Text generation models

OpenAI's text generation models (often called generative pre-trained transformers or large language models) have been trained to understand natural language, code, and images. The models provide text outputs in response to their inputs. The inputs to these models are also referred to as "prompts". Designing a prompt is essentially how you “program” a large language model model, usually by providing instructions or some examples of how to successfully complete a task.

Using OpenAI's text generation models, you can build applications to:

- Draft documents
- Write computer code
- Answer questions about a knowledge base
- Analyze texts
- Give software a natural language interface
- Tutor in a range of subjects
- Translate languages
- Simulate characters for games

With the release of `gpt-4-turbo`, you can now build systems that also process and understand images.

Explore GPT-4 Turbo with image inputs

Check out the vision guide for more detail.

GPT-4 Turbo

Try out GPT-4 Turbo in the playground.

To use one of these models 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-turbo` and `gpt-3.5-turbo`, are accessed through the chat completions API endpoint.

	MODEL FAMILIES	API ENDPOINT
Newer models (2023–)	`gpt-4`, `gpt-4-turbo-preview`, `gpt-3.5-turbo`	https://api.openai.com/v1/chat/completions
Updated legacy models (2023)	`gpt-3.5-turbo-instruct`, `babbage-002`, `davinci-002`	https://api.openai.com/v1/completions

You can experiment with various models in the chat playground. If you're not sure which model to use, then use `gpt-4-turbo` or `gpt-3.5-turbo`.

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 Chat Completions API call looks like the following:

python

```python
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.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 the World Series in 2
    {"role": "user", "content": "Where was it played?"}
  ]
)
```

To learn more, you can view the full API reference documentation for the Chat API.

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 makes sense in the context of the prior messages about the World Series of 2020. Because the models have no memory of past requests, all relevant information must be supplied as part of the conversation history in each request. If a conversation cannot fit within the 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 Globe Life
        "role": "assistant"
      },
      "logprobs": null
    }
  ],
  "created": 1677664795,
  "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",
  "model": "gpt-3.5-turbo-0613",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 17,
    "prompt_tokens": 57,
    "total_tokens": 74
```

```
  }
}
```

The assistant's reply can be extracted with:

```python
completion.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
- `length`: Incomplete model output due to `max_tokens` parameter or token limit
- `function_call`: The model decided to call a function
- `content_filter`: Omitted content due to a flag from our content filters
- `null`: API response still in progress or incomplete

Depending on input parameters, the model response may include different information.

JSON mode New

A common way to use Chat Completions is to instruct the model to always return a JSON object that makes sense for your use case, by specifying this in the system message. While this does work in some cases, occasionally the models may generate output that does not parse to valid JSON objects.

To prevent these errors and improve model performance, when calling `gpt-4-turbo-preview` or `gpt-3.5-turbo-0125`, you can set response_format to `{ "type": "json_object" }` to enable JSON mode. When JSON mode is enabled, the model is constrained to only generate strings that parse into valid JSON object.

Important notes:

- When using JSON mode, **always** instruct the model to produce JSON via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure you don't forget, the API will throw an error if the string `"JSON"` does not appear somewhere in the context.
- The JSON in the message the model returns may be partial (i.e. cut off) if `finish_reason` is `length`, which indicates the generation exceeded `max_tokens` or the conversation exceeded the token limit. To guard against this, check `finish_reason` before parsing the response.

- JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors.

```python
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
  model="gpt-3.5-turbo-0125",
  response_format={ "type": "json_object" },
  messages=[
    {"role": "system", "content": "You are a helpful assistant designed to output JSON.
    {"role": "user", "content": "Who won the world series in 2020?"}
  ]
)
print(response.choices[0].message.content)
```

In this example, the response includes a JSON object that looks something like the following:

```
"content": "{\"winner\": \"Los Angeles Dodgers\"}"`
```

Note that JSON mode is always enabled when the model is generating arguments as part of function calling.

Reproducible outputs Beta

Chat Completions are non-deterministic by default (which means model outputs may differ from request to request). That being said, we offer some control towards deterministic outputs by giving you access to the seed parameter and the system_fingerprint response field.

To receive (mostly) deterministic outputs across API calls, you can:

- Set the seed parameter to any integer of your choice and use the same value across requests you'd like deterministic outputs for.
- Ensure all other parameters (like `prompt` or `temperature`) are the exact same across requests.

Sometimes, determinism may be impacted due to necessary changes OpenAI makes to model configurations on our end. To help you keep track of these changes, we expose the system_fingerprint

field. If this value is different, you may see different outputs due to changes we've made on our systems.

Deterministic outputs

Explore the new seed parameter 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 (4097 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).

To see how many tokens are used by an API call, check the `usage` field in the API response (e.g., `response['usage']['total_tokens']`).

Chat models like `gpt-3.5-turbo` and `gpt-4-turbo-preview` 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 4097 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.

Parameter details

Frequency and presence penalties

The frequency and presence penalties found in the Chat Completions API and Legacy Completions API can be used to reduce the likelihood of sampling repetitive sequences of tokens.

DEEP DIVE

Penalties behind the scenes

Reasonable values for the penalty coefficients are around 0.1 to 1 if the aim is to just reduce repetitive samples somewhat. If the aim is to strongly suppress repetition, then one can increase the coefficients up to 2, but this can noticeably degrade the quality of samples. Negative values can be used to increase the likelihood of repetition.

Token log probabilities

The logprobs parameter found in the Chat Completions API and Legacy Completions API, when requested, provides the log probabilities of each output token, and a limited number of the most likely tokens at each token position alongside their log probabilities. This can be useful in some cases to assess the confidence of the model in its output, or to examine alternative responses the model might have given.

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 legacy Completions API call looks like the following:

```python
from openai import OpenAI
client = OpenAI()

response = client.completions.create(
  model="gpt-3.5-turbo-instruct",
  prompt="Write a tagline for an ice cream shop."
)
```

See the full API reference documentation to learn more.

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 Cream
    }
  ],
  "created": 1683130927,
```

```
  "id": "cmpl-7C9Wxi9Du4j1lQjdjhxBlO22M61LD",
  "model": "gpt-3.5-turbo-instruct",
  "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.

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 to Fre
```

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 is the underlying models that are available in each. The chat completions API is the interface to our most capable model (`gpt-4-turbo-preview`), and our most cost effective model (`gpt-3.5-turbo`).

Which model should I use?

We generally recommend that you use either `gpt-4-turbo-preview` 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-turbo-preview` generally performs better on a wide range of evaluations. In particular, `gpt-4-`

`turbo-preview` is more capable at carefully following complex instructions. By contrast `gpt-3.5-turbo` is more likely to follow just one part of a complex multi-part instruction. `gpt-4-turbo-preview` is less likely than `gpt-3.5-turbo` to make up information, a behavior known as "hallucination". `gpt-4-turbo-preview` also has a larger context window with a maximum size of 128,000 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.

Prompt engineering

An awareness of the best practices for working with OpenAI models can make a significant difference in application performance. The failure modes that each exhibit and the ways of working around or correcting those failure modes are not always intuitive. There is an entire field related to working with language models 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 prompt engineering 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.

FAQ

How should I set the temperature parameter?

Lower values for temperature result in more consistent outputs (e.g. 0.2), while higher values generate more diverse and creative results (e.g. 1.0). Select a temperature value based on the desired trade-off between coherence and creativity for your specific application. The temperature can range is from 0 to 2.

Is fine-tuning available for the latest models?

Yes, for some. Currently, you can only fine-tune `gpt-3.5-turbo` and our updated base models (`babbage-002` and `davinci-002`). 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. We also encourage you to read our safety guide for more information on how to build safer systems.

Should I use ChatGPT or the API?

ChatGPT offers a chat interface for our models and a range of built-in features such as integrated browsing, code execution, plugins, and more. By contrast, using OpenAI's API provides more flexibility but requires that you write code or send the requests to our models programmatically.