Built-in tools

==============

Use built-in tools like web search and file 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](/docs/guides/tools-web-search) to use the latest information from the web to generate a model response.

Include web search results for the completion

```javascript

import OpenAI from "openai";

const client = new OpenAI();

const completion = await client.chat.completions.create({

model: "gpt-4o-search-preview",

web\_search\_options: {},

messages: [{

"role": "user",

"content": "What was a positive news story from today?"

}],

});

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

```

```python

from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(

model="gpt-4o-search-preview",

web\_search\_options={},

messages=[

{

"role": "user",

"content": "What was a positive news story from today?",

}

],

)

print(completion.choices[0].message.content)

```

```bash

curl -X POST "https://api.openai.com/v1/chat/completions" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-H "Content-type: application/json" \

-d '{

"model": "gpt-4o-search-preview",

"web\_search\_options": {},

"messages": [{

"role": "user",

"content": "What was a positive news story from today?"

}]

}'

```

Available tools

---------------

Here's an overview of the tools available in the OpenAI platform—select one of them for further guidance on usage. Note that currently the file search and computer use tools are only available using the new [Responses API](/docs/api-reference/responses).

[

Web search

Include data from the Internet in model response generation.

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

Function calling

Enable the model to call custom code that you define, giving it access to additional data and capabilities.

](/docs/guides/function-calling)

Usage in the API

----------------

When making a request to generate a [chat completion](/docs/api-reference/chat/create), you can enable web search by using the `web\_search` parameter. This allows the model to access current Internet information during response generation. For additional details, refer to the [web search guide](/docs/guides/tools-web-search).

### 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](/docs/guides/function-calling).

Web search

==========

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

Using the [Chat Completions API](/docs/api-reference/chat), you can directly access the fine-tuned models and tool used by [Search in ChatGPT](https://openai.com/index/introducing-chatgpt-search/).

When using Chat Completions, the model always retrieves information from the web before responding to your query. To use `web\_search\_preview` as a tool that models like `gpt-4o` and `gpt-4o-mini` invoke only when necessary, switch to using the [Responses API](/docs/guides/tools-web-search?api-mode=responses).

Currently, you need to use one of these models to use web search in Chat Completions:

\* `gpt-4o-search-preview`

\* `gpt-4o-mini-search-preview`

Web search parameter example

```javascript

import OpenAI from "openai";

const client = new OpenAI();

const completion = await client.chat.completions.create({

model: "gpt-4o-search-preview",

web\_search\_options: {},

messages: [{

"role": "user",

"content": "What was a positive news story from today?"

}],

});

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

```

```python

from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(

model="gpt-4o-search-preview",

web\_search\_options={},

messages=[

{

"role": "user",

"content": "What was a positive news story from today?",

}

],

)

print(completion.choices[0].message.content)

```

```bash

curl -X POST "https://api.openai.com/v1/chat/completions" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-H "Content-type: application/json" \

-d '{

"model": "gpt-4o-search-preview",

"web\_search\_options": {},

"messages": [{

"role": "user",

"content": "What was a positive news story from today?"

}]

}'

```

Output and citations

--------------------

The API response item in the `choices` array will include:

\* `message.content` with the text result from the model, inclusive of any inline citations

\* `annotations` with a list of 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 and title of the cited source, as well as the start and end index characters in the model's response where those sources were used.

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.

```json

[

{

"index": 0,

"message": {

"role": "assistant",

"content": "the model response is here...",

"refusal": null,

"annotations": [

{

"type": "url\_citation",

"url\_citation": {

"end\_index": 985,

"start\_index": 764,

"title": "Page title...",

"url": "https://..."

}

}

]

},

"finish\_reason": "stop"

}

]

```

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](https://en.wikipedia.org/wiki/ISO\_3166-1), like `US`.

\* The `timezone` field is an [IANA timezone](https://timeapi.io/documentation/iana-timezones) like `America/Chicago`.

Customizing user location

```python

from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(

model="gpt-4o-search-preview",

web\_search\_options={

"user\_location": {

"type": "approximate",

"approximate": {

"country": "GB",

"city": "London",

"region": "London",

}

},

},

messages=[{

"role": "user",

"content": "What are the best restaurants around Granary Square?",

}],

)

print(completion.choices[0].message.content)

```

```javascript

import OpenAI from "openai";

const client = new OpenAI();

const completion = await client.chat.completions.create({

model: "gpt-4o-search-preview",

web\_search\_options: {

user\_location: {

type: "approximate",

approximate: {

country: "GB",

city: "London",

region: "London",

},

},

},

messages: [{

"role": "user",

"content": "What are the best restaurants around Granary Square?",

}],

});

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

```

```bash

curl -X POST "https://api.openai.com/v1/chat/completions" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-H "Content-type: application/json" \

-d '{

"model": "gpt-4o-search-preview",

"web\_search\_options": {

"user\_location": {

"type": "approximate",

"approximate": {

"country": "GB",

"city": "London",

"region": "London"

}

}

},

"messages": [{

"role": "user",

"content": "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\*\*: Pricing of our search tool varies based on the value of this parameter. Higher context sizes are more expensive. See tool pricing [here](/docs/pricing).

\* \*\*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](/docs/pricing) for details on costs associated with each context size.

Customizing search context size

```python

from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(

model="gpt-4o-search-preview",

web\_search\_options={

"search\_context\_size": "low",

},

messages=[{

"role": "user",

"content": "What movie won best picture in 2025?",

}],

)

print(completion.choices[0].message.content)

```

```javascript

import OpenAI from "openai";

const client = new OpenAI();

const completion = await client.chat.completions.create({

model: "gpt-4o-search-preview",

web\_search\_options: {

search\_context\_size: "low",

},

messages: [{

"role": "user",

"content": "What movie won best picture in 2025?",

}],

});

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

```

```bash

curl -X POST "https://api.openai.com/v1/chat/completions" \

-H "Authorization: Bearer $OPENAI\_API\_KEY" \

-H "Content-type: application/json" \

-d '{

"model": "gpt-4o-search-preview",

"web\_search\_options": {

"search\_context\_size": "low"

},

"messages": [{

"role": "user",

"content": "What movie won best picture in 2025?"

}]

}'

```

Limitations

-----------

Below are a few notable implementation considerations when using web search.

\* This tool does not support zero data retention or data residency ([data retention policies](/docs/guides/your-data)).

\* The [`gpt-4o-search-preview`](/docs/models/gpt-4o-search-preview) and [`gpt-4o-mini-search-preview`](/docs/models/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](/docs/api-reference/responses), web search has the same tiered rate limits as the models above.

File search

===========

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

Overview

--------

File search is a tool available in the [Responses API](/docs/api-reference/responses). 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](/docs/guides/retrieval).

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

Follow these steps to create a vector store and upload a file to it. You can use [this example file](https://cdn.openai.com/API/docs/deep\_research\_blog.pdf) or upload your own.

#### Upload the file to the File API

Upload a file

```python

import requests

from io import BytesIO

from openai import OpenAI

client = OpenAI()

def create\_file(client, file\_path):

if file\_path.startswith("http://") or file\_path.startswith("https://"):

# Download the file content from the URL

response = requests.get(file\_path)

file\_content = BytesIO(response.content)

file\_name = file\_path.split("/")[-1]

file\_tuple = (file\_name, file\_content)

result = client.files.create(

file=file\_tuple,

purpose="assistants"

)

else:

# Handle local file path

with open(file\_path, "rb") as file\_content:

result = client.files.create(

file=file\_content,

purpose="assistants"

)

print(result.id)

return result.id

# Replace with your own file path or URL

file\_id = create\_file(client, "https://cdn.openai.com/API/docs/deep\_research\_blog.pdf")

```

```javascript

import fs from "fs";

import OpenAI from "openai";

const openai = new OpenAI();

async function createFile(filePath) {

let result;

if (filePath.startsWith("http://") || filePath.startsWith("https://")) {

// Download the file content from the URL

const res = await fetch(filePath);

const buffer = await res.arrayBuffer();

const urlParts = filePath.split("/");

const fileName = urlParts[urlParts.length - 1];

const file = new File([buffer], fileName);

result = await openai.files.create({

file: file,

purpose: "assistants",

});

} else {

// Handle local file path

const fileContent = fs.createReadStream(filePath);

result = await openai.files.create({

file: fileContent,

purpose: "assistants",

});

}

return result.id;

}

// Replace with your own file path or URL

const fileId = await createFile(

"https://cdn.openai.com/API/docs/deep\_research\_blog.pdf"

);

console.log(fileId);

```

#### Create a vector store

Create a vector store

```python

vector\_store = client.vector\_stores.create(

name="knowledge\_base"

)

print(vector\_store.id)

```

```javascript

const vectorStore = await openai.vectorStores.create({

name: "knowledge\_base",

});

console.log(vectorStore.id);

```

#### Add the file to the vector store

Add a file to a vector store

```python

client.vector\_stores.files.create(

vector\_store\_id=vector\_store.id,

file\_id=file\_id

)

print(result)

```

```javascript

await openai.vectorStores.files.create(

vectorStore.id,

{

file\_id: fileId,

}

});

```

#### Check status

Run this code until the file is ready to be used (i.e., when the status is `completed`).

Check status

```python

result = client.vector\_stores.files.list(

vector\_store\_id=vector\_store.id

)

print(result)

```

```javascript

const result = await openai.vectorStores.files.list({

vector\_store\_id: vectorStore.id,

});

console.log(result);

```

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.

At the moment, you can search in only one vector store at a time, so you can include only one vector store ID when calling the file search tool.

File search tool

```python

from openai import OpenAI

client = OpenAI()

response = client.responses.create(

model="gpt-4o-mini",

input="What is deep research by OpenAI?",

tools=[{

"type": "file\_search",

"vector\_store\_ids": ["<vector\_store\_id>"]

}]

)

print(response)

```

```javascript

import OpenAI from "openai";

const openai = new OpenAI();

const response = await openai.responses.create({

model: "gpt-4o-mini",

input: "What is deep research by OpenAI?",

tools: [{

type: "file\_search",

vector\_store\_ids: ["<vector\_store\_id>"],

}],

});

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

{

"output": [

{

"type": "file\_search\_call",

"id": "fs\_67c09ccea8c48191ade9367e3ba71515",

"status": "completed",

"queries": ["What is deep research?"],

"search\_results": null

},

{

"id": "msg\_67c09cd3091c819185af2be5d13d87de",

"type": "message",

"role": "assistant",

"content": [

{

"type": "output\_text",

"text": "Deep research is a sophisticated capability that allows for extensive inquiry and synthesis of information across various domains. It is designed to conduct multi-step research tasks, gather data from multiple online sources, and provide comprehensive reports similar to what a research analyst would produce. This functionality is particularly useful in fields requiring detailed and accurate information...",

"annotations": [

{

"type": "file\_citation",

"index": 992,

"file\_id": "file-2dtbBZdjtDKS8eqWxqbgDi",

"filename": "deep\_research\_blog.pdf"

},

{

"type": "file\_citation",

"index": 992,

"file\_id": "file-2dtbBZdjtDKS8eqWxqbgDi",

"filename": "deep\_research\_blog.pdf"

},

{

"type": "file\_citation",

"index": 1176,

"file\_id": "file-2dtbBZdjtDKS8eqWxqbgDi",

"filename": "deep\_research\_blog.pdf"

},

{

"type": "file\_citation",

"index": 1176,

"file\_id": "file-2dtbBZdjtDKS8eqWxqbgDi",

"filename": "deep\_research\_blog.pdf"

}

]

}

]

}

]

}

```

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

```python

response = client.responses.create(

model="gpt-4o-mini",

input="What is deep research by OpenAI?",

tools=[{

"type": "file\_search",

"vector\_store\_ids": ["<vector\_store\_id>"],

"max\_num\_results": 2

}]

)

print(response)

```

```javascript

const response = await openai.responses.create({

model: "gpt-4o-mini",

input: "What is deep research by OpenAI?",

tools: [{

type: "file\_search",

vector\_store\_ids: ["<vector\_store\_id>"],

max\_num\_results: 2,

}],

});

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

```python

response = client.responses.create(

model="gpt-4o-mini",

input="What is deep research by OpenAI?",

tools=[{

"type": "file\_search",

"vector\_store\_ids": ["<vector\_store\_id>"]

}],

include=["file\_search\_call.results"]

)

print(response)

```

```javascript

const response = await openai.responses.create({

model: "gpt-4o-mini",

input: "What is deep research by OpenAI?",

tools: [{

type: "file\_search",

vector\_store\_ids: ["<vector\_store\_id>"],

}],

include: ["file\_search\_call.results"],

});

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](/docs/guides/retrieval), which covers:

\* How to [set attributes on vector store files](/docs/guides/retrieval#attributes)

\* How to [define filters](/docs/guides/retrieval#attribute-filtering)

Metadata filtering

```python

response = client.responses.create(

model="gpt-4o-mini",

input="What is deep research by OpenAI?",

tools=[{

"type": "file\_search",

"vector\_store\_ids": ["<vector\_store\_id>"],

"filters": {

"type": "eq",

"key": "type",

"value": "blog"

}

}]

)

print(response)

```

```javascript

const response = await openai.responses.create({

model: "gpt-4o-mini",

input: "What is deep research by OpenAI?",

tools: [{

type: "file\_search",

vector\_store\_ids: ["<vector\_store\_id>"],

filters: {

type: "eq",

key: "type",

value: "blog"

}

}]

});

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|

|.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|

Limitations

-----------

Below are some usage limitations on file search that implementors should be aware of.

\* Projects are limited to a total size of 100GB for all Files

\* Vector stores are limited to a total of 10k files

\* Individual files can be a max of 512MB (roughly 5M tokens per file)

Computer use

============

Build a computer-using agent that can perform tasks on your behalf.

Overview

--------

\*\*Computer use\*\* is a practical application of our [Computer-Using Agent](https://openai.com/index/computer-using-agent/) (CUA) model, `computer-use-preview`, which combines the vision capabilities of [GPT-4o](/docs/models/gpt-4o) with advanced reasoning to simulate controlling computer interfaces and performing tasks.

Computer use is available through the [Responses API](/docs/guides/responses-vs-chat-completions). It is not available on Chat Completions.

Computer use is in beta. Because the model is still in preview and may be susceptible to exploits and inadvertent mistakes, we discourage trusting it in fully authenticated environments or for high-stakes tasks. See [limitations](#limitations) and [risk and safety best practices](#risks-and-safety) below. You must use the Computer Use tool in line with OpenAI's [Usage Policy](https://openai.com/policies/usage-policies/) and [Business Terms](https://openai.com/policies/business-terms/).

How it works

------------

The computer use tool operates in a continuous loop. It sends computer actions, like `click(x,y)` or `type(text)`, which your code executes on a computer or browser environment and then returns screenshots of the outcomes back to the model.

In this way, your code simulates the actions of a human using a computer interface, while our model uses the screenshots to understand the state of the environment and suggest next actions.

This loop lets you automate many tasks requiring clicking, typing, scrolling, and more. For example, booking a flight, searching for a product, or filling out a form.

Refer to the [integration section](#integration) below for more details on how to integrate the computer use tool, or check out our sample app repository to set up an environment and try example integrations.

[

CUA sample app

Examples of how to integrate the computer use tool in different environments

](https://github.com/openai/openai-cua-sample-app)

Setting up your environment

---------------------------

Before integrating the tool, prepare an environment that can capture screenshots and execute the recommended actions. We recommend using a sandboxed environment for safety reasons.

In this guide, we'll show you examples using either a local browsing environment or a local virtual machine, but there are more example computer environments in our sample app.

Set up a local browsing environment

If you want to try out the computer use tool with minimal setup, you can use a browser automation framework such as [Playwright](https://playwright.dev/) or [Selenium](https://www.selenium.dev/).

Running a browser automation framework locally can pose security risks. We recommend the following setup to mitigate them:

\* Use a sandboxed environment

\* Set `env` to an empty object to avoid exposing host environment variables to the browser

\* Set flags to disable extensions and the file system

#### Start a browser instance

You can start browser instances using your preferred language by installing the corresponding SDK.

For example, to start a Playwright browser instance, install the Playwright SDK:

\* Python: `pip install playwright`

\* JavaScript: `npm i playwright` then `npx playwright install`

Then run the following code:

Start a browser instance

```javascript

import { chromium } from "playwright";

const browser = await chromium.launch({

headless: false,

chromiumSandbox: true,

env: {},

args: ["--disable-extensions", "--disable-file-system"],

});

const page = await browser.newPage();

await page.setViewportSize({ width: 1024, height: 768 });

await page.goto("https://bing.com");

await page.waitForTimeout(10000);

browser.close();

```

```python

from playwright.sync\_api import sync\_playwright

with sync\_playwright() as p:

browser = p.chromium.launch(

headless=False,

chromium\_sandbox=True,

env={},

args=[

"--disable-extensions",

"--disable-file-system"

]

)

page = browser.new\_page()

page.set\_viewport\_size({"width": 1024, "height": 768})

page.goto("https://bing.com")

page.wait\_for\_timeout(10000)

```

Set up a local virtual machine

If you'd like to use the computer use tool beyond just a browser interface, you can set up a local virtual machine instead, using a tool like [Docker](https://www.docker.com/). You can then connect to this local machine to execute computer use actions.

#### Start Docker

If you don't have Docker installed, you can install it from [their website](https://www.docker.com). Once installed, make sure Docker is running on your machine.

#### Create a Dockerfile

Create a Dockerfile to define the configuration of your virtual machine.

Here is an example Dockerfile that starts an Ubuntu virtual machine with a VNC server:

Dockerfile

```json

FROM ubuntu:22.04

ENV DEBIAN\_FRONTEND=noninteractive

# 1) Install Xfce, x11vnc, Xvfb, xdotool, etc., but remove any screen lockers or power managers

RUN apt-get update && apt-get install -y xfce4 xfce4-goodies x11vnc xvfb xdotool imagemagick x11-apps sudo software-properties-common imagemagick && apt-get remove -y light-locker xfce4-screensaver xfce4-power-manager || true && apt-get clean && rm -rf /var/lib/apt/lists/\*

# 2) Add the mozillateam PPA and install Firefox ESR

RUN add-apt-repository ppa:mozillateam/ppa && apt-get update && apt-get install -y --no-install-recommends firefox-esr && update-alternatives --set x-www-browser /usr/bin/firefox-esr && apt-get clean && rm -rf /var/lib/apt/lists/\*

# 3) Create non-root user

RUN useradd -ms /bin/bash myuser && echo "myuser ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers

USER myuser

WORKDIR /home/myuser

# 4) Set x11vnc password ("secret")

RUN x11vnc -storepasswd secret /home/myuser/.vncpass

# 5) Expose port 5900 and run Xvfb, x11vnc, Xfce (no login manager)

EXPOSE 5900

CMD ["/bin/sh", "-c", " Xvfb :99 -screen 0 1280x800x24 >/dev/null 2>&1 & x11vnc -display :99 -forever -rfbauth /home/myuser/.vncpass -listen 0.0.0.0 -rfbport 5900 >/dev/null 2>&1 & export DISPLAY=:99 && startxfce4 >/dev/null 2>&1 & sleep 2 && echo 'Container running!' && tail -f /dev/null "]

```

#### Build the Docker image

Build the Docker image by running the following command in the directory containing the Dockerfile:

```bash

docker build -t cua-image .

```

#### Run the Docker container locally

Start the Docker container with the following command:

```bash

docker run --rm -it --name cua-image -p 5900:5900 -e DISPLAY=:99 cua-image

```

#### Execute commands on the container

Now that your container is running, you can execute commands on it. For example, we can define a helper function to execute commands on the container that will be used in the next steps.

Execute commands on the container

```python

def docker\_exec(cmd: str, container\_name: str, decode=True) -> str:

safe\_cmd = cmd.replace('"', '\"')

docker\_cmd = f'docker exec {container\_name} sh -c "{safe\_cmd}"'

output = subprocess.check\_output(docker\_cmd, shell=True)

if decode:

return output.decode("utf-8", errors="ignore")

return output

class VM:

def \_\_init\_\_(self, display, container\_name):

self.display = display

self.container\_name = container\_name

vm = VM(display=":99", container\_name="cua-image")

```

```javascript

async function dockerExec(cmd, containerName, decode = true) {

const safeCmd = cmd.replace(/"/g, '\"');

const dockerCmd = `docker exec ${containerName} sh -c "${safeCmd}"`;

const output = await execAsync(dockerCmd, {

encoding: decode ? "utf8" : "buffer",

});

const result = output && output.stdout ? output.stdout : output;

if (decode) {

return result.toString("utf-8");

}

return result;

}

const vm = {

display: ":99",

containerName: "cua-image",

};

```

Integrating the CUA loop

------------------------

These are the high-level steps you need to follow to integrate the computer use tool in your application:

1. \*\*Send a request to the model\*\*: Include the `computer` tool as part of the available tools, specifying the display size and environment. You can also include in the first request a screenshot of the initial state of the environment.

2. \*\*Receive a response from the model\*\*: Check if the response has any `computer\_call` items. This tool call contains a suggested action to take to progress towards the specified goal. These actions could be clicking at a given position, typing in text, scrolling, or even waiting.

3. \*\*Execute the requested action\*\*: Execute through code the corresponding action on your computer or browser environment.

4. \*\*Capture the updated state\*\*: After executing the action, capture the updated state of the environment as a screenshot.

5. \*\*Repeat\*\*: Send a new request with the updated state as a `computer\_call\_output`, and repeat this loop until the model stops requesting actions or you decide to stop.

![Computer use diagram](https://cdn.openai.com/API/docs/images/cua\_diagram.png)

### 1\. Send a request to the model

Send a request to create a Response with the `computer-use-preview` model equipped with the `computer\_use\_preview` tool. This request should include details about your environment, along with an initial input prompt.

If you want to show a summary of the reasoning performed by the model, you can include the `generate\_summary` parameter in the request. This can be helpful if you want to debug or show what's happening behind the scenes in your interface. The summary can either be `concise` or `detailed`.

Optionally, you can include a screenshot of the initial state of the environment.

To be able to use the `computer\_use\_preview` tool, you need to set the `truncation` parameter to `"auto"` (by default, truncation is disabled).

Send a CUA request

```javascript

import OpenAI from "openai";

const openai = new OpenAI();

const response = await openai.responses.create({

model: "computer-use-preview",

tools: [

{

type: "computer\_use\_preview",

display\_width: 1024,

display\_height: 768,

environment: "browser", // other possible values: "mac", "windows", "ubuntu"

},

],

input: [

{

role: "user",

content: "Check the latest OpenAI news on bing.com.",

},

// Optional: include a screenshot of the initial state of the environment

// {

// type: "input\_image",

// image\_url: `data:image/png;base64,${screenshot\_base64}`

// }

],

reasoning: {

generate\_summary: "concise",

},

truncation: "auto",

});

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

```

```python

from openai import OpenAI

client = OpenAI()

response = client.responses.create(

model="computer-use-preview",

tools=[{

"type": "computer\_use\_preview",

"display\_width": 1024,

"display\_height": 768,

"environment": "browser" # other possible values: "mac", "windows", "ubuntu"

}],

input=[

{

"role": "user",

"content": "Check the latest OpenAI news on bing.com."

}

# Optional: include a screenshot of the initial state of the environment

# {

# type: "input\_image",

# image\_url: f"data:image/png;base64,{screenshot\_base64}"

# }

],

reasoning={

"generate\_summary": "concise",

},

truncation="auto"

)

print(response.output)

```

### 2\. Receive a suggested action

The model returns an output that contains either a `computer\_call` item, just text, or other tool calls, depending on the state of the conversation.

Examples of `computer\_call` items are a click, a scroll, a key press, or any other event defined in the [API reference](/docs/api-reference/computer-use). In our example, the item is a click action:

CUA suggested action

```json

"output": [

{

"type": "reasoning",

"id": "rs\_67cc...",

"summary": [

{

"type": "summary\_text",

"text": "Clicking on the browser address bar."

}

]

},

{

"type": "computer\_call",

"id": "cu\_67cc...",

"call\_id": "call\_zw3...",

"action": {

"type": "click",

"button": "left",

"x": 156,

"y": 50

},

"pending\_safety\_checks": [],

"status": "completed"

}

]

```

#### Reasoning items

The model may return a `reasoning` item in the response output for some actions. If you don't use the `previous\_response\_id` parameter as shown in [Step 5](#5-repeat) and manage the inputs array on your end, make sure to include those reasoning items along with the computer calls when sending the next request to the CUA model–or the request will fail.

The reasoning items are only compatible with the same model that produced them (in this case, `computer-use-preview`). If you implement a flow where you use several models with the same conversation history, you should filter these reasoning items out of the inputs array you send to other models.

#### Safety checks

The model may return safety checks with the `pending\_safety\_check` parameter. Refer to the section on how to [acknowledge safety checks](#acknowledge-safety-checks) below for more details.

### 3\. Execute the action in your environment

Execute the corresponding actions on your computer or browser. How you map a computer call to actions through code depends on your environment. This code shows example implementations for the most common computer actions.

Playwright

Execute the action

```javascript

async function handleModelAction(page, action) {

// Given a computer action (e.g., click, double\_click, scroll, etc.),

// execute the corresponding operation on the Playwright page.

const actionType = action.type;

try {

switch (actionType) {

case "click": {

const { x, y, button = "left" } = action;

console.log(`Action: click at (${x}, ${y}) with button '${button}'`);

await page.mouse.click(x, y, { button });

break;

}

case "scroll": {

const { x, y, scrollX, scrollY } = action;

console.log(

`Action: scroll at (${x}, ${y}) with offsets (scrollX=${scrollX}, scrollY=${scrollY})`

);

await page.mouse.move(x, y);

await page.evaluate(`window.scrollBy(${scrollX}, ${scrollY})`);

break;

}

case "keypress": {

const { keys } = action;

for (const k of keys) {

console.log(`Action: keypress '${k}'`);

// A simple mapping for common keys; expand as needed.

if (k.includes("ENTER")) {

await page.keyboard.press("Enter");

} else if (k.includes("SPACE")) {

await page.keyboard.press(" ");

} else {

await page.keyboard.press(k);

}

}

break;

}

case "type": {

const { text } = action;

console.log(`Action: type text '${text}'`);

await page.keyboard.type(text);

break;

}

case "wait": {

console.log(`Action: wait`);

await page.waitForTimeout(2000);

break;

}

case "screenshot": {

// Nothing to do as screenshot is taken at each turn

console.log(`Action: screenshot`);

break;

}

// Handle other actions here

default:

console.log("Unrecognized action:", action);

}

} catch (e) {

console.error("Error handling action", action, ":", e);

}

}

```

```python

def handle\_model\_action(page, action):

"""

Given a computer action (e.g., click, double\_click, scroll, etc.),

execute the corresponding operation on the Playwright page.

"""

action\_type = action.type

try:

match action\_type:

case "click":

x, y = action.x, action.y

button = action.button

print(f"Action: click at ({x}, {y}) with button '{button}'")

# Not handling things like middle click, etc.

if button != "left" and button != "right":

button = "left"

page.mouse.click(x, y, button=button)

case "scroll":

x, y = action.x, action.y

scroll\_x, scroll\_y = action.scroll\_x, action.scroll\_y

print(f"Action: scroll at ({x}, {y}) with offsets (scroll\_x={scroll\_x}, scroll\_y={scroll\_y})")

page.mouse.move(x, y)

page.evaluate(f"window.scrollBy({scroll\_x}, {scroll\_y})")

case "keypress":

keys = action.keys

for k in keys:

print(f"Action: keypress '{k}'")

# A simple mapping for common keys; expand as needed.

if k.lower() == "enter":

page.keyboard.press("Enter")

elif k.lower() == "space":

page.keyboard.press(" ")

else:

page.keyboard.press(k)

case "type":

text = action.text

print(f"Action: type text: {text}")

page.keyboard.type(text)

case "wait":

print(f"Action: wait")

time.sleep(2)

case "screenshot":

# Nothing to do as screenshot is taken at each turn

print(f"Action: screenshot")

# Handle other actions here

case \_:

print(f"Unrecognized action: {action}")

except Exception as e:

print(f"Error handling action {action}: {e}")

```

Docker

Execute the action

```javascript

async function handleModelAction(vm, action) {

// Given a computer action (e.g., click, double\_click, scroll, etc.),

// execute the corresponding operation on the Docker environment.

const actionType = action.type;

try {

switch (actionType) {

case "click": {

const { x, y, button = "left" } = action;

const buttonMap = { left: 1, middle: 2, right: 3 };

const b = buttonMap[button] || 1;

console.log(`Action: click at (${x}, ${y}) with button '${button}'`);

await dockerExec(

`DISPLAY=${vm.display} xdotool mousemove ${x} ${y} click ${b}`,

vm.containerName

);

break;

}

case "scroll": {

const { x, y, scrollX, scrollY } = action;

console.log(

`Action: scroll at (${x}, ${y}) with offsets (scrollX=${scrollX}, scrollY=${scrollY})`

);

await dockerExec(

`DISPLAY=${vm.display} xdotool mousemove ${x} ${y}`,

vm.containerName

);

// For vertical scrolling, use button 4 for scroll up and button 5 for scroll down.

if (scrollY !== 0) {

const button = scrollY < 0 ? 4 : 5;

const clicks = Math.abs(scrollY);

for (let i = 0; i < clicks; i++) {

await dockerExec(

`DISPLAY=${vm.display} xdotool click ${button}`,

vm.containerName

);

}

}

break;

}

case "keypress": {

const { keys } = action;

for (const k of keys) {

console.log(`Action: keypress '${k}'`);

// A simple mapping for common keys; expand as needed.

if (k.includes("ENTER")) {

await dockerExec(

`DISPLAY=${vm.display} xdotool key 'Return'`,

vm.containerName

);

} else if (k.includes("SPACE")) {

await dockerExec(

`DISPLAY=${vm.display} xdotool key 'space'`,

vm.containerName

);

} else {

await dockerExec(

`DISPLAY=${vm.display} xdotool key '${k}'`,

vm.containerName

);

}

}

break;

}

case "type": {

const { text } = action;

console.log(`Action: type text '${text}'`);

await dockerExec(

`DISPLAY=${vm.display} xdotool type '${text}'`,

vm.containerName

);

break;

}

case "wait": {

console.log(`Action: wait`);

await new Promise((resolve) => setTimeout(resolve, 2000));

break;

}

case "screenshot": {

// Nothing to do as screenshot is taken at each turn

console.log(`Action: screenshot`);

break;

}

// Handle other actions here

default:

console.log("Unrecognized action:", action);

}

} catch (e) {

console.error("Error handling action", action, ":", e);

}

}

```

```python

def handle\_model\_action(vm, action):

"""

Given a computer action (e.g., click, double\_click, scroll, etc.),

execute the corresponding operation on the Docker environment.

"""

action\_type = action.type

try:

match action\_type:

case "click":

x, y = int(action.x), int(action.y)

button\_map = {"left": 1, "middle": 2, "right": 3}

b = button\_map.get(action.button, 1)

print(f"Action: click at ({x}, {y}) with button '{action.button}'")

docker\_exec(f"DISPLAY={vm.display} xdotool mousemove {x} {y} click {b}", vm.container\_name)

case "scroll":

x, y = int(action.x), int(action.y)

scroll\_x, scroll\_y = int(action.scroll\_x), int(action.scroll\_y)

print(f"Action: scroll at ({x}, {y}) with offsets (scroll\_x={scroll\_x}, scroll\_y={scroll\_y})")

docker\_exec(f"DISPLAY={vm.display} xdotool mousemove {x} {y}", vm.container\_name)

# For vertical scrolling, use button 4 (scroll up) or button 5 (scroll down)

if scroll\_y != 0:

button = 4 if scroll\_y < 0 else 5

clicks = abs(scroll\_y)

for \_ in range(clicks):

docker\_exec(f"DISPLAY={vm.display} xdotool click {button}", vm.container\_name)

case "keypress":

keys = action.keys

for k in keys:

print(f"Action: keypress '{k}'")

# A simple mapping for common keys; expand as needed.

if k.lower() == "enter":

docker\_exec(f"DISPLAY={vm.display} xdotool key 'Return'", vm.container\_name)

elif k.lower() == "space":

docker\_exec(f"DISPLAY={vm.display} xdotool key 'space'", vm.container\_name)

else:

docker\_exec(f"DISPLAY={vm.display} xdotool key '{k}'", vm.container\_name)

case "type":

text = action.text

print(f"Action: type text: {text}")

docker\_exec(f"DISPLAY={vm.display} xdotool type '{text}'", vm.container\_name)

case "wait":

print(f"Action: wait")

time.sleep(2)

case "screenshot":

# Nothing to do as screenshot is taken at each turn

print(f"Action: screenshot")

# Handle other actions here

case \_:

print(f"Unrecognized action: {action}")

except Exception as e:

print(f"Error handling action {action}: {e}")

```

### 4\. Capture the updated screenshot

After executing the action, capture the updated state of the environment as a screenshot, which also differs depending on your environment.

Playwright

Capture and send the updated screenshot

```javascript

async function getScreenshot(page) {

// Take a full-page screenshot using Playwright and return the image bytes.

return await page.screenshot();

}

```

```python

def get\_screenshot(page):

"""

Take a full-page screenshot using Playwright and return the image bytes.

"""

return page.screenshot()

```

Docker

Capture and send the updated screenshot

```javascript

async function getScreenshot(vm) {

// Take a screenshot, returning raw bytes.

const cmd = `export DISPLAY=${vm.display} && import -window root png:-`;

const screenshotBuffer = await dockerExec(cmd, vm.containerName, false);

return screenshotBuffer;

}

```

```python

def get\_screenshot(vm):

"""

Takes a screenshot, returning raw bytes.

"""

cmd = (

f"export DISPLAY={vm.display} && "

"import -window root png:-"

)

screenshot\_bytes = docker\_exec(cmd, vm.container\_name, decode=False)

return screenshot\_bytes

```

### 5\. Repeat

Once you have the screenshot, you can send it back to the model as a `computer\_call\_output` to get the next action. Repeat these steps as long as you get a `computer\_call` item in the response.

Repeat steps in a loop

```javascript

import OpenAI from "openai";

const openai = new OpenAI();

async function computerUseLoop(instance, response) {

/\*\*

\* Run the loop that executes computer actions until no 'computer\_call' is found.

\*/

while (true) {

const computerCalls = response.output.filter(

(item) => item.type === "computer\_call"

);

if (computerCalls.length === 0) {

console.log("No computer call found. Output from model:");

response.output.forEach((item) => {

console.log(JSON.stringify(item, null, 2));

});

break; // Exit when no computer calls are issued.

}

// We expect at most one computer call per response.

const computerCall = computerCalls[0];

const lastCallId = computerCall.call\_id;

const action = computerCall.action;

// Execute the action (function defined in step 3)

handleModelAction(instance, action);

await new Promise((resolve) => setTimeout(resolve, 1000)); // Allow time for changes to take effect.

// Take a screenshot after the action (function defined in step 4)

const screenshotBytes = await getScreenshot(instance);

const screenshotBase64 = Buffer.from(screenshotBytes).toString("base64");

// Send the screenshot back as a computer\_call\_output

response = await openai.responses.create({

model: "computer-use-preview",

previous\_response\_id: response.id,

tools: [

{

type: "computer\_use\_preview",

display\_width: 1024,

display\_height: 768,

environment: "browser",

},

],

input: [

{

call\_id: lastCallId,

type: "computer\_call\_output",

output: {

type: "input\_image",

image\_url: `data:image/png;base64,${screenshotBase64}`,

},

},

],

truncation: "auto",

});

}

return response;

}

```

```python

import time

import base64

from openai import OpenAI

client = OpenAI()

def computer\_use\_loop(instance, response):

"""

Run the loop that executes computer actions until no 'computer\_call' is found.

"""

while True:

computer\_calls = [item for item in response.output if item.type == "computer\_call"]

if not computer\_calls:

print("No computer call found. Output from model:")

for item in response.output:

print(item)

break # Exit when no computer calls are issued.

# We expect at most one computer call per response.

computer\_call = computer\_calls[0]

last\_call\_id = computer\_call.call\_id

action = computer\_call.action

# Execute the action (function defined in step 3)

handle\_model\_action(instance, action)

time.sleep(1) # Allow time for changes to take effect.

# Take a screenshot after the action (function defined in step 4)

screenshot\_bytes = get\_screenshot(instance)

screenshot\_base64 = base64.b64encode(screenshot\_bytes).decode("utf-8")

# Send the screenshot back as a computer\_call\_output

response = client.responses.create(

model="computer-use-preview",

previous\_response\_id=response.id,

tools=[

{

"type": "computer\_use\_preview",

"display\_width": 1024,

"display\_height": 768,

"environment": "browser"

}

],

input=[

{

"call\_id": last\_call\_id,

"type": "computer\_call\_output",

"output": {

"type": "input\_image",

"image\_url": f"data:image/png;base64,{screenshot\_base64}"

}

}

],

truncation="auto"

)

return response

```

#### Handling conversation history

You can use the `previous\_response\_id` parameter to link the current request to the previous response. We recommend using this method if you don't want to manage the conversation history on your side.

If you do not want to use this parameter, you should make sure to include in your inputs array all the items returned in the response output of the previous request, including reasoning items if present.

### Acknowledge safety checks

We have implemented safety checks in the API to help protect against prompt injection and model mistakes. These checks include:

\* Malicious instruction detection: we evaluate the screenshot image and check if it contains adversarial content that may change the model's behavior.

\* Irrelevant domain detection: we evaluate the `current\_url` (if provided) and check if the current domain is considered relevant given the conversation history.

\* Sensitive domain detection: we check the `current\_url` (if provided) and raise a warning when we detect the user is on a sensitive domain.

If one or multiple of the above checks is triggered, a safety check is raised when the model returns the next `computer\_call`, with the `pending\_safety\_checks` parameter.

Pending safety checks

```json

"output": [

{

"type": "reasoning",

"id": "rs\_67cb...",

"summary": [

{

"type": "summary\_text",

"text": "Exploring 'File' menu option."

}

]

},

{

"type": "computer\_call",

"id": "cu\_67cb...",

"call\_id": "call\_nEJ...",

"action": {

"type": "click",

"button": "left",

"x": 135,

"y": 193

},

"pending\_safety\_checks": [

{

"id": "cu\_sc\_67cb...",

"code": "malicious\_instructions",

"message": "We've detected instructions that may cause your application to perform malicious or unauthorized actions. Please acknowledge this warning if you'd like to proceed."

}

],

"status": "completed"

}

]

```

You need to pass the safety checks back as `acknowledged\_safety\_checks` in the next request in order to proceed. In all cases where `pending\_safety\_checks` are returned, actions should be handed over to the end user to confirm model behavior and accuracy.

\* `malicious\_instructions` and `irrelevant\_domain`: end users should review model actions and confirm that the model is behaving as intended.

\* `sensitive\_domain`: ensure an end user is actively monitoring the model actions on these sites. Exact implementation of this "watch mode" may vary by application, but a potential example could be collecting user impression data on the site to make sure there is active end user engagement with the application.

Acknowledge safety checks

```python

from openai import OpenAI

client = OpenAI()

response = client.responses.create(

model="computer-use-preview",

previous\_response\_id="<previous\_response\_id>",

tools=[{

"type": "computer\_use\_preview",

"display\_width": 1024,

"display\_height": 768,

"environment": "browser"

}],

input=[

{

"type": "computer\_call\_output",

"call\_id": "<call\_id>",

"acknowledged\_safety\_checks": [

{

"id": "<safety\_check\_id>",

"code": "malicious\_instructions",

"message": "We've detected instructions that may cause your application to perform malicious or unauthorized actions. Please acknowledge this warning if you'd like to proceed."

}

],

"output": {

"type": "computer\_screenshot",

"image\_url": "<image\_url>"

}

}

],

truncation="auto"

)

```

```javascript

import OpenAI from "openai";

const openai = new OpenAI();

const response = await openai.responses.create({

model: "computer-use-preview",

previous\_response\_id: "<previous\_response\_id>",

tools: [{

type: "computer\_use\_preview",

display\_width: 1024,

display\_height: 768,

environment: "browser"

}],

input: [

{

"type": "computer\_call\_output",

"call\_id": "<call\_id>",

"acknowledged\_safety\_checks": [

{

"id": "<safety\_check\_id>",

"code": "malicious\_instructions",

"message": "We've detected instructions that may cause your application to perform malicious or unauthorized actions. Please acknowledge this warning if you'd like to proceed."

}

],

"output": {

"type": "computer\_screenshot",

"image\_url": "<image\_url>"

}

}

],

truncation: "auto",

});

```

### Final code

Putting it all together, the final code should include:

1. The initialization of the environment

2. A first request to the model with the `computer` tool

3. A loop that executes the suggested action in your environment

4. A way to acknowledge safety checks and give end users a chance to confirm actions

To see end-to-end example integrations, refer to our CUA sample app repository.

[

CUA sample app

Examples of how to integrate the computer use tool in different environments

](https://github.com/openai/openai-cua-sample-app)

Limitations

-----------

We recommend using the `computer-use-preview` model for browser-based tasks. The model may be susceptible to inadvertent model mistakes, especially in non-browser environments that it is less used to.

For example, `computer-use-preview`'s performance on OSWorld is currently 38.1%, indicating that the model is not yet highly reliable for automating tasks on an OS. More details about the model and related safety work can be found in our updated [system card](https://openai.com/index/operator-system-card/).

Some other behavior limitations to be aware of:

\* The [`computer-use-preview` model](/docs/models/computer-use-preview) has constrained rate limits and feature support, described on its model detail page.

\* This tool does not support zero data retention ([data retention policies](/docs/guides/your-data)).

Risks and safety

----------------

Computer use presents unique risks that differ from those in standard API features or chat interfaces, especially when interacting with the internet.

There are a number of best practices listed below that you should follow to mitigate these risks.

#### Human in the loop for high-stakes tasks

Avoid tasks that are high-stakes or require high levels of accuracy. The model may make mistakes that are challenging to reverse. As mentioned above, the model is still prone to mistakes, especially on non-browser surfaces. While we expect the model to request user confirmation before proceeding with certain higher-impact decisions, this is not fully reliable. Ensure a human is in the loop to confirm model actions with real-world consequences.

#### Beware of prompt injections

A prompt injection occurs when an AI model mistakenly follows untrusted instructions appearing in its input. For the `computer-use-preview` model, this may manifest as it seeing something in the provided screenshot, like a malicious website or email, that instructs it to do something that the user does not want, and it complies. To avoid prompt injection risk, limit computer use access to trusted, isolated environments like a sandboxed browser or container.

#### Use blocklists and allowlists

Implement a blocklist or an allowlist of websites, actions, and users. For example, if you're using the computer use tool to book tickets on a website, create an allowlist of only the websites you expect to use in that workflow.

#### Send user IDs

Send end-user IDs (optional param) to help OpenAI monitor and detect abuse.

#### Use our safety checks

The following safety checks are available to protect against prompt injection and model mistakes:

\* Malicious instruction detection

\* Irrelevant domain detection

\* Sensitive domain detection

When you receive a `pending\_safety\_check`, you should increase oversight into model actions, for example by handing over to an end user to explicitly acknowledge the desire to proceed with the task and ensure that the user is actively monitoring the agent's actions (e.g., by implementing something like a watch mode similar to [Operator](https://operator.chatgpt.com/)). Essentially, when safety checks fire, a human should come into the loop.

Read the [acknowledge safety checks](#acknowledge-safety-checks) section above for more details on how to proceed when you receive a `pending\_safety\_check`.

Where possible, it is highly recommended to pass in the optional parameter `current\_url` as part of the `computer\_call\_output`, as it can help increase the accuracy of our safety checks.

Using current URL

```json

{

"type": "computer\_call\_output",

"call\_id": "call\_7OU...",

"acknowledged\_safety\_checks": [],

"output": {

"type": "computer\_screenshot",

"image\_url": "..."

},

"current\_url": "https://openai.com"

}

```

#### Additional safety precautions

Implement additional safety precautions as best suited for your application, such as implementing guardrails that run in parallel of the computer use loop.

#### Comply with our Usage Policy

Remember, you are responsible for using our services in compliance with the [OpenAI Usage Policy](https://openai.com/policies/usage-policies/) and [Business Terms](https://openai.com/policies/business-terms/), and we encourage you to employ our safety features and tools to help ensure this compliance.