Exploring LangChain and LlamaIndex to Achieve Standardization and Interoperability in Large Language Models

Majid · Follow
Published in Badal-io
22 min read · May 11

Listen Share More

Over the past three months, there has been an exponential acceleration in the development and implementation of Large Language Models (LLMs). From generating AI-generated songs like Drake's "Tootsie Slide" to Chat GBT's intelligent customer service chatbots and the impressive natural language processing capabilities of Prompt Agent GPT.

In this blog post, we will explore the world of LLMs and delve into two fascinating tools: LangChain and LlamaIndex, which together provide standardization and interoperability. We will review key applications of each, address where they overlap, and where one might serve you more effectively. Note that these libraries are at their early stages and are receiving significant updates monthly. Therefore, some details mentioned about the code may not hold depending on when you read it.

What is a LLM?

Large Language Models (LLMs) are machine learning models that can generate human-like text and respond to prompts in natural language. These models are trained on vast amounts of data, including books, articles, and websites, and use statistical patterns to predict the most likely words or phrases to follow a given input.

LangChain — Standardizing Interactions

Langchain is a response to the intense competition among LLMs, which have grown increasingly complex with frequent updates and a massive number of parameters. Previously, utilizing these models required cloning the code, downloading trained weights, and manually configuring settings for each application. Fine-tuning was also a non-trivial task. To streamline the process, LLM providers like Huggingface and Cohere emerged, offering well-engineered APIs that abstract away many of the aforementioned challenges. However, some use cases still require carefully crafted prompts, and not all APIs offer the same features. This has led to a demand for standardized interactions and interoperability, enabling users to avoid vendor lock-in and switch seamlessly between providers based on performance, cost, and regulation.

We start with the building blocks of LangChain:

Models

Are divided into 3 categories:

1. **LLMs:** An LLM is essentially a self-contained language model that takes in textual input and produces output text as a result.
2. **Chat models:** Chat models, whether provided by OpenAI, Cohere, HuggingFace, etc. are similar to LLMs but conceptually different in that they work with message objects rather than pure text. There are three types of message objects, namely, SystemMessage, HumanMessage, and AIMessage. These are simple wrappers around text that have no special effects on their own but help make distinctions between entities in a conversation. The best practice is to use HumanMessages for text input by a human user, AIMessages for a text generated by the chat model, and SystemMessages to provide some context for a chat model to give it some clue on how it should respond in a conversation. In the example below:

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)

chat = ChatOpenAI(temperature=0)

messages = [
    SystemMessage(content="You are a hilarious doctor."),
```

```
    HumanMessage(content="Describe chicken pox to me.")
]
chat(messages)
```

The system messages and human messages are simply strung together in a prompt and sent over to the chat model that will return a text response which will be wrapped in an AIMessage.

The documentation admits that abstractions for chat models are still immature due to the recency of these models.

3. **Embedding models:** Embedding models are used to create vector representations for texts. The most popular application of these embedding models is for semantic search where a query embedding is compared to embeddings of a set of documents. LangChain exposes two methods of embed_query and embed_document mainly due to the fact that some LLM providers use different methods to generate embeddings for queries than for documents.

Prompts

Prompting is the new programming. A prompt is an input to a language model which elicits a desired response. Sometimes the response could be fundamentally different depending on how the prompt is phrased especially for more complicated tasks. In fact, Langchain and LlamaIndex use carefully designed prompts for many of their tasks and that's where a lot of their utility lies. Consider the example of querying a SQL database and getting a response in natural language. Instead of you calling an LLM with your own prompt and including the query, instructions on how to interpret a schema, call a database and fetch the results and translate it back to English, you simply provide your data schema and query. This query and schema are placed in a prompt template that will elicit the desired response.

There are four categories of LangChain prompt templates you should be familiar with are:

1. **LLM Prompt Templates:** In order to parametrize your prompts and avoid hardcoding them, Langchain provides an object which is built upon Python's formatted strings (Currently, Langchain supports Jinja and will soon incorporate other templating languages).

```
from langchain import PromptTemplate
prompt = PromptTemplate(
   input_variables=["topic", "city"],
   template="Tell me a about {topic} in {city}."
)

prompt.format(topic="food", city="Rome")
# Tell me about food in Rome
```

You can also check out LangChainHub to see if any of the prepared templates fits your needs

2. **Chat Prompt Templates:** As explained before, here instead of string objects, we talk about message objects. Don't fret, there is nothing terribly different about them. They just give structure to conversational use cases. Messages are divided into three categories: 1. HumanMessages 2. AIMessages 3. SystemMessages. The first two are self-explanatory. SystemMessages are neither from AIs nor from Humans. Typically they set the context for a chat. "You are a helpful AI that helps with weather forecast" is an example of SystemMessage.

All this to say, ChatPromptTemplates are not simply made from strings like LLM prompts are. Rather they are constructed upon MessageTemplates (HumanMessages, AIMessages, and SystemMessages):

```
systemTemplate = SystemMessagePromptTemplate.from_template("You are a helpful AI that t
humanTemplate = HumanMessagePromptTemplate.from_template('{input}')
chatTemplate = ChatPromptTemplate.from_messages([systemTemplate, humanTemplate])

chatTemplate.format_prompt(topic="food", country="Italy", input="I like to learn a new
# output would be: [AIMessage(content='You are a helpful AI that talks about food in It
# HumanMessage(content='I like to learn a new recipe', additional_kwargs={})]
```

Example Selectors: Won't get into this for now. Suffice it to say LangChain gives the flexibility to determine the strategy by which you select input examples to a language model from a list of few-shot learning examples. For example, an example selector that works

based on the input length, adjusts the number of examples selected from your prompt based on the length of the rest of the prompt

Output Parsers: Inputs and prompts are only on one side of LLMs. Sometimes how the output is formatted becomes crucially important e.g. for downstream tasks. You can use off-the-shelf output parsers LangChain provides or create one for your costume use case. For example, an output parser that would parse the LLM output into a list of comma-separated values to be stored as CSV files.

Indexes

This tool revolves around the retrieval of relevant information from a set of documents given a query. The ingredients for building this system are a tool to load our documents, a tool to create embedding vectors for those documents and one that will store and keep track of these vectors and documents for us. Accomplishing this via LlamaIndex is a bit different than LangChain as LangChain offers a bit more granularity through the following classes:

Document Loaders: This tool helps with loading your documents from a variety of sources and formats (HTML, PDF, Email, Git, Notion, etc.) and uses Unstructured under the hood. In the example below taken from the official docs, the text document is loaded via a simple text loader.

```
from langchain.document_loaders import TextLoader
loader = TextLoader('../state_of_the_union.txt')
documents = loader.load()
```

Text Splitters: It's neither ideal nor possible to dump very long documents into a vector embedding model. Ideally, we want to chunk our document into very coherent pieces that discuss one topic so we get as accurate a vector representation as possible. Moreover, the token size limitations of LLMs are another reason we would want to split our documents. LangChain provides a variety of splitters, however, I feel for a production-ready solution or product you would construct your own splitter, especially for the first reason.

```
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
```

VectorStores: These are essentially databases where you store your embedding vectors (as the name suggests) and they expose semantic similarity search functionalities. LangChain has support for a good number of players in this field such as PineCone and Chroma.

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
embedder = OpenAIEmbeddings()
db = Chroma.from_documents(texts, embedder)
output = db.similarity_search(query)
# This output would be a list of relevant documents
```

Retrievers: Are very closely related to VectorStore indexes like “db” in the code above. It is important to note that a VectoreStore index has other utilities than solely querying for relevant documents. However, the retriever interface is specifically designed for document retrieval. You can determine the method by which the similar documents are retrieved plus the number of similar documents. These objects can be passed around in chains that need a retrieval component. For example, the VectorstoreIndexCreator class that offers a conversational experience uses retriever objects to identify salient information in one of the chain steps:

```
from langchain.chains import RetrievalQA
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter

loader = TextLoader('../state_of_the_union.txt', encoding='utf8')
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)


embedder = OpenAIEmbeddings()
db = Chroma.from_documents(texts, embeddings)

retriever = db.as_retriever()
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff", retriever=retriever)

query = "What did the president say about Ketanji Brown Jackson"
qa.run(query)
```

```
# This will return a chat like response for the query rather than only listing the rele
```

Memory

This one is simple. Your interaction with LLMs is typically not stored in a memory, however, you can see how the existence of memory becomes very important in applications such as chatbots. Langchain provides memory objects that can be passed around in chains or you can use them standalone to investigate the history of an interaction, extract a summary, etc.

There are a variety of memory and history classes for various purposes. The docs mention these two:

ChatMessageHistory: These serve the latter use case. Use them to dig into previous interactions and/or create context and coherence.

In the example below each call to a language model is stored in a history object. This history can later be used to prompt a new language model and provide context.

```
from langchain.llm import LLM
from langchain.chat import ChatMessageHistory

# Create a new ChatMessageHistory object and add some messages
history = ChatMessageHistory()
history.add_user_message("Hello!")
history.add_ai_message("Hi there!")
history.add_user_message("How are you?")

# Create a new LLM object and train it on the chat history
llm = LLM()
llm(f"Given the history: {history.messages} tell me what the first human message was")
```

ConversationBufferMemory: This is just a thin wrapper around ChatMessageHistory to make it easier to load the history in different formats and pass it around in chains and models.

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()

memory.chat_memory.add_user_message("hi!")
memory.chat_memory.add_ai_message("whats up?")

print(memory.buffer)
# This will return: 'Human: hi!\nAI: whats up?'

# passing to a conversation chain to provide previous context:
from langchain.chains import ConversationChain

llm = OpenAI(temperature=0)
conversation = ConversationChain(
   llm=llm,
   verbose=True,
   memory=memory
)

conversation.predict(input="what was my first message to you?")

# This will predict:
# > Entering new ConversationChain chain...
# Prompt after formatting:
# The following is a friendly conversation between a human and an AI. The AI is talkat

# Current conversation:
# [HumanMessage(content='hi!', additional_kwargs={}), AIMessage(content='whats up?', ad
# Human: what was my first message to you?
# AI:

# > Finished chain.
# ' Your first message to me was "hi!"'
```

Saving History in a file: You can save your message history by converting them to dict objects and saving those (as json or pickle etc.):

```
from langchain.schema import messages_from_dict, messages_to_dict

dicts = messages_to_dict(history.messages)
# Output:
# [{'type': 'human', 'data': {'content': 'hi!', 'additional_kwargs': {}}},
#  {'type': 'ai', 'data': {'content': 'whats up?', 'additional_kwargs': {}}}]

loaded_history = messages_from_dict(dicts)
```

```
print(loaded_history)
# Output
# [HumanMessage(content='hi!', additional_kwargs={}),
#  AIMessage(content='whats up?', additional_kwargs={})]
```

Chains

Chains allow you to combine multiple components (e.g. language models, prompts, agents, memory objects, indexes, etc.) and create something fancy.

There is a great deal of flexibility here for a host of use cases. Let's go over two simple examples. A popular usage of chains is to combine prompts with LLM or chat models. This way you only input a few keywords instead of repeating an entire prompt every time and you get back an elaborate response.

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.llms import OpenAI
llm = OpenAI(temperature=0.9)
prompt = PromptTemplate(
   input_variables=["continent"],
   template="Name one nice summer vacation spot in {continent}?"
)

chain = LLMChain(llm=llm, prompt=prompt)
print(chain.run("Europe"))
# This will output something like:
# French Riviera
```

Now let's say you want to add some options for food to your vacation. You can take the output of this chain and input it into another chain that proposes foods for a given region. Let's first create such a chain:

```
prompt2 = PromptTemplate(
   input_variables=["destination"],
   template="What food should I try when in {destination}?"
)
chain2 = LLMChain(llm=llm, prompt=prompt2)
chain2.run("Ibiza")
```

```
# Output:
# Paella: This traditional Spanish dish is a must-try when in Ibiza. It is a rice dish
```

You can string the above two chains together using the SequentialChain class:

```
from langchain.chains import SimpleSequentialChain
overall_chain = SimpleSequentialChain(chains=[chain, chain2], verbose=True)
overall_chain.run("Europe")
# Output:
# > Entering new SimpleSequentialChain chain...
# The French Riviera.
# When in the French Riviera, you should definitely try some of the local specialties s
# > Finished chain.
# \n\nWhen in the French Riviera, you should definitely try some of the local specialt
```

Of course, there are other classes besides SequentialChain and LLM chain and you can even go ahead and construct your own costume chain class to achieve a certain behavior.

Agents and tools

Language models by themselves, no matter how sophisticated, are limited to the corpus they were trained on. As a result, for example, you can't ask your LLM to tell you the weather forecast for tomorrow because the poor thing does not have access to any recent weather information. Unless you hook it up to a tool (like a weather API) that the LLM can read that data from and compose the answer in natural language for you.

You might think that simply creating a chain that will link all the right tools with the LLM resolves this issue. For the weather forecast use-case above it could but imagine a scenario where you don't have the foresight of what exactly your user will ask for. They might ask about a math problem or the recent stock news or the weather forecast. Chains are not the solution here as they assume all the tools and links will have to be used and executed.

Agents, on the other hand, can decide which tools are relevant for each query and only use those as long as you provide them with a list of tools.

The pattern for using tools and agents is simple:

1. Load the necessary tools

2. Initialize an agent by specifying those tools, a language model, and an agent type you would like to use

3. Call the agent with your query

As an example, let's ask a language model what today's temperature would be to the power of two. A good old LLM cannot answer this standalone because it does not have access to current weather information nor does it have a calculator to compute powers of numbers. Therefore we create an agent from our language model and arm it with the right tools:

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import OpenAI


llm = OpenAI(temperature=0)
# load tools
tools = load_tools(['openweathermap-api', 'llm-math'], llm=llm)
# initialize agent
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbo
# call the agent with your query
agent.run("what is the temperature of today in Baja California in celcius to the power
```

This agent will first find out the current temperature using the OpenWeatherMap API and then will use the LLM-math tool to find the square of that number. Note that some tools like LLM-math require an LLM to do their job. Likely the tool is made of a chain that uses a language model, hence, we pass it to the load_tools function as an argument.

Tools: You can use any off-the-shelf tool available in LangChain or create your own. An important aspect of a tool is its description which is the main piece of information the agent uses to decide whether it should use that tool for any given query. For example, here is the description registered for **OpenWeatherMap-API :**

> *A wrapper around OpenWeatherMap API. Useful for fetching current weather information for a specified location. Input should be a location string (e.g. 'London,GB').*

prompting a language model with:

"given {query} and the tool with {tool_description} should I use this tool to

answer the query?"

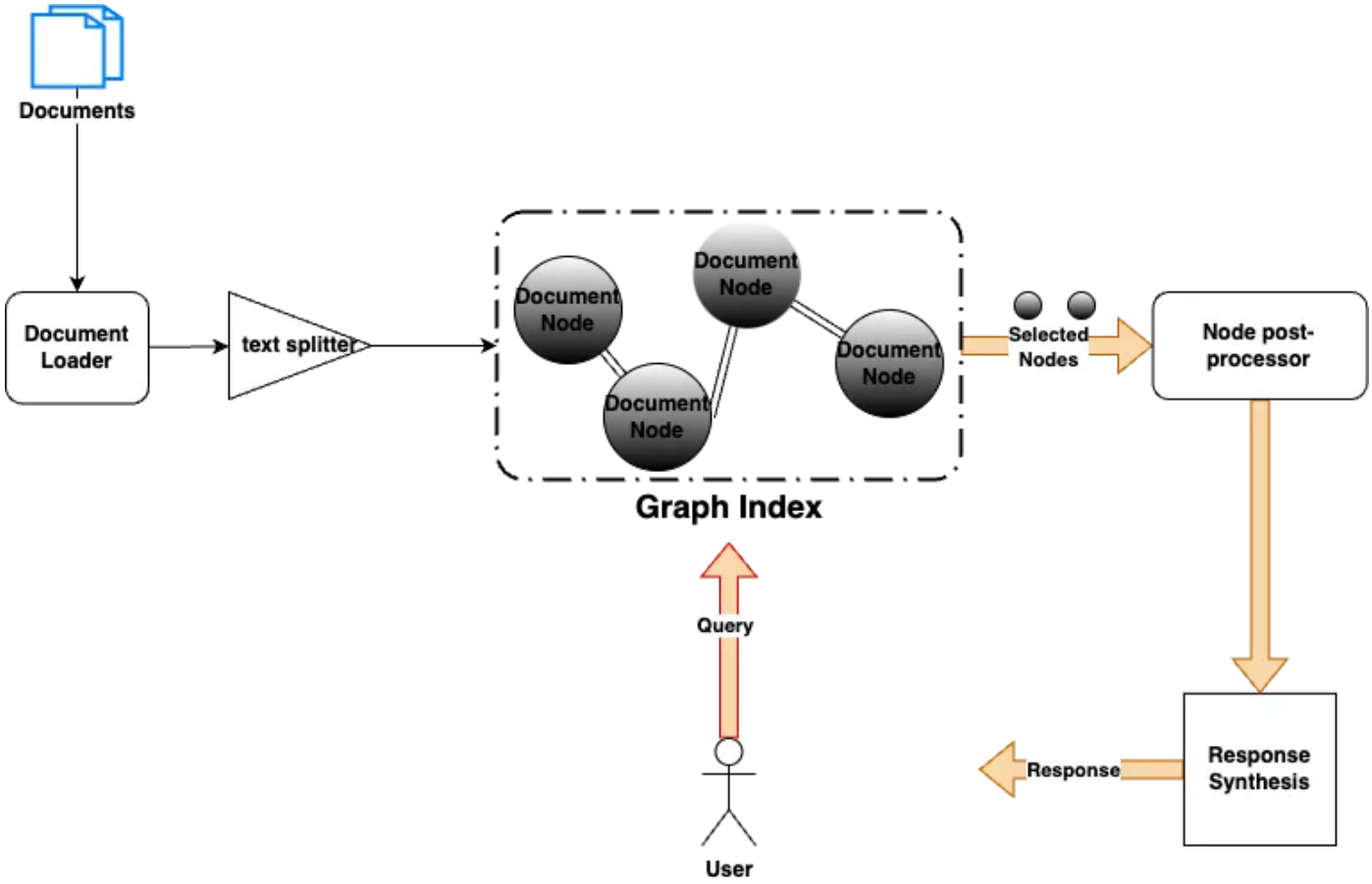
helps with this process and is similar to what's happening under the hood of an agent. Therefore, the description you register for your costume tool is of high importance.

LlamaIndex

LlamaIndex is a software tool designed to simplify the process of searching and summarizing documents using a conversational interface powered by large language models (LLMs). LangChain serves as the foundation for much of the tool's functionality. Additionally, graph indexes are a key feature of LlamaIndex, helping to efficiently organize and optimize the data it processes. Overall, the goal of LlamaIndex is to enhance document management through advanced technology, providing an intuitive and efficient way to search and summarize documents using LLMs and innovative indexing techniques.

Llama heavily uses prompting to achieve a lot of the utility it offers.

The figure below illustrates the overall workflow of a Llama index:



The overall workflow of LlamaIndex

The knowledge base (e.g., organizational docs) is chunked up and each chunk is stored in a node object which will collectively form a graph (index) with other nodes. The principal reason for this chunking is the fact that LLMs have limited input token capacity and as a result, a strategy to feed large documents in a smooth, continuous manner in prompts would be helpful.

The graph index can be a simple list structure, a tree structure, or a keyword table. In addition, one can compose an index from different indexes. This one is useful when we want to organize documents into a hierarchy for better search results. For example, we can create separate list indexes over confluence, Google Docs, and emails and create an overarching tree index over the list indexes.

Let's get more granular and find out how the documents are chunked up. LangChain provides a set of textSplitter classes that are designed to break up input to a language model in order to stay within token limitations for LLMs. The textSplitter can split by the number of characters, number of tokens, or other costume measurements. Since these chunks are to be sequentially fed to an LLM, there is some overlap between them to maintain context. LlamaIndex uses these textSplitter classes to chunk the docs. However, if we need control

over how our documents are chunked, custom splitters can be created or a simpler approach would be to create our chunks beforehand and use those to create nodes of a graph index. This is possible because, in LlamaIndex, a graph can be created from either a set of documents or a set of nodes.

Querying

Querying an index graph results in two main operations. First, a set of relevant nodes to the query are fetched. Then, a ***response_sythesis*** module is executed given these nodes and the initial query to generate a coherent answer. How and whether a node is deemed relevant depends on the index type. Let's examine how those relevant nodes are retrieved in different configurations:

Querying list index: A list index simply uses all the nodes in the list sequentially to generate a response. The query along with the information in the first node is sent to the LLM in a prompt. The prompt will look something like "*given this {context}, answer this {query}"* in which the node will provide the context and the query is the initial query. The returned

{query} according to the context". This process continues until all the nodes are traversed.

As you can tell, this index by default retrieves and passes all nodes in an in… response synthesis module. However, if the query_mode parameter is specified as "embedding", only the top most similar nodes (measured by vector similarities) will be retrieved for response_synthesis.

Querying a vector index: A vector index calculates embeddings for each document node and has them stored in a vector database like PineCone or Vertex AI matching engine. Compared to the list index, the difference in retrieval is the fact that only those nodes above a certain relevance threshold to the query are fetched and sent over to the response_sythesis model.

Response synthesis

This module offers a few options on how the response is created.

Create and refine: This is the default mode for a list index. The list of nodes is traversed sequentially and at each step, the query, the response so far, and the context of the current node are embedded in a prompt template that prompts an LLM to refine the response to the query according to the new information in the current node.

Tree summarize: This is very similar to the tree index in that a tree will be created from the selected candidate nodes. However, the summarization prompt which is used to derive the parent nodes will be seeded with the query. What's more, the tree construction continues until we arrive at a single root node which will contain the answer to the query composed from the information in all the selected nodes.

Compact: This one is to save money :) Basically, the response synthesizer is instructed to stuff as many nodes as possible in the prompt before maxing out the token limitation of the LLM. If there are too many nodes to stuff in one prompt, the synthesizer will do it in steps where at each step maximum possible number of nodes are inserted into the prompt and the answer will be refined in the next steps.

Note that the prompts used to communicate with the LLMs are customizable. For instance, you can seed a tree construction with your own costume summary prompt.

Composability

One helpful feature of LlamaIndex is the ability to compose an index from other indexes (rather than nodes). Imagine a scenario where you need to search or summarize several heterogeneous sources of data. You can simply create a separate index over each data source and create a list index over these indexes. A list index is suitable because it creates and refines an answer (be it the summary or answer to a query) iteratively by stepping through each index. Note that you need to register a summary for each lower-level index. This is because similar to other modules and classes, this feature depends on prompting LLMs to for example identify the relevant sub-indexes. For example in a tree index with a branch factor of 1, this summary is used to identify the correct document to route the query to. However, this summary itself can be easily obtained via a tree index over that document if chosen to not input manually.

Data Connectors

Your data is likely not just a simple text file. You have your confluence pages, your pdf stored on disk, G-suite documents on clouds, etc. Luckily, LlamaIndex provides a host of data connectors and loaders available on LlamaHub to simplify and standardize the loading process. This is similar to LangChain's data loaders, however, for better accuracy and I suggest not solely relying on data loaders and default text splitters. No one knows your document structure better than you and you can load, preprocess and split your documents best. For instance, the confluence data loader from Llama is simply a wrapper around the html2text python library and dumps the entire confluence page into a string variable. Although this could be a quick approach to POC, better question-answering and retrieval

accuracies depend on how the HTML is parsed and how much of the hierarchical structure of the confluence page is preserved.

Query transformations

You can arm your query engine with query transformations to achieve more accurate answers, especially for more complex queries. The idea is to rephrase the query into simpler terms, borrow some hypothetical knowledge from generic LLMs, or break down a query into a sequence of smaller digestible queries. Here are the options to transform the query:

HyDE (Hypothetical Document Embedding): This transformation simply prompts an LLM with the query to retrieve a general hypothetical answer without considering the specific documents. The embedding of this hypothetical answer together with the query (depending on whether you set ***include_original*** to TRUE) is then used to retrieve the right information in your specific documents. Intuitively, this general answer informs the query engine what a legitimate answer should look like. However, the less context in the initial query, the higher the likelihood of this general hypothetical answer being irrelevant and even a total hallucination.

Single-step query decomposition: Imagine a scenario where your query could use some adjustments based on the contents of your document. This module uses the prompt template below to modify the query accordingly.

```
"The original question is as follows: {query_str}\n"
"We have an opportunity to answer some, or all of the question from a "
"knowledge source. "
"Context information for the knowledge source is provided below. \n"
"Given the context, return a new question that can be answered from "
"the context. The question can be the same as the original question, "
"or a new question that represents a subcomponent of the overall question.\n"
"As an example: "
"\n\n"
"Question: How many Grand Slam titles does the winner of the 2020 Australian "
"Open have?\n"
"Knowledge source context: Provides information about the winners of the 2020 "
"Australian Open\n"
"New question: Who was the winner of the 2020 Australian Open? "
"\n\n"
"Question: What is the current population of the city in which Paul Graham found "
"his first company, Viaweb?\n"
"Knowledge source context: Provides information about Paul Graham's "
"professional career, including the startups he's founded. "
```

```
    "New question: In which city did Paul Graham found his first company, Viaweb? "
    "\n\n"
    "Question: {query_str}\n"
    "Knowledge source context: {context_str}\n"
    "New question: "
```

his is very helpful in cases when a query is asked over a composed index and each sub-index only answers a part of the query. The module then breaks down the query into sub-queries and will route each sub-query to the appropriate sub-index and will at the end collect the sub-answers into a complete answer.

Multi-step query decomposition: Sometimes a complicated query needs to be broken down into more straightforward sub-queries. These sub-queries can be asked one by one in steps to eventually arrive at a satisfactory answer. The following prompt template is the backbone of this module:

```
    "The original question is as follows: {query_str}\n"
    "We have an opportunity to answer some, or all of the question from a "
    "knowledge source. "
    "Context information for the knowledge source is provided below, as "
    "well as previous reasoning steps.\n"
    "Given the context and previous reasoning, return a question that can "
    "be answered from "
    "the context. This question can be the same as the original question, "
    "or this question can represent a subcomponent of the overall question."
    "It should not be irrelevant to the original question.\n"
    "If we cannot extract more information from the context, provide 'None' "
    "as the answer. "
    "Some examples are given below: "
    "\n\n"
    "Question: How many Grand Slam titles does the winner of the 2020 Australian "
    "Open have?\n"
    "Knowledge source context: Provides names of the winners of the 2020 "
    "Australian Open\n"
    "Previous reasoning: None\n"
    "Next question: Who was the winner of the 2020 Australian Open? "
    "\n\n"
    "Question: Who was the winner of the 2020 Australian Open?\n"
    "Knowledge source context: Provides names of the winners of the 2020 "
    "Australian Open\n"
    "Previous reasoning: None.\n"
    "New question: Who was the winner of the 2020 Australian Open? "
    "\n\n"
```

```
    "Question: How many Grand Slam titles does the winner of the 2020 Australian "
    "Open have?\n"
    "Knowledge source context: Provides information about the winners of the 2020 "
    "Australian Open\n"
    "Previous reasoning:\n"
    "- Who was the winner of the 2020 Australian Open? \n"
    "- The winner of the 2020 Australian Open was Novak Djokovic.\n"
    "New question: None"
    "\n\n"
    "Question: How many Grand Slam titles does the winner of the 2020 Australian "
    "Open have?\n"
    "Knowledge source context: Provides information about the winners of the 2020 "
    "Australian Open - includes biographical information for each winner\n"
    "Previous reasoning:\n"
    "- Who was the winner of the 2020 Australian Open? \n"
    "- The winner of the 2020 Australian Open was Novak Djokovic.\n"
    "New question: How many Grand Slam titles does Novak Djokovic have? "
    "\n\n"
    "Question: {query_str}\n"
    "Knowledge source context: {context_str}\n"
    "Previous reasoning: {prev_reasoning}\n"
    "New question: "
```

As you can see, at each step a new sub-query is devised based on the previous sub-queries and their answers and reasons in order to get us one step closer to the final answer.

Node Postprocessors

Node post-processors come after retrieval and before response_sythesis to refine the set of selected nodes. There are a few different classes offered by Llama for example, *KeywordNodePostprocessor* class filters the retrieved nodes further according to an exclude and/or include keyword list.

Storage

Storage is rather an important aspect of this library for developers. We need storage for vectors (document embeddings), nodes (document chunks), and the index itself. By default, almost everything is stored in memory except for vector store services such as PineCone which stores your vectors in their databases. These in-memory storage objects can be written to disk to persist the information to be able to load it back in later. We'll go over each storage to see the available options:

1. **Document stores:** So far only MongoDB is supported as an alternative to in-memory storage. Namely, there are two classes: MongoDocumentStore and

SimpleDocumentStore that handle the storage of your document nodes either in a MongoDB server or in memory respectively.

2. **Index stores:** Similar to document stores, the two classes MongoIndexStore and SimpleIndexStore handle the storage of index metadata either in MongoDB or in memory respectively.

3. **Vector stores:** Along with the SimpleVectorStore class that holds your vectors in memory, LlamaIndex provides support for a variety of vector databases similar to LangChain. It is important to note that some vector databases store documents as well as vectors while others like PineCone only store vectors. However, hosted databases like PineCone allow for very efficient complex calculations on these vectors compared to in-memory databases such as Chroma.

Storage context: After configuring your storage objects according to your needs or leaving them as defaults, you create from them a storage_context object that your indexes can use to take everything into account:

```
Storage_context = StorageContext.from_defaults(
docstore = MongoDocumentStore.from_uri(uri="<mongodb+srv://...>")
index_store = MongoIndexStore.from_uri(uri="<mongodb+srv://...>")
Vector_store = PineconeVectorStore(config)
)

index = load_index_from_storage(storage_context, index_id="<index_id>")
```

LangChain vs LlamaIndex

As you can tell, LlamaIndex has a lot of overlap with LangChain for its main selling points, i.e. data augmented summarization and question answering. LangChain is imported quite often in many modules, for example when splitting up documents into chunks. You can use data loaders and data connectors from both to access your documents.

LangChain offers more granular control and covers a wider variety of use cases. However, one great advantage of LlamaIndex is the ability to create hierarchical indexes. Managing indexes as your corpora grows in size becomes tricky and having a streamlined logical way to segment and combine individual indexes over a variety of data sources proves very helpful.

Overall these two helpful libraries are very new and are receiving updates weekly or monthly. I would not be surprised if LangChain subsumed LlamaIndex in the near future to offer a consolidated source for all applications.

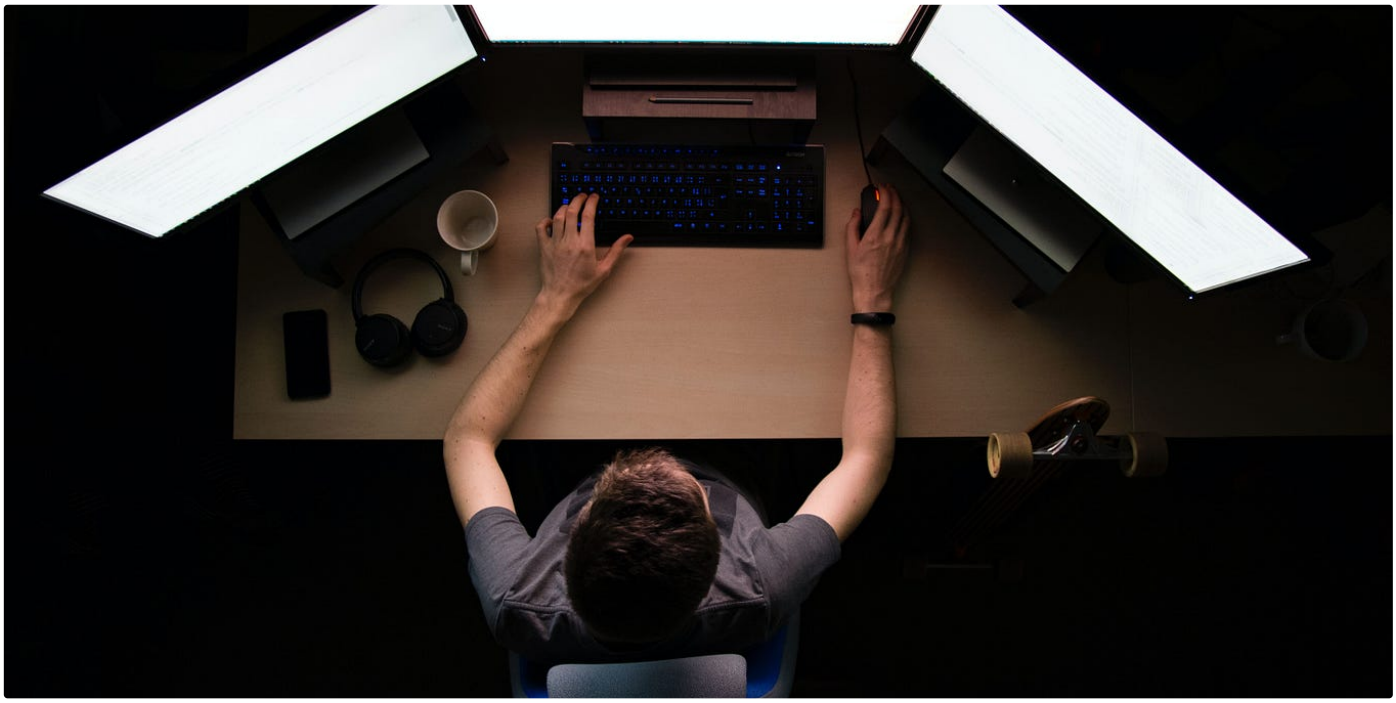
NLP | Large Language Models | Langchain | Llamaindex | ChatGPT

Follow

Written by Majid

60 Followers · Writer for Badal-io

More from Majid and Badal-io



Majid in Badal-io

Chat with your confluence

The narrative is familiar—businesses across the board grappling with the mammoth task of data retrieval, losing valuable man-hours in…

24 min read · Aug 3

13 1

Majid in Badal-io

Building a Real-Time Product Recommender System with Graph Databases: Leveraging Neo4j and BigQuery…

Introduction

11 min read · May 2

2

Rana Fahim Hashemi in Badal-io

Reducing False Anomaly Detection Alerts on Google Analytics Time Series Data with BigQuery ML ARIMA...

Introduction

13 min read · Apr 11

Majid

Laptop Recommendation System

Let's quickly build a baseline laptop recommendation web app! We will create a Flask app with Swagger for the front-end and deploy it on...

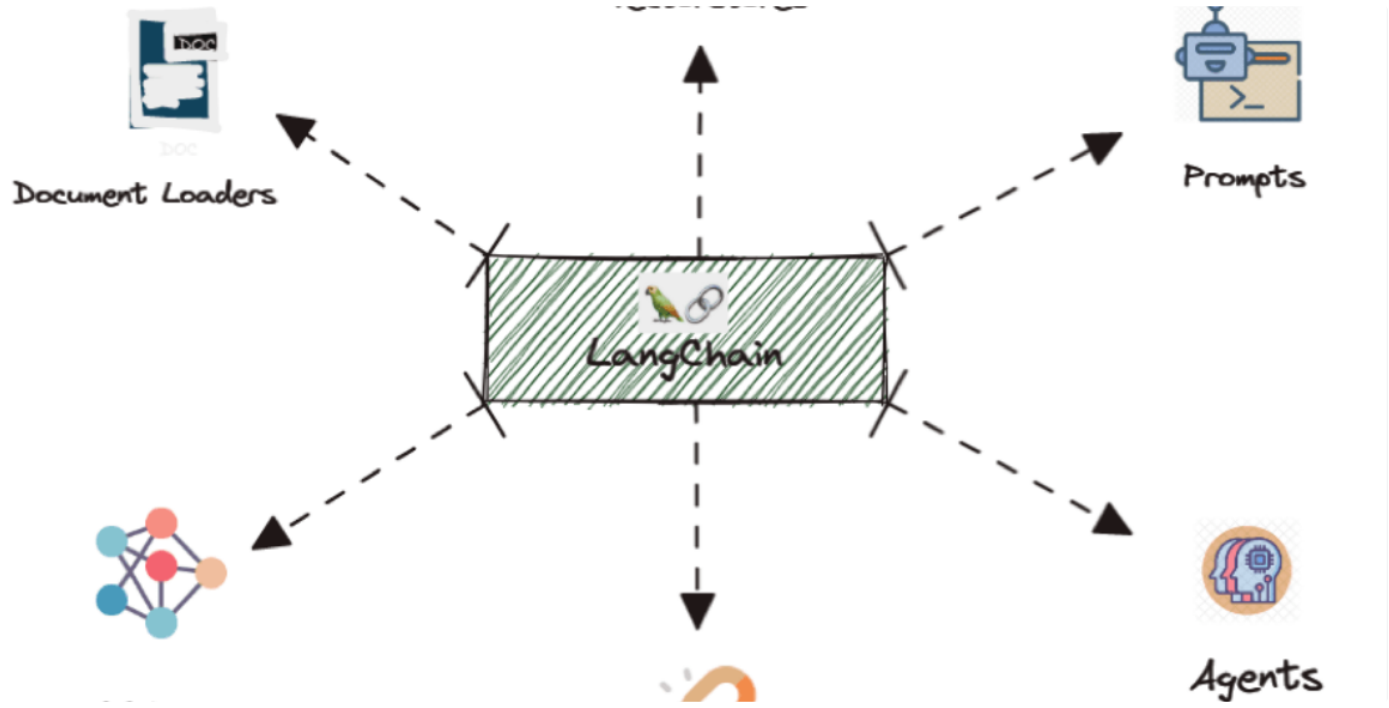
6 min read · Apr 28, 2021

7

See all from Majid

See all from Badal-io

Recommended from Medium



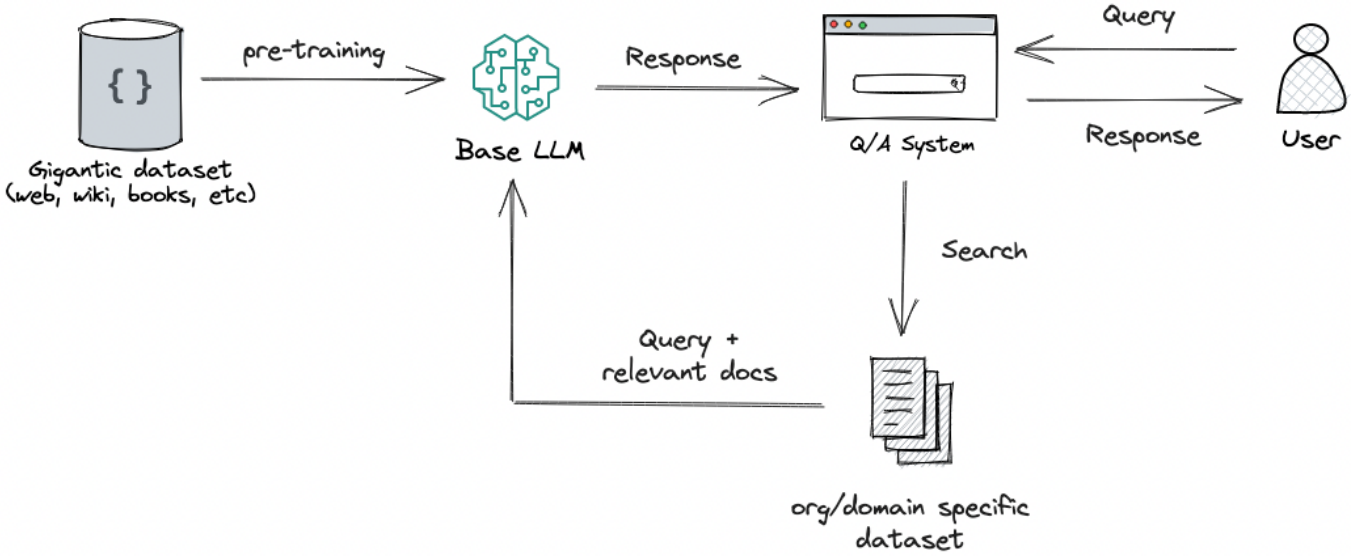
Zeeshan Malik

Connecting ChatGPT with your own Data using Llama Index and LangChain

In the last three months, there has been a rapid increase in the use of Large Language Models (LLMs) for a variety of applications, such as...

5 min read · Jun 11

104 2



Heiko Hotz in Towards Data Science

RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM Application?

The definitive guide for choosing the right method for your use case

· 19 min read · Aug 25

2.3K 17

Lists

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves

Natural Language Processing

669 stories · 283 saves

What is ChatGPT?

9 stories · 182 saves

ChatGPT

21 stories · 179 saves

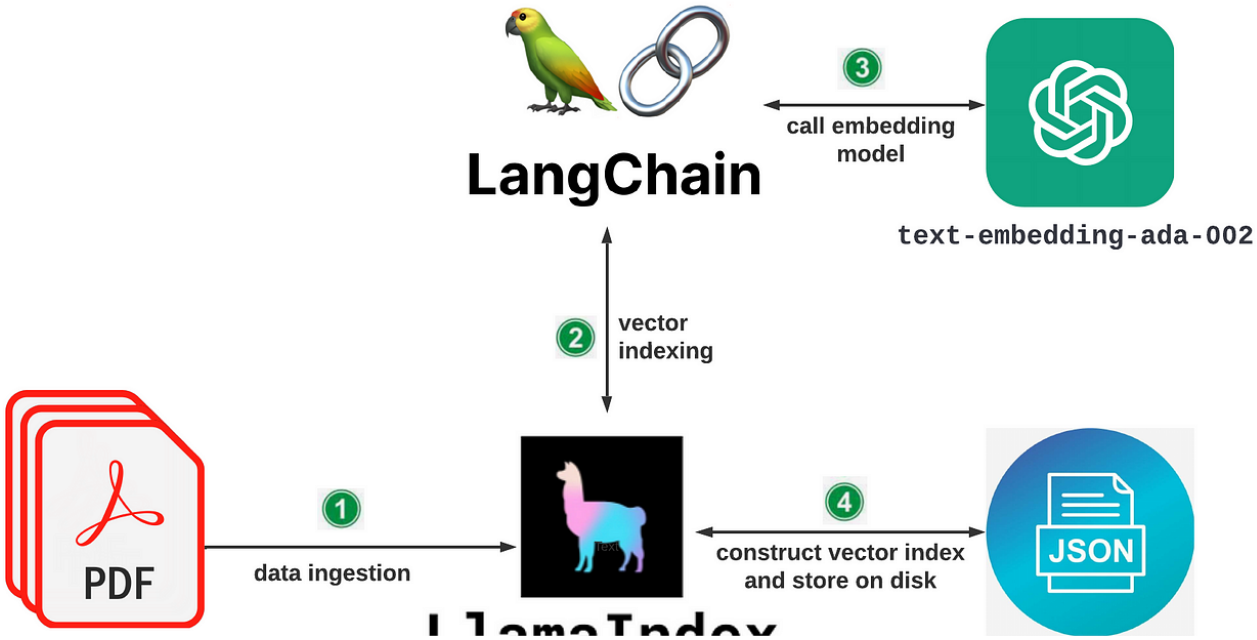
Ryan Nguyen in Towards AI

So, You Want To Improve Your RAG Pipeline

Ways to go from prototype to production with LlamaIndex

· 9 min read · Sep 27

176 2

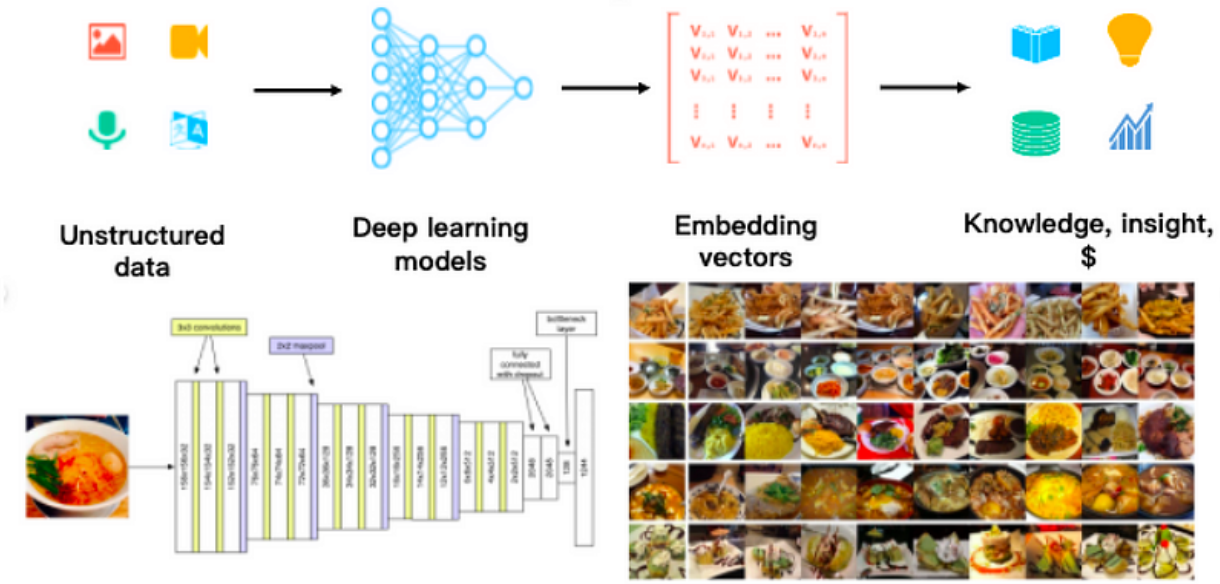


Wenqi Glantz in Better Programming

Building Your Own DevSecOps Knowledge Base with OpenAI, LangChain, and LlamaIndex

Building your custom knowledge base chatbot

· 11 min read · Jun 24



Jayita Bhattacharyya in GoPenAI

Primer on Vector Databases and Retrieval-Augmented Generation (RAG) using Langchain, Pinecone &...

Vector Databases Generation (RAG) Langchain Pinecone HuggingFace Large Language model generative ai

9 min read · Aug 16

226 1

Shaelander Chauhan

“Using Llama Index: Querying Databases in Plain English without SQL Expertise”

With Llama Index, users can ask questions in plain English instead of writing complex SQL queries.

2 min read · Jul 17

62