Azure OpenAI Responses API

08/27/2025

The Responses API is a new stateful API from Azure OpenAI. It brings together the best capabilities from the chat completions and assistants API in one unified experience. The Responses API also adds support for the new `computer-use-preview` model which powers the Computer use capability.

Responses API

API support

- v1 API is required for access to the latest features

Region Availability

The responses API is currently available in the following regions:

- australiaeast
- eastus
- eastus2
- francecentral
- japaneast
- norwayeast
- polandcentral
- southindia
- swedencentral
- switzerlandnorth
- uaenorth
- uksouth
- westus
- westus3

Model support

- `gpt-5` (Version: `2025-08-07`)

- `gpt-5-mini` (Version: `2025-08-07`)
- `gpt-5-nano` (Version: `2025-08-07`)
- `gpt-5-chat` (Version: `2025-08-07`)
- `gpt-4o` (Versions: `2024-11-20`, `2024-08-06`, `2024-05-13`)
- `gpt-4o-mini` (Version: `2024-07-18`)
- `computer-use-preview`
- `gpt-4.1` (Version: `2025-04-14`)
- `gpt-4.1-nano` (Version: `2025-04-14`)
- `gpt-4.1-mini` (Version: `2025-04-14`)
- `gpt-image-1` (Version: `2025-04-15`)
- `o1` (Version: `2024-12-17`)
- `o3-mini` (Version: `2025-01-31`)
- `o3` (Version: `2025-04-16`)
- `o4-mini` (Version: `2025-04-16`)

Not every model is available in the regions supported by the responses API. Check the models page for model region availability.

> **Note**
>
> Not currently supported:
>
> - The web search tool
> - Image generation using multi-turn editing and streaming - coming soon
> - Images can't be uploaded as a file and then referenced as input. Coming soon.
>
> There's a known issue with the following:
>
> - PDF as an input file **is now supported**, but setting file upload purpose to `user_data` is not currently supported.
> - Performance issues when background mode is used with streaming. The issue is expected to be resolved soon.

Reference documentation

- Responses API reference documentation

Getting started with the responses API

To access the responses API commands, you need to upgrade your version of the OpenAI library.

Windows Command Prompt

```
pip install --upgrade openai
```

Generate a text response

Python (API Key)

Python

```python
import os
from openai import OpenAI

client = OpenAI(
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    base_url="https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
)

response = client.responses.create(
  model="gpt-4.1-nano", # Replace with your model deployment name
  input="This is a test.",
)

print(response.model_dump_json(indent=2))
```

ⓘ Important

Use API keys with caution. Don't include the API key directly in your code, and never post it publicly. If you use an API key, store it securely in Azure Key Vault. For more information about using API keys securely in your apps, see API keys with Azure Key Vault.

For more information about AI services security, see Authenticate requests to Azure AI services.

Retrieve a response

To retrieve a response from a previous call to the responses API.

Python (API Key)

Python

```python
import os
from openai import OpenAI

client = OpenAI(
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    base_url="https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
)

response = client.responses.retrieve("resp_67cb61fa3a448190bcf2c42d96f0d1a8")
```

Important

Use API keys with caution. Don't include the API key directly in your code, and never post it publicly. If you use an API key, store it securely in Azure Key Vault. For more information about using API keys securely in your apps, see API keys with Azure Key Vault.

For more information about AI services security, see Authenticate requests to Azure AI services.

Delete response

By default response data is retained for 30 days. To delete a response, you can use `response.delete"("{response_id})`

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
```

```
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.delete("resp_67cb61fa3a448190bcf2c42d96f0d1a8")

print(response)
```

Chaining responses together

You can chain responses together by passing the `response.id` from the previous response to the `previous_response_id` parameter.

Python

```
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model="gpt-4o",  # replace with your model deployment name
    input="Define and explain the concept of catastrophic forgetting?"
)

second_response = client.responses.create(
    model="gpt-4o",  # replace with your model deployment name
    previous_response_id=response.id,
    input=[{"role": "user", "content": "Explain this at a level that could be under-
stood by a college freshman"}]
)
print(second_response.model_dump_json(indent=2))
```

Note from the output that even though we never shared the first input question with the `second_response` API call, by passing the `previous_response_id` the model has full context of previous question and response to answer the new question.

Output:

JSON

```
{
  "id": "resp_67cbc9705fc08190bbe455c5ba3d6daf",
```

```
  "created_at": 1741408624.0,
  "error": null,
  "incomplete_details": null,
  "instructions": null,
  "metadata": {},
  "model": "gpt-4o-2024-08-06",
  "object": "response",
  "output": [
    {
      "id": "msg_67cbc970fd0881908353a4298996b3f6",
      "content": [
        {
          "annotations": [],
          "text": "Sure! Imagine you are studying for exams in different subjects
like math, history, and biology. You spend a lot of time studying math first and get
really good at it. But then, you switch to studying history. If you spend all your
time and focus on history, you might forget some of the math concepts you learned
earlier because your brain fills up with all the new history facts. \n\nIn the world
of artificial intelligence (AI) and machine learning, a similar thing can happen with
computers. We use special programs called neural networks to help computers learn
things, sort of like how our brain works. But when a neural network learns a new
task, it can forget what it learned before. This is what we call \"catastrophic
forgetting.\"\n\nSo, if a neural network learned how to recognize cats in pictures,
and then you teach it how to recognize dogs, it might get really good at recognizing
dogs but suddenly become worse at recognizing cats. This happens because the process
of learning new information can overwrite or mess with the old information in its
\"memory.\"\n\nScientists and engineers are working on ways to help computers remem-
ber everything they learn, even as they keep learning new things, just like students
have to remember math, history, and biology all at the same time for their exams.
They use different techniques to make sure the neural network doesn’t forget the im-
portant stuff it learned before, even when it gets new information.",
          "type": "output_text"
        }
      ],
      "role": "assistant",
      "status": null,
      "type": "message"
    }
  ],
  "parallel_tool_calls": null,
  "temperature": 1.0,
  "tool_choice": null,
  "tools": [],
  "top_p": 1.0,
  "max_output_tokens": null,
  "previous_response_id": "resp_67cbc96babbc8190b0f69aedc655f173",
  "reasoning": null,
  "status": "completed",
  "text": null,
  "truncation": null,
  "usage": {
```

```
    "input_tokens": 405,
    "output_tokens": 285,
    "output_tokens_details": {
      "reasoning_tokens": 0
    },
    "total_tokens": 690
  },
  "user": null,
  "reasoning_effort": null
}
```


Chaining responses manually

Alternatively you can manually chain responses together using the method below:

Python

```
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)


inputs = [{"type": "message", "role": "user", "content": "Define and explain the concept of catastrophic forgetting?"}]

response = client.responses.create(
    model="gpt-4o",  # replace with your model deployment name
    input=inputs
)

inputs += response.output

inputs.append({"role": "user", "type": "message", "content": "Explain this at a level that could be understood by a college freshman"})


second_response = client.responses.create(
    model="gpt-4o",
    input=inputs
)

print(second_response.model_dump_json(indent=2))
```

Streaming

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    input = "This is a test",
    model = "o4-mini", # replace with model deployment name
    stream = True
)

for event in response:
    if event.type == 'response.output_text.delta':
        print(event.delta, end='')
```

Function calling

The responses API supports function calling.

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model="gpt-4o",  # replace with your model deployment name
    tools=[
        {
            "type": "function",
            "name": "get_weather",
            "description": "Get the weather for a location",
            "parameters": {
                "type": "object",
                "properties": {
```

```
                    "location": {"type": "string"},
                },
                "required": ["location"],
            },
        }
    ],
    input=[{"role": "user", "content": "What's the weather in San Francisco?"}],
)

print(response.model_dump_json(indent=2))

# To provide output to tools, add a response for each tool call to an array passed
# to the next response as `input`
input = []
for output in response.output:
    if output.type == "function_call":
        match output.name:
            case "get_weather":
                input.append(
                    {
                        "type": "function_call_output",
                        "call_id": output.call_id,
                        "output": '{"temperature": "70 degrees"}',
                    }
                )
            case _:
                raise ValueError(f"Unknown function call: {output.name}")

second_response = client.responses.create(
    model="gpt-4o",
    previous_response_id=response.id,
    input=input
)

print(second_response.model_dump_json(indent=2))
```

Code Interpreter

The Code Interpreter tool enables models to write and execute Python code in a secure, sandboxed environment. It supports a range of advanced tasks, including:

- Processing files with varied data formats and structures
- Generating files that include data and visualizations (for example, graphs)
- Iteratively writing and running code to solve problems—models can debug and retry code until successful

- Enhancing visual reasoning in supported models (for example, o3, o4-mini) by enabling image transformations such as cropping, zooming, and rotation
- This tool is especially useful for scenarios involving data analysis, mathematical computation, and code generation.

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses?api-
version=preview \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -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?"
    }'
```

Python

```
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

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

response = 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(response.output)
```

Containers

> **Important**
>
> Code Interpreter has additional charges beyond the token based fees for Azure OpenAI usage. If your Responses API calls Code Interpreter simultaneously in two different threads, two code interpreter sessions are created. Each session is active by default for 1 hour with an idle timeout of 30 minutes.

The Code Interpreter tool requires a container—a fully sandboxed virtual machine where the model can execute Python code. Containers can include uploaded files or files generated during execution.

To create a container, specify `"container": { "type": "auto", "files": ["file-1", "file-2"] }` in the tool configuration when creating a new Response object. This automatically creates a new container or reuses an active one from a previous code_interpreter_call in the model's context. The `code_interpreter_call` in the output of the APIwill contain the `container_id` that was generated. This container expires if it is not used for 20 minutes.

File inputs and outputs

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. It will cite these files in the annotations of its next message.

Any files in the model input get automatically uploaded to the container. You do not have to explicitly upload it to the container.

Supported Files

Expand table

File format	MIME type
`.c`	text/x-c
`.cs`	text/x-csharp
`.cpp`	text/x-c++

File format	MIME type
`.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

File format	MIME type
.png	image/png
.tar	application/x-tar
.xlsx	application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
.xml	application/xml or "text/xml"
.zip	application/zip

List input items

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.input_items.list("resp_67d856fcfba0819081fd3cffee2aa1c0")

print(response.model_dump_json(indent=2))
```

Output:

JSON

```json
{
  "data": [
    {
      "id": "msg_67d856fcfc1c8190ad3102fc01994c5f",
      "content": [
        {
          "text": "This is a test.",
          "type": "input_text"
        }
      ],
      "role": "user",
      "status": "completed",
      "type": "message"
    }
```

```
  ],
  "has_more": false,
  "object": "list",
  "first_id": "msg_67d856fcfc1c8190ad3102fc01994c5f",
  "last_id": "msg_67d856fcfc1c8190ad3102fc01994c5f"
}
```

Image input

Image url

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model="gpt-4o",
    input=[
        {
            "role": "user",
            "content": [
                { "type": "input_text", "text": "what is in this image?" },
                {
                    "type": "input_image",
                    "image_url": "<image_URL>"
                }
            ]
        }
    ]
)

print(response)
```

Base64 encoded image

Python

```python
import base64
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

def encode_image(image_path):
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode("utf-8")

# Path to your image
image_path = "path_to_your_image.jpg"

# Getting the Base64 string
base64_image = encode_image(image_path)

response = client.responses.create(
    model="gpt-4o",
    input=[
        {
            "role": "user",
            "content": [
                { "type": "input_text", "text": "what is in this image?" },
                {
                    "type": "input_image",
                    "image_url": f"{base64_image}"
                }
            ]
        }
    ]
)

print(response)
```

File input

Models with vision capabilities support PDF input. PDF files can be provided either as Base64-encoded data or as file IDs. To help models interpret PDF content, both the extracted text and an image of each page are included in the model's context. This is useful when key information is conveyed through diagrams or non-textual content.

> **Note**

- All extracted text and images are put into the model's context. Make sure you understand the pricing and token usage implications of using PDFs as input.
- You can upload up to 100 pages and 32MB of total content in a single request to the API, across multiple file inputs.
- Only models that support both text and image inputs, such as `gpt-4o`, `gpt-4o-mini`, or `o1`, can accept PDF files as input.
- A `purpose` of `user_data` is currently not supported. As a temporary workaround you will need to set purpose to `assistants`.

Convert PDF to Base64 and analyze

Python

```python
import base64
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

with open("PDF-FILE-NAME.pdf", "rb") as f: # assumes PDF is in the same directory as the executing script
    data = f.read()

base64_string = base64.b64encode(data).decode("utf-8")

response = client.responses.create(
    model="gpt-4o-mini", # model deployment name
    input=[
        {
            "role": "user",
            "content": [
                {
                    "type": "input_file",
                    "filename": "PDF-FILE-NAME.pdf",
                    "file_data": f"data:application/pdf;base64,{base64_string}",
                },
                {
                    "type": "input_text",
                    "text": "Summarize this PDF",
```

```
                },
            ],
        },
    ]
)

print(response.output_text)
```

Upload PDF and analyze

Upload the PDF file. A `purpose` of `user_data` is currently not supported. As a workaround you will need to set purpose to `assistants`.

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)


# Upload a file with a purpose of "assistants"
file = client.files.create(
  file=open("nucleus_sampling.pdf", "rb"), # This assumes a .pdf file in the same directory as the executing script
  purpose="assistants"
)

print(file.model_dump_json(indent=2))
file_id = file.id
```

Output:

```
{
  "id": "assistant-KaVLJQTiWEvdz8yJQHHkqJ",
  "bytes": 4691115,
  "created_at": 1752174469,
  "filename": "nucleus_sampling.pdf",
  "object": "file",
  "purpose": "assistants",
  "status": "processed",
```

```
  "expires_at": null,
  "status_details": null
}
```

You will then take the value of the `id` and pass that to a model for processing under `file_id`:

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model="gpt-4o-mini",
    input=[
        {
            "role": "user",
            "content": [
                {
                    "type": "input_file",
                    "file_id":"assistant-KaVLJQTiWEvdz8yJQHHkqJ"
                },
                {
                    "type": "input_text",
                    "text": "Summarize this PDF",
                },
            ],
        },
    ]
)

print(response.output_text)
```

Bash

```bash
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/files \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -F purpose="assistants" \
  -F file="@your_file.pdf" \

curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
```

```
        "model": "gpt-4.1",
        "input": [
            {
                "role": "user",
                "content": [
                    {
                        "type": "input_file",
                        "file_id": "assistant-123456789"
                    },
                    {
                        "type": "input_text",
                        "text": "ASK SOME QUESTION RELATED TO UPLOADED PDF"
                    }
                ]
            }
        ]
    }'
```

Using remote MCP servers

You can extend the capabilities of your model by connecting it to tools hosted on remote Model Context Protocol (MCP) servers. These servers are maintained by developers and organizations and expose tools that can be accessed by MCP-compatible clients, such as the Responses API.

Model Context Protocol (MCP) is an open standard that defines how applications provide tools and contextual data to large language models (LLMs). It enables consistent, scalable integration of external tools into model workflows.

The following example demonstrates how to use the fictitious MCP server to query information about the Azure REST API. This allows the model to retrieve and reason over repository content in real time.

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "github",
      "server_url": "https://contoso.com/Azure/azure-rest-api-specs",
      "require_approval": "never"
```

```
    }
  ],
  "input": "What is this repo in 100 words?"
}'
```

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)
response = client.responses.create(
    model="gpt-4.1", # replace with your model deployment name
    tools=[
        {
            "type": "mcp",
            "server_label": "github",
            "server_url": "https://contoso.com/Azure/azure-rest-api-specs",
            "require_approval": "never"
        },
    ],
    input="What transport protocols are supported in the 2025-03-26 version of the
MCP spec?",
)

print(response.output_text)
```

The MCP tool works only in the Responses API, and is available across all newer 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.

Approvals

By default, the Responses API requires explicit approval before any data is shared with a remote MCP server. This approval step helps ensure transparency and gives you control over what information is sent externally.

We recommend reviewing all data being shared with remote MCP servers and optionally logging it for auditing purposes.

When an approval is required, the model returns a `mcp_approval_request` item in the response output. This object contains the details of the pending request and allows you to inspect or

modify the data before proceeding.

JSON

```
{
  "id": "mcpr_682bd9cd428c8198b170dc6b549d66fc016e86a03f4cc828",
  "type": "mcp_approval_request",
  "arguments": {},
  "name": "fetch_azure_rest_api_docs",
  "server_label": "github"
}
```

To proceed with the remote MCP call, you must respond to the approval request by creating a new response object that includes an mcp_approval_response item. This object confirms your intent to allow the model to send the specified data to the remote MCP server.

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
  "model": "gpt-4.1",
  "tools": [
    {
      "type": "mcp",
      "server_label": "github",
      "server_url": "https://contoso.com/Azure/azure-rest-api-specs",
      "require_approval": "never"
    }
  ],
  "previous_response_id": "resp_682f750c5f9c8198aee5b480980b5cf60351aee697a7cd77",
  "input": [{
    "type": "mcp_approval_response",
    "approve": true,
    "approval_request_id": "mcpr_682bd9cd428c8198b170dc6b549d66fc016e86a03f4cc828"
  }]
}'
```

Python

```
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
```

```
)

response = client.responses.create(
    model="gpt-4.1", # replace with your model deployment name
    tools=[
        {
            "type": "mcp",
            "server_label": "github",
            "server_url": "https://contoso.com/Azure/azure-rest-api-specs",
            "require_approval": "never"
        },
    ],
    previous_response_id="resp_682f750c5f9c8198aee5b480980b5cf60351aee697a7cd77",
    input=[{
        "type": "mcp_approval_response",
        "approve": True,
        "approval_request_id":
"mcpr_682bd9cd428c8198b170dc6b549d66fc016e86a03f4cc828"
    }],
)
```

Authentication

Unlike the GitHub MCP server, most remote MCP servers require authentication. The MCP tool in the Responses API supports custom headers, allowing you to securely connect to these servers using the authentication scheme they require.

You can specify headers such as API keys, OAuth access tokens, or other credentials directly in your request. The most commonly used header is the `Authorization` header.

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
        "model": "gpt-4.1",
        "input": "What is this repo in 100 words?"
        "tools": [
            {
                "type": "mcp",
                "server_label": "github",
                "server_url": "https://contoso.com/Azure/azure-rest-api-specs",
                "headers": {
                    "Authorization": "Bearer $YOUR_API_KEY"
                }
```

```
        ]
    }'
```

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model="gpt-4.1",
    input="What is this repo in 100 words?",
    tools=[
        {
            "type": "mcp",
            "server_label": "github",
            "server_url": "https://gitmcp.io/Azure/azure-rest-api-specs",
            "headers": {
                "Authorization": "Bearer $YOUR_API_KEY"
        }
    ]
)

print(response.output_text)
```

Background tasks

Background mode allows you to run long-running tasks asynchronously using models like o3 and o1-pro. This is especially useful for complex reasoning tasks that can take several minutes to complete, such as those handled by agents like Codex or Deep Research.

By enabling background mode, you can avoid timeouts and maintain reliability during extended operations. When a request is sent with `"background": true`, the task is processed asynchronously, and you can poll for its status over time.

To start a background task, set the background parameter to true in your request:

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
```

```
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
    "model": "o3",
    "input": "Write me a very long story",
    "background": true
  }'
```

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.create(
    model = "o3",
    input = "Write me a very long story",
    background = True
)

print(response.status)
```

Use the `GET` endpoint to check the status of a background response. Continue polling while the status is queued or in_progress. Once the response reaches a final (terminal) state, it will be available for retrieval.

Bash

```
curl GET https://YOUR-RESOURCE-
NAME.openai.azure.com/openai/v1/responses/resp_1234567890 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN"
```

Python

```python
from time import sleep
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)
```

```
response = client.responses.create(
    model = "o3",
    input = "Write me a very long story",
    background = True
)

while response.status in {"queued", "in_progress"}:
    print(f"Current status: {response.status}")
    sleep(2)
    response = client.responses.retrieve(response.id)

print(f"Final status: {response.status}\nOutput:\n{response.output_text}")
```

You can cancel an in-progress background task using the `cancel` endpoint. Canceling is idempotent—subsequent calls will return the final response object.

Bash

```
curl -X POST https://YOUR-RESOURCE-
NAME.openai.azure.com/openai/v1/responses/resp_1234567890/cancel \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN"
```

Python

```
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

response = client.responses.cancel("resp_1234567890")

print(response.status)
```

Stream a background response

To stream a background response, set both `background` and `stream` to true. This is useful if you want to resume streaming later in case of a dropped connection. Use the sequence_number from each event to track your position.

Bash

```bash
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
    "model": "o3",
    "input": "Write me a very long story",
    "background": true,
    "stream": true
  }'
```

Python

```python
import os
from openai import OpenAI

client = OpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  api_key=os.getenv("AZURE_OPENAI_API_KEY")
)

# Fire off an async response but also start streaming immediately
stream = client.responses.create(
    model="o3",
    input="Write me a very long story",
    background=True,
    stream=True,
)

cursor = None
for event in stream:
    print(event)
    cursor = event["sequence_number"]
```

> **Note**
>
> Background responses currently have a higher time-to-first-token latency than synchronous responses. Improvements are underway to reduce this gap.

Limitations

- Background mode requires `store=true`. Stateless requests are not supported.

- You can only resume streaming if the original request included `stream=true`.
- To cancel a synchronous response, terminate the connection directly.

Resume streaming from a specific point

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses/resp_1234567890?
stream=true&starting_after=42 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN"
```

Encrypted Reasoning Items

When using the Responses API in stateless mode — either by setting `store` to false or when your organization is enrolled in zero data retention — you must still preserve reasoning context across conversation turns. To do this, include encrypted reasoning items in your API requests.

To retain reasoning items across turns, add `reasoning.encrypted_content` to the `include` parameter in your request. This ensures that the response includes an encrypted version of the reasoning trace, which can be passed along in future requests.

Bash

```
curl https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/responses \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AZURE_OPENAI_AUTH_TOKEN" \
  -d '{
    "model": "o4-mini",
    "reasoning": {"effort": "medium"},
    "input": "What is the weather like today?",
    "tools": [<YOUR_FUNCTION GOES HERE>],
    "include": ["reasoning.encrypted_content"]
  }'
```

Image generation (preview)

The Responses API enables image generation as part of conversations and multi-step workflows. It supports image inputs and outputs within context and includes built-in tools for generating and editing images.

Compared to the standalone Image API, the Responses API offers several advantages:

- **Streaming**: Display partial image outputs during generation to improve perceived latency.
- **Flexible inputs**: Accept image File IDs as inputs, in addition to raw image bytes.

> **Note**
>
> The image generation tool in the Responses API is only supported by the `gpt-image-1` model. You can however call this model from this list of supported models - `gpt-4o`, `gpt-4o-mini`, `gpt-4.1`, `gpt-4.1-mini`, `gpt-4.1-nano`, `o3`.

Use the Responses API if you want to:

- Build conversational image experiences with GPT Image.
- Stream partial image results during generation for a smoother user experience.

Generate an image

Python

```
from openai import AzureOpenAI
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

token_provider = get_bearer_token_provider(
    DefaultAzureCredential(), "https://cognitiveservices.azure.com/.default"
)

client = AzureOpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  azure_ad_token_provider=token_provider,
  api_version="preview",
  default_headers={"x-ms-oai-image-generation-deployment":"YOUR-GPT-IMAGE1-DEPLOYMENT-NAME"}
)

response = client.responses.create(
    model="o3",
    input="Generate an image of gray tabby cat hugging an otter with an orange scarf",
    tools=[{"type": "image_generation"}],
)

# Save the image to a file
image_data = [
    output.result
    for output in response.output
```

```
    if output.type == "image_generation_call"
]

if image_data:
    image_base64 = image_data[0]
    with open("otter.png", "wb") as f:
        f.write(base64.b64decode(image_base64))
```

Streaming

You can stream partial images using Responses API. The `partial_images` can be used to receive 1-3 partial images

Python

```
from openai import AzureOpenAI
from azure.identity import DefaultAzureCredential, get_bearer_token_provider

token_provider = get_bearer_token_provider(
    DefaultAzureCredential(), "https://cognitiveservices.azure.com/.default"
)

client = AzureOpenAI(
  base_url = "https://YOUR-RESOURCE-NAME.openai.azure.com/openai/v1/",
  azure_ad_token_provider=token_provider,
  api_version="preview",
  default_headers={"x-ms-oai-image-generation-deployment":"YOUR-GPT-IMAGE1-DEPLOYMENT-NAME"}
)

stream = client.responses.create(
    model="gpt-4.1",
    input="Draw a gorgeous image of a river made of white owl feathers, snaking its way through a serene winter landscape",
    stream=True,
    tools=[{"type": "image_generation", "partial_images": 2}],
)

for event in stream:
    if event.type == "response.image_generation_call.partial_image":
        idx = event.partial_image_index
        image_base64 = event.partial_image_b64
        image_bytes = base64.b64decode(image_base64)
        with open(f"river{idx}.png", "wb") as f:
            f.write(image_bytes)
```

Reasoning models

For examples of how to use reasoning models with the responses API see the reasoning models guide.

Computer use

Computer use with Playwright has moved to the dedicated computer use model guide