GPT models

Looking for ChatGPT? Head to chat.openai.com.

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

Using GPTs, you can build applications to:

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

...and much more!

To use a GPT model 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` and `gpt-3.5-turbo`, are accessed through the chat completions API endpoint. Currently, only the older legacy models are available via the completions API endpoint.

MODEL FAMILIES	API ENDPOINT

Newer models (2023–)	gpt-4, gpt-3.5-turbo	https://api.openai.com/v1/chat/completions
Updated base models (2023)	babbage-002, davinci-002	https://api.openai.com/v1/completions
Legacy models (2020–2022)	text-davinci-003, text-davinci-002, davinci, curie, babbage, ada	https://api.openai.com/v1/completions

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

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 API call looks as follows:

```
import openai

openai.ChatCompletion.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
        {"role": "user", "content": "Where was it played?"}
    ]
)
```

See the full API reference documentation here.

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 Fie
        "role": "assistant"
      }
    }
  ],
  "created": 1677664795,
  "id": "chatcmpl-7QyqpwdfhqwajicIEznoc6Q47XAyW",
  "model": "gpt-3.5-turbo-0613",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 17,
    "prompt_tokens": 57,
    "total_tokens": 74
  }
}
```


In Python, the assistant's reply can be extracted with `response['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 (like providing functions as shown below), the model response may include different information.

Function calling

In an API call, you can describe functions to `gpt-3.5-turbo-0613` and `gpt-4-0613`, and have the model intelligently choose to output a JSON object containing arguments to call those functions. The Chat completions API does not call the function; instead, the model generates JSON that you can use to call the function in your code.

The latest models (`gpt-3.5-turbo-0613` and `gpt-4-0613`) have been fine-tuned to both detect when a function should to be called (depending on the input) and to respond with JSON that adheres to the function signature. With this capability also comes potential risks. We strongly recommend building in user confirmation flows before taking actions that impact the world on behalf of users (sending an email, posting something online, making a purchase, etc).

> Under the hood, functions are injected into the system message in a syntax the model has been trained on. This means functions count against the model's context limit and are billed as input tokens. If running into context limits, we suggest limiting the number of functions or the length of documentation you provide for function parameters.

Function calling allows you to more reliably get structured data back from the model. For example, you can:

- Create chatbots that answer questions by calling external APIs (e.g. like ChatGPT Plugins)

- e.g. define functions like `send_email(to: string, body: string)`, or `get_current_weather(location: string, unit: 'celsius' | 'fahrenheit')`

- Convert natural language into API calls
  - e.g. convert "Who are my top customers?" to `get_customers(min_revenue: int, created_before: string, limit: int)` and call your internal API
- Extract structured data from text
  - e.g. define a function called `extract_data(name: string, birthday: string)`, or `sql_query(query: string)`

...and much more!

The basic sequence of steps for function calling is as follows:

1. Call the model with the user query and a set of functions defined in the functions parameter.
2. The model can choose to call a function; if so, the content will be a stringified JSON object adhering to your custom schema (note: the model may generate invalid JSON or hallucinate parameters).
3. Parse the string into JSON in your code, and call your function with the provided arguments if they exist.
4. Call the model again by appending the function response as a new message, and let the model summarize the results back to the user.

You can see these steps in action through the example below:

```
import openai
import json


# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit="fahrenheit"):
```

```python
    """Get the current weather in a given location"""
    weather_info = {
        "location": location,
        "temperature": "72",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)


def run_conversation():
    # Step 1: send the conversation and available functions to GPT
    messages = [{"role": "user", "content": "What's the weather like in Boston?
    functions = [
        {
            "name": "get_current_weather",
            "description": "Get the current weather in a given location",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco,
                    },
                    "unit": {"type": "string", "enum": ["celsius", "fahrenheit"
                },
                "required": ["location"],
            },
        }
    ]
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-0613",
```

```
        messages=messages,
        functions=functions,
        function_call="auto",  # auto is default, but we'll be explicit
    )
    response_message = response["choices"][0]["message"]

    # Step 2: check if GPT wanted to call a function
    if response_message.get("function_call"):
        # Step 3: call the function
        # Note: the JSON response may not always be valid; be sure to handle er
        available_functions = {
            "get_current_weather": get_current_weather,
        }  # only one function in this example, but you can have multiple
        function_name = response_message["function_call"]["name"]
        fuction_to_call = available_functions[function_name]
        function_args = json.loads(response_message["function_call"]["arguments
        function_response = fuction_to_call(
            location=function_args.get("location"),
            unit=function_args.get("unit"),
        )

        # Step 4: send the info on the function call and function response to G
        messages.append(response_message)  # extend conversation with assistant
        messages.append(
            {
                "role": "function",
                "name": function_name,
                "content": function_response,
            }
        )  # extend conversation with function response
        second_response = openai.ChatCompletion.create(
```

```
            model="gpt-3.5-turbo-0613",
            messages=messages,
        )  # get a new response from GPT where it can see the function response
        return second_response


print(run_conversation())
```

> Hallucinated outputs in function calls can often be mitigated with a system message. For example, if you find that a model is generating function calls with functions that weren't provided to it, try using a system message that says: "Only use the functions you have been provided with."

In the example above, we sent the function response back to the model and let it decide the next step. It responded with a user-facing message which was telling the user the temperature in Boston, but depending on the query, it may choose to call a function again.

For example, if you ask the model "Find the weather in Boston this weekend, book dinner for two on Saturday, and update my calendar" and provide the corresponding functions for these queries, it may choose to call them back to back and only at the end create a user-facing message.

If you want to force the model to call a specific function you can do so by setting `function_call: {"name": "<insert-function-name>"}`. You can also force the model to generate a user-facing message by setting `function_call: "none"`. Note that the default behavior (`function_call: "auto"`) is for the model to decide on its own whether to call a function and if so which function to call.

You can find more examples of function calling in the OpenAI cookbook:

Function calling

Learn from more examples demonstrating function calling

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 API call looks as follows:

```
import openai

response = openai.Completion.create(
  model="text-davinci-003",
  prompt="Write a tagline for an ice cream shop."
)
```

See the full API reference documentation to learn more.

Token log probabilities

The completions API can provide a limited number of log probabilities associated with the most likely tokens for each output token. This feature is controlled by using the logprobs field. This can be useful in some cases to assess the confidence of the model in its output.

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 Shac
    }
  ],
  "created": 1683130927,
  "id": "cmpl-7C9Wxi9Du4j1lQjdjhxBlO22M61LD",
  "model": "text-davinci-003",
  "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 but also includes the optional field `logprobs`.

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 French ex
```

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 derives mainly from the underlying GPT models that are available in each. The chat completions API is the interface to our most capable model (`gpt-4`), and our most cost effective model (`gpt-3.5-turbo`). For reference, `gpt-3.5-turbo` performs at a similar capability level to `text-davinci-003` but at 10% the price per token! See pricing details here.

Which model should I use?

We generally recommend that you use either `gpt-4` 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` generally performs better on a wide range of evaluations. In particular, `gpt-4` 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` is less likely than `gpt-3.5-turbo` to make up information, a behavior known as "hallucination". `gpt-4` also has a larger context window with a maximum size of 8,192 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.

GPT best practices

An awareness of the best practices for working with GPTs can make a significant difference in application performance. The failure modes that GPTs exhibit and the ways of working around or correcting those failure modes are not always intuitive. There is a skill to working with GPTs 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 GPT best practices 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.

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 (4096 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` 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 4096 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. They work by directly modifying the logits (un-normalized log-probabilities) with an additive contribution.

```
mu[j] -> mu[j] - c[j] * alpha_frequency - float(c[j] > 0) * alpha_presence
```

Where:

- `mu[j]` is the logits of the j-th token
- `c[j]` is how often that token was sampled prior to the current position
- `float(c[j] > 0)` is 1 if `c[j] > 0` and 0 otherwise
- `alpha_frequency` is the frequency penalty coefficient
- `alpha_presence` is the presence penalty coefficient

As we can see, the presence penalty is a one-off additive contribution that applies to all tokens that have been sampled at least once and the frequency penalty is a contribution that is proportional to how often a particular token has already been sampled.

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.

FAQ

Why are model outputs inconsistent?

The API is non-deterministic by default. This means that you might get a slightly different completion every time you call it, even if your prompt stays the same. Setting temperature to 0 will make the outputs mostly deterministic, but a small amount of variability will remain.

How should I set the temperature parameter?

Lower values for temperature result in more consistent outputs, while higher values generate more diverse and creative results. Select a temperature value based on the desired trade-off between coherence and creativity for your specific application.

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.

Should I use ChatGPT or the API?

ChatGPT offers a chat interface to the models in the OpenAI API 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.