


Using tools

Use tools like remote MCP servers or web search to extend the model's capabilities.

When generating model responses, you can extend model capabilities using built-in **tools**. These tools help models access additional context and information from the web or your files. The example below uses the **web search tool** to use the latest information from the web to generate a model response.

Include web search results for the model response

javascript 



```
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const response = await client.responses.create({
5   model: "gpt-4.1",
6   tools: [ { type: "web_search_preview" } ],
7   input: "What was a positive news story from today?",
8 });
9
10 console.log(response.output_text);
```

You can include several built-in tools from the available tools list below and let the model decide which tools to use based on the conversation.

Available tools

Here's an overview of the tools available in the OpenAI platform—select one of them for further guidance on usage.



Function calling

Call custom code to give the model access to additional data and capabilities.



Web search

Include data from the Internet in model response generation.



Remote MCP servers

Give the model access to new capabilities via Model Context Protocol (MCP) servers.



File search

Search the contents of uploaded files for context when generating a response.



Image Generation

Generate or edit images using GPT Image.



Code interpreter

Allow the model to execute code in a secure container.



Computer use

Create agentic workflows that enable a model to control a computer interface.

Usage in the API

When making a request to generate a [model response](#), you can enable tool access by specifying configurations in the `tools` parameter. Each tool has its own unique configuration requirements—see the [Available tools](#) section for detailed instructions.

Based on the provided [prompt](#), the model automatically decides whether to use a configured tool. For instance, if your prompt requests information beyond the model's training cutoff date and web search is enabled, the model will typically invoke the web search tool to retrieve relevant, up-to-date information.

You can explicitly control or guide this behavior by setting the `tool_choice` parameter [in the API request](#).

Function calling

In addition to built-in tools, you can define custom functions using the `tools` array. These custom functions allow the model to call your application's code, enabling access to specific data or capabilities not directly available within the model.

Learn more in the [function calling guide](#).

Function calling

Enable models to fetch data and take actions.

Function calling provides a powerful and flexible way for OpenAI models to interface with your code or external services. This guide will explain how to connect the models to your own custom code to fetch data or take action.

[Get weather](#)[Send email](#)[Search knowledge base](#)

Function calling example with get_weather function

javascript ↕



```
1  import { OpenAI } from "openai";
2
3  const openai = new OpenAI();
4
5  const tools = [{
6    "type": "function",
7    "name": "get_weather",
8    "description": "Get current temperature for a given location.",
9    "parameters": {
10     "type": "object",
11     "properties": {
12       "location": {
13         "type": "string",
14         "description": "City and country e.g. Bogotá, Colombia"
15       }
16     },
17     "required": [
18       "location"
19     ],
20     "additionalProperties": false
21   }
22 ];
23
24
25 const response = await openai.responses.create({
26   model: "gpt-4.1",
27   input: [{ role: "user", content: "What is the weather like in Paris today?" }],
28   tools,
29 });
30
```

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

Output



```
1 [{  
2   "type": "function_call",  
3   "id": "fc_12345xyz",  
4   "call_id": "call_12345xyz",  
5   "name": "get_weather",  
6   "arguments": "{\"location\":\"Paris, France\"}"  
7 }]
```

📘 Experiment with function calling and [generate function schemas](#) in the [Playground](#)!

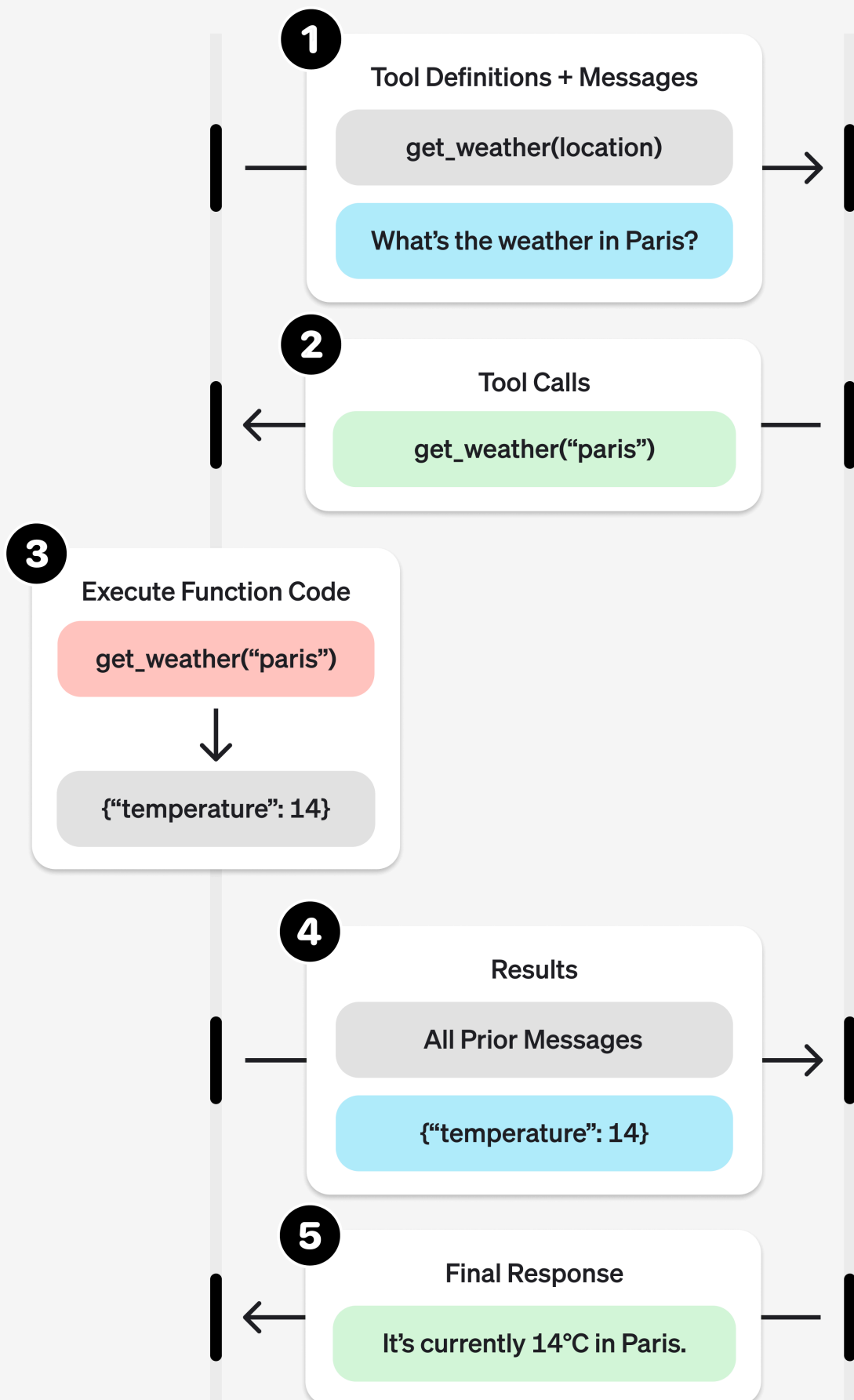
Overview

You can give the model access to your own custom code through **function calling**. Based on the system prompt and messages, the model may decide to call these functions — **instead of (or in addition to) generating text or audio**.

You'll then execute the function code, send back the results, and the model will incorporate them into its final response.

Developer

Model




Function calling has two primary use cases:

Fetching Data	Retrieve up-to-date information to incorporate into the model's response (RAG). Useful for searching knowledge bases and retrieving specific data from APIs (e.g. current weather data).
Taking Action	Perform actions like submitting a form, calling APIs, modifying application state (UI/frontend or backend), or taking agentic workflow actions (like handing off the conversation).

Sample function

Let's look at the steps to allow a model to use a real `get_weather` function defined below:

Sample `get_weather` function implemented in your codebase

javascript ↕ 

```
1 async function getWeather(latitude, longitude) {
2   const response = await fetch(`https://api.open-meteo.com/v1/forecast?latitude=${latitude}&longitude=${longitude}&current_weather=true`);
3   const data = await response.json();
4   return data.current.temperature_2m;
5 }
```

Unlike the diagram earlier, this function expects precise `latitude` and `longitude` instead of a general `location` parameter. (However, our models can automatically determine the coordinates for many locations!)

Function calling steps

- 1 Call model with **functions defined** – along with your system and user messages.

Step 1: Call model with `get_weather` tool defined

javascript ↕ 

```
1 import { OpenAI } from "openai";
2
3 const openai = new OpenAI();
4
5 const tools = [{
6   type: "function",
7   name: "get_weather",
8   description: "Get current temperature for provided coordinates in celsius.",
9   parameters: {
10    type: "object",
11    properties: {
12      latitude: { type: "number" },
13      longitude: { type: "number" }
14    },
15    required: ["latitude", "longitude"],
16    additionalProperties: false
17  }
18 }];
```

```

18     },
19     strict: true
20   }];
21
22   const input = [
23     {
24       role: "user",
25       content: "What's the weather like in Paris today?"
26     }
27   ];
28
29   const response = await openai.responses.create({
30     model: "gpt-4.1",
31     input,
32     tools,
33   });

```

- 2 **Model decides to call function(s)** – model returns the **name** and **input arguments**.

response.output



```

1  [{
2    "type": "function_call",
3    "id": "fc_12345xyz",
4    "call_id": "call_12345xyz",
5    "name": "get_weather",
6    "arguments": "{\"latitude\":48.8566,\"longitude\":2.3522}"
7  }]

```

- 3 **Execute function code** – parse the model's response and **handle function calls**.

Step 3: Execute get_weather function

javascript



```

1  const toolCall = response.output[0];
2  const args = JSON.parse(toolCall.arguments);
3
4  const result = await getWeather(args.latitude, args.longitude);

```

- 4 **Supply model with results** – so it can incorporate them into its final response.

Step 4: Supply result and call model again

javascript



```

1  input.push(toolCall); // append model's function call message
2  input.push({           // append result message
3    type: "function_call_output",
4    call_id: toolCall.call_id,
5  });

```



```

6     output: result.toString()
7   });
8
9   const response2 = await openai.responses.create({
10     model: "gpt-4.1",
11     input,
12     tools,
13     store: true,
14   });
15
console.log(response2.output_text)

```

5 Model responds – incorporating the result in its output.

response_2.output_text



"The current temperature in Paris is 14°C (57.2°F)."

Defining functions

Functions can be set in the `tools` parameter of each API request.

A function is defined by its schema, which informs the model what it does and what input arguments it expects. It comprises the following fields:

FIELD	DESCRIPTION
type	This should always be function
name	The function's name (e.g. get_weather)
description	Details on when and how to use the function
parameters	JSON schema defining the function's input arguments
strict	Whether to enforce strict mode for the function call

Take a look at this example or generate your own below (or in our [Playground](#)).

```

1  {
2    "type": "function",
3    "name": "get_weather",
4    "description": "Retrieves current weather for the given location.",
5    "parameters": {
6      "type": "object",
7      "properties": {
8        "location": {
9

```



```

10     "type": "string",
11     "description": "City and country e.g. Bogotá, Colombia"
12 },
13 "units": {
14     "type": "string",
15     "enum": [
16         "celsius",
17         "fahrenheit"
18     ],
19     "description": "Units the temperature will be returned in."
20 }
21 },
22 "required": [
23     "location",
24     "units"
25 ],
26 "additionalProperties": false
27 },
28 "strict": true
29 }

```

Because the `parameters` are defined by a [JSON schema](#), you can leverage many of its rich features like property types, enums, descriptions, nested objects, and, recursive objects.

Best practices for defining functions

- 1 Write clear and detailed function names, parameter descriptions, and instructions.
 - Explicitly describe the purpose of the function and each parameter (and its format), and what the output represents.
 - Use the system prompt to describe when (and when not) to use each function. Generally, tell the model *exactly* what to do.
 - Include examples and edge cases, especially to rectify any recurring failures. (**Note:** Adding examples may hurt performance for [reasoning models](#).)
- 2 Apply software engineering best practices.
 - Make the functions obvious and intuitive. ([principle of least surprise](#))
 - Use **enums** and object structure to make invalid states unrepresentable. (e.g. `toggle_light(on: bool, off: bool)` allows for invalid calls)
 - **Pass the intern test.** Can an intern/human correctly use the function given nothing but what you gave the model? (If not, what questions do they ask you? Add the answers to the prompt.)
- 3 Offload the burden from the model and use code where possible.
 - Don't make the model fill arguments you already know. For example, if you already have an `order_id` based on a previous menu, don't have an `order_id` param – instead,

have no params `submit_refund()` and pass the `order_id` with code.

- **Combine functions that are always called in sequence.** For example, if you always call `mark_location()` after `query_location()`, just move the marking logic into the query function call.

4 Keep the number of functions small for higher accuracy.

- **Evaluate your performance** with different numbers of functions.
- **Aim for fewer than 20 functions** at any one time, though this is just a soft suggestion.

5 Leverage OpenAI resources.

- **Generate and iterate on function schemas** in the [Playground](#).
- Consider **[fine-tuning](#)** to increase function calling accuracy for large numbers of functions or difficult tasks. ([cookbook](#))

Token Usage

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 you run into token limits, we suggest limiting the number of functions or the length of the descriptions you provide for function parameters.

It is also possible to use [fine-tuning](#) to reduce the number of tokens used if you have many functions defined in your tools specification.

Handling function calls

When the model calls a function, you must execute it and return the result. Since model responses can include zero, one, or multiple calls, it is best practice to assume there are several.

The response `output` array contains an entry with the `type` having a value of `function_call`. Each entry with a `call_id` (used later to submit the function result), `name`, and JSON-encoded `arguments`.

Sample response with multiple function calls




```
1  [
2    {
3      "id": "fc_12345xyz",
4      "call_id": "call_12345xyz",
5      "type": "function_call",
6      "name": "get_weather",
7      "arguments": "{\"location\":\"Paris, France\"}"
8    },
9    {
```

```

10     "id": "fc_67890abc",
11     "call_id": "call_67890abc",
12     "type": "function_call",
13     "name": "get_weather",
14     "arguments": "{\"location\":\"Bogotá, Colombia\"}"
15 },
16 {
17     "id": "fc_99999def",
18     "call_id": "call_99999def",
19     "type": "function_call",
20     "name": "send_email",
21     "arguments": "{\"to\":\"bob@email.com\",\"body\":\"Hi bob\"}"
22 }
23 ]

```

Execute function calls and append results

javascript 

```

1  for (const toolCall of response.output) {
2      if (toolCall.type !== "function_call") {
3          continue;
4      }
5
6      const name = toolCall.name;
7      const args = JSON.parse(toolCall.arguments);
8
9      const result = callFunction(name, args);
10     input.push({
11         type: "function_call_output",
12         call_id: toolCall.call_id,
13         output: result.toString()
14     });
15 }

```

In the example above, we have a hypothetical `call_function` to route each call. Here's a possible implementation:

Execute function calls and append results

javascript 

```

1  const callFunction = async (name, args) => {
2      if (name === "get_weather") {
3          return getWeather(args.latitude, args.longitude);
4      }
5      if (name === "send_email") {
6          return sendEmail(args.to, args.body);
7      }
8  };

```

Formatting results

A result must be a string, but the format is up to you (JSON, error codes, plain text, etc.). The model will interpret that string as needed.

If your function has no return value (e.g. `send_email`), simply return a string to indicate success or failure. (e.g. `"success"`)

Incorporating results into response

After appending the results to your `input`, you can send them back to the model to get a final response.

Send results back to model

javascript ↕ 

```
1 const response = await openai.responses.create({  
2   model: "gpt-4.1",  
3   input,  
4   tools,  
5 });
```

Final response



"It's about 15°C in Paris, 18°C in Bogotá, and I've sent that email to Bob."

Additional configurations

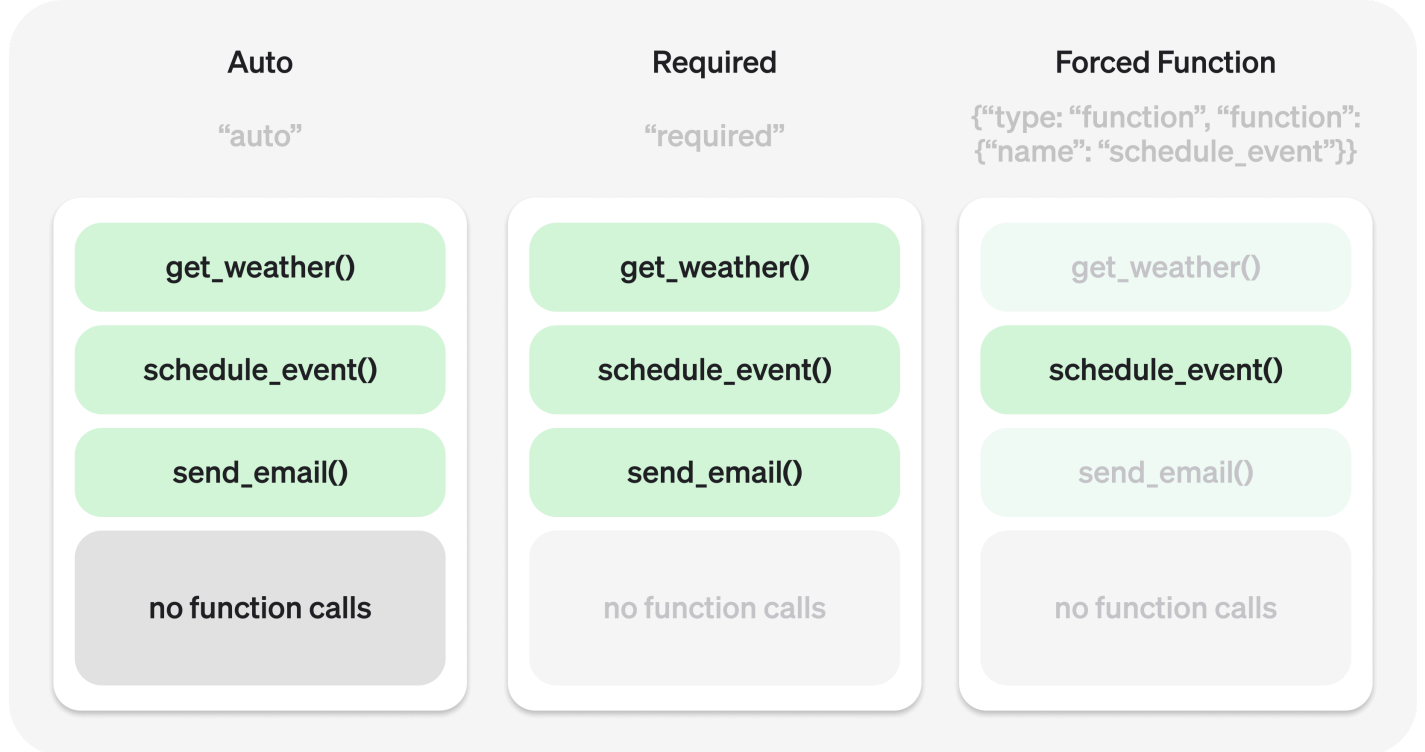
Tool choice

By default the model will determine when and how many tools to use. You can force specific behavior with the `tool_choice` parameter.

1 **Auto: (Default)** Call zero, one, or multiple functions. `tool_choice: "auto"`

2 **Required:** Call one or more functions. `tool_choice: "required"`

1 **Forced Function:** Call exactly one specific function.
`tool_choice: {"type": "function", "name": "get_weather"}`



You can also set `tool_choice` to `"none"` to imitate the behavior of passing no functions.

Parallel function calling

The model may choose to call multiple functions in a single turn. You can prevent this by setting `parallel_tool_calls` to `false`, which ensures exactly zero or one tool is called.

Note: Currently, if you are using a fine tuned model and the model calls multiple functions in one turn then `strict mode` will be disabled for those calls.

Note for `gpt-4.1-nano-2025-04-14`: This snapshot of `gpt-4.1-nano` can sometimes include multiple tools calls for the same tool if parallel tool calls are enabled. It is recommended to disable this feature when using this nano snapshot.

Strict mode

Setting `strict` to `true` will ensure function calls reliably adhere to the function schema, instead of being best effort. We recommend always enabling strict mode.

Under the hood, strict mode works by leveraging our `structured outputs` feature and therefore introduces a couple requirements:

- 1 `additionalProperties` must be set to `false` for each object in the `parameters`.
- 2 All fields in `properties` must be marked as `required`.

You can denote optional fields by adding `null` as a `type` option (see example below).

Strict mode enabled

Strict mode disabled

```
1  {
2    "type": "function",
3    "name": "get_weather",
4    "description": "Retrieves current weather for the given location.",
5    "strict": true,
6    "parameters": {
7      "type": "object",
8      "properties": {
9        "location": {
10         "type": "string",
11         "description": "City and country e.g. Bogotá, Colombia"
12       },
13       "units": {
14         "type": ["string", "null"],
15         "enum": ["celsius", "fahrenheit"],
16         "description": "Units the temperature will be returned in."
17       }
18     },
19     "required": ["location", "units"],
20     "additionalProperties": false
21   }
22 }
```

 All schemas generated in the [playground](#) have strict mode enabled.

While we recommend you enable strict mode, it has a few limitations:

- 1 Some features of JSON schema are not supported. (See [supported schemas](#).)

Specifically for fine tuned models:

- 1 Schemas undergo additional processing on the first request (and are then cached). If your schemas vary from request to request, this may result in higher latencies.
- 2 Schemas are cached for performance, and are not eligible for [zero data retention](#).

Streaming

Streaming can be used to surface progress by showing which function is called as the model fills its arguments, and even displaying the arguments in real time.

Streaming function calls is very similar to streaming regular responses: you set `stream` to `true` and get different `event` objects.

```
1  import { OpenAI } from "openai";
2
3  const openai = new OpenAI();
4
5  const tools = [{
6    type: "function",
7    name: "get_weather",
8    description: "Get current temperature for provided coordinates in celsius.",
9    parameters: {
10     type: "object",
11     properties: {
12       latitude: { type: "number" },
13       longitude: { type: "number" }
14     },
15     required: ["latitude", "longitude"],
16     additionalProperties: false
17   },
18   strict: true
19 }];
20
21
22 const stream = await openai.responses.create({
23   model: "gpt-4.1",
24   input: [{ role: "user", content: "What's the weather like in Paris today?" }],
25   tools,
26   stream: true,
27   store: true,
28 });
29
30 for await (const event of stream) {
31   console.log(event)
32 }
```

```
1  {"type":"response.output_item.added","response_id":"resp_1234xyz","output_index":0,"it
2  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
3  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
4  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
5  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
6  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
7  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
8  {"type":"response.function_call_arguments.delta","response_id":"resp_1234xyz","item_id
9  {"type":"response.function_call_arguments.done","response_id":"resp_1234xyz","item_id"
10 {"type":"response.output_item.done","response_id":"resp_1234xyz","output_index":0,"ite
```


Instead of aggregating chunks into a single `content` string, however, you're aggregating chunks into an encoded `arguments` JSON object.

When the model calls one or more functions an event of type `response.output_item.added` will be emitted for each function call that contains the following fields:

FIELD	DESCRIPTION
<code>response_id</code>	The id of the response that the function call belongs to
<code>output_index</code>	The index of the output item in the response. This represents the individual function calls in the response.
<code>item</code>	The in-progress function call item that includes a name, arguments and id field


Afterwards you will receive a series of events of type

`response.function_call_arguments.delta` which will contain the `delta` of the `arguments` field. These events contain the following fields:

FIELD	DESCRIPTION
<code>response_id</code>	The id of the response that the function call belongs to
<code>item_id</code>	The id of the function call item that the delta belongs to
<code>output_index</code>	The index of the output item in the response. This represents the individual function calls in the response.
<code>delta</code>	The delta of the arguments field.

Below is a code snippet demonstrating how to aggregate the `delta` s into a final `tool_call` object.

Accumulating `tool_call` deltas

javascript 

```
1  const finalToolCalls = {};  
2  
3  for await (const event of stream) {  
4    if (event.type === 'response.output_item.added') {  
5      finalToolCalls[event.output_index] = event.item;  
6    } else if (event.type === 'response.function_call_arguments.delta') {  
7      const index = event.output_index;  
8  
9      if (finalToolCalls[index]) {  
10         finalToolCalls[index].arguments += event.delta;  
11      }  
12    }  
13  }
```

Accumulated final_tool_calls[0]



```
1 {
2   "type": "function_call",
3   "id": "fc_1234xyz",
4   "call_id": "call_2345abc",
5   "name": "get_weather",
6   "arguments": "{\\"location\\":\\"Paris, France\\"}"
7 }
```

When the model has finished calling the functions an event of type

`response.function_call_arguments.done` will be emitted. This event contains the entire function call including the following fields:

FIELD	DESCRIPTION
response_id	The id of the response that the function call belongs to
output_index	The index of the output item in the response. This represents the individual function calls in the response.
item	The function call item that includes a name, arguments and id field.

Web search

Allow models to search the web for the latest information before generating a response.

Using the [Responses API](#), you can enable web search by configuring it in the `tools` array in an API request to generate content. Like any other tool, the model can choose to search the web or not based on the content of the input prompt.

Web search tool example

javascript ↕

```
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const response = await client.responses.create({
5   model: "gpt-4.1",
6   tools: [ { type: "web_search_preview" } ],
7   input: "What was a positive news story from today?",
8 });
9
10 console.log(response.output_text);
```

> Web search tool versions

You can also force the use of the `web_search_preview` tool by using the `tool_choice` parameter, and setting it to `{type: "web_search_preview"}` - this can help ensure lower latency and more consistent results.

Output and citations

Model responses that use the web search tool will include two parts:

- A `web_search_call` output item with the ID of the search call.
- A `message` output item containing:
 - The text result in `message.content[0].text`
 - Annotations `message.content[0].annotations` for the cited URLs

By default, the model's response will include inline citations for URLs found in the web search results. In addition to this, the `url_citation` annotation object will contain the URL, title and

location of the cited source.

❗ When displaying web results or information contained in web results to end users, inline citations must be made clearly visible and clickable in your user interface.

```
1  [
2    {
3      "type": "web_search_call",
4      "id": "ws_67c9fa0502748190b7dd390736892e100be649c1a5ff9609",
5      "status": "completed"
6    },
7    {
8      "id": "msg_67c9fa077e288190af08fdffda2e34f20be649c1a5ff9609",
9      "type": "message",
10     "status": "completed",
11     "role": "assistant",
12     "content": [
13       {
14         "type": "output_text",
15         "text": "On March 6, 2025, several news...",
16         "annotations": [
17           {
18             "type": "url_citation",
19             "start_index": 2606,
20             "end_index": 2758,
21             "url": "https://...",
22             "title": "Title..."
23           }
24         ]
25       }
26     ]
27   }
28 ]
```

User location

To refine search results based on geography, you can specify an approximate user location using country, city, region, and/or timezone.

- The `city` and `region` fields are free text strings, like `Minneapolis` and `Minnesota` respectively.
- The `country` field is a two-letter [ISO country code](#), like `US`.
- The `timezone` field is an [IANA timezone](#) like `America/Chicago`.

```
1 import OpenAI from "openai";
2 const openai = new OpenAI();
3
4 const response = await openai.responses.create({
5   model: "gpt-4.1",
6   tools: [{
7     type: "web_search_preview",
8     user_location: {
9       type: "approximate",
10      country: "GB",
11      city: "London",
12      region: "London"
13    }
14  }],
15   input: "What are the best restaurants around Granary Square?",
16 });
17 console.log(response.output_text);
```

Search context size

When using this tool, the `search_context_size` parameter controls how much context is retrieved from the web to help the tool formulate a response. The tokens used by the search tool do **not** affect the context window of the main model specified in the `model` parameter in your response creation request. These tokens are also **not** carried over from one turn to another — they're simply used to formulate the tool response and then discarded.

Choosing a context size impacts:

- **Cost:** Pricing of our search tool varies based on the value of this parameter. Higher context sizes are more expensive. See tool pricing [here](#).
- **Quality:** Higher search context sizes generally provide richer context, resulting in more accurate, comprehensive answers.
- **Latency:** Higher context sizes require processing more tokens, which can slow down the tool's response time.



Available values:

- `high` : Most comprehensive context, highest cost, slower response.
- `medium` (default): Balanced context, cost, and latency.
- `low` : Least context, lowest cost, fastest response, but potentially lower answer quality.

Again, tokens used by the search tool do **not** impact main model's token usage and are not carried over from turn to turn. Check the [pricing page](#) for details on costs associated with each context

size.

Customizing search context size

javascript  

```
1 import OpenAI from "openai";
2 const openai = new OpenAI();
3
4 const response = await openai.responses.create({
5   model: "gpt-4.1",
6   tools: [{
7     type: "web_search_preview",
8     search_context_size: "low",
9   }],
10   input: "What movie won best picture in 2025?",
11 });
12 console.log(response.output_text);
```

Usage notes

API AVAILABILITY	RATE LIMITS	NOTES
<ul style="list-style-type: none">✔ Responses✔ Chat Completions✖ Assistants	Same as tiered rate limits for underlying model used with the tool.	Pricing ZDR and data residency

Limitations

- Web search is currently not supported in the `gpt-4.1-nano` model.
- The `gpt-4o-search-preview` and `gpt-4o-mini-search-preview` models used in Chat Completions only support a subset of API parameters - view their model data pages for specific information on rate limits and feature support.
- When used as a tool in the [Responses API](#), web search has the same tiered rate limits as the models above.
- Web search is limited to a context window size of 128000 (even with `gpt-4.1` and `gpt-4.1-mini` models).
- [Refer to this guide](#) for data handling, residency, and retention information.




Remote MCP

Allow models to use remote MCP servers to perform tasks.


Model Context Protocol (MCP) is an open protocol that standardizes how applications provide tools and context to LLMs. The MCP tool in the Responses API allows developers to give the model access to tools hosted on **Remote MCP servers**. These are MCP servers maintained by developers and organizations across the internet that expose these tools to MCP clients, like the Responses API.

Calling a remote MCP server with the Responses API is straightforward. For example, here's how you can use the [DeepWiki](#) MCP server to ask questions about nearly any public GitHub repository.

A Responses API request with MCP tools enabled

javascript 

```
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const resp = await client.responses.create({
5   model: "gpt-4.1",
6   tools: [
7     {
8       type: "mcp",
9       server_label: "deepwiki",
10      server_url: "https://mcp.deepwiki.com/mcp",
11      require_approval: "never",
12    },
13  ],
14   input: "What transport protocols are supported in the 2025-03-26 version of the M
15 });
16
17 console.log(resp.output_text);
```

 It is very important that developers trust any remote MCP server they use with the Responses API. A malicious server can exfiltrate sensitive data from anything that enters the model's context. Carefully review the [Risks and Safety](#) section below before using this tool.

The MCP ecosystem

We are still in the early days of the MCP ecosystem. Some popular remote MCP servers today include [Cloudflare](#), [Hubspot](#), [Intercom](#), [Paypal](#), [Pipedream](#), [Plaid](#), [Shopify](#), [Stripe](#), [Square](#), [Twilio](#) and [Zapier](#). We expect many more servers—and registries making it easy to discover these servers—to launch in the coming months. The MCP protocol itself is also early, and we expect to add many more updates to our MCP tool as the protocol evolves.

How it works

The MCP tool works only in the [Responses API](#), and is available across all our new models (gpt-4o, gpt-4.1, and our reasoning models). When you're using the MCP tool, you only pay for [tokens](#) used when importing tool definitions or making tool calls—there are no additional fees involved.

Step 1: Getting the list of tools from the MCP server

The first thing the Responses API does when you attach a remote MCP server to the `tools` array, is attempt to get a list of tools from the server. The Responses API supports remote MCP servers that support either the Streamable HTTP or the HTTP/SSE transport protocol.

If successful in retrieving the list of tools, a new `mcp_list_tools` output item will be visible in the Response object that is created for each MCP server. The `tools` property of this object will show the tools that were successfully imported.

```
1  {
2    "id": "mcpL_682d4379df088191886b70f4ec39f90403937d5f622d7a90",
3    "type": "mcp_list_tools",
4    "server_label": "deepwiki",
5    "tools": [
6      {
7        "name": "read_wiki_structure",
8        "input_schema": {
9          "type": "object",
10         "properties": {
11           "repoName": {
12             "type": "string",
13             "description": "GitHub repository: owner/repo (e.g. \"facebook/react\")"
14           }
15         },
16       },
17       "required": [
18         "repoName"
19       ],
20       "additionalProperties": false,
21       "annotations": null,
22       "description": "",
23       "$schema": "http://json-schema.org/draft-07/schema#"
24     ]
25   }
```

```

25     },
26     // ... other tools
27   ]
28 }

```

As long as the `mcp_list_tools` item is present in the context of the model, we will not attempt to pull a refreshed list of tools from an MCP server. We recommend you keep this item in the model's context as part of every conversation or workflow execution to optimize for latency.

Filtering tools

Some MCP servers can have dozens of tools, and exposing many tools to the model can result in high cost and latency. If you're only interested in a subset of tools an MCP server exposes, you can use the `allowed_tools` parameter to only import those tools.

Constrain allowed tools

javascript ↕ 

```

1  import OpenAI from "openai";
2  const client = new OpenAI();
3
4  const resp = await client.responses.create({
5    model: "gpt-4.1",
6    tools: [{
7      type: "mcp",
8      server_label: "deepwiki",
9      server_url: "https://mcp.deepwiki.com/mcp",
10     require_approval: "never",
11     allowed_tools: ["ask_question"],
12   }],
13   input: "What transport protocols does the 2025-03-26 version of the MCP spec (model
14 });
15
16 console.log(resp.output_text);

```

Step 2: Calling tools

Once the model has access to these tool definitions, it may choose to call them depending on what's in the model's context. When the model decides to call an MCP tool, we make an request to the remote MCP server to call the tool, take it's output and put that into the model's context. This creates an `mcp_call` item which looks like this:

```

1  {
2    "id": "mcp_682d437d90a88191bf88cd03aae0c3e503937d5f622d7a90",
3    "type": "mcp_call",
4    "approval_request_id": null,
5

```

```

6   "arguments": "{\\"repoName\\":\\"modelcontextprotocol/modelcontextprotocol\\",\\"question\\":\\"What is the latest version of the Model Context Protocol (MCP) specification?\\",\\"server_label\\":\\"deepwiki\\"}",
7   "error": null,
8   "name": "ask_question",
9   "output": "The 2025-03-26 version of the Model Context Protocol (MCP) specification",
10  "server_label": "deepwiki"
}

```

As you can see, this includes both the arguments the model decided to use for this tool call, and the `output` that the remote MCP server returned. All models can choose to make multiple (MCP) tool calls in the Responses API, and so, you may see several of these items generated in a single Response API request.

Failed tool calls will populate the error field of this item with MCP protocol errors, MCP tool execution errors, or general connectivity errors. The MCP errors are documented in the MCP spec [here](#).

Approvals

By default, OpenAI will request your approval before any data is shared with a remote MCP server. Approvals help you maintain control and visibility over what data is being sent to an MCP server. We highly recommend that you carefully review (and optionally, log) all data being shared with a remote MCP server. A request for an approval to make an MCP tool call creates a

`mcp_approval_request` item in the Response's output that looks like this:


```

1 {
2   "id": "mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa",
3   "type": "mcp_approval_request",
4   "arguments": "{\\"repoName\\":\\"modelcontextprotocol/modelcontextprotocol\\",\\"question\\":\\"What is the latest version of the Model Context Protocol (MCP) specification?\\",\\"server_label\\":\\"deepwiki\\"}",
5   "name": "ask_question",
6   "server_label": "deepwiki"
7 }

```

You can then respond to this by creating a new Response object and appending an `mcp_approval_response` item to it.

Approving the use of tools in an API request

javascript ↕ 

```

1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const resp = await client.responses.create({
5   model: "gpt-4.1",
6   tools: [{
7     type: "mcp",
8     server_label: "deepwiki",
9     server_url: "https://mcp.deepwiki.com/mcp",

```

```

10     }],
11     previous_response_id: "resp_682d498bdefc81918b4a6aa477bfafd904ad1e533afccbfa",
12     input: [{
13         type: "mcp_approval_response",
14         approve: true,
15         approval_request_id: "mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa"
16     }],
17 });
18
19 console.log(resp.output_text);

```

Here we're using the `previous_response_id` parameter to chain this new Response, with the previous Response that generated the approval request. But you can also pass back the [outputs from one response, as inputs into another](#) for maximum control over what enters the model's context.

If and when you feel comfortable trusting a remote MCP server, you can choose to skip the approvals for reduced latency. To do this, you can set the `require_approval` parameter of the MCP tool to an object listing just the tools you'd like to skip approvals for like shown below, or set it to the value `'never'` to skip approvals for all tools in that remote MCP server.

Never require approval for some tools

javascript 

```

1  import OpenAI from "openai";
2  const client = new OpenAI();
3
4  const resp = await client.responses.create({
5      model: "gpt-4.1",
6      tools: [
7          {
8              type: "mcp",
9              server_label: "deepwiki",
10             server_url: "https://mcp.deepwiki.com/mcp",
11             require_approval: {
12                 never: {
13                     tool_names: ["ask_question", "read_wiki_structure"]
14                 }
15             }
16         }
17     ],
18     input: "What transport protocols does the 2025-03-26 version of the MCP spec (mode
19 });
20
21 console.log(resp.output_text);

```

Authentication

Unlike the DeepWiki MCP server, most other MCP servers require authentication. The MCP tool in the Responses API gives you the ability to flexibly specify headers that should be included in any request made to a remote MCP server. These headers can be used to share API keys, OAuth access tokens, or any other authentication scheme the remote MCP server implements.

The most common header used by remote MCP servers is the `Authorization` header. This is what passing this header looks like:

Use Stripe MCP tool javascript

```
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const resp = await client.responses.create({
5   model: "gpt-4.1",
6   input: "Create a payment link for $20",
7   tools: [
8     {
9       type: "mcp",
10      server_label: "stripe",
11      server_url: "https://mcp.stripe.com",
12      headers: {
13        Authorization: "Bearer $STRIPE_API_KEY"
14      }
15    }
16  ]
17 });
18
19 console.log(resp.output_text);
```

To prevent the leakage of sensitive keys, the Responses API does not store the values of **any** string you provide in the `headers` object. These values will also not be visible in the Response object created. Additionally, because some remote MCP servers generate authenticated URLs, we also discard the *path* portion of the `server_url` in our responses (i.e. `example.com/mcp` becomes `example.com`). Because of this, you must send the full path of the MCP `server_url` and any relevant `headers` in every Responses API creation request you make.

Risks and safety

The MCP tool permits you to connect OpenAI to services that have not been verified by OpenAI and allows OpenAI to access, send and receive data, and take action in these services. All MCP servers are third-party services that are subject to their own terms and conditions.

If you come across a malicious MCP server, please report it to `security@openai.com`.

Connecting to trusted servers

Pick official servers hosted by the service providers themselves (e.g. we recommend connecting to the Stripe server hosted by Stripe themselves on `mcp.stripe.com`, instead of a Stripe MCP server hosted by a third party). Because there aren't too many official remote MCP servers today, you may be tempted to use a MCP server hosted by an organization that doesn't operate that server and simply proxies request to that service via your API. If you must do this, be extra careful in doing your due diligence on these "aggregators", and carefully review how they use your data.

Log and review data being shared with third party MCP servers.

Because MCP servers define their own tool definitions, they may request for data that you may not always be comfortable sharing with the host of that MCP server. Because of this, the MCP tool in the Responses API defaults to requiring approvals of each MCP tool call being made. When developing your application, review the type of data being shared with these MCP servers carefully and robustly. Once you gain confidence in your trust of this MCP server, you can skip these approvals for more performant execution.

We also recommend logging any data sent to MCP servers. If you're using the Responses API with `store=true`, these data are already logged via the API for 30 days unless Zero Data Retention is enabled for your organization. You may also want to log these data in your own systems and perform periodic reviews on this to ensure data is being shared per your expectations.

Malicious MCP servers may include hidden instructions (prompt injections) designed to make OpenAI models behave unexpectedly. While OpenAI has implemented built-in safeguards to help detect and block these threats, it's essential to carefully review inputs and outputs, and ensure connections are established only with trusted servers.

MCP servers may update tool behavior unexpectedly, potentially leading to unintended or malicious behavior.

Implications on Zero Data Retention and Data Residency

The MCP tool is compatible with Zero Data Retention and Data Residency, but it's important to note that MCP servers are third-party services, and data sent to an MCP server is subject to their data retention and data residency policies.

In other words, if you're an organization with Data Residency in Europe, OpenAI will limit inference and storage of Customer Content to take place in Europe up until the point communication or data is sent to the MCP server. It is your responsibility to ensure that the MCP server also adheres to any Zero Data Retention or Data Residency requirements you may have. Learn more about Zero Data Retention and Data Residency [here](#).


Usage notes

API AVAILABILITY	RATE LIMITS	NOTES
<div><div>✔</div><div>Responses</div></div> <div><div>⊗</div><div>Chat Completions</div></div> <div><div>⊗</div><div>Assistants</div></div>	<div><div>Tier 1</div><div>200 RPM</div></div> <div><div>Tier 2 and 3</div><div>1000 RPM</div></div> <div><div>Tier 4 and 5</div><div>2000 RPM</div></div>	<div><div>Pricing</div><div>ZDR and data residency</div></div>

File search

Allow models to search your files for relevant information before generating a response.

File search is a tool available in the [Responses API](#). It enables models to retrieve information in a knowledge base of previously uploaded files through semantic and keyword search. By creating vector stores and uploading files to them, you can augment the models' inherent knowledge by giving them access to these knowledge bases or `vector_stores`.

 To learn more about how vector stores and semantic search work, refer to our [retrieval guide](#).

This is a hosted tool managed by OpenAI, meaning you don't have to implement code on your end to handle its execution. When the model decides to use it, it will automatically call the tool, retrieve information from your files, and return an output.


How to use

Prior to using file search with the Responses API, you need to have set up a knowledge base in a vector store and uploaded files to it.

> Create a vector store and upload a file

Once your knowledge base is set up, you can include the `file_search` tool in the list of tools available to the model, along with the list of vector stores in which to search.

File search tool

javascript 

```
1 import OpenAI from "openai";
2 const openai = new OpenAI();
3
4 const response = await openai.responses.create({
5   model: "gpt-4o-mini",
6   input: "What is deep research by OpenAI?",
7   tools: [{
8     type: "file_search",
9     vector_store_ids: ["<vector_store_id>"],
10  }],
```

```
11 });  
12 console.log(response);
```

When this tool is called by the model, you will receive a response with multiple outputs:

- 1 A `file_search_call` output item, which contains the id of the file search call.
- 2 A `message` output item, which contains the response from the model, along with the file citations.

File search response

json 

```
1  {  
2    "output": [  
3      {  
4        "type": "file_search_call",  
5        "id": "fs_67c09ccea8c48191ade9367e3ba71515",  
6        "status": "completed",  
7        "queries": ["What is deep research?"],  
8        "search_results": null  
9      },  
10     {  
11       "id": "msg_67c09cd3091c819185af2be5d13d87de",  
12       "type": "message",  
13       "role": "assistant",  
14       "content": [  
15         {  
16           "type": "output_text",  
17           "text": "Deep research is a sophisticated capability that allows for extens",  
18           "annotations": [  
19             {  
20               "type": "file_citation",  
21               "index": 992,  
22               "file_id": "file-2dtbBZdjtDKS8eqWxqbgDi",  
23               "filename": "deep_research_blog.pdf"  
24             },  
25             {  
26               "type": "file_citation",  
27               "index": 992,  
28               "file_id": "file-2dtbBZdjtDKS8eqWxqbgDi",  
29               "filename": "deep_research_blog.pdf"  
30             },  
31             {  
32               "type": "file_citation",  
33               "index": 1176,  
34               "file_id": "file-2dtbBZdjtDKS8eqWxqbgDi",  
35               "filename": "deep_research_blog.pdf"  
36             }  
37           ],  
38         }  
39       ]  
40     }  
41   ]  
42 }
```

```

40         "type": "file_citation",
41         "index": 1176,
42         "file_id": "file-2dtbBZdjtDKS8eqWxqbgDi",
43         "filename": "deep_research_blog.pdf"
44     }
45 ]
46 }
47 ]
48 }
49 ]
50 }

```

Retrieval customization

Limiting the number of results

Using the file search tool with the Responses API, you can customize the number of results you want to retrieve from the vector stores. This can help reduce both token usage and latency, but may come at the cost of reduced answer quality.

Limit the number of results

javascript 

```

1  const response = await openai.responses.create({
2      model: "gpt-4o-mini",
3      input: "What is deep research by OpenAI?",
4      tools: [{
5          type: "file_search",
6          vector_store_ids: ["<vector_store_id>"],
7          max_num_results: 2,
8      }],
9  });
10 console.log(response);

```

Include search results in the response

While you can see annotations (references to files) in the output text, the file search call will not return search results by default.

To include search results in the response, you can use the `include` parameter when creating the response.

Include search results

javascript 

```

1  const response = await openai.responses.create({
2      model: "gpt-4o-mini",

```

```

3     input: "What is deep research by OpenAI?",
4     tools: [{
5         type: "file_search",
6         vector_store_ids: ["<vector_store_id>"],
7     }],
8     include: ["file_search_call.results"],
9 });
10 console.log(response);


```

Metadata filtering

You can filter the search results based on the metadata of the files. For more details, refer to our [retrieval guide](#), which covers:

- How to [set attributes on vector store files](#)
- How to [define filters](#)

Metadata filtering

javascript 

```

1  const response = await openai.responses.create({
2      model: "gpt-4o-mini",
3      input: "What is deep research by OpenAI?",
4      tools: [{
5          type: "file_search",
6          vector_store_ids: ["<vector_store_id>"],
7          filters: {
8              type: "eq",
9              key: "type",
10             value: "blog"
11         }
12     }]
13 });
14 console.log(response);

```

Supported files

For `text/` MIME types, the encoding must be one of `utf-8`, `utf-16`, or `ascii`.

FILE FORMAT	MIME TYPE
-------------	-----------

.c	text/x-c
----	----------

.cpp	text/x-c++
------	------------

.cs	text/x-csharp
-----	---------------

.css	text/css
------	----------

FILE FORMAT	MIME TYPE
.doc	application/msword
.docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
.go	text/x-golang
.html	text/html
.java	text/x-java
.js	text/javascript
.json	application/json
.md	text/markdown
.pdf	application/pdf
.php	text/x-php
.pptx	application/vnd.openxmlformats-officedocument.presentationml.presentation
.py	text/x-python
.py	text/x-script.python
.rb	text/x-ruby
.sh	application/x-sh
.tex	text/x-tex
.ts	application/typescript
.txt	text/plain

Usage notes

API AVAILABILITY	RATE LIMITS	NOTES
<div><div>✔ Responses</div><div>⊗ Chat Completions</div><div>✔ Assistants</div></div>	<div>Tier 1</div> <div>100 RPM</div> <div>Tier 2 and 3</div> <div>500 RPM</div> <div>Tier 4 and 5</div> <div>1000 RPM</div>	<div>Pricing</div> <div>ZDR and data residency</div>

 Copy page

Code Interpreter

Allow models to write and run Python to solve problems.


The Code Interpreter tool allows models to write and run Python code in a sandboxed environment to solve complex problems in domains like data analysis, coding, and math. Use it for:

- Processing files with diverse data and formatting
- Generating files with data and images of graphs
- Writing and running code iteratively to solve problems—for example, a model that writes code that fails to run can keep rewriting and running that code until it succeeds

Code Interpreter is available in the [Responses API](#) across all models.


Our latest reasoning models o3 and o4-mini are trained to use Code Interpreter to deeply understand images. They can crop, zoom in, rotate, and perform other image processing techniques to boost their visual intelligence.

Code Interpreter is charged at \$0.03 per container creation. See the [pricing page](#) for information about usage cost.

 While we call this tool Code Interpreter, the model knows it as the `python` tool. Models usually understand prompts that refer to the code interpreter tool. However, the most explicit way to invoke this tool is to ask for "the python tool" in your prompts.

Here's an example of calling the Responses API with a tool call to Code Interpreter:

Use the Responses API with Code Interpreter

javascript  

```
1 import OpenAI from "openai";
2 const client = new OpenAI();
3
4 const resp = await client.responses.create({
5   model: "gpt-4.1",
6   tools: [
7     {
8       type: "code_interpreter",
9       container: { type: "auto" }
10    }
11  ]
12 }
```



```

12     ],
13     instructions: "You are a personal math tutor. When asked a math question, write and
14     input: \"I need to solve the equation 3x + 11 = 14. Can you help me?\",
15 });
16
console.log(resp.output_text);

```

Containers

The Code Interpreter tool requires a [container object](#). A container is a fully sandboxed virtual machine that the model can run Python code in. This container can contain files that you upload, or that it generates.

There are two ways to create containers:

- 1 Auto mode: as seen in the example above, you can do this by passing the `"container": { "type": "auto", files: ["file-1", "file-2"] }` property in the tool configuration while creating a new Response object. This automatically creates a new container, or reuses an active container that was used by a previous `code_interpreter_call` item in the model's context. Look for the `code_interpreter_call` item in the output of this API request to find the `container_id` that was generated or used.
- 2 Explicit mode: here, you explicitly [create a container](#) using the `v1/containers` endpoint, and assign its `id` as the `container` value in the tool configuration in the Response object. For example:

Use explicit container creation

javascript ↕ 

```

1  import OpenAI from "openai";
2  const client = new OpenAI();
3
4  const container = await client.containers.create({ name: "test-container" });
5
6  const resp = await client.responses.create({
7    model: "gpt-4.1",
8    tools: [
9      {
10        type: "code_interpreter",
11        container: container.id
12      }
13    ],
14    tool_choice: "required",
15    input: "use the python tool to calculate what is 4 * 3.82. and then find its square",
16  });
17
18

```

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

Note that containers created with the auto mode are also accessible using the `v1/containers` endpoint.

Expiration

We highly recommend you treat containers as ephemeral and store all data related to the use of this tool on your own systems. Expiration details:

- A container expires if it is not used for 20 minutes. When this happens, using the container in `v1/responses` will fail. You'll still be able to see a snapshot of the container's metadata at its expiry, but all data associated with the container will be discarded from our systems and not recoverable. You should download any files you may need from the container while it is active.
- You can't move a container from an expired state to an active one. Instead, create a new container and upload files again. Note that any state in the old container's memory (like python objects) will be lost.
- Any container operation, like retrieving the container, or adding or deleting files from the container, will automatically refresh the container's `last_active_at` time.

Work with files

When running Code Interpreter, the model can create its own files. For example, if you ask it to construct a plot, or create a CSV, it creates these images directly on your container. When it does so, it cites these files in the `annotations` of its next message. Here's an example:

```
1  {
2    "id": "msg_682d514e268c8191a89c38ea318446200f2610a7ec781a4f",
3    "content": [
4      {
5        "annotations": [
6          {
7            "file_id": "cfile_682d514b2e00819184b9b07e13557f82",
8            "index": null,
9            "type": "container_file_citation",
10           "container_id": "cntr_682d513bb0c48191b10bd4f8b0b3312200e64562acc2e0af",
11           "end_index": 0,
12           "filename": "cfile_682d514b2e00819184b9b07e13557f82.png",
13           "start_index": 0
14         }
15       ],
16       "text": "Here is the histogram of the RGB channels for the uploaded image. Each
17       "type": "output_text",
```

```

19     "logprobs": []
20   }
21 ],
22   "role": "assistant",
23   "status": "completed",
24   "type": "message"
25 }

```

You can download these constructed files by calling the [get container file content](#) method.

Any [files in the model input](#) get automatically uploaded to the container. You do not have to explicitly upload it to the container.

Supported files

FILE FORMAT	MIME TYPE
.c	text/x-c
.cs	text/x-csharp
.cpp	text/x-c++
.csv	text/csv
.doc	application/msword
.docx	application/vnd.openxmlformats-officedocument.wordprocessingml.document
.html	text/html
.java	text/x-java
.json	application/json
.md	text/markdown
.pdf	application/pdf
.php	text/x-php
.pptx	application/vnd.openxmlformats-officedocument.presentationml.presentation
.py	text/x-python
.py	text/x-script.python
.rb	text/x-ruby
.tex	text/x-tex
.txt	text/plain

FILE FORMAT	MIME TYPE
.css	text/css
.js	text/javascript
.sh	application/x-sh
.ts	application/typescript
.csv	application/csv
.jpeg	image/jpeg
.jpg	image/jpeg
.gif	image/gif
.pkl	application/octet-stream
.png	image/png
.tar	application/x-tar
.xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
.xml	application/xml or "text/xml"
.zip	application/zip

Usage notes


API AVAILABILITY	RATE LIMITS	NOTES
<div><div>✔</div><div>Responses</div></div> <div><div>⊗</div><div>Chat Completions</div></div> <div><div>✔</div><div>Assistants</div></div>	100 RPM per org	<div>Pricing</div> <div>ZDR and data residency</div>

 Copy page

Image generation

Allow models to generate or edit images.

The image generation tool allows you to generate images using a text prompt, and optionally image inputs. It leverages the [GPT Image model](#), and automatically optimizes text inputs for improved performance.


 To learn more about image generation, refer to our dedicated [image generation guide](#).

Usage

When you include the `image_generation` tool in your request, the model can decide when and how to generate images as part of the conversation, using your prompt and any provided image inputs.


The `image_generation_call` tool call result will include a base64-encoded image.

Generate an image

javascript 

```
1  import OpenAI from "openai";
2  const openai = new OpenAI();
3
4  const response = await openai.responses.create({
5    model: "gpt-4.1-mini",
6    input: "Generate an image of gray tabby cat hugging an otter with an orange scarf",
7    tools: [{type: "image_generation"}],
8  });
9
10 // Save the image to a file
11 const imageData = response.output
12   .filter((output) => output.type === "image_generation_call")
13   .map((output) => output.result);
14
15 if (imageData.length > 0) {
16   const imageBase64 = imageData[0];
17   const fs = await import("fs");
18   fs.writeFileSync("otter.png", Buffer.from(imageBase64, "base64"));
19 }
```

You can [provide input images](#) using file IDs or base64 data.

 To force the image generation tool call, you can set the parameter `tool_choice` to `{"type": "image_generation"}`.

Tool options

You can configure the following output options as parameters for the [image generation tool](#):

- Size: Image dimensions (e.g., 1024x1024, 1024x1536)
- Quality: Rendering quality (e.g. low, medium, high)
- Format: File output format
- Compression: Compression level (0-100%) for JPEG and WebP formats
- Background: Transparent or opaque

`size`, `quality`, and `background` support the `auto` option, where the model will automatically select the best option based on the prompt.

For more details on available options, refer to the [image generation guide](#).

Revised prompt

When using the image generation tool, the mainline model (e.g. `gpt-4.1`) will automatically revise your prompt for improved performance.

You can access the revised prompt in the `revised_prompt` field of the image generation call:

```
1 {  
2   "id": "ig_123",  
3   "type": "image_generation_call",  
4   "status": "completed",  
5   "revised_prompt": "A gray tabby cat hugging an otter. The otter is wearing an orange",  
6   "result": "..."  
7 }
```

Prompting tips

Image generation works best when you use terms like "draw" or "edit" in your prompt.

For example, if you want to combine images, instead of saying "combine" or "merge", you can say something like "edit the first image by adding this element from the second image".



Multi-turn editing

You can iteratively edit images by referencing previous response or image IDs. This allows you to refine images across multiple turns in a conversation.

Using previous response ID

Using image ID

Multi-turn image generation

javascript  

```
1  import OpenAI from "openai";
2  const openai = new OpenAI();
3
4  const response = await openai.responses.create({
5    model: "gpt-4.1-mini",
6    input:
7      "Generate an image of gray tabby cat hugging an otter with an orange scarf",
8    tools: [{ type: "image_generation" }],
9  });
10
11  const imageData = response.output
12    .filter((output) => output.type === "image_generation_call")
13    .map((output) => output.result);
14
15  if (imageData.length > 0) {
16    const imageBase64 = imageData[0];
17    const fs = await import("fs");
18    fs.writeFileSync("cat_and_otter.png", Buffer.from(imageBase64, "base64"));
19  }
20
21
22  // Follow up
23
24  const response_fwup = await openai.responses.create({
25    model: "gpt-4.1-mini",
26    previous_response_id: response.id,
27    input: "Now make it look realistic",
28    tools: [{ type: "image_generation" }],
29  });
30
31  const imageData_fwup = response_fwup.output
32    .filter((output) => output.type === "image_generation_call")
33    .map((output) => output.result);
34
35  if (imageData_fwup.length > 0) {
36    const imageBase64 = imageData_fwup[0];
37    const fs = await import("fs");
38    fs.writeFileSync(
39      "cat_and_otter_realistic.png",
40      Buffer.from(imageBase64, "base64")
41    );
42  }
```




```
);  
}
```

Streaming

The image generation tool supports streaming partial images as the final result is being generated. This provides faster visual feedback for users and improves perceived latency.

You can set the number of partial images (1-3) with the `partial_images` parameter.

Stream an image

javascript 

```
1  import OpenAI from "openai";  
2  import fs from "fs";  
3  const openai = new OpenAI();  
4  
5  const stream = await openai.responses.create({  
6    model: "gpt-4.1",  
7    input:  
8      "Draw a gorgeous image of a river made of white owl feathers, snaking its way thro  
9    stream: true,  
10    tools: [{ type: "image_generation", partial_images: 2 }],  
11  });  
12  
13  for await (const event of stream) {  
14    if (event.type === "response.image_generation_call.partial_image") {  
15      const idx = event.partial_image_index;  
16      const imageBase64 = event.partial_image_b64;  
17      const imageBuffer = Buffer.from(imageBase64, "base64");  
18      fs.writeFileSync(`river${idx}.png`, imageBuffer);  
19    }  
20  }
```

Supported models

The image generation tool is supported for the following models:

- `gpt-4o`
- `gpt-4o-mini`
- `gpt-4.1`
- `gpt-4.1-mini`
- `gpt-4.1-nano`
- `o3`

The model used for the image generation process is always `gpt-image-1` , but these models can be used as the mainline model in the Responses API as they can reliably call the image generation tool when needed.