Jeff Prosise

TECHORAMA

Building Virtual Assistants with the OpenAI Assistants API

Review my session

Assistants API

- High-level API for creating intelligent virtual assistants
- Provides conversation context and features three built-in tools

Function Calling

Extends an LLMs ability to accomplish tasks by calling **user-provided functions**. Function descriptions enable assistant to determine which function(s) to call.

File Search

Uses **Retrieval Augmented Generation** (RAG) to put LLMs over documents. Vectorizes PDFs, DOCX files, and other document types.

Code Interpreter

Generates code and **runs it in a sandbox** to fulfill requests. Provides an assistant with the ability to **do math, generate charts and graphs**, and more.

Creating an Assistant

```python
from openai import OpenAI

client = OpenAI(api_key='OPENAI_API_KEY')

assistant = client.beta.assistants.create(
    name='LISA',
    instructions='You are an expert who answers questions about LLMs',
    model='gpt-4o'
)
```

Retrieving an Assistant by ID

```
from openai import NotFoundError

try:
    assistant = client.beta.assistants.retrieve('assistant_id')

except NotFoundError:
    print('Assistant not found')
```

Retrieving an Assistant by Name

```python
def get_assistant_by_name(name):
    for assistant in client.beta.assistants.list():
        if assistant.name == name:
            return assistant
    return None

assistant = get_assistant_by_name('LISA')
```

Running an Assistant

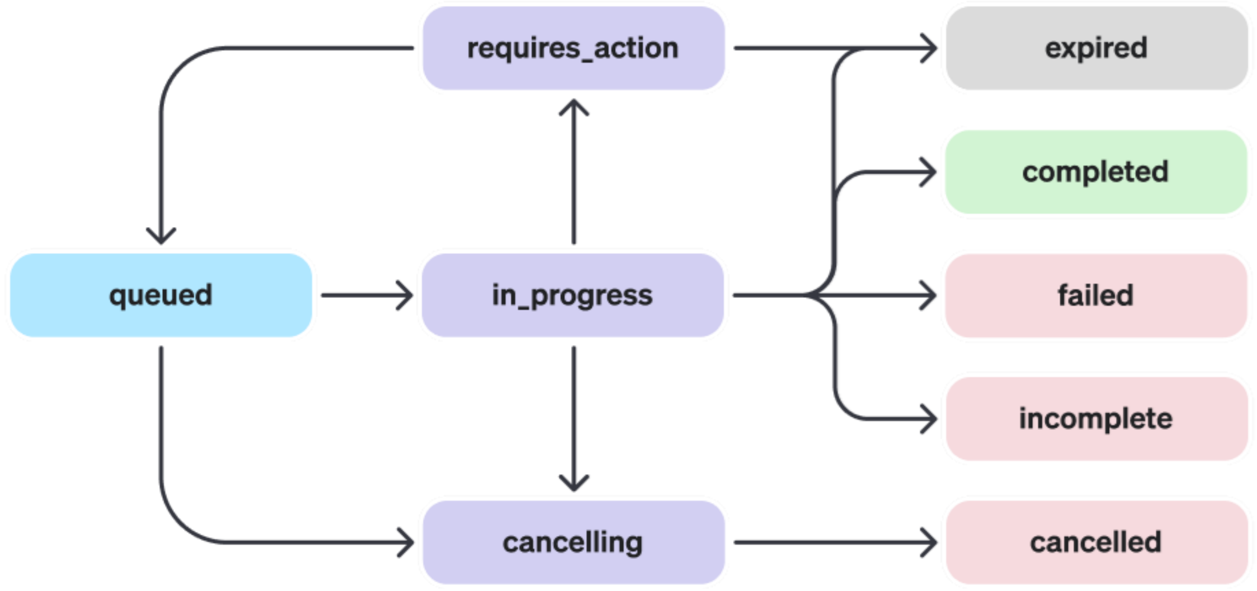
```
thread = client.beta.threads.create()

client.beta.threads.messages.create(
    thread_id=thread.id,
    role='user',
    content='How many parameters does ChatGPT have?'
)

run = client.beta.threads.runs.create_and_poll(
    thread_id=thread.id,
    assistant_id=assistant.id
)
```

Retrieving the Response

```python
if run.status == 'completed':
    messages = client.beta.threads.messages.list(thread_id=thread.id)
    print(messages.data[0].content[0].text.value)
else: # run.status is expired, failed, incomplete, or cancelled
    print(run.last_error)
```



Streaming the Response, Method 1

```
stream = client.beta.threads.runs.create(
    thread_id=thread.id,
    assistant_id=assistant.id,
    stream=True
)

for event in stream:
    if event.event == 'thread.message.delta':
        for content in event.data.delta.content or []:
            if content.type == 'text' and content.text and content.text.value:
                print(content.text.value, end='', flush=True)
```

Streaming the Response, Method 2

```python
with client.beta.threads.runs.stream(
    thread_id=thread.id,
    assistant_id=assistant.id
) as stream:
    for text in stream.text_deltas:
        print(text, end='', flush=True)
```

AssistantEventHandler

- Event-based wrapper for Server-Sent Events (SSEs)
- Subclass **AssistantEventHandler** and subscribe to relevant events

```
class EventHandler(AssistantEventHandler):
    @override
    def on_text_delta(self, delta, snapshot):
        print(delta.value, end='', flush=True) # Stream text response in chunks

    @override
    def on_text_done(self, text):
        print(text.value) # Print completed text response
```

AssistantEventHandler Overrides

`on_event`	Called for every server-sent event
`on_text_delta`	Called when a new chunk of text output has been generated
`on_text_done`	Called when text generation is complete
`on_image_file_done`	Called when an image file has been generated
• • •	
`on_timeout`	Called when the request times out
`on_exception`	Called when an exception occurs while streaming
`on_end`	Called when streaming has finished, even if an exception occurred

Streaming the Response, Method 3

```
class EventHandler(AssistantEventHandler):
    @override
    def on_text_delta(self, delta, snapshot):
        print(delta.value, end='', flush=True)

with client.beta.threads.runs.stream(
    thread_id=thread.id,
    assistant_id=assistant.id,
    event_handler=EventHandler()
) as stream:
    stream.until_done()
```

Demo

Getting Started with the Assistants API

File Search

- Employs Retrieval-Augmented Generation (RAG) to use documents as sources of information for answering questions
- Supports vector stores and "chunking" and vectorizing of documents
  - Supports more than 20 file types, including PDF, DOC, DOCX, HTML, PPTX, TXT, and MD files
    - https://platform.openai.com/docs/assistants/tools/file-search/supported-files
  - Supports up to 10,000 files per vector store and file sizes up to 512 MB
  - First gigabyte free; after that, 10 cents per day per GB
- Vector stores may be attached to assistants or threads

Creating a Vector Store

```python
# Create the vector store
vector_store = client.beta.vector_stores.create(name='Financial Reports')

# Load documents
file_paths = ['docs/microsoft.pdf', 'docs/google.pptx', 'docs/meta.docx']
files = [open(path, 'rb') for path in file_paths]

# Upload the documents and add them to the vector store
client.beta.vector_stores.file_batches.upload_and_poll(
    vector_store_id=vector_store.id,
    files=files
)
```

Specifying a Vector Store's Lifetime

```
# Create a vector store that persists until it isn't accessed for 30 days
vector_store = client.beta.vector_stores.create(
    name='Financial Reports',
    expires_after={
        'anchor': 'last_active_at',
        'days': 30
    }
)
```

Retrieving a Vector Store by ID

```python
from openai import NotFoundError

try:
    vector_store = client.beta.vector_stores.retrieve('vector_store_id')

except NotFoundError:
    print('Vector store not found')
```

Retrieving a Vector Store by Name

```python
def get_vector_store_by_name(name):
    for vector_store in client.beta.vector_stores.list():
        if vector_store.name == name:
            return vector_store
    return None

vector_store = get_vector_store_by_name('Financial Reports')
```

Connecting an Assistant to a Vector Store

```
assistant = client.beta.assistants.create(
    name='LISA',
    instructions='''
        You are an expert who answers questions about financial reports using a
        vector store. If a question can't be answered using the vector store, say
        "I'm sorry, but I don't know."
        ''',
    model='gpt-4o',
    tools=[{ 'type': 'file_search' }],
    tool_resources={ 'file_search': { 'vector_store_ids': [vector_store.id] }}
)
```

Demo

File Search

Function Calling

- Extends an LLM's powers with functions that are called when needed
  - Get weather or flight information by making external API calls
  - Access calendars or send e-mails using external API calls
  - Generate and execute database queries
- You write the functions and provide detailed JSON function descriptions to the Assistants API
- You call the functions when run status is **requires_action** and pass the output back to the Assistants API as "tool output"
  - Tool output must be a string (can be JSON)

Defining a Function

```python
def get_haversine_distance(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
    radius_earth = 3958.8  # Radius of Earth in miles
    return np.abs(radius_earth * c)
```

Describing a Function

```
tools = [{
    'type': 'function',
    'function': {
        'name': 'get_haversine_distance',
        'description': 'Computes the distance in miles between two latitudes and longitudes',
        'parameters': {
            'type': 'object',
            'properties': {
                'lat1': {
                    'type': 'number', # number, string, boolean, array, object
                    'description': 'Latitude at the origin'
                },
                  ...
            },
            'required': ['lat1', 'lon1', 'lat2', 'lon2']
        }
    }
}]
```

Making Functions Available to an Assistant

```
assistant = client.beta.assistants.create(
    name='LISA',
    instructions='You are an expert in geography who can calculate distances',
    model='gpt-4o',
    tools=tools
)
```

Calling Functions

```
for event in stream:
    if event.event == 'thread.run.requires_action':
        tool_outputs = []

        # Call each function requested by the Assistants API and collect the output
        for tool_call in event.data.required_action.submit_tool_outputs.tool_calls:
            function_name = tool_call.function.name
            # TODO: Retrieve input parameters and call function
              ...
            tool_output = { 'tool_call_id': tool_call.id, 'output': output }
            tool_outputs.append(tool_output)

        # Pass the tool outputs to the Assistants API
        client.beta.threads.runs.submit_tool_outputs(tool_outputs=tool_outputs, ...)
```

Demo

Function Calling

Database Search

- Assistants API lacks a database search tool, but you can create one using **Function Calling**
- Provide high-level database information in the function description so the Assistants API knows when to call it

```
tools = [{
    'type': 'function',
    'function': {
        'name': 'query_database',
        'description': 'Queries the database to answer questions about products',
          ...
    }
}]
```

Demo

Database Search

Code Interpreter

- **Code Interpreter** tool gives LLM the ability to do math, generate charts and graphs, and generally solve problems by running code
- Tool generates code, runs it in a sandbox, and returns the results

```
assistant = client.beta.assistants.create(
    name='LISA',
    instructions='You are an expert in geography who can calculate distances',
    model='gpt-4o',
    tools=[{ 'type': 'code_interpreter' }]
)
```

Demo

Code Interpreter