

Assistants File Search Beta

[Copy page](#)


File Search augments the Assistant with knowledge from outside its model, such as proprietary product information or documents provided by your users. OpenAI automatically parses and chunks your documents, creates and stores the embeddings, and use both vector and keyword search to retrieve relevant content to answer user queries.

Quickstart

In this example, we'll create an assistant that can help answer questions about companies' financial statements.

Step 1: Create a new Assistant with File Search Enabled

Create a new assistant with `file_search` enabled in the `tools` parameter of the Assistant.

```
python   
1 from openai import OpenAI  
2  
3 client = OpenAI()  
4  
5 assistant = client.beta.assistants.create(  
6     name="Financial Analyst Assistant",  
7     instructions="You are an expert financial analyst. Use your knowledge base to answer questions",  
8     model="gpt-4o",  
9     tools=[{"type": "file_search"}],  
10 )
```

Once the `file_search` tool is enabled, the model decides when to retrieve content based on user messages.

Step 2: Upload files and add them to a Vector Store

To access your files, the `file_search` tool uses the Vector Store object.

Upload your files and create a Vector Store to contain them.

Once the Vector Store is created, you should poll its status until all files are out of the `in_progress` state to

ensure that all content has finished processing. The SDK provides helpers to uploading and polling in one shot.

```

1 # Create a vector store caled "Financial Statements"
2 vector_store = client.beta.vector_stores.create(name="Financial Statements")
3
4 # Ready the files for upload to OpenAI
5 file_paths = ["edgar/goog-10k.pdf", "edgar/brka-10k.txt"]
6 file_streams = [open(path, "rb") for path in file_paths]
7
8 # Use the upload and poll SDK helper to upload the files, add them to the vector store,
9 # and poll the status of the file batch for completion.
10 file_batch = client.beta.vector_stores.file_batches.upload_and_poll(
11     vector_store_id=vector_store.id, files=file_streams
12 )
13
14 # You can print the status and the file counts of the batch to see the result of this operat
15 print(file_batch.status)
16 print(file_batch.file_counts)

```

Step 3: Update the assistant to use the new Vector Store

To make the files accessible to your assistant, update the assistant's `tool_resources` with the new `vector_store` id.

```

1 assistant = client.beta.assistants.update(
2     assistant_id=assistant.id,
3     tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},
4 )

```

Step 4: Create a thread

You can also attach files as Message attachments on your thread. Doing so will create another `vector_store` associated with the thread, or, if there is already a vector store attached to this thread, attach the new files to the existing thread vector store. When you create a Run on this thread, the file search tool will query both the `vector_store` from your assistant and the `vector_store` on the thread.

In this example, the user attached a copy of Apple's latest 10-K filing.

```

1 # Upload the user provided file to OpenAI
2 message_file = client.files.create(
3     file=open("edgar/aapl-10k.pdf", "rb"), purpose="assistants"
4 )
5
6 # Create a thread and attach the file to the message

```

```

7  thread = client.beta.threads.create(
8      messages=[
9          {
10             "role": "user",
11             "content": "How many shares of AAPL were outstanding at the end of of October 2023?",
12             # Attach the new file to the message.
13             "attachments": [
14                 { "file_id": message_file.id, "tools": [{"type": "file_search"}] }
15             ],
16         }
17     ]
18 )

```

The thread now has a vector store with that file in its tool resources.

```

print(thread.tool_resources.file_search)

```

Vector stores created using message attachments have a default expiration policy of 7 days after they were last active (defined as the last time the vector store was part of a run). This default exists to help you manage your vector storage costs. You can override these expiration policies at any time. Learn more [here](#).

Step 5: Create a run and check the output

Now, create a Run and observe that the model uses the File Search tool to provide a response to the user's question.

With streaming

Without streaming

python 

```

1  from typing_extensions import override
2  from openai import AssistantEventHandler, OpenAI
3
4  client = OpenAI()
5
6  class EventHandler(AssistantEventHandler):
7      @override
8      def on_text_created(self, text) -> None:
9          print(f"\nassistant > ", end="", flush=True)
10
11     @override
12     def on_tool_call_created(self, tool_call):
13         print(f"\nassistant > {tool_call.type}\n", flush=True)
14
15     @override
16     def on_message_done(self, message) -> None:
17         # print a citation to the file searched
18         message_content = message.content[0].text
19         annotations = message_content.annotations
20         citations = []
21         for index, annotation in enumerate(annotations):

```

```

        message_content.value = message_content.value.replace(
            annotation.text, f"[{index}]"
        )
    if file_citation := getattr(annotation, "file_citation", None):
        cited_file = client.files.retrieve(file_citation.file_id)
        citations.append(f"[{index}] {cited_file.filename}")

    print(message_content.value)
    print("\n".join(citations))

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

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

```

Your new assistant will query both attached vector stores (one containing `goog-10k.pdf` and `brka-10k.txt` , and the other containing `aapl-10k.pdf`) and return this result from `aapl-10k.pdf` .

To retrieve the contents of the file search results that were used by the model, use the `include` query parameter and provide a value of

`step_details.tool_calls[*].file_search.results[*].content` in the format
`?include[]=step_details.tool_calls[*].file_search.results[*].content` .

How it works

The `file_search` tool implements several retrieval best practices out of the box to help you extract the right data from your files and augment the model's responses. The `file_search` tool:

- Rewrites user queries to optimize them for search.

- Breaks down complex user queries into multiple searches it can run in parallel.

- Runs both keyword and semantic searches across both assistant and thread vector stores.

- Reranks search results to pick the most relevant ones before generating the final response.

By default, the `file_search` tool uses the following settings but these can be [configured](#) to suit your needs:

- Chunk size: 800 tokens

- Chunk overlap: 400 tokens

- Embedding model: `text-embedding-3-large` at 256 dimensions

Maximum number of chunks added to context: 20 (could be fewer)

Ranker: `auto` (OpenAI will choose which ranker to use)

Score threshold: 0 minimum ranking score

Known Limitations

We have a few known limitations we're working on adding support for in the coming months:

- 1 Support for deterministic pre-search filtering using custom metadata.
- 2 Support for parsing images within documents (including images of charts, graphs, tables etc.)
- 3 Support for retrievals over structured file formats (like `csv` or `jsonl`).
- 4 Better support for summarization — the tool today is optimized for search queries.

Vector stores

Vector Store objects give the File Search tool the ability to search your files. Adding a file to a `vector_store` automatically parses, chunks, embeds and stores the file in a vector database that's capable of both keyword and semantic search. Each `vector_store` can hold up to 10,000 files. Vector stores can be attached to both Assistants and Threads. Today, you can attach at most one vector store to an assistant and at most one vector store to a thread.

Creating vector stores and adding files

You can create a vector store and add files to it in a single API call:

python 

```
1 vector_store = client.beta.vector_stores.create(  
2     name="Product Documentation",  
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5']  
4 )
```

Adding files to vector stores is an async operation. To ensure the operation is complete, we recommend that you use the 'create and poll' helpers in our official SDKs. If you're not using the SDKs, you can retrieve the `vector_store` object and monitor its `file_counts` property to see the result of the file ingestion operation.

Files can also be added to a vector store after it's created by [creating vector store files](#).

python 

```
1 file = client.beta.vector_stores.files.create_and_poll(  
2     vector_store_id="vs_abc123",  
3  
4
```

```
file_id="file-abc123"  
)
```

Alternatively, you can add several files to a vector store by [creating batches](#) of up to 500 files.

python  

```
1 batch = client.beta.vector_stores.file_batches.create_and_poll(  
2     vector_store_id="vs_abc123",  
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5']  
4 )
```

Similarly, these files can be removed from a vector store by either:

Deleting the [vector store file object](#) or,

By deleting the underlying [file object](#) (which removes the file it from all `vector_store` and `code_interpreter` configurations across all assistants and threads in your organization)

The maximum file size is 512 MB. Each file should contain no more than 5,000,000 tokens per file (computed automatically when you attach a file).

File Search supports a variety of file formats including `.pdf`, `.md`, and `.docx`. More details on the file extensions (and their corresponding MIME-types) supported can be found in the [Supported files](#) section below.

Attaching vector stores

You can attach vector stores to your Assistant or Thread using the `tool_resources` parameter.

python  

```
1 assistant = client.beta.assistants.create(  
2     instructions="You are a helpful product support assistant and you answer questions based c  
3     model="gpt-4o",  
4     tools=[{"type": "file_search"}],  
5     tool_resources={  
6         "file_search": {  
7             "vector_store_ids": ["vs_1"]  
8         }  
9     }  
10 )  
11  
12 thread = client.beta.threads.create(  
13     messages=[ { "role": "user", "content": "How do I cancel my subscription?" } ],  
14     tool_resources={  
15         "file_search": {  
16             "vector_store_ids": ["vs_2"]  
17         }  
18     }  
19 )
```

```
}  
)
```

You can also attach a vector store to Threads or Assistants after they're created by updating them with the right `tool_resources` .

Ensuring vector store readiness before creating runs

We highly recommend that you ensure all files in a `vector_store` are fully processed before you create a run. This will ensure that all the data in your `vector_store` is searchable. You can check for `vector_store` readiness by using the polling helpers in our SDKs, or by manually polling the `vector_store` object to ensure the `status` is `completed` .

As a fallback, we've built a **60 second maximum wait** in the Run object when the **thread's** vector store contains files that are still being processed. This is to ensure that any files your users upload in a thread are fully searchable before the run proceeds. This fallback wait *does not* apply to the assistant's vector store.

Customizing File Search settings

You can customize how the `file_search` tool chunks your data and how many chunks it returns to the model context.

Chunking configuration

By default, `max_chunk_size_tokens` is set to `800` and `chunk_overlap_tokens` is set to `400` , meaning every file is indexed by being split up into 800-token chunks, with 400-token overlap between consecutive chunks.

You can adjust this by setting `chunking_strategy` when adding files to the vector store. There are certain limitations to `chunking_strategy` :

`max_chunk_size_tokens` must be between 100 and 4096 inclusive.

`chunk_overlap_tokens` must be non-negative and should not exceed

`max_chunk_size_tokens / 2` .

Number of chunks

By default, the `file_search` tool outputs up to 20 chunks for `gpt-4*` models and up to 5 chunks for `gpt-3.5-turbo` . You can adjust this by setting `file_search.max_num_results` in the tool when creating the assistant or the run.

Note that the `file_search` tool may output fewer than this number for a myriad of reasons:

The total number of chunks is fewer than `max_num_results` .

The total token size of all the retrieved chunks exceeds the token "budget" assigned to the `file_search` tool. The `file_search` tool currently has a token budget of:

4,000 tokens for `gpt-3.5-turbo`

16,000 tokens for `gpt-4*` models

Improve file search result relevance with chunk ranking

By default, the file search tool will return all search results to the model that it thinks have any level of relevance when generating a response. However, if responses are generated using content that has low relevance, it can lead to lower quality responses. You can adjust this behavior by both inspecting the file search results that are returned when generating responses, and then tuning the behavior of the file search tool's ranker to change how relevant results must be before they are used to generate a response.

Inspecting file search chunks

The first step in improving the quality of your file search results is inspecting the current behavior of your assistant. Most often, this will involve investigating responses from your assistant that are not performing well. You can get [granular information about a past run step](#) using the REST API, specifically using the `include` query parameter to get the file chunks that are being used to generate results.

Include file search results in response when creating a run

python ↕ 

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 run_step = client.beta.threads.runs.steps.retrieve(
5     thread_id="thread_abc123",
6     run_id="run_abc123",
7     step_id="step_abc123",
8     include=["step_details.tool_calls[*].file_search.results[*].content"]
9 )
10
11 print(run_step)
```

You can then log and inspect the search results used during the run step, and determine whether or not they are consistently relevant to the responses your assistant should generate.

Configure ranking options

If you have determined that your file search results are not sufficiently relevant to generate high quality responses, you can adjust the settings of the result ranker used to choose which search results should be used to generate responses. You can adjust this setting

`file_search.ranking_options` in the tool when **creating the assistant** or **creating the run**.

The settings you can configure are:

`ranker` - Which ranker to use in determining which chunks to use. The available values are `auto`, which uses the latest available ranker, and `default_2024_08_21`.

`score_threshold` - a ranking between 0.0 and 1.0, with 1.0 being the highest ranking. A higher number will constrain the file chunks used to generate a result to only chunks with a higher possible relevance, at the cost of potentially leaving out relevant chunks.

Managing costs with expiration policies

The `file_search` tool uses the `vector_stores` object as its resource and you will be billed based on the `size` of the `vector_store` objects created. The size of the vector store object is the sum of all the parsed chunks from your files and their corresponding embeddings.

You first GB is free and beyond that, usage is billed at \$0.10/GB/day of vector storage. There are no other costs associated with vector store operations.

In order to help you manage the costs associated with these `vector_store` objects, we have added support for expiration policies in the `vector_store` object. You can set these policies when creating or updating the `vector_store` object.

python 

```
1 vector_store = client.beta.vector_stores.create_and_poll(  
2     name="Product Documentation",  
3     file_ids=['file_1', 'file_2', 'file_3', 'file_4', 'file_5'],  
4     expires_after={  
5         "anchor": "last_active_at",  
6         "days": 7  
7     }  
8 )
```

Thread vector stores have default expiration policies

Vector stores created using thread helpers (like `tool_resources.file_search.vector_stores` in Threads or `message.attachments` in Messages) have a default expiration policy of 7 days after they were last active (defined as the last time the vector store was part of a run).

When a vector store expires, runs on that thread will fail. To fix this, you can simply recreate a new `vector_store` with the same files and reattach it to the thread.

python 

```
1 all_files = list(client.beta.vector_stores.files.list("vs_expired"))  
2  
3 vector_store = client.beta.vector_stores.create(name="rag-store")  
4 client.beta.threads.update(  
5     "thread_abc123",  
6     tool_resources={"file_search": {"vector_store_ids": [vector_store.id]}},  
7 )  
8  
9 for file_batch in chunked(all_files, 100):  
10     client.beta.vector_stores.file_batches.create_and_poll(  
11         vector_store_id=vector_store.id,  
12         file_ids=file_batch
```

```
vector_store_id=vector_store.id, file_ids=[file.id for file in file_batch]
)
```

Supported files

For `text/` MIME types, the encoding must be one of `utf-8`, `utf-16`, or `ascii`.

FILE FORMAT	MIME TYPE
<code>.c</code>	<code>text/x-c</code>
<code>.cpp</code>	<code>text/x-c++</code>
<code>.cs</code>	<code>text/x-csharp</code>
<code>.css</code>	<code>text/css</code>
<code>.doc</code>	<code>application/msword</code>
<code>.docx</code>	<code>application/vnd.openxmlformats-officedocument.wordprocessingml.document</code>
<code>.go</code>	<code>text/x-golang</code>
<code>.html</code>	<code>text/html</code>
<code>.java</code>	<code>text/x-java</code>
<code>.js</code>	<code>text/javascript</code>
<code>.json</code>	<code>application/json</code>
<code>.md</code>	<code>text/markdown</code>
<code>.pdf</code>	<code>application/pdf</code>
<code>.php</code>	<code>text/x-php</code>
<code>.pptx</code>	<code>application/vnd.openxmlformats-officedocument.presentationml.presentation</code>
<code>.py</code>	<code>text/x-python</code>
<code>.py</code>	<code>text/x-script.python</code>
<code>.rb</code>	<code>text/x-ruby</code>
<code>.sh</code>	<code>application/x-sh</code>
<code>.tex</code>	<code>text/x-tex</code>
<code>.ts</code>	<code>application/typescript</code>
<code>.txt</code>	<code>text/plain</code>

