OpenAI Platform

Text generation

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: "system", content: "You are a helpful assistant." },
        {
            role: "user",
            content: "Write a haiku about recursion in programming.",
        },
    ],
});

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

Choosing a model

When making a text generation request, the first option to configure is which model you want to generate the response. The model you choose can greatly influence the output, and impact how

much each generation request costs.

> A **large model** like `gpt-4o` will offer a very high level of intelligence and strong performance, while having a higher cost per token.
>
> A **small model** like `gpt-4o-mini` offers intelligence not quite on the level of the larger model, but is 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 one works best for your prompts! More information on choosing a model can also be found here.

Building prompts

The process of crafting prompts to get the right output from a model is called **prompt engineering**. By giving the model precise instructions, examples, and necessary context information (like private or specialized information that wasn't included in the model's training data), you can improve the quality and accuracy of the model's output. Here, we'll get into some high-level guidance on building prompts, but you might also find the prompt engineering guide helpful.

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.

User messages

User messages contain instructions that request a particular type of output from the model. You can think of `user` messages as the messages you might type in to ChatGPT as an end user.

Here's an example of a user message prompt that asks the `gpt-4o` model to generate a haiku poem based on a prompt.

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "Write a haiku about programming."
        }
      ]
    }


```

```
    ]
});
```

System messages

Messages with the `system` role act as top-level instructions to the model, and typically describe what the model is supposed to do and how it should generally behave and respond.

Here's an example of a system 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": "system",
      "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?"
        }
      ]
    }
  ]
});
```

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.
```

Assistant messages

Messages with the `assistant` role are presumed to have been generated by the model, perhaps in a previous generation request (see the "Conversations" section below). They can also be used to provide examples to the model for how it should respond to the current request - a technique known as **few-shot learning**.

Here's an example of using an assistant message to capture the results of a previous text generation result, and making a new request based on that.

```
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." }]
    }
  ]
});
```

Giving the model additional data to use for generation

The message types above can also be used to provide additional information to the model which may be outside 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 here.

Conversations and context

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

```
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." }]
    }
  ]
});
```

By using alternating `user` and `assistant` messages, you can 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 and more turns in a conversation, you will 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 how many tokens can be used during the lifecycle of a text generation request.

> **Output tokens** are the tokens that are generated by a model in response to a prompt. Each model supports different limits for output tokens, documented here. 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 tokens and output tokens (and for some models, reasoning tokens), documented here. 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.

You can use the tokenizer tool (which uses the tiktoken library) to see how many tokens are present in a string of text.

Optimizing model outputs

As you iterate on your prompts, you will be continually trying to improve **accuracy**, **cost**, and **latency**.

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 about optimizing for accuracy here.
Cost	Drive down the total cost of model usage 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 here.
Latency	Decrease the time it takes to generate responses to your prompts.	Optimzing latency is a multi-faceted process including prompt engineering, parallelism in your own code, and more. Learn more here.

Next steps

There's much more to explore in text generation - here's 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.