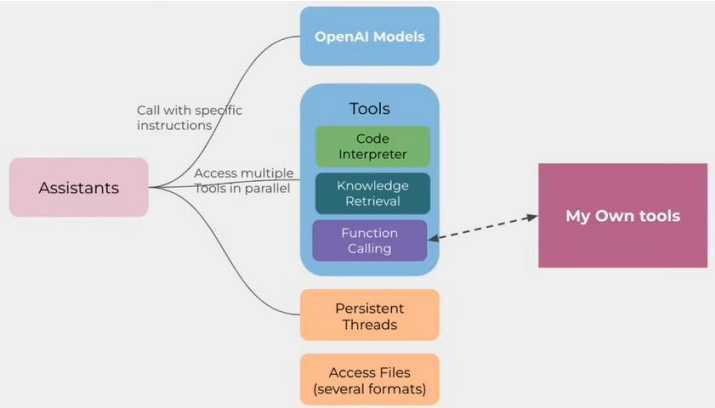
Assistant API

- The Assistants API allows you to build AI assistants within your own applications.
- A tool to allow developers to craft powerful AI assistants that can perform variety of tasks. Assistant API extends the OpenAI
- An Assistant has instructions and can leverage models, tools, and files to respond to user queries.
- The Assistants API currently supports three types of tools:
    - Code Interpreter,
    - File Search,
    - Function calling.

How Assistants work



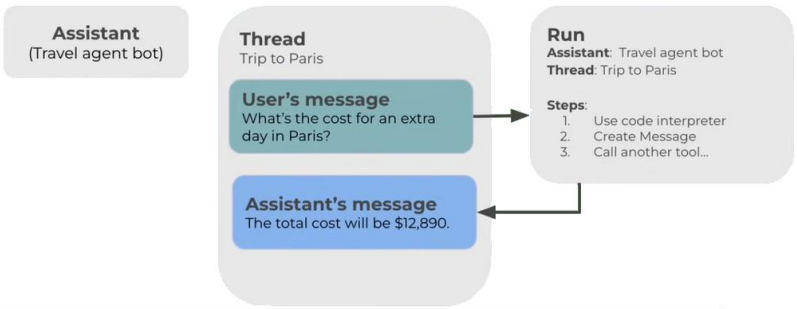
1. Assistants can call OpenAI's models with specific instructions to tune their personality and capabilities.
2. Assistants can access multiple tools in parallel. These can be both OpenAI-hosted tools — like code_interpreter and file_search — or tools you build / host (via function calling).
3. Assistants 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.

Assistant API vs Chat Completion API

Aspect	Assistants API	Chat Completions API
Initial Setup	Create an Assistant with defined capabilities.	No explicit setup of an Assistant is required.

Session Management	Initiate and manage a thread for ongoing conversations.	No explicit session or thread management; each request is independent.
Interaction Handling	Interact through the Runs API, considering the entire conversation context.	Send the entire chat history in each request, including system prompts and previous interactions.
Context Management	Persistent context through the thread, suitable for extended conversations.	Context is provided in each request; best for single interactions or where full context is included each time.
Complexity	More complex setup, offering detailed control and customization.	Simpler and more straightforward, with less granular control.
Ideal Use Cases	Best for detailed, context-heavy conversational applications.	Suited for simpler chatbots or applications where each response is standalone.
Capabilities	Advanced capabilities like integration with a code interpreter, online search for information queries, the ability to retrieve knowledge from uploaded Files, and function calling.	Primarily focused on function calling, with less emphasis on extended capabilities beyond generating text responses.

Assistants API Building Blocks



1. **Create Assistants in Playground:**

   Name: Personal Trainer

   Instructions: You are the best personal trainer and nutritionist who knows how to get clients to build lean muscles. You've trained high-caliber athletes and movie stars.

2. **Create and Run user message**

   a. Expand run and set **Add run instructions =** Please address the user as Sandeep Soni.

b. **Build the context:**
    i. **Enter your message:** I want to build muscles. What food should I eat.
    ii. **Click on +**
c. Create and Execute Run
    i. **Enter your message:** How many glasses of water should I drink .
    ii. **Click on Run**
d. Create and Execute Run
    i. **Enter your message:** What leg exercises do you recommend.
    ii. **Click on Run**

Assistant API

Create Assistance and Tread

start.py

```
from openai import OpenAI
from util import GetOpenAIClient

# Get OpenAI Client from Util.
client = GetOpenAIClient()

assistant = client.beta.assistants.create(
    name="Personal Trainer",
    instructions="""
        You are the best personal trainer and nutritionist who knows how to get clients to build lean muscles.
        You have trained high-caliber athletes and movie stars.
        """,
    tools=[{"type": "code_interpreter"}],
    model="gpt-4o"
)
print(assistant.id)

# Create a thread
thread = client.beta.threads.create(
    messages=[{
        "role": "user",
        "content":"How do I get started working out to lose fat"
```

```
    }]
)
print(thread.id)
```

Create Run and Poll (This can be executed multiple times with different prompts.

run.py

```
from openai import OpenAI
from util import GetOpenAIClient

# Get OpenAI Client from Util.
client = GetOpenAIClient()

#Run the code and record the Assistance ID and Thread ID
assistant_id = "asst_XXXXXXXXXXXXXX"
thread_id = "thread_XXXXXXXXXXXXX"

# Poll the existing runs
runs = client.beta.threads.runs.list(thread_id=thread_id)
for run in runs:
    print(f"Run ID: {run.id}, Status: {run.status}")

# Read text from console
user_input = input("Enter your prompt: ")

# Use the user input in the message creation
message = client.beta.threads.messages.create(
    thread_id=thread_id,
    role="user",
    content=user_input
)

# Create a new run for the assistant
run = client.beta.threads.runs.create_and_poll(
    thread_id=thread_id,
    assistant_id=assistant_id,
```

```
    instructions="Please address the user as Sandeep Soni. The user has a premium account."
)

# Poll the run until it is completed
print("Polling for run to complete...")
while True:
    if run.status == 'completed':
        messages = client.beta.threads.messages.list(
            thread_id=thread_id
        )
        for message in messages:
            print(f"Response=\n{message.content[0].text.value}")
            print("----------------------------------------------------------------------")
        break
    else:
        print(f"Run ID: {run.id}, Status: {run.status}")
    time.sleep(5)
```

Note: We can execute run.py multiple times and keep adding new questions. It remembers the content and will provide output based on previous questions.