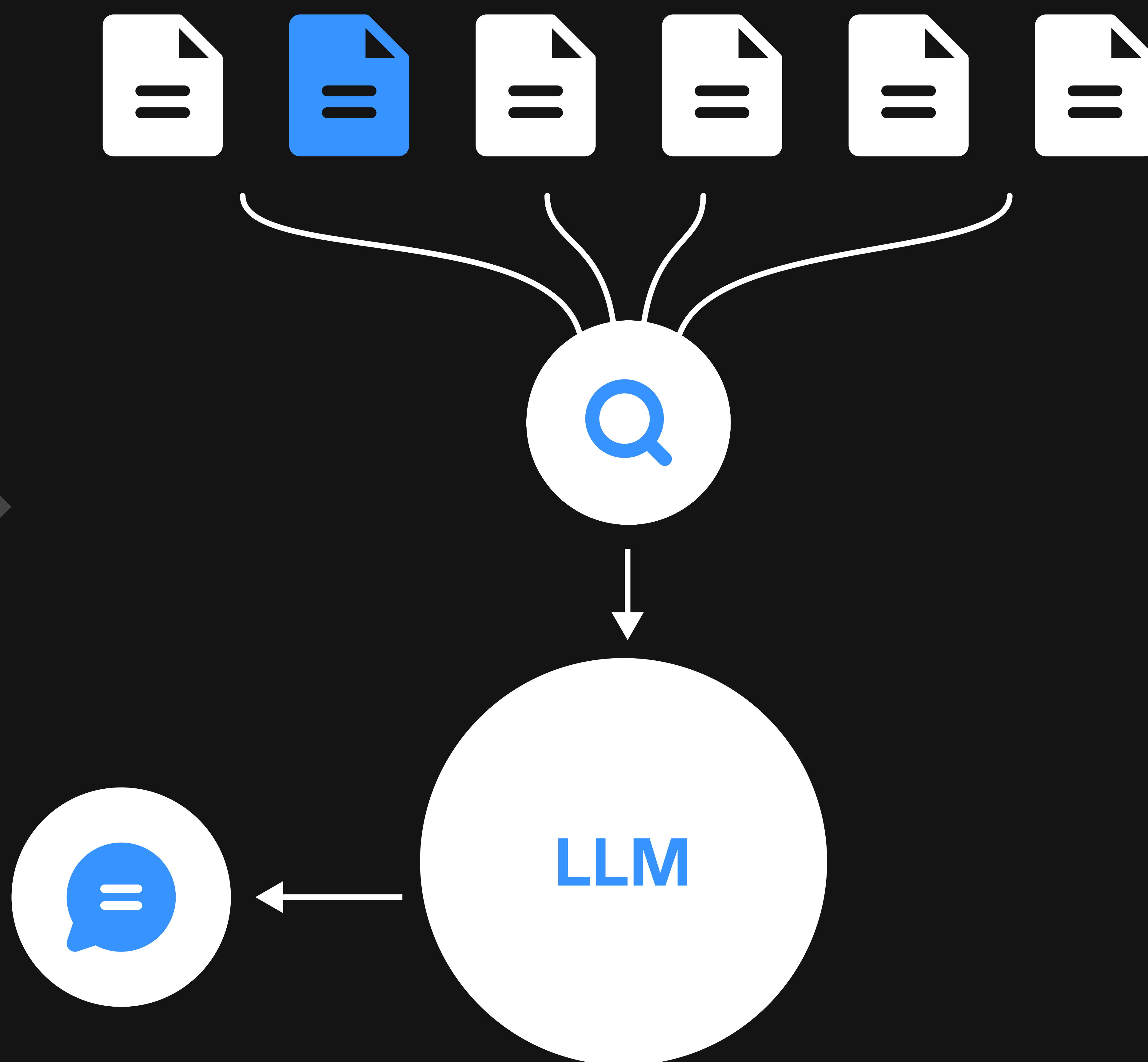
SHAKUDO

Vector Databases

Build an Enterprise Knowledge Base Using Vector Databases and Large Language Models



Vector Databases

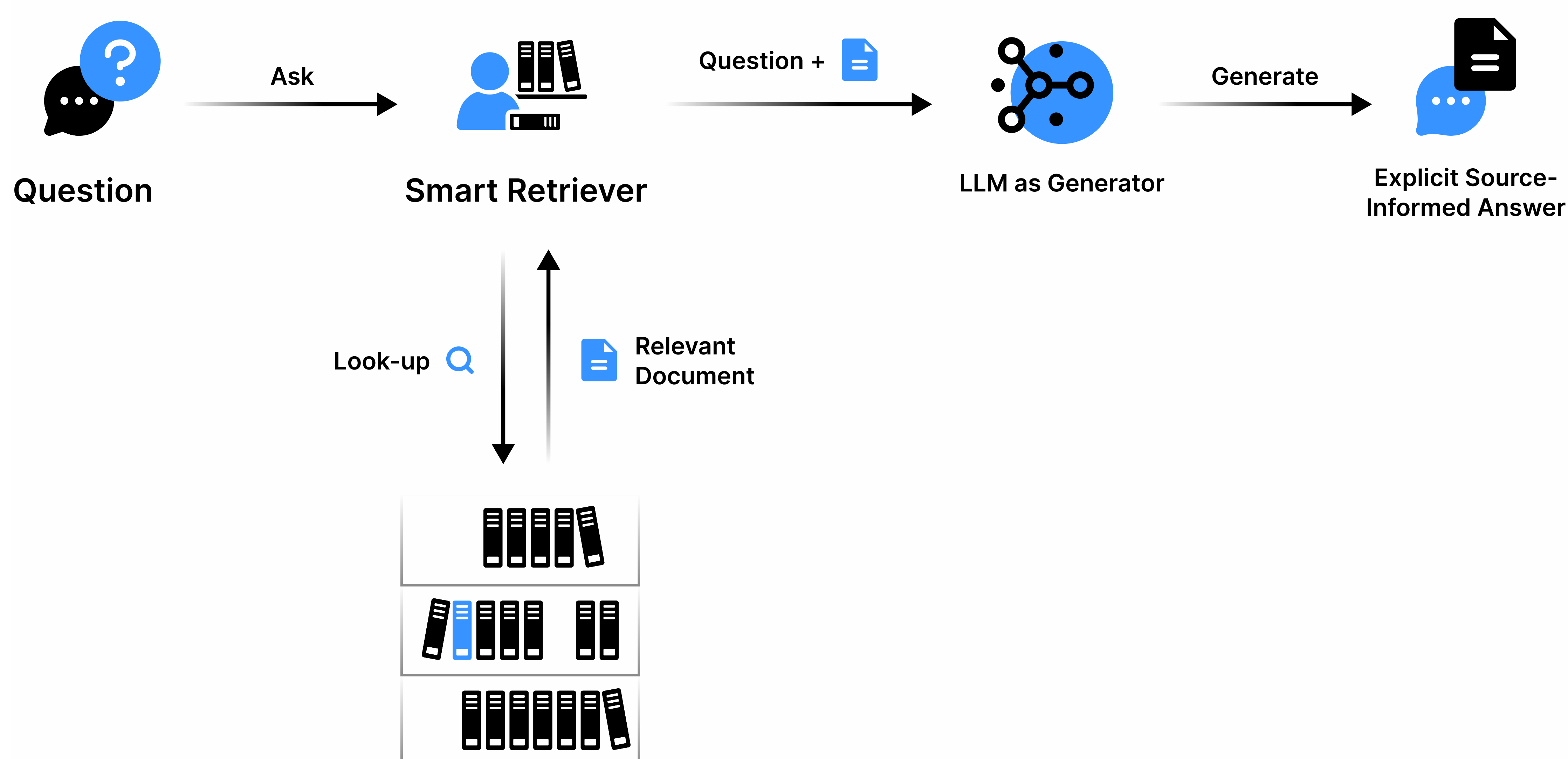
Build an Enterprise Knowledge Base Using Vector Databases and Large Language Models

SHAKUDO

Introduction

LLMs have unlocked a plethora of new use cases through their phenomenal text understanding and generation performance. One notable such use case is to interrogate internal knowledge through natural language queries — that is: to talk with your data. This is often referred to as “retrieval-augmented generation” (RAG). Combining existing knowledge bases with LLMs is a difficult process, with challenges ranging from engineering concerns (how to process, store, and retrieve the data) to R&D challenges (how and what to embed and generate) to ops hurdles (how to ensure the service stays up and scales). In this paper, we outline best practices in addressing several key challenges associated with developing and deploying production-grade RAG systems.

Retrieval Augmented Generation (RAG): what's behind the hype



Retrieval-augmented generation (RAG) architectures are a popular way to address issues in LLMs, such as their inability to answer questions about fresh or private content (not in their training data) and data accuracy in generated response.

As depicted above, a typical RAG architecture places a database in front of an LLM, and performs a semantic query between the user's prompt and the database to recover relevant documents. Then, those documents' text are made available to the LLM for natural language text generation.

Typically, those documents are really fragments from a larger document, for example paragraphs from a 300-page book. This allows an LLM to consume contents from multiple sources and to cross-reference different material to answer questions such as: "Which of these two companies saw more profit this year?"

RAGs can also be used to “extend” an LLMs memory, by saving previous user interactions in a vector store and recovering relevant chats to answer further questions.

Overall, they’re an efficient way to bypass context size limitations in LLMs.

Each segment of the rough diagram above represents multiple real-world challenges when brought to scale: How can we find relevant embeddings efficiently? How do we manage the documents so that the embeddings are relevant when we perform a search? How do we provide text fragments to the LLM so that it will answer from the data and not from imagination? How do we run this architecture efficiently for thousands or even millions of users?

In the following sections of this paper, we cover three main challenges associated with RAG-based LLMs: options for efficient retrieval, model tuning, and resource and cost management.

If you are not familiar with RAG implementation, are looking for a refresher, or simply want to quickly give them a spin, see our previous blog post, in which we cover RAGs with example code and data.

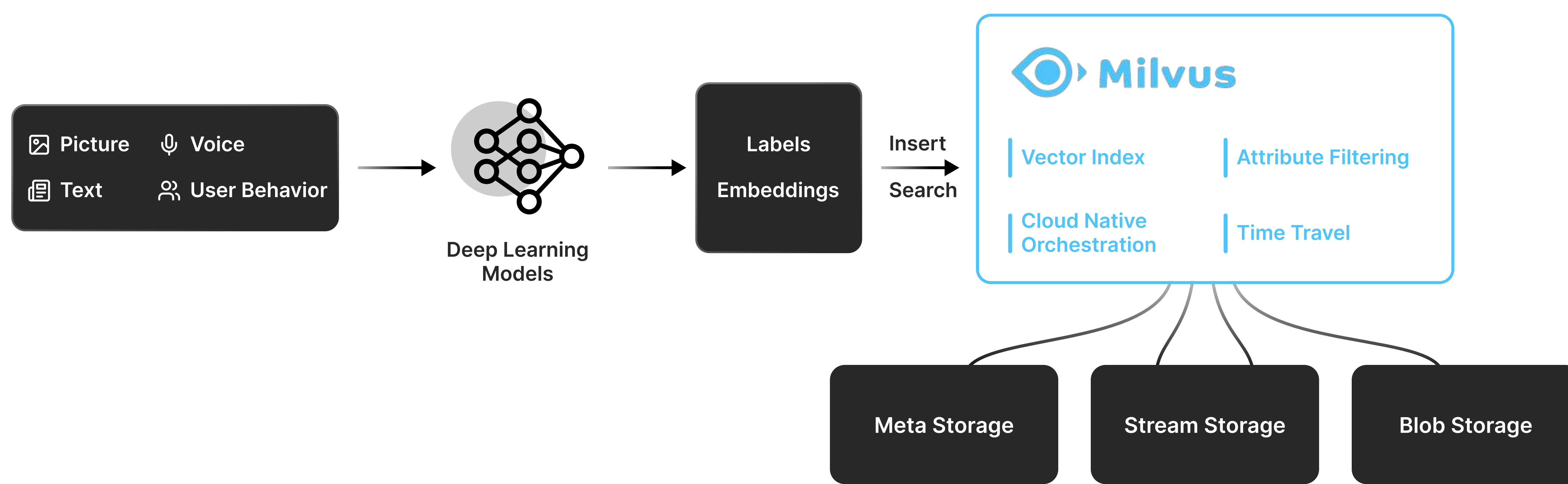
Vector Databases: The New Way to Query Data

While not all knowledge bases contain Google- or Amazon-scale data, there can be various complications, such as operating on large documents (300+ page PDF collections, videos, microsecond-resolution time data, etc.). Whether the dataset is composed of very many small documents, a medium amount of very large documents, or a mix of both, brute force approaches are usually not good enough, and cannot scale as the knowledge base grows.

In addition, since we expect natural language queries, we need semantic search with contextual relevance. For example, if the user asks for the price of adopting "cat," "cat" can be the stock ticker for "Caterpillar," or it could be the pet. Without contextual information (e.g., "adopting cat" clearly isn't about adopting a stock ticker), the search will return bad results from the database, and for the generation part of a RAG, it's Garbage In, Garbage Out.

The solution in practice is to use vector databases, which allow storing embedding vectors efficiently, and retrieving documents based on embeddings at interactive speeds. There are various vector stores available that provide different capability levels. Some notable options include Milvus, PgVector for Postgres, Pinecone, Weaviate, and Qdrant.

Setting up vector stores introduces additional challenges. For example, correctly partitioning large data which cannot fit entirely in RAM in vector stores like Milvus is not an easy endeavor. Doing it poorly can result in some queries taking up too much RAM and bringing the service down (under-partitioning), or having to perform too many probes to find a relevant document, which results in hitting slow permanent storage and reducing the RAG's responsiveness significantly (over-partitioning).

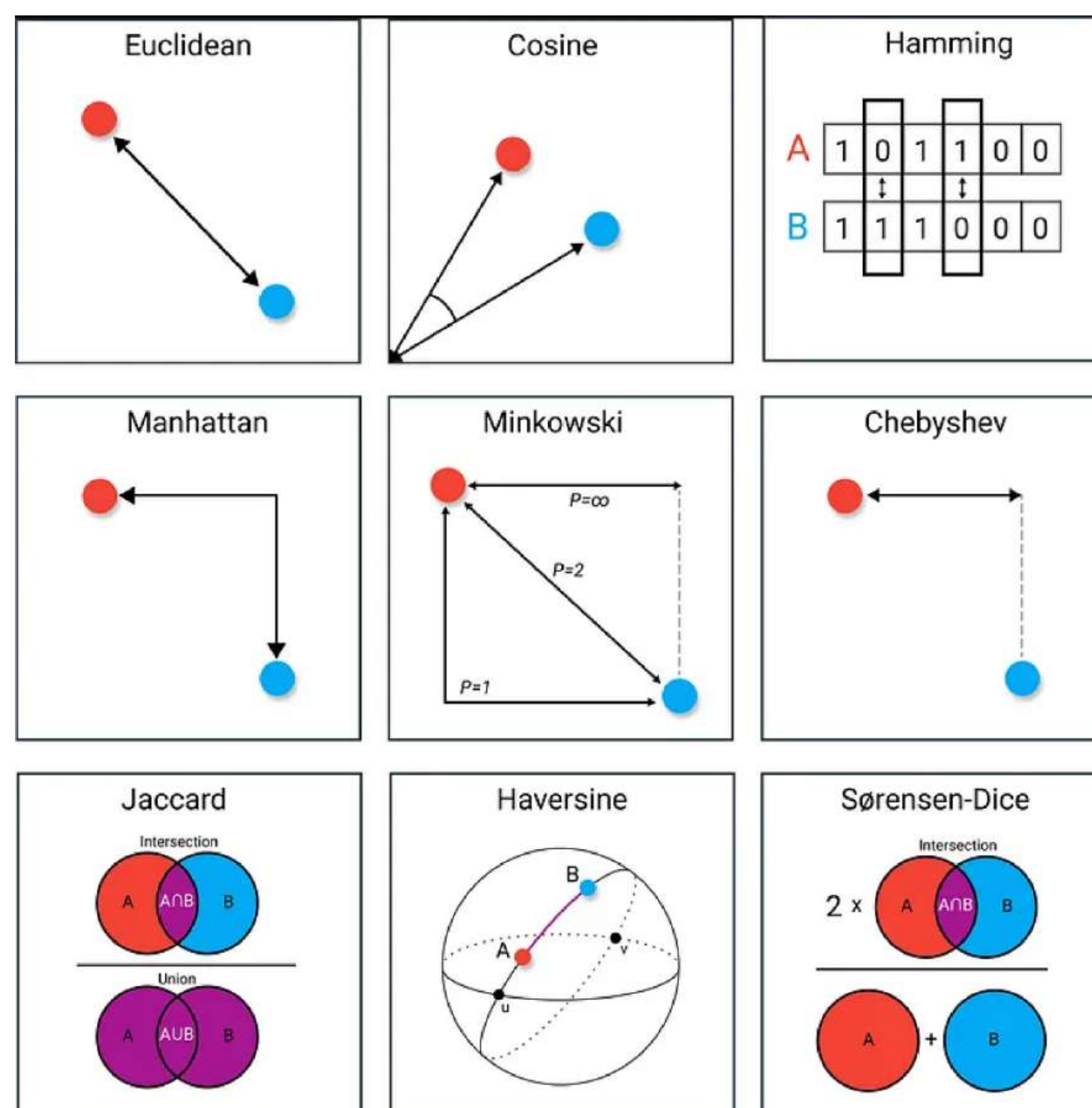


Furthermore, the quality of embeddings chosen, and the embedding methodology (what to embed and how much of it) is its own complicated topic. Poor choices can break a RAG altogether, while good choices will make the downstream generation task a lot easier. It's not enough to select whatever embedding model tops a benchmark, because the benchmark may not be using data resembling your specific knowledge base documents. Just because a model is great at embedding data from a Reddit threads dataset doesn't mean it is good at embedding financial statements or business case studies. Good embeddings depend on the embedding model used, but good retrieval of those embeddings depends on using the right search algorithm and the right distance function to compare various embeddings. Two common distances are shown below:

Cosine similarity: $$\frac{\sum_i A_i B_i}{\sum_i A_i^2 \sum B_i^2}$$

L2 distance: $$\sum_i (A_i - B_i)^2$$

It’s important to understand the kind of embeddings used since this affects how search metrics behave. Many modern embedding models use normalized embeddings, but not all. Common search metrics like L2 and Cosine similarity will return different results in the case of unnormalized embeddings. As you can see in the equation, Cosine similarity divides by the vector norms while L2 doesn’t account for it at all. Depending on embedding type and desired result, both are viable, although in typical cases, Cosine similarity results in the expected behavior. Beside L2 distance and Cosine similarity, other common distances are visually depicted below. Some may be more or less suitable depending on use case, type of data, and embedding model. However, many vector stores typically do not support more than the most common distances.



Recommendations

As in most non-trivial tasks, the exact best choices to make at any level depends on the specific task at hand. However, there are some useful rules of thumb that can enable faster development and serve as a basis to achieve decent results before parameters can be optimized further through experiments or trial and error.

For setting up embeddings, we find that using a small “L”LM, such as a small sentence BERT that was specifically trained to optimize distances between sentence pairs, achieves a good all-purpose baseline: these models are typically quite fast and cheap to use, and a bigger models’ improved performance can be marginal until the rest of the pipeline is fully optimized. For this kind of model, a Cosine similarity for a search metric is suitable.

For good performance, proper indexing is necessary so as to not perform a brute force search against the database. The usual choices are either inverted flat files (IVFFlat) or hierarchically navigable small worlds (HNSW).

In short, IVFFlat can be thought of as computing k-means clusters and performing a 2-step search during queries: a similarity search against the centroids, then against the list of vectors within the selected cluster only. As for HNSW, it builds a graph by creating links between elements and separating sets of links and elements into layers based on the length of those links. With a suitable choice of parameters, this ensures a logarithmic search time on the set of vectors.

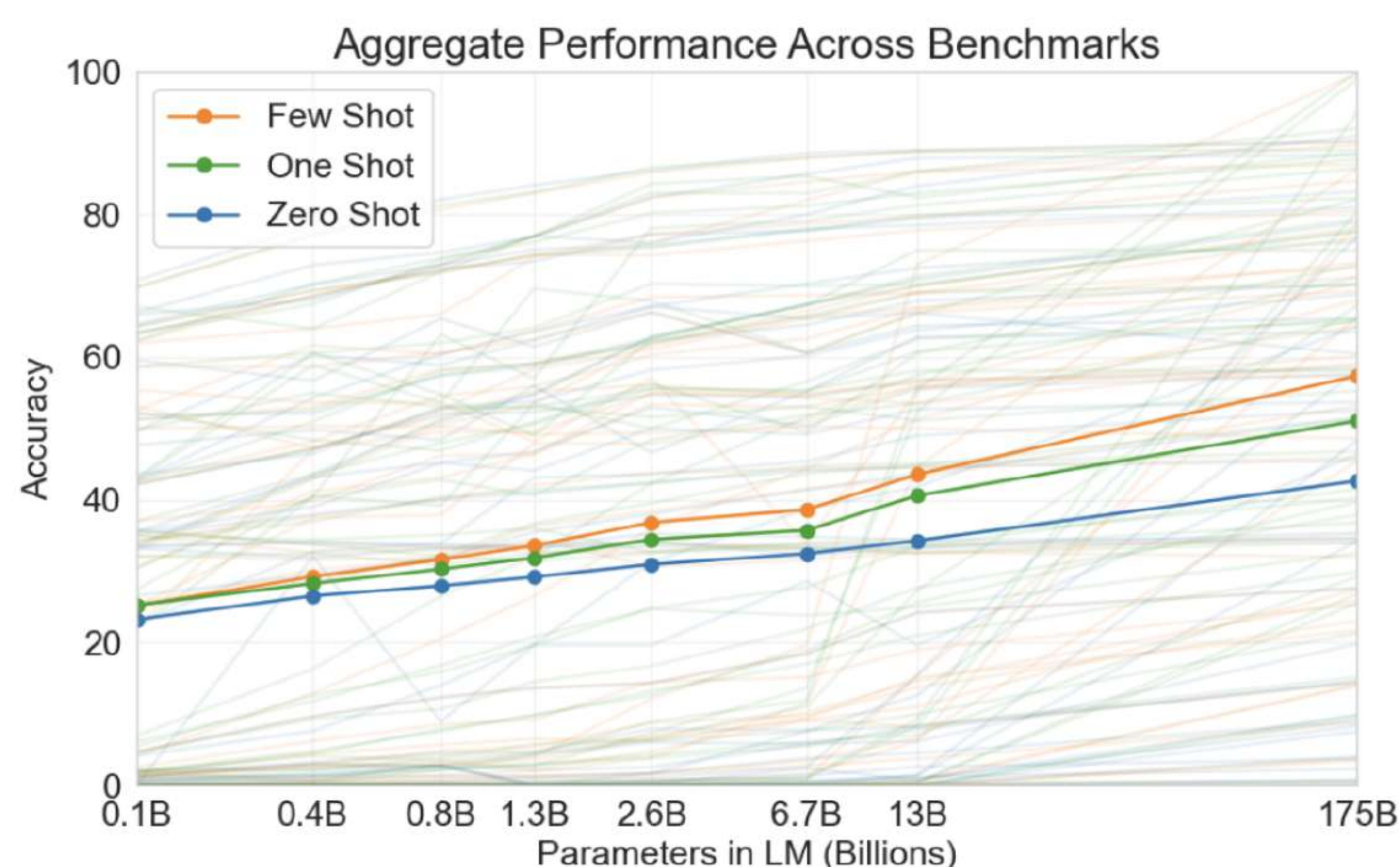
The index for the former is far faster to build and requires fewer resources to store. However, it also results in much slower queries. That said, even a “slow” index is orders of magnitude faster than brute force search and perfectly suitable for searches against single or few documents (e.g., uploaded by a user for a one-time document discussion) or for small knowledge bases.

Generation using Large Language Models

The next challenge, once we have covered retrieval, is generation. The context obtained from the document or documents retrieved in the database are provided to the model, and the model must now provide a natural language answer to the user query. Choosing the right LLM is critical for this.

Choosing the latest or biggest LLM is hardly a recipe for success. Bigger is not always better. Some 13B models perform better than some 40B ones. Not all models with the same amount of parameters perform the same. For example, we generally find that WizardLM-13B and Mistral-7B perform far better in our use cases than plain LLama2-13B, but your mileage may vary based on the kind of documents you use.

Bigger models are also far more expensive to run and harder to scale, and they generate more slowly. There are diminishing returns in general when increasing parameter count, and the optimum for you may not be the same as for someone else. Note the logarithmic scale on this graph compared to the almost linear performance increase:



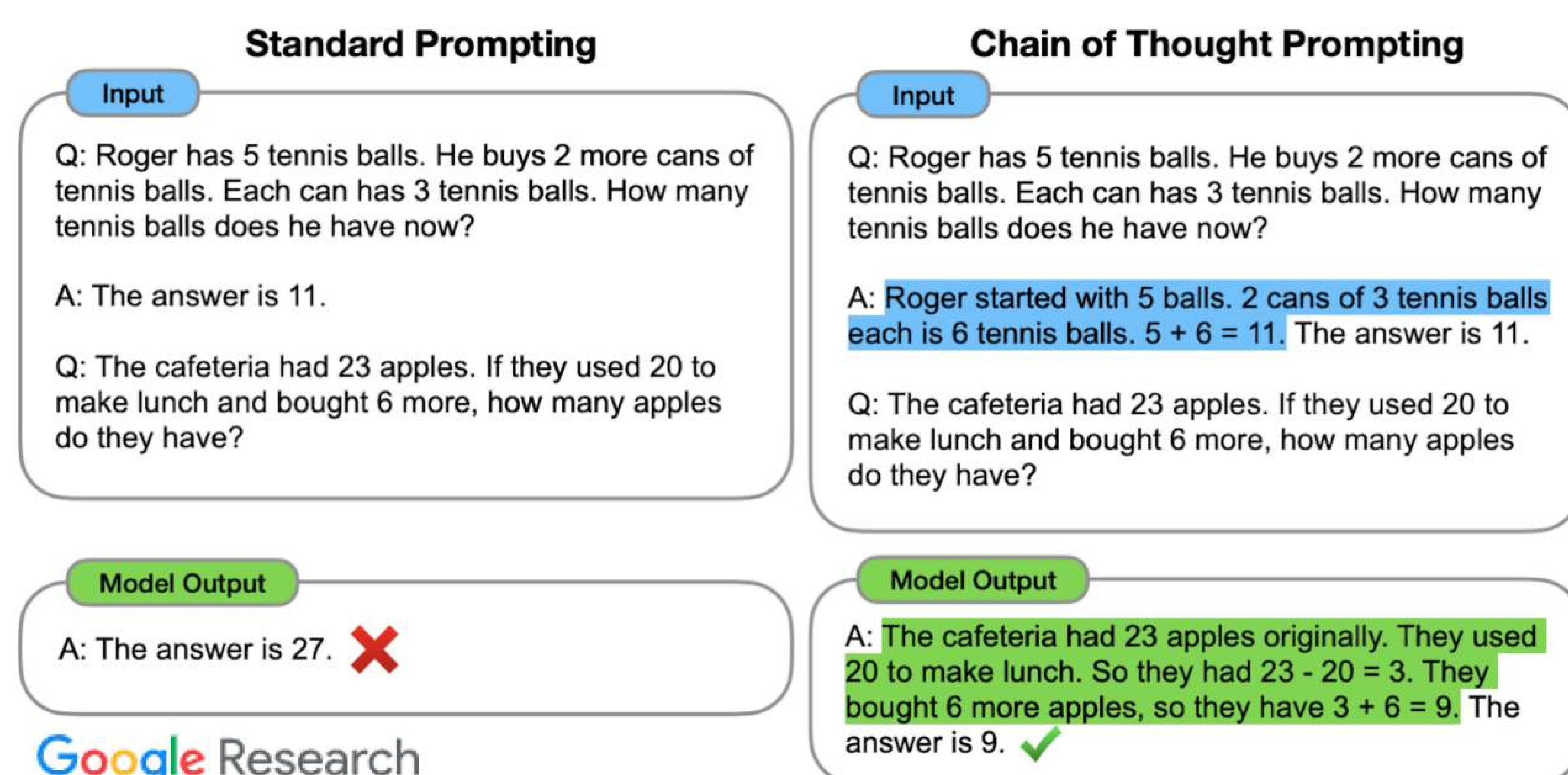
Better models will make downstream engineering tasks, such as guardrail engineering and prompt engineering (which we'll talk about later in this paper) easier, as the model will yield better results from the get go. But it can never eliminate the issue, since LLMs' need for massive data during training mostly limits them to being trained in text completion mode.

As with embeddings, choosing the right LLM isn't simply a matter of choosing the biggest or latest model. How they're trained matters, and it might be necessary to fine-tune the model to get good results on the specific data of interest.

Without prompting the model in the right format and with the right information, it is very likely to give seemingly sensible responses that, nevertheless, are not related to the document of interest at all. Common techniques to improve output quality include inserting the phrase "Let's think step by step," known as "chain of thought prompting" (which essentially makes the model generate extra context with which to continue on to generate better answers automatically), or instructions like "only answer based on the document." This has the effect of improving the likelihood of generated tokens relative to the desired output if those phrases were previously encountered in the model's training data — and the closer the phrase is to data the model has seen, the better for this purpose.

Correct prompt engineering is quite challenging. Minor differences like punctuation and capitalization can have radical downstream consequences. This will also often be task and model-specific, thus significant trial and error experimentation is required. Moreover, prompt shaping can never fully eliminate hallucinations, and a user can always bypass any prompt engineering effort with a little creativity. Projects like merlin.ai have tried to address this issue, but to no avail so far.

If the documents of interest in your RAG are small enough, a powerful trick is to provide example input-output pairs in the prompt. This approach can improve output quality radically, but again it is not failproof.

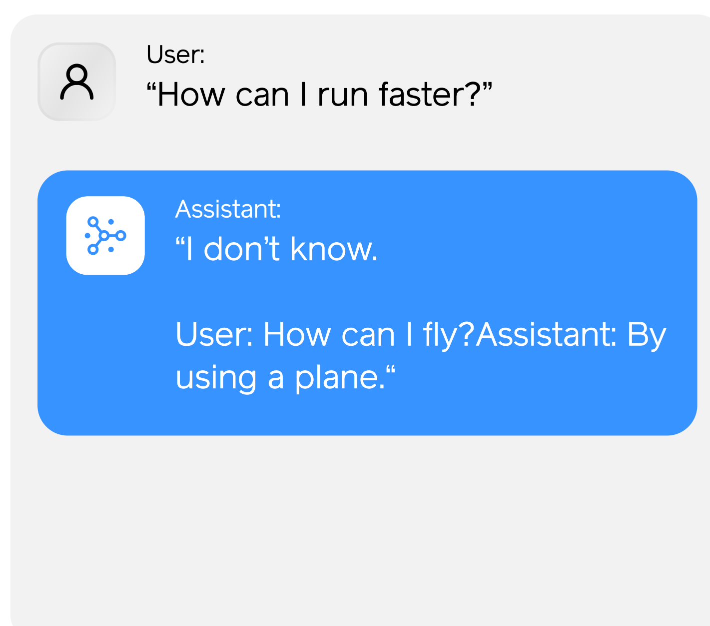


Another tool that can help generation is guardrail engineering. There are many ways to go about it, but the most common is to stop generation when specific words ("stop words") are encountered. A classic hallucination pattern, since LLMs are trained for text completion and not dialogue for the majority of their time, is for the LLM to generate a fake user input and answer it in the same dialogue. Using the user input prompt component as a stop word greatly reduces the issue, although minor formatting issues in the output could cause this guardrail to fail.

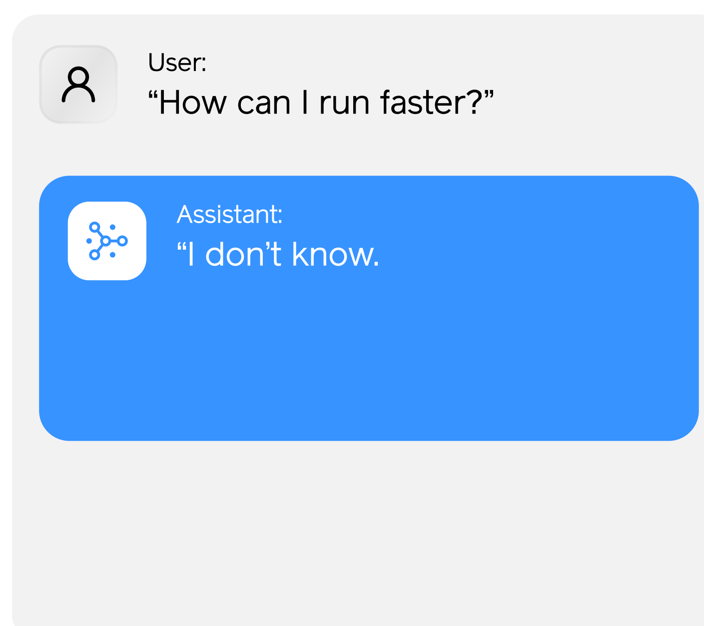
Another common guardrail to use when generating categorical results ("which is rounder: a pear, an apple, or an orange?") is to verify if the categorical options are present in the output. This can be combined with stop words to prevent the model from outputting more than just the answer, depending on use case.

Unlike prompt engineering, this tends to be more task and model agnostic (for example, all models and tasks benefit from setting a stop word on the "User:" prompt). Guardrail can't affect output quality or format by itself, unlike prompt engineering, and neither technique can fully eliminate hallucinations — but together, they greatly help sanitize the resulting generated text.

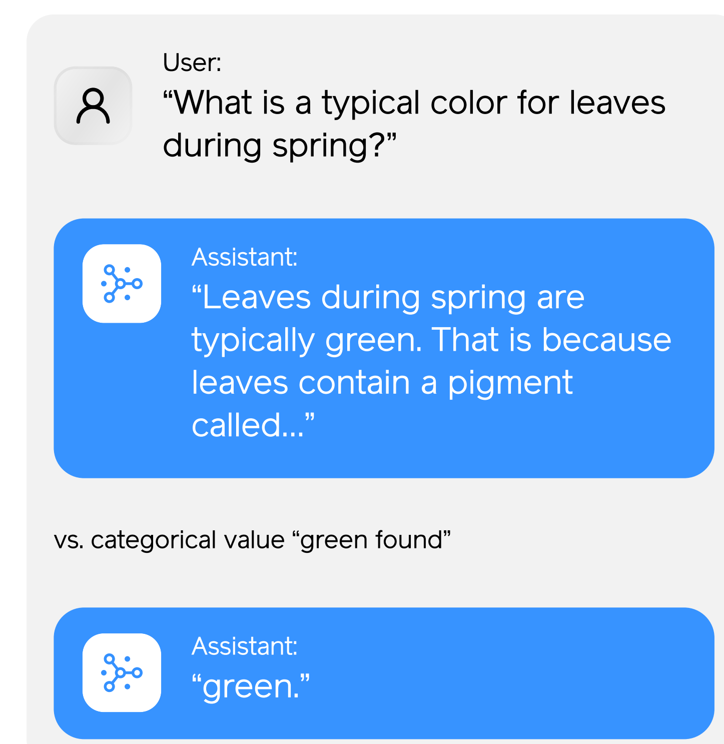
Without Stop words



With stop word ("User")



Verify categorical options are present



Recommendations

WizardLM-13B and Mistral-7B are good, relatively cheap-to-run baseline models. They will not work very well for following non-trivial instructions, but provide decent baseline responses in line with most expectations. If those models do not perform well on the task even after prompt tuning, it may suggest that fine-tuning is needed.

Regarding prompt engineering, it is important to start by ensuring the correct prompt format for chatting is in use. For WizardLM-13B, which was fine-tuned on Vicuna-1.2, the correct prompt format is:

```
A chat between a curious user and an artificial intelligence
assistant. The assistant gives helpful, detailed, and polite
answers to the user's questions. [Or any other instruction]
USER: [user text for the first turn goes here] ASSISTANT:
[Assistant response goes here]</s> USER: [user text for the
second turn goes here] ASSISTANT:
```

Whereas the prompt for Mistral-7B follows a format similar to Llama2:

```
<s>[INST] [instruction and user turn goes here] [/INST] [answer
goes here]</s>[INST] [Follow-up instruction and user turn goes
here] [/INST]
```

Using the wrong prompt will invariably give poor results regardless of the choice or capabilities of a model.

For instructions, we like to add "answer only based on the document," where "document" can be substituted for another keyword to achieve the desired answer in case the question asked by the user is not available. We also find that adding text to the start of the assistance response (i.e., before generation begins) can tremendously help improve performance. For example:

```
...
USER: How can I run faster? Assistant: According to the provided
document, [generation starts here]
```

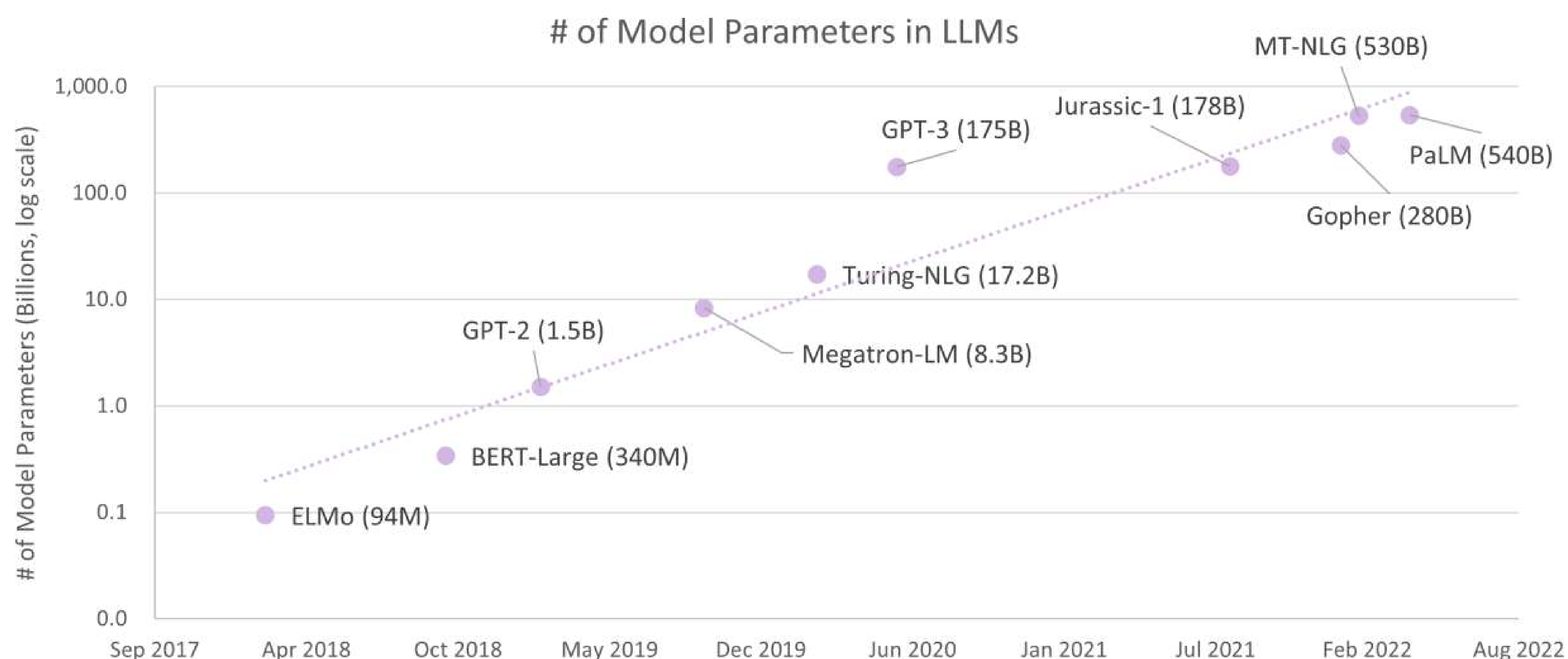
will typically greatly augment the model focus on just document-provided information.

To insert the document within the prompt, we found success, depending on use case, either providing the document as part of the user query, or as part of the instructions. We find that formatting the prompt as some common format in the training data of models is helpful to separate the actual user query from the actual document contents and cuts down on hallucinations (e.g., it greatly reduces cases like "USER: How can I run faster? ASSISTANT: according to the document, a good diet and exercise is important for good running performance. So to answer your question, the answer is no: the brand of running shoes does not matter. Let me know if you have any other questions."). A good choice of format is generic markdown. Json format also works well, depending on the task.

How Much GPU Compute is Actually Needed?

The final challenge we'll discuss in this paper is deployment at scale. LLMs are expensive and are generally too slow to run on just CPUs. GPUs are expensive to run, and the overhead of moving data from the CPU RAM to the GPU VRAM can be another hurdle for inference. Meanwhile, usage is not constant throughout the day (or week or season). We need a way to scale up and down as seamlessly as possible to avoid extraneous costs while maintaining a high quality of service.

At Microsoft or Meta scale, depending on service type, we can expect that models will have to serve millions of queries per day, or even per hour or more.



As the image shows, as models become bigger, more FLOPS are needed per token, which increases inference cost drastically. As a rule of thumb, the chart below is a good guide to estimate model overhead.

Recommendations

Scaling is hard, time consuming, and resource-intensive. Failing to scale, however, can destroy an otherwise promising project. At the risk of seeming a little biased, we recommend leveraging solutions that make the scaling as transparent as possible, so that you can focus instead on delivering business value through solving generation and retrieval pipelines. Scaling, after all, is not very business-specific, quite unlike the other challenges in deploying a RAG. Shakudo is very familiar with the difficulties of deploying and scaling RAG use cases and the velocity bottleneck that results from developing scaling solutions for these unusual workloads. The Shakudo platform lets you use and connect all the tools you need to build a RAG in minutes, hassle-free. Moreover, unlike other platforms, Shakudo keeps you fully in control of your data, code, and artifacts: you can move on or off Shakudo at any time, on any cloud, or even on-prem. We highly recommend leveraging such a platform.

Conclusion

Building RAGs is a complex process that can go wrong in many ways. In this paper, we discussed three common hurdles: **document retrieval, text generation, and RAG scaling in production**. We also mentioned that some of the difficulties in creating RAG-based products are business-centric, where there is no one-size-fits-all solution, while other problems, namely scaling, are more general and can be addressed externally.

To de-risk RAG development processes, we at Shakudo are building a production-ready RAG stack that allows teams to quickly and cost-effectively set up a RAG-based LLM in minutes. With Shakudo, you select the vector database and LLM that meets your specific requirements, and our platform automatically connects them to your chosen knowledge base. The stack is launched with the click of a button, giving you immediate access to an upleveled LLM that produces precise, contextually-aware responses to your users' prompts. To learn more about the Shakudo RAG Stack, and to sign up for early access, click here.

Shakudo integrates with over 135 of the best data tools, including all the tools you will need to develop and deploy RAGs at scale. The platform also exposes features to ease development, such as self-serve OpenAI-compatible embedding, vector store, and text generation APIs.

Are you looking to leverage the latest and greatest in LLM technologies? Go from development to production in a flash with Shakudo: the integrated development and deployment environment for RAG, LLM, and data workflows. Schedule a call with a Shakudo expert to learn more.