RAG Gen AI App using LangChain + OpenAI

Steps:

- Scraping data from a webpage
- Chunking the data
- Embedding the chunks
- Storing in a vector database (FAISS)
- Creating a retriever
- Passing relevant documents to an LLM to generate answers

🧠 High-Level Structure

```
scss
CopyEdit
📄 Web Page (docs)
   ↓
🧩 Chunk Text (Text Splitter)
   ↓
🔢 Convert to Vectors (Embeddings)
   ↓
💾 Store in Vector DB (FAISS)
   ↓
🔍 Retrieve Relevant Chunks (Retriever)
   ↓
💬 Generate Answers (LLM + Prompt)
```

1. 📦 Setup API Keys and Environment

```
import os
from dotenv import load_dotenv
load_dotenv()

os.environ['OPENAI_API_KEY'] = os.getenv("OPENAI_API_KEY")
os.environ["LANGCHAIN_API_KEY"] = os.getenv("LANGCHAIN_API_KEY")
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = os.getenv("LANGCHAIN_PROJECT")
```

✅ Loads your API keys securely and configures LangSmith for tracking.

2. 🌐 Load Web Data

```
from langchain_community.document_loaders import WebBaseLoader

loader = WebBaseLoader("https://docs.smith.langchain.com/administration/tu
torials/manage_spend")
docs = loader.load()
docs
```

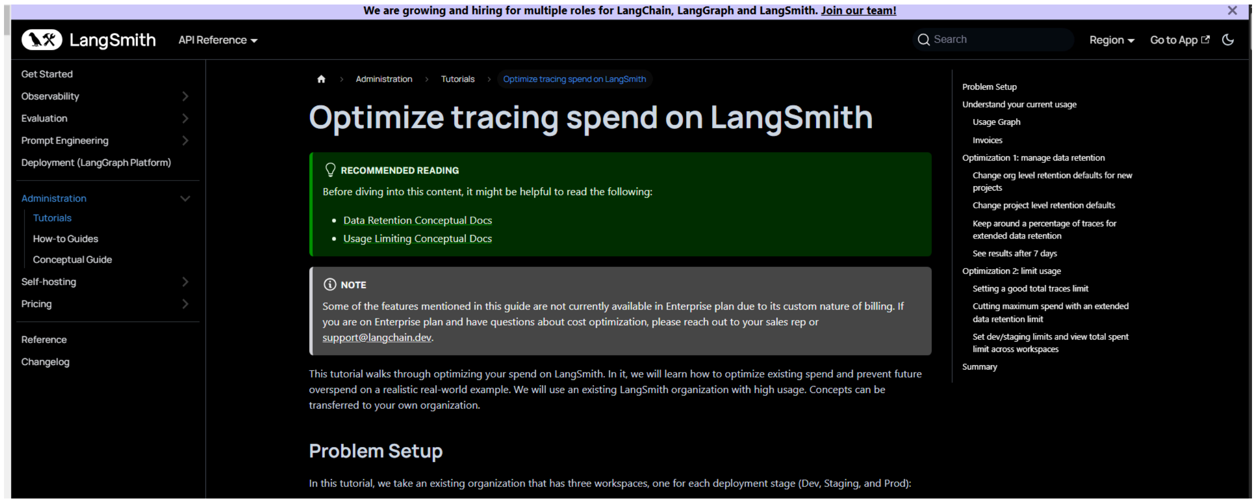
```
Output:

[Document(metadata={'source': 'https://docs.smith.langchain.com/administra
tion/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith |
🦜🛠️ LangSmith', 'description': 'Before diving into this content, it might be hel
pful to read the following:', 'language': 'en'}, page_content='\n\n\n\n\nOptimiz
e tracing spend on LangSmith | 🦜🛠️ LangSmith\n\n\n\n\n\n\n\n\nSkip to main
contentWe are growing and hiring for multiple roles for LangChain, LangGraph
and LangSmith. Join our team!API ReferenceRESTPythonJS/TSSearchRegion
USEUGo to AppGet StartedObservabilityEvaluationPrompt EngineeringDeploy
ment (LangGraph Platfo
...
```

✅ 👆 Scrapes the page content and loads it as documents into `docs`.

Webpage:

https://docs.smith.langchain.com/administration/tutorials/manage_spend



We are growing and hiring for multiple roles for LangChain, LangGraph and LangSmith. Join our team!

Optimize tracing spend on LangSmith

RECOMMENDED READING

Before diving into this content, it might be helpful to read the following:

- Data Retention Conceptual Docs
- Usage Limiting Conceptual Docs

NOTE

Some of the features mentioned in this guide are not currently available in Enterprise plan due to its custom nature of billing. If you are on Enterprise plan and have questions about cost optimization, please reach out to your sales rep or support@langchain.dev.

This tutorial walks through optimizing your spend on LangSmith. In it, we will learn how to optimize existing spend and prevent future overspend on a realistic real-world example. We will use an existing LangSmith organization with high usage. Concepts can be transferred to your own organization.

Problem Setup

In this tutorial, we take an existing organization that has three workspaces, one for each deployment stage (Dev, Staging, and Prod):

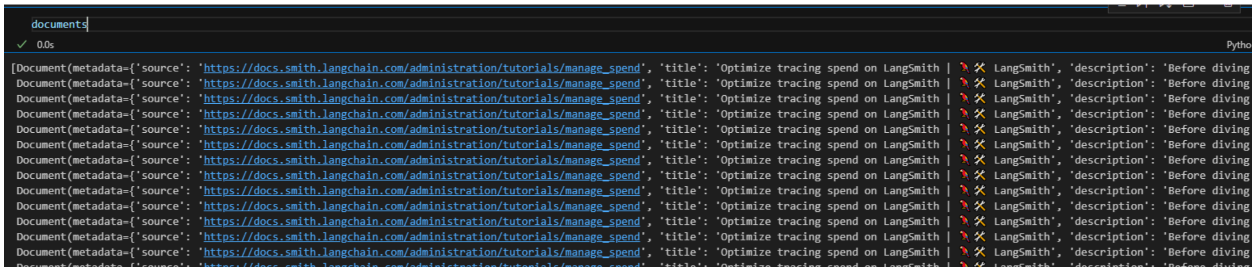
Problem Setup
Understand your current usage
Usage Graph
Invoices
Optimization 1: manage data retention
Change org level retention defaults for new projects
Change project level retention defaults
Keep around a percentage of traces for extended data retention
See results after 7 days
Optimization 2: limit usage
Setting a good total traces limit
Cutting maximum spend with an extended data retention limit
Set dev/staging limits and view total spent limit across workspaces
Summary

3. ✂️ Split Text into Chunks

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
documents = text_splitter.split_documents(docs)
```

✅ Breaks long text into manageable parts (chunks), with overlap to preserve context.



```
documents
✓ 0.0s
[Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
 Document(metadata={'source': 'https://docs.smith.langchain.com/administration/tutorials/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangSmith', 'description': 'Before diving
```

4. 🔤 Generate Embeddings (Vectors)

```
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
```

✅ Converts each chunk of text into a vector (number list) using OpenAI's embedding model.

5. 💾 Store Vectors in Vector DB (FAISS)

```
from langchain_community.vectorstores import FAISS

vectorstoredb = FAISS.from_documents(documents, embeddings)
```

✅ Stores all vectors in a searchable FAISS database.

6. 🔍 Query Similar Texts (Optional Check)

```
query = "LangSmith has two usage limits: total traces and extended"
result = vectorstoredb.similarity_search(query)
print(result[0].page_content)
```

```
Output:

'use usage limits to prevent future overspend.LangSmith has two usage limits:
total traces and extended retention traces. These correspond to the two metri
cs we\'ve\nbeen tracking on our usage graph. We can use these in tandem to
have granular control over spend.To set limits, we navigate back to Settings →
Usage and Billing → Usage configuration. There is a table at the\nbottom of th
e page that lets you set usage limits per workspace. For each workspace, the
two limits appear, along\nwith a cost estimate:Lets start by setting limits on ou
r production usage, since that is where the majority of spend comes from.Sett
```

```
ing a good total traces limit\u200bPicking the right "total traces" limit depends
on the expected load of traces that you will send to LangSmith. You should\nc
learly think about your assumptions before setting a limit.For example:Current
Load: Our gen AI application is called between 1.2-1.5 times per second, and e
ach API request has a trace associated with it,'
```

✅ Checks how well your query finds the most relevant document.

7. 🧠 Load LLM

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")
```

✅ Loads OpenAI's latest GPT model.

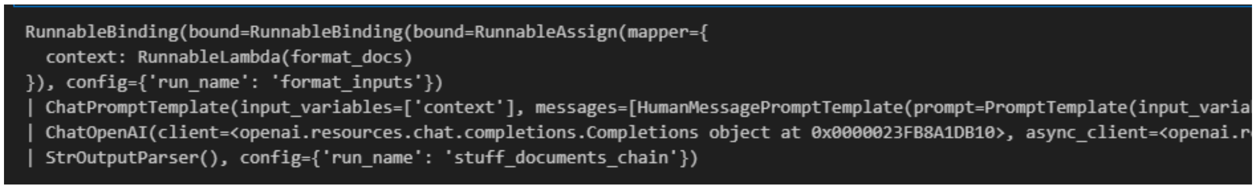
8. 💬 Define Prompt and Response Chain

```
## Retrieval Chain, Document chain

from langchain.chains.combine_documents import create_stuff_documents_c
hain
from langchain_core.prompts import ChatPromptTemplate

prompt=ChatPromptTemplate.from_template(
    """
Answer the following question based only on the provided context:
<context>
{context}
</context>
"""
)

document_chain=create_stuff_documents_chain(llm,prompt)
document_chain
```



```
RunnableBinding(bound=RunnableBinding(bound=RunnableAssign(mapper={
  context: RunnableLambda(format_docs)
}), config={'run_name': 'format_inputs'})
| ChatPromptTemplate(input_variables=['context'], messages=[HumanMessagePromptTemplate(prompt=PromptTemplate(input_varia
| ChatOpenAI(client=<openai.resources.chat.completions.Completions object at 0x0000023FB8A1DB10>, async_client=<openai.r
| StrOutputParser(), config={'run_name': 'stuff_documents_chain'})
```

`ChatPromptTemplate.` `from_template`

- **What it is:** This method is a simpler way to create a prompt template, especially useful when you have a single, coherent block of text that includes placeholders. While it's a "chat" prompt template, it's more flexible and often used when you're injecting large chunks of information (like retrieved documents) directly into the prompt.
- **Why it's used now:** In your current RAG (Retrieval-Augmented Generation) example, you're going to inject a `<context>` block directly into the prompt. It's often easier to define this entire structure as one string template rather than breaking it down into separate system/user messages, especially when the context itself is dynamic. It implicitly creates a single "user" message with the full template content.
- **Structure:** It takes a single string that contains placeholders (like `{context}` and `{input}`).

In Simple Words:

- **`from_messages` is like giving the AI a script with different characters (system, user).** You tell it who says what.
- **`from_template` is like giving the AI a fill-in-the-blanks letter.** It just fills in the missing parts of a single message.

- You choose based on whether you want to strictly define roles in a conversation or just inject dynamic content into a single, larger instruction.

`<context> {context} </context>`

For the LLM: These tags act as clear visual markers. They tell the LLM, "Hey, everything between these `<context>` and `</context>` tags is the specific information you should use to answer the upcoming question. Do *not* use your general knowledge for this part." This helps prevent the LLM from "hallucinating" (making up answers) or using outdated information.

What is `create_stuff_documents_chain(...)` ?

🔧How it works:

1. It receives a list of `Document` objects (e.g., from your `retriever`).
2. It takes the `page_content` from each `Document` and concatenates them into one long string.
3. It then takes this combined string and injects it into the `{context}` placeholder of your `prompt` template.
4. Finally, it sends this fully formed prompt to the `llm` to get an answer.

It's a type of **Document Chain** — a chain that handles:

- Taking retrieved docs
- Formatting them into a prompt
- Getting a smart response from the LLM

🔧 Internally:

- When you create a document chain using `create_stuff_documents_chain(...)`
- LangChain **automatically wraps** the output using a `StrOutputParser()`

✨ Why is it called “stuff”?

LangChain offers multiple **strategies** to feed documents to LLMs:

Strategy	Description
`stuff`	Stuff all docs into the prompt at once
`map_reduce`	Summarize each doc, then combine summaries
`refine`	Add one doc at a time, refining answer progressively

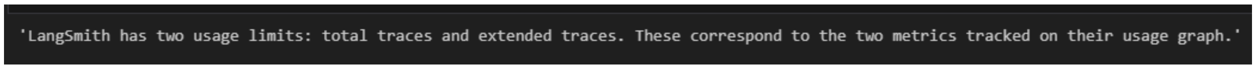
So `create_stuff_documents_chain()` is the simplest — just “stuff it all in.”

❗ When *Not* to Use Stuffing

If your docs are **too big** to all fit in one prompt (e.g., long PDFs), stuffing doesn't work well.

Instead, use `map_reduce` or `refine` strategies — or a **retriever-based RAG chain**.

```
from langchain_core.documents import Document
document_chain.invoke({
    "input":"LangSmith has two usage limits: total traces and extended",
    "context":[Document(page_content="LangSmith has two usage limits: total traces and extended traces. These correspond to the two metrics we've been tracking on our usage graph. ")]
})
```



```
'LangSmith has two usage limits: total traces and extended traces. These correspond to the two metrics tracked on their usage graph.'
```

9. 🧲 Set Up Retriever + Final Retrieval Chain

```
retriever = vectorstoredb.as_retriever()

from langchain.chains import create_retrieval_chain
retrieval_chain = create_retrieval_chain(retriever, document_chain)
```

✅ Retriever finds the most relevant docs for a query and passes them to the LLM.

What is a `Retriever`?

In LangChain, a `Retriever` is like a very smart librarian. Its job is to take a natural language question (your query) and go into the `vectorstoredb` (the library) to find the most relevant "books" or "pages" (document chunks) that might contain the answer. It doesn't answer the question itself; it just finds the relevant context.

`retrieval_chain = create_retrieval_chain(retriever, document_chain)`

This is where you combine the "Retrieval" part and the "Generation" (or "Document Answering") part into one seamless process.

- **`create_retrieval_chain`: It automates the process of connecting the document finding step with the answer generation step.**
- **`retriever` (Your Librarian):** This is the first component you pass. It's responsible for finding the relevant documents.
- **`document_chain` (Your Expert and Report Writer):** This is the second component you pass. Remember, your `document_chain` already contains:
    1. Your `ChatPromptTemplate` (with the `<context>{context}</context>` format).
    2. Your `ChatOpenAI` LLM.
    3. The automatically added `StrOutputParser`
- **Stage 1: Retrieval:** It first takes the user's `input` question and passes it to the `retriever`. The `retriever` goes to the `vectorstoredb` and fetches the most relevant `Document` objects.
- **Stage 2: Document Answering (Generation):** It then takes these *retrieved `Document` objects* and the original `input` question, and passes them both to your

`document_chain`. The `document_chain` then uses the `prompt` to formulate a query with the retrieved context for the LLM, gets the LLM's answer, and extracts the string content.

✅ `retrieval_chain` Internal Flow of this chain:

```
User Input (query) →
    🔽
Retriever.fetch_relevant_docs(query) →
    🔽
Docs injected into prompt as {context} →
    🔽
LLM answers the query using the documents →
    🔽
Final response returned
```

🔚 10. Invoke the App

```
response = retrieval_chain.invoke({"input": "LangSmith has two usage limits: total traces and extended"})
print(response['answer'])
```

✅ LLM answers the question using the most relevant documents only.

💡 **`response` (The Team's Result):** The `response` you get back from `retrieval_chain.invoke()` is a **Python dictionary**.

It contains key pieces of information from the entire process:

- `'input'`: Your original question.

- `'context'`: The actual `Document` objects that the `retriever` found and passed to the `document_chain`. (This is very useful for debugging! You can see *what* information the LLM actually received.)
- `'answer'`: This is the final, plain string answer generated by your LLM based on the retrieved context and your question.

`print(response['answer'])`

(Getting the Final Answer): This line simply accesses the `'answer'` key from the `response` dictionary and prints its value, which is the clear, concise answer from the LLM.

```
response
```

```
Output:

{'input': 'LangSmith has two usage limits: total traces and extended',
 'context': [Document(page_content='use usage limits to prevent future overs
pend.LangSmith has two usage limits: total tra,
 ...
 metadata={'source': 'https://docs.smith.langchain.com/tutorials/Administrato
rs/manage_spend', 'title': 'Optimize tracing spend on LangSmith | 🦜🛠️ LangS
mith', 'description': 'Before diving into this content, it might be helpful to read t
he following:', 'language': 'en'}),

Document(page_content="more than 10% of traces.noteIf you want to keep a
subset of traces for longer
...
...

 'answer': 'To set usage limits in LangSmith, navigate to **Settings → Usage a
nd Billing → Usage configuration**. There, you will find a table at the bottom o
f the page that allows you to set usage limits per workspace. The two types of
limits you can set are **total traces** and **extended retention traces**, each
with an associated cost estimate.'}
```

✅ Full Process Visualization:

```
Your Custom Documents (Vector DB)
        🔽
Retriever (finds best chunks)
        🔽
Document Chain (adds prompt)
        🔽
LLM (generates answer)
        🔽
   Final Output
```