Streaming is now available in the Assistants API.

Learn more Dismiss

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.

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.

Overview

A typical integration of the Assistants API has the following flow:

1. Create an Assistant by defining its custom instructions and picking a model. If helpful, add files and 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 asks questions.
4. Run the Assistant on the Thread to generate a response by calling the model and the tools.

This starter guide walks through the key steps to create and run an Assistant that uses Code Interpreter. In this example, we're creating an Assistant that is a personal math tutor, with the Code Interpreter tool enabled.

> 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
> ```

Step 1: Create an Assistant

An Assistant represents an entity that can be configured to respond to a user's messages using several parameters like `model`, `instructions`, and `tools`.

Create an Assistant python

```python
from openai import OpenAI
client = OpenAI()

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

Step 2: Create a Thread

A Thread represents a conversation between a user and one or many Assistants. You can create a Thread when a user (or your AI application) starts a conversation with your Assistant.

Create a Thread python

```python
thread = client.beta.threads.create()
```

Step 3: Add a Message to the Thread

The contents of the messages your users or applications create are added as Message objects to the Thread. Messages can contain both text and files. There is no limit to the number of Messages you can add to Threads — we smartly truncate any context that does not fit into the model's context window.

Add a Message to the Thread python

```python
message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="I need to solve the equation `3x + 11 = 14`. Can you help
)
```

Step 4: Create a Run

Once all the user Messages have been added to the Thread, you can Run the Thread with any Assistant. Creating a Run uses the model and tools associated with the Assistant to generate a response. These responses are added to the Thread as `assistant` Messages.

With streaming **Without streaming**

You can use the 'create and stream' helpers in the Python and Node SDKs to create a run and stream the response.

Create and Stream a Run **python**

```python
from typing_extensions import override
from openai import AssistantEventHandler

# First, we create a EventHandler class to define
# how we want to handle the events in the response stream.

class EventHandler(AssistantEventHandler):
  @override
  def on_text_created(self, text) -> None:
    print(f"\nassistant > ", end="", flush=True)

  @override
  def on_text_delta(self, delta, snapshot):
    print(delta.value, end="", flush=True)

  def on_tool_call_created(self, tool_call):
    print(f"\nassistant > {tool_call.type}\n", flush=True)

  def on_tool_call_delta(self, delta, snapshot):
    if delta.type == 'code_interpreter':
      if delta.code_interpreter.input:
        print(delta.code_interpreter.input, end="", flush=True)
      if delta.code_interpreter.outputs:
        print(f"\n\noutput >", flush=True)
        for output in delta.code_interpreter.outputs:
          if output.type == "logs":
            print(f"\n{output.logs}", flush=True)

# Then, we use the `create_and_stream` SDK helper
# with the `EventHandler` class to create the Run
# and stream the response.
```

```

with client.beta.threads.runs.create_and_stream(
  thread_id=thread.id,
  assistant_id=assistant.id,
  instructions="Please address the user as Jane Doe. The user has a pr
  event_handler=EventHandler(),
) as stream:
  stream.until_done()
```

See the full list of Assistants streaming events in our API reference here. You can also see a list of SDK event listeners for these events in the Python & Node repository documentation.

Next

1. Dive deeper into How Assistants work
2. Learn more about Tools
3. Explore the Assistants playground

Was this page useful?