Assistants API Beta

The Assistants API allows you to build AI assistants within your own applications. An Assistant has instructions and can leverage models, tools, and knowledge to respond to user queries. The Assistants API currently supports three types of tools: Code Interpreter, Retrieval, and Function calling. In the future, we plan to release more OpenAI-built tools, and allow you to provide your own tools on our platform.

You can explore the capabilities of the Assistants API using the Assistants playground or by building a step-by-step integration outlined in this guide. At a high level, a typical integration of the Assistants API has the following flow:

1. Create an Assistant in the API by defining its custom instructions and picking a model. If helpful, enable tools like Code Interpreter, Retrieval, and Function calling.
2. Create a Thread when a user starts a conversation.
3. Add Messages to the Thread as the user ask questions.
4. Run the Assistant on the Thread to trigger responses. This automatically calls the relevant tools.

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

Calls to the Assistants API require that you pass a beta HTTP header. This is handled automatically if you're using OpenAI's official Python or Node.js SDKs.

```
OpenAI-Beta: assistants=v1
```

This starter guide walks through the key steps to create and run an Assistant that uses Code Interpreter.

Assistants playground

In addition to the Assistants API, we also provide an Assistants playground (sign in required). The playground is a great way to explore the capabilities of the Assistants API and learn how to build your own Assistant without writing any code.

Step 1: Create an Assistant

An Assistant represents an entity that can be configured to respond to users' Messages using several parameters like:

- Instructions: how the Assistant and model should behave or respond
- Model: you can specify any GPT-3.5 or GPT-4 models. The Retrieval tool requires at least `gpt-3.5-turbo-1106` (newer versions are supported) or `gpt-4-turbo-preview` models.
  - Note: Support for fine-tuned models in the Assistants API is coming soon
- Tools: the API supports Code Interpreter and Retrieval that are built and hosted by OpenAI.
- Functions: the API allows you to define custom function signatures, with similar behavior as our function calling feature.

In this example, we're creating an Assistant that is a personal math tutor, with the Code Interpreter tool enabled.

node.js

```
const assistant = await openai.beta.assistants.create({
  name: "Math Tutor",
  instructions: "You are a personal math tutor. Write and run code to answer
  tools: [{ type: "code_interpreter" }],
  model: "gpt-4-turbo-preview"
});
```

Step 2: Create a Thread

A Thread represents a conversation. We recommend creating one Thread per user as soon as the user initiates the conversation. Pass any user-specific context and files in this thread by creating Messages.

node.js

```
const thread = await openai.beta.threads.create();
```

Threads don't have a size limit. You can add as many Messages as you want to a Thread. The Assistant will ensure that requests to the model fit within the maximum context window, using relevant optimization techniques such as truncation which we have tested extensively with ChatGPT. When you use the Assistants API, you delegate control over how many input tokens are passed to the model for any given Run, this means you have less control over the cost of running your Assistant in some cases but do not have to deal with the complexity of managing the context window yourself.

Organizations that have enabled the Threads page can view Threads created through the Assistants API and Assistants playground. Threads page permissions can be managed in Organization settings.

Step 3: Add a Message to a Thread

A Message contains text, and optionally any files that you allow the user to upload. Messages need to be added to a specific Thread. Adding images via message objects like in Chat Completions using GPT-4 with Vision is not supported today, but we plan to add support for them in the coming months. You can still upload images and have them processes via retrieval.

node.js

```
const message = await openai.beta.threads.messages.create(
  thread.id,
  {
    role: "user",
    content: "I need to solve the equation `3x + 11 = 14`. Can you help me?"
  }
);
```

Now if you list the Messages in a Thread, you will see that this message has been appended.

```
{
  "object": "list",
  "data": [
    {
      "created_at": 1696995451,
      "id": "msg_abc123",
      "object": "thread.message",
      "thread_id": "thread_abc123",
      "role": "user",
```

```
      "content": [{
        "type": "text",
        "text": {
          "value": "I need to solve the equation `3x + 11 = 14`. Can you hel
          "annotations": []
        }
          }],
        ...
```

Step 4: Run the Assistant

For the Assistant to respond to the user message, you need to create a Run. This makes the Assistant read the Thread and decide whether to call tools (if they are enabled) or simply use the model to best answer the query. As the run progresses, the assistant appends Messages to the thread with the `role="assistant"`. The Assistant will also automatically decide what previous Messages to include in the context window for the model. This has both an impact on pricing as well as model performance. The current approach has been optimized based on what we learned building ChatGPT and will likely evolve over time.

You can optionally pass new instructions to the Assistant while creating the Run but note that these instructions override the default instructions of the Assistant.

node.js

```
const run = await openai.beta.threads.runs.create(
  thread.id,
  {
    assistant_id: assistant.id,
    instructions: "Please address the user as Jane Doe. The user has a premiu
  }
);
```

Step 5: Check the Run status

By default, a Run goes into the `queued` state. You can periodically retrieve the Run to check on its status to see if it has moved to `completed`.

node.js

```
const run = await openai.beta.threads.runs.retrieve(
  thread.id,
```

```
  run.id
);
```

Step 6: Display the Assistant's Response

Once the Run completes, you can list the Messages added to the Thread by the Assistant.

node.js

```
const messages = await openai.beta.threads.messages.list(
  thread.id
);
```

And finally, display them to the user! During this Run, the Assistant added two new Messages to the Thread. Here is an example of what that might look like:

ROLE	CONTENT
`user`	I need to solve the equation `3x + 11 = 14`. Can you help me?
`assistant`	Certainly, Jane Doe. To solve the equation `(3x + 11 = 14)` for `(x)`, you'll want to isolate `(x)` on one side of the equation. Here's how you can do that: 1 Subtract 11 from both sides of the equation to get `(3x = 3)`. 2 Then, divide both sides by 3 to solve for `(x)`. Let me calculate the value of `(x)` for you.
`assistant`	The solution to the equation `(3x + 11 = 14)` is `(x = 1)`.

You can also retrieve the Run Steps of this Run if you'd like to explore or display the inner workings of the Assistant and its tools.

Next

1. Dive deeper into How Assistants work
2. Learn more about Tools

3 Explore the Assistants playground

Was this page useful?