

# Assistants API deep dive Beta

[Copy page](#)


In-depth guide to creating and managing assistants.

As described in the [Assistants Overview](#), there are several concepts involved in building an app with the Assistants API.

This guide goes deeper into each of these concepts.

If you want to get started coding right away, check out the [Assistants API Quickstart](#).

## Creating Assistants

 We recommend using OpenAI's [latest models](#) with the Assistants API for best results and maximum compatibility with tools.

To get started, creating an Assistant only requires specifying the `model` to use. But you can further customize the behavior of the Assistant:

- 1 Use the `instructions` parameter to guide the personality of the Assistant and define its goals. Instructions are similar to system messages in the Chat Completions API.
- 2 Use the `tools` parameter to give the Assistant access to up to 128 tools. You can give it access to OpenAI-hosted tools like `code_interpreter` and `file_search`, or call a third-party tools via a `function` calling.
- 3 Use the `tool_resources` parameter to give the tools like `code_interpreter` and `file_search` access to files. Files are uploaded using the `File` [upload endpoint](#) and must have the `purpose` set to `assistants` to be used with this API.

For example, to create an Assistant that can create data visualization based on a `.csv` file, first upload a file.

python 

```
1 file = client.files.create(  
2     file=open("revenue-forecast.csv", "rb"),  
3     purpose='assistants'  
4 )
```

Then, create the Assistant with the `code_interpreter` tool enabled and provide the file as a resource to the tool.

```
1 assistant = client.beta.assistants.create(  
2     name="Data visualizer",  
3     description="You are great at creating beautiful data visualizations. You analyze data pro  
4     model="gpt-4o",  
5     tools=[{"type": "code_interpreter"}],  
6     tool_resources={  
7         "code_interpreter": {  
8             "file_ids": [file.id]  
9         }  
10    }  
11 )
```

You can attach a maximum of 20 files to `code_interpreter` and 10,000 files to `file_search` (using `vector_store` [objects](#)).

Each file can be at most 512 MB in size and have a maximum of 5,000,000 tokens. By default, the size of all the files uploaded in your project cannot exceed 100 GB, but you can reach out to our support team to increase this limit.

## Managing Threads and Messages

Threads and Messages represent a conversation session between an Assistant and a user. There is a limit of 100,000 Messages per Thread. Once the size of the Messages exceeds the context window of the model, the Thread will attempt to smartly truncate messages, before fully dropping the ones it considers the least important.

You can create a Thread with an initial list of Messages like this:

```
1 thread = client.beta.threads.create(  
2     messages=[  
3         {  
4             "role": "user",  
5             "content": "Create 3 data visualizations based on the trends in this file.",  
6             "attachments": [  
7                 {  
8                     "file_id": file.id,  
9                     "tools": [{"type": "code_interpreter"}]  
10                }  
11            ]  
12        }  
13    ]  
14 )
```

Messages can contain text, images, or file attachment. Message `attachments` are helper methods that add files to a thread's `tool_resources`. You can also choose to add files to the

`thread.tool_resources` directly.

## Creating image input content

Message content can contain either external image URLs or File IDs uploaded via the [File API](#). Only [models](#) with Vision support can accept image input. Supported image content types include png, jpg, gif, and webp. When creating image files, pass `purpose="vision"` to allow you to later download and display the input content. Currently, there is a 100GB limit per project. Please contact us to request a limit increase.

Tools cannot access image content unless specified. To pass image files to Code Interpreter, add the file ID in the message `attachments` list to allow the tool to read and analyze the input. Image URLs cannot be downloaded in Code Interpreter today.

python ↕

```
1 file = client.files.create(
2     file=open("myimage.png", "rb"),
3     purpose="vision"
4 )
5 thread = client.beta.threads.create(
6     messages=[
7         {
8             "role": "user",
9             "content": [
10                {
11                    "type": "text",
12                    "text": "What is the difference between these images?"
13                },
14                {
15                    "type": "image_url",
16                    "image_url": {"url": "https://example.com/image.png"}
17                },
18                {
19                    "type": "image_file",
20                    "image_file": {"file_id": file.id}
21                },
22            ],
23        }
24    ]
25 )
```

## Low or high fidelity image understanding

By controlling the `detail` parameter, which has three options, `low`, `high`, or `auto`, you have control over how the model processes the image and generates its textual understanding.

`low` will enable the "low res" mode. The model will receive a low-res 512px x 512px version of the image, and represent the image with a budget of 85 tokens. This allows the API to

return faster responses and consume fewer input tokens for use cases that do not require high detail.

`high` will enable "high res" mode, which first allows the model to see the low res image and then creates detailed crops of input images based on the input image size. Use the [pricing calculator](#) to see token counts for various image sizes.

python 

```
1 thread = client.beta.threads.create(  
2     messages=[  
3         {  
4             "role": "user",  
5             "content": [  
6                 {  
7                     "type": "text",  
8                     "text": "What is this an image of?"  
9                 },  
10                {  
11                    "type": "image_url",  
12                    "image_url": {  
13                        "url": "https://example.com/image.png",  
14                        "detail": "high"  
15                    }  
16                },  
17            ],  
18        }  
19    ]  
20 )
```

## Context window management

The Assistants API automatically manages the truncation to ensure it stays within the model's maximum context length. You can customize this behavior by specifying the maximum tokens you'd like a run to utilize and/or the maximum number of recent messages you'd like to include in a run.

### Max Completion and Max Prompt Tokens

To control the token usage in a single Run, set `max_prompt_tokens` and `max_completion_tokens` when creating the Run. These limits apply to the total number of tokens used in all completions throughout the Run's lifecycle.

For example, initiating a Run with `max_prompt_tokens` set to 500 and `max_completion_tokens` set to 1000 means the first completion will truncate the thread to 500 tokens and cap the output at 1000 tokens. If only 200 prompt tokens and 300 completion tokens are used in the first completion, the second completion will have available limits of 300 prompt tokens and 700 completion tokens.

If a completion reaches the `max_completion_tokens` limit, the Run will terminate with a status of `incomplete`, and details will be provided in the `incomplete_details` field of the Run object.

❗ When using the File Search tool, we recommend setting the `max_prompt_tokens` to no less than 20,000. For longer conversations or multiple interactions with File Search, consider increasing this limit to 50,000, or ideally, removing the `max_prompt_tokens` limits altogether to get the highest quality results.

## Truncation Strategy

You may also specify a truncation strategy to control how your thread should be rendered into the model's context window. Using a truncation strategy of type `auto` will use OpenAI's default truncation strategy. Using a truncation strategy of type `last_messages` will allow you to specify the number of the most recent messages to include in the context window.

## Message annotations

Messages created by Assistants may contain `annotations` within the `content` array of the object. Annotations provide information around how you should annotate the text in the Message.

There are two types of Annotations:

- `file_citation`: File citations are created by the `file_search` tool and define references to a specific file that was uploaded and used by the Assistant to generate the response.
- `file_path`: File path annotations are created by the `code_interpreter` tool and contain references to the files generated by the tool.

When annotations are present in the Message object, you'll see illegible model-generated substrings in the text that you should replace with the annotations. These strings may look something like `【13†source】` or `sandbox:/mnt/data/file.csv`. Here's an example python code snippet that replaces these strings with the annotations.

python 

```
1 # Retrieve the message object
2 message = client.beta.threads.messages.retrieve(
3     thread_id="...",
4     message_id="..."
5 )
6
7 # Extract the message content
8 message_content = message.content[0].text
9 annotations = message_content.annotations
10 citations = []
11
12 # Iterate over the annotations and add footnotes
13 for index, annotation in enumerate(annotations):
```

```

# Replace the text with a footnote
message_content.value = message_content.value.replace(annotation.text, f' [{index}]')

# Gather citations based on annotation attributes
if (file_citation := getattr(annotation, 'file_citation', None)):
    cited_file = client.files.retrieve(file_citation.file_id)
    citations.append(f'[{index}] {file_citation.quote} from {cited_file.filename}')
elif (file_path := getattr(annotation, 'file_path', None)):
    cited_file = client.files.retrieve(file_path.file_id)
    citations.append(f'[{index}] Click <here> to download {cited_file.filename}')
# Note: File download functionality not implemented above for brevity

# Add footnotes to the end of the message before displaying to user
message_content.value += '\n' + '\n'.join(citations)

```

## Runs and Run Steps

When you have all the context you need from your user in the Thread, you can run the Thread with an Assistant of your choice.

python 

```

1 run = client.beta.threads.runs.create(
2     thread_id=thread.id,
3     assistant_id=assistant.id
4 )

```

By default, a Run will use the `model` and `tools` configuration specified in Assistant object, but you can override most of these when creating the Run for added flexibility:

python 

```

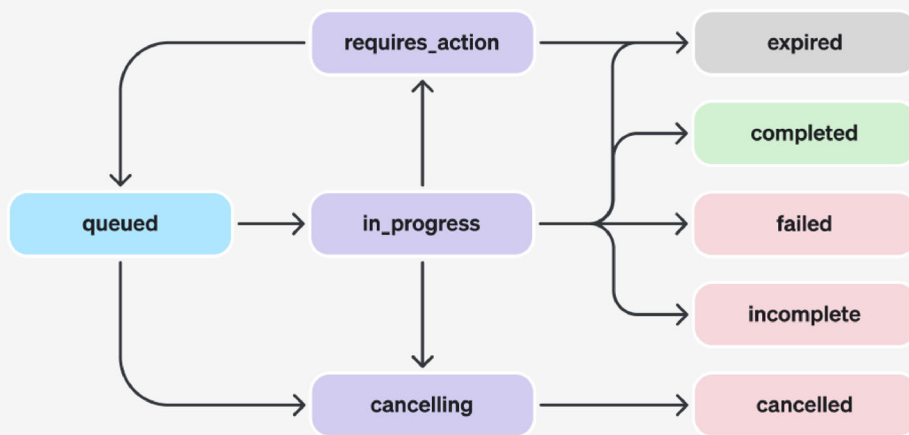
1 run = client.beta.threads.runs.create(
2     thread_id=thread.id,
3     assistant_id=assistant.id,
4     model="gpt-4o",
5     instructions="New instructions that override the Assistant instructions",
6     tools=[{"type": "code_interpreter"}, {"type": "file_search"}]
7 )

```

Note: `tool_resources` associated with the Assistant cannot be overridden during Run creation. You must use the [modify Assistant](#) endpoint to do this.

## Run lifecycle

Run objects can have multiple statuses.



STATUS	DEFINITION
queued	When Runs are first created or when you complete the <code>required_action</code> , they are moved to a <code>queued</code> status. They should almost immediately move to <code>in_progress</code> .
in_progress	While <code>in_progress</code> , the Assistant uses the model and tools to perform steps. You can view progress being made by the Run by examining the <a href="#">Run Steps</a> .
completed	The Run successfully completed! You can now view all Messages the Assistant added to the Thread, and all the steps the Run took. You can also continue the conversation by adding more user Messages to the Thread and creating another Run.
requires_action	When using the <a href="#">Function calling</a> tool, the Run will move to a <code>required_action</code> state once the model determines the names and arguments of the functions to be called. You must then run those functions and <a href="#">submit the outputs</a> before the run proceeds. If the outputs are not provided before the <code>expires_at</code> timestamp passes (roughly 10 mins past creation), the run will move to an <code>expired</code> status.
expired	This happens when the function calling outputs were not submitted before <code>expires_at</code> and the run expires. Additionally, if the runs take too long to execute and go beyond the time stated in <code>expires_at</code> , our systems will expire the run.
cancelling	You can attempt to cancel an <code>in_progress</code> run using the <a href="#">Cancel Run</a> endpoint. Once the attempt to cancel succeeds, status of the Run moves to <code>cancelled</code> . Cancellation is attempted but not guaranteed.
cancelled	Run was successfully cancelled.
failed	You can view the reason for the failure by looking at the <code>last_error</code> object in the Run. The timestamp for the failure will be recorded under <code>failed_at</code> .
incomplete	Run ended due to <code>max_prompt_tokens</code> or <code>max_completion_tokens</code> reached. You can view the specific reason by looking at the <code>incomplete_details</code> object in the Run.

## Polling for updates

If you are not using [streaming](#), in order to keep the status of your run up to date, you will have to periodically [retrieve the Run](#) object. You can check the status of the run each time you retrieve the object to determine what your application should do next.

You can optionally use Polling Helpers in our [Node](#) and [Python](#) SDKs to help you with this. These helpers will automatically poll the Run object for you and return the Run object when it's in a terminal state.

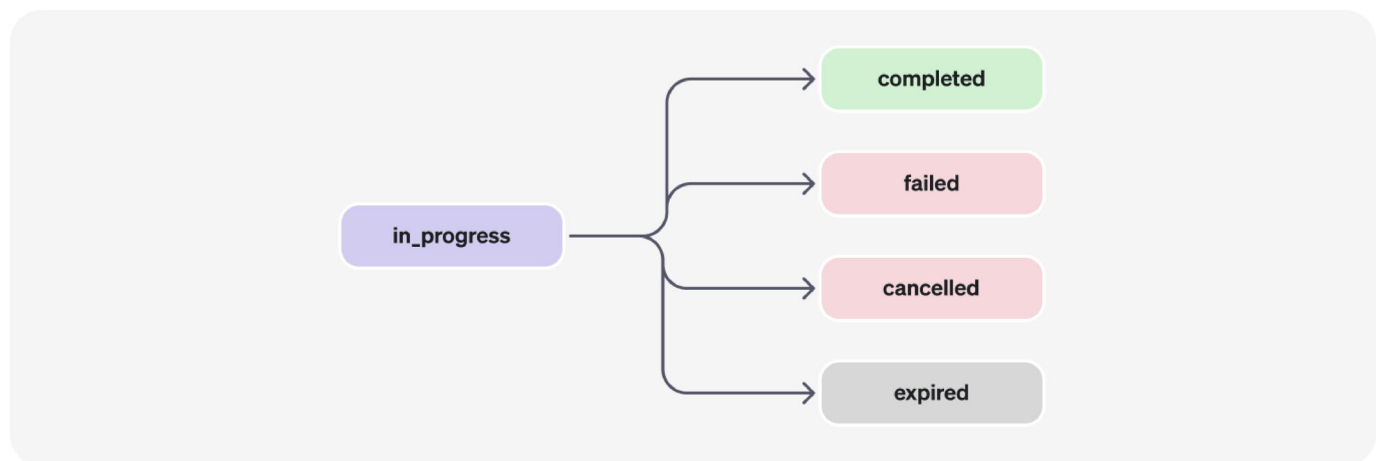
## Thread locks

When a Run is `in_progress` and not in a terminal state, the Thread is locked. This means that:

New Messages cannot be added to the Thread.

New Runs cannot be created on the Thread.

## Run steps



Run step statuses have the same meaning as Run statuses.

Most of the interesting detail in the Run Step object lives in the `step_details` field. There can be two types of step details:

- 1 `message_creation` : This Run Step is created when the Assistant creates a Message on the Thread.
- 2 `tool_calls` : This Run Step is created when the Assistant calls a tool. Details around this are covered in the relevant sections of the [Tools](#) guide.

## Data Access Guidance

Currently, Assistants, Threads, Messages, and Vector Stores created via the API are scoped to the Project they're created in. As such, any person with API key access to that Project is able to read or write Assistants, Threads, Messages, and Runs in the Project.

We strongly recommend the following data access controls:

*Implement authorization.* Before performing reads or writes on Assistants, Threads, Messages, and Vector Stores, ensure that the end-user is authorized to do so. For example, store in your database the object IDs that the end-user has access to, and check it before fetching the object ID with the API.



*Restrict API key access.* Carefully consider who in your organization should have API keys and be part of a Project. Periodically audit this list. API keys enable a wide range of operations including reading and modifying sensitive information, such as Messages and Files.

*Create separate accounts.* Consider creating separate Projects for different applications in order to isolate data across multiple applications.

