Prompt engineering

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

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

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

Prompt examples
Explore prompt examples to learn what GPT models can do

Six strategies for getting better results

Write clear instructions

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

Tactics:

- Include details in your query to get more relevant answers
- Ask the model to adopt a persona
- Use delimiters to clearly indicate distinct parts of the input

- Specify the steps required to complete a task
- Provide examples
- Specify the desired length of the output

Provide reference text

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

Tactics:

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

Split complex tasks into simpler subtasks

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

Tactics:

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

Give the model time to "think"

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

Tactics:

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

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

Use external tools

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

Tactics:

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

Test changes systematically

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

SYSTEM	When I ask for help to write something, you will reply with a document that contains at least one joke or playful comment in every paragraph.
USER	Write a thank you note to my steel bolt vendor for getting the delivery in on time and in short notice. This made it possible for us to deliver an important order.

Open in Playground ↗

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

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

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

Open in Playground ↗

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

Open in Playground ↗

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

Open in Playground ↗

For straightforward tasks such as these, using delimiters might not make a difference in the output quality. However, the more complex a task is the more important it is to disambiguate task details. Don't make the model work to understand exactly what you are asking of them.

Tactic: Specify the steps required to complete a task

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

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

Open in Playground ↗

Tactic: Provide examples

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

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

Open in Playground ↗

Tactic: Specify the desired length of the output

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

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

"""insert text here"""

Open in Playground ↗

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

"""insert text here"""

Open in Playground ↗

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

"""insert text here"""

Open in Playground ↗

Strategy: Provide reference text

Tactic: Instruct the model to answer using a reference text

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

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

USER <insert articles, each delimited by triple quotes>

Question: <insert question here>

Open in Playground ↗

Given that all models have limited context windows, we need some way to dynamically lookup information that is relevant to the question being asked. Embeddings can be used

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

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

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

SYSTEM	You will be provided with a document delimited by triple quotes and a question. Your task is to answer the question using only the provided document and to cite the passage(s) of the document used to answer the question. If the document does not contain the information needed to answer this question then simply write: "Insufficient information." If an answer to the question is provided, it must be annotated with a citation. Use the following format for to cite relevant passages ({"citation": ...}).
USER	"""<insert document here>""" 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 category. This process can also be applied recursively to decompose a task into a sequence of stages. The advantage of this approach is that each query will contain only those instructions that are required to perform the next stage of a task which can result in lower error rates compared to using a single query to perform the whole task. This can also result in lower costs since larger prompts cost more to run (see pricing information).

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

SYSTEM

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

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

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

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

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

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

USER

I need to get my internet working again.

Open in Playground ↗

Based on the classification of the customer query, a set of more specific instructions can be provided to a model for it to handle next steps. For example, suppose the customer requires help with "troubleshooting".

SYSTEM	You will be provided with customer service inquiries that require troubleshooting in a technical support context. Help the user by: - Ask them to check that all cables to/from the router are connected. Note that it is common for cables to come loose over time. - If all cables are connected and the issue persists, ask them which router model they are using - Now you will advise them how to restart their device: -- If the model number is MTD-327J, advise them to push the red button and hold it for 5 seconds, then wait 5 minutes before testing the connection. -- If the model number is MTD-327S, advise them to unplug and replug it, then wait 5 minutes before testing the connection. - If the customer's issue persists after restarting the device and waiting 5 minutes, connect them to IT support by outputting {"IT support requested"}. - If the user starts asking questions that are unrelated to this topic then confirm if they would like to end the current chat about troubleshooting and classify their request according to the following scheme: <insert primary/secondary classification scheme from above here>
USER	I need to get my internet working again.

Open in Playground ↗

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

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

Since models have a fixed context length, dialogue between a user and an assistant in which the entire conversation is included in the context window cannot continue indefinitely.

There are various workarounds to this problem, one of which is to summarize previous turns in the conversation. Once the size of the input reaches a predetermined threshold length, this could trigger a query that summarizes part of the conversation and the summary of the prior conversation could be included as part of the system message. Alternatively, prior conversation could be summarized asynchronously in the background throughout the entire conversation.

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

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

Since models have a fixed context length, they cannot be used to summarize a text longer than the context length minus the length of the generated summary in a single query.

To summarize a very long document such as a book we can use a sequence of queries to summarize each section of the document. Section summaries can be concatenated and summarized producing summaries of summaries. This process can proceed recursively until an entire document is summarized. If it's necessary to use information about earlier sections in order to make sense of later sections, then a further trick that can be useful is to include a running summary of the text that precedes any given point in the book while summarizing content at that point. The effectiveness of this procedure for summarizing books has been studied in previous research by OpenAI using variants of GPT-3.

Strategy: Give models time to "think"

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

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

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

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 ↗

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

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

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

SYSTEM

Follow these steps to answer the user queries.

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

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

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

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

USER Problem Statement: <insert problem statement>

Student Solution: <insert student solution>

Open in Playground ↗

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

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

USER <insert problem statement>

Open in Playground ↗

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

SYSTEM Compare your solution to the student's solution and evaluate if the student's solution is correct or not.

USER Problem statement: """<insert problem statement>"""

Your solution: """<insert model generated solution>"""

Student's solution: """<insert student's solution>"""

Open in Playground ↗

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

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

Open in Playground ↗

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

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

SYSTEM	You will be provided with a document delimited by triple quotes. Your task is to select excerpts which pertain to the following question: "What significant paradigm shifts have occurred in the history of artificial intelligence." Ensure that excerpts contain all relevant context needed to interpret them - in other words don't extract small snippets that are missing important context. Provide output in JSON format as follows: [{"excerpt": "..."}, ... {"excerpt": "..."}]
USER	"""<insert document here>"""
ASSISTANT	[{"excerpt": "the model writes an excerpt here"}, ...

{"excerpt": "the model writes another excerpt here"}]

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

Open in Playground ↗

Strategy: Use external tools

Tactic: Use embeddings-based search to implement efficient knowledge retrieval

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

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

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

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

Language models cannot be relied upon to perform arithmetic or long calculations accurately on their own. In cases where this is needed, a model can be instructed to write and run code instead of making its own calculations. In particular, a model can be instructed to put code that is meant to be run into a designated format such as triple

backtick. After an output is produced, the code can be extracted and run. Finally, if necessary, the output from the code execution engine (i.e. Python interpreter) can be provided as an input to the model for the next query.

SYSTEM	You can write and execute Python code by enclosing it in triple backticks, e.g. ```code goes here```. Use this to perform calculations.
USER	Find all real-valued roots of the following polynomial: 3*x**5 - 5*x**4 - 3*x**3 - 7*x - 10.

Open in Playground ↗

Another good use case for code execution is calling external APIs. If a model is instructed in the proper use of an API, it can write code that makes use of it. A model can be instructed in how to use an API by providing it with documentation and/or code samples showing how to use the API.

SYSTEM	You can write and execute Python code by enclosing it in triple backticks. Also note that you have access to the following module to help users send messages to their friends: ```python import message message.write(to="John", message="Hey, want to meetup after work?")```

Open in Playground ↗

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

Tactic: Give the model access to specific functions

The Chat Completions API allows passing a list of function descriptions in requests. This enables models to generate function arguments according to the provided schemas. Generated function arguments are returned by the API in JSON format and can be used to execute function calls. Output provided by function calls can then be fed back into a

model in the following request to close the loop. This is the recommended way of using OpenAI models to call external functions. To learn more see the function calling section in our introductory text generation guide and more function calling examples in the OpenAI Cookbook.

Strategy: Test changes systematically

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

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

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

DIFFERENCE TO DETECT	SAMPLE SIZE NEEDED FOR 95% CONFIDENCE
30%	~10
10%	~100
3%	~1,000
1%	~10,000

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

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

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

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

For example, using the following system message:

SYSTEM

You will be provided with text delimited by triple quotes that is supposed to be the answer to a question. Check if the following pieces of information are directly contained in the answer:

- Neil Armstrong was the first person to walk on the moon.
- The date Neil Armstrong first walked on the moon was July 21, 1969.

For each of these points perform the following steps:

1 - Restate the point.
2 - Provide a citation from the answer which is closest to this point.
3 - Consider if someone reading the citation who doesn't know the topic could directly infer the point. Explain why or why not before making up your mind.
4 - Write "yes" if the answer to 3 was yes, otherwise write "no".

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

Open in Playground ↗

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

SYSTEM <insert system message above>

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

Open in Playground ↗

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

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

Open in Playground ↗

Here's an example input where none are satisfied:

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

Open in Playground ↗

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

SYSTEM	Use the following steps to respond to user inputs. Fully restate each step before proceeding. i.e. "Step 1: Reason...". Step 1: Reason step-by-step about whether the information in the submitted answer compared to the expert answer is either: disjoint, equal, a subset, a superset, or overlapping (i.e. some intersection but not subset/superset). Step 2: Reason step-by-step about whether the submitted answer contradicts any aspect of the expert answer. Step 3: Output a JSON object structured like: {"type_of_overlap":

"disjoint" or "equal" or "subset" or "superset" or "overlapping", "contradiction": true or false}

Open in Playground ↗

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

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

Open in Playground ↗

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

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

Open in Playground ↗

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

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

Open in Playground ↗

Other resources

For more inspiration, visit the OpenAI Cookbook, which contains example code and also links to third-party resources such as:

- Prompting libraries & tools
- Prompting guides