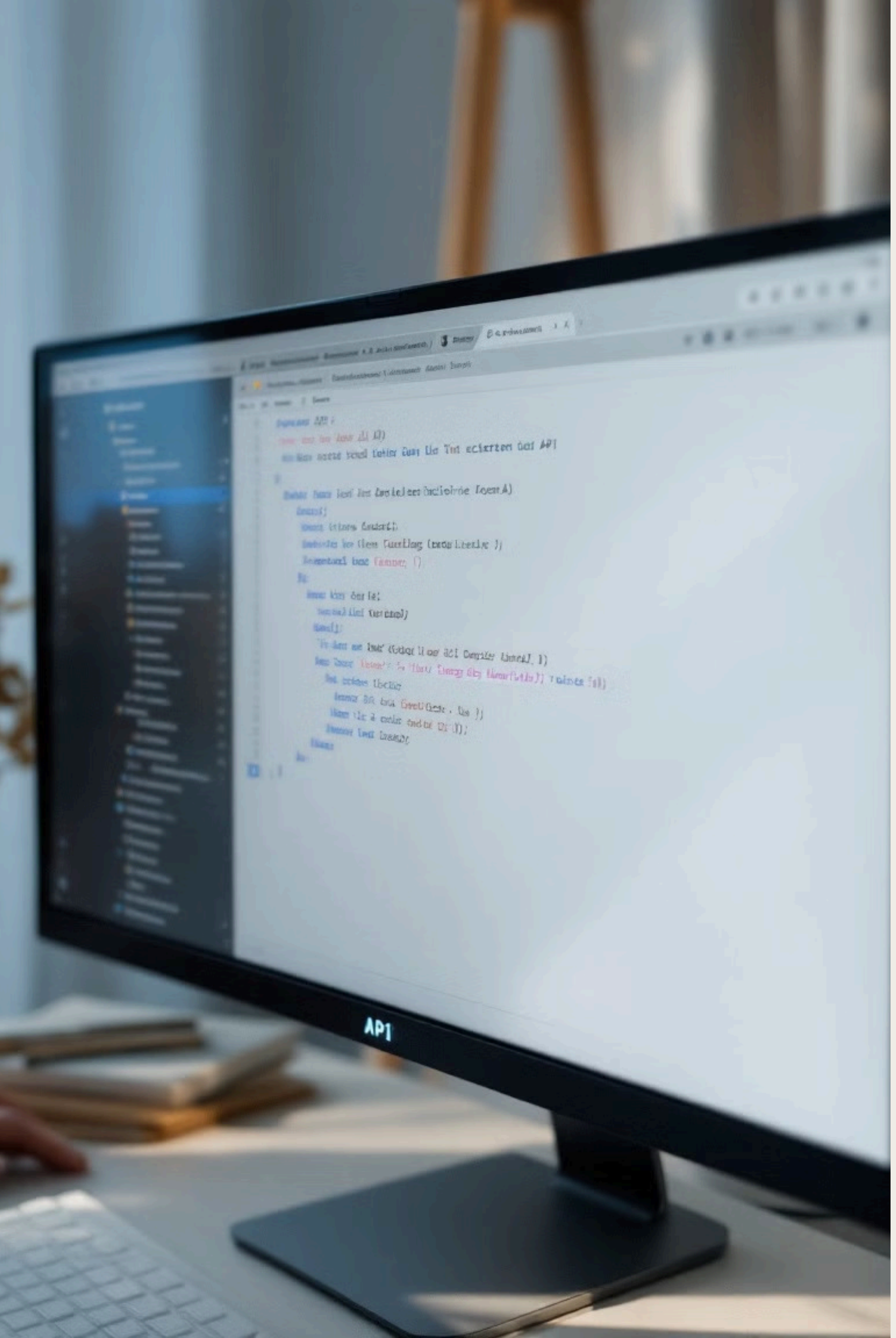
OpenAI Tutourial

Overview



Setting API KEY

The Chat Completions API is the core interface for interacting with OpenAI's language models. It provides a flexible way to hold conversations with AI models like GPT-4o.

API Key Setup:

```
from openai import OpenAI
client = OpenAI(api_key = "YOUR API_KEY")
```

Chat Completions API

Basic Parameters:

- Model (request): Select different model like "gpt-4o" or "gpt-o1" model
- Messages (resuest): A list of messages that form the conversation, include "developer"(system), "assitant" and "user"
- Temparture: controls randomness in the output between 0 and 2. Higher values like 0.8 will make the output more random
- More Like "max_token", "stream"...

This is a basic chat completions api request structure:

```
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello, how can AI help in education?"}
    ]
)
```

Conversation Management

```
# Building a conversation
conversation = [
    {"role": "system", "content": "You are a mathematics tutor helping with calculus."}
]

# First user question
conversation.append({"role": "user", "content": "What is a derivative?"})

response = client.chat.completions.create(
    model="gpt-4o",
    messages=conversation
)

# Add the assistant's response to the conversation
conversation.append({"role": "assistant", "content": response.choices[0].message.content})

# Second user question
conversation.append({"role": "user", "content": "Can you give me an example of using derivatives?"})

# Get next response with full conversation history
next_response = client.chat.completions.create(
    model="gpt-4o",
    messages=conversation
)

print(next_response.choices[0].message.content)
```

Function Calling

1. Call model with functions defined – along with your system and user messages

2. Model decides to call function(s) – model returns the name and input arguments

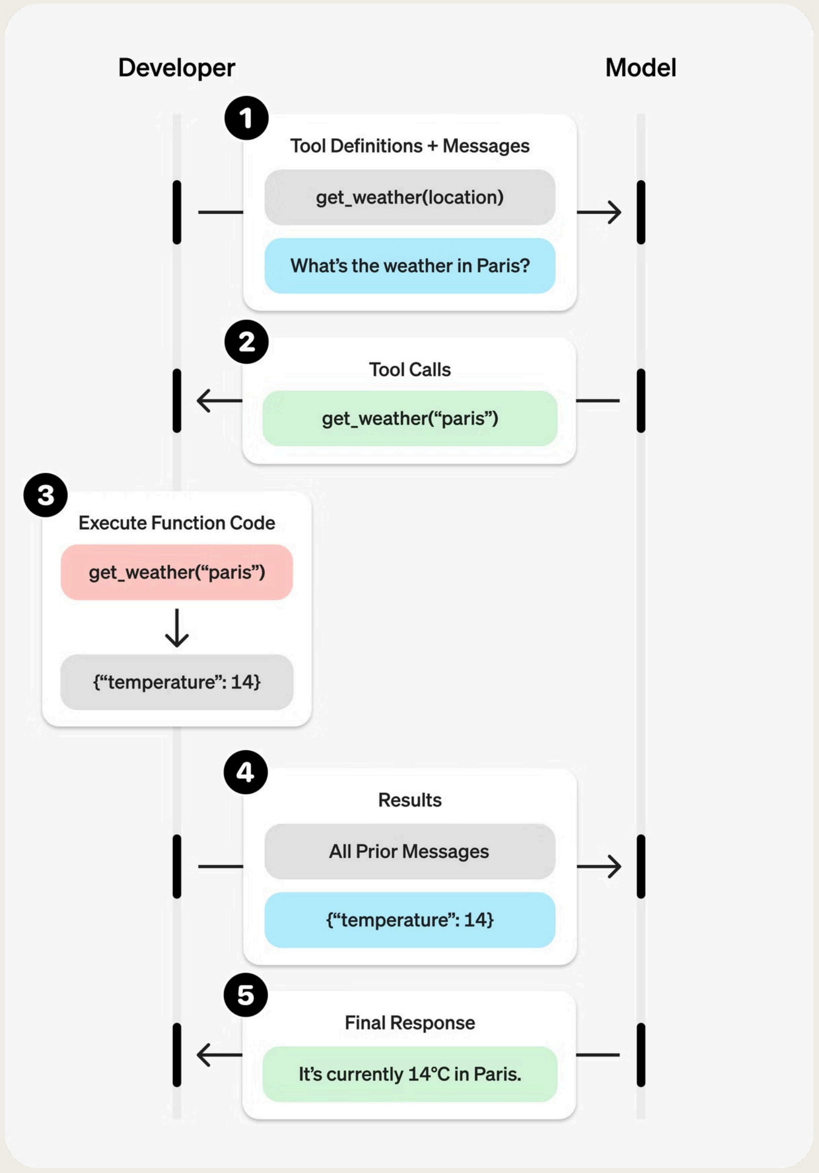
3. Execute function code – parse the model's response and handle function calls

4. Supply model with results – so it can incorporate them into its final response

5. Model responds – incorporating the result in its output

Function Calling Sample Code

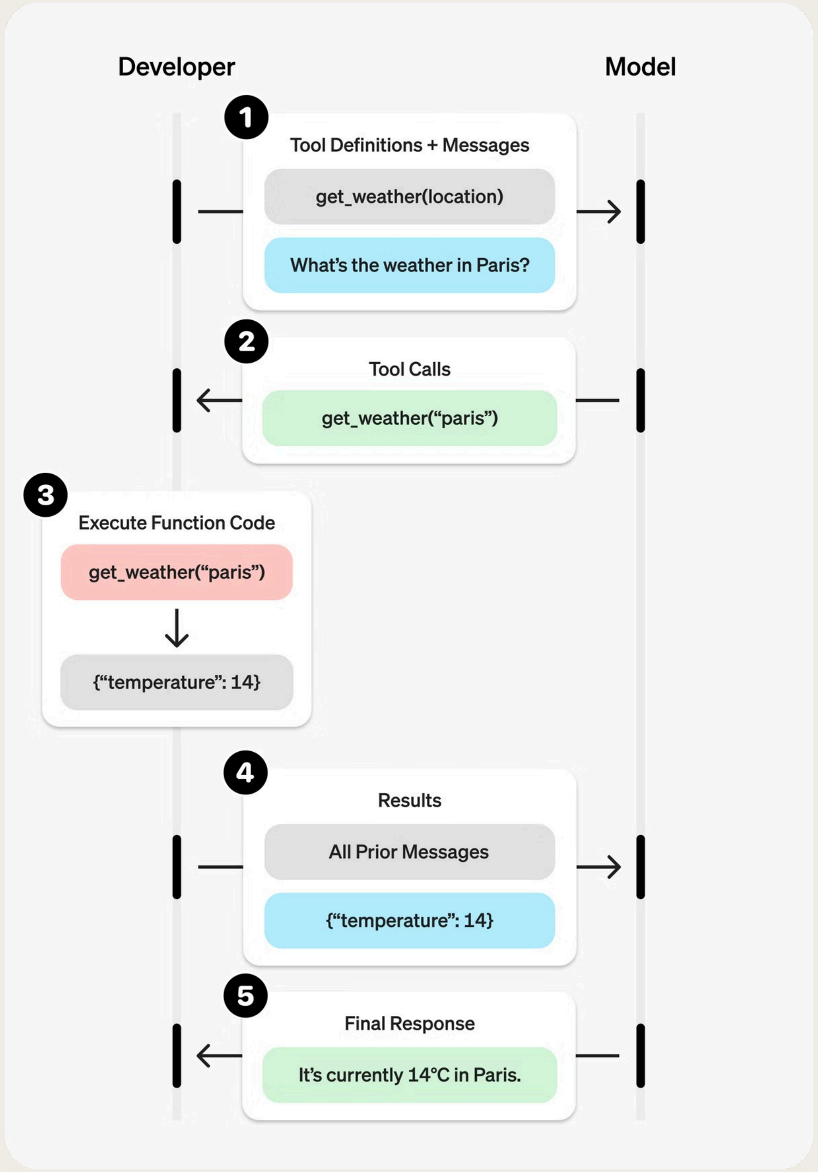
```
tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get current temperature for provided coordinates in celsius.",
        "parameters": {
            "type": "object",
            "properties": {
                "latitude": {"type": "number"},
                "longitude": {"type": "number"}
            },
            "required": ["latitude", "longitude"],
            "additionalProperties": False
        },
        "strict": True
    }
}]

messages = [{"role": "user", "content": "What's the weather like in Paris today?"}]

completion = client.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,
)
```



Made with Gamma

Function Calling Sample Code

```
# Step 3: Execute get_weather function
tool_call = completion.choices[0].message.tool_calls[0]
args = json.loads(tool_call.function.arguments)

result = get_weather(args["latitude"], args["longitude"])

# Step 4: Supply result and call model again
messages.append(completion.choices[0].message)
messages.append({
    "role": "tool",
    "tool_call_id": tool_call.id,
    "content": str(result)
})

completion_2 = client.chat.completions.create(
    model="gpt-4o",
    messages=messages,
    tools=tools,
)
```



Made with Gamma

Structure Output Json Mode

Structured Outputs is a feature that ensures the model will always generate responses that adhere to your supplied JSON Schema, so you don't need to worry about the model omitting a required key, or hallucinating an invalid enum value.

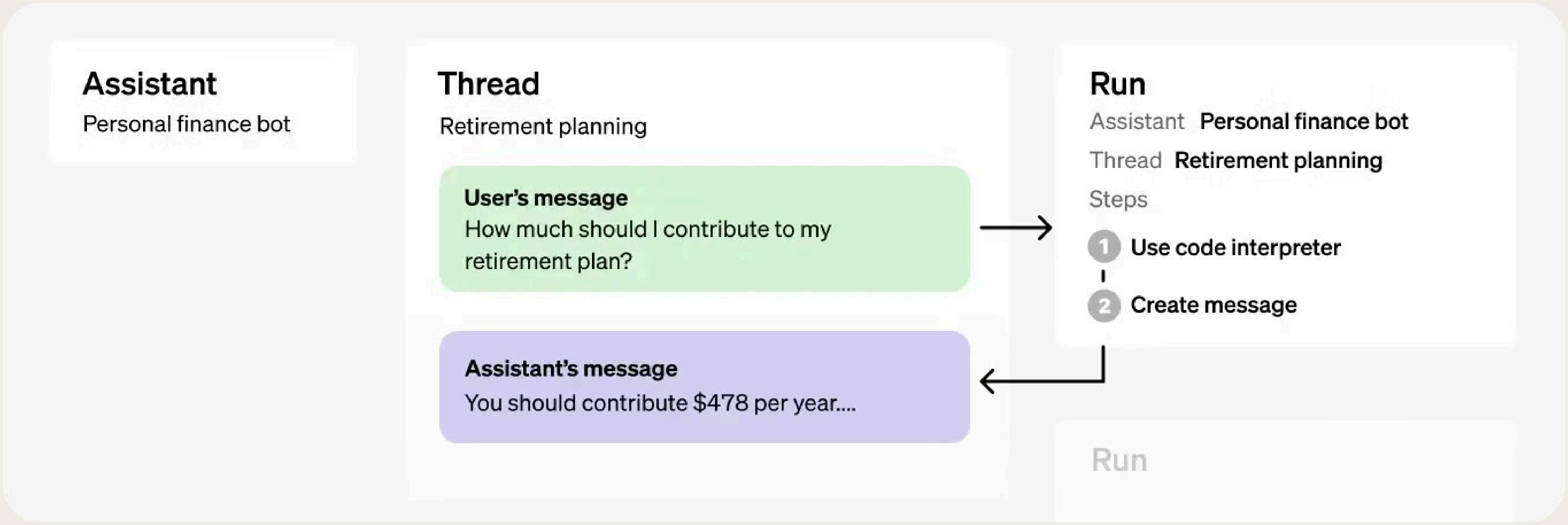
```
class CalendarEvent(BaseModel):
    name: str
    date: str
    participants: list[str]

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "Extract the event information."},
        {"role": "user", "content": "Alice and Bob are going to a science fair on Friday."},
    ],
    response_format=CalendarEvent,
)

event = completion.choices[0].message.parsed
```

Vision Image Input

```
response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": "What are in these images? Is there any difference between them?",
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": "https://upload.wikimedia.org/wikipedia/.../2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg",
                    },
                },
                {
                    "type": "image_url",
                    "image_url": {"url": f"data:image/jpeg;base64,{base64_image}"},
                },
            ],
        }
    ],
)
```

Made with Gamma

Assistants API Overview


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

1 **Persistent Conversation Management**

Assistants API automatically maintains conversation history and context across sessions, eliminating the need for developers to manually track and manage message history with each request.

2 **Integrated Tool Ecosystem**

Assistants API provides built-in tools like Code Interpreter, Retrieval, and File Processing that work seamlessly within the API, enabling complex capabilities without requiring separate infrastructure or custom implementations.

3 **Structured Interaction Framework**

Assistants API offers a more organized architecture through its Assistant, Thread, and Run components, creating clearer separation of concerns and more maintainable code for complex applications.

Assistant API Steps

1 **Create an Assistant by defining its custom instructions and picking a model. If helpful, add files and enable tools like Code Interpreter, File Search, and Function calling.**

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

2 **Create a Thread when a user starts a conversation.**

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.

3 **Add Messages to the Thread as the user asks questions.**

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 a limit of 100,000 Messages per Thread and we smartly truncate any context that does not fit into the model's context window.

4 **Run the Assistant on the Thread to generate a response by calling the model and the tools.**

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.

Step 1: Create an Assistant

```
# Basic assistant creation
assistant = client.beta.assistants.create(
    name="Math Tutor",
    instructions="You are a helpful math tutor. Explain concepts clearly and provide step-by-step solutions.",
    model="gpt-4o"
)

print(f"Created assistant with ID: {assistant.id}")
```

Step 2: Create a Thread

```
# Create a new thread
thread = client.beta.threads.create()
print(f"Created thread with ID: {thread.id}")
```

Multiple:

```
# Create threads for different users or conversations
user1_thread = client.beta.threads.create()
user2_thread = client.beta.threads.create()
```

Step 3: Add a Message to the Thread

```
# Add a message to the thread
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 me?"
)
```

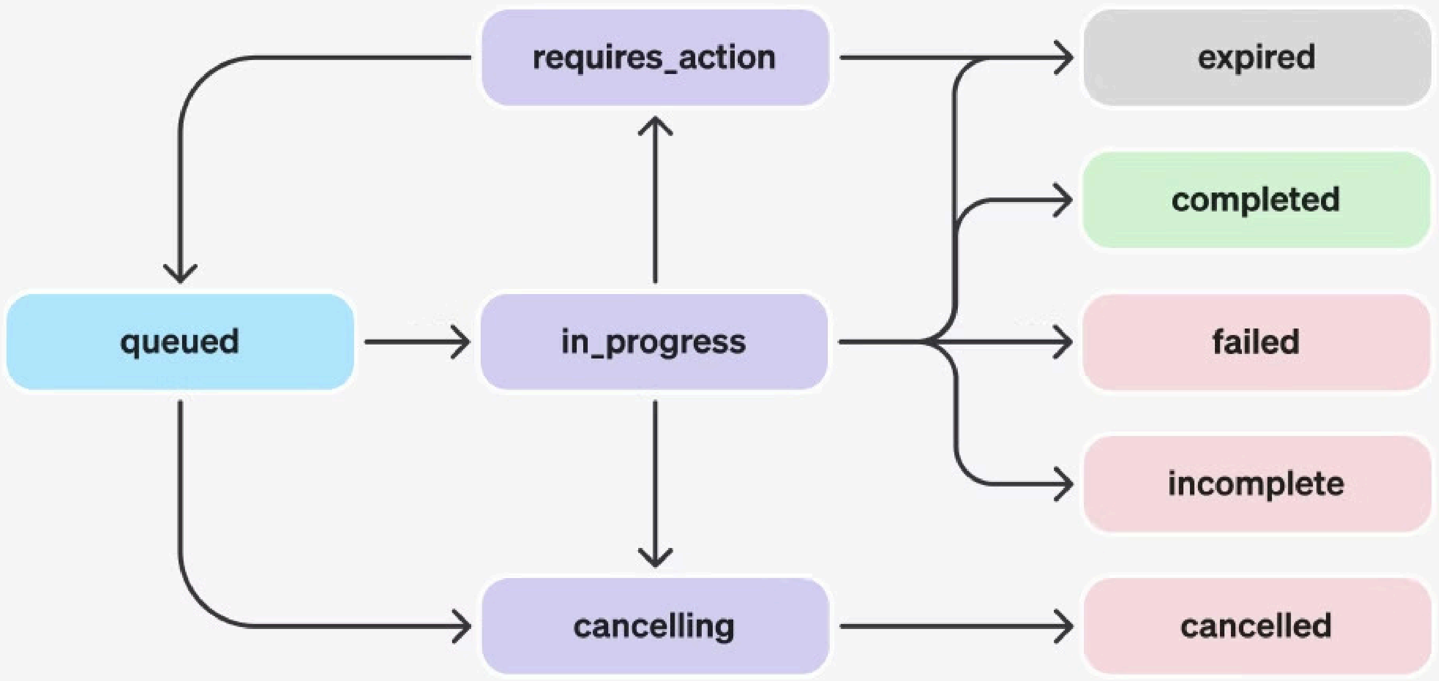
Multiple:

```
# Add messages to different threads
client.beta.threads.messages.create(
    thread_id=user1_thread.id,
    role="user",
    content="I need help with trigonometry"
)

client.beta.threads.messages.create(
    thread_id=user2_thread.id,
    role="user",
    content="Can you explain calculus to me?"
)
```

Step 4: Create a Run

```
# Run the assistant on the thread
run = client.beta.threads.runs.create(
    thread_id=thread.id,
    assistant_id=assistant.id
)
```

Runs and Run Steps



Assistants API Tools

1 **File Search**

Built-in RAG tool to process and search through files

2 **Code Interpreter**

Write and run python code, process files and diverse data

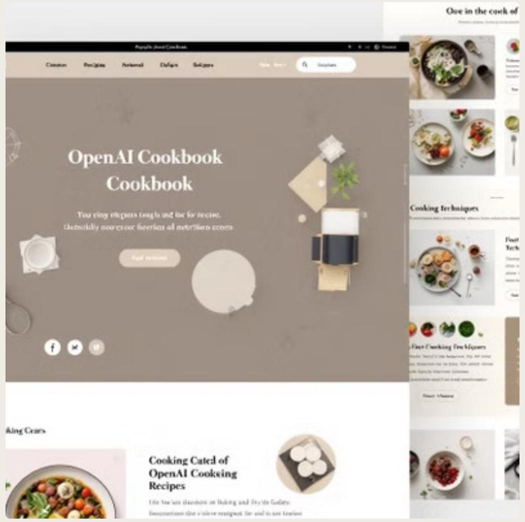
3 **Function Calling**

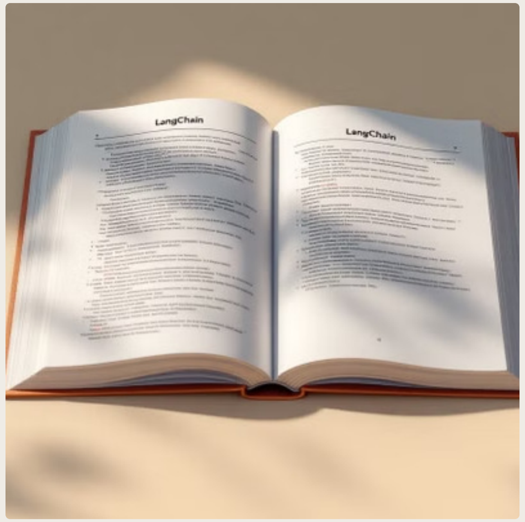
Use your own custom functions to interact with your application

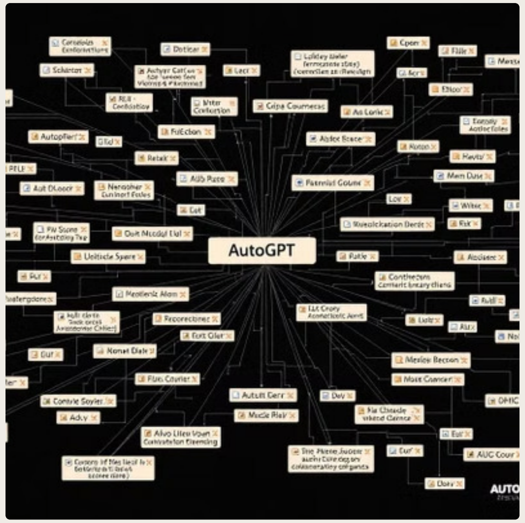
File Search

```
# Create a vector store caled "Financial Statements"
vector_store = client.beta.vector_stores.create(name="Financial Statements")

# Ready the files for upload to OpenAI
file_paths = ["edgar/goog-10k.pdf", "edgar/brka-10k.txt"]
file_streams = [open(path, "rb") for path in file_paths]

# Use the upload and poll SDK helper to upload the files, add them to the vector store,
# and poll the status of the file batch for completion.
file_batch = client.beta.vector_stores.file_batches.upload_and_poll(
  vector_store_id=vector_store.id, files=file_streams
)

# You can print the status and the file counts of the batch to see the result of this operation.
print(file_batch.status)
print(file_batch.file_counts)

# Update the assistant to use the new Vector Store
assistant = client.beta.assistants.update(
  assistant_id=assistant.id,
  tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},
)
```

Extended Reading







- OpenAI Document
- Prompt enginnering
- LangChain
- AutoGPT