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. 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

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
- Ask the model to adopt a persona
- Use delimiters to clearly indicate distinct parts of the input
- Specify the steps required to complete a task
- Provide examples
- Specify the desired length of the output

Provide reference text

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

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
- For dialogue applications that require very long conversations, summarize or filter previous dialogue
- Summarize long documents piecewise and construct a full summary recursively

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
- Use inner monologue or a sequence of queries to hide the model's reasoning process
- Ask the model if it missed anything on previous passes

Use external tools

Compensate for the weaknesses of 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
- Use code execution to perform more accurate calculations or call external APIs
- Give the model access to specific functions

Test changes systematically

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

Tactics

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

Strategy: Write clear instructions

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

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

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

Tactic: Ask the model to adopt a persona

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

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

Open in Playground ↗

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

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

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

"""insert text here"""

Open in Playground ↗

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

USER <article> insert first article here </article>

<article> insert second article here </article>

Open in Playground ↗

SYSTEM You will be provided with a thesis abstract and a suggested title for it. The thesis title should give the reader a good idea of the topic of the thesis but should also be eye-catching. If the title does not meet these criteria, suggest 5 alternatives.

USER Abstract: insert abstract here

Title: insert title here

Open in Playground ↗

asking of them.

Tactic: Specify the steps required to complete a task

Some tasks are best specified as a sequence of steps. Writing the steps out explicitly can make it easier for the model to follow them.

SYSTEM	Use the following step-by-step instructions to respond to user inputs. Step 1 - The user will provide you with text in triple quotes. Summarize this text in one sentence with a prefix that says "Summary: ". Step 2 - Translate the summary from Step 1 into Spanish, with a prefix that says "Translation: ".
USER	"""insert text here"""

Open in Playground ↗

Tactic: Provide examples

Providing general instructions that apply to all examples is generally more efficient than demonstrating all permutations of a task by example, but in some cases providing examples may be easier. For example, if you intend for the model to copy a particular style of responding to user queries which is difficult to describe explicitly. This is known as "few-shot" prompting.

SYSTEM	Answer in a consistent style.
USER	Teach me about patience.
ASSISTANT	The river that carves the deepest valley flows from a modest spring; the grandest symphony originates from a single note; the most intricate tapestry begins with a solitary thread.
USER	Teach me about the ocean.

Open in Playground ↗

output length can be specified in terms of the count of words, sentences, paragraphs, bullet points, etc. Note however that instructing the model to generate a specific number of words does not work with high precision. The model can more reliably generate outputs with a specific number of paragraphs or bullet points.

USER Summarize the text delimited by triple quotes in about 50 words.

"""insert text here"""

Open in Playground ↗

USER Summarize the text delimited by triple quotes in 2 paragraphs.

"""insert text here"""

Open in Playground ↗

USER Summarize the text delimited by triple quotes in 3 bullet points.

"""insert text here"""

Open in Playground ↗

Strategy: Provide reference text

Tactic: Instruct the model to answer using a reference text

If we can provide a model with trusted information that is relevant to the current query, then we can instruct the model to use the provided information to compose its answer.

SYSTEM Use the provided articles delimited by triple quotes to answer questions. If the answer cannot be found in the articles, write "I could not find an answer."

Question: <insert question here>

Open in Playground ↗

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 can be used to implement efficient knowledge retrieval. See the tactic "Use embeddings-based search to implement efficient knowledge retrieval" for more details on how to implement this.

Tactic: Instruct the model to answer with citations from a reference text

If the input has been supplemented with relevant knowledge, it's straightforward to request that the model add citations to its answers by referencing passages from provided documents. Note that citations in the output can then be verified programmatically by string matching within the provided documents.

SYSTEM	You will be provided with a document delimited by triple quotes and a question. Your task is to answer the question using only the provided document and to cite the passage(s) of the document used to answer the question. If the document does not contain the information needed to answer this question then simply write: "Insufficient information." If an answer to the question is provided, it must be annotated with a citation. Use the following format for to cite relevant passages ({"citation": …}).
USER	"""<insert document here>""" Question: <insert question here>

Open in Playground ↗

Strategy: Split complex tasks into simpler subtasks

Tactic: Use intent classification to identify the most relevant instructions for a user query

For tasks in which lots of independent sets of instructions are needed to handle different cases, it can be beneficial to first classify the type of query and to use that classification to determine which instructions are needed. This can be achieved by defining fixed categories and hardcoding instructions that are relevant for handling tasks in a given lower error rates compared to using a single query to perform the whole task. This can also result in lower costs since larger prompts cost more to run (see pricing information).

Suppose for example that for a customer service application, queries could be usefully classified as follows:

SYSTEM

You will be provided with customer service queries. Classify each query into a primary category and a secondary category. Provide your output in json format with the keys: primary and secondary.

Primary categories: Billing, Technical Support, Account Management, or General Inquiry.

Billing secondary categories:
- Unsubscribe or upgrade
- Add a payment method
- Explanation for charge
- Dispute a charge

Technical Support secondary categories:
- Troubleshooting
- Device compatibility
- Software updates

Account Management secondary categories:
- Password reset
- Update personal information
- Close account
- Account security

General Inquiry secondary categories:
- Product information
- Pricing
- Feedback
- Speak to a human

USER

I need to get my internet working again.

Open in Playground ↗

requires help with "troubleshooting".

SYSTEM You will be provided with customer service inquiries that require troubleshooting in a technical support context. Help the user by:

- Ask them to check that all cables to/from the router are connected. Note that it is common for cables to come loose over time.
- If all cables are connected and the issue persists, ask them which router model they are using
- Now you will advise them how to restart their device:
-- If the model number is MTD-327J, advise them to push the red button and hold it for 5 seconds, then wait 5 minutes before testing the connection.
-- If the model number is MTD-327S, advise them to unplug and replug it, then wait 5 minutes before testing the connection.
- If the customer's issue persists after restarting the device and waiting 5 minutes, connect them to IT support by outputting {"IT support requested"}.
- If the user starts asking questions that are unrelated to this topic then confirm if they would like to end the current chat about troubleshooting and classify their request according to the following scheme:

<insert primary/secondary classification scheme from above here>

USER I need to get my internet working again.

Open in Playground ↗

Notice that the model has been instructed to emit special strings to indicate when the state of the conversation changes. This enables us to turn our system into a state machine where the state determines which instructions are injected. By keeping track of state, what instructions are relevant at that state, and also optionally what state transitions are allowed from that state, we can put guardrails around the user experience that would be hard to achieve with a less structured approach.

Tactic: For dialogue applications that require very long conversations, summarize or filter previous dialogue

Since 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.

summary of the prior conversation could be included as part of the system message. Alternatively, prior conversation could be summarized asynchronously in the background throughout the entire conversation.

An alternative solution is to dynamically select previous parts of the conversation that are most relevant to the current query. See the tactic "Use embeddings-based search to implement efficient knowledge retrieval".

Tactic: Summarize long documents piecewise and construct a full summary recursively

Since 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 by OpenAI using variants of GPT-3.

Strategy: Give GPTs time to "think"

Tactic: Instruct the model to work out its own solution before rushing to a conclusion

Sometimes we get better results when we explicitly instruct the model to reason from first principles before coming to a conclusion. Suppose for example we want a model to evaluate a student’s solution to a math problem. The most obvious way to approach this is to simply ask the model if the student's solution is correct or not.

SYSTEM	Determine if the student's solution is correct or not.
USER	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

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 ↗

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

total cost for the first year of operations is 360x + 100,000.

Open in Playground ↗

Tactic: Use inner monologue or a sequence of queries to hide the model's reasoning process

The previous tactic demonstrates that it is sometimes important for the model to reason in detail about a problem before answering a specific question. For some applications, the reasoning process that a model uses to arrive at a final answer would be inappropriate to share with the user. For example, in tutoring applications we may want to encourage students to work out their own answers, but a model's reasoning process about the student's solution could reveal the answer to the student.

Inner monologue is a tactic that can be used to mitigate this. The idea of inner monologue is to instruct the model to put parts of the output that are meant to be hidden from the user into a structured format that makes parsing them easy. Then before presenting the output to the user, the output is parsed and only part of the output is made visible.

SYSTEM

Follow these steps to answer the user queries.

Step 1 - First work out your own solution to the problem. Don't rely on the student's solution since it may be incorrect. Enclose all your work for this step within triple quotes (""").

Step 2 - Compare your solution to the student's solution and evaluate if the student's solution is correct or not. Enclose all your work for this step within triple quotes (""").

Step 3 - If the student made a mistake, determine what hint you could give the student without giving away the answer. Enclose all your work for this step within triple quotes (""").

Step 4 - If the student made a mistake, provide the hint from the previous step to the student (outside of triple quotes). Instead of writing "Step 4 - ..." write "Hint:".

USER

Problem Statement: <insert problem statement>

Student Solution: <insert student solution>

Alternatively, this can be achieved with a sequence of queries in which all except the last have their output hidden from the end user.

First, we can ask the model to solve the problem on its own. Since this initial query doesn't require the student's solution, it can be omitted. This provides the additional advantage that there is no chance that the model's solution will be biased by the student's attempted solution.

USER	<insert problem statement>

Open in Playground ↗

Next, we can have the model use all available information to assess the correctness of the student's solution.

SYSTEM	Compare your solution to the student's solution and evaluate if the student's solution is correct or not.
USER	Problem statement: """<insert problem statement>""" Your solution: """<insert model generated solution>""" Student's solution: """<insert student's solution>"""

Open in Playground ↗

Finally, we can let the model use its own analysis to construct a reply in the persona of a helpful tutor.

SYSTEM	You are a math tutor. If the student made an error, offer a hint to the student in a way that does not reveal the answer. If the student did not make an error, simply offer them an encouraging comment.
USER	Problem statement: """<insert problem statement>""" Your solution: """<insert model generated solution>""" Student's solution: """<insert student's solution>"""

Open in Playground ↗

Tactic: Ask the model if it missed anything on previous passes

Suppose that we are using a model to list excerpts from a source which are relevant to a particular question. After listing each excerpt the model needs to determine if it should start writing another or if it should stop. If the source document is large, it is common for a model to stop too early and fail to list all relevant excerpts. In that case, better performance can often be obtained by prompting the model with followup queries to find any excerpts it missed on previous passes.

SYSTEM

You will be provided with a document delimited by triple quotes. Your task is to select excerpts which pertain to the following question: "What significant paradigm shifts have occurred in the history of artificial intelligence."

Ensure that excerpts contain all relevant context needed to interpret them - in other words don't extract small snippets that are missing important context. Provide output in JSON format as follows:

[{"excerpt": "..."},
...
{"excerpt": "..."}]

USER

"""<insert document here>"""

ASSISTANT

[{"excerpt": "the model writes an excerpt here"},
...
{"excerpt": "the model writes another excerpt here"}]

USER

Are there more relevant excerpts? Take care not to repeat excerpts. Also ensure that excerpts contain all relevant context needed to interpret them - in other words don't extract small snippets that are missing important context.

Open in Playground ↗

Strategy: Use external tools

A model can leverage external sources of information if provided as part of its input. This can help the model to generate more informed and up-to-date responses. For example, if a user asks a question about a specific movie, it may be useful to add high quality information about the movie (e.g. actors, director, etc…) to the model's input. Embeddings can be used to implement efficient knowledge retrieval, so that relevant information can be added to the model input dynamically at run-time.

A text embedding is a vector that can measure the relatedness between text strings. Similar or relevant strings will be closer together than unrelated strings. This fact, along with the existence of fast vector search algorithms means that embeddings can be used to implement efficient knowledge retrieval. In particular, a text corpus can be split up into chunks, and each chunk can be embedded and stored. Then a given query can be embedded and vector search can be performed to find the embedded chunks of text from the corpus that are most related to the query (i.e. closest together in the embedding space).

Example implementations can be found in the OpenAI Cookbook. See the tactic "Instruct the model to use retrieved knowledge to answer queries" for an example of how to use knowledge retrieval to minimize the likelihood that a model will make up incorrect facts.

Tactic: Use code execution to perform more accurate calculations or call external APIs

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 ↗

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

SYSTEM

You can write and execute Python code by enclosing it in triple backticks. Also note that you have access to the following module to help users send messages to their friends:

~~~
```python
import message
message.write(to="John", message="Hey, want to meetup after work?")```
~~~

Open in Playground ↗

WARNING: Executing code produced by a model is not inherently safe and precautions should be taken in any application that seeks to do this. In particular, a sandboxed code execution environment is needed to limit the harm that untrusted code could cause.

Tactic: Give the model access to specific functions

The Chat Completions API allows passing a list of function descriptions in requests. This enables models to generate function arguments according to the provided schemas. Generated function arguments are returned by the API in JSON format and can be used to execute function calls. Output provided by function calls can then be fed back into a model in the following request to close the loop. This is the recommended way of using GPT models to call external functions. To learn more see the function calling section in our introductory GPT guide and more function calling examples in the OpenAI Cookbook.

Strategy: Test changes systematically

Sometimes it can be hard to tell whether a change — e.g., a new instruction or a new design — makes your system better or worse. Looking at a few examples may hint at which is better, but with small sample sizes it can be hard to distinguish between a true improvement or random luck. Maybe the change helps performance on some inputs, but hurts performance on others.

Evaluation procedures (or "evals") are useful for optimizing system designs. Good evals are:

- Representative of real-world usage (or at least diverse)
- Contain many test cases for greater statistical power (see table below for guidelines)
- Easy to automate or repeat

10%	~100
3%	~1,000
1%	~10,000

Evaluation of outputs can be done by computers, humans, or a mix. Computers can automate evals with objective criteria (e.g., questions with single correct answers) as well as some subjective or fuzzy criteria, in which model outputs are evaluated by other model queries. OpenAI Evals is an open-source software framework that provides tools for creating automated evals.

Model-based evals can be useful when there exists a range of possible outputs that would be considered equally high in quality (e.g. for questions with long answers). The boundary between what can be realistically evaluated with a model-based eval and what requires a human to evaluate is fuzzy and is constantly shifting as models become more capable. We encourage experimentation to figure out how well model-based evals can work for your use case.

Tactic: Evaluate model outputs with reference to gold-standard answers

Suppose it is known that the correct answer to a question should make reference to a specific set of known facts. Then we can use a model query to count how many of the required facts are included in the answer.

For example, using the following system message:

SYSTEM	You will be provided with text delimited by triple quotes that is supposed to be the answer to a question. Check if the following pieces of information are directly contained in the answer: - Neil Armstrong was the first person to walk on the moon. - The date Neil Armstrong first walked on the moon was July 21, 1969. For each of these points perform the following steps: 1 - Restate the point. 2 - Provide a citation from the answer which is closest to this point. 3 - Consider if someone reading the citation who doesn't know the topic could directly infer the point. Explain why or why not before making up your mind.

Finally, provide a count of how many "yes" answers there are. Provide this count as {"count": <insert count here>}.

Here's an example input where both points are satisfied:

SYSTEM	<insert system message above>
USER	"""Neil Armstrong is famous for being the first human to set foot on the Moon. This historic event took place on July 21, 1969, during the Apollo 11 mission."""

Open in Playground ↗

Here's an example input where only one point is satisfied:

SYSTEM	<insert system message above>
USER	"""Neil Armstrong made history when he stepped off the lunar module, becoming the first person to walk on the moon."""

Open in Playground ↗

Here's an example input where none are satisfied:

SYSTEM	<insert system message above>
USER	"""In the summer of '69, a voyage grand, Apollo 11, bold as legend's hand. Armstrong took a step, history unfurled, "One small step," he said, for a new world."""

Open in Playground ↗

There are many possible variants on this type of model-based eval. Consider the following variation which tracks the kind of overlap between the candidate answer and the gold-standard answer, and also tracks whether the candidate answer contradicts any part of the gold-standard answer.

Step 1: Reason step-by-step about whether the information in the submitted answer compared to the expert answer is either: disjoint, equal, a subset, a superset, or overlapping (i.e. some intersection but not subset/superset).

Step 2: Reason step-by-step about whether the submitted answer contradicts any aspect of the expert answer.

Step 3: Output a JSON object structured like: {"type_of_overlap": "disjoint" or "equal" or "subset" or "superset" or "overlapping", "contradiction": true or false}

Here's an example input with a substandard answer which nonetheless does not contradict the expert answer:

SYSTEM	<insert system message above>
USER	Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time.""" Submitted Answer: """Didn't he walk on the moon or something?""" Expert Answer: """Neil Armstrong is most famous for being the first person to walk on the moon. This historic event occurred on July 21, 1969."""

Open in Playground ↗

Here's an example input with answer that directly contradicts the expert answer:

SYSTEM	<insert system message above>
USER	Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time.""" Submitted Answer: """On the 21st of July 1969, Neil Armstrong became the second person to walk on the moon, following after Buzz Aldrin."""

1969.

Open in Playground ↗

Here's an example input with a correct answer that also provides a bit more detail than is necessary:

SYSTEM	<insert system message above>
USER	Question: """What event is Neil Armstrong most famous for and on what date did it occur? Assume UTC time.""" Submitted Answer: """At approximately 02:56 UTC on July 21st 1969, Neil Armstrong became the first human to set foot on the lunar surface, marking a monumental achievement in human history.""" Expert Answer: """Neil Armstrong is most famous for being the first person to walk on the moon. This historic event occurred on July 21, 1969."""

Open in Playground ↗

Fine-tuning

Learn how to customize a model for your application.

Introduction

On July 6, 2023, we announced 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 to learn more about how fine-tuned model training and usage are billed.

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 to add additional data without having to start from scratch.

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

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

Your data must be a JSONL document, where each line is a prompt-completion pair corresponding to a training example. You can use our CLI data preparation tool to easily convert your data into this file format.

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

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 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

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

The following assumes you've already prepared training data following the above instructions.

Start your fine-tuning job using the OpenAI CLI:

```
openai api fine_tunes.create -t <TRAIN_FILE_ID_OR_PATH> -m <BASE_MO
```

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.

Running the above command does several things:

1. Uploads the file using the files API (or uses an already-uploaded file)
2. Creates a fine-tune job

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 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.

```
# 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 fai
# 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

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, and make requests to it using the 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 api completions.create -m <FINE_TUNED_MODEL> -p <YOUR_PROMPT
```

cURL:

```
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:

```
import openai
openai.Completion.create(
    model=FINE_TUNED_MODEL,
    prompt=YOUR_PROMPT)
```

Node.js:

```
const response = await openai.createCompletion({
  model: FINE_TUNED_MODEL
  prompt: YOUR_PROMPT,
});
```

You may continue to use all the other Completions parameters like `temperature`, `frequency_penalty`, `presence_penalty`, etc, on these requests to fine-tuned models.

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:

Python:

```
import openai
openai.Model.delete(FINE_TUNED_MODEL)
```

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 for your use case below.**

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, 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

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.

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

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

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. 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

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\nAc      s
{"prompt":"Company: Loft conversion specialists\nProduct: -\nAd:Straight
```

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.

```
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 Ta
    "max_tokens": 1,
    "model": "YOUR_FINE_TUNED_MODEL_NAME"
  }'
```

Which will return either `yes` or `no`.

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:

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 po
    "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,
    ]
}
```

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:<d
    "completion": " <numerical_category>"
}
```

For example:

```
{
    "prompt": "Subject: Update my address\nFrom:Joe Doe\nTo:support@ou
    "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

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`

- 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

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 Andr
    "completion": "Looking for a smartphone that can do it all? Look n
}
```

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

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:

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

For example:

```
{
    "prompt": "Portugal will be removed from the UK's green travel lis
    "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

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    or
{"prompt":"Summary: <summary of the interaction so far>\n\nSpecific infor
```

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

```
{
    "prompt": "Item=handbag, Color=army_green, price=$99, size=S-> ,
    "completion": " This stylish small green handbag will add a unique
}
```

Won't work as well as:

```
{
    "prompt": "Item is a handbag. Colour is army green. Price is midra
    "completion": " This stylish small green handbag will add a unique
}
```

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

Customize your model name

You can add a suffix of up to 40 characters to your fine-tuned model name using the suffix parameter.

OpenAI CLI:

```
openai api fine_tunes.create -t test.jsonl -m ada --suffix "custom     l
```

The resulting name would be:

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

Analyzing your fine-tuned model

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

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.

```
# 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 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.

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

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 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

`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

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

You can sync your fine-tunes with Weights & Biases to track experiments, models, and datasets.

To get started, you will need a Weights & Biases 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
```

```
openai wandb sync
```

You can read the Weights & Biases documentation for more information on this integration.

GPT models

Looking for ChatGPT? Head to 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

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:

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

See the full API reference documentation here.

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 model's token limit, it will need to be shortened in some way.

To mimic the effect seen in ChatGPT where the text is returned iteratively, set the stream parameter to true.

Chat completions response format

An example chat completions API response looks as follows:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "The 2020 World Series was played in Texas at Glob
        "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 parameter

- `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

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 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)`

1 Call the model with the user query and a set of functions defined in the functions parameter.

2 The model can choose to call 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
    functions = [
        {
            "name": "get_current_weather",
            "description": "Get the current weather in a given locati
            "parameters": {
                "type": "object",
                "properties": {
                        "description": "The city and state, e.g. San
                    },
                    "unit": {"type": "string", "enum": ["celsius", "f
                },
                "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 explic
    )
    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
        available_functions = {
            "get_current_weather": get_current_weather,
        }  # only one function in this example, but you can have mult
        function_name = response_message["function_call"]["name"]
        fuction_to_call = available_functions[function_name]
        function_args = json.loads(response_message["function_call"][
        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 res
        messages.append(response_message)  # extend conversation with
        messages.append(
            {
                "role": "function",
                "name": function_name,
                "content": function_response,
            }
        )  # extend conversation with function response
        second_response = openai.ChatCompletion.create(
        )  # get a new response from GPT where it can see the functio
        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

Learn from more examples demonstrating function calling

Completions API Legacy

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

An example API call looks as follows:

```
response = openai.Completion.create(
  model="text-davinci-003",
  prompt="Write a tagline for an ice cream shop."
)
```

See the full API reference documentation to learn more.

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 field. This can be useful in some cases to assess the confidence of the model in its output.

Inserting text

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

DEEP DIVE

Inserting text

Completions response format

An example completions API response looks as follows:

```
{
  "choices": [
    {
      "finish_reason": "length",
      "index": 0,
      "logprobs": null,
      "text": "\n\n\"Let Your Sweet Tooth Run Wild at Our Creamy Ice
    }
  ],
  "created": 1683130927,
  "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

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     re
```

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

The difference between these APIs 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.

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. In particular, `gpt-4` is more 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 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

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 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.

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 page for more information).

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 Python library. Example code can be found in the OpenAI Cookbook's guide on how to count tokens with tiktoken.

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

If a conversation has too many tokens to fit within a model's maximum limit (e.g., more than 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

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?

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.

and `ada`). See the fine-tuning guide for more details on how to use fine-tuned models.

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

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

How can I make my application more safe?

If you want to add a moderation layer to the outputs of the Chat API, you can follow our moderation guide to prevent content that violates OpenAI's usage policies from being shown.

Prompt templates

Prompt templates are pre-defined recipes for generating prompts for language models.

A template may include instructions, few shot examples, and specific context and questions appropriate for a given task.

LangChain provides tooling to create and work with prompt templates.

LangChain strives to create model agnostic templates to make it easy to reuse existing templates across different language models.

Typically, language models expect the prompt to either be a string or else a list of chat messages.

Prompt template

Use `PromptTemplate` to create a template for a string prompt.

By default, `PromptTemplate` uses Python's str.format syntax for templating; however other templating syntax is available (e.g., `jinja2`).

```
from langchain import PromptTemplate

prompt_template = PromptTemplate.from_template(
    "Tell me a {adjective} joke about {content}."
)
prompt_template.format(adjective="funny", content="chickens")
```

```
"Tell me a funny joke about chickens."
```

The template supports any number of variables, including no variables:

```
from langchain import PromptTemplate

prompt_template = PromptTemplate.from_template(
"Tell me a joke"
)
prompt_template.format()
```

For additional validation, specify `input_variables` explicitly. These variables will be compared against the variables present in the template string during instantiation, raising an exception if there is a mismatch; for example,

```
from langchain import PromptTemplate

invalid_prompt = PromptTemplate(
    input_variables=["adjective"],
    template="Tell me a {adjective} joke about {content}."
)
```

You can create custom prompt templates that format the prompt in any way you want. For more information, see Custom Prompt Templates.

Chat prompt template

The prompt to Chat Models is a list of chat messages.

Each chat message is associated with content, and an additional parameter called `role`. For example, in the OpenAI Chat Completions API, a chat message can be associated with an AI assistant, a human or a system role.

Create a chat prompt template like this:

```
from langchain.prompts import ChatPromptTemplate

template = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful AI bot. Your name is {name}."),
    ("human", "Hello, how are you doing?"),
    ("ai", "I'm doing well, thanks!"),
    ("human", "{user_input}"),
])

messages = template.format_messages(
    name="Bob",
    user_input="What is your name?"
)
```

`ChatPromptTemplate.from_messages` accepts a variety of message representations.

For example, in addition to using the 2-tuple representation of (type, content) used above, you could pass in an instance of `MessagePromptTemplate` or `BaseMessage`.

```
from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage,
HumanMessagePromptTemplate

template = ChatPromptTemplate.from_messages(
    [
        SystemMessage(
            content=(
                "You are a helpful assistant that re-writes the user's
text to "
                "sound more upbeat."
            )
        ),
        HumanMessagePromptTemplate.from_template("{text}"),
    ]
)

from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI()
llm(template.format_messages(text='i dont like eating tasty things.'))
```

```
AIMessage(content='I absolutely adore indulging in delicious treats!',
additional_kwargs={}, example=False)
```

This provides you with a lot of flexibility in how you construct your chat prompts.