Text generation

Copy page

Learn how to generate text from a prompt.

OpenAI provides simple APIs to use a large language model to generate text from a prompt, as you might using ChatGPT. These models have been trained on vast quantities of data to understand multimedia inputs and natural language instructions. From these prompts, models can generate almost any kind of text response, like code, mathematical equations, structured JSON data, or human-like prose.

Quickstart

To generate text, you can use the chat completions endpoint in the REST API, as seen in the examples below. You can either use the REST API from the HTTP client of your choice, or use one of OpenAI's official SDKs for your preferred programming language.

Generate prose | Analyze an image | Generate JSON data

Create a human-like response to a prompt — javascript

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

const completion = await openai.chat.completions.create({
    model: "gpt-4o",
    messages: [
        { role: "developer", content: "You are a helpful assistant." },
        {
            role: "user",
            content: "Write a haiku about recursion in programming.",
        },
    ],
    store: true,
});

console.log(completion.choices[0].message);
```

Choosing a model

When making a text generation request, your first decision is which model you want to generate the response. The model you choose influences output and impacts cost.

A **large model** like `gpt-4o` offers a very high level of intelligence and strong performance, with higher cost per token.

A **small model** like `gpt-4o-mini` offers intelligence not quite on the level of the larger model, but it's faster and less expensive per token.

A **reasoning model** like the `o1` family of models is slower to return a result, and uses more tokens to "think," but is capable of advanced reasoning, coding, and multi-step planning.

Experiment with different models in the Playground to see which works best for your prompts! You might also benefit from our model selection best practices.

Building prompts

The process of crafting prompts to get the right output from a model is called **prompt engineering**. You can improve output by giving the model precise instructions, examples, and necessary context information—like private or specialized information not included in the model's training data.

Below is high-level guidance on building prompts. For more in-depth strategies and tactics, see the prompt engineering guide.

Messages and roles

In the chat completions API, you create prompts by providing an array of `messages` that contain instructions for the model. Each message can have a different `role`, which influences how the model might interpret the input.

ROLE	DESCRIPTION	USAGE EXAMPLE
`user`	Instructions that request some output from the model. Similar to messages you'd type in ChatGPT as an end user.	Pass your end-user's message to the model. `Write a haiku about programming.`
`developer`	Instructions to the model that are prioritized ahead of user messages, following chain of command. Previously called the `system` prompt.	Describe how the model should generally behave and respond. `You are a helpful assistant that answers programming questions in the style of a southern belle from the southeast United States.` Now, any response to a `user` message should have a southern belle personality and tone.

`assistant`	A message generated by the model, perhaps in a previous generation request (see the "Conversations" section below).	Provide examples to the model for how it should respond to the current request. For example, to get a model to respond correctly to knock-knock jokes, you might provide a full back-and-forth dialogue of a knock-knock joke.

Message roles may help you get better responses, especially if you want a model to follow hierarchical instructions. They're not deterministic, so the best way to use them is just trying things and seeing what gives you good results.

Here's an example of a developer message that modifies the behavior of the model when generating a response to a `user` message:

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "developer",
      "content": [
        {
          "type": "text",
          "text": `
            You are a helpful assistant that answers programming
            questions in the style of a southern belle from the
            southeast United States.
          `
        }
      ]
    },
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "Are semicolons optional in JavaScript?"
        }
      ]
    }
  ],
  store: true,
});
```

This prompt returns a text output in the rhetorical style requested:

```
Well, sugar, that's a fine question you've got there! Now, in the
world of JavaScript, semicolons are indeed a bit like the pearls
on a necklace - you might slip by without 'em, but you sure do look
more polished with 'em in place.

Technically, JavaScript has this little thing called "automatic
```

```
semicolon insertion" where it kindly adds semicolons for you
where it thinks they oughta go. However, it's not always perfect,
bless its heart. Sometimes, it might get a tad confused and cause
all sorts of unexpected behavior.
```

Giving the model additional data to use for generation

You can also use the message types above to provide additional information to the model, outside of its training data. You might want to include the results of a database query, a text document, or other resources to help the model generate a relevant response. This technique is often referred to as **retrieval augmented generation**, or RAG. Learn more about RAG techniques.

Conversations and context

While each text generation request is independent and stateless (unless you're using assistants), you can still implement **multi-turn conversations** by providing additional messages as parameters to your text generation request. Consider a "knock knock" joke:

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "user",
      "content": [{ "type": "text", "text": "knock knock." }]
    },
    {
      "role": "assistant",
      "content": [{ "type": "text", "text": "Who's there?" }]
    },
    {
      "role": "user",
      "content": [{ "type": "text", "text": "Orange." }]
    }
  ],
  store: true,
});
```

By using alternating `user` and `assistant` messages, you capture the previous state of a conversation in one request to the model.

Managing context for text generation

As your inputs become more complex, or you include more turns in a conversation, you'll need to consider both **output token** and **context window** limits. Model inputs and outputs are metered in **tokens**, which are parsed from inputs to analyze their content and intent and assembled to render logical outputs. Models have limits on token usage during the lifecycle of a text generation request.

Output tokens are the tokens generated by a model in response to a prompt. Each model has different limits for output tokens. For example, `gpt-4o-2024-08-06` can generate a maximum of 16,384 output tokens.

A **context window** describes the total tokens that can be used for both input and output tokens (and for some models, reasoning tokens). Compare the context window limits of our models. For example, `gpt-4o-2024-08-06` has a total context window of 128k tokens.

If you create a very large prompt (usually by including a lot of conversation context or additional data/examples for the model), you run the risk of exceeding the allocated context window for a model, which might result in truncated outputs.

Use the tokenizer tool, built with the tiktoken library, to see how many tokens are in a particular string of text.

Optimizing model outputs

As you iterate on your prompts, you'll continually aim to improve **accuracy**, **cost**, and **latency**. Below, find techniques that optimize for each goal.

	GOAL	AVAILABLE TECHNIQUES
Accuracy	Ensure the model produces accurate and useful responses to your prompts.	Accurate responses require that the model has all the information it needs to generate a response, and knows how to go about creating a response (from interpreting input to formatting and styling). Often, this will require a mix of prompt engineering, RAG, and model fine-tuning. Learn more about optimizing for accuracy.
Cost	Drive down total cost of using models by reducing token usage and using cheaper models when possible.	To control costs, you can try to use fewer tokens or smaller, cheaper models. Learn more about optimizing for cost.
Latency	Decrease the time it takes to generate responses to your prompts.	Optimizing for low latency is a multifaceted process including prompt engineering and parallelism in your own code. Learn more about optimizing for latency.

Next steps

There's much more to explore in text generation. Here are a few resources to go even deeper.

Prompt examples
Get inspired by example prompts for a variety of use cases.

Build a prompt in the Playground

Use the Playground to develop and iterate on prompts.

Browse the Cookbook

The Cookbook has complex examples covering a variety of use cases.

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.