

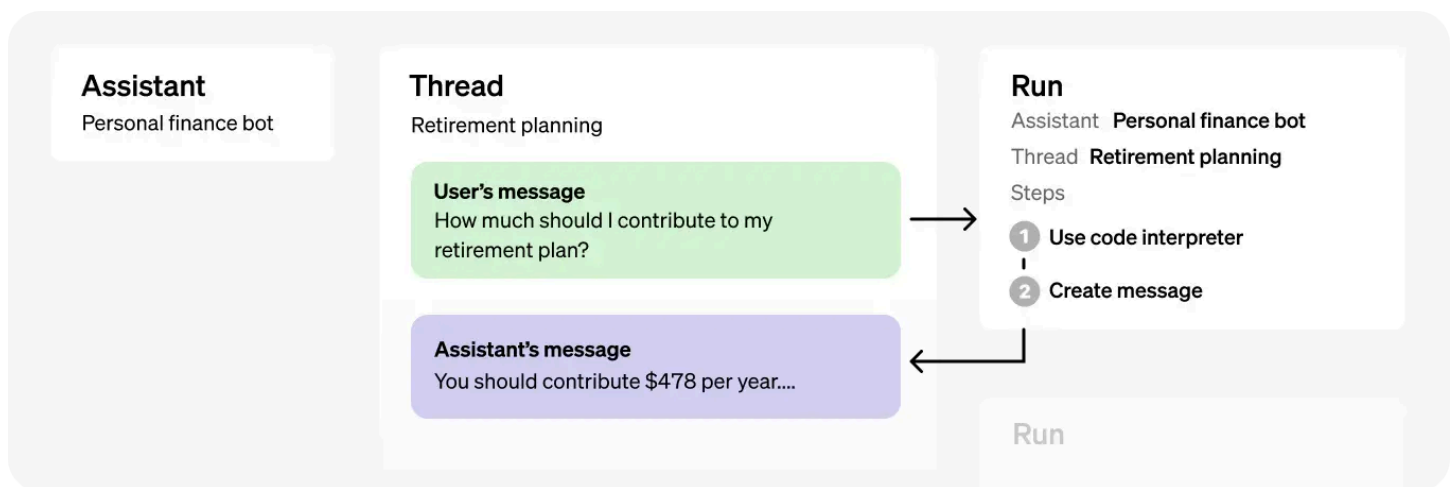
How Assistants work Beta

The Assistants API is designed to help developers build powerful AI assistants capable of performing a variety of tasks.

The Assistants API is in **beta** and we are actively working on adding more functionality. Share your feedback in our [Developer Forum](#)!

- 1 Assistants can call OpenAI's **models** with specific instructions to tune their personality and capabilities.
- 2 Assistants can access **multiple tools in parallel**. These can be both OpenAI-hosted tools — like [code_interpreter](#) and [file_search](#) — or tools you build / host (via [function calling](#)).
- 3 Assistants can access **persistent Threads**. Threads simplify AI application development by storing message history and truncating it when the conversation gets too long for the model's context length. You create a Thread once, and simply append Messages to it as your users reply.
- 4 Assistants can access files in several formats — either as part of their creation or as part of Threads between Assistants and users. When using tools, Assistants can also create files (e.g., images, spreadsheets, etc) and cite files they reference in the Messages they create.


Objects



OBJECT	WHAT IT REPRESENTS
Assistant	Purpose-built AI that uses OpenAI's models and calls tools
Thread	A conversation session between an Assistant and a user. Threads store Messages and automatically handle truncation to fit content into a model's context.
Message	A message created by an Assistant or a user. Messages can include text, images, and other files. Messages stored as a list on the Thread.

OBJECT	WHAT IT REPRESENTS
Run	An invocation of an Assistant on a Thread. The Assistant uses its configuration and the Thread's Messages to perform tasks by calling models and tools. As part of a Run, the Assistant appends Messages to the Thread.
Run Step	A detailed list of steps the Assistant took as part of a Run. An Assistant can call tools or create Messages during its run. Examining Run Steps allows you to introspect how the Assistant is getting to its final results.


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.

python ▾



```
1 assistant = client.beta.assistants.create(  
2     name="Data visualizer",  
3     description="You are great at creating beautiful data visualizations. You analyze dat  
4     model="gpt-4-turbo",  
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 by your organization 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 no limit to the number of Messages you can store in a 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:

python ▾



```
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 )
```

```
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. At the moment, user-created Messages cannot contain image files but we plan to add support for this in the future.

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:

- 1 `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.
- 2 `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 information present in 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):  
14     # Replace the text with a footnote  
15     message_content.value = message_content.value.replace(annotation.text, f' 【{index}】')  
16  
17     # Gather citations based on annotation attributes  
18     if (file_citation := getattr(annotation, 'file_citation', None)):  
19         cited_file = client.files.retrieve(file_citation.file_id)  
20         citations.append(f' 【{index}】 {file_citation.quote} from {cited_file.filename}')  
21     elif (file_path := getattr(annotation, 'file_path', None)):  
22         cited_file = client.files.retrieve(file_path.file_id)  
23         citations.append(f' 【{index}】 Click <here> to download {cited_file.filename}')  
24     # Note: File download functionality not implemented above for brevity  
25
```


```
26 # Add footnotes to the end of the message before displaying to user
27 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 )
```

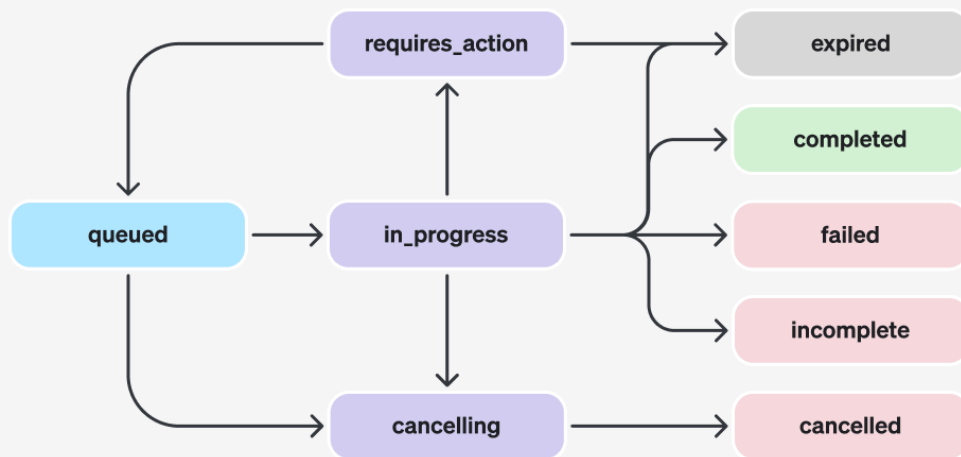
By default, a Run will use the `model` and `tools` configuration specified in Assistant object, but you can override most of these when creating the Run for added flexibility:

```
python ▾ 
1 run = client.beta.threads.runs.create(
2     thread_id=thread.id,
3     assistant_id=assistant.id,
4     model="gpt-4-turbo",
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 required_action, they are moved to a queued status. They should almost immediately move to in_progress.
in_progress	While in_progress, the Assistant uses the model and tools to perform steps. You can view progress being made by the Run by examining the Run Steps .
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 Function calling tool, the Run will move to a required_action state once the model determines the names and arguments of the functions to be called. You must then run those functions and submit the outputs before the run proceeds. If the outputs are not provided before the expires_at timestamp passes (roughly 10 mins past creation), the run will move to an expired status.
expired	This happens when the function calling outputs were not submitted before expires_at and the run expires. Additionally, if the runs take too long to execute and go beyond the time stated in expires_at, our systems will expire the run.
cancelling	You can attempt to cancel an in_progress run using the Cancel Run endpoint. Once the attempt to cancel succeeds, status of the Run moves to cancelled. Cancellation is attempted but not guaranteed.
cancelled	Run was successfully cancelled.
failed	You can view the reason for the failure by looking at the last_error object in the Run. The timestamp for the failure will be recorded under failed_at.

STATUS	DEFINITION
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.

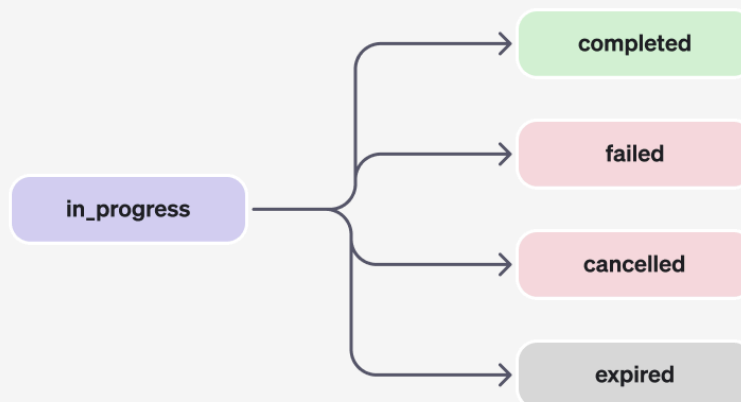
You can optionally use Polling Helpers in our [Node](#) and [Python](#) SDKs to help you with this. These helpers will automatically poll the Run object for you and return the Run object when it's in a terminal state.

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.

Next

Now that you have explored how Assistants work, the next step is to explore [Assistant Tools](#) which covers topics like Function calling, File Search, and Code Interpreter.