OpenAI Platform

Text generation and prompting

Copy page

Learn how to prompt a model to generate text.

With the OpenAI API, you can use a large language model to generate text from a prompt, as you might using ChatGPT. Models can generate almost any kind of text response—like code, mathematical equations, structured JSON data, or human-like prose.

Here's a simple example using the Responses API.

Generate text from a simple prompt — javascript

```javascript
import OpenAI from "openai";
const client = new OpenAI();

const response = await client.responses.create({
    model: "gpt-4.1",
    input: "Write a one-sentence bedtime story about a unicorn."
});

console.log(response.output_text);
```

An array of content generated by the model is in the `output` property of the response. In this simple example, we have just one output which looks like this:

```json
[
    {
        "id": "msg_67b73f697ba4819183a15cc17d011509",
        "type": "message",
        "role": "assistant",
        "content": [
            {
                "type": "output_text",
                "text": "Under the soft glow of the moon, Luna the unicorn danced thr
                "annotations": []
            }
        ]
    }
]
```

The `output` array often has more than one item in it! It can contain tool calls, data about reasoning tokens generated by reasoning models, and other items. It is not safe to assume that the model's text output is present at `output[0].content[0].text`.

Some of our official SDKs include an `output_text` property on model responses for convenience, which aggregates all text outputs from the model into a single string. This may be useful as a shortcut to access text output from the model.

In addition to plain text, you can also have the model return structured data in JSON format - this feature is called **Structured Outputs**.

Choosing a model

A key choice to make when generating content through the API is which model you want to use - the `model` parameter of the code samples above. You can find a full listing of available models here. Here are a few factors to consider when choosing a model for text generation.

- **Reasoning models** generate an internal chain of thought to analyze the input prompt, and excel at understanding complex tasks and multi-step planning. They are also generally slower and more expensive to use than GPT models.

- **GPT models** are fast, cost-efficient, and highly intelligent, but benefit from more explicit instructions around how to accomplish tasks.
- **Large and small (mini or nano) models** offer trade-offs for speed, cost, and intelligence. Large models are more effective at understanding prompts and solving problems across domains, while small models are generally faster and cheaper to use.

When in doubt, `gpt-4.1` offers a solid combination of intelligence, speed, and cost effectiveness.

Prompt engineering

Prompt engineering is the process of writing effective instructions for a model, such that it consistently generates content that meets your requirements.

Because the content generated from a model is non-deterministic, it is a combination of art and science to build a prompt that will generate content in the format you want. However, there are a number of techniques and best practices you can apply to consistently get good results from a model.

Some prompt engineering techniques will work with every model, like using message roles. But different model types (like reasoning versus GPT models) might need to be prompted differently to produce the best results. Even different snapshots of models within the same family could produce different results. So as you are building more complex applications, we strongly recommend that you:

- Pin your production applications to specific model snapshots (like `gpt-4.1-2025-04-14` for example) to ensure consistent behavior.
- Build evals that will measure the behavior of your prompts, so that you can monitor the performance of your prompts as you iterate on them, or when you change and upgrade model versions.

Now, let's examine some tools and techniques available to you to construct prompts.

Message roles and instruction following

You can provide instructions to the model with differing levels of authority using the `instructions` API parameter or **message roles**.

The `instructions` parameter gives the model high-level instructions on how it should behave while generating a response, including tone, goals, and examples of correct responses. Any instructions provided this way will take priority over a prompt in the `input` parameter.

Generate text with instructions javascript

```javascript
import OpenAI from "openai";
const client = new OpenAI();

const response = await client.responses.create({
    model: "gpt-4.1",
    instructions: "Talk like a pirate.",
    input: "Are semicolons optional in JavaScript?",
});

console.log(response.output_text);
```

The example above is roughly equivalent to using the following input messages in the `input` array:

Generate text with messages using different roles javascript

```javascript
import OpenAI from "openai";
const client = new OpenAI();


const response = await client.responses.create({
    model: "gpt-4.1",
    input: [
        {
            role: "developer",
            content: "Talk like a pirate."
        },
        {
            role: "user",
            content: "Are semicolons optional in JavaScript?",
        },
    ],
});

console.log(response.output_text);
```

ⓘ Note that the `instructions` parameter only applies to the current response generation request. If you are managing conversation state with the `previous_response_id` parameter, the `instructions` used on previous turns will not be present in the context.

The OpenAI model spec describes how our models give different levels of priority to messages with different roles.

DEVELOPER	USER	ASSISTANT
developer messages are instructions provided by the application developer, prioritized ahead of user messages.	user messages are instructions provided by an end user, prioritized behind developer messages.	Messages generated by the model have the assistant role.

A multi-turn conversation may consist of several messages of these types, along with other content types provided by both you and the model. Learn more about managing conversation state here.

You could think about `developer` and `user` messages like a function and its arguments in a programming language.

- `developer` messages provide the system's rules and business logic, like a function definition.
- `user` messages provide inputs and configuration to which the `developer` message instructions are applied, like arguments to a function.

Message formatting with Markdown and XML

When writing `developer` and `user` messages, you can help the model understand logical boundaries of your prompt and context data using a combination of Markdown formatting and XML tags.

Markdown headers and lists can be helpful to mark distinct sections of a prompt, and to communicate hierarchy to the model. They can also potentially make your prompts more readable during development. XML tags can help delineate where one piece of content (like a supporting document used for reference) begins and ends. XML attributes can also be used to define metadata about content in the prompt that can be referenced by your instructions.

In general, a developer message will contain the following sections, usually in this order (though the exact optimal content and order may vary by which model you are using):

- **Identity:** Describe the purpose, communication style, and high-level goals of the assistant.
- **Instructions:** Provide guidance to the model on how to generate the response you want. What rules should it follow? What should the model do, and what should the model never do? This section could contain many subsections as relevant for your use case, like how the model should call custom functions.
- **Examples:** Provide examples of possible inputs, along with the desired output from the model.
- **Context:** Give the model any additional information it might need to generate a response, like private/proprietary data outside its training data, or any other data you know will be

particularly relevant. This content is usually best positioned near the end of your prompt, as you may include different context for different generation requests.

Below is an example of using Markdown and XML tags to construct a `developer` message with distinct sections and supporting examples.

Example prompt API request

A developer message for code generation

```
# Identity

You are coding assistant that helps enforce the use of snake case
variables in JavaScript code, and writing code that will run in
Internet Explorer version 6.

# Instructions

* When defining variables, use snake case names (e.g. my_variable)
  instead of camel case names (e.g. myVariable).
* To support old browsers, declare variables using the older
  "var" keyword.
* Do not give responses with Markdown formatting, just return
  the code as requested.

# Examples

<user_query>
How do I declare a string variable for a first name?
</user_query>

<assistant_response>
var first_name = "Anna";
</assistant_response>
```

Save on cost and latency with prompt caching

When constructing a message, you should try and keep content that you expect to use over and over in your API requests at the beginning of your prompt, **and** among the first API parameters you pass in the JSON request body to Chat Completions or Responses. This enables you to maximize cost and latency savings from prompt caching.

Few-shot learning

Few-shot learning lets you steer a large language model toward a new task by including a handful of input/output examples in the prompt, rather than fine-tuning the model. The model implicitly "picks up" the pattern from those examples and applies it to a prompt. When providing examples, try to show a diverse range of possible inputs with the desired outputs.

Typically, you will provide examples as part of a `developer` message in your API request. Here's an example `developer` message containing examples that show a model how to classify positive or negative customer service reviews.

```
# Identity

You are a helpful assistant that labels short product reviews as
Positive, Negative, or Neutral.

# Instructions

* Only output a single word in your response with no additional formatting
  or commentary.
* Your response should only be one of the words "Positive", "Negative", or
  "Neutral" depending on the sentiment of the product review you are given.

```

```
# Examples

<product_review id="example-1">
I absolutely love this headphones — sound quality is amazing!
</product_review>

<assistant_response id="example-1">
Positive
</assistant_response>

<product_review id="example-2">
Battery life is okay, but the ear pads feel cheap.
</product_review>

<assistant_response id="example-2">
Neutral
</assistant_response>

<product_review id="example-3">
Terrible customer service, I'll never buy from them again.
</product_review>

<assistant_response id="example-3">
Negative
</assistant_response>
```

Include relevant context information

It is often useful to include additional context information the model can use to generate a response within the prompt you give the model. There are a few common reasons why you might do this:

- To give the model access to proprietary data, or any other data outside the data set the model was trained on.
- To constrain the model's response to a specific set of resources that you have determined will be most beneficial.

The technique of adding additional relevant context to the model generation request is sometimes called **retrieval-augmented generation (RAG)**. You can add additional context to the prompt in many different ways, from querying a vector database and including the text you get back into a prompt, or by using OpenAI's built-in file search tool to generate content based on uploaded documents.

Planning for the context window

Models can only handle so much data within the context they consider during a generation request. This memory limit is called a **context window**, which is defined in terms of tokens (chunks of data you pass in, from text to images).

Models have different context window sizes from the low 100k range up to one million tokens for newer GPT-4.1 models. Refer to the model docs for specific context window sizes per model.

Prompting GPT-4.1 models

GPT models like `gpt-4.1` benefit from precise instructions that explicitly provide the logic and data required to complete the task in the prompt. GPT-4.1 in particular is highly steerable and responsive to well-specified prompts. To get the most out of GPT-4.1, refer to the prompting guide in the cookbook.

GPT-4.1 prompting guide

Get the most out of prompting GPT-4.1 with the tips and tricks in this prompting guide, extracted from real-world use cases and practical experience.

GPT-4.1 prompting best practices

While the cookbook has the best and most comprehensive guidance for prompting this model, here are a few best practices to keep in mind.

Building agentic workflows

Using long context

GPT-4.1 has a performant 1M token input context window, and will be useful for a variety of long context tasks, including structured document parsing, re-ranking, selecting relevant information while ignoring irrelevant context, and performing multi-hop reasoning using context.

Optimal Context Size

We show perfect performance at needle-in-a-haystack evals up to our full context size, and we've observed very strong performance at complex tasks with a mix of relevant and irrelevant code and documents in the range of hundreds of thousands of tokens.

Delimiters

We tested a variety of delimiters for separating context provided to the model against our long context evals. Briefly, XML and the format demonstrated by Lee et al. (ref) tend to perform well, while JSON performed worse for this task. See our cookbook for prompt examples.

Prompt Organization

Especially in long context usage, placement of instructions and context can substantially impact performance. In our experiments, we found that it was optimal to put critical instructions, including the user query, at both the top and the bottom of the prompt; this elicited marginally better performance from the model than putting them only at the top, and much better performance than only at the bottom.

Prompting for chain of thought

As mentioned above, GPT-4.1 isn't a reasoning model, but prompting the model to think step by step (called "chain of thought") can be an effective way for a model to break down problems into more manageable pieces. The model has been trained to perform well at agentic reasoning and real-world problem solving, so it shouldn't require much prompting to do well.

We recommend starting with this basic chain-of-thought instruction at the end of your prompt:

```
First, think carefully step by step about what documents are needed to answer the
```

From there, you should improve your CoT prompt by auditing failures in your particular examples and evals, and addressing systematic planning and reasoning errors with more explicit instructions. See our cookbook for a prompt example demonstrating a more opinionated reasoning strategy.

Instruction following

GPT-4.1 exhibits outstanding instruction-following performance, which developers can leverage to precisely shape and control the outputs for their particular use cases. However, since the model follows instructions more literally than its predecessors, may need to provide more explicit specification around what to do or not do, and existing prompts optimized for other models may not immediately work with this model.

Recommended Workflow

Here is our recommended workflow for developing and debugging instructions in prompts:

- Start with an overall “Response Rules” or “Instructions” section with high-level guidance and bullet points.
- If you'd like to change a more specific behavior, add a section containing more details for that category, like `## Sample Phrases`.
- If there are specific steps you'd like the model to follow in its workflow, add an ordered list and instruct the model to follow these steps.
- If behavior still isn't working as expected, check for conflicting, underspecified, or incorrect` instructions and examples. If there are conflicting instructions, GPT-4.1 tends to follow the one closer to the end of the prompt.
- Add examples that demonstrate desired behavior; ensure that any important behavior demonstrated in your examples are also cited in your rules.
- It's generally not necessary to use all-caps or other incentives like bribes or tips, but developers can experiment with this for extra emphasis if so desired.

Common Failure Modes

These failure modes are not unique to GPT-4.1, but we share them here for general awareness and ease of debugging.

- Instructing a model to always follow a specific behavior can occasionally induce adverse effects. For instance, if told “you must call a tool before responding to the user,” models may hallucinate tool inputs or call the tool with null values if they do not have enough information. Adding “if you don’t have enough information to call the tool, ask the user for the information you need” should mitigate this.
- When provided sample phrases, models can use those quotes verbatim and start to sound repetitive to users. Ensure you instruct the model to vary them as necessary.
- Without specific instructions, some models can be eager to provide additional prose to explain their decisions, or output more formatting in responses than may be desired. Provide instructions and potentially examples to help mitigate.

See our cookbook for an example customer service prompt that demonstrates these principles.

Prompting reasoning models

There are some differences to consider when prompting a reasoning model versus prompting a GPT model. Generally speaking, reasoning models will provide better results on tasks with only high-level guidance. This differs from GPT models, which benefit from very precise instructions.

You could think about the difference between reasoning and GPT models like this.

- A reasoning model is like a senior co-worker. You can give them a goal to achieve and trust them to work out the details.
- A GPT model is like a junior coworker. They'll perform best with explicit instructions to create a specific output.

For more information on best practices when using reasoning models, refer to this guide.

Next steps

Now that you known the basics of text inputs and outputs, you might want to check out one of these resources next.

Build a prompt in the Playground
Use the Playground to develop and iterate on prompts.

Generate JSON data with Structured Outputs
Ensure JSON data emitted from a model conforms to a JSON schema.

Full API reference
Check out all the options for text generation in the API reference.