

# Intro to Keeping Knowledge Organized with Indexes

Regardless of the chosen model or prompt formulation, language models have inherent limitations that cannot be resolved with the techniques we learned. These models have a **cut-off date for the training process**, which means they typically **lack access to trending news and the latest developments**. This limitation can result in the models providing responses that may not be factually accurate and potentially hallucinating information.

In this module, we will delve into techniques that enable us to provide accurate context to language models, enhancing their ability to answer questions effectively. Additional context can be sourced from various channels such as databases, URLs, or different file types. Several preprocessing steps are necessary to facilitate this process. These include utilizing splitters to ensure the content's length falls within the model's input window size and converting text into embedding vectors, which aids in identifying contextually similar resources.

- **Exploring The Role of LangChain's Indexes and Retrievers:** To kick off the module, we introduce the Deep Lake database and its seamless integration with the LangChain library. This lesson highlights the benefits of utilizing Deep Lake, including the ability to retrieve pertinent documents for contextual use. Additionally, we delve into the limitations of this approach and present solutions to overcome them.
- **Streamlined Data Ingestion: Text, PyPDF, Selenium URL Loaders, and Google Drive Sync:** The LangChain library offers a variety of helper classes designed to facilitate data loading and extraction from diverse sources. Regardless of whether the information originates from a PDF file or website content, these classes streamline the process of handling different data formats.
- **What are Text Splitters and Why They are Useful:** The length of the contents may vary depending on their source. For instance, a PDF file containing a book may exceed the input window size of the model, making it incompatible with direct processing. However, splitting the large text into smaller segments will allow us to use the most relevant chunk as the context instead of expecting the model to comprehend the whole book and answer a question. This lesson will thoroughly explore different approaches that enable us to accomplish this objective.
- **Exploring the World of Embeddings:** Embeddings are high-dimensional vectors that capture semantic information. Large language models can transform textual data into embedding space, allowing for versatile representations across languages. These embeddings serve as valuable tools to identify relevant information by quantifying the distance between data points, thereby indicating closer semantic meaning for points closer together. The LangChain integration provides necessary functions for both transforming and calculating similarities.
- **Build a Customer Support Question Answering Chatbot:** This practical example demonstrates the utilization of a website's content as supplementary context for a chatbot to respond to user queries effectively. The code implementation involves employing the mentioned data loaders, storing the corresponding embeddings in the Deep Lake dataset, and ultimately retrieving the most pertinent documents based on the user's question.
- **Conversation Intelligence: Gong.io Open-Source Alternative AI Sales Assistant:** In this lesson, we will explore how LangChain, Deep Lake, and GPT-4 can be used to develop a sales assistant able to give advice to salesman, taking into considerations internal guidelines.
- **FableForge: Creating Picture Books with OpenAI, Replicate, and Deep Lake:** In this final lesson, we are going to delve into a use case of AI technology in the creative domain of children's picture book creation in a project called "FableForge", leveraging both OpenAI GPT-3.5 LLM for writing the story and Stable Diffusion for generating images for it.

To summarize, this module will teach you how to enrich language models with additional context to improve the quality of their responses. It can eliminate issues like hallucinations. In the current module, we focus on utilizing external documents and retrieving information from databases. Furthermore, In future modules, we will explore incorporating internet search results to enable the models to answer trending questions.

# Exploring The Role of LangChain's Indexes and Retrievers

## Introduction

In LangChain, indexes and retrievers play a crucial role in structuring documents and fetching relevant data for LLMs. We will explore some of the advantages and disadvantages of using document based LLMs (i.e., LLMs that leverage relevant pieces of documents inside their prompts), with a particular focus on the role of indexes and retrievers.

An **index** is a powerful data structure that meticulously **organizes and stores documents** to enable efficient searching, while a **retriever** harnesses the index to locate **and return pertinent documents in response to user queries**.

Within LangChain, the primary index types are centered on **vector databases**, with **embeddings-based indexes** being the most prevalent.

Retrievers focus on **extracting relevant documents to merge with prompts** for language models. A retriever exposes a **get\_relevant\_documents** method, which accepts a query string as input and returns a list of related documents.

Here we use the **TextLoader** class to load a text file.

Remember to install the required packages with the following command:

```
pip install langchain==0.0.208 deeplake openai tiktoken.
```

```
from langchain.document_loaders import TextLoader
```

```
# text to write to a local file
```

```
# taken from https://www.theverge.com/2023/3/14/23639313/google-ai-language-model-palm-api-challenge-openai
```

```
text = """Google opens up its AI language model PaLM to challenge OpenAI and GPT-3
```

```
Google is offering developers access to one of its most advanced AI language models: PaLM.
```

```
The search giant is launching an API for PaLM alongside a number of AI enterprise tools it says will help businesses “generate text, images, code, videos, audio, and more from simple natural language prompts.”
```

```
PaLM is a large language model, or LLM, similar to the GPT series created by OpenAI or Meta’s LLaMA family of models. Google first announced PaLM in April 2022. Like other LLMs, PaLM is a flexible system that can potentially carry out all sorts of text generation and editing tasks. You could train PaLM to be a conversational chatbot like ChatGPT, for example, or you could use it for tasks like summarizing text or even writing code.
```

```
(It’s similar to features Google also announced today for its Workspace apps like Google Docs and Gmail.) """
```

```
# write text to local file
```

```
with open("my_file.txt", "w") as file:
```

```
    file.write(text)
```

```
# use TextLoader to load text from local file
```

```
loader = TextLoader("my_file.txt")
```

```
docs_from_file = loader.load()
```

```
print(len(docs_from_file))
```

```
# 1
```

Then, we use **CharacterTextSplitter** to split the docs into texts.

```
from langchain.text_splitter import CharacterTextSplitter
```

```
# create a text splitter
```

```
text_splitter = CharacterTextSplitter(chunk_size=200, chunk_overlap=20)
```

```
# split documents into chunks
```

```
docs = text_splitter.split_documents(docs_from_file)
```

```
print(len(docs))
```

```
# 2
```

These embeddings allow us to effectively search for documents or portions of documents that relate to our query by examining their semantic similarities.

```
from langchain.embeddings import OpenAIEmbeddings
```

```
# Before executing the following code, make sure to have
```

```
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
```

```
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
```

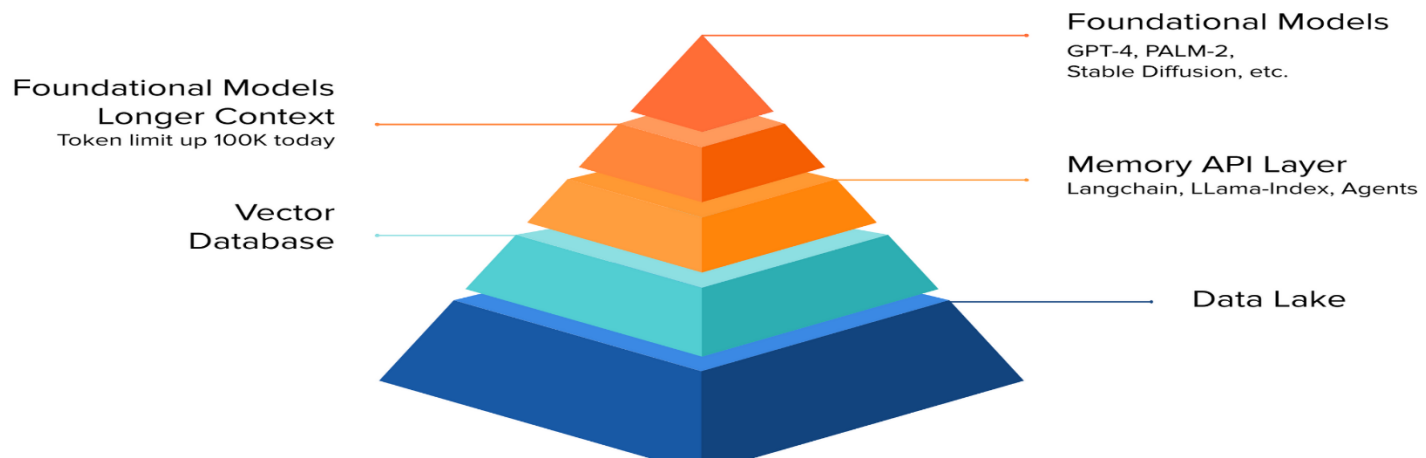
We'll employ the Deep Lake vector store with our embeddings in place.

Deep Lake provides several advantages over the typical vector store:

- It's **multimodal**, which means that it can be used to store items of diverse modalities, such as texts, images, audio, and video, along with their vector representations.
- It's **serverless**, which means that we can create and manage cloud datasets without the need to create and managing a database instance. This aspect gives a great speedup to new projects.
- It's possible to easily create a streaming **data loader** out of the data loaded into a Deep Lake dataset, which is convenient for fine-tuning machine learning models using common frameworks like PyTorch and TensorFlow.
- Data can be queried and visualized easily from the web.

Thanks to its nature, Deep Lake is well suited for being the **serverless memory** that LLM chains and agents need for several tasks, like storing relevant documents for question-answering or storing images to control some guided image-generation tasks. Here's a diagram that visually summarizes this aspect.

## Serving as **Serverless Memory** for Foundational Models



Let's create an instance of a Deep Lake dataset.

```
from langchain.vectorstores import DeepLake

# Before executing the following code, make sure to have your
# ActiveLoop key saved in the "ACTIVELOOP_TOKEN" environment variable.

# create Deep Lake dataset
# TODO: use your organization id here. (by default, org id is your username)
my_active_loop_org_id = "<YOUR-ACTIVELOOP-ORG-ID>"
my_active_loop_dataset_name = "langchain_course_indexers_retrievers"
dataset_path = f"hub://{my_active_loop_org_id}/{my_active_loop_dataset_name}"
db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)

# add documents to our Deep Lake dataset
db.add_documents(docs)
```

In this example, we are adding text documents to the dataset. However, being Deep Lake multimodal, we could have also added images to it, specifying an image embedder model. This could be useful for searching images according to a text query or using an image as a query (and thus looking for similar images).

As datasets become bigger, storing them in local memory becomes less manageable. In this example, we could have also used a local vector store, as we are uploading only two documents. However, in a typical production scenario, thousands or millions of documents could be used and accessed from different programs, thus having the need for a centralized cloud dataset. Back to the code example of this lesson. Next, we create a retriever.

```
# create retriever from db
retriever = db.as_retriever()
```

Once we have the retriever, we can start with **question-answering**.

```
from langchain.chains import RetrievalQA
from langchain.LLms import OpenAI
```

```
# create a retrieval chain
qa_chain = RetrievalQA.from_chain_type(
    Llm=OpenAI(model="text-davinci-003"),
    chain_type="stuff",
    retriever=retriever
)
```

We can query our document that is an about specific topic that can be found in the documents.

```
query = "How Google plans to challenge OpenAI?"
response = qa_chain.run(query)
print(response)
```

You should see something like the following.

Google plans to challenge OpenAI by offering access to its AI language model PaLM, which is similar to OpenAI's GPT series and Meta's LLaMA family of models. PaLM is a large language model that can be used for tasks like summarizing text or writing code.

## What occurred behind the scenes?

Initially, we employed a so-called "**stuff chain**" (refer to CombineDocuments Chains). Stuffing is one way to supply information to the LLM. Using this technique, we "**stuff**" **all the information** into the LLM's prompt. However, this method is only **effective with shorter documents**, as most LLMs have a context length limit.

**Additionally, a similarity search** is conducted using the **embeddings** to identify matching documents to be used as context for the LLM. Although it might not seem particularly useful with just one document, we are effectively working with multiple documents since we "chunked" our text. Preselecting the most suitable documents based on semantic similarity enables us to provide the model with meaningful knowledge through the prompt while remaining within the allowed context size.

So, in this exploration, we have discovered the significant role that indexes and retrievers play in improving the performance of Large Language Models when handling document-based data.

The system becomes more efficient in finding and presenting relevant information by converting documents and user queries into numerical vectors (embeddings) and storing them in specialized databases like Deep Lake, which serves as our vector store database.

The retriever's ability to identify documents that are closely related to a user's query in the embedding space demonstrates the effectiveness of this approach in enhancing the overall language understanding capabilities of LLMs.

## A Potential Problem

This method has a downside: you might **not know how to get the right documents** later when storing data. In the Q&A example, we cut the text into equal parts, causing both useful and useless text to show up when a user asks a question.

Including unrelated information in the LLM prompt is **detrimental** because:

1. It can divert the LLM's focus from pertinent details.
2. It occupies valuable space that could be utilized for more relevant information.

## Possible Solution

A **DocumentCompressor** abstraction has been introduced to address this issue, allowing **compress\_documents** on the **retrieved documents**.

The **ContextualCompressionRetriever** is a wrapper around another retriever in LangChain. It takes a **base retriever** and a **DocumentCompressor** and automatically **compresses the retrieved documents from the base retriever**. This means that only the **most relevant parts of the retrieved documents are returned, given a specific query**.

A popular compressor choice is the **LLMChainExtractor**, which uses an LLMChain to extract only the statements relevant to the query from the documents. To improve the retrieval process, a **ContextualCompressionRetriever** is used, wrapping the base retriever with an **LLMChainExtractor**. The LLMChainExtractor **iterates over the initially returned documents and extracts only the content relevant** to the query.

Here's an example of how to use **ContextualCompressionRetriever** with **LLMChainExtractor**:

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor

# create GPT3 wrapper
Llm = OpenAI(model="text-davinci-003", temperature=0)

# create compressor for the retriever
compressor = LLMChainExtractor.from_llm(Llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=retriever )
```

Once we have created the `compression_retriever`, we can use it to retrieve the compressed relevant documents to a query.

```
# retrieving compressed documents

retrieved_docs = compression_retriever.get_relevant_documents(
    "How Google plans to challenge OpenAI?" )

print(retrieved_docs[0].page_content)
```

You should see an output like the following.

Google **is** offering developers access to one of its most advanced AI language models: PaLM. The search giant **is** launching an API **for** PaLM alongside a number of AI enterprise tools it says will **help** businesses “generate text, images, code, videos, audio, **and** more **from** simple natural language prompts.”

Compressors aim to make it easy to pass **only** the relevant information to the LLM. Doing this also enables you to pass along **more** information to the LLM since in the initial retrieval step, you can focus on recall (e.g., by increasing the number of documents returned) and let the compressors handle precision:

## Contextual compression | [🔗 Langchain](#)

One challenge with retrieval is that usually you don't know the specific queries your document storage system will face when you ingest data into the system. This means that the information most relevant to a query may be buried in a document with a lot of irrelevant text. Passing that full document through your application can lead to more expensive LLM calls and poorer responses.  
[python.langchain.com](https://python.langchain.com)

## Conclusion

In summary, LangChain's indexes and retrievers offer modular, flexible, and customizable solutions for working with unstructured data and language models. However, they have limited support for structured data and are mainly focused on vector databases. In the next lesson, we'll see some convenient LangChain classes for loading data from different sources, that is data loaders.

## RESOURCES:

### Improving Document Retrieval with Contextual Compression

Note: This post assumes some familiarity with LangChain and is moderately technical.

💡 TL;DR: We've introduced a new abstraction and a new document Retriever to facilitate the post-processing of retrieved documents. Specifically, the new abstraction makes it easy to take a set of retrieved documents and extract from them  
[blog.langchain.dev](https://blog.langchain.dev)



# Streamlined Data Ingestion: Text, PyPDF, Selenium URL Loaders, and Google Drive Sync

## Introduction

The TextLoader handles plain text files, while the PyPDFLoader specializes in PDF files, offering easy access to content and metadata. SeleniumURLLoader is designed for loading HTML documents from URLs that require JavaScript rendering. Lastly, the Google Drive Loader provides seamless integration with Google Drive, allowing for the import of data from Google Docs or folders.

## TextLoader

Import the LangChain and necessary loaders from `langchain.document_loaders`.

Remember to install the required packages with the following command:

```
pip install langchain==0.0.208 deeplake openai tiktoken.
```

```
from langchain.document_loaders import TextLoader
```

```
loader = TextLoader('file_path.txt')
```

```
documents = loader.load()
```

The sample code.

```
[Document(page_content='<FILE_CONTENT>', metadata={'source': 'file_path.txt'})]
```

The output.

You can use the `encoding` argument to change the encoding type. (For example: `encoding="ISO-8859-1"`)

## PyPDFLoader (PDF)

The LangChain library provides two methods for loading and processing PDF files: `PyPDFLoader` and `PDFMinerLoader`. We mainly focus on the former, which is used to load PDF files into an array of documents, where each document contains the page content and metadata with the page number. First, install the package using Python Package Manager (PIP).

```
!pip install -q pypdfCopy
```

Here's a code snippet to load and split a PDF file using PyPDFLoader:

```
from langchain.document_loaders import PyPDFLoader
```

```
loader = PyPDFLoader("example_data/layout-parser-paper.pdf")
```

```
pages = loader.load_and_split()
```

```
print(pages[0])
```



The sample code.

```
Document(page_content='<PDF_CONTENT>', metadata={'source':  
'/home/cloudsuperadmin/scrape-chain/langchain/deep_learning_for_nlp.pdf', 'page': 0})
```

The output.

Using PyPDFLoader offers advantages such as simple, straightforward usage and easy access to page content and metadata, like page numbers, in a structured format. However, it has disadvantages, including limited text extraction capabilities compared to PDFMinerLoader.

## SeleniumURLLoader (URL)

The **SeleniumURLLoader** module offers a robust yet user-friendly approach for loading HTML documents from a list of URLs requiring JavaScript rendering. Here is a guide and example for using this class which starts by installing the package using the Python Package Manager (PIP). The codes has been tested for unstructured and selenium libraries with 0.7.7 and 4.10.0, respectively. However, feel free to install the latest versions.

```
!pip install -q unstructured seleniumCopy
```

Instantiate the **SeleniumURLLoader** class by providing a list of URLs to load, for example:

```
from langchain.document_loaders import SeleniumURLLoader
```

```
urls = [  
    "https://www.youtube.com/watch?v=TFa539R09EQ&t=139s",  
    "https://www.youtube.com/watch?v=6Zv6A_9urh4&t=112s" ]
```

```
loader = SeleniumURLLoader(urls=urls)
```

```
data = loader.load()
```

```
print(data[0])
```

```
Document(page_content="OPENASSISTANT TAKES ON CHATGPT!\n\nInfo\n\nShopping\n\nWatch later\n\nShare\n\nCopy link\n\nTap to unmute\n\nIf playback doesn't begin shortly, try restarting your device.\n\nYou're signed out\n\nVideos you watch may be added to the TV's watch history and influence TV recommendations. To avoid this, cancel and sign in to YouTube on your computer.\n\nUp next\n\nLiveUpcoming\n\nPlay Now\n\nMachine Learning Street Talk\n\nSubscribe\n\nSubscribed\n\nSwitch camera\n\nShare\n\nAn error occurred while retrieving sharing information. Please try again later.\n\n2:19\n\n2:19 / 59:51\n\nWatch full video\n\n•\n\nScroll for details\n\nNew!\n\nWatch ads now so you can enjoy fewer interruptions\n\nGot it\n\nAbout\n\nPress\n\nCopyright\n\nContact us\n\nCreators\n\nAdvertise\n\nDevelopers\n\nTerms\n\nPrivacy\n\nPolicy & Safety\n\nHow YouTube works\n\nTest new features\n\nNFL Sunday Ticket\n\n© 2023 Google LLC", metadata={'source': 'https://www.youtube.com/watch?v=TFa539R09EQ&t=139s'})
```

The SeleniumURLLoader class includes the following attributes:

- **URLs** (List[str]): List of URLs to load.
- **continue\_on\_failure** (bool, default=True): Continues loading other URLs on failure if True.
- **browser** (str, default="chrome"): Browser selection, either 'Chrome' or 'Firefox'.
- **executable\_path** (Optional[str], default=None): Browser executable path.
- **headless** (bool, default=True): Browser runs in headless mode if True.

Customize these attributes during SeleniumURLLoader instance initialization, such as using Firefox instead of Chrome by setting the browser to "firefox":

```
Loader = SeleniumURLLoader(urls=urls, browser="firefox")Copy
```

Upon invoking the load() method, a list of Document instances containing the loaded content is returned. Each Document instance includes a page\_content attribute with the extracted text from the HTML and a metadata attribute containing the source URL.

Bear in mind that SeleniumURLLoader may be slower than other loaders since it initializes a browser instance for each URL. Nevertheless, it is advantageous for loading pages necessitating JavaScript rendering.

This approach **will not work in Google Colab** environment without further configuration which is not in the scope of this course. Try running the code directly using the Python interpreter.

## Google Drive loader

The LangChain Google Drive Loader efficiently imports data from Google Drive by using the `GoogleDriveLoader` class. It can fetch data from a list of Google Docs document IDs or a single folder ID.

Prepare necessary credentials and tokens:

- By default, the GoogleDriveLoader searches for the credentials.json file in ~/.credentials/credentials.json. Use the `credentials_file` keyword argument to modify this path.
- The token.json file follows the same principle and will be created automatically upon the loader's first use.

### To set up the credentials\_file, follow these steps:

1. Create a new Google Cloud Platform project or use an existing one by visiting the Google Cloud Console. Ensure that billing is enabled for your project.
2. Enable the Google Drive API by navigating to its dashboard in the Google Cloud Console and clicking "Enable."
3. Create a service account by going to the Service Accounts page in the Google Cloud Console. Follow the prompts to set up a new service account.
4. Assign necessary roles to the service account, such as "Google Drive API - Drive File Access" and "Google Drive API - Drive Metadata Read/Write Access," depending on your needs.
5. After creating the service account, access the "Actions" menu next to it, select "Manage keys," click "Add Key," and choose "JSON" as the key type. This generates a JSON key file and downloads it to your computer, which serves as your credentials\_file.

Retrieve the folder or document ID from the URL:

- Folder: [https://drive.google.com/drive/u/0/folders/{folder\\_id}](https://drive.google.com/drive/u/0/folders/{folder_id})
- Document: [https://docs.google.com/document/d/{document\\_id}/edit](https://docs.google.com/document/d/{document_id}/edit)

Import the GoogleDriveLoader class:

```
from langchain.document_loaders import GoogleDriveLoader
```

Instantiate GoogleDriveLoader:

```
loader = GoogleDriveLoader(  
    folder_id="your_folder_id",  
    recursive=False # Optional: Fetch files from subfolders recursively. Defaults to  
    False.)
```

Load the documents:

```
docs = loader.load()
```

Note that currently, only Google Docs are supported.

## Conclusion

In conclusion, the process of streamlined data ingestion has been significantly simplified with the integration of various powerful loaders, including TextLoader, PyPDFLoader, SeleniumURLLoader, and Google Drive Loader. Each of these tools caters to specific file types and data sources, ensuring efficient and comprehensive data management.

In the next lesson, we'll learn about common ways of splitting texts into smaller chunks, so that they can easily be inserted into prompts with limited tokens size.

# What are Text Splitters and Why They are Useful

## Introduction

Large Language Models, while recognized for creating human-like text, **can also "hallucinate"** and produce seemingly plausible yet incorrect or nonsensical information. Interestingly, this tendency can be advantageous in creative tasks, as it generates a range of unique and imaginative ideas, sparking new perspectives and driving the creative process. However, this poses a challenge in situations where accuracy is critical, such as code reviews, insurance-related tasks, or research question responses.

One approach to **mitigating hallucination is to provide documents as sources of information** to the LLM and ask it to generate an answer based on the knowledge extracted from the document. This can help **reduce the likelihood of hallucination**, and users can verify the information with the source document.

Let's discuss the pros and cons of this approach:

### Pros:

1. **Reduced hallucination:** By providing a source document, the LLM is more likely to generate content based on the given information, reducing the chances of creating false or irrelevant information.
2. **Increased accuracy:** With a reliable source document, the LLM can generate more accurate answers, especially in use cases where accuracy is crucial.
3. **Verifiable information:** Users can cross-check the generated content with the source document to ensure the information is accurate and reliable.

### Cons:

1. **Limited scope:** Relying on a single document may limit the scope of the generated content, as the LLM will only have access to the information provided in the document.
2. **Dependence on document quality:** The accuracy of the generated content heavily depends on the quality and reliability of the source document. The LLM will likely generate incorrect or misleading content if the document contains inaccurate or biased information.
3. **Inability to eliminate hallucination completely:** Although providing a document as a base reduces the chances of hallucination, it does not guarantee that the LLM will never generate false or irrelevant information.

Addressing another challenge, LLMs have **a maximum prompt size**, preventing them from feeding entire documents. This makes it crucial to divide documents into smaller parts, and Text Splitters prove to be extremely useful in achieving this.

Text Splitters help **break down large text documents into smaller**, more digestible pieces that language models can process more effectively.

Using a Text Splitter can also **improve vector store search results**, as smaller segments might be more likely to match a query. Experimenting with different chunk sizes and overlaps can be beneficial in tailoring results to suit your specific needs.

## Customizing Text Splitter

When handling lengthy pieces of text, it's crucial to break them down into manageable chunks. This seemingly simple task can quickly become complex, as keeping semantically related text segments intact is essential. The definition of "semantically related" may vary depending on the type of text. In this article, we'll explore various strategies to achieve this.

At a high level, text splitters follow these steps:

1. **Divide the text into small**, semantically meaningful chunks (often sentences).
2. **Combine these small chunks** into a larger one until a specific size is reached (determined by a particular function).
3. Once the desired size is attained, **separate that chunk as an individual piece of text**, then **start forming a new chunk** with some **overlap** to **maintain context** between segments.

Consequently, there are two primary dimensions to consider when customizing your text splitter:

- The **method** used to split the text
- The **approach** for measuring chunk size

## Character Text Splitter

This type of splitter can be used in various scenarios where you must split long text pieces into smaller, semantically meaningful chunks. For example, you might use it to split a long article into smaller chunks for easier processing or analysis. The splitter allows you to customize the chunking process along two axes - **chunk size and chunk overlap** - to balance the trade-offs between splitting the text into manageable pieces and preserving semantic context between chunks.

Load the documents using the `PyPDFLoader` class.

You need to install the `pypdf` package using Python Package Manager. (`pip install -q pypdf`)

Remember to install also the required packages with the following command:

```
pip install langchain==0.0.208 deeplake openai tiktoken.
```

(You can download a sample PDF file from the following link or use any PDF file that you have)

[The One Page Linux Manual.pdf](#)94.3KB

```
from langchain.document_loaders import PyPDFLoader
loader = PyPDFLoader("The One Page Linux Manual.pdf")
pages = loader.load_and_split()
```

By loading the text file, we can ask more specific questions related to the subject, which helps minimize the likelihood of LLM hallucinations and ensures more accurate, context-driven responses.

```
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=20)
texts = text_splitter.split_documents(pages)

print(texts[0])
print(f"You have {len(texts)} documents")
print("Preview:")
print(texts[0].page_content)
```

The sample code.

```
page_content='THE ONE      PAGE LINUX MANUALA summary of useful Linux
commands\nVersion 3.0 May 1999 squadron@powerup.com.au\nStarting & Stopping\nshutdown
-h now Shutdown the system now and do not\nreboot\nhalt Stop all processes - same as
above\nshutdown -r 5 Shutdown the system in 5 minutes and\nreboot\nshutdown -r now
Shutdown the system now and reboot\nreboot Stop all processes and then reboot -
same\nas above\nstartx Start the X system\nAccessing & mounting file systems\nmount -
t iso9660 /dev/cdrom\n/mnt/cdromMount the device cdrom\nand call it cdrom under
the\n/mnt directory\nmount -t msdos /dev/hdd\n/mnt/ddriveMount hard disk “d” as
a\nmsdos ...' metadata={'source': 'The One Page Linux Manual.pdf', 'page': 0}
```

You have 2 documents

Preview:

THE ONE PAGE LINUX MANUALA summary of useful Linux commands

Version 3.0 May 1999 squadron@powerup.com.au

Starting & Stopping

shutdown -h now Shutdown the system now and do not  
reboot

halt Stop all processes - same as above

shutdown -r 5 Shutdown the system in 5 minutes and  
reboot

shutdown -r now Shutdown the system now and reboot  
reboot Stop all processes and then reboot - same

as above

startx Start the X system

Accessing & mounting file systems

mount -t iso9660 /dev/cdrom

...

The output.

**No universal approach** for chunking text will fit all scenarios - what's effective for one case might not be suitable for another. Finding the **best chunk size** for your project means going through a few steps.

First, **clean up your data** by getting rid of anything that's not needed, like HTML tags from websites. Then, **pick a few different chunk sizes** to test. The best size will depend on what kind of data you're working with and the model you're using. Finally, **test how well each size works** by running some queries and comparing the results. You might need to try a few different sizes before finding the best one. This process might take some time, but getting the best results from your project is worth it.

## Recursive Character Text Splitter

The Recursive Character Text Splitter is a text splitter designed to split the text into chunks based on a list of characters provided. It attempts to **split text using the characters from a list in order until the resulting chunks are small enough**. By default, the list of characters used for splitting is `["\n\n", "\n", " ", ""]`, which tries to keep paragraphs, sentences, and words together as long as possible, as they are generally the most semantically related pieces of text. This means that the class first tries to split the text into two new-line characters. If the resulting chunks are still larger than the desired chunk size, it will then try to split the output by a single new-line character, followed by a space character, and so on, until the desired chunk size is achieved.

To use the `RecursiveCharacterTextSplitter`, you can create an instance of it and provide the following parameters:

**chunk\_size** : The maximum size of the chunks, as measured by the `length_function` (default is 100).

**chunk\_overlap**: The maximum overlap between chunks to maintain continuity between them (default is 20).

**length\_function**: parameter is used to calculate the length of the chunks. By default, it is set to `len`, which counts the number of characters in a chunk. However, you can also pass a token counter or any other function that calculates the length of a chunk based on your specific requirements.

Using a token counter instead of the default `len` function can benefit specific scenarios, such as when working with language models with token limits. For example, OpenAI's GPT-3 has a token limit of 4096 tokens per request, so you might want to count tokens instead of characters to better manage and optimize your requests.

Here's an example of how to use `RecursiveCharacterTextSplitter`.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
# Load a Long document
```

```
with open('The One Page Linux Manual.pdf', encoding= 'unicode_escape') as f:
    sample_text = f.read()
```

Create an instance of the `RecursiveCharacterTextSplitter` class with the desired parameters. The default list of characters to split by is `["\n\n", "\n", " ", ""]`.

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=50,
    chunk_overlap=10,
    length_function=len,)
```

The text is first split by two new-line characters (`\n\n`). Then, since the chunks are still larger than the desired chunk size (50), the class tries to split the output by a single new-line character (`\n`). The final output consists of two chunks, each with a length of 50 characters **or less**.

```
texts = text_splitter.create_documents([sample_text])
print(texts)
```



The sample code.

```
[Document(page_content='Building LLM applications for production', metadata={}),
Document(page_content='Apr 11, 2023 \x95 Chip Huyen text', metadata={}),
Document(page_content='A question that I\x92ve been asked a lot recently is',
metadata={}),
Document(page_content='is how large language models (LLMs) will change',
metadata={}),
Document(page_content='change machine learning workflows. After working',
metadata={}),
Document(page_content='working with several companies who are working', metadata={}),
Document(page_content='working with LLM applications and personally going',
metadata={})]Copy
```

The output.

In this example, the text is loaded from a file, and the `RecursiveCharacterTextSplitter` is used to split it into chunks with a maximum size of 50 characters and an overlap of 10 characters. The output will be a list of documents containing the split text.

To use a token counter, you can create a custom function that calculates the number of tokens in a given text and pass it as the `length_function` parameter. This will ensure that your text splitter calculates the length of chunks based on the number of tokens instead of the number of characters. The exploration of this concept will be part of our upcoming lessons.

## NLTK Text Splitter

The `NLTKTextSplitter` in LangChain is an implementation of a text splitter that uses the Natural Language Toolkit (**NLTK**) library to split text based on tokenizers. The goal is to split long texts into smaller chunks without breaking the structure of sentences and paragraphs.

If it is your first time using this package, it is required to install the `NLTK` library using `pip install -q nltk` and run the following Python code to download the packages that LangChain needs.

```
import nltk; nltk.download('punkt');

# Load a long document
with open('/home/cloudsuperadmin/scrape-chain/Langchain/LLM.txt', encoding='unicode_escape') as f:
    sample_text = f.read()

from langchain.text_splitter import NLTKTextSplitter
text_splitter = NLTKTextSplitter(chunk_size=500)
texts = text_splitter.split_text(sample_text)
print(texts)
```

[ 'Building LLM applications for production\nApr 11, 2023 \x95 Chip Huyen text \n\nA question that I\x92ve has been asked a lot recently is how large language models (LLMs) will change machine learning workflows.\n\nAfter working with several companies who are working with LLM applications and personally going down a rabbit hole building my applications, I realized two things:\n\nIt\x92s easy to make something cool with LLMs, but very hard to make something production-ready with them.', 'LLM limitations are exacerbated by a lack of engineering rigor in prompt engineering, partially due to the ambiguous nature of natural languages, and partially due to the nascent nature of the field.\n\nThis post consists of three parts .\n\nPart 1 discusses the key challenges of productionizing LLM applications and the solutions that I\x92ve seen.\n\nPart 2[...]

However, as mentioned in your context, the NLTKTextSplitter is **not specifically designed to handle word segmentation in English sentences without spaces**. For this purpose, you can use alternative libraries like **pyenchant** or **word segment**.

## SpacyTextSplitter

The `SpacyTextSplitter` helps split large text documents into smaller chunks **based on a specified size**. This is useful for better management **of large text inputs**. It's important to note that the `SpacyTextSplitter` is an alternative to NLTK-based sentence splitting. You can create a `SpacyTextSplitter` object by specifying the `chunk_size` parameter, measured by a length function passed to it, which defaults to the number of characters.

```
from langchain.text_splitter import SpacyTextSplitter

# Load a long document
with open('/home/cloudsuperadmin/scrape-chain/langchain/LLM.txt', encoding='unicode_escape') as f:
    sample_text = f.read()

# Instantiate the SpacyTextSplitter with the desired chunk size
text_splitter = SpacyTextSplitter(chunk_size=500, chunk_overlap=20)

# Split the text using SpacyTextSplitter
texts = text_splitter.split_text(sample_text)

# Print the first chunk
print(texts[0])
```

The sample code.

Building LLM applications for production

Apr 11, 2023 • Chip Huyen text

A question that I've been asked a lot recently is how large language models (LLMs) will change machine learning workflows.

After working with several companies who are working with LLM applications and personally going down a rabbit hole building my applications, I realized two things:

It's easy to make something cool with LLMs, but very hard to make something production-ready with them.

The output.

## MarkdownTextSplitter

The `MarkdownTextSplitter` is designed to split text written **using Markdown languages** like headers, code blocks, or dividers. It is implemented as a simple **subclass of `RecursiveCharacterSplitter` with Markdown-specific separators**. By default, these separators are determined by the Markdown syntax, but they can be customized by providing a list of characters during the initialization of the `MarkdownTextSplitter` instance. The chunk size, which is initially set to the number of characters, is measured by the length function passed in. To customize the chunk size, provide an integer value when initializing an instance.

```
from langchain.text_splitter import MarkdownTextSplitter

markdown_text = """
#
# Welcome to My Blog!
## Introduction

Hello everyone! My name is John Doe and I am a _software developer_. I specialize
in Python, Java, and JavaScript.

Here's a list of my favorite programming languages:

1. Python
2. JavaScript
3. Java

You can check out some of my projects on [GitHub](https://github.com).

## About this Blog

In this blog, I will share my journey as a software developer. I'll post tutorials,
my thoughts on the latest technology trends, and occasional book reviews.

Here's a small piece of Python code to say hello:

``` python
def say_hello(name):
    print(f"Hello, {name}!")
```

```

say_hello("John")
\``
Stay tuned for more updates!
## Contact Me
Feel free to reach out to me on [Twitter](https://twitter.com) or send me an email at
johndoe@email.com.      ""
markdown_splitter = MarkdownTextSplitter(chunk_size=100, chunk_overlap=0)
docs = markdown_splitter.create_documents([markdown_text])
print(docs)

```

The sample code.

```

[Document(page_content='# \n\n# Welcome to My Blog!', metadata={}),
Document(page_content='Introduction', metadata={}),

Document(page_content='Hello everyone! My name is **John Doe** and I am a _software
developer_. I specialize in Python,', metadata={}),

Document(page_content='Java, and JavaScript.', metadata={}), Document(page_content="Here's a
list of my favorite programming languages:\n\n1. Python\n2. JavaScript\n3. Java",
metadata={}),

Document(page_content='You can check out some of my projects on
[GitHub](https://github.com).', metadata={}),

Document(page_content='About this Blog', metadata={}),

Document(page_content="In this blog, I will share my journey as a software developer. I'll
post tutorials, my thoughts on", metadata={}),

Document(page_content='the latest technology trends, and occasional book reviews.',
metadata={}),

Document(page_content="Here's a small piece of Python code to say hello:", metadata={}),
Document(page_content='\`\`\`python\ndef say_hello(name):\n    print(f"Hello,
{name}!")\n\nsay_hello("John")\n\`\`\'', metadata={}),

Document(page_content='Stay tuned for more updates!', metadata={}),
Document(page_content='Contact Me', metadata={}),

Document(page_content='Feel free to reach out to me on [Twitter](https://twitter.com) or send
me an email at', metadata={}),

Document(page_content='johndoe@email.com.', metadata={})]

```

The output.

The MarkdownTextSplitter offers a practical solution for dividing text while preserving the structure and meaning provided by Markdown formatting. By recognizing the Markdown syntax (e.g., headings, lists, and code blocks), you can intelligently divide the content based on its structure and hierarchy, resulting in more semantically coherent chunks. This splitter is especially valuable when managing extensive Markdown documents.

## TokenTextSplitter

The main advantage of using `TokenTextSplitter` over other text splitters, like `CharacterTextSplitter`, is that it **respects the token boundaries**, ensuring that the **chunks do not split tokens in the middle**. This can be particularly helpful in maintaining the semantic integrity of the text when working with language models and embeddings.

This type of splitter breaks down raw text strings into smaller pieces by initially converting the **text into BPE (Byte Pair Encoding) tokens**, and subsequently **dividing these tokens into chunks**. It then reassembles the tokens within each chunk back into text.

The `tiktoken` python package is required for using this class. (`pip install -q tiktoken`)

```
from langchain.text_splitter import TokenTextSplitter

# Load a long document

with open('/home/cloudsuperadmin/scrape-chain/langchain/LLM.txt', encoding='unicode_escape') as f:
    sample_text = f.read()

# Initialize the TokenTextSplitter with desired chunk size and overlap
text_splitter = TokenTextSplitter(chunk_size=100, chunk_overlap=50)

# Split into smaller chunks
texts = text_splitter.split_text(sample_text)
print(texts[0])
```

The sample code.

Building LLM applications for production

Apr 11, 2023 • Chip Huyen text

A question that I've been asked a lot recently is how large language models (LLMs) will change machine learning workflows. After working with several companies who are working with LLM applications and personally going down a rabbit hole building my applications, I realized two things:

It's easy to make something cool with LLMs, but very hard to make something with production.

The output.

The `chunk_size` parameter sets the maximum number of BPE tokens in each chunk, while `chunk_overlap` defines the number of overlapping tokens between adjacent chunks. By modifying these parameters, you can fine-tune the granularity of the text chunks.

One **potential drawback of using `TokenTextSplitter`** is that it may require **additional computation** when converting text to BPE tokens and back.

If you need a **faster and simpler text-splitting** method, you might consider using **CharacterTextSplitter**, which directly splits the text based on character count, offering a more straightforward approach to text segmentation.

## RECAP:

Text splitters are essential for managing long text, improving language model processing efficiency, and enhancing vector store search results. Customizing text splitters involves selecting the splitting method and measuring chunk size.

CharacterTextSplitter is an example that helps balance manageable pieces and semantic context preservation. Experimenting with different chunk sizes and overlaps tailor the results for specific use cases.

RecursiveCharacterTextSplitter focuses on preserving semantic relationships while offering customizable chunk sizes and overlaps.

NLTKTextSplitter utilizes the Natural Language Toolkit library for more accurate text segmentation. SpacyTextSplitter leverages the popular SpaCy library to split texts based on linguistic features. MarkdownTextSplitter is tailored for Markdown-formatted texts, ensuring content is split meaningfully according to the syntax. Lastly, TokenTextSplitter employs BPE tokens for splitting, offering a fine-grained approach to text segmentation.

## Conclusion

Selecting the appropriate text splitter depends on the specific requirements and nature of the text you are working with, ensuring optimal results for your text processing tasks.

In the next lesson, we'll learn more about how word embeddings work and how embedding models are used with indexers in LangChain.

## RESOURCES:

### [Split by character | 📄 Langchain](#)

This is the simplest method. This splits based on characters (by default "\n\n") and measure chunk length by number of characters.

[python.langchain.com](https://python.langchain.com)

### [Split code | 📄 Langchain](#)

CodeTextSplitter allows you to split your code with multiple language support. Import enum Language and specify the language.

[python.langchain.com](https://python.langchain.com)

### [Recursively split by character | 📄 Langchain](#)

This text splitter is the recommended one for generic text. It is parameterized by a list of characters. It tries to split on them in order until the chunks are small enough. The default list is ["\n\n", "\n", " ", ""]. This has the effect of trying to keep all paragraphs (and then sentences, and then words) together as long as possible, as those would generically seem to be the strongest semantically related pieces of text.

[python.langchain.com](https://python.langchain.com)

You can find the code of this lesson in this online [Notebook](#).

# Exploring the World of Embeddings

## Introduction

Vector embeddings are among the most intriguing and beneficial aspects of machine learning, playing a pivotal role in many natural language processing, recommendation, and search algorithms. If you've interacted with recommendation engines, voice assistants, or language translators, you've engaged with systems that utilize embeddings.

**Embeddings** are **dense vector representations of data** that encapsulate semantic information, making them suitable for various machine-learning tasks such as clustering, recommendation, and classification. They transform human-perceived semantic similarity into closeness in vector space and can be generated for different data types, including text, images, and audio.

For **text data**, models like the **GPT family of models and Llama** are employed to create vector embeddings for words, sentences, or paragraphs. In the case **of images**, **convolutional neural networks (CNNs)** such as **VGG and Inception** can generate embeddings. **Audio recordings** can be converted into **vectors using image embedding** techniques applied to visual representations of **audio frequencies, like spectrograms**. Deep neural networks are commonly employed to train models that convert objects into vectors. The resulting embeddings are typically high-dimensional and dense.

Embeddings are extensively used in similarity search applications, such as KNN and ANN, which require calculating distances between vectors to determine similarity. Nearest neighbor search can be employed for tasks like de-duplication, recommendations, anomaly detection, and reverse image search.

## Similarity search and vector embeddings

OpenAI offers a powerful language model called GPT-3, which can be used for various tasks, such as generating embeddings and performing similarity searches. In this example, we'll use the OpenAI API to generate embeddings for a set of documents and then perform a similarity search using cosine similarity.

First, let's install the required packages with the following command:

```
pip install langchain==0.0.208 deeplake openai tiktoken scikit-learn.
```

Next, create an API key from the OpenAI website and set it as an environment variable:

```
export OPENAI_API_KEY="your-api-key"Copy
```

Let's generate embeddings for our documents and perform a similarity search:

```
import openai
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from langchain.embeddings import OpenAIEmbeddings

# Define the documents
```



```

documents = [
    "The cat is on the mat.",
    "There is a cat on the mat.",
    "The dog is in the yard.",
    "There is a dog in the yard.",
]

# Initialize the OpenAIEmbeddings instance
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# Generate embeddings for the documents
document_embeddings = embeddings.embed_documents(documents)

# Perform a similarity search for a given query
query = "A cat is sitting on a mat."
query_embedding = embeddings.embed_query(query)

# Calculate similarity scores
similarity_scores = cosine_similarity([query_embedding], document_embeddings)[0]

# Find the most similar document
most_similar_index = np.argmax(similarity_scores)
most_similar_document = documents[most_similar_index]

print(f"Most similar document to the query '{query}':")
print(most_similar_document)

```

# the output:

Most similar document to the query 'A cat is sitting on a mat.':

The cat is on the mat

We initialize the OpenAI API client by setting the OpenAI API key. This allows us to use OpenAI's services for generating embeddings.

We then define a list of documents as strings. These documents are the text data we want to analyze for semantic similarity.

In order to perform this analysis, we need to convert our documents into a format that our similarity computation algorithm can understand. This is where `OpenAIEmbeddings` class comes in.

We use it to **generate embeddings for each document**, transforming them into vectors that represent their semantic content.

Similarly, we also **transform our query string into an embedding**. The query string is the text we want to find the most similar document too.

With our documents and query now in the form of embeddings, we **compute the cosine similarity between the query embedding and each document embedding**. The cosine similarity is a metric used to determine how similar two vectors are. In our case, it gives us a list of similarity scores for our query against each document.

*With our similarity scores in hand, we then **identify the document most similar** to our query. We do this by finding the index of the **highest similarity score and retrieving the corresponding document** from our list of documents.*

***Embedding vectors** positioned near each other are regarded as similar. At times, they are directly applied to display related items in online shops. In other instances, they are incorporated into various models to share insights across akin items rather than considering them as entirely distinct entities. This renders embeddings effective in representing aspects like web browsing patterns, textual data, and e-commerce transactions for subsequent model applications.*

## Embedding Models

**Embedding models** are a type of machine learning model that convert discrete data into continuous vectors. In the context of natural language processing, these discrete data points can be words, sentences, or even entire documents. The generated vectors, also known as embeddings, are designed to capture the semantic meaning of the original data.

For instance, words that are semantically similar (e.g., 'cat' and 'kitten') would have similar embeddings. These embeddings are dense, which means that they use many dimensions (often hundreds) to capture nuances in meaning.

The primary benefit of embeddings is that they **allow us to use mathematical operations** to reason about **semantic meaning**. For example, we can calculate the cosine similarity between two embeddings to assess how semantically similar the corresponding words or documents are. We initialize our embedding model.

For this task, we've chosen the pre-trained "**sentence-transformers/all-mpnet-base-v2**" model. This model is designed to transform sentences into embeddings - vectors that encapsulate the semantic meaning of the sentences. The `model_kwargs` parameter is used here to specify that we want our computations to be performed on the CPU.

```
from langchain.llms import HuggingFacePipeline
from langchain.embeddings import HuggingFaceEmbeddings

model_name = "sentence-transformers/all-mpnet-base-v2"
model_kwargs = {'device': 'cpu'}
hf = HuggingFaceEmbeddings(model_name=model_name, model_kwargs=model_kwargs)
documents = ["Document 1", "Document 2", "Document 3"]
doc_embeddings = hf.embed_documents(documents)Copy
```

Now that we have our model, we define a list of documents - these are the pieces of text that we want to convert into semantic embeddings.

With our model and documents ready, we move on to generate the embeddings. We do this by calling the `embed_documents` method on our HuggingFaceEmbeddings instance, passing our list of documents as an argument. This method processes each document and returns a corresponding list of embeddings.

These embeddings are now ready for any downstream tasks such as classification, clustering, or similarity analysis. They represent our original documents in a form that machines can understand and process, enabling us to perform complex semantic tasks.

## Cohere embeddings

Cohere is dedicated to making its innovative **multilingual language models** accessible to all, thereby democratizing advanced NLP technologies worldwide. Their Multilingual Model, which maps text into a semantic vector space for better text similarity understanding, significantly **enhances multilingual applications such as search operations**. Unlike their **English language model**, the multilingual model uses **dot product computations** resulting in superior performance.

These multilingual embeddings are represented in a **768-dimensional vector space**.

To activate the power of the Cohere API, one needs to acquire an API key. Here's a step-by-step guide to doing so:

1. Visit the [Cohere Dashboard](#).
2. If you haven't already, you must either log in or sign up for a Cohere account. Please note that you agree to adhere to the Terms of Use and Privacy Policy by signing up.
3. When you're logged in, the dashboard provides an intuitive interface to create and manage your API keys. Once we have the API key, we initialize an instance of the CohereEmbeddings class within LangChain, specifying the "**embed-multilingual-v2.0**" model.

We then specify a list of texts in various languages. The `embed_documents()` method is subsequently invoked to generate unique embeddings for each text in the list.

To illustrate the results, we print each text alongside its corresponding embedding. For simplicity, we only display the first 5 dimensions of each embedding. You also need to install the cohere package by running the following command `pip install cohere`.

```
import cohere

from langchain.embeddings import CohereEmbeddings

# Initialize the CohereEmbeddings object
cohere = CohereEmbeddings(
    model="embed-multilingual-v2.0",
    cohere_api_key="your_cohere_api_key" )

# Define a list of texts
texts = [
    "Hello from Cohere!",
```

```
"□□□□□□ □□ □□□□□!",  
"Hallo von Cohere!",  
"Bonjour de Cohere!",  
"¡Hola desde Cohere!",  
"Olá do Cohere!",  
"Ciao da Cohere!",  
"您好, 来自 Cohere!",  
"कोहरे से नमस्ते!" ]
```

```
# Generate embeddings for the texts
```

```
document_embeddings = cohere.embed_documents(texts)
```

```
# Print the embeddings
```

```
for text, embedding in zip(texts, document_embeddings):
```

```
    print(f"Text: {text}")
```

```
    print(f"Embedding: {embedding[:5]}") # print first 5 dimensions of each embedding
```

Your output should be similar to the following.

Text: Hello from Cohere!

Embedding: [0.23439695, 0.50120056, -0.048770234, 0.13988855, -0.1800725]

Text: !كوهير من مرحبًا

Embedding: [0.25350592, 0.29968268, 0.010332941, 0.12572688, -0.18180023]

Text: Hallo von Cohere!

Embedding: [0.10278442, 0.2838264, -0.05107267, 0.23759139, -0.07176493]

Text: Bonjour de Cohere!

Embedding: [0.15180704, 0.28215882, -0.056877363, 0.117460854, -0.044658754]

Text: ¡Hola desde Cohere!

Embedding: [0.2516583, 0.43137372, -0.08623046, 0.24681088, -0.11645193]

Text: Olá do Cohere!

Embedding: [0.18696906, 0.39113742, -0.046254586, 0.14583701, -0.11280365]

Text: Ciao da Cohere!

Embedding: [0.1157251, 0.43330532, -0.025885003, 0.14538017, 0.07029742]

Text:您好, 来自 Cohere !

Embedding: [0.24605744, 0.3085744, -0.11160592, 0.266223, -0.051633865]

Text: कोहरे से नमस्ते!

Embedding: [0.19287698, 0.6350239, 0.032287907, 0.11751755, -0.2598813]

LangChain, a comprehensive library designed for language understanding and processing, serves as an ideal conduit for Cohere's advanced language models. It simplifies the integration of Cohere's multilingual embeddings into a developer's workflow, thus enabling a broader range of applications, from semantic search to customer feedback analysis and content moderation, across a multitude of languages.

When used in tandem with Cohere, LangChain eliminates the need for complex pipelines, making the process of generating and manipulating high-dimensional embeddings straightforward and efficient. Given a list of multilingual texts, the `embed_documents()` method in LangChain's CohereEmbeddings class, connected to Cohere's embedding endpoint, can swiftly generate unique semantic embeddings for each text.

## Deep Lake Vector Store

**Vector stores** are data structures or databases designed to store and manage high-dimensional vectors efficiently. They enable efficient similarity search, nearest neighbor search, and other vector-related operations. Vector stores can be built using various data structures such as approximate nearest neighbor (ANN) techniques, KD trees, or Vantage Point trees.

**Deep Lake**, serves as both a data lake for deep learning and a multi-modal vector store. As a **multi-modal vector store**, it allows users to **store images, audio, videos, text, and metadata** in a format **optimized for deep learning**. It enables hybrid search, allowing users to search both embeddings and their attributes.

Users can **save data locally, in their cloud, or on Activeloop storage**. Deep Lake supports the training of **PyTorch and TensorFlow models** while streaming data with minimal boilerplate code. It also provides features like version control, dataset queries, and distributed workloads using a simple Python API.

Moreover, as the size of datasets increases, it becomes increasingly difficult to store them in local memory. A local vector store could have been utilized in this particular instance since only a few documents are being uploaded. However, the necessity for a centralized cloud dataset arises in a typical production setting, where thousands or millions of documents may be involved and accessed by various programs.

Let's see how to use Deep Lake for our example.

## Creating Deep Lake Vector Store embeddings example

Deep Lake provides well-written documentation, and besides other examples for which they added Jupyter Notebooks, we can follow the one for vector store creation.

This task aims to leverage the power of NLP technologies, particularly OpenAI and Deep Lake, to generate and manipulate high-dimensional embeddings. These embeddings can be used for a variety of purposes, such as searching for relevant documents, moderating content, and answering questions. In this case, we will create a Deep Lake database for a retrieval-based question-answering system.

First, we need to import the required packages and ensure that the Activeloop and OpenAI keys are stored in the environment variables, `ACTIVELOOP_TOKEN` and `OPENAI_API_KEY`. Getting `ACTIVELOOP_TOKEN` is straightforward, you can easily generate one on the Activeloop page.

The installation of the `deeplake` library using `pip`, and the initialization of the OpenAI and Activeloop API keys:

```
pip install deeplakeCopy
```

Then make sure to specify the right API keys in the “`OPENAI_API_KEY`” and “`ACTIVELOOP_TOKEN`” environmental variables.

Next, the necessary modules from the `langchain` package are imported.

```
from langchain.embeddings.openai import OpenAIEmbeddings  
from langchain.vectorstores import DeepLake  
from langchain.text_splitter import RecursiveCharacterTextSplitter  
from langchain.chat_models import ChatOpenAI  
from langchain.chains import RetrievalQACopy
```

We then create some documents using the `RecursiveCharacterTextSplitter` class.

```
# create our documents
```

```
texts = [  
    "Napoleon Bonaparte was born in 15 August 1769",  
    "Louis XIV was born in 5 September 1638",  
    "Lady Gaga was born in 28 March 1986",  
    "Michael Jeffrey Jordan was born in 17 February 1963"  
]  
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)  
docs = text_splitter.create_documents(texts)Copy
```

The next step is to create a Deep Lake database and load our documents into it.

```
# initialize embeddings model
```

```
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
```

```
# create Deep Lake dataset
```

```
# TODO: use your organization id here. (by default, org id is your username)
```

```

my_activeloop_org_id = "<YOUR-ACTIVELoop-ORG-ID>"
my_activeloop_dataset_name = "Langchain_course_embeddings"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"
db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)
# add documents to our Deep Lake dataset
db.add_documents(docs)

```

*If everything worked correctly, you should see a printed output like this:*

Your Deep Lake dataset has been successfully created!

The dataset [is](#) private so make sure you are logged [in](#)!

We now create a retriever from the database.

```

# create retriever from db
retriever = db.as_retriever()

```

Finally, we create a **RetrievalQA** chain in LangChain and run it

```

# instantiate the LLM wrapper
model = ChatOpenAI(model='gpt-3.5-turbo')

# create the question-answering chain
qa_chain = RetrievalQA.from_LLM(model, retriever=retriever)

# ask a question to the chain
qa_chain.run("When was Michael Jordan born?")

```

This returns:

```
'Michael Jordan was born on 17 February 1963.'
```

This pipeline demonstrates how to leverage the power of the LangChain, OpenAI, and Deep Lake libraries and products to create a conversational AI model capable of retrieving and answering questions based on the content of a given repository.

Let's break down each step to understand how these technologies work together.

1. **OpenAI and LangChain Integration:** LangChain, a library built for chaining NLP models, is designed to work seamlessly with OpenAI's GPT-3.5-turbo model for language understanding and generation. You've initialized OpenAI embeddings using **OpenAIEmbeddings()**, and these embeddings are later used to transform the text into a high-dimensional vector representation. This vector representation captures the semantic essence of the text and is essential for information retrieval tasks.
2. **Deep Lake:** Deep Lake is a Vector Store for creating, storing, and querying vector representations (also known as embeddings) of data.



3. **Text Retrieval:** Using the `db.as_retriever()` function, you've transformed the Deep Lake dataset into a retriever object. This object is designed to fetch the most relevant pieces of text from the dataset based on the semantic similarity of their embeddings.
4. **Question Answering:** The final step involves setting up a `RetrievalQA` chain from LangChain. This chain is designed to accept a natural language question, transform it into an embedding, retrieve the most relevant document chunks from the Deep Lake dataset, and generate a natural language answer. The `ChatOpenAI` model, which is the underlying model of this chain, is responsible for both the question embedding and the answer generation.

## Conclusion

In conclusion, vector embeddings are a cornerstone in capturing and understanding the rich contextual information in our textual data. This representation becomes increasingly important when dealing with language models like GPT-3.5-turbo, which have a limited token capacity.

In this tutorial, we've used **embeddings from OpenAI** and incorporated embeddings from **Hugging Face and Cohere**. The former, a well-known AI research organization, provides Transformer-based models that are highly versatile and widely used. **Cohere offers innovative multilingual language models** that are a significant asset in a globally interconnected world.

Building upon these technologies, we've walked through the **process of creating a conversational AI application, specifically a Q&A system leveraging Deep Lake**. This application demonstrates the potential of these combined technologies - LangChain for chaining together complex NLP tasks, Hugging Face, Cohere, and OpenAI for generating high-quality embeddings, and Deep Lake for managing these embeddings in a vector store.

In the next lesson we'll build a customer support question-answering chatbot leveraging our new knowledge about indexes and retrievers.

# Build a Customer Support Question Answering Chatbot

## Introduction

As we witness accelerated technological progress, large language models like GPT-4 and ChatGPT have emerged as significant breakthroughs in the tech landscape. These state-of-the-art models demonstrate exceptional prowess in content generation. However, they are not without their share of challenges, such as **biases and hallucinations**. Despite these limitations, LLMs have the potential to bring about a transformative impact on chatbot development.

Traditional, primarily intent-based chatbots have been designed to respond to specific user intents. These intents comprise a collection of sample questions and corresponding responses. For instance, a "Restaurant Recommendations" intent might include sample questions like "Can you suggest a good Italian restaurant nearby?" or "Where can I find the best sushi in town?" with responses such as "You can try the Italian restaurant 'La Trattoria' nearby" or "The top-rated sushi place in town is 'Sushi Palace.'"

When users interact with the chatbot, their **queries are matched to the most similar intent, generating the associated response**. However, as LLMs continue to evolve, chatbot development is **shifting toward more sophisticated and dynamic solutions capable** of handling a broader range of user inquiries with greater precision and nuance.

## Having a Knowledge Base

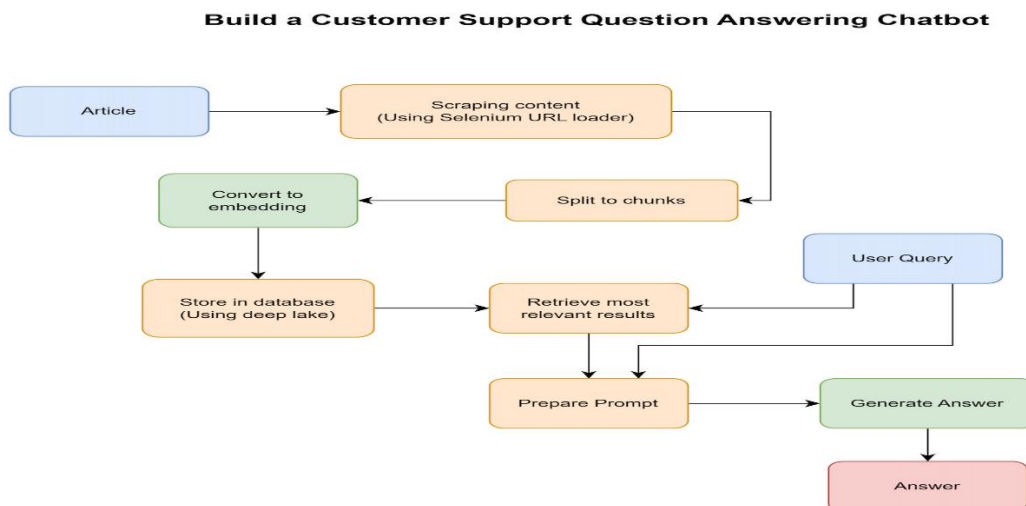
LLMs can significantly enhance chatbot functionality by associating broader intents with documents from a Knowledge Base (KB) instead of specific questions and answers. This approach streamlines intent management and generates more tailored responses to user inquiries.

GPT3 has a maximum prompt size of around 4,000 tokens, which is substantial but insufficient for incorporating an entire knowledge base in a single prompt.

Future LLMs may not have this limitation while retaining the text generation capabilities. However, for now, we need to design a solution around it.

## Workflow

This project aims to build a chatbot that leverages GPT3 to search for answers within documents..



First we scrape some content from online articles, we split them into small chunks, compute their embeddings and store them in Deep Lake. Then, we use a user query to retrieve the most relevant chunks from Deep Lake, we put them into a prompt, which will be used to generate the final answer by the LLM.

It is important to note that there is always a risk of **generating hallucinations or false information** when using LLMs. Although this might not be acceptable for many customers support use cases, the chatbot can still be helpful for assisting operators in drafting answers that they can double-check before sending them to the user.

In the next steps, we'll explore how to manage conversations with GPT-3 and provide examples to demonstrate the effectiveness of this workflow:

First, set up the `OPENAI_API_KEY` and `ACTIVELOOP_TOKEN` environment variables with your API keys and tokens.

As we're going to use the `SeleniumURLLoader` LangChain class, and it uses the `unstructured` and `selenium` Python library, let's install it using `pip`. It is recommend to install the latest version of the library. Nonetheless, please be aware that the code has been tested specifically on version `0.7.7`.

```
pip install unstructured selenium
```

Remember to install the required packages with the following command:

```
pip install langchain==0.0.208 deeplake openai tiktoken
```

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import DeepLake
from langchain.text_splitter import CharacterTextSplitter
from langchain import OpenAI
from langchain.document_loaders import SeleniumURLLoader
from langchain import PromptTemplate
```

These libraries provide functionality for handling OpenAI embeddings, managing vector storage, splitting text, and interacting with the OpenAI API. They also enable the creation of a context-aware question-answering system, incorporating retrieval and text generation.

The database for our chatbot will consist of articles regarding technical issues.

```
# we'll use information from the following articles
```

```
urls = ['https://beebom.com/what-is-nft-explained/',
        'https://beebom.com/how-delete-spotify-account/',
        'https://beebom.com/how-download-gif-twitter/',
        'https://beebom.com/how-use-chatgpt-linux-terminal/',
        'https://beebom.com/how-delete-spotify-account/',
        'https://beebom.com/how-save-instagram-story-with-music/',
        'https://beebom.com/how-install-pip-windows/',
        'https://beebom.com/how-check-disk-usage-linux/']
```

## 1: Split the documents into chunks and compute their embeddings

We load the documents from the provided URLs and split them into chunks using the `CharacterTextSplitter` with a chunk size of 1000 and no overlap:

```
# use the selenium scraper to load the documents
loader = SeleniumURLLoader(urls=urls)
docs_not_split = loader.load()

# we split the documents into smaller chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
docs = text_splitter.split_documents(docs_not_split)
```

Next, we compute the embeddings using `OpenAIEmbeddings` and store them in a Deep Lake vector store on the cloud. In an ideal production scenario, we could upload a whole website or course lesson on a Deep Lake dataset, allowing for search among even thousands or millions of documents. As we are using a cloud serverless Deep Lake dataset, applications running on different locations can easily access the same centralized dataset without the need of deploying a vector store on a custom machine.

Let's now modify the following code by adding your ActiveLoop organization ID. It worth noting that the org id is your username by default.

```
# Before executing the following code, make sure to have
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# create Deep Lake dataset
# TODO: use your organization id here. (by default, org id is your username)
my_active_loop_org_id = "<YOUR-ACTIVELOOP-ORG-ID>"
my_active_loop_dataset_name = "Langchain_course_customer_support"
dataset_path = f"hub://{my_active_loop_org_id}/{my_active_loop_dataset_name}"
db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)

# add documents to our Deep Lake dataset
db.add_documents(docs)
```

To retrieve the most similar chunks to a given query, we can use the `similarity_search` method of the Deep Lake vector store:

```
# Let's see the top relevant documents to a specific query
query = "how to check disk usage in linux?"
```

```
docs = db.similarity_search(query)
print(docs[0].page_content)
```

The previous code will show something like the following output.

Home How To How to Check Disk Usage [in](#) Linux (4 Methods)

How to Check Disk Usage [in](#) Linux (4 Methods)

Beebom Staff

Last Updated: February 21, 2023 3:15 pm

There may be times when you need to download some important files [or](#) transfer some photos to your Linux system, but face a problem of insufficient disk space. You head over to your [file](#) manager to delete the large files which you no longer require, but you have no clue which of them are occupying most of your disk space. In this article, we will show some easy methods to check disk usage [in](#) Linux [from](#) both the terminal [and](#) the GUI application.

Monitor Disk Usage [in](#) Linux (2023)

Table of Contents

Check Disk Space Using the df Command

Display Disk Usage [in](#) Human Readable FormatDisplay Disk Occupancy of a Particular Type

Check Disk Usage using the du Command

Display Disk Usage [in](#) Human Readable FormatDisplay Disk Usage [for](#) a Particular DirectoryCompare Disk Usage of Two Directories

## 2: Craft a prompt for GPT-3 using the suggested strategies

We will create a prompt template that incorporates role-prompting, relevant Knowledge Base information, and the user's question:

```
# Let's write a prompt for a customer support chatbot that
```

```
# answer questions using information extracted from our db
```

```
template = """"You are an exceptional customer support chatbot that gently answer questions.
```

```
You know the following context information.
```

```
{chunks_formatted}
```

```
Answer to the following question from a customer. Use only information from the previous context information. Do not invent stuff.
```

```
Question: {query}
```

```
Answer: """"
```

```
prompt = PromptTemplate(
    input_variables=["chunks_formatted", "query"],    template=template )
```

The template sets the chatbot's persona as an exceptional customer support chatbot. The template takes two input variables: `chunks_formatted`, which consists of the pre-formatted chunks from articles, and `query`, representing the customer's question. The objective is to generate an accurate answer using only the provided chunks without creating any false or invented information.

### 3: Utilize the GPT3 model with a temperature of 0 for text generation

To generate a response, we first retrieve the top-k (e.g., top-3) chunks most similar to the user query, format the prompt, & send formatted prompt to the GPT3 model with a temperature of 0.

```
# the full pipeline
# user question
query = "How to check disk usage in linux?"

# retrieve relevant chunks
docs = db.similarity_search(query)
retrieved_chunks = [doc.page_content for doc in docs]

# format the prompt
chunks_formatted = "\n\n".join(retrieved_chunks)
prompt_formatted = prompt.format(chunks_formatted=chunks_formatted, query=query)

# generate answer
Llm = OpenAI(model="text-davinci-003", temperature=0)
answer = Llm(prompt_formatted)
print(answer)
```

The output:

You can check disk usage in Linux using the `df` command to check disk space and the `du` command to check disk usage. You can also use the GUI application to check disk usage in a human readable format. For more information, please refer to the article "How to Check Disk Usage in Linux (4 Methods)" on Beebom.

## Issues with Generating Answers using GPT-3

In the previous example, the chatbot generally performs well. However, there are certain situations where it could fail.

Suppose we ask, "Is the Linux distribution free?" and provide GPT-3 with a document about kernel features as context. It might generate an answer like "Yes, the Linux distribution is free to

download and use," even if such information is not present in the context document. **Producing false information** is highly undesirable for customer service chatbots!

GPT-3 is less likely to generate false information when the answer to the user's question is contained within the context. Since user questions are often brief and ambiguous, **we cannot always rely on the semantic search step** to retrieve the correct document. Thus, there is always a risk of generating false information.

## Conclusion

GPT-3 is highly effective in creating conversational chatbots capable of answering specific questions based on the contextual information provided in the prompt. However, it can be **challenging to ensure that the model generates answers solely based on the context**, as it has a **tendency to hallucinate** (i.e., generate new, potentially false information). The severity of generating false information varies depending on the use case.

To conclude, we implemented a context-aware question-answering system using LangChain, following the provided code and strategies. The process involved splitting documents into chunks, computing their embeddings, implementing a retriever to find similar chunks, crafting a prompt for GPT-3, and using the GPT3 model for text generation. This approach demonstrates the potential of leveraging GPT-3 to create powerful and contextually accurate chatbots while also highlighting the need to be cautious about the possibility of generating false information.

In the next lesson, you'll see how to leverage Deep Lake and GPT-4 for building a sales assistant.

## Sources:

### [Learn Prompting: Your Guide to Communicating with AI](#)

Learn Prompting is the largest and most comprehensive course in prompt engineering available on the internet, with over 60 content modules, translated into 9 languages, and a thriving community.

[learnprompting.org](https://learnprompting.org)



# Conversation Intelligence: Gong.io

## Open-Source Alternative AI Sales Assistant

In this lesson we will explore how LangChain, Deep Lake, and GPT-4 can be used to develop a sales assistant able to give advice to salesman, taking into considerations internal guidelines.

### Introduction

This particular article provides an in-depth view of a sales call assistant that connects you to a chatbot that understands the context of your conversation. A great feature of SalesCopilot is its ability to detect potential objections from customers and deliver recommendations on how best to address them.

The article is a journey that reveals the challenges faced and the solutions discovered during the creation of a the project. You'll learn about the two distinct text-splitting approaches that didn't work and how these failures paved the way for an effective solution.

Firstly, the authors tried to rely solely on the LLM, but they encountered issues such as response inconsistency and slow response times with GPT-4. Secondly, they naively split the custom knowledge base into chunks, but this approach led to context misalignment and inefficient results. After these unsuccessful attempts, a more intelligent way of splitting the knowledge base based on its structure was adopted. This change greatly improved the quality of responses and ensured better context grounding for LLM responses. This process is explained in detail, helping you grasp how to navigate similar challenges in your own AI projects.

Next, the article explores how SalesCopilot was integrated with Deep Lake. This integration enhanced SalesCopilot's capabilities by retrieving the most relevant responses from a custom knowledge base, thereby creating a persistent, efficient, and highly adaptable solution for handling customer objections.

By the end of this lesson, you'll learn how to utilize LLMs, how to intelligently split your knowledge base, and integrate it with a vector database like Deep Lake for optimal performance.

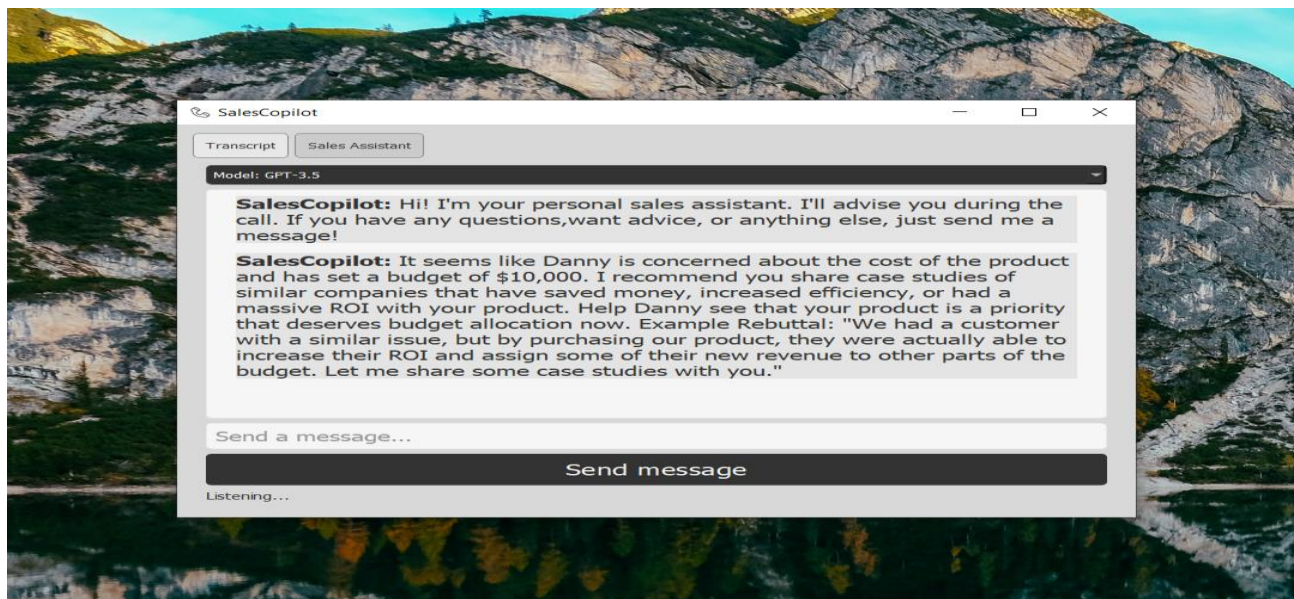
LangChain & Vector DBs: 60+ lessons & projects in our course. [Enroll for free](#)

SalesCopilot: Enhancing AI Sales Via Custom Knowledge Base Integration, Deep Lake, & LangChain

### What is SalesCopilot?

**SalesCopilot** is a sales call assistant that transcribes audio in real-time and connects the user to a chatbot with full knowledge of the transcript, powered by GPT-3.5 or GPT-4. This live chat allows for highly relevant assistance to be provided within seconds upon the user's request.

Additionally, SalesCopilot is able to detect potential objections from the customer (e.g. "It's too expensive" or "The product doesn't work for us") and provide well-informed recommendations to the salesperson on how best to handle them. Relying solely on the LLM to come up with these recommendations has some flaws - ChatGPT isn't fine tuned to be a great salesperson, and it may give recommendations that don't align with your personal approach. Integrating it with Deep Lake and a custom knowledge base is the perfect solution - let's dive into how it works!



## What Did and Didn't Work while Building an Open-Source Conversational Intelligence Assistant

Before we look at the exact solution we eventually decided on, let's take a glance at the approaches that didn't work, and what we learned from them:

### Didn't Work: No Custom Knowledge Base

This was the first solution we tried - **instead of using a custom knowledge base, we could completely rely on the LLM.**

Unfortunately, we ran into some issues:

- **GPT-4 is awesome, but way too slow:** To get the highest quality responses without a custom knowledge base, using GPT-4 is the best choice. However, the time it takes for the API to return a response means that by the time the user gets the advice they need, it's too late. This means we have to use GPT-3.5, which comes with a noticeable drop in quality.
- **Response quality is inconsistent:** Relying solely on the LLM, sometimes we get great responses that are exactly what we're looking for. However, without any additional info, guidelines, or domain-specific information, we also get bad responses that aren't very on-topic.
- **Cramming information into the prompt is not token-efficient:** Using the OpenAI API, cost is a consideration, as you pay for the amount of tokens in your prompt + the completion. To ground the LLM and keep the responses high-quality, we could fill the prompt with tons of relevant information, how we'd like it to respond in different situations, etc. This solution isn't ideal, because it means every time we query the LLM we have to pass all that information, and all those tokens, to the API. The costs can add up, and GPT-3.5 can get confused if you give it too much info at once.

That didn't work - the main issue is that we need a way to efficiently ground the LLM's response. The next thing we tried was to use a custom knowledge base combined with a vector database to pass the LLM relevant info for each individual customer objection.

### Didn't Work: Naively Splitting the Custom Knowledge Base

To leverage our custom knowledge base, our initial approach was to **split the knowledge base into chunks of equal length using LangChain's built-in text splitters**. Then we took the detected customer objection, embedded it, and searched the database for those chunks that were most relevant. This allowed us to pass relevant excerpts from our knowledge base to the LLM every time we wanted a response, which improved the quality of the responses and made the prompts to the LLM shorter and more efficient. However, our **"naive" approach to splitting the custom knowledge base had a major flaw.**

To illustrate the issue we faced, let's look at an example. Say we have the following text:

Objection: "There's no money."

It could be that your prospect's business simply isn't big enough or generating enough cash right now to afford a product like yours. Track their growth and see how you can help your prospect get to a place where your offering would fit into their business.

3

4Objection: "We don't have any budget left this year."

5A variation of the "no money" objection, what your prospect's telling you here is that they're having cash flow issues. But if there's a pressing problem, it needs to get solved eventually. Either help your prospect secure a budget from executives to buy now or arrange a follow-up call for when they expect funding to return.

6

7Objection: "We need to use that budget somewhere else."

8Prospects sometimes try to earmark resources for other uses. It's your job to make your product/service a priority that deserves budget allocation now. Share case studies of similar companies that have saved money, increased efficiency, or had a massive ROI with you.

If we naively split this text, we might end up with individual sections that look like this:

1A variation of the "no money" objection, what your prospect's telling you here is that they're having cash flow issues. But if there's a pressing problem, it needs to get solved eventually. Either help your prospect secure a budget from executives to buy now or arrange a follow-up call for when they expect funding to return.

2

3Objection: "We need to use that budget somewhere else."

Here, we see that the advice does not match the objection. When we try to retrieve the most relevant chunk for the objection "We need to use that budget somewhere else", this will likely be our top result, which isn't what we want. When we pass it to the LLM, it might be confusing.

What we really need to do is split the text in a more sophisticated way, that maintains semantic boundaries between each chunk. This will improve retrieval performance and keep the LLM responses higher quality.

## Did Work: Intelligent Splitting

In our example text, there is a set structure to each individual objection and its recommended response. Rather than split the text based on size, why don't we split the text based on its structure? We want each chunk to begin with the objection, and end before the "Objection" of the next chunk.

Here's how we could do it:

```
text = ""
```

```
Objection: "There's no money."
```

```
It could be that your prospect's business simply isn't big enough or generating enough cash right now to afford a product like yours. Track their growth and see how you can help your prospect get to a place where your offering would fit into their business.
```

```
Objection: "We don't have any budget left this year."
```

```
A variation of the "no money" objection, what your prospect's telling you here is that they're having cash flow issues. But if there's a pressing problem, it needs to get solved eventually. Either help your prospect secure a budget from executives to buy now or arrange a follow-up call for when they expect funding to return.
```

```
Objection: "We need to use that budget somewhere else."
```

```
Prospects sometimes try to earmark resources for other uses. It's your job to make your product/service a priority that deserves budget allocation now. Share case studies of similar companies that have saved money, increased efficiency, or had a massive ROI with you.
```

```
""
```

```
# Split the text into a list using the keyword "Objection: "
```

```
objections_list = text.split("Objection: ")[1:] # We ignore the first split as it is empty
```

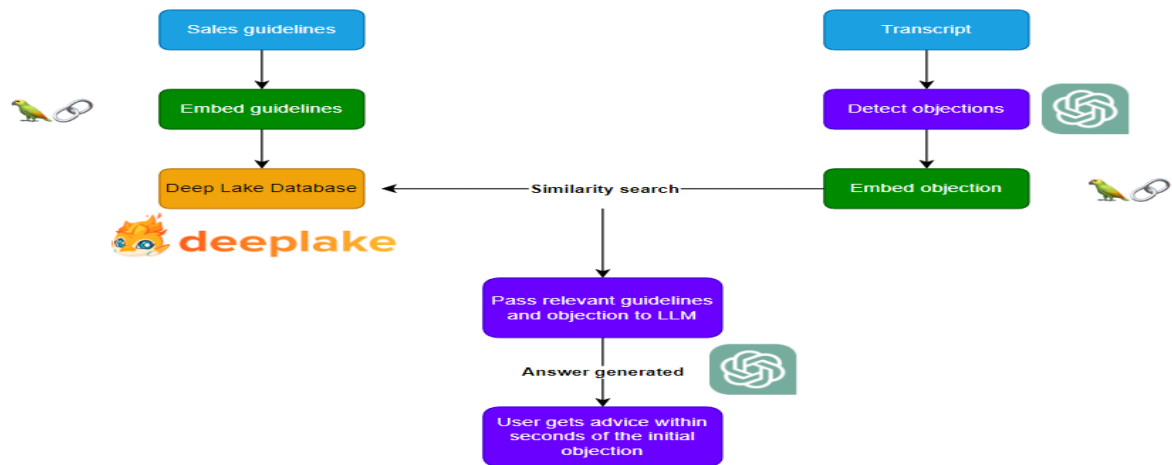
```
# Now, prepend "Objection: " to each item as splitting removed it
```

```
objections_list = ["Objection: " + objection for objection in objections_list]
```

This gave us the best results. Nailing the way we split and embed our knowledge base means more relevant documents are retrieved and the LLM gets the best possible context to generate a response from. Now let's see how we integrated this solution with Deep Lake and SalesCopilot!

## Integrating SalesCopilot with Deep Lake

By using Deep Lake as a vector database, we can quickly and easily retrieve only the most relevant info to provide to the LLM. We can also persist the vector database, so we don't have to re-create it every time we load the app. The knowledge base we're using here is [this list of common customer objections](#). Before we get into the code, here's a rough overview of how it works:



First, we take our knowledge base and embed it, storing the embeddings in a Deep Lake vector database. Then, when we detect an objection in the transcript, we embed the objection and use it to search our database, retrieving the most similar guidelines. We then pass those guidelines along with the objection to the LLM and send the result to the user.

## Creating, Loading, and Querying Our Database for AI

Define a class that handles the database creation, database loading, and database querying.

```

1import os
2import re
3from langchain.embeddings import OpenAIEmbeddings
4from langchain.vectorstores import DeepLake
5
6class DeepLakeLoader:
7    def __init__(self, source_data_path):
8        self.source_data_path = source_data_path
9        self.file_name = os.path.basename(source_data_path) # What we'll name our database
10        self.data = self.split_data()
11        if self.check_if_db_exists():
12            self.db = self.load_db()
13        else:
14            self.db = self.create_db()
  
```

There's a few things happening here. First, the data is being processed by a method called `split_data`:

```

1def split_data(self):
2    """
3    Preprocess the data by splitting it into passages.
4    If using a different data source, this function will need to be modified.
5    Returns:
6        split_data (list): List of passages. """
  
```

```

10 with open(self.source_data_path, 'r') as f:
11     content = f.read()
12 split_data = re.split(r'(?=\d+\. )', content)
13 if split_data[0] == "":
14     split_data.pop(0)
15 split_data = [entry for entry in split_data if len(entry) >= 30]
16 return split_data

```

Since we know the structure of our knowledge base, we use this method to split it into individual entries, each representing an example of a customer objection. When we run our similarity search using the detected customer objection, this will improve the results, as outlined above.

After preprocessing the data, we check if we've already created a database for this data. One of the great things about Deep Lake is that it provides us with persistent storage, so we only need to create the database once. If you restart the app, the database doesn't disappear!

Creating and loading the database is super easy:

```

1 def load_db(self):
2     """
3     Load the database if it already exists.
4     Returns:
5     DeepLake: DeepLake object. """
6     return DeepLake(dataset_path=f'deeplake/{self.file_name}', embedding_function=OpenAIEmbeddings(), read_only=True)
7
10 def create_db(self):
11     """
12     Create the database if it does not already exist.
13     Databases are stored in the deeplake directory.
14     Returns:
15     DeepLake: DeepLake object. """
16     return DeepLake.from_texts(self.data, OpenAIEmbeddings(), dataset_path=f'deeplake/{self.file_name}')

```

Just like that, our knowledge base becomes a vector database that we can now query.

```

1 def query_db(self, query):
2     """
3     Query the database for passages that are similar to the query.
4     Args:
5     query (str): Query string.
6     Returns:
7     content (list): List of passages that are similar to the query. """
8     results = self.db.similarity_search(query, k=3)
9     content = []
10    for result in results:
11        content.append(result.page_content)
12    return content

```

We don't want the metadata to be passed to the LLM, so we take the results of our similarity search and pull just the content from them. And that's it! We now have our custom knowledge base stored in a Deep Lake vector database and ready to be queried!

## Connecting Our Database to GPT-4

Now, all we need to do is connect our LLM to the database. First, we need to create a DeepLakeLoader instance with the path to our data.

```
db = DeepLakeLoader('data/salestesting.txt')
```

Next, we take the detected objection and use it to query the database:

```
results = db.query_db(detected_objection)
```

To have our LLM generate a message based off these results and the objection, we use [LangChain](#). In this example, we use a placeholder for the prompt - if you want to check out the prompts used in SalesCopilot, check out the [prompts.py file](#).

```
1from langchain.chat_models import ChatOpenAI
2from langchain.schema import SystemMessage, HumanMessage, AIMessage
3
4chat = ChatOpenAI()
5system_message = SystemMessage(content=objection_prompt)
6human_message = HumanMessage(content=f'Customer objection: {detected_objection} | Relevant guidelines:
{results}')
7
8response = chat([system_message, human_message])
```

To print the response:

```
1print(response.content)
```

And we're done! In just a few lines of code, we've got our response from the LLM, informed by our own custom knowledge base.

If you want to check out the full code for SalesCopilot, [click here](#) to visit the GitHub repo.

## Leveraging Custom Knowledge Bases with Deep Lake

Integrating SalesCopilot with Deep Lake allows for a significant enhancement of its capabilities, with immediate and relevant responses based on a custom knowledge base. The beauty of this solution is its adaptability. You can curate your knowledge base according to your own unique sales techniques and customer scenarios, ensuring SalesCopilot's responses are perfectly suited for your situation.

An efficient vector storage solution is essential to working with large knowledge bases and connecting them to LLM's, allowing the LLM to offer knowledgeable, situation-specific advice. On top of that, Deep Lake's persistent storage means we only create the database once, which saves computational resources and time.

In conclusion, the integration of SalesCopilot with Deep Lake creates a powerful tool that combines the speed and intelligence of LLM's with the rapid, precise information retrieval of a vector database. This hybrid system offers a highly adaptable, efficient, and effective solution to handling customer objections. The efficiency and simplicity Deep Lake brings to applications like this alongside its seamless integration make it a top choice for [vector storage solutions](#).

- [AI Story Generator: OpenAI Function Calling, LangChain, & Stable Diffusion](#) on Jun 19, 2023 [Blog](#)
- [How to Compare Large Language Models: GPT-4 & 3.5 vs Anthropic Claude vs Cohere](#) on Jun 9, 2023

In the next lesson, we'll see how to leverage both LLMs and image generation models to write a creative children's picture book.



# FableForge: Creating Picture Books with OpenAI, Replicate, and Deep Lake

In this lesson, we are going to delve into a use case of AI technology in the creative domain of children's picture book creation in a project called "FableForge", leveraging both **OpenAI GPT-3.5 LLM for writing the story** and **Stable Diffusion for generating images** for it.

## Introduction

This lesson's project, FableForge, is an application that **generates picture books from a single text prompt**. It utilizes the power of OpenAI's language model **GPT-3.5 to write the story**. Then, the **text is transformed into visual prompts for Stable Diffusion**, an AI that creates corresponding images, resulting in a complete picture book. The data, both text and images, are then stored in a Deep Lake dataset for easy analysis.

The article guides us through the steps of building FableForge, detailing the challenges, successes, and methodologies adopted. You will learn how the team leveraged the **"function calling" feature newly introduced by OpenAI**, which is used in this project specifically to structure text data suitable for Stable Diffusion, a task that initially proved difficult due to the model's tendency to include non-visual content in the prompts. We'll see how to overcome this by using a function providing structured, actionable output for external tools.

We'll delve into each component of FableForge, including the generation of text and images, combining them into a book format, storing the data into Deep Lake, and finally presenting it all through a user-friendly interface with Streamlit. We'll explore the process of text generation, extracting visual prompts, assembling PDFs, and uploading the data to Deep Lake.

By the end of this lesson, you'll gain a comprehensive understanding of how various AI tools and methodologies can be effectively integrated to overcome challenges and open new frontiers in creative domains.

LangChain & Vector DBs: 60+ lessons & projects in our course. [Enroll for free](#)

## AI Story Generator:

### OpenAI Function Calling, LangChain, & Stable Diffusion

Go from Prompt to an Illustrated Children's Book with LangChain & OpenAI Function Calling.

Improve the Gen AI App with Fine-Tuning and Model Training Deep Lake

#### Meet FableForge, AI Picture Books Generator powered by OpenAI, LangChain, Stable Diffusion, & Deep Lake

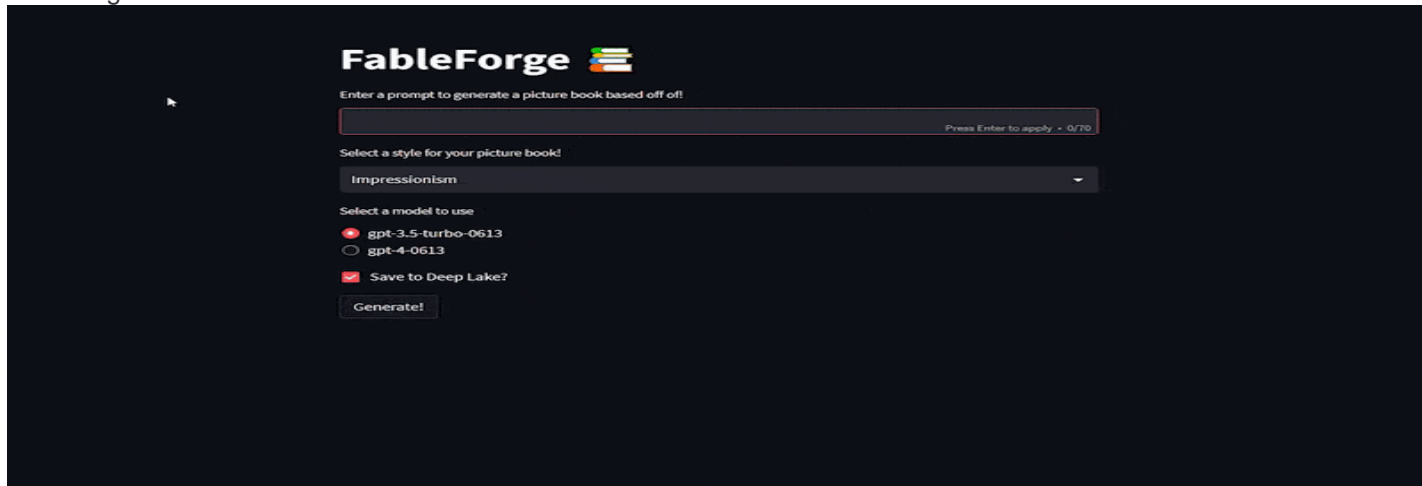
Imagine a world where children's picture books are created on-demand by children from a single prompt. With each generated image, the text and prompt pairs are stored for further finetuning if the child likes the story - to fit one human's imagination perfectly.

This is the grand vision of FableForge.

FableForge is an open-source app that generates children's picture books from a single prompt. First, GPT-3.5/4 is instructed to write a short children's book. Then, using the **new function calling feature OpenAI** just announced, the text from each book page is transformed into a prompt for Stable Diffusion.



These prompts are sent to Replicate, corresponding images are generated, and all the elements are combined for a complete picture book. The matching images and prompts are stored in a Deep Lake vector database, allowing easy storing and visualizing of multimodal data (image and text pairs). Beyond that, the generated data can be streamed to machine learning frameworks in real time while training, to finetune our generative AI model. While the latter is beyond the scope of this example, we'd love to cover how it all works together.

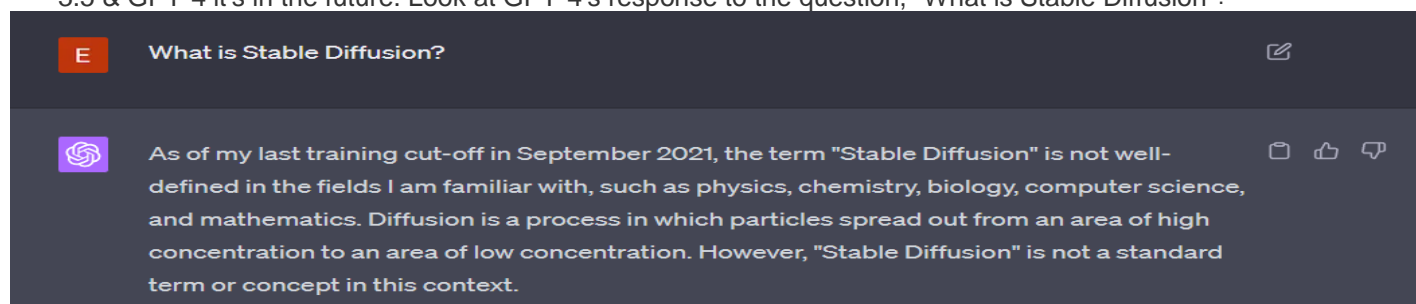


## What Did and Didn't Work while Building FableForge?

### Didn't Work: Instructing GPT-4 To Generate Stable Diffusion Prompts

Initially, it seemed like it might be possible to send the LLM the text of our book and tell it to generate a prompt for each page. However, this didn't work for a few reasons:

- **Stable Diffusion released in 2022:** While it might seem like Stable Diffusion is already "old news", to GPT-3.5 & GPT-4 it's in the future. Look at GPT-4's response to the question, "What is Stable Diffusion ?"



- **Teaching the LLM how to prompt is difficult:** It's possible to instruct the LLM to generate prompts without the LLM knowing what Stable Diffusion is; giving it the exact format to generate a prompt with has decent results. Unfortunately, the often injects plot details or non-visual content into the prompts, no matter how often you tell it not to. These details skew the relevance of the prompts and negatively impact the quality of the generated images.

## What Did Work: Function Calling Capabilities

### What is OpenAI Function Calling?

On June 13th, [OpenAI announced a huge update to the chat completions API - function calling!](#) This means we can provide the chat model with a function, and the chat model will output a JSON object according to that function's parameters.

Now, the chat models can interpret natural language input into a structured format suitable for external tools, APIs, or database queries. The chat models are designed to detect when a function needs to be called based on the user's input and can then respond with JSON that conforms to the described function's signature.

In essence, function calling is a way to bridge the gap between unstructured language input and structured, actionable output that other systems, tools, or services can use.

## How FableForge Uses Functions

For our Stable Diffusion prompts, we need structured data that strictly adheres to specific rules - a function is perfect for that! Let's take a look at one of the functions we used:

```
1 get_visual_description_function = [{
2   'name': 'get_passage_setting',
3   'description': 'Generate and describe the visuals of a passage in a book. Visuals only, no characters, plot, or
  people.',
4   'parameters': {
5     'type': 'object',
6     'properties': {
7       'setting': {
8         'type': 'string',
9         'description': 'The visual setting of the passage, e.g. a green forest in the pacific northwest',
10      },
11     'time_of_day': {
12       'type': 'string',
13       'description': 'The time of day of the passage, e.g. nighttime, daytime. If unknown, leave blank.',
14     },
15     'weather': {
16       'type': 'string',
17       'description': 'The weather of the passage, eg. rain. If unknown, leave blank.',
18     },
19     'key_elements': {
20       'type': 'string',
21       'description': 'The key visual elements of the passage, eg tall trees',
22     },
23     'specific_details': {
24       'type': 'string',
25       'description': 'The specific visual details of the passage, eg moonlight',
26     }
27   },
28   'required': ['setting', 'time_of_day', 'weather', 'key_elements', 'specific_details']
29 } ] }
```

With this, we can send the chat model a page from our book, the function, and instructions to infer the details from the provided page. In return, we get structured data that we can use to form a great Stable Diffusion prompt!

## LangChain and OpenAI Function Calling

When we created FableForge, OpenAI announced the new function calling capabilities. Since then, [LangChain](#) - the open-source library we use to interact with OpenAI's Large Language Models - has added even better support for using functions.

Our implementation of functions using LangChain is as follows:

**1. Define our function:** First, we define our function, as we did above with `get_visual_description_function`.

**2. Give the chat model access to our function:** Next, we call our chat model, including our function within the `functions` parameter, like so:

```
response= self.chat([HumanMessage(content=f'{page}')],functions=get_visual_description_function)
```

- **Parse the JSON object:** When the chat model uses our function, it provides the output as a JSON object. To convert the JSON object into a Python dictionary containing the function output, we can do the following:

```
function_dict = json.loads(response.additional_kwargs['function_call']['arguments'])
```

In the function, we defined earlier, 'setting' was one of the parameters. To access this, we can write:

```
setting = function_dict['setting']
```

And we're done! We can follow the same steps for the each of the other parameters to extract them.

## Perfecting the Process: Using Deep Lake for Storage and Analysis

The final step breakthrough for perfecting FableForge was using Deep Lake to store the generated images and text. With Deep Lake, we could store multiple modalities of data, such as images and text, in the cloud. The web-based UI provided by Deep Lake made it incredibly straightforward to display, analyze, and optimize the generated images and prompts, improving the quality of our picture book output. For future Stable Diffusion endeavors, we now have a decently-sized dataset showing us what prompts work, and what prompts don't!



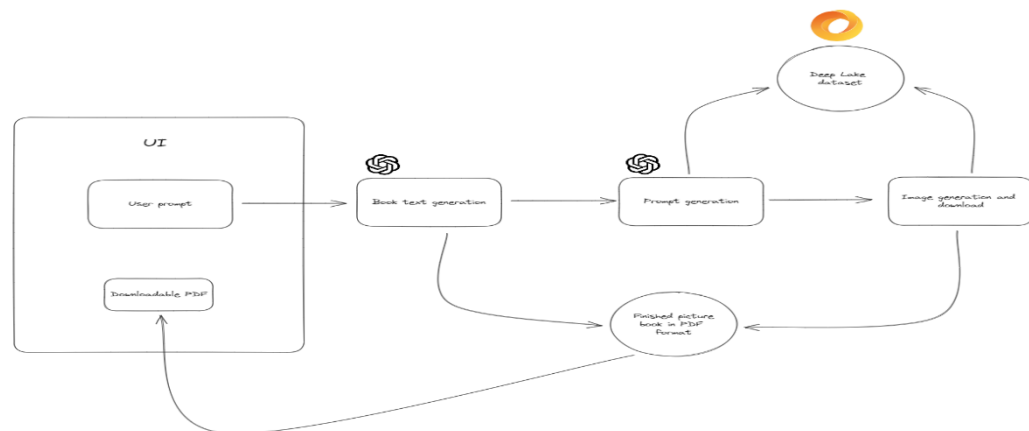
## Building FableForge

FableForge's open-sourced code is located [here](#).

FableForge consists of four main components:

1. The generation of the text and images
2. The combining of the text and images to create the book
3. Saving the images and prompts to the Deep Lake dataset
4. The UI

Let's take a look at each component individually, starting with the generation of the text and images. Here's a high-level overview of the architecture:



## First Component: AI Book Generation

All code for this component can be found in the `api_utils.py` file.

**1. Text Generation:** To generate the text for the children's book, we use LangChain and the ChatOpenAI chat model.

```
1 def get_pages(self):
2     pages = self.chat([HumanMessage(content=f'{self.book_text_prompt} Topic: {self.input_text}')]).content
3     return pages
```

`self.book_text_prompt` is a simple prompt instructing the model to generate a children's story. We specify the number of pages inside the prompt and what format the text should come in. The full prompt can be found in the `prompts.py` file.

**2. Visual Prompts Generation:** To produce the prompts we will use with Stable Diffusion, we use functions, as outlined above. First, we send the whole book to the model:

3.

```
1 def get_prompts(self):
2     base_atmosphere = self.chat([HumanMessage(content=f'Generate a visual description of the overall
lightning/atmosphere of this book using the function.'
3                                     f'{self.book_text}')]), functions=get_lighting_and_atmosphere_function)
4     summary = self.chat([HumanMessage(content=f'Generate a concise summary of the setting and visual details
of the book')]).content
```

Since we want our book to have a consistent style throughout, we will take the contents of `base_atmosphere` and append it to each individual prompt we generate later on. To further ensure our visuals stay consistent, we generate a concise summary of the visuals of the book. This summary will be sent to the model later on, accompanying each individual page, to generate our Stable Diffusion prompts.

```
1 def generate_prompt(page, base_dict):
2     prompt = self.chat([HumanMessage(content=f'General book info: {base_dict}. Passage: {page}.
Infer details about passage if they are missing, '
3         f'use function with inferred detailsm as if you were illustrating the passage.']),
        functions=get_visual_description_function    )
```

This method will be called for each individual page of the book. We send the model the info we just gathered along with a page from the book, and give it access to the `get_visual_description_function` function. The output of this will be a JSON object containing all the elements we need to form our prompts!

```
1 for i, prompt in enumerate(prompt_list):
2     entry = f'{prompt['setting']}, {prompt['time_of_day']}, {prompt['weather']}, {prompt['key_elements']},
{prompt['specific_details']}, " \
3         f"{base_dict['lighting']}, {base_dict['mood']}, {base_dict['color_palette']}, in the style of {style}"
```

Here, we combine everything. Now that we have our prompts, we can send them to Replicate's Stable Diffusion API and get our images. Once those are downloaded, we can move on to the next step.

## Second Component: Combining Text and Images

Now that we have our text and images, we can open up MS Paint and copy-paste the text onto each corresponding image. That would be different, and it's also time-consuming; instead, let's do it programmatically. In `pdf_gen_utils.py`, we turn our ingredients into a proper book in these steps:

**1.Text Addition and Image Conversion:** First, we take each image, resize it, and apply a fading mask to the bottom - a white space for us to place our text. We then add the text to the faded area, convert it into a PDF, and save it.

**2. Cover Generation:** A book needs a cover that follows a different format than the rest of the pages. Instead of a fading mask, we take the cover image and place a white box over a portion for the title to be placed within. The other steps (resizing and saving as PDF) are the same as above.

**3. PDF Assembly:** Once we have completed all the pages, we combine them into a single PDF and delete the files we no longer need.

## Third Component: Saving to Deep Lake

Now that we have finalized our picture book, we want to store the images and prompts in Deep Lake. For this, we created a `SaveToDeepLake` class:

```
1 import deeplake
2
3 class SaveToDeepLake:
4     def __init__(self, buildbook_instance, name=None, dataset_path=None):
5         self.dataset_path = dataset_path
6         try:
7             self.ds = deeplake.load(dataset_path, read_only=False)
8             self.loaded = True
9         except:
10             self.ds = deeplake.empty(dataset_path)
11             self.loaded = False
12
13             self.prompt_list = buildbook_instance.sd_prompts_list
14             self.images = buildbook_instance.source_files
15
16     def fill_dataset(self):
17         if not self.loaded:
18             self.ds.create_tensor('prompts', htype='text')
19             self.ds.create_tensor('images', htype='image', sample_compression='png')
20         for i, prompt in enumerate(self.prompt_list):
21             self.ds.append({'prompts': prompt, 'images': deeplake.read(self.images[i])})
```

When initialized, the class first tries to load a Deep Lake dataset from the provided path. If the dataset doesn't exist, a new one is created.

If the dataset already existed, we simply added the prompts and images. The images can be easily uploaded using `deeplake.read()`, as Deep Lake is built to handle multimodal data.

If the dataset is empty, we must first create the tensors to store our data. In this case, we create a tensor 'prompts' for our prompts and 'images' for our images. Our images are in PNG format, so we set `sample_compression` to 'png'.

Once uploaded, we can view them in the UI, as shown above.

All code can be found in the `deep_lake_utils.py` file.

## Final Component: Streamlit UI

To create a quick and simple UI, we used Streamlit. The complete code can be found in `main.py`.

Our UI has three main features:

**1. Prompt Format:** In this text input box, we allow the user to specify the prompt to generate the book based on. This could be anything from a theme, a plot, a time, and so on.

**2. Book Generation:** Once the user has input their prompt, they can click the *Generate* button to generate the book. The app will run through all of the steps outlined above until it completes the generation. The user will then have a button to download their finished book.

**3.Saving to Deep Lake:** The user can click the *Save to Deep Lake* checkbox to save the prompts and images to their Deep Lake vector database. Once the book is generated, this will run in the background, filling the user's dataset with all their generated prompts and images. Streamlit is an excellent choice for quick prototyping and smaller projects like FableForge - the entire UI is less than 60 lines of code!

## **Conclusion: The Future of AI-Generated Picture Books with FableForge & Deep Lake**

Developing FableForge was a perfect example of how new AI tools and methodologies can be leveraged to overcome hurdles. By leveraging the power of LangChain, OpenAI's function calling feature, Stable Diffusion's image generation abilities, and Deep Lake's multimodal dataset storage and analysis capabilities, we created an app that opens up a new frontier in children's picture book creation. Everyone can create an app like this - we did it, too. What will matter for you in the end, however, is having the data as the moat - and using the data you gather from your users to finetune models - providing them personal, curated experiences as they immerse themselves into fiction. This is where Deep Lake comes into place. With its 'data lake' features of visualization of multimodal data and streaming capability, Deep Lake enables teams to finetune their LLM performance or train entirely new ML models in a cost-effective manner.

- [Announcing Gen AI 360: Foundational Model Certification in Collaboration with Intel & TowardsAI](#) on Jun 20, 2023 [Blog](#)
- [Conversation Intelligence: Gong.io Open-Source Alternative AI Sales Assistant](#) on Jun 13, 2023