Using tools

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

When generating model responses, you can extend capabilities using built-in tools and remote MCP servers. These enable the model to search the web, retrieve from your files, call your own functions, or access third-party services.

Web search

Include web search results for the model response

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

const response = await client.responses.create({
    model: "gpt-5",
    tools: [
        { type: "web_search_preview" },
    ],
    input: "What was a positive news story from today?",
});

console.log(response.output_text);
 from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-5",
    tools=[{"type": "web_search_preview"}],
    input="What was a positive news story from today?"
)
```

```
print(response.output_text)
```

```
 curl "https://api.openai.com/v1/responses" \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $OPENAI_API_KEY" \
    -d '{
        "model": "gpt-5",
        "tools": [{"type": "web_search_preview"}],
        "input": "what was a positive news story from today?"
    }'
```

File search

Search your files in a response

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

response = client.responses.create(
    model="gpt-4.1",
    input="What is deep research by OpenAI?",
    tools=[{
        "type": "file_search",
        "vector_store_ids": ["<vector_store_id>"]
    }]
)
print(response)
```

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

const response = await openai.responses.create({
    model: "gpt-4.1",
    input: "What is deep research by OpenAI?",
    tools: [
        {
```

```
        type: "file_search",
        vector_store_ids: ["<vector_store_id>"],
      },
    ],
});
console.log(response);
```

Function calling

Call your own function

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

const tools = [
    {
        type: "function",
        name: "get_weather",
        description: "Get current temperature for a given location.",
        parameters: {
            type: "object",
            properties: {
                location: {
                    type: "string",
                    description: "City and country e.g. Bogotá, Colombia",
                },
            },
            required: ["location"],
            additionalProperties: false,
        },
        strict: true,
    },
];

const response = await client.responses.create({
```

```
    model: "gpt-5",
    input: [
        { role: "user", content: "What is the weather like in Paris today?" },
    ],
    tools,
});

console.log(response.output[0].to_json());
 from openai import OpenAI

client = OpenAI()

tools = [
    {
        "type": "function",
        "name": "get_weather",
        "description": "Get current temperature for a given location.",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "City and country e.g. Bogotá, Colombia",
                }
            },
            "required": ["location"],
            "additionalProperties": False,
        },
        "strict": True,
    },
]

response = client.responses.create(
    model="gpt-5",
```

```
    input=[
        {"role": "user", "content": "What is the weather like in Paris today?"},
    ],
    tools=tools,
)

print(response.output[0].to_json())
 curl -X POST https://api.openai.com/v1/responses \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-5",
    "input": [
      {"role": "user", "content": "What is the weather like in Paris today?"}
    ],
    "tools": [
      {
        "type": "function",
        "name": "get_weather",
        "description": "Get current temperature for a given location.",
        "parameters": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City and country e.g. Bogotá, Colombia"
            }
          },
          "required": ["location"],
          "additionalProperties": false
        },
        "strict": true
      }
    ]
```

```
  }'
```

Remote MCP

Call a remote MCP server

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "deepwiki",
      "server_url": "https://mcp.deepwiki.com/mcp",
      "require_approval": "never"
    }
  ],
  "input": "What transport protocols are supported in the 2025-03-26 version of the MCP spec?"
}'
 import OpenAI from "openai";
const client = new OpenAI();

const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [
        {
            type: "mcp",
            server_label: "deepwiki",
            server_url: "https://mcp.deepwiki.com/mcp",
            require_approval: "never",
        },
    ],
```

```
    input: "What transport protocols are supported in the 2025-03-26 version of the MCP spec?",
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    tools=[
        {
            "type": "mcp",
            "server_label": "deepwiki",
            "server_url": "https://mcp.deepwiki.com/mcp",
            "require_approval": "never",
        },
    ],
    input="What transport protocols are supported in the 2025-03-26 version of the MCP spec?",
)

print(resp.output_text)
```

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.

](/docs/guides/function-calling)[

Web search

Include data from the Internet in model response generation.

](/docs/guides/tools-web-search)[

Remote MCP servers

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

](/docs/guides/tools-remote-mcp)[

File search

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

](/docs/guides/tools-file-search)[

Image generation

Generate or edit images using GPT Image.

](/docs/guides/tools-image-generation)[

Code interpreter

Allow the model to execute code in a secure container.

](/docs/guides/tools-code-interpreter)[

Computer use

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

](/docs/guides/tools-computer-use)

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.

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

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

const response = await client.responses.create({
    model: "gpt-5",
    tools: [
        { type: "web_search_preview" },
    ],
    input: "What was a positive news story from today?",
});

console.log(response.output_text);
 from openai import OpenAI
client = OpenAI()
```

```
response = client.responses.create(
    model="gpt-5",
    tools=[{"type": "web_search_preview"}],
    input="What was a positive news story from today?"
)

print(response.output_text)
```

```
 curl "https://api.openai.com/v1/responses" \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $OPENAI_API_KEY" \
    -d '{
        "model": "gpt-5",
        "tools": [{"type": "web_search_preview"}],
        "input": "what was a positive news story from today?"
    }'
```

Web search tool versions

The current default version of the web search tool is:

web_search_preview

Which points to a dated version:

web_search_preview_2025_03_11

As the tool evolves, future dated snapshot versions will be documented in the API reference.

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, along with the action taken in web_search_call.action. The action is one of:

    - search, which represents a web search. It will usually (but not always) includes the search query and domains which were searched. Search actions incur a tool call cost (see pricing).
    - open_page, which represents a page being opened. Only emitted by Deep Research models.
    - find_in_page, which represents searching within a page. Only emitted by Deep Research models.
- 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.


```
[
  {
    "type": "web_search_call",
    "id": "ws_67c9fa0502748190b7dd390736892e100be649c1a5ff9609",
    "status": "completed"
  },
  {
    "id": "msg_67c9fa077e288190af08fdffda2e34f20be649c1a5ff9609",
    "type": "message",
    "status": "completed",
    "role": "assistant",
    "content": [
      {
        "type": "output_text",
        "text": "On March 6, 2025, several news...",
```

```
            "annotations": [
              {
                "type": "url_citation",
                "start_index": 2606,
                "end_index": 2758,
                "url": "https://...",
                "title": "Title..."
              }
            ]
          }
        ]
    }
]
```

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.

Note that user location is not supported for deep research models using web search.

Customizing user location

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

response = client.responses.create(
  model="o4-mini",
  tools=[{
    "type": "web_search_preview",
```

```
        "user_location": {
            "type": "approximate",
            "country": "GB",
            "city": "London",
            "region": "London",
        }
    }],
    input="What are the best restaurants around Granary Square?",
)

print(response.output_text)
```
```
 import OpenAI from "openai";
const openai = new OpenAI();

const response = await openai.responses.create({
    model: "o4-mini",
    tools: [{
        type: "web_search_preview",
        user_location: {
            type: "approximate",
            country: "GB",
            city: "London",
            region: "London"
        }
    }],
    input: "What are the best restaurants around Granary Square?",
});
console.log(response.output_text);
```
```
 curl "https://api.openai.com/v1/responses" \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $OPENAI_API_KEY" \
    -d '{
        "model": "o4-mini",
        "tools": [{
```

```
        "type": "web_search_preview",
        "user_location": {
            "type": "approximate",
            "country": "GB",
            "city": "London",
            "region": "London"
        }
    }],
    "input": "What are the best restaurants around Granary Square?"
}'
```

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**: Search content tokens are free for some models, but may be billed at a model's text token rates for others. Refer to pricing for details.
- **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, slower response.
- **medium** (default): Balanced context and latency.
- **low**: Least context, fastest response, but potentially lower answer quality.

Context size configuration is not supported for o3, o3-pro, o4-mini, and deep research models.

Customizing search context size

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

response = client.responses.create(
    model="gpt-4.1",
    tools=[{
        "type": "web_search_preview",
        "search_context_size": "low",
    }],
    input="What movie won best picture in 2025?",
)

print(response.output_text)
 import OpenAI from "openai";
const openai = new OpenAI();

const response = await openai.responses.create({
    model: "gpt-4.1",
    tools: [{
        type: "web_search_preview",
        search_context_size: "low",
    }],
    input: "What movie won best picture in 2025?",
});
console.log(response.output_text);
 curl "https://api.openai.com/v1/responses" \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $OPENAI_API_KEY" \
    -d '{
        "model": "gpt-4.1",
```

```
    "tools": [{
        "type": "web_search_preview",
        "search_context_size": "low"
    }],
    "input": "What movie won best picture in 2025?"
}'
```

Usage notes

|| |ResponsesChat CompletionsAssistants|Same as tiered rate limits for underlying model used with the tool.|PricingZDR 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.

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
- Boosting visual intelligence in our latest reasoning models (like o3 and o4-mini). The model can use this tool to crop, zoom, rotate, and otherwise process and transform images.

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

Use the Responses API with Code Interpreter

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "model": "gpt-4.1",
    "tools": [{
      "type": "code_interpreter",
      "container": { "type": "auto" }
    }],
    "instructions": "You are a personal math tutor. When asked a math question,
write and run code using the python tool to answer the question.",
    "input": "I need to solve the equation 3x + 11 = 14. Can you help me?"
  }'
 import OpenAI from "openai";
const client = new OpenAI();

const instructions = `
You are a personal math tutor. When asked a math question,
write and run code using the python tool to answer the question.
`;

const resp = await client.responses.create({
```

```
  model: "gpt-4.1",
  tools: [
    {
      type: "code_interpreter",
      container: { type: "auto" },
    },
  ],
  instructions,
  input: "I need to solve the equation 3x + 11 = 14. Can you help me?",
});

console.log(JSON.stringify(resp.output, null, 2));
```

```
 from openai import OpenAI

client = OpenAI()

instructions = """
You are a personal math tutor. When asked a math question,
write and run code using the python tool to answer the question.
"""

resp = client.responses.create(
    model="gpt-4.1",
    tools=[
        {
            "type": "code_interpreter",
            "container": {"type": "auto"}
        }
    ],
    instructions=instructions,
    input="I need to solve the equation 3x + 11 = 14. Can you help me?",
)
```

```
print(resp.output)
```

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.

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", "file_ids": ["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

```
curl https://api.openai.com/v1/containers \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
        "name": "My Container"
      }'

# Use the returned container id in the next call:
curl https://api.openai.com/v1/responses \
```

```
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-4.1",
    "tools": [{
      "type": "code_interpreter",
      "container": "cntr_abc123"
    }],
    "tool_choice": "required",
    "input": "use the python tool to calculate what is 4 * 3.82. and then find its
square root and then find the square root of that result"
  }'
 from openai import OpenAI
client = OpenAI()

container = client.containers.create(name="test-container")

response = client.responses.create(
    model="gpt-4.1",
    tools=[{
        "type": "code_interpreter",
        "container": container.id
    }],
    tool_choice="required",
    input="use the python tool to calculate what is 4 * 3.82. and then find its
square root and then find the square root of that result"
)

print(response.output_text)
 import OpenAI from "openai";
const client = new OpenAI();

const container = await client.containers.create({ name: "test-container" });
```

```
const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [
      {
        type: "code_interpreter",
        container: container.id
      }
    ],
    tool_choice: "required",
    input: "use the python tool to calculate what is 4 * 3.82. and then find its
square root and then find the square root of that result"
});

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:


```
{
  "id": "msg_682d514e268c8191a89c38ea318446200f2610a7ec781a4f",
  "content": [
    {
      "annotations": [
        {
          "file_id": "cfile_682d514b2e00819184b9b07e13557f82",
          "index": null,
          "type": "container_file_citation",
          "container_id":
"cntr_682d513bb0c48191b10bd4f8b0b3312200e64562acc2e0af",
          "end_index": 0,
          "filename": "cfile_682d514b2e00819184b9b07e13557f82.png",
          "start_index": 0
        }
      ],
      "text": "Here is the histogram of the RGB channels for the uploaded
image. Each curve represents the distribution of pixel intensities for the red,
green, and blue channels. Peaks toward the high end of the intensity scale
(right-hand side) suggest a lot of brightness and strong warm tones, matching
the orange and light background in the image. If you want a different style of
histogram (e.g., overall intensity, or quantized color groups), let me know!",
      "type": "output_text",
      "logprobs": []
    }
  ],
  "role": "assistant",
  "status": "completed",
```

```
    "type": "message"
}
```

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.

Uploading and downloading files

Add new files to your container using Create container file. This endpoint accepts either a multipart upload or a JSON body with a file_id. List existing container files with List container files and download bytes from Retrieve container file content.

Dealing with citations

Files and images generated by the model are returned as annotations on the assistant's message. container_file_citation annotations point to files created in the container. They include the container_id, file_id, and filename. You can parse these annotations to surface download links or otherwise process the files.

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
.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

|| |ResponsesChat CompletionsAssistants|100 RPM per org|PricingZDR and data residency|

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

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "deepwiki",
      "server_url": "https://mcp.deepwiki.com/mcp",
      "require_approval": "never"
    }
  ],
  "input": "What transport protocols are supported in the 2025-03-26 version of the MCP spec?"
}'
 import OpenAI from "openai";
const client = new OpenAI();

const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [
        {
            type: "mcp",
            server_label: "deepwiki",
            server_url: "https://mcp.deepwiki.com/mcp",
```

```
        require_approval: "never",
      },
    ],
    input: "What transport protocols are supported in the 2025-03-26 version of the MCP spec?",
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    tools=[
        {
            "type": "mcp",
            "server_label": "deepwiki",
            "server_url": "https://mcp.deepwiki.com/mcp",
            "require_approval": "never",
        },
    ],
    input="What transport protocols are supported in the 2025-03-26 version of the MCP spec?",
)

print(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.

```
{
  "id": "mcpl_682d4379df088191886b70f4ec39f90403937d5f622d7a90",
  "type": "mcp_list_tools",
  "server_label": "deepwiki",
  "tools": [
    {
      "name": "read_wiki_structure",
      "input_schema": {
        "type": "object",
        "properties": {
          "repoName": {
            "type": "string",
            "description": "GitHub repository: owner/repo (e.g. \"facebook/react\")"
```

```
        }
      },
      "required": [
        "repoName"
      ],
      "additionalProperties": false,
      "annotations": null,
      "description": "",
      "$schema": "http://json-schema.org/draft-07/schema#"
    }
  },
  // ... other tools
 ]
}
```

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

```
curl https://api.openai.com/v1/responses \
 -H "Content-Type: application/json" \
 -H "Authorization: Bearer $OPENAI_API_KEY" \
 -d '{
 "model": "gpt-4.1",
 "tools": [
  {
```

```
      "type": "mcp",
      "server_label": "deepwiki",
      "server_url": "https://mcp.deepwiki.com/mcp",
      "require_approval": "never",
      "allowed_tools": ["ask_question"]
    }
  ],
  "input": "What transport protocols does the 2025-03-26 version of the MCP
spec (modelcontextprotocol/modelcontextprotocol) support?"
}'
 import OpenAI from "openai";
const client = new OpenAI();

const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [{
        type: "mcp",
        server_label: "deepwiki",
        server_url: "https://mcp.deepwiki.com/mcp",
        require_approval: "never",
        allowed_tools: ["ask_question"],
    }],
    input: "What transport protocols does the 2025-03-26 version of the MCP
spec (modelcontextprotocol/modelcontextprotocol) support?",
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    tools=[{
```

```
        "type": "mcp",
        "server_label": "deepwiki",
        "server_url": "https://mcp.deepwiki.com/mcp",
        "require_approval": "never",
        "allowed_tools": ["ask_question"],
    }],
    input="What transport protocols does the 2025-03-26 version of the MCP
spec (modelcontextprotocol/modelcontextprotocol) support?",
)

print(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:


```
{
  "id": "mcp_682d437d90a88191bf88cd03aae0c3e503937d5f622d7a90",
  "type": "mcp_call",
  "approval_request_id": null,
  "arguments":
"{\"repoName\":\"modelcontextprotocol/modelcontextprotocol\",\"question\":\"
What transport protocols does the 2025-03-26 version of the MCP spec
support?\"}",
  "error": null,
  "name": "ask_question",
  "output": "The 2025-03-26 version of the Model Context Protocol (MCP)
specification supports two standard transport mechanisms: `stdio` and
`Streamable HTTP` ...",
  "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:

{
  "id": "mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa",
  "type": "mcp_approval_request",
  "arguments":
"{\"repoName\":\"modelcontextprotocol/modelcontextprotocol\",\"question\":\"
What transport protocols are supported in the 2025-03-26 version of the MCP
spec?\"}",
  "name": "ask_question",
  "server_label": "deepwiki"
}

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

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "deepwiki",
      "server_url": "https://mcp.deepwiki.com/mcp"
    }
  ],
  "previous_response_id":
"resp_682d498bdefc81918b4a6aa477bfafd904ad1e533afccbfa",
  "input": [{
    "type": "mcp_approval_response",
    "approve": true,
    "approval_request_id":
"mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa"
  }]
}'
 import OpenAI from "openai";
const client = new OpenAI();

const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [{
        type: "mcp",
        server_label: "deepwiki",
        server_url: "https://mcp.deepwiki.com/mcp",
    }],
    previous_response_id:
"resp_682d498bdefc81918b4a6aa477bfafd904ad1e533afccbfa",
    input: [{
```

```
        type: "mcp_approval_response",
        approve: true,
        approval_request_id:
"mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa"
    }],
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    tools=[{
        "type": "mcp",
        "server_label": "deepwiki",
        "server_url": "https://mcp.deepwiki.com/mcp",
    }],

previous_response_id="resp_682d498bdefc81918b4a6aa477bfafd904ad1e533
afccbfa",
    input=[{
        "type": "mcp_approval_response",
        "approve": True,
        "approval_request_id":
"mcpr_682d498e3bd4819196a0ce1664f8e77b04ad1e533afccbfa"
    }],
)

print(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 enter's 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

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "deepwiki",
      "server_url": "https://mcp.deepwiki.com/mcp",
      "require_approval": {
          "never": {
            "tool_names": ["ask_question", "read_wiki_structure"]
          }
      }
    }
  ],
  "input": "What transport protocols does the 2025-03-26 version of the MCP spec (modelcontextprotocol/modelcontextprotocol) support?"
}'
 import OpenAI from "openai";
const client = new OpenAI();
```

```
const resp = await client.responses.create({
    model: "gpt-4.1",
    tools: [
        {
            type: "mcp",
            server_label: "deepwiki",
            server_url: "https://mcp.deepwiki.com/mcp",
            require_approval: {
                never: {
                    tool_names: ["ask_question", "read_wiki_structure"]
                }
            }
        },
    ],
    input: "What transport protocols does the 2025-03-26 version of the MCP
spec (modelcontextprotocol/modelcontextprotocol) support?",
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    tools=[
        {
            "type": "mcp",
            "server_label": "deepwiki",
            "server_url": "https://mcp.deepwiki.com/mcp",
            "require_approval": {
                "never": {
                    "tool_names": ["ask_question", "read_wiki_structure"]
                }
```

```
        }
      },
    ],
    input="What transport protocols does the 2025-03-26 version of the MCP spec (modelcontextprotocol/modelcontextprotocol) support?",
)

print(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

```
curl https://api.openai.com/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
  "model": "gpt-4.1",
  "input": "Create a payment link for $20",
  "tools": [
    {
      "type": "mcp",
      "server_label": "stripe",
      "server_url": "https://mcp.stripe.com",
      "headers": {
        "Authorization": "Bearer $STRIPE_API_KEY"
```

```
    }
   }
 ]
}'
 import OpenAI from "openai";
const client = new OpenAI();

const resp = await client.responses.create({
    model: "gpt-4.1",
    input: "Create a payment link for $20",
    tools: [
        {
            type: "mcp",
            server_label: "stripe",
            server_url: "https://mcp.stripe.com",
            headers: {
                Authorization: "Bearer $STRIPE_API_KEY"
            }
        }
    ]
});

console.log(resp.output_text);
 from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    input="Create a payment link for $20",
    tools=[
        {
            "type": "mcp",
            "server_label": "stripe",
```

```
            "server_url": "https://mcp.stripe.com",
            "headers": {
               "Authorization": "Bearer $STRIPE_API_KEY"
            }
         }
      ]
)

print(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

|| |ResponsesChat CompletionsAssistants|Tier 1200 RPMTier 2 and 31000 RPMTier 4 and 52000 RPM|PricingZDR and data residency|