Weights & Biases

W&B Fully Connected > Articles

Building Advanced Query Engine and Evaluation with LlamaIndex and W&B

This report showcases a few cool evaluation strategies and touches upon a few advanced features in LlamaIndex that can be used to build LLM-based QA bots. It also shows, the usefulness of W&B for building such a system.

Ayush Thakur

Created on July 20 | Last edited on August 1

Introduction

If you are trying to leverage an LLM to build a powerful search engine, a chatbot or any other **LLM-based system over your data**, LlamaIndex should be one of the first options to explore. Why is that? Well, you can literally build a search engine on top of your own data in just a few lines of code. And even that's just scratching the surface here.

As a Keras user for many years now, I think the design of LlamaIndex also follows the "progressive disclosure of complexity" philosophy. You can build a system in a few lines of code to set up a working baseline but can keep modifying any component to your requirement. Also, you can subclass the base component logic to build custom components.

In this report, we will first build a simple QA bot for the latest Llama2 paper. After all, I haven't had a chance to read it in full, but I do have questions the bot can answer. We'll set up a simple evaluation strategy and showcase a few advanced LlamaIndex usage to improve the baseline performance. We will also show the importance of Weights and Biases tooling in an LLM-based system design lifecycle.

Last thing: if you're building LLM-based apps or are curious about doing so, we'd like to recommend our c[illegible] It's free, it's interactive, and we th[illegible]

Share 3 comments 14 stars

Sign up for our free LLM course

Table of Contents

- Introduction
  - Table of Contents
- Simple QA Bot using LlamaIndex
- Weights & Biases Prompts
- Setting up Evaluation using LlamaIndex
  - Generating Questions using LlamaIndex
  - Evaluating for Hallucination
  - Evaluating the Retrieved Chunks
- Improving the Query Engine
  - Keyword Table Index
  - Cross Encoder Reranker
  - Active Retrieval Augmented Generation
- Managing Index with Weights and Biases
- Conclusion

Simple QA Bot using LlamaIndex

We can build a QA bot over the Llama2 paper with just a few lines of code. A system where LLM(s) are given access to private data without fine-tuning the model on that data and the resulting response from the LLM is augmented by the context provided by the document is called **R**etrieval **A**ugmented **G**eneration (**RAG**). A RAG system typically has two stages - **Indexing** and **Querying**.

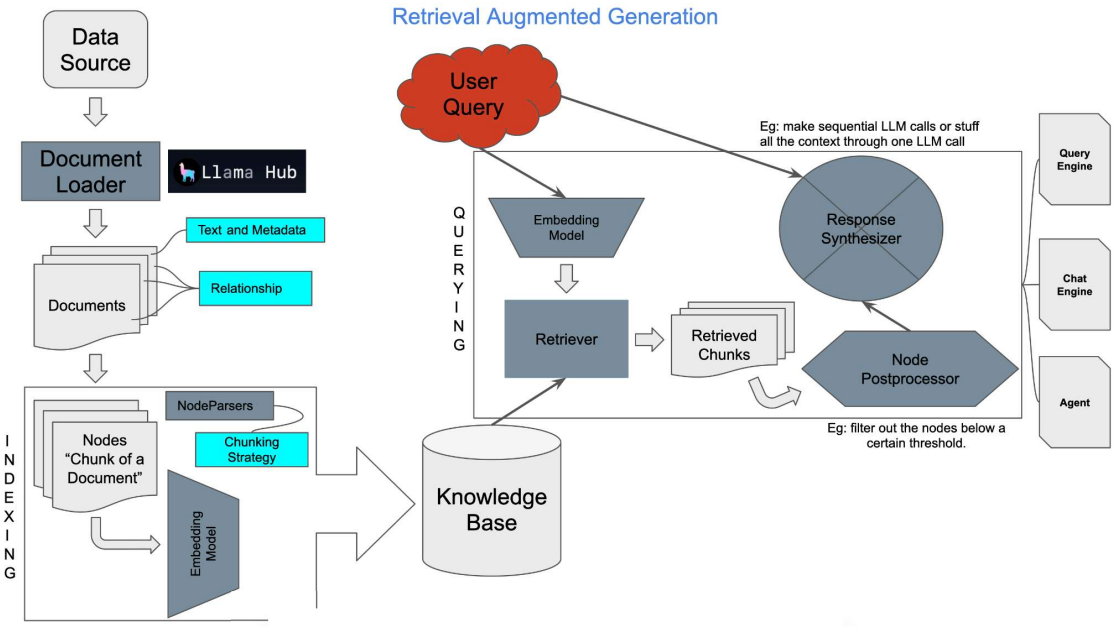


Figure 1: A simple archi tem.

- **Indexing**: In this stage, a data source is first loaded such that we have access to the text, associated metadata like page number, and the relationship between

each *document* (imagine it to be one page of a paper). The documents are then chunked into smaller units called *nodes*. Each node is then converted to a value (*embedding*) so that a *distance-measuring* computation can be done to find the most relevant chunk given a user *query*.

- **Querying**: In this stage, the user *query* is embedded, and a few chunks (context) are retrieved using a distance-measuring computation. This query and the retrieved chunks are sent to LLM(s) to generate *a response.* The query and the context are either concatenated (*stuff)* into a single string block (*prompt).* Or they can be sent to the LLM sequentially. The resulting response should ideally base the answer on the provided context.

Below, the QA bot uses all the components mentioned above, but the underlying logic, system prompts, distance computation for retrieval etc. are abstracted away.

```
!pip install llama-hub

from llama_index import VectorStoreIndex
from llama_hub.file.pdf.base import PDFReader

# Data Loader
loader = PDFReader()
documents = loader.load_data(file=Path('../llama2.pdf'))

# Chunking and Embedding of the chunks.
index = VectorStoreIndex.from_documents(documents)

# Retrieval, node poseprocessing, response synthesis.
query_engine = index.as_query_engine()

# Run the query engine on a user question.
response = query_engine.query("Who wrote this paper?")
```

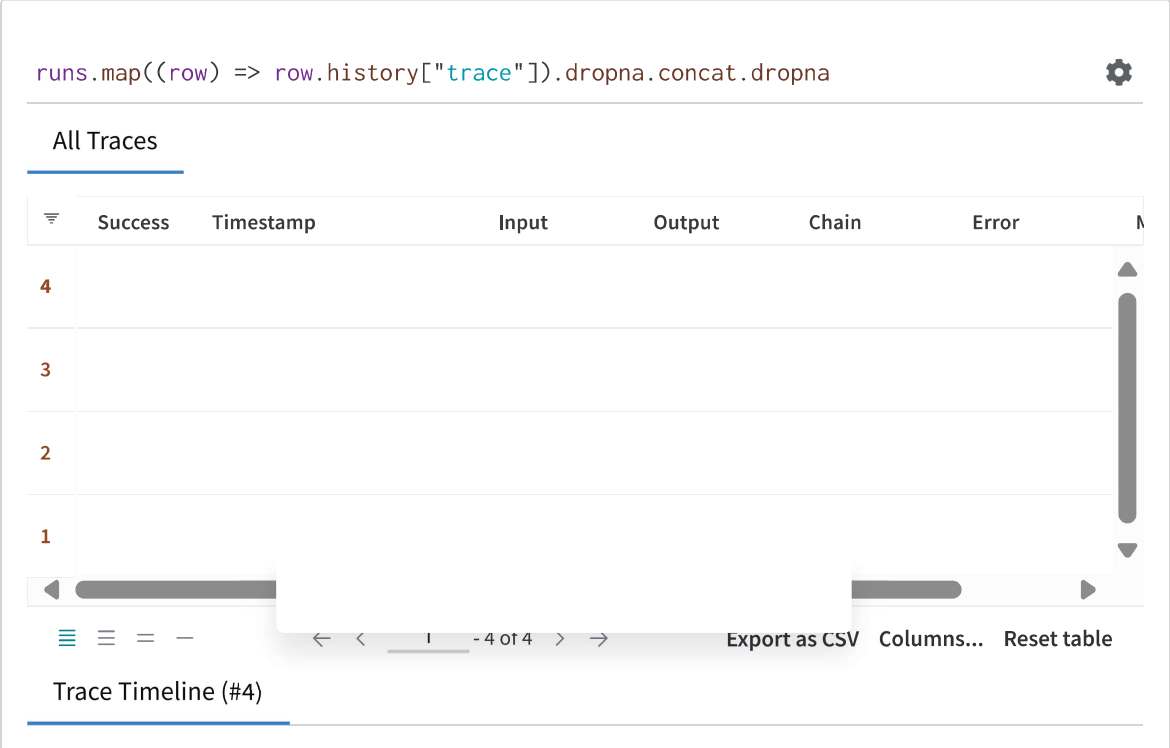
Weights & Biases Prompts

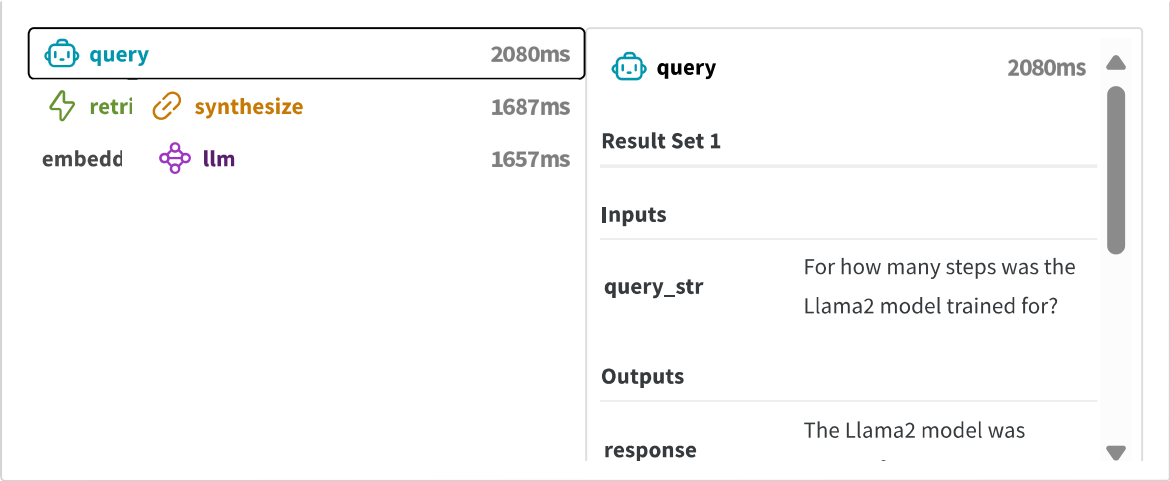
This is great! We can build such a system easily. But with abstraction comes the inability to peek inside. What happens when we want to know which nodes were retrieved? How many nodes were used as the context? Which LLM was used? How long did it take to generate the response? What were the intermediate steps?

W&B Prompts is powerful tooling to answering those exact questions. It gives us the ability to peek into the intermediate steps—what's more, it's easy to use. In software engineering, tracing involves specialized logging to record information about a program's execution. Programmers typically use this information for debugging purposes. Inspired b of the LlamaIndex trace (order of execu

Luckily, W&B Prompts is instrumented with LlamaIndex and can be used using the `WandbCallbackHandler`. Let's see how we can use it:

```
from llama_index import VectorStoreIndex
from llama_index import download_loader
from llama_index import ServiceContext
from llama_index.callbacks import CallbackManager, WandbCallbackHandler

# Data Loader
PDFReader = download_loader("PDFReader")
loader = PDFReader()
documents = loader.load_data(file=Path('../llama2.pdf'))

# initialise WandbCallbackHandler and pass any wandb.init args
wandb_args = {"project":"llama-index-report"}
wandb_callback = WandbCallbackHandler(run_args=wandb_args)

# pass wandb_callback to the service context
callback_manager = CallbackManager([wandb_callback])
service_context = ServiceContext.from_defaults(callback_manager=callback_manager)

# Chunking and Embedding of the chunks.
index = VectorStoreIndex.from_documents(documents, service_context=service_context)

# Retrieval, node poseprocessing, response synthesis.
query_engine = index.as_query_engine()

# Run the query engine on a user question.
response = query_engine.query("Who wrote this paper?")
```





Above, you'll see Prompts in action. Let's look at the question up top: "For how many steps was the Llama2 model trained?" The QA bot responded, "The Llama2 model was trained for 2 trillion tokens". Factually this response is correct but is this response actually answering the question? Nope. Sounds like we should consider setting up an evaluation pipeline.

Setting up Evaluation using LlamaIndex

We have a baseline QA bot, but to even think about improving it, we need a baseline score. This is where evaluating an LLM-based system comes into play. This is a tricky topic and it depends on the system you are trying to build.

In this W&B report, "How to Evaluate, Compare, and Optimize LLM Systems?" I have tried to cover the whats and hows of evaluating an LLM-based system. We will not go into detail, but broadly speaking, there are three main categories:

- **Eyeballing**: while building a baseline LLM system, we usually eyeball to evaluate the performance of our model.
- **Supervised**: This is the recommended way of evaluation where you involve humans to generate an annotated eval dataset for evaluation.
- **LLMs evaluate LLMs**: In this paradigm, we leverage a powerful LLM to generate proxy targets based on some context. In our case of a QA bot, we can ask an LLM to generate question-answer pairs.

Generating Questions using LlamaIndex

Using LlamaIndex's, `DatasetGenerator`, we can easily generate questions that can be used to evaluate using one of the following strategies but not limited to:

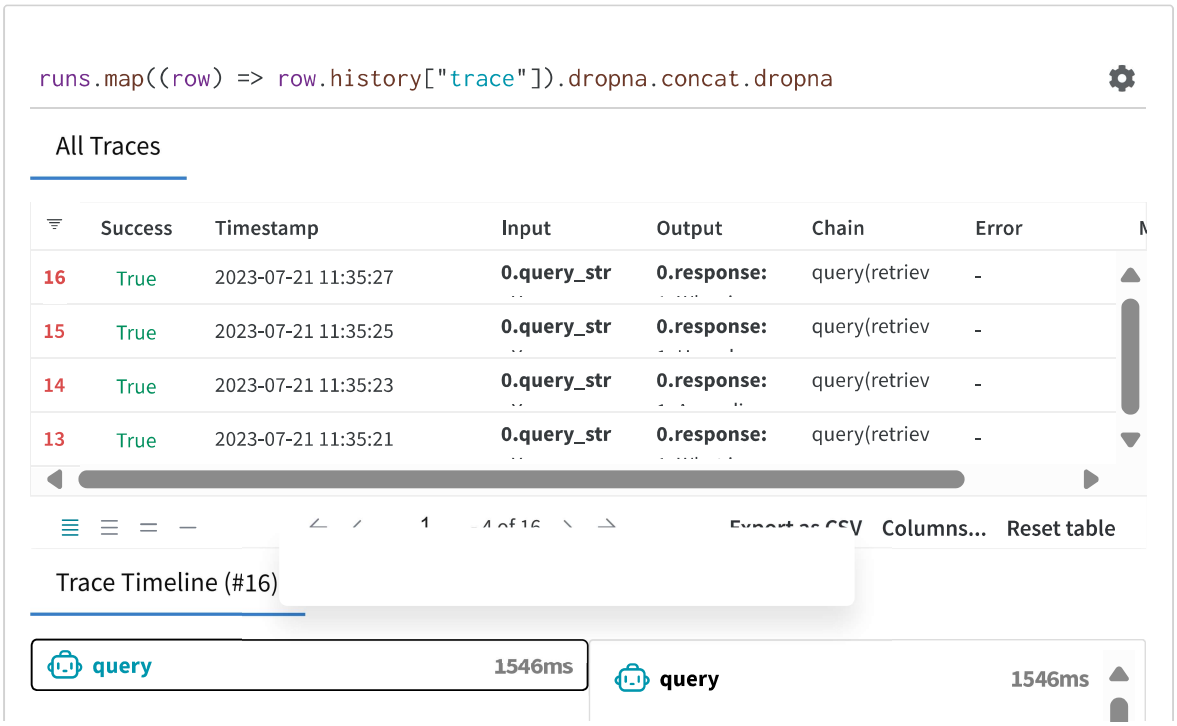
- **Evaluating respor** nse coming from the provided context,
- **Relevance of the retrieved chunks**: Evaluate each retrieved chunk (node) against the generated response to see if that node contains the answer to the query.

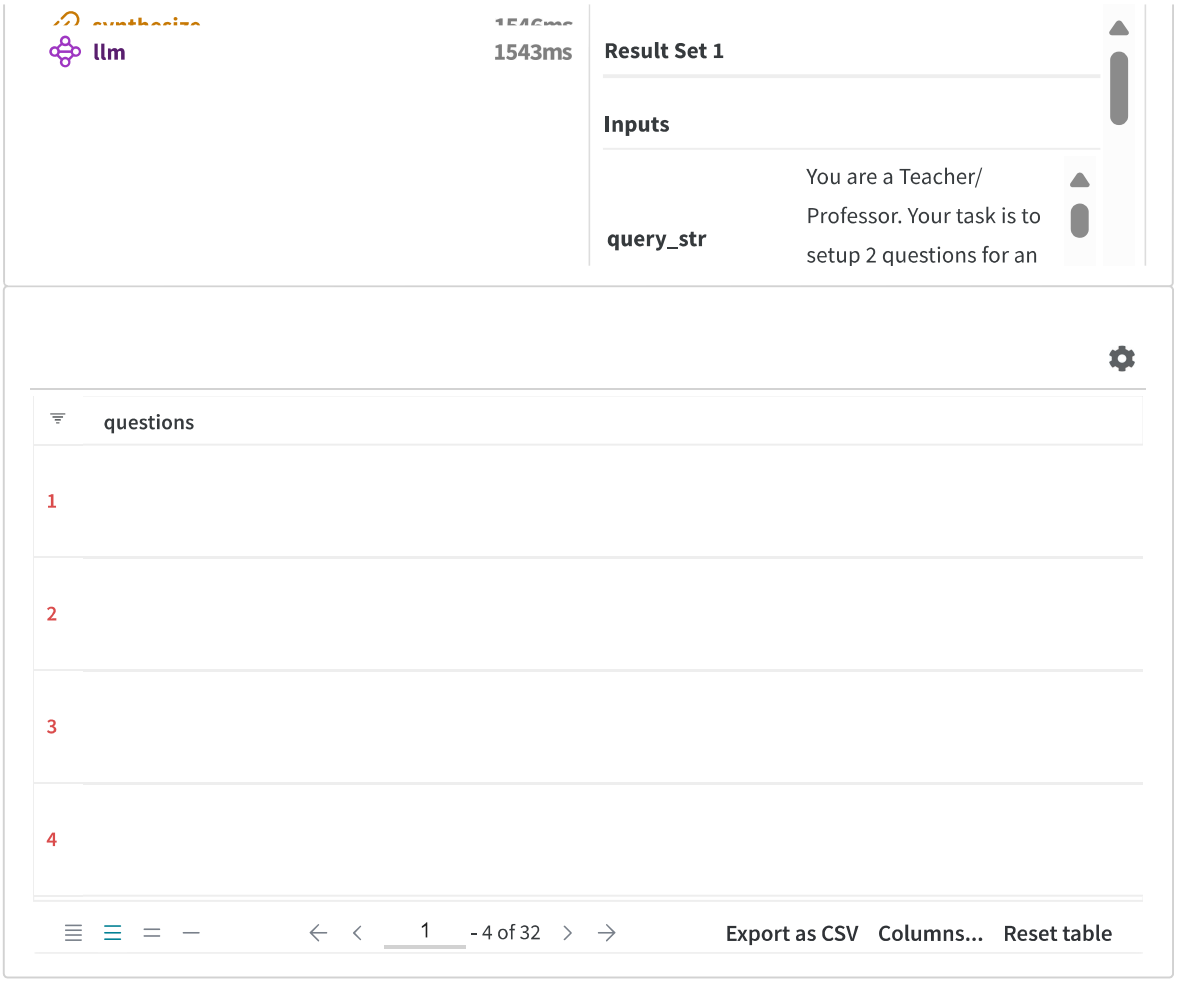
- **Evaluating the answer quality**: Does the query + generated response come from the provided context?

Using `DatasetGenerator` is easy, and one can pass the loaded documents to the `DatasetGenerator.from_documents` method. Calling `generate_questions_from_nodes()` on the object's instance will generate N questions per chunk. The default chunk size is 512, and N is 10. You might quickly realize that it will take a long time and a lot of API calls to generate a lot of questions. Let's customize the data generation process.

```
# Let's use GPT 3.5 as our LLM of choice.
llm = OpenAI(temperature=0, model="gpt-3.5-turbo")
service_context = ServiceContext.from_defaults(llm=llm, callback_manager=callback_ma

# Let's just use a meaningful subset of the shuffled documents.
random_documents = copy.deepcopy(documents)
random.shuffle(random_documents)
random_documents = random_documents[:10]

# Let's reduce the number of questions per chunk.
data_generator = DatasetGenerator.from_documents(
    random_documents, service_context=service_context, num_questions_per_chunk=2
)
# Let's reduce the number of questions per chunk from 10 to 2.
```

Note that usage of `call_manager` in the `ServiceContext`. Yeah, we can even track this with Weights and Biases Prompts. Note that the input prompt is a template used for generating the questions. Check out the list of questions in the W&B Table below. After all: **it's always a good idea to store an expensive process**.





Evaluating for Hallucination

The LLM generates a response using the provided context (chunks). In this evaluation strategy, we ask another LLM (GPT-4) to say a YES if the response was generated using the provided context and NO otherwise.

One can implement this strategy in LlamaIndex using the `ResponseEvaluator` class. In this evaluation mode, we call the `evaluate` method of the instance of this class.

```
from llama_index.evaluation import ResponseEvaluator

# Let's use GPT 3.5 for evaluation.
llm = OpenAI(temperature=0, model="gpt-3.5-turbo")
service_context = ServiceContext.from_defaults(llm=llm)

# define evaluator
evaluator = Respons

# query index
query_engine = index.as_query_engine()
```

```
# Get evaluation result
response = query_engine.query(question.questions)
eval_result = evaluator.evaluate(response)
```

We will evaluate our QA bot on the generated questions. **The accuracy of the QA bot, as shown below, is less than 50%** (14/32 = 0.4375). Can we improve this score?

> 💡 Should we blindly trust an LLM-based evaluation? No but it's a decent indication. Note that in this report, the evaluation prompt was not tweaked. For a robust evaluation, bringing humans in the loop is crucial, especially for demanding usecases (eg: medical).

	question	llm_response	eval_result
1			
2			
3			

1 - 3 of 32 Export as CSV Colun

	eval_result	count
1	NO	
2	YES	

1

Hallucination Accuracy (%)

baseline_hallucination

43.75

Evaluating the Retrieved Chunks

In this evaluation strategy, we evaluate every chunk using an LLM. The LLM is provided with the ge[illegible] If the chunk contributes to the res[illegible]NO.

Here, we can use the same `ResponseEvaluator` class. The only difference is using the `evaluate_source_nodes` method to pass the generated response. This evaluation used the GPT-4 model instead of GPT-3.5 to leverage a larger context window for future evaluations where the number of retrieved documents can be large.

Each retrieved chunk (in this case, chunk 2) was evaluated using GPT-4 against the generated response. If, according to GPT-4, the chunk contributed to the response, it responds with a YES (and NO otherwise). It was then converted to `eval_score` by scoring each YES to 1 and NO to 0 and normalizing by the number of chunks. The idea is to retrieve the most relevant chunks (`eval_score` = 1).

Shown below is also the overall "Retrieval Accuracy". It was computed by adding individual `eval_score` and dividing it by the total number of rows. A Retrieval Accuracy of 1 will indicate that all the retrieved chunks contribute to the generated response.

	question	retrieved chunks	llm_response	raw_eval_result	eval_score
10					
11					
12					

10 - 12 of 32 Export as CSV Columns... Reset table

Retrieval Accuracy

0.6406

Improving the Query Engine

The Query Engine, as shown in Figure 1, consists of multiple components. In this section, we will look into using a few advanced components to improve the overall performance of our QA bot.

Keyword Table Index

Till now, we have been using the embedding-based index, where each chunk is embedded using an embedding model. During query time, the query is also embedded with the same model. We then take the query and stored embedding (in a vector store) and compute the cosine similarity (a distance measuring function) between them. We then select the top K chunks closest to the query embedding. This works but is tightly dependent on the chunks (chunk size, overlapping of chunks, etc). Interpreting embeddings is hard as well.

The keyword table index extracts keywords from each Node and builds a mapping from each keyword to the corresponding Nodes of that keyword. The keywords can be extracted using regex matching (SimpleKeywordTableIndex), or an LLM can extract relevant keywords (KeywordTableIndex). This can be particularly useful for not selecting chunks with profanity or sensitive information. Under the hood, this is a simple indexing strategy and might not be on par with embedding-based indexing.

Building this index is simple: replace the `VectoreStoreIndex` with `KeywordTableIndex` and pass the documents to it. `SimpleKeywordTableIndex` is very fast (regex matching) while building the `KeywordTableIndex` will take a while.

```
keyword_index = KeywordTableIndex.from_documents(documents)
```

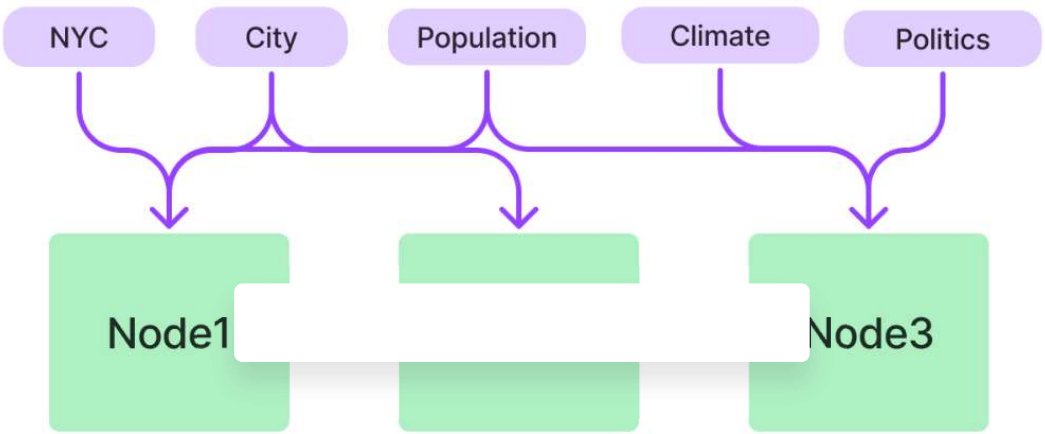


Figure 2: Extract keywords from the chunks and create a mapping table for them. (Source)

Using the same evaluation strategy above, the hallucination accuracy improved from ~43% to ~59%. Note that at the same time, weirdly, the retrieval accuracy dropped by 65%. We can say that the generated responses used the provided context (low hallucination), but the retrieved contexts were irrelevant. Given the simple nature of this index, the retrieved chunks might not be relevant after all. We will not dive into further investigation and accept the generated metrics.

Hallucination Accuracy (%)	Retrieval Accuracy
keyword_table_index	keyword_table_index
59.375	0.01667

Cross Encoder Reranker

Any retrieval mechanism retrieving multiple chunks from a large document will be efficient to a degree, but it will still select some irrelevant candidates. **Re-ranking** the retrieved chunks and then dropping the lower-scored chunks from the final context can help improve the context provided to the LLM.

One such popular and easy-to-use re-ranking strategy is the cross-encoder reranker. This reranker uses a Transformer based model like a BERT, which is fast to run on modern hardware. It has a binary classification head on top of it. The user query and a candidate chunk are passed to the model. The head outputs a score of 0 or 1, indicating whether we should keep or drop the chunk. One can also take the logits of the head and use them as a confidence score. The chunks can be re-ranked based on the score, and then the top K is selected from the lot.

The name cross-encoder comes from BERT being an encoder-type transformer and attention happening across the query and the document. In practice, the retriever selects, say, 10-100 candidate chunks (depending on the speed of the retriever), and the cross-encoder narrows it down to, say, 3 top chunks.

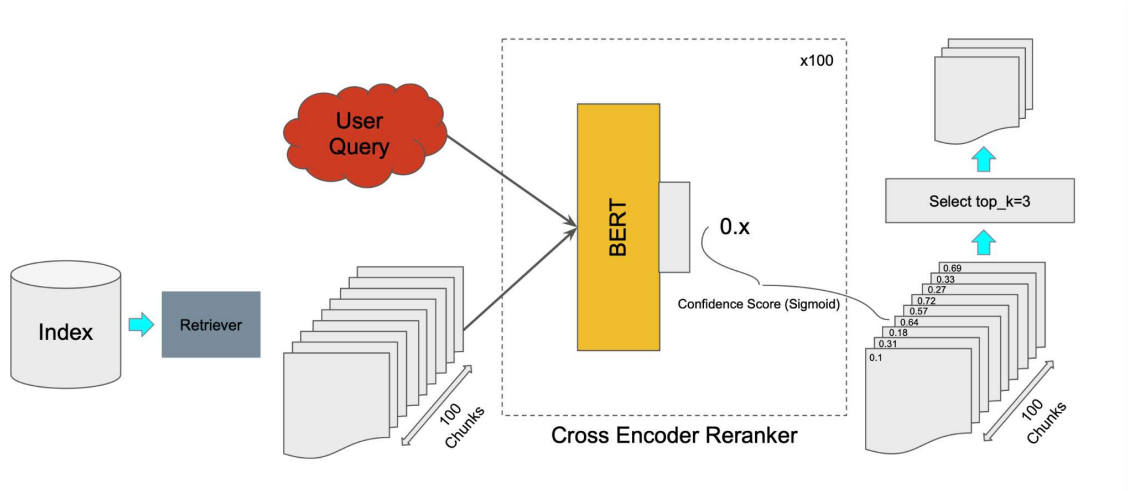


Figure 3: Cross Encoder Reranker

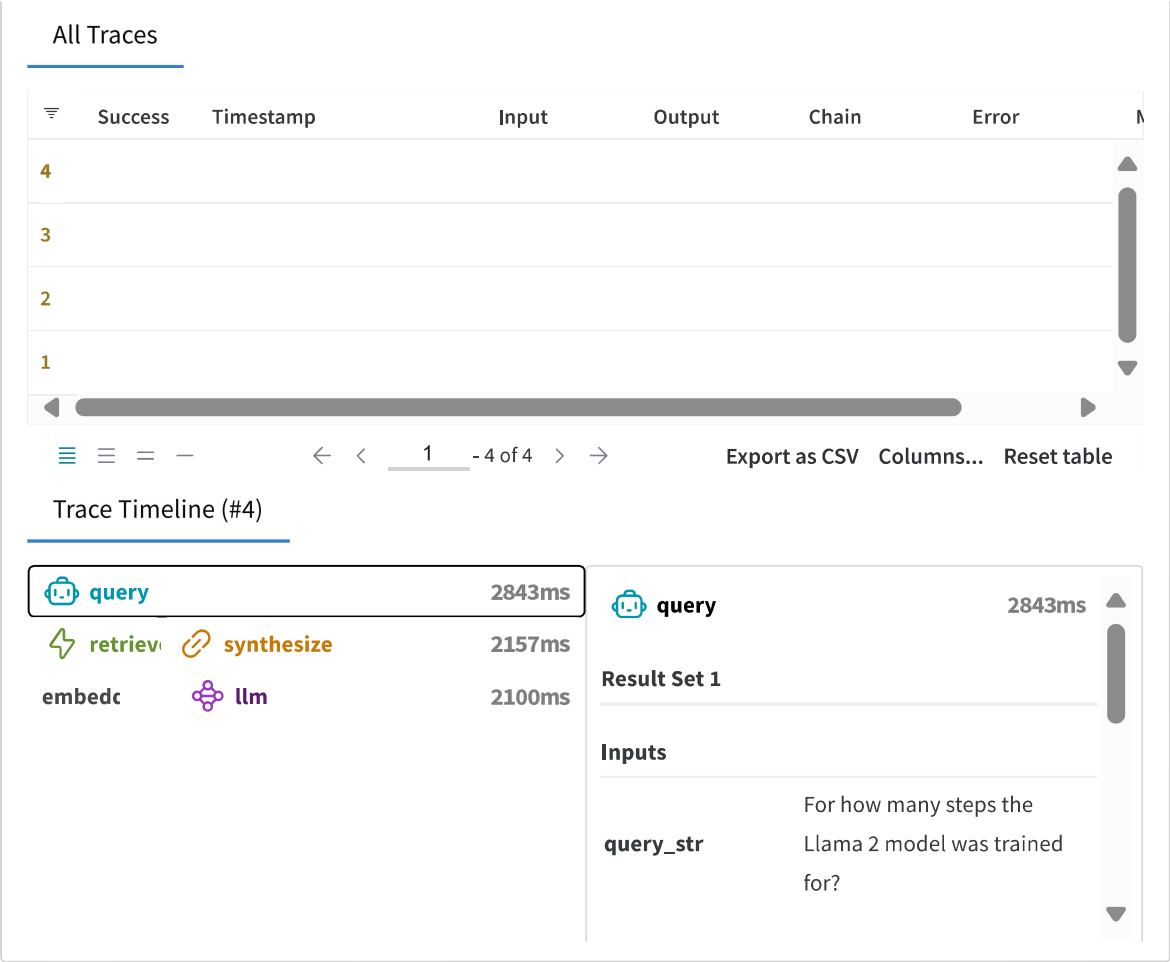
Luckily, implementing this in LlamaIndex is very easy. We need the `sentence-transformers` library, which one can install using `pip install sentence-transformers`. We can use any retriever of choice - in this case, we will use the embedding-based (`VectorStoreIndex`) index (the retriever sits on top of the index). Under the hood, the re-ranker is a sentence transformer model - `cross-encoder/ms-marco-MiniLM-L-2-v2`. You can find the full list of relevant models here. Check out the code snippet below to use a cross-encoder re-ranker.

💡 A re-er in LlamaIndex's terminology is a "Node Postprepocessing" step as shown in Figure 1.

```
from llama_index.indices.postprocessor import SentenceTransformerRerank

# Create an index of choice
index = VectorStoreIndex.from_documents(documents, service_context=service_context)

# Initialize the reranker
rerank = SentenceTransformerRerank(
    model="cross-encoder/ms-marco-MiniLM-L-2-v2", top_n=3 # Note here
)

# Build the query engine
query_engine = index.as_query_engine(similarity_top_k=10, node_postprocessors=[reran
```

You can then use this query engine to `query` a question. To the question - "Who wrote this paper?" the QA bot gave a long list of true names. No other query engine has been able to answer this question so far. The bot, however, was confused about the number of steps required to train the model - it said pre-training and fine-tuning, which might be cons

```
runs.map((row) => row.history["trace"]).dropna.concat.dropna
```



Active Retrieval Augmented Generation

So far, we have looked into single-time retrieval augmented LLM generation, i.e., the chunks (context) were retrieved only once. This seems to outperform even fine-tuned LLMs on short-form knowledge-intensive generation. But most applications deal with long-form documents, and improving the retrieved context's quality is not straightforward - the user's query might not have relevant information for quality retrieval. Even as humans, we don't usually come up with the most complete question.

To answer a relatively vague question, we keep collecting new information. What if the retrieval system has the ability to decide when and what to retrieve across the course of generations?

The authors of Active Retrieval Augmented Generation proposed a **F**orward-**L**ooking **A**ctive **RE**trieval augmented generation (FLARE) method. This method has two variants - direct (uses M). Below we discuss and build usi

In the instruct-based FLARE method, we start with the input $\mathbf{x}$ (query) and some retrieved chunks D_x. Some response y_1 is generated. Based on the response, the LLM generates further questions in the format of "[Search(query)]". Once a search query is generated, the generation process is stopped, and the new query is used to retrieve chunks which are then prepended before the user input to generate a further response. This process repeats N times to generate the final response.

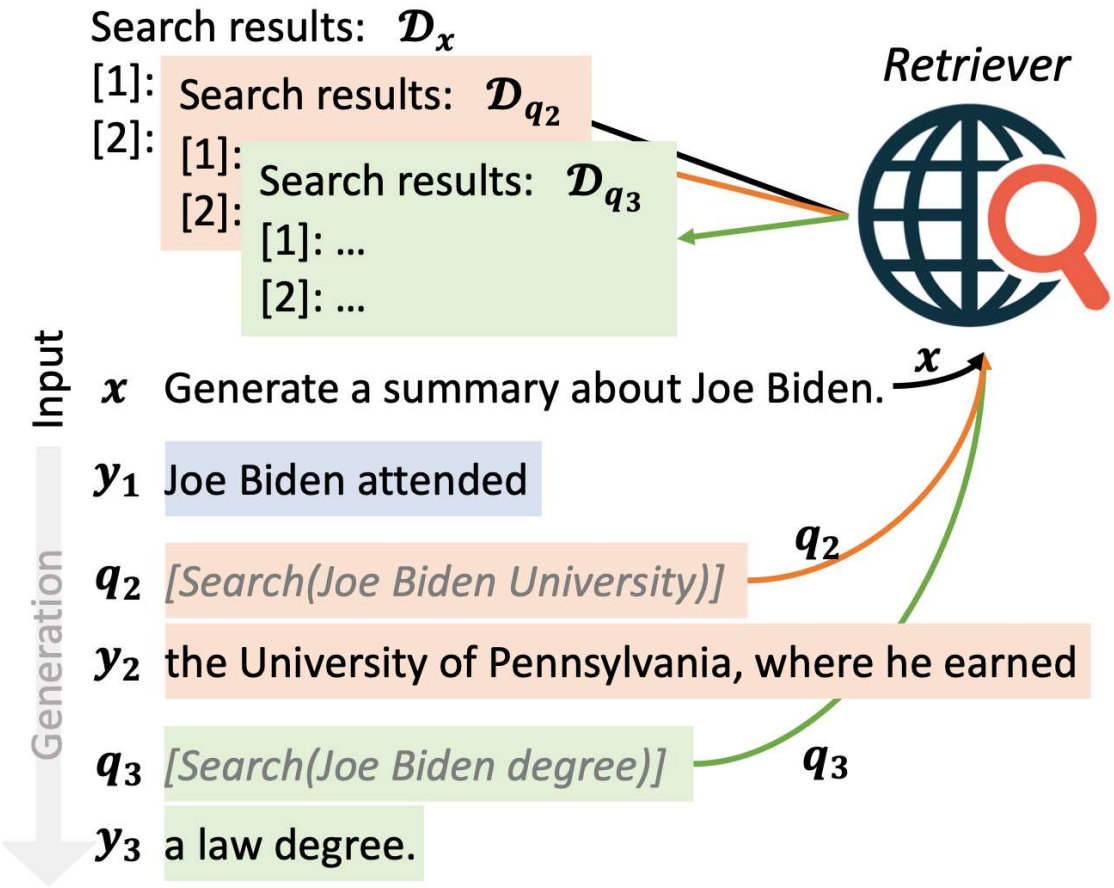


Figure 4: FLARE with Retrieval Instruction. (Source)

LlamaIndex already has the implementation of FLARE (instruct) with an easy-to-use API, as shown below.

```
# Import
from llama_index.query_engine import FLAREInstructQueryEngine

# Create an index and associated query engine
index_query_engine = VectorStoreIndex.from_documents(documents).as_query_engine()

# Initialize the FLARE (instruct) query engine.
flare_query_engine = FLAREInstructQueryEngine(
    query_engine=in
    service_context=
    max_iterations=7,
    verbose=True,
```

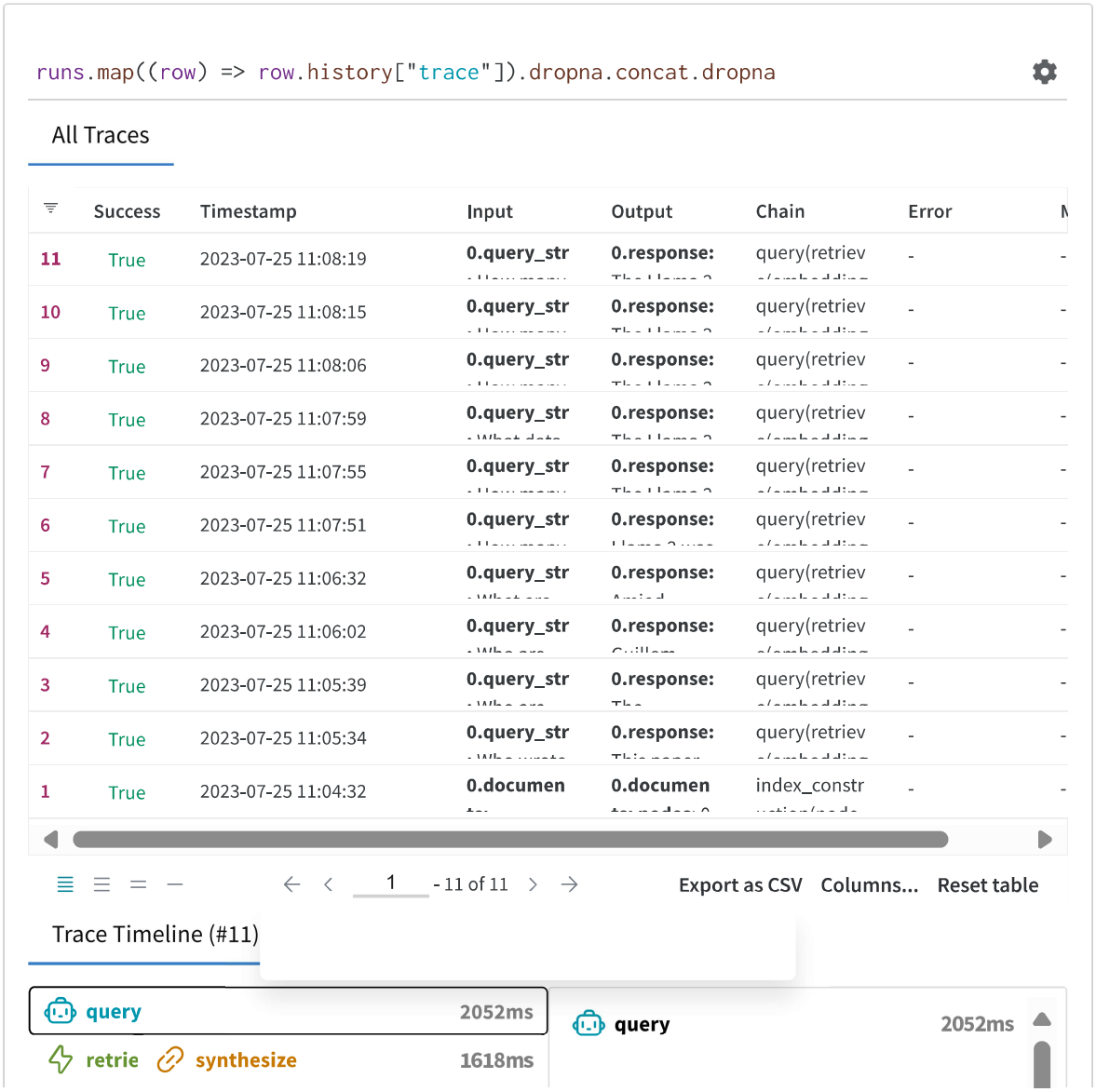
```
)
```

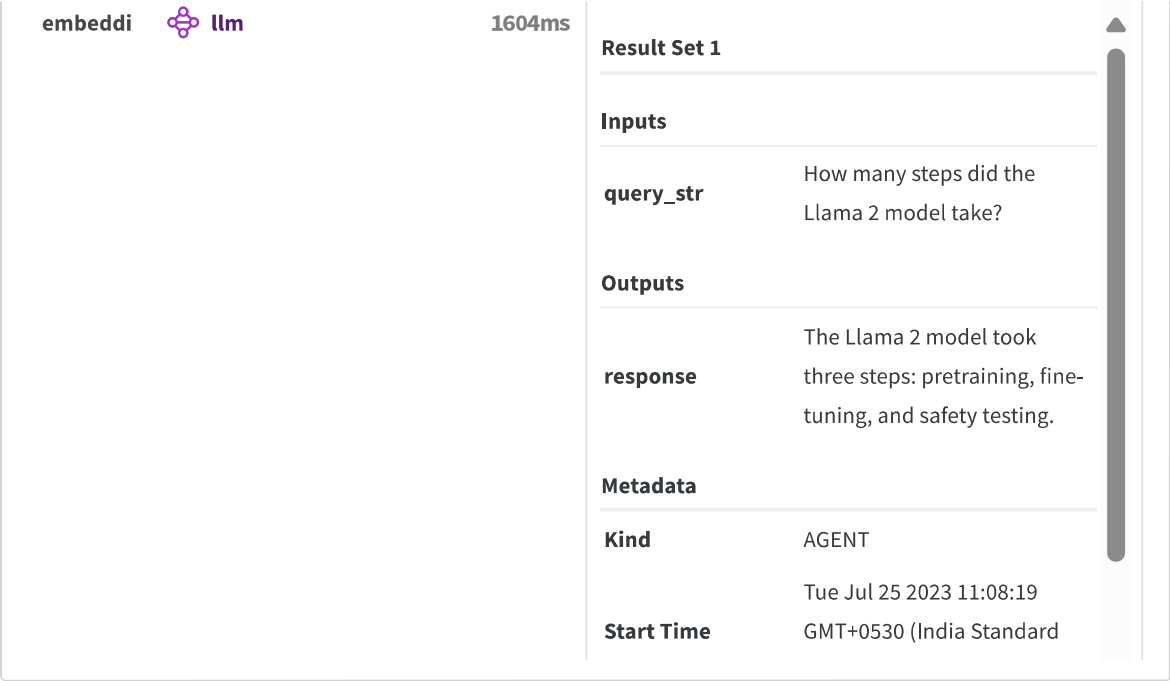
To the question, "For how many steps the Llama 2 model was trained for?"

The QA bot responded,

> "The Llama 2 model was trained for approximately 6 months steps. The Llama 2 model took 2 trillion tokens of data for pretraining. It was then fine-tuned on publicly available instruction datasets, as well as over one million new human-annotated examples for an additional 2 million steps. The Llama 2 model took two steps for fine-tuning: supervised fine-tuning (SFT) and reinforcement learning with human feedback (RLHF). The SFT step took 1 million steps and the RLHF step took 1 million steps. The total number of steps for the Llama 2 model was 3 million. The total number of steps for the Llama 2 model was 3 million."

This response has far more dense information than the previous query engine's response. Below you can see how the FLARE instruct created new queries and generated a response for them (read from bottom to top). The response from all the generated and user queries are concatenated to generate the final response shown above.





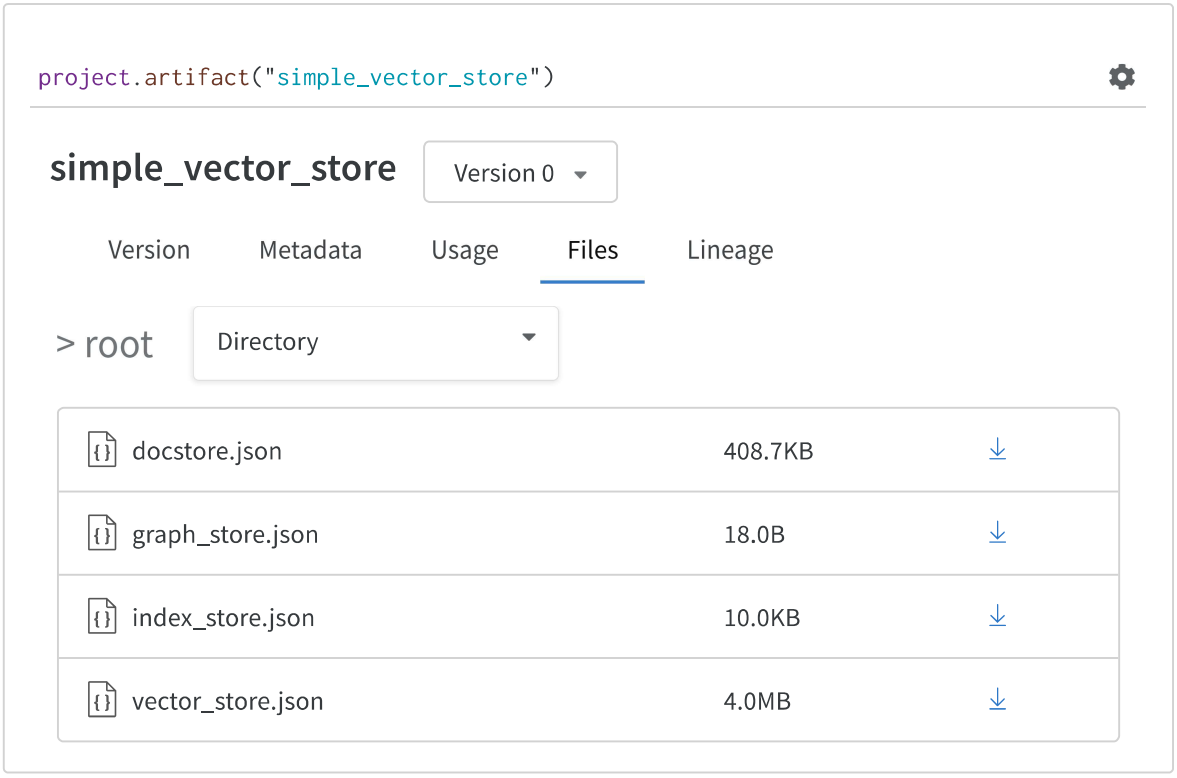
Managing Index with Weights and Biases

Generating an index is both a time taking process and costs some money. This is very much true when trying to embed a huge document. We would typically not want to repeat this process every time we initialize the QA bot. Well, the obvious action is to store the index (save it) and later load the same index.

You can use the `WandbCallbackHandler` not only to visualize and inspect the execution flow of your index creation, query engine, etc. but also use this handler to persist (save) your created indices as W&B Artifacts, allowing you to version control your indices.

```
from llama_index import ServiceContext
from llama_index.callbacks import CallbackManager, WandbCallbackHandler
from llama_index import load_index_from_storage

# initialise WandbCallbackHandler and pass any wandb.init args
wandb_args = {"project":"llama-index-report"}
wandb_callback = WandbCallbackHandler(run_args=wandb_args)

# pass wandb_callback to the service context
callback_manager = (
service_context = ServiceContext.from_defaults(callback_manager=callback_manager)

# Create the index
```

```
index = VectorStoreIndex.from_documents(documents, service_context=service_context)

# Save the index as W&B Artifacts
wandb_callback.persist_index(index, index_name="simple_vector_store")

# Load the index from the W&B Artifacts
storage_context = wandb_callback.load_storage_context(
    artifact_url="ayush-thakur/llama-index-report/simple_vector_store:v0" # Click on
)

# Load the index and initialize a query engine
loaded_index = load_index_from_storage(storage_context, service_context=service_cont
```



Conclusion

LlamaIndex is a powerhouse in terms of the features it provides. However, the main takeaway is the ability of LlamaIndex to disclose the complexity progressively. One can build a quick bot in a few lines of code, eyeball the results and decide to change the LLM, use a different chunk size for document parsing, or filter out chunks below a certain threshold. The list can go on. This report tried a few interesting features we get out-of-the-box usi

Evaluation is a tricky area and needs more research. I believe the best form of evaluation is to bring the stakeholders (humans) in the loop. For many applications,

evaluation can be strategically thought of - build a crude system, ship it and let the users interact with it, build a way for the users to provide feedback and use it as the gold-standard eval set. For other use cases, like prescribing medicines based on the diagnosis - a more robust evaluation is necessary.

Finally, W&B Prompts integration with LlamaIndex can help debug, visualize and inspect under-the-hood execution flow of most query engines. This PR added the said integration, and we welcome to improve and expand this. :) LLM-based systems require three crucial components besides LLM itself - a powerful tool to interact with the LLM, useful tooling (like W&B Prompts) and a proper evaluation strategy. You get the first two with LlamaIndex and Weights and Biases, while evaluation is an open-ended and subject-to-use case topic.

I hope you find this report insightful and showed an interesting trick. If you have any questions/suggestions, please drop in the comment below or contact @ayushthakur0.

How to Evaluate, Compare, and Optimize LLM Systems

This article provides an interactive look into how to go about evaluating your large language model (LLM) systems and how to ...

A Gentle Introduction to LLM APIs

In this article, we dive into how large language models (LLMs) work, starting with tokenization and sampling, before exploring how to us...

How to Run LLMs Locally With llama.cpp and GGML

This article explores how to run LLMs locally on your computer using llama.cpp — a repository that enables you to run a model locally in n...

Prompt Engineering LLMs with LangChain and W&B

Join us for tips and tricks to improve your prompt engineering for LLMs. Then, stick around and find out how LangChain and W&B can ...

Tags: Art mediate

Created with ❤️ on Weights & Biases.

https://wandb.ai/ayush-thakur/llama-index-report/reports/Building-Advanced-Query-Engine-and-Evaluation-with-LlamaIndex-and-W-B--Vmlldzo0OTIzMjMy

Never lose track of
another ML project.
Try Weights & Biases today.

TRY W&B NOW

Weights & Biases
Get weekly updates with the latest ML news.

Subscribe

PRODUCTS
Dashboard Sweeps Artifacts Reports Tables

QUICKSTART
Documentation

RESOURCES
Courses Forum Tutorials Benchmarks

W&B
About Us Authors Contact Terms of Service Privacy Policy

Copyright ©2024 Weights & Biases. All rights reserved.