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!

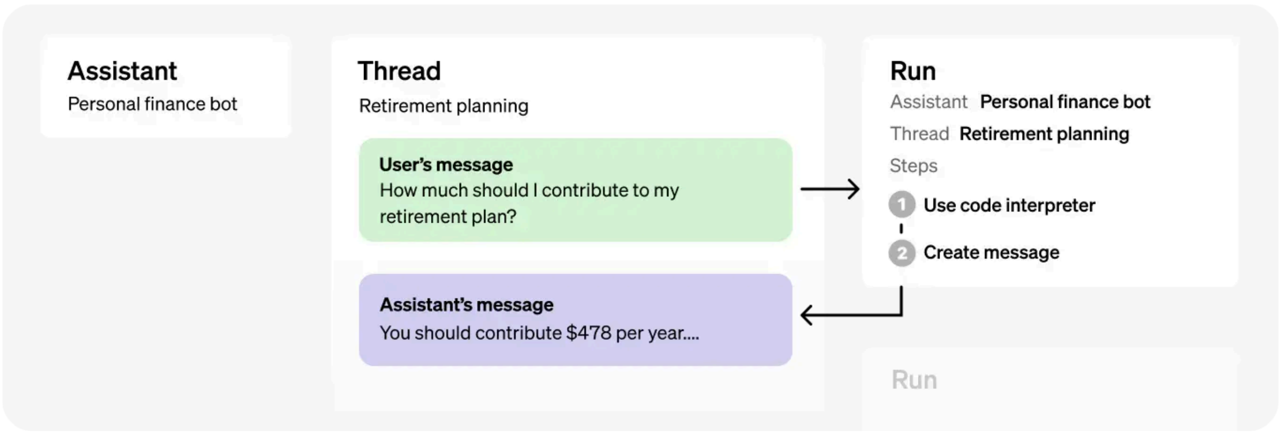
1 Assistants can call OpenAI's **models** with specific instructions to tune their personality and capabilities.

2 Assistants can access **multiple tools in parallel.** These can be both OpenAI-hosted tools — like Code interpreter and Knowledge retrieval — 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


Assistant
Personal finance bot
Thread
Retirement planning
User's message
How much should I contribute to my retirement plan?
Assistant's message
You should contribute $478 per year....
Run
Assistant Personal finance bot
Thread Retirement planning
Steps
1 Use code interpreter
2 Create message
Run

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.
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 `retrieval`, or call a third-party tools via a `function` calling.
3. Use the `file_ids` parameter to give the tools like `code_interpreter` and `retrieval` 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.

node.js

```
const file = await openai.files.create({
  file: fs.createReadStream("mydata.csv"),
  purpose: "assistants",
});
```

And then create the Assistant with the uploaded file.

node.js

```
const assistant = await openai.beta.assistants.create({
  name: "Data visualizer",
  description: "You are great at creating beautiful data visualizations. You
  model: "gpt-4-turbo-preview",
  tools: [{"type": "code_interpreter"}],
  file_ids: [file.id]
});
```

You can attach a maximum of 20 files per Assistant, and they can be at most 512 MB each. The size of all the files uploaded by your organization should not exceed 100 GB. You can request an increase in this storage limit using our help center. In addition to the 512 MB file size limit, each file can only contain 2,000,000 tokens. Assistant or Message creation will fail if any attached files exceed the token limit.

You can also use the `AssistantFile` object to create, delete, or view associations between Assistant and File objects. Note that deleting an `AssistantFile` doesn't delete the original File object, it simply deletes the association between that File and the Assistant. To delete a File, use the File delete endpoint instead.

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 include as many messages as possible that fit in the context window and drop the oldest messages.

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

node.js

```
const thread = await openai.beta.threads.create({
  messages: [
    {
      "role": "user",
      "content": "Create 3 data visualizations based on the trends in this fi
      "file_ids": [file.id]
    }
  ]
});
```

Messages can contain text, images, or files. At the moment, user-created Messages cannot contain image files but we plan to add support for this in the future. Messages also have the same file size and token limits as Assistants (512 MB file size limit and 2,000,000 token limit).

Context window management

The Assistants API automatically manages the context window such that you never exceed the model's context length. Once the size of the Messages in a Thread exceeds the context window of the model, the Thread will attempt to include as many messages as possible that fit in the context window and drop the oldest messages. Note that this truncation strategy will evolve over time to become more sophisticated.

Currently, the Assistant will include the maximum number of messages that fit in the context length. We plan to explore the ability for you to control the input / output token count beyond the model you select, as well as the ability to automatically generate summaries of the previous messages and pass that as context. If your use case requires a more advanced level of control, you can manually generate summaries and control context with the Chat Completion API.

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 `retrieval` tool and define references to a specific quote in 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
# Retrieve the message object
message = client.beta.threads.messages.retrieve(
  thread_id="...",
  message_id="..."
)

# Extract the message content
message_content = message.content[0].text
annotations = message_content.annotations
citations = []

# Iterate over the annotations and add footnotes
for index, annotation in enumerate(annotations):
    # Replace the text with a footnote
    message_content.value = message_content.value.replace(annotation.text, f

    # 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.
    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.fi
        # Note: File download functionality not implemented above for brevit

# 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.

node.js

```
const run = await openai.beta.threads.runs.create(
  thread.id,
  { assistant_id: assistant.id }
);
```

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:

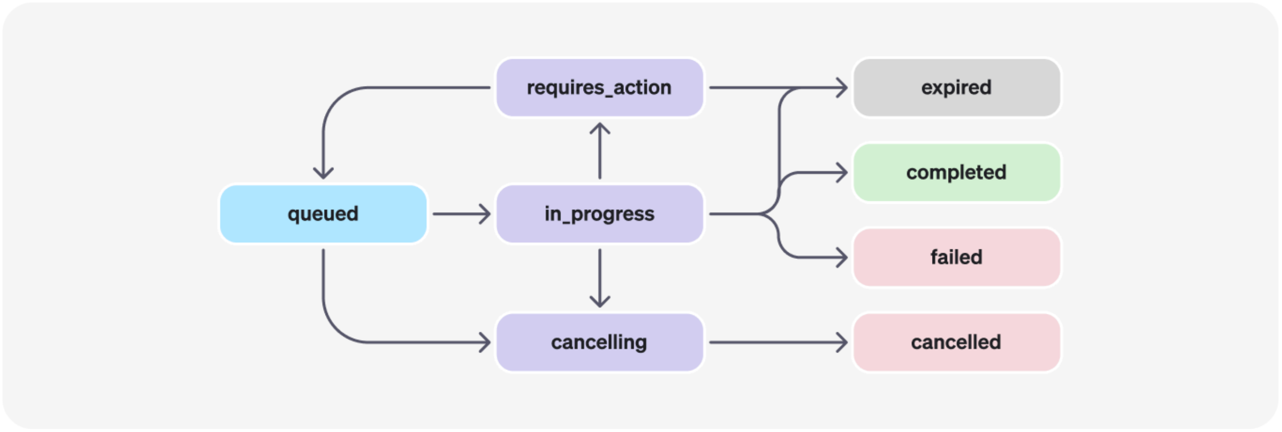
node.js

```
const run = await openai.beta.threads.runs.create(
  thread.id,
  {
    assistant_id: assistant.id,
    model: "gpt-4-turbo-preview",
    instructions: "New instructions that override the Assistant instructions"
    tools: [{"type": "code_interpreter"}, {"type": "retrieval"}]
  }
);
```

Note: `file_ids` 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`.

STATUS	DEFINITION
`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`.

Polling for updates

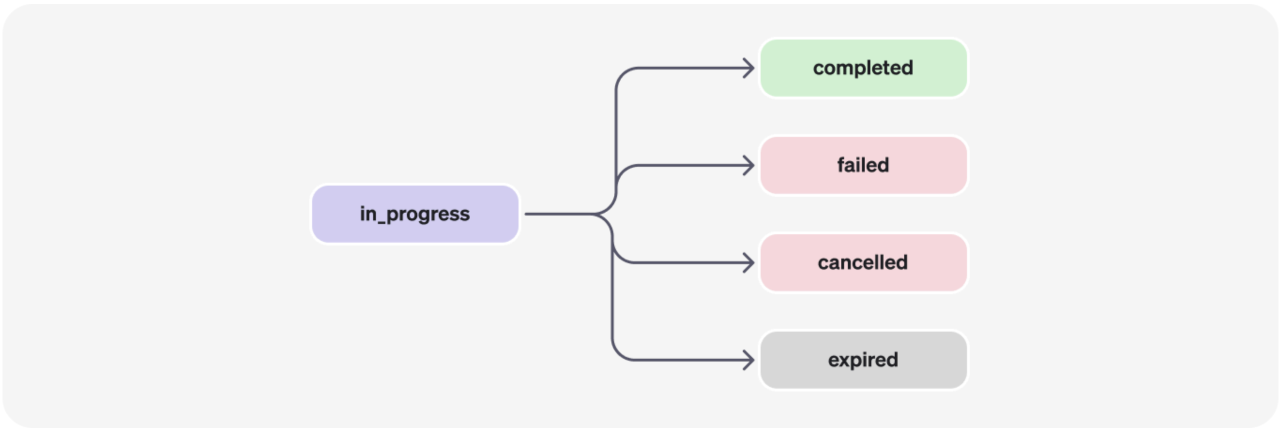
In order to keep the status of your run up to date, you will have to periodically retrieve the Run object. You can check the status of the run each time you retrieve the object to determine what your application should do next. We plan to add support for streaming to make this simpler in the near future.

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 files created via the API are scoped to the entire organization. As such, any person with API key access to the organization is able to read or write assistants, threads, messages, and files in the organization.

We strongly recommend the following data access controls:

- *Implement authorization.* Before performing reads or writes on assistants, threads, messages, and files, 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 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 accounts / organizations for different applications in order to isolate data across multiple applications.

Limitations

During this beta, there are several known limitations we are looking to address:

- Support for streaming output (including Messages and Run Steps).
- Support for notifications to share object status updates without the need for polling.
- Support for DALL·E or Browsing as a tool.
- Support for user message creation with images.
- 60 req/min limit at the user account level.

We are actively working to add these features and welcome feedback on our Developer Forum as to what else would make building an Assistant more powerful.

Next