overview](https://platform.openai.com/docs/guides/moderation/overview)**

The [moderations](https://platform.openai.com/docs/api-reference/moderations) endpoint is a tool you can use to check whether content complies with OpenAI's [usage policies](https://openai.com/policies/usage-policies). Developers can thus identify content that our usage policies prohibits 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 harrassment. |
| 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. |
| 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 when monitoring the inputs and outputs of OpenAI APIs. We currently disallow other use cases. Accuracy may be lower on longer pieces of text. 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](https://platform.openai.com/docs/guides/moderation/quickstart)**

To obtain a classification for a piece of text, make a request to the [moderation endpoint](https://platform.openai.com/docs/api-reference/moderations) as demonstrated in the following code snippets:

Example: Getting moderations

python



Copy‍

response = openai.Moderation.create(

input="Sample text goes here"

)

output = response["results"][0]

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

* flagged: Set to true if the model classifies the content as violating OpenAI's usage policies, false otherwise.
* categories: Contains a dictionary of per-category binary usage policies 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.

{

"id": "modr-XXXXX",

"model": "text-moderation-005",

"results": [

{

"flagged": true,

"categories": {

"sexual": false,

"hate": false,

"harassment": false,

"self-harm": false,

"sexual/minors": false,

"hate/threatening": false,

"violence/graphic": false,

"self-harm/intent": false,

"self-harm/instructions": false,

"harassment/threatening": true,

"violence": true,

},

"category\_scores": {

"sexual": 1.2282071e-06,

"hate": 0.010696256,

"harassment": 0.29842457,

"self-harm": 1.5236925e-08,

"sexual/minors": 5.7246268e-08,

"hate/threatening": 0.0060676364,

"violence/graphic": 4.435014e-06,

"self-harm/intent": 8.098441e-10,

"self-harm/instructions": 2.8498655e-11,

"harassment/threatening": 0.63055265,

"violence": 0.99011886,

}

}

]

}

OpenAI will continuously upgrade the moderation endpoint's underlying model. Therefore, custom policies that rely on category\_scores may need recalibration over time.

# [Rate limits](https://platform.openai.com/docs/guides/rate-limits/rate-limits)

## [Overview](https://platform.openai.com/docs/guides/rate-limits/overview)

### [What are rate limits?](https://platform.openai.com/docs/guides/rate-limits/what-are-rate-limits)

A rate limit is a restriction that an API imposes on the number of times a user or client can access the server within a specified period of time.

### [Why do we have rate limits?](https://platform.openai.com/docs/guides/rate-limits/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. It is recommended to follow this guidance before filling out the [Rate Limit Increase Request form](https://docs.google.com/forms/d/e/1FAIpQLSc6gSL3zfHFlL6gNIyUcjkEv29jModHGxg5_XGyr-PrE2LaHw/viewform) with details regarding how to fill it out in the last section.

### [What are the rate limits for our API?](https://platform.openai.com/docs/guides/rate-limits/what-are-the-rate-limits-for-our-api)

You can view the rate limits for your organization under the [rate limits](https://platform.openai.com/account/rate-limits) section of the account management page.

We enforce rate limits at the [organization level](https://platform.openai.com/docs/guides/production-best-practices), not user level, based on the specific endpoint used as well as the type of account you have. Rate limits are measured in three ways: **RPM** (requests per minute), **RPD** (requests per day), and **TPM** (tokens per minute). The table below highlights the default rate limits for our API.

FREE USERSPAID USERS (FIRST 48 HOURS)PAID USERS (AFTER 48 HOURS)

|  | **REQUESTS PER DAY (RPD)** | **REQUESTS PER MINUTE (RPM)** | **TOKENS PER MINUTES (TPM)** |
| --- | --- | --- | --- |
| Text & Embedding | 200 RPD | 3 RPM | 150,000 TPM |
| Chat | 200 RPD | 3 RPM | 40,000 TPM |
| Image | 200 RPD | 5 images / min | --- |
| Audio | 200 RPD | 3 RPM | --- |
| Edit (Deprecated) | 200 RPD | 3 RPM | 150,000 TPM |

For free trial users in the first 48 hour, we have added per day rate limits. The normal RPM and TPM limits apply but there is also a separate limits on the number of requests per day for each endpoint.

It is important to note that the rate limit can be hit by either option depending on what occurs first. For example, you might send 20 requests with only 100 tokens to the Edit endpoint and that would fill your limit, even if you did not send 150k tokens within those 20 requests.

#### [GPT-4 rate limits](https://platform.openai.com/docs/guides/rate-limits/gpt-4-rate-limits)

During the [rollout of GPT-4](http://openai.com/blog/gpt-4-api-general-availability), the model will have more aggressive rate limits to keep up with demand. You can [view your current rate limits](https://platform.openai.com/account/rate-limits) in the rate limits section of the account page. We are unable to accommodate requests for rate limit increases due to capacity constraints. We are prioritizing general access to GPT-4 first and will subsequently raise rate limits automatically as capacity allows.

### [How do rate limits work?](https://platform.openai.com/docs/guides/rate-limits/how-do-rate-limits-work)

If your rate limit is 60 requests per minute and 150k tokens per minute, you’ll be limited either by reaching the requests/min cap or running out of tokens—whichever happens first. For example, if your max requests/min is 60, you should be able to send 1 request per second. If you send 1 request every 800ms, once you hit your rate limit, you’d only need to make your program sleep 200ms in order to send one more request otherwise subsequent requests would fail. With the default of 3,000 requests/min, customers can effectively send 1 request every 20ms, or every .02 seconds.

### [Rate limits in headers](https://platform.openai.com/docs/guides/rate-limits/rate-limits-in-headers)

In addition to seeing your rate limit on your [account page](https://platform.openai.com/account/rate-limits), 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. |

### [What happens if I hit a rate limit error?](https://platform.openai.com/docs/guides/rate-limits/what-happens-if-i-hit-a-rate-limit-error)

Rate limit errors look like this:

Rate limit reached for default-text-davinci-002 in organization org-{id} on requests per min. Limit: 20.000000 / min. Current: 24.000000 / min.

If you hit a rate limit, it means you've made too many requests in a short period of time, and the API is refusing to fulfill further requests until a specified amount of time has passed.

### [Rate limits vs max\_tokens](https://platform.openai.com/docs/guides/rate-limits/rate-limits-vs-max_tokens)

Each [model we offer](https://platform.openai.com/docs/models/overview) has a limited number of tokens that can be passed in as input when making a request. You cannot increase the maximum number of tokens a model takes in. For example, if you are using text-ada-001, the maximum number of tokens you can send to this model is 2,048 tokens per request.

## [Error Mitigation](https://platform.openai.com/docs/guides/rate-limits/error-mitigation)

### [What are some steps I can take to mitigate this?](https://platform.openai.com/docs/guides/rate-limits/what-are-some-steps-i-can-take-to-mitigate-this)

The OpenAI Cookbook has a [Python notebook](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_handle_rate_limits.ipynb) that explains how to avoid rate limit errors, as well an example [Python script](https://github.com/openai/openai-cookbook/blob/main/examples/api_request_parallel_processor.py) 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](https://platform.openai.com/docs/guides/rate-limits/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**

**Example #2: Using the backoff library**

**Example 3: Manual backoff implementation**

#### [Reduce the max\_tokens to match the size of your completions](https://platform.openai.com/docs/guides/rate-limits/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](https://platform.openai.com/docs/guides/rate-limits/batching-requests)

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**

**Example with batching**

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.

## [Request Increase](https://platform.openai.com/docs/guides/rate-limits/request-increase)

### [When should I consider applying for a rate limit increase?](https://platform.openai.com/docs/guides/rate-limits/when-should-i-consider-applying-for-a-rate-limit-increase)

Our default rate limits help us maximize stability and prevent abuse of our API. We increase limits to enable high-traffic applications, so the best time to apply for a rate limit increase is when you feel that you have the necessary traffic data to support a strong case for increasing the rate limit. Large rate limit increase requests without supporting data are not likely to be approved. If you're gearing up for a product launch, please obtain the relevant data through a phased release over 10 days.

Keep in mind that rate limit increases can sometimes take 7-10 days so it makes sense to try and plan ahead and submit early if there is data to support you will reach your rate limit given your current growth numbers.

### [Will my rate limit increase request be rejected?](https://platform.openai.com/docs/guides/rate-limits/will-my-rate-limit-increase-request-be-rejected)

A rate limit increase request is most often rejected because it lacks the data needed to justify the increase. We have provided numerical examples below that show how to best support a rate limit increase request and try our best to approve all requests that align with our safety policy and show supporting data. We are committed to enabling developers to scale and be successful with our API.

### [I’ve implemented exponential backoff for my text/code APIs, but I’m still hitting this error. How do I increase my rate limit?](https://platform.openai.com/docs/guides/rate-limits/i-ve-implemented-exponential-backoff-for-my-text-code-apis-but-i-m-still-hitting-this-error-how-do-i-increase-my-rate-limit)

We understand the frustration that limited rate limits can cause, and we would love to raise the defaults for everyone. However, due to shared capacity constraints, we can only approve rate limit increases for paid customers who have demonstrated a need through our [Rate Limit Increase Request form](https://docs.google.com/forms/d/e/1FAIpQLSc6gSL3zfHFlL6gNIyUcjkEv29jModHGxg5_XGyr-PrE2LaHw/viewform). To help us evaluate your needs properly, we ask that you please provide statistics on your current usage or projections based on historic user activity in the 'Share evidence of need' section of the form. If this information is not available, we recommend a phased release approach. Start by releasing the service to a subset of users at your current rate limits, gather usage data for 10 business days, and then submit a formal rate limit increase request based on that data for our review and approval.

We will review your request and if it is approved, we will notify you of the approval within a period of 7-10 business days.

Here are some examples of how you might fill out this form:

**DALL-E API examples**

| **MODEL** | **ESTIMATE TOKENS/MINUTE** | **ESTIMATE REQUESTS/MINUTE** | **# OF USERS** | **EVIDENCE OF NEED** | **1 HOUR MAX THROUGHPUT COST** |
| --- | --- | --- | --- | --- | --- |
| DALL-E API | N/A | 50 | 1000 | Our app is currently in production and based on our past traffic, we make about 10 requests per minute. | $60 |
| DALL-E API | N/A | 150 | 10,000 | Our app is gaining traction in the App Store and we’re starting to hit rate limits. Can we get triple the default limit of 50 img/min? If we need more we’ll submit a new form. Thanks! | $180 |

Collapse‍

**Language model examples**

| **MODEL** | **ESTIMATE TOKENS/MINUTE** | **ESTIMATE REQUESTS/MINUTE** | **# OF USERS** | **EVIDENCE OF NEED** | **1 HOUR MAX THROUGHPUT COST** |
| --- | --- | --- | --- | --- | --- |
| gpt-3.5-turbo | 325,000 | 4,000 | 50 | We’re releasing to an initial group of alpha testers and need a higher limit to accommodate their initial usage. We have a link here to our google drive which shows analytics and api usage. | $390 |
| gpt-4 | 750,000 | 10,000 | 10,000 | Our application is receiving a lot of interest; we have 50,000 people on our waitlist. We’d like to roll out to groups of 1,000 people/day until we reach 50,000 users. Please see this link of our current token/minute traffic over the past 30 days. This is for 500 users, and based on their usage, we think 750,000 tokens/minute and 10,000 requests/minute will work as a good starting point. | $900 |

Collapse‍

**Code model examples**

| **MODEL** | **ESTIMATE TOKENS/MINUTE** | **ESTIMATE REQUESTS/MINUTE** | **# OF USERS** | **EVIDENCE OF NEED** | **1 HOUR MAX THROUGHPUT COST** |
| --- | --- | --- | --- | --- | --- |
| code-davinci-002 | 150,000 | 1,000 | 15 | We are a group of researchers working on a paper. We estimate that we will need a higher rate limit on code-davinci-002 in order to complete our research before the end of the month. These estimates are based on the following calculation [...] | Codex models are now deprecated and no longer accessible. |

Collapse‍

Please note that these examples are just general use case scenarios, the actual usage rate will vary depending on the specific implementation and usage.

**[Error codes](https://platform.openai.com/docs/guides/error-codes/error-codes)**

This guide includes an overview on error codes you might see from both the [API](https://platform.openai.com/docs/introduction) and our [official Python library](https://platform.openai.com/docs/libraries/python-library). Each error code mentioned in the overview has a dedicated section with further guidance.

**[API errors](https://platform.openai.com/docs/guides/error-codes/api-errors)**

| **CODE** | **OVERVIEW** |
| --- | --- |
| 401 - Invalid Authentication | **Cause:** Invalid Authentication **Solution:** Ensure the correct [API key](https://platform.openai.com/account/api-keys) and requesting organization are being used. |
| 401 - Incorrect API key provided | **Cause:** The requesting API key is not correct. **Solution:** Ensure the API key used is correct, clear your browser cache, or [generate a new one](https://platform.openai.com/account/api-keys). |
| 401 - You must be a member of an organization to use the API | **Cause:** Your account is not part of an organization. **Solution:** Contact us to get added to a new organization or ask your organization manager to [invite you to an organization](https://platform.openai.com/account/members). |
| 429 - Rate limit reached for requests | **Cause:** You are sending requests too quickly. **Solution:** Pace your requests. Read the [Rate limit guide](https://platform.openai.com/docs/guides/rate-limits). |
| 429 - You exceeded your current quota, please check your plan and billing details | **Cause:** You have hit your maximum monthly spend (hard limit) which you can view in the [account billing section](https://platform.openai.com/account/billing/limits). **Solution:** [Apply for a quota increase](https://platform.openai.com/forms/quota-increase). |
| 500 - The server had an error while processing your request | **Cause:** Issue on our servers. **Solution:** Retry your request after a brief wait and contact us if the issue persists. Check the [status page](https://status.openai.com/). |
| 503 - The engine is currently overloaded, please try again later | **Cause:** Our servers are experiencing high traffic. **Solution:** Please retry your requests after a brief wait. |

**401 - Invalid Authentication**

**401 - Incorrect API key provided**

**401 - You must be a member of an organization to use the API**

**429 - Rate limit reached for requests**

**429 - You exceeded your current quota, please check your plan and billing details**

**503 - The engine is currently overloaded, please try again later**

**[Python library error types](https://platform.openai.com/docs/guides/error-codes/python-library-error-types)**

| **TYPE** | **OVERVIEW** |
| --- | --- |
| APIError | **Cause:** Issue on our side. **Solution:** Retry your request after a brief wait and contact us if the issue persists. |
| Timeout | **Cause:** Request timed out. **Solution:** Retry your request after a brief wait and contact us if the issue persists. |
| RateLimitError | **Cause:** You have hit your assigned rate limit. **Solution:** Pace your requests. Read more in our [Rate limit guide](https://platform.openai.com/docs/guides/rate-limits). |
| APIConnectionError | **Cause:** Issue connecting to our services. **Solution:** Check your network settings, proxy configuration, SSL certificates, or firewall rules. |
| InvalidRequestError | **Cause:** Your request was malformed or missing some required parameters, such as a token or an input. **Solution:** The error message should advise you on the specific error made. Check the [documentation](https://platform.openai.com/docs/api-reference/) 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. |
| AuthenticationError | **Cause:** Your API key or token was invalid, expired, or revoked. **Solution:** 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. |
| ServiceUnavailableError | **Cause:** Issue on our servers. **Solution:** Retry your request after a brief wait and contact us if the issue persists. Check the [status page](https://status.openai.com/). |

**APIError**

An `APIError` 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](https://status.openai.com/) for more information.

If you encounter an APIError, 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](https://community.openai.com/) but be sure to omit any sensitive information.

Collapse‍

**Timeout**

A `Timeout` 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 Timeout 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.

Collapse‍

**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](https://platform.openai.com/docs/guides/rate-limits) 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.

Collapse‍

**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.

Collapse‍

**InvalidRequestError**

An 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 InvalidRequestError, 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](https://platform.openai.com/docs/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.

Collapse‍

**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.

Collapse‍

**ServiceUnavailableError**

A `ServiceUnavailableError` indicates that our servers are temporarily unable to handle your request. This could be due to a planned or unplanned maintenance, a system upgrade, or a server failure. These errors can also be returned during periods of high traffic.

We apologize for any inconvenience and we are working hard to restore our services as soon as possible.

If you encounter a ServiceUnavailableError, please try the following steps:

* Wait a few minutes and retry your request. Sometimes, the issue may be resolved quickly and your request may succeed on the next attempt.
* Check our [status page](https://status.openai.com/) 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.

Collapse‍

**[Persistent errors](https://platform.openai.com/docs/guides/error-codes/persistent-errors)**

If the issue persists, [contact our support team via chat](https://help.openai.com/en/) 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](https://community.openai.com/) but be sure to omit any sensitive information.

**[Handling errors](https://platform.openai.com/docs/guides/error-codes/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:

try:

#Make your OpenAI API request here

response = openai.Completion.create(prompt="Hello world",

model="text-davinci-003")

except openai.error.APIError as e:

#Handle API error here, e.g. retry or log

print(f"OpenAI API returned an API Error: {e}")

pass

except openai.error.APIConnectionError as e:

#Handle connection error here

print(f"Failed to connect to OpenAI API: {e}")

pass

except openai.error.RateLimitError as e:

#Handle rate limit error (we recommend using exponential backoff)

print(f"OpenAI API request exceeded rate limit: {e}")

pass

**[Safety best practices](https://platform.openai.com/docs/guides/safety-best-practices/safety-best-practices)**

**[Use our free Moderation API](https://platform.openai.com/docs/guides/safety-best-practices/use-our-free-moderation-api)**

OpenAI's [Moderation API](https://platform.openai.com/docs/guides/moderation) 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](https://platform.openai.com/docs/guides/safety-best-practices/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)](https://platform.openai.com/docs/guides/safety-best-practices/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](https://platform.openai.com/docs/guides/safety-best-practices/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)](https://platform.openai.com/docs/guides/safety-best-practices/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](https://platform.openai.com/docs/guides/safety-best-practices/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](https://platform.openai.com/docs/guides/safety-best-practices/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](https://platform.openai.com/docs/guides/safety-best-practices/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.

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](https://openai.com/security/disclosure/).

**[End-user IDs](https://platform.openai.com/docs/guides/safety-best-practices/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 identifer

python



Copy‍

response = openai.Completion.create(

model="text-davinci-003",

prompt="This is a test",

max\_tokens=5,

user="user123456"

)

# [Production best practices](https://platform.openai.com/docs/guides/production-best-practices/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.

## [Setting up your organization](https://platform.openai.com/docs/guides/production-best-practices/setting-up-your-organization)

Once you [log in](https://platform.openai.com/login) to your OpenAI account, you can find your organization name and ID in your [organization settings](https://platform.openai.com/account/org-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](https://platform.openai.com/docs/api-reference/requesting-organization) 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](https://platform.openai.com/account/api-keys) will be billed. You can change your default organization in your [user settings](https://platform.openai.com/account/api-keys).

You can invite new members to your organization from the [members](https://platform.openai.com/account/members) settings 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](https://platform.openai.com/docs/guides/production-best-practices/managing-billing-limits)

New free trial users receive an initial credit of $5 that expires after three months. Once the credit has been used or expires, you can choose to enter [billing information](https://platform.openai.com/account/billing/overview) to continue your use of the API. If no billing information is entered, you will still have login access but will be unable to make any further API requests.

Once you’ve entered your billing information, you will have an approved usage limit of $120 per month, which is set by OpenAI. To increase your quota beyond the $120 monthly billing limit, please submit a [quota increase request](https://platform.openai.com/forms/quota-increase).

If you’d like to be notified when your usage exceeds a certain amount, you can set a soft limit through the [usage limits](https://platform.openai.com/account/billing/limits) page. When the soft limit is reached, the owners of the organization will receive an email notification. You can also set a hard limit so that, once the hard limit 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](https://platform.openai.com/docs/guides/production-best-practices/api-keys)

The OpenAI API uses API keys for authentication. Visit your [API keys](https://platform.openai.com/account/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](https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety).

### [Staging accounts](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/managing-rate-limits)

When using our API, it's important to understand and plan for [rate limits](https://platform.openai.com/docs/guides/rate-limits).

## [Improving latencies](https://platform.openai.com/docs/guides/production-best-practices/improving-latencies)

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:

**Network**

End user to API latency

**Server**

Time to process prompt tokens

**Server**

Time to sample/generate tokens

**Network**

API to end user latency

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](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/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](https://help.openai.com/en/articles/5072263-how-do-i-use-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](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/batching)

Depending on your use case, batching *may help*. If you are sending multiple requests to the same endpoint, you can [batch the prompts](https://platform.openai.com/docs/guides/rate-limits/batching-requests) 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](https://platform.openai.com/docs/guides/production-best-practices/managing-costs)

To monitor your costs, you can set a soft limit in your account to receive an email alert once you pass a certain usage threshold. You can also set a hard limit. Please be mindful of the potential for a hard limit to cause disruptions to your application/users. Use the [usage tracking dashboard](https://platform.openai.com/account/usage) to monitor your token usage during the current and past billing cycles.

### [Text generation](https://platform.openai.com/docs/guides/production-best-practices/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](https://openai.com/api/pricing/), 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](https://platform.openai.com/docs/guides/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](https://platform.openai.com/tokenizer) 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](https://help.openai.com/en/articles/6614209-how-do-i-check-my-token-usage).

## [MLOps strategy](https://platform.openai.com/docs/guides/production-best-practices/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](https://platform.openai.com/docs/guides/production-best-practices/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](https://www.openai.com/security) and [trust and compliance portal](https://trust.openai.com/) provide our most comprehensive and up-to-date documentation. For reference, here is our [Privacy Policy](https://openai.com/privacy/) and [Terms of Use](https://openai.com/api/policies/terms/).

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](https://platform.openai.com/docs/guides/production-best-practices/safety-best-practices)

When creating your application with our API, consider our [safety best practices](https://platform.openai.com/docs/guides/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.