Applied AI Software Engineering: RAG

Retrieval-Augmented Generation (RAG) is a common building block of AI software engineering. A deep dive into what it is, its limitations, and some alternative use cases. By Ross McNairn.

GERGELY OROSZ AND ROSS MCNAIRN

MAY 14, 2024 · PAID

131 5 5 Share ···

👋 *Hi, this is Gergely with a subscriber-only issue of the Pragmatic Engineer Newsletter. In every issue, I cover challenges at Big Tech and startups through the lens of engineering managers and senior engineers. To get articles like this in your inbox, every week, subscribe:*

I recently spoke with Karthik Hariharan, who heads up engineering at VC firm Goodwater Capital, and he highlighted a trend he'd spotted:

> "There's an engineering project I'm seeing almost every startup building a Large Language Model (LLM) put in place: building their own Retrieval Augmentation Generation (RAG) pipelines.
>
> RAGs are a common pattern for anyone building an LLM application. This is because it provides a layer of 'clean prompts' and fine-tuning. There are some existing open-source solutions, but almost everyone just builds their own, anyway."

I asked a few Artificial Intelligence (AI) startups about this, and sure enough, all do build their own RAG. So, I reached out to a startup I know is doing the same: Wordsmith AI. It's an AI startup for in-house legal teams that's making heavy use of RAG, and was co-founded by Ross McNairn. He and I worked for years together at Skyscanner and he offered to share Wordsmith AI's approach for building RAG pipelines, and some learnings. *Declaration of interest: I'm an investor in Wordsmith, and the company has recently launched out of stealth.*

Today, we cover:

1. Providing an LLM with additional context
2. The simplest RAGs
3. What is a RAG pipeline?
4. Preparing the RAG pipeline data store
5. Bringing it all together
6. RAG limitations
7. Real-world learnings building RAG pipelines

Today's article includes a "code-along," so you can build your own RAG. View the code used in this article **at this GitHub repository: hello-wordsmith**. To keep up with Ross, subscribe to his blog or follow him on LinkedIn.

With that, it's over to Ross:

Introduction

Hi there! This post is designed to help you get familiar with one of the most fundamental patterns of AI software engineering: RAG, aka Retrieval Augmented Generation.

I co-founded a legal tech startup called Wordsmith, where we are building a platform for running a modern in-house legal team. Our founding team previously worked at Meta, Skyscanner, Travelperk and KPMG.

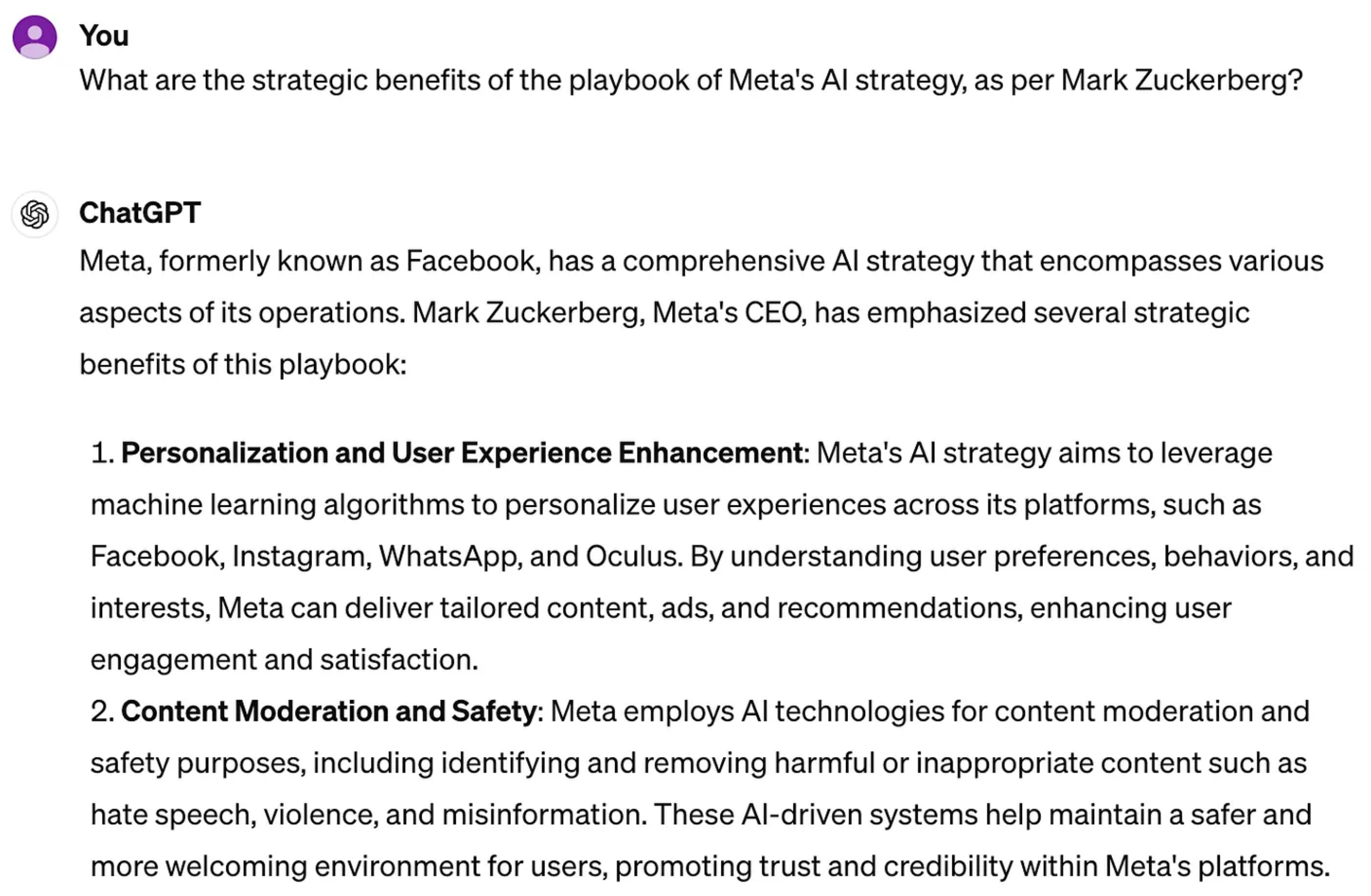
We are working in a targeted domain – legal texts – and building AI agents to give in-house legal teams a suite of AI tools to remove bottlenecks and improve how they work with the rest of the business. Performance and accuracy are key characteristics for us, so we've invested a lot of time and effort in how to best enrich and "turbo charge" these agents with custom data and objectives.

We ended up building our RAG pipeline, and I will now walk you through how we did it and why. We'll go into our learnings, and how we benchmark our solution. I hope that the lessons we learned are useful for all budding AI engineers.

1. Providing an LLM with additional context

Have you ever asked ChatGPT a question it does not know how to answer, or its answer is too high level? We've all been there, and all too often, interacting with a GPT feels like talking to someone who speaks really well, but doesn't know the facts. Even worse, they can make up the information in their responses!

Here is one example. On 1 February 2024, during an earnings call, Mark Zuckerberg laid out the strategic benefits of Meta's AI strategy. But when we ask ChatGPT a question about this topic, this model will make up an answer that is high-level, but is not really what we want:



You
What are the strategic benefits of the playbook of Meta's AI strategy, as per Mark Zuckerberg?

ChatGPT
Meta, formerly known as Facebook, has a comprehensive AI strategy that encompasses various aspects of its operations. Mark Zuckerberg, Meta's CEO, has emphasized several strategic benefits of this playbook:

1. **Personalization and User Experience Enhancement**: Meta's AI strategy aims to leverage machine learning algorithms to personalize user experiences across its platforms, such as Facebook, Instagram, WhatsApp, and Oculus. By understanding user preferences, behaviors, and interests, Meta can deliver tailored content, ads, and recommendations, enhancing user engagement and satisfaction.
2. **Content Moderation and Safety**: Meta employs AI technologies for content moderation and safety purposes, including identifying and removing harmful or inappropriate content such as hate speech, violence, and misinformation. These AI-driven systems help maintain a safer and more welcoming environment for users, promoting trust and credibility within Meta's platforms.

ChatGPT 3.5's answer to a question about Meta's AI strategy. The answer is generalized, and misses a critical source which answers the question

This makes sense, as the model's training cutoff date was before Mark Zuckerberg made the comments. If the model had access to that information, it would have likely been able to summarize the facts of that meeting, which are:

> “So I thought it might be useful to lay out the strategic benefits [of Meta’s open source strategy) here. (...)
>
> The short version is that open sourcing improves our models. (...)
>
> First, open-source software is typically safer and more secure as well as more compute-efficient to operate due to all the ongoing feedback, scrutiny and development from the community. (...)
>
> Second, open-source software often becomes an industry standard. (...)
>
> Third, open source is hugely popular with developers and researchers. (...)
>
> The next part of our playbook is just taking a long-term approach towards the development.”

LLMs’ understanding of the world is limited to the data they’re trained on. If you’ve been using ChatGPT for some time, you might remember this constraint in the earlier version of ChatGPT, when the bot responded: “I have no knowledge after April 2021,” in several cases.

Providing an LLM with additional information

There is a bunch of additional information you want an LLM to use. In the above example, I might have the transcripts of all of Meta’s shareholders meetings that I want the LLM to use. But how can we provide this additional information to an existing model?

Option 1: input via a prompt

The most obvious solution is to input the additional information via a prompt; for example, by prompting “Using the following information: [input a bunch of data] please answer the question of [ask your question].”

This is a pretty good approach. The biggest problem is that this may not scale because of these reasons:

- **The input tokens limit**. Every model has an input prompt token limit. At the time of publication this is 4.069 tokens for GPT-3, 16,385 for GPT-3.5, 8,192 for GPT-4, 128,000 for GPT-4 Turbo, 200.000 for Anthropic models. Google's Gemini model allows for an impressive one million token limit. While a million-token limit greatly increases the possibilities, it might still be too low for use cases with a lot of additional text to input.
- **Performance.** The performance of LLMs substantially decreases with longer input prompts; in particular, you get degradation of context in the middle of your prompt. Even when creating long input prompts is a possibility, the performance tradeoff might make it impractical.

Option 2: fine-tune the model

We know LLMs are based on a massive weights matrix. *Read more on how ChatGPT works in this Pragmatic Engineer issue. All LLMs use the same principles.*

An option is to update these weight matrices based on additional information we'd like our model to know. This can be a good option, but it is a much higher upfront cost in terms of time, money, and computing resources. Also, it can only be done with access to the model's weightings, which is not the case when you use models like ChatGPT, Anthropic, and other "closed source" models.

Option 3: RAG

The term 'RAG' originated in a 2020 paper led by Patrick Lewis. One thing many people notice is that "Retrieval Augmented Generation" sounds a bit ungrammatical. Patrick agrees, and has said this:

> "We always planned to have a nicer-sounding name, but when it came time to write the paper, no one had a better idea."

RAG is a collection of techniques which help to modify a LLM, so it can fill in the gaps and speak with authority, and some RAG implementations even let you cite sources.

The biggest benefits of the RAG approach:

Give a LLM domain-specific knowledge You can pick what data you want your LLM to draw from, and even turn it into a specialist on any topic there is data about.

This flexibility means you can also extend your LLMs' awareness far beyond the model's training cutoff dates, and even expose it to near-real time data, if available.

Optimal cost and speed. For all but a handful of companies, it's impractical to even consider training their own foundational model as a way to personalize the output of an LLM, due to the very high cost and skill thresholds.

In contrast, deploying a RAG pipeline will get you up-and-running relatively quickly for minimal cost. The tooling available means a single developer can have something very basic functional in a few hours.

Reduce hallucinations. "Hallucination" is the term for when LLMs "make up" responses. A well-designed RAG pipeline that presents relevant data will all but eliminate this frustrating side effect, and your LLM will speak with much greater authority and relevance on the domain about which you have provided data.

For example, in the legal sector it's often necessary to ensure an LLM draws its insight from a specific jurisdiction. Take the example of asking a model a seemingly simple question, like:

How do I hire someone?

Your LLM will offer context based on the training data. However, you do *not* want the model to extract hiring practices from a US state like California, and combine this with British visa requirements!

With RAG, you control the underlying data source, meaning you can scope the LLM to only have access to a single jurisdiction's data, which ensures responses are consistent.

Better transparency and observability. Tracing inputs and answers through LLMs is very hard. The LLM can often feel like a “black box,” where you have no idea where some answers come from. With RAG, you see the additional source information injected, and debug your responses.

2. The simplest RAGs

The best way to understand new technology is often just to play with it. Getting a basic implementation up and running is relatively simple, and can be done with just a few lines of code. To help, Wordsmith has created a wrapper around the LlamaIndex open source project to help abstract away some complexity. You can get up and running, easily. It has a README file in place that will get you set up with a local RAG pipeline on your machine, and which chunks and embeds a copy of the US Constitution, and lets you search away with your command line.

This is as simple as RAGs get; you can “swap out” the additional context provided in this example by simply changing the source text documents!

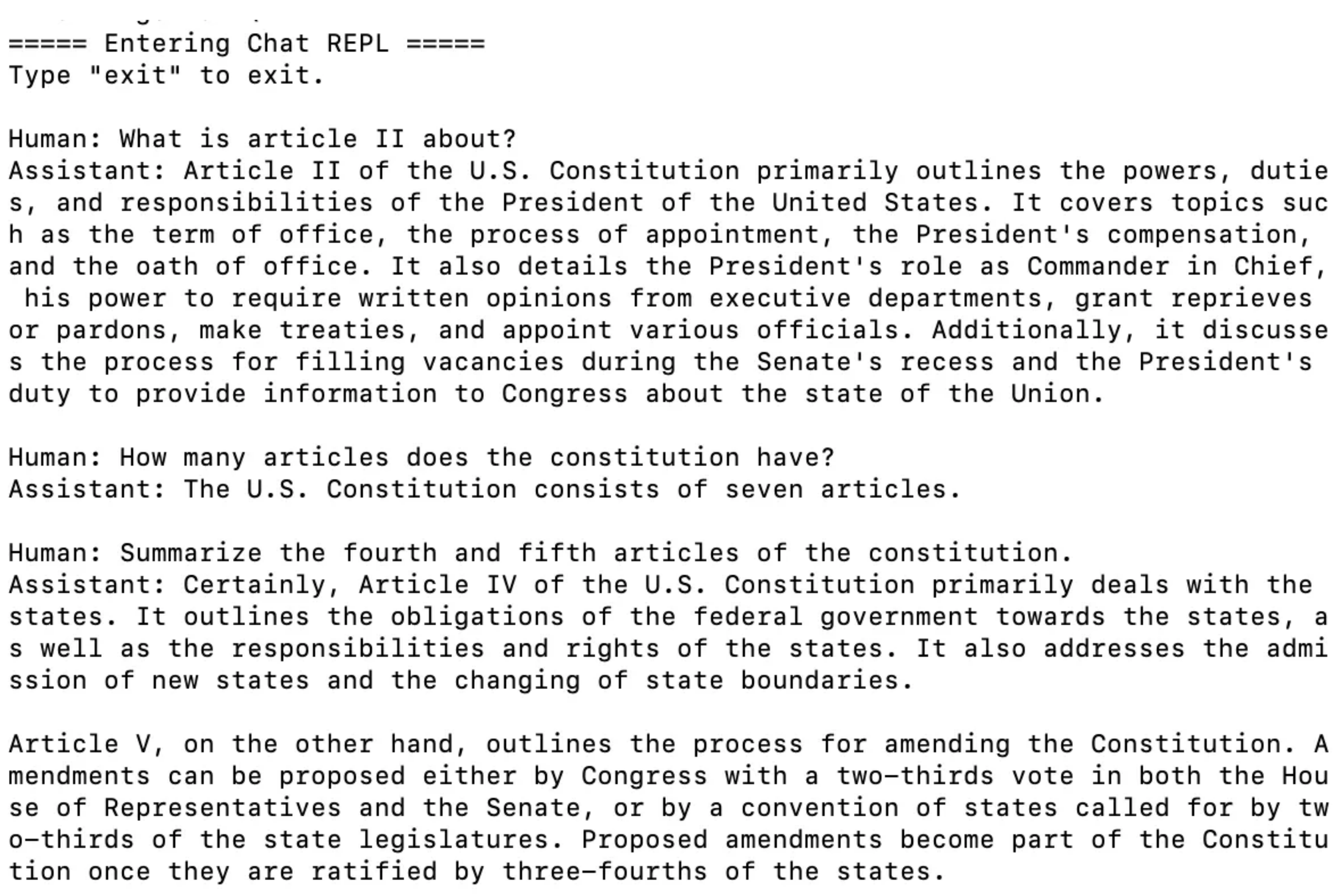
This article is designed as a code-along, so I'm going to link you to sections of this repo, so you can see where specific concepts manifest in code.

To follow along with the example, the following is needed:

- An active OpenAI subscription with API usage. Set one up here if needed. *Note: running a query will cost in the realm of $0.25-$0.50 per run.*
- Follow the instructions to set up a virtual Python environment, configure your OpenAI key, and start the virtual assistant.

This example will load the text of the US constitution from this text file, as a RAG input. However, the application can be extended to load your own data from a text file, and to “chat” with this data.

Here’s an example of how the application works when set up, and when the OpenAI API key is configured:



```
===== Entering Chat REPL =====
Type "exit" to exit.

Human: What is article II about?
Assistant: Article II of the U.S. Constitution primarily outlines the powers, dutie
s, and responsibilities of the President of the United States. It covers topics suc
h as the term of office, the process of appointment, the President's compensation,
and the oath of office. It also details the President's role as Commander in Chief,
 his power to require written opinions from executive departments, grant reprieves
or pardons, make treaties, and appoint various officials. Additionally, it discusse
s the process for filling vacancies during the Senate's recess and the President's
duty to provide information to Congress about the state of the Union.

Human: How many articles does the constitution have?
Assistant: The U.S. Constitution consists of seven articles.

Human: Summarize the fourth and fifth articles of the constitution.
Assistant: Certainly, Article IV of the U.S. Constitution primarily deals with the
states. It outlines the obligations of the federal government towards the states, a
s well as the responsibilities and rights of the states. It also addresses the admi
ssion of new states and the changing of state boundaries.

Article V, on the other hand, outlines the process for amending the Constitution. A
mendments can be proposed either by Congress with a two-thirds vote in both the Hou
se of Representatives and the Senate, or by a convention of states called for by tw
o-thirds of the state legislatures. Proposed amendments become part of the Constitu
tion once they are ratified by three-fourths of the states.
```

The example RAG pipeline application answering questions using the US Constitution supplied as additional context

If you've followed along and have run this application: congratulations! You have just executed a RAG pipeline. Now, let's get into explaining how it works.

3. What is a RAG pipeline?

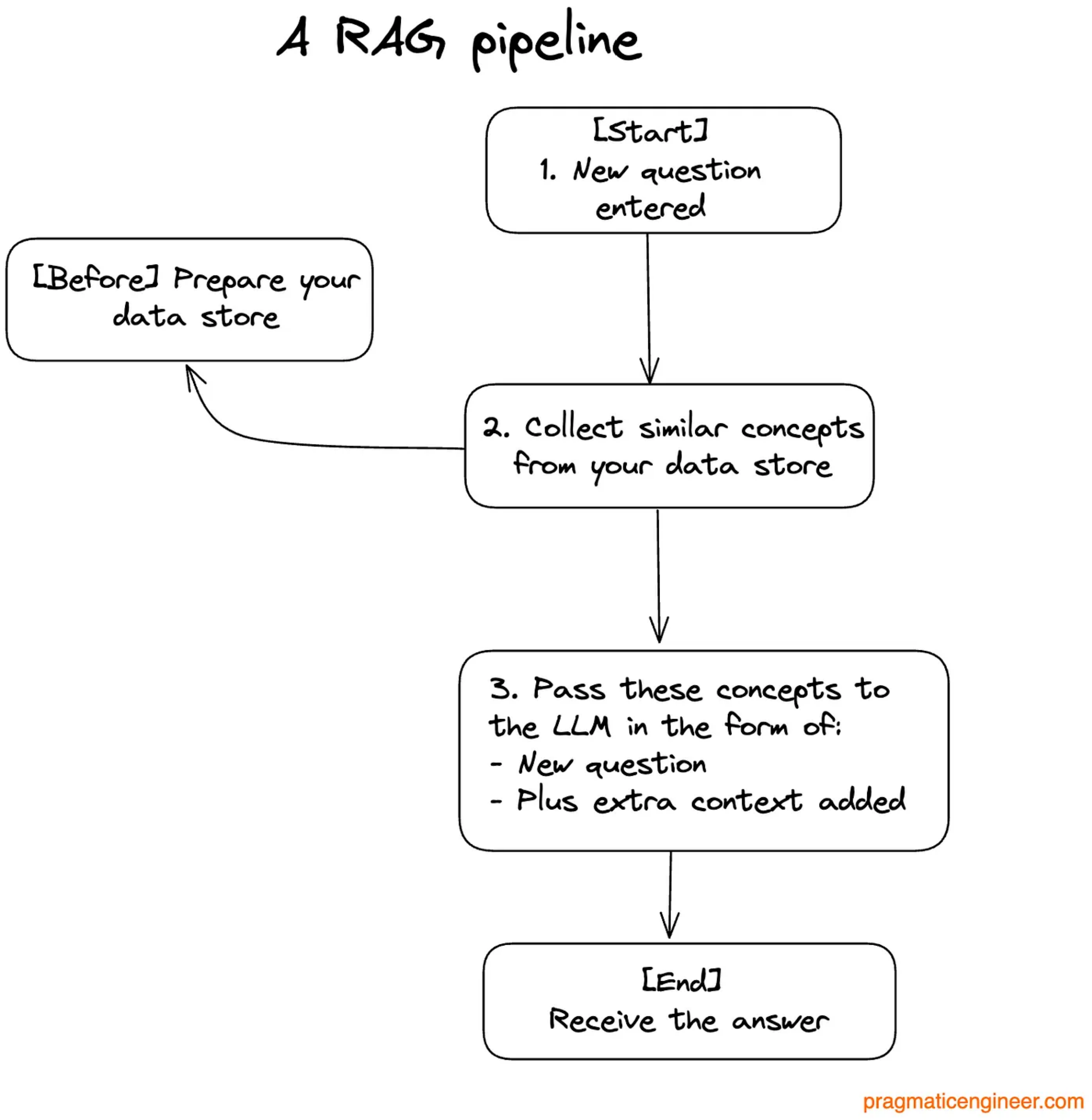
A RAG pipeline is a collection of technologies needed to enable the capability of answering using provided context. In our example, this context is the US Constitution and our LLM model is enriched with additional data extracted from the US Constitution document.

Here are the steps to building a RAG pipeline:

Step 1: Take an inbound query and deconstruct it into relevant concepts
Step 2: Collect similar concepts from your data store

Step 3: Recombine these concepts with your original query to build a more relevant, authoritative answer.

Weaving this together:

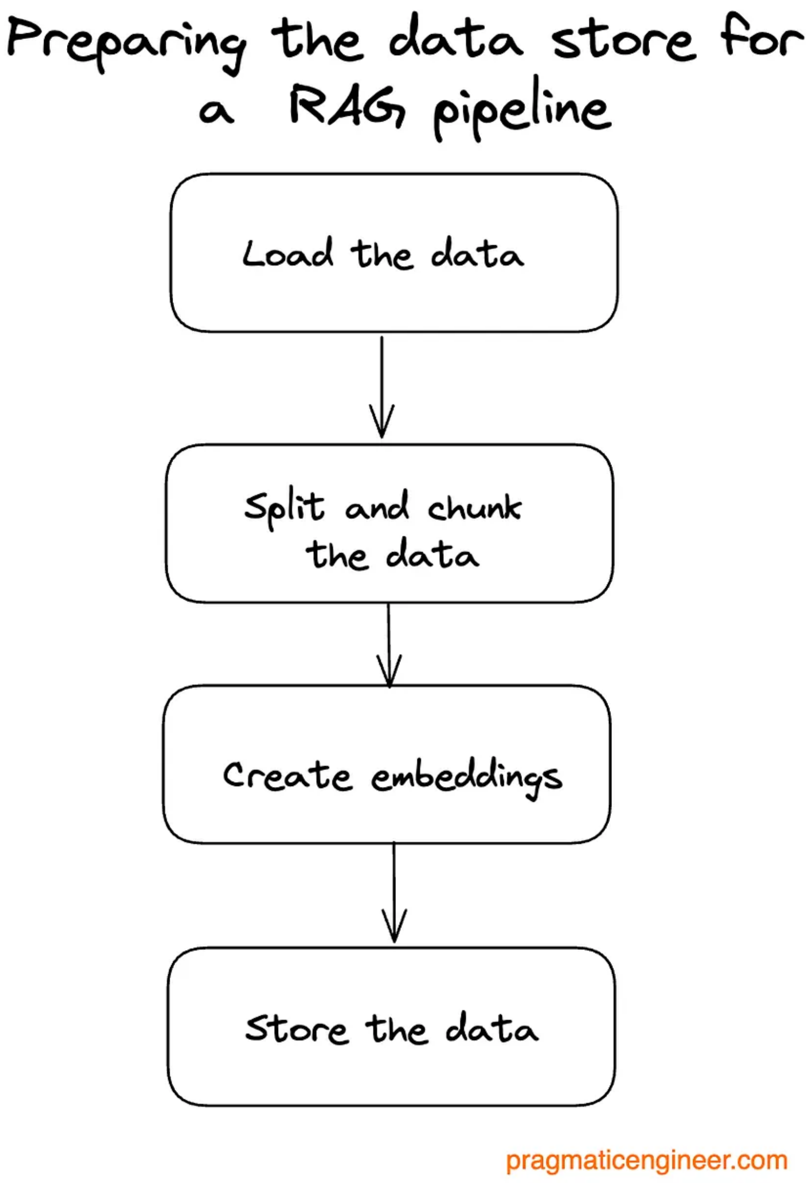

A RAG pipeline
[Start]
1. New question entered
[Before] Prepare your data store
2. Collect similar concepts from your data store
3. Pass these concepts to the LLM in the form of:
- New question
- Plus extra context added
[End]
Receive the answer
pragmaticengineer.com

A RAG pipeline at work. It extends the context an LLM has access to, by fetching similar concepts from the data store to answer a question

While this process appears simple, there is quite a bit of nuance in how to approach each step. A number of decisions are required to tailor to your use case, starting with how to prepare the data for use in your pipeline.

4. Preparing the RAG pipeline data store

To start, we need to identify the data we will use to enrich responses with. We then load it into a vector data store, which is what we use during the search phase.



Steps in preparing a RAG pipeline

Load the data

Before we can harness the data, we need to convert it into a format that lets us perform the manipulations we will need, such as creating embeddings (basically, vectors.)

In our example RAG pipeline, we use the LlamaIndex, a comprehensive data framework to bridge storing data, and make this data accessible to LLMs. *Another popular option for an LLM data framework is Langchain.*

Here's how we prepare our data in the code:

```
        package_directory = os.path.dirname(os.path.abspath(__file__))
        dataset_path = os.path.join(package_directory, "public_wordsmith_dataset")
        docs = SimpleDirectoryReader(
            input_dir=dataset_path, filename_as_id=True
        ).load_data()
```

Loading all files in the "public_wordsmith_dataset" directory. For now, this is one file: the us_constitution.txt. You can add any file to this directory to include in the RAG pipeline.

Cleaning up data before storing it is a common-enough step with real world RAG applications, but we've omitted it from our simple use case. For example, if your application uses web page data as HTML files in the RAG pipeline, then you'll need to add preprocessing to remove HTML tags and anything else that is irrelevant for text processing.

There is an ever-growing list of services and tools that assist with the cleaning of data. FireCrawl is a good choice for working with web pages. In general, it helps get from "raw" data to "cleaned" data. There are many similar tools which clean data for AI use cases; it's a vibrant and fast-evolving part of the AI ecosystem.

Split and chunk the data

Once we've loaded and cleaned the data, we want to split our document into 'chunks.' These are the parts we want to retrieve and pass on as context to our LLM.

With RAG pipelines, it's common enough to work with long documents, such as wiki or Confluence pages, contracts, and other lengthy documentation. So, why not just feed the whole document into the LLM? Why "chunk it up?" The reason is that feeding a long document into an LLM can cause these issues:

- **Degraded performance.** LLMs like ChatGPT have self-attention (every token being aware of every other token) scale quadratically. This means that when predicting the 100th token, around 10,000 operations are needed. But to predict the 1,000th token, circa 1 million operations need to be done. This means that the longer the input, the worse the output performance of the LLM.
- **Less useful output.** We've observed that inputting a long document results in the LLM receiving a lot of irrelevant data, and responses can become confusing.
- **Increased cost.** The longer the input, the higher the cost of operating the model. This cost will be crystal clear when using an API like OpenAI, which bills you. If running your own infrastructure, you'll observe higher required compute resource usage, which translates to higher compute cost.

Therefore, deconstructing a long document into "small enough" pieces is a smart move. With these pieces sized correctly, we can pass in the relevant pieces, and make the LLMs' answers more specific, accurate, and faster, at a lower cost!

Deciding how to chunk your data is a major decision with RAG pipelines. There are many options to choose from when chunking data, but all choices have their own set of tradeoffs. Here are some examples:

- The simplest approach: break the text into 250-500 character chunks.
- A slightly more advanced approach: split the text by paragraph.
- Even more advanced option: divide chunks by "concept" and do some preprocessing to make breakpoints between chunks logical.

Chunking strategies is an area you can get very deep into. There's a variety of strategies to use, which the article Chunking strategies for LLM applications by Roie Schwaber-Cohen, goes through:

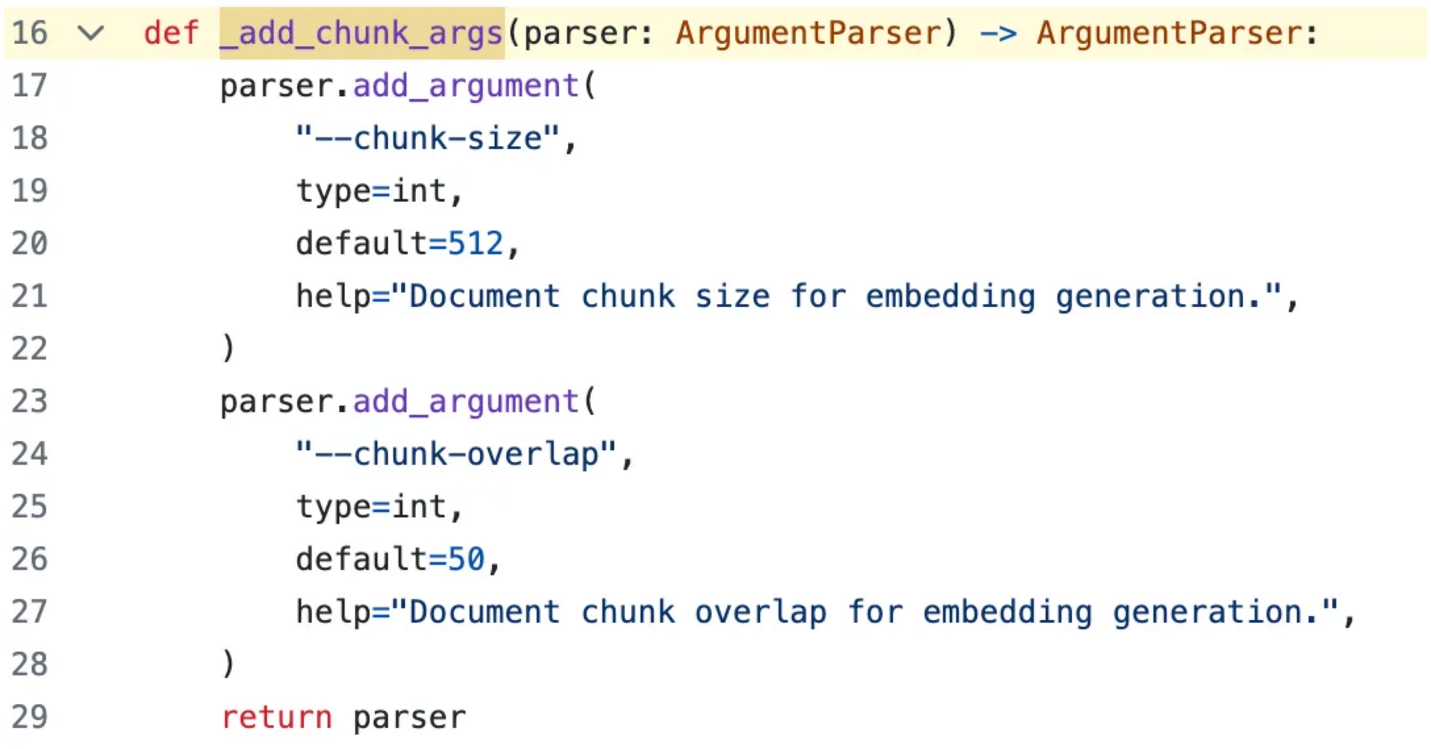
- **Fixed-size chunking:** split by the number of tokens
- **"Content-aware" chunking:** chunking by sentence.
- **Recursive chunking**: divide the input text into smaller chunks in a hierarchical, iterative manner using a set of separators

- **Specialized chunking**: when working with structured and formatted content like Markdown or LaTeX
- **Semantic chunking**: attempting to take the meaning of segments within the document.

In general, smaller chunks tend to help create smaller, more relevant concepts when you retrieve them. At the same time, they lead to very narrow responses because small chunks can become disconnected from related chunks.

Chunking is more an art than a science. My advice is to spend plenty of time iterating your chunking strategy! Get it right for *your* use case and the source data you have to work with.

In our code, chunking happens here:



```python
def _add_chunk_args(parser: ArgumentParser) -> ArgumentParser:
    parser.add_argument(
        "--chunk-size",
        type=int,
        default=512,
        help="Document chunk size for embedding generation.",
    )
    parser.add_argument(
        "--chunk-overlap",
        type=int,
        default=50,
        help="Document chunk overlap for embedding generation.",
    )
    return parser
```

The code that does the chunking. We use a fixed-size chunking strategy, breaking our document down every 512 characters

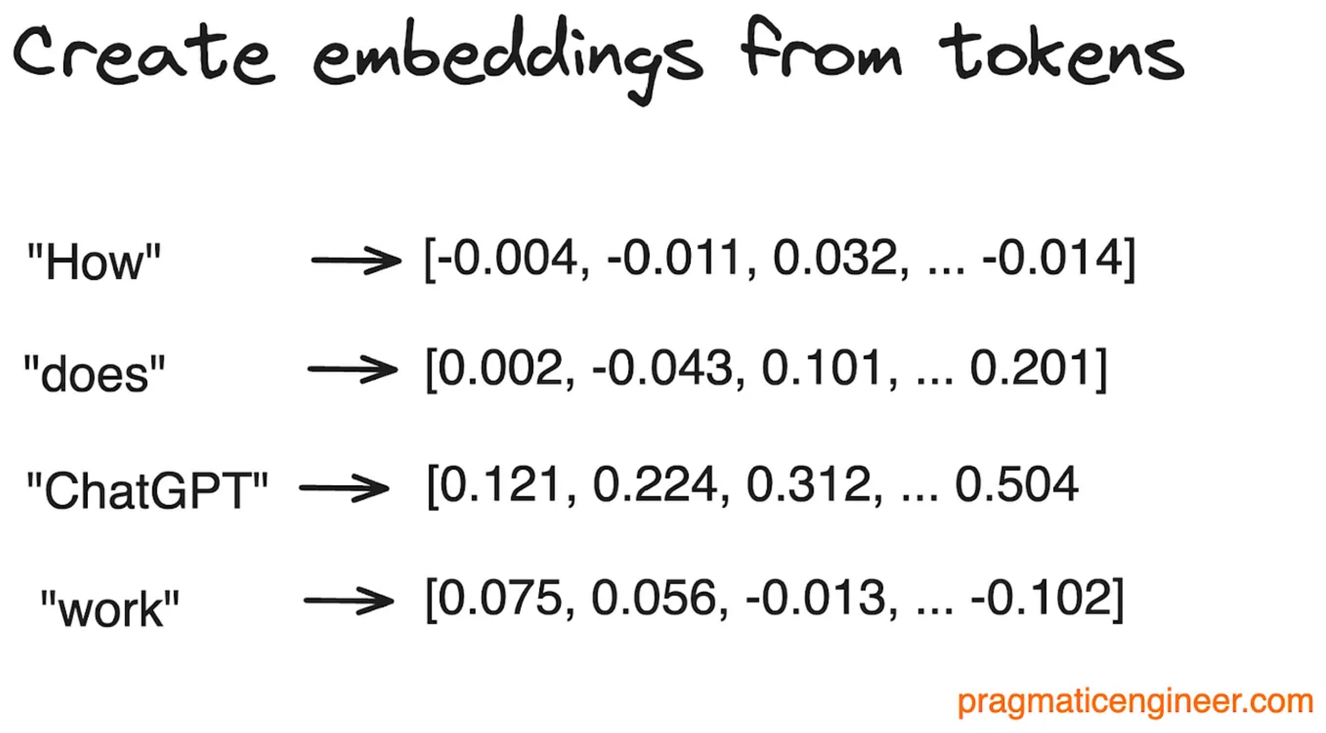
This step splits the data up and embeds it, which requires a little more explanation.

Create embeddings

We've broken our documents into chunks, hooray! But how will we know which chunks will be relevant for a question that's asked?

Here's how Evan Morikawa of OpenAI defines the concept of embeddings in the article, Scaling ChatGPT:

> "An embedding is a multi-dimensional representation of a token. We [OpenAI] explicitly train some of our models to explicitly allow the capture of semantic meanings and relationships between words or phrases. For example, the embedding for "dog" and "puppy" are closer together in several dimensions than "dog" and "computer" are. These multi-dimensional embeddings help machines understand human language more efficiently."


Create embeddings from tokens
"How" → [-0.004, -0.011, 0.032, ... -0.014]
"does" → [0.002, -0.043, 0.101, ... 0.201]
"ChatGPT" → [0.121, 0.224, 0.312, ... 0.504
"work" → [0.075, 0.056, -0.013, ... -0.102]
pragmaticengineer.com

Creating an embedding from a token (a string). Source: Scaling ChatGPT

Let me offer an alternative way for thinking about embeddings which I use, and that builds on the concept of vectors, and K nearest neighbor vector search.

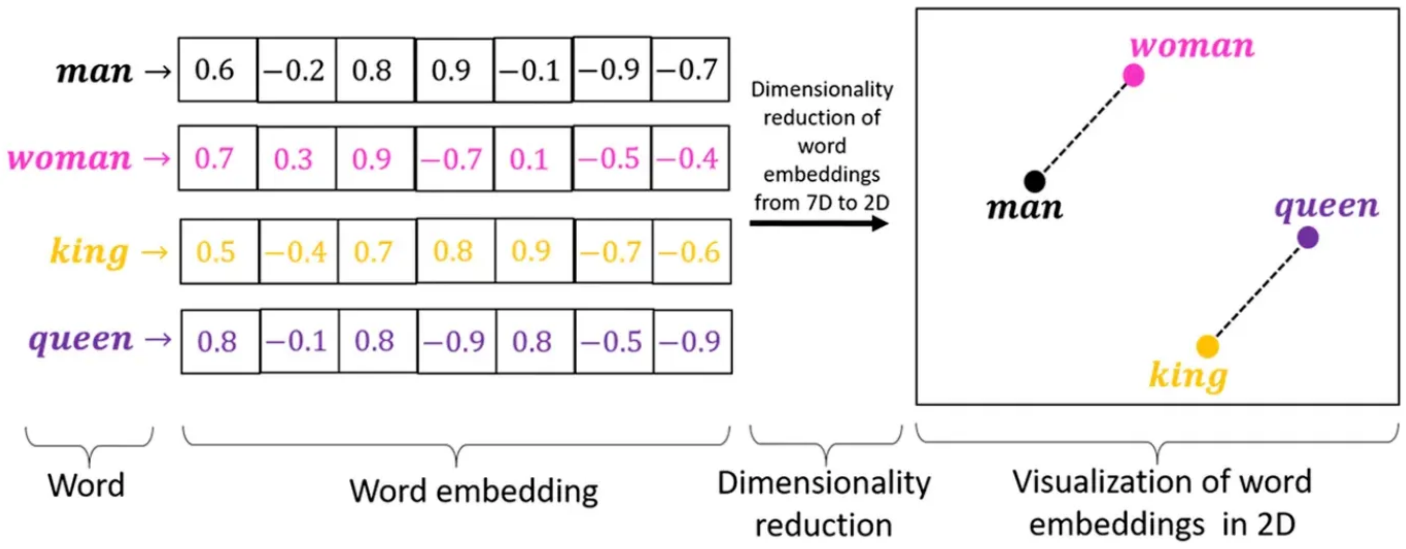
Vector. A vector is a string of numbers, which allows a piece of information like a sentence, or a paragraph to be expressed in a way an algorithm understands. When compared correctly, similar, related concepts are closer to each other in the vector space.

Imagine using keyword search and a more traditional search method for searching chunks; retrieving all the records from a database would typically require that you pre-categorized all the data, or had an index that could help you look them up. But we want something more flexible, as we don't have well structured metadata for every query we might want to run. Using vectors makes it easier to deal with this kind of problem space.

K-nearest neighbors (KNN.) KNN is an algorithm that takes a bunch of vectors and organizes them, based on how similar they are to each other. Using KNN on a collection of vectors, we find “similar concept groups".

A vector embedding is a vector that represents a concept like a token, a sentence, a paragraph, or anything else. It’s effectively the “fingerprint of an idea.”

Using vector embeddings makes it simpler for the AI model to interact with these concepts. It also makes it straightforward to search for similar concepts when it wants to query your database. For example the Vector of an *‘apple’* and the vector of a *‘pear’* will be more similar than the vectors of an *‘apple’* and an *‘app’*.


man → 0.6 −0.2 0.8 0.9 −0.1 −0.9 −0.7
woman → 0.7 0.3 0.9 −0.7 0.1 −0.5 −0.4
king → 0.5 −0.4 0.7 0.8 0.9 −0.7 −0.6
queen → 0.8 −0.1 0.8 −0.9 0.8 −0.5 −0.9
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

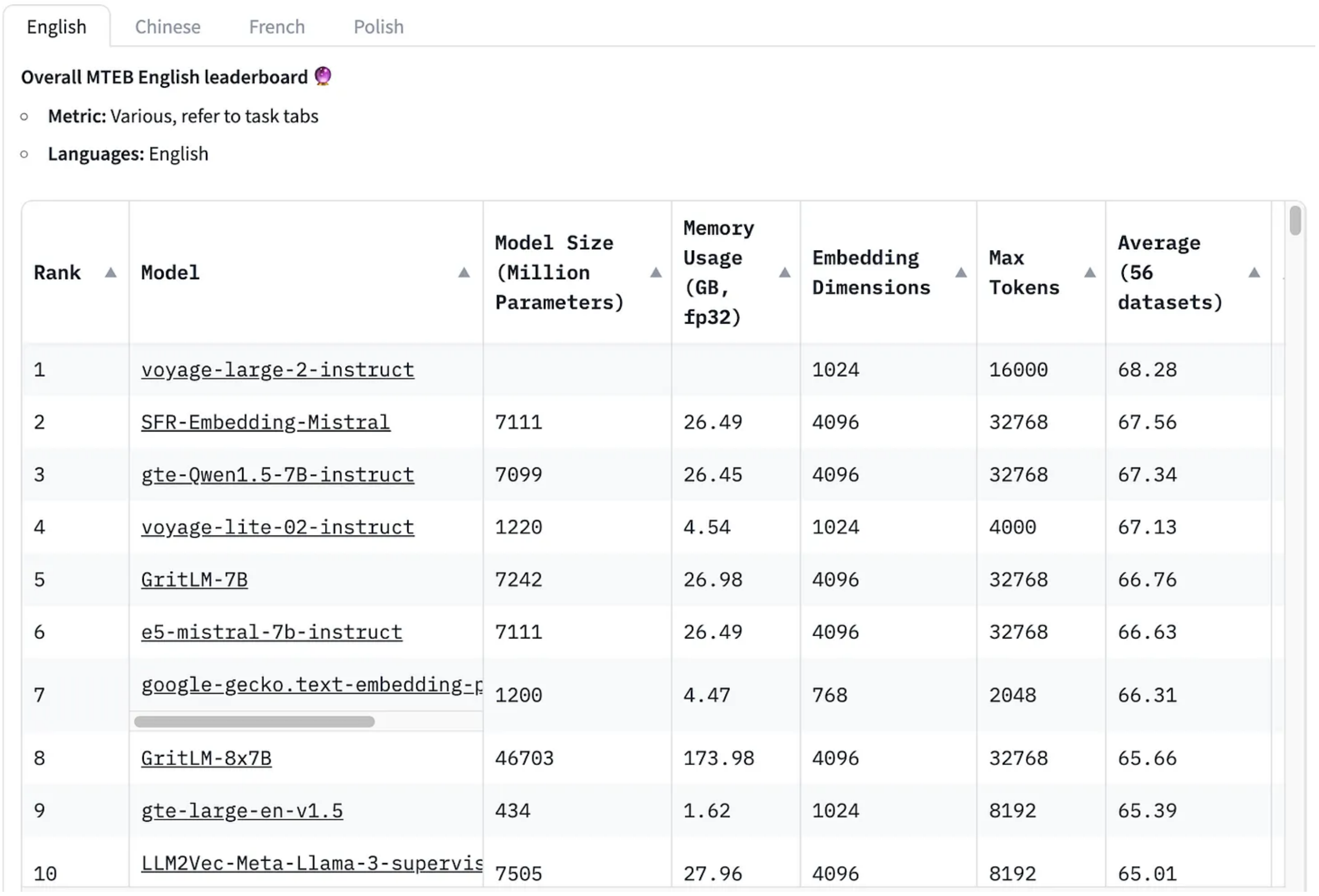
Turning words into embedding and visualizing those embeddings in a 2D space. Source: Hariom Gautam on Medium

Pre-trained embedding models generate a vector embedding from any input. Thanks to the pre-training, they already categorize the inputs reliably enough.

OpenAI offers an API called Embeddings that can be used to process chunks. Feed in a chunk, and receive an embedded vector in return. Of course, using this API comes with its own cost. The collected works of William Shakespeare are 3,000 pages long, or circa 835,000 words. Embedding the complete text with OpenAI's ada v2 embedding model would cost about $0.10 (as the cost is $0.10 per 1M tokens, and an English word usually comes to about 1.3 tokens)

When we query our data, the query goes through a similar process to embedding. The semantics of the query are deconstructed into vectors, and these vectors make it easy to compare with stored data.

The open source community also offers some exceptional options. The online AI community, Hugging Face, has a leaderboard of the best embedding models available, ranking them according to the Massive Text Embedding Benchmark (MTEB.)



English Chinese French Polish

Overall MTEB English leaderboard 🔮

- **Metric:** Various, refer to task tabs
- **Languages:** English

Rank	Model	Model Size (Million Parameters)	Memory Usage (GB, fp32)	Embedding Dimensions	Max Tokens	Average (56 datasets)
1	voyage-large-2-instruct			1024	16000	68.28
2	SFR-Embedding-Mistral	7111	26.49	4096	32768	67.56
3	gte-Qwen1.5-7B-instruct	7099	26.45	4096	32768	67.34
4	voyage-lite-02-instruct	1220	4.54	1024	4000	67.13
5	GritLM-7B	7242	26.98	4096	32768	66.76
6	e5-mistral-7b-instruct	7111	26.49	4096	32768	66.63
7	google-gecko.text-embedding-p	1200	4.47	768	2048	66.31
8	GritLM-8x7B	46703	173.98	4096	32768	65.66
9	gte-large-en-v1.5	434	1.62	1024	8192	65.39
10	LLM2Vec-Meta-Llama-3-supervis	7505	27.96	4096	8192	65.01

A screenshot of LLM models ranked by MTEB. Source: Hugging Face

What the best model is for you will depend on your use case, and most RAG pipelines will want to use a semantic embedding model like Bidirectional Encoder Representations from Transformers (BERT.)

Store the data

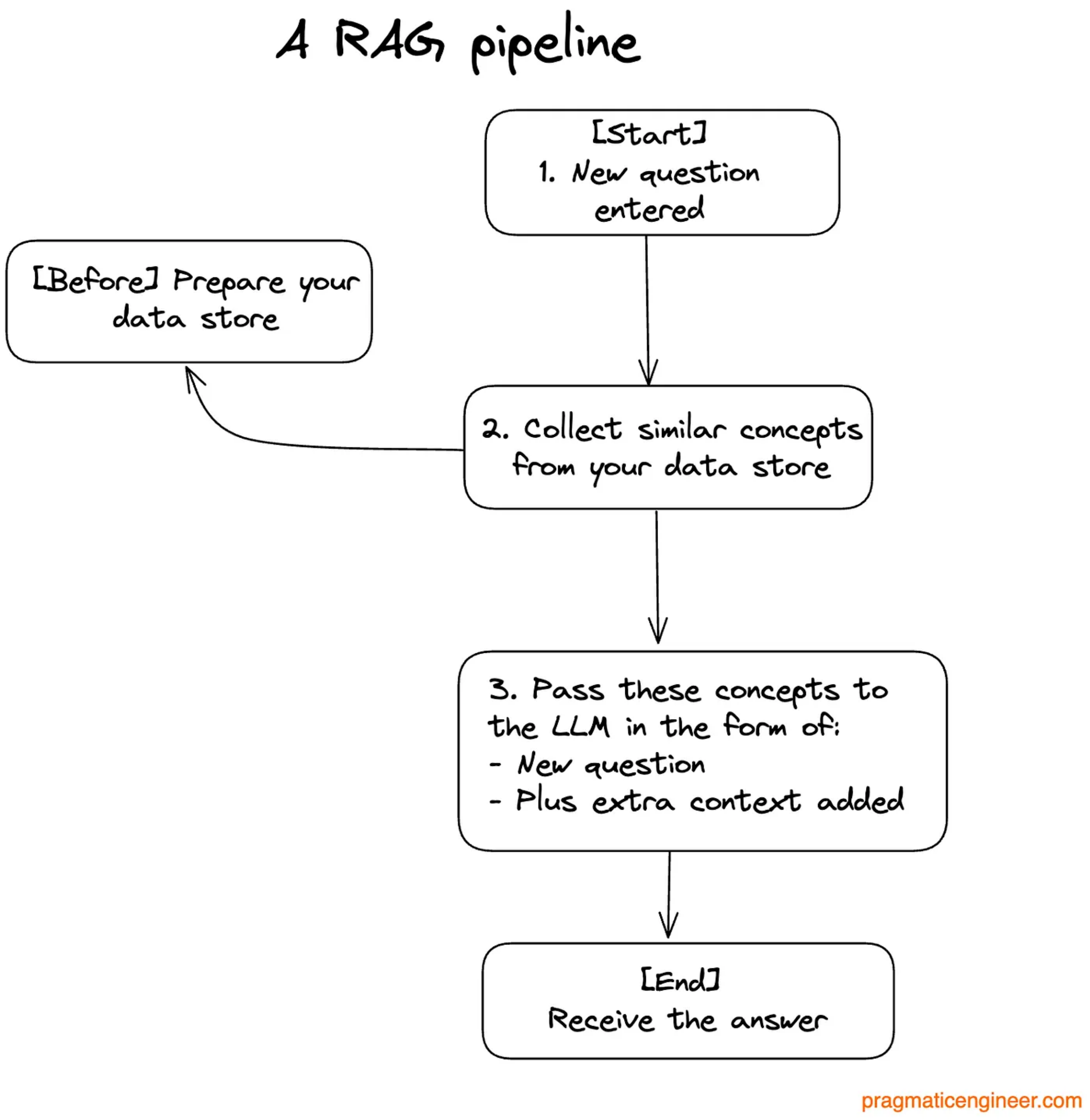
Now we've extracted and cleaned the data, chunked it, and created our embedding, it's time to store it. We need to choose a database that's effective at running our KNN operations. Vector databases are tailored to support KNN lookup with great performance, so using one will provide optimal performance for search.

Popular vector databases include Pinecone and Weaviate. Additionally, all major cloud providers offer multiple vector databases. Vector databases are a fast-evolving space, so you need to do research.

When prototyping, you can get away with using a more traditional database like MySQL or PostgreSQL to store embeddings. Should your application receive production traffic, the performance of these SQL databases will likely become critical enough to justify moving to a vector-based one.

5. Bringing it all together

With our data pipeline prepared, the remaining steps are surprisingly simple!

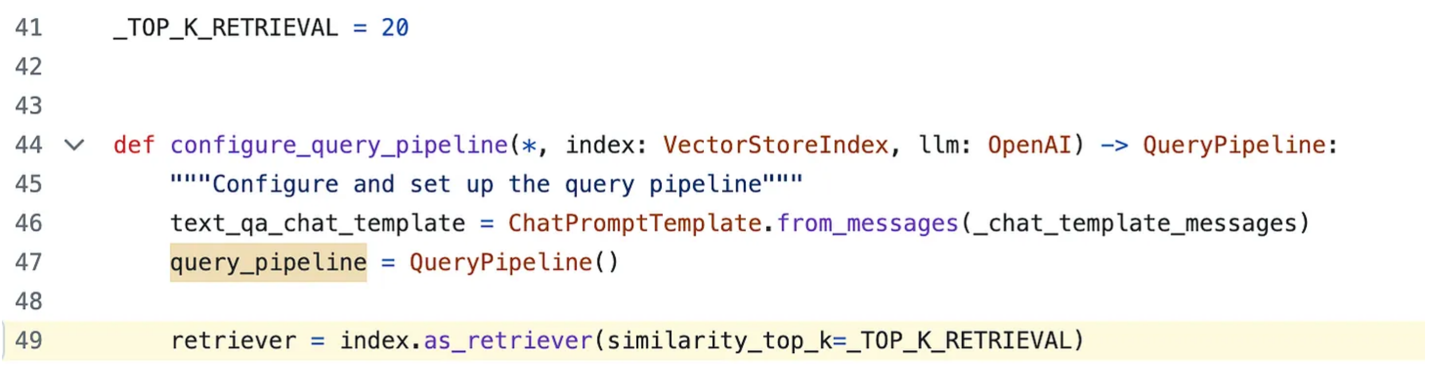

A RAG pipeline
[Start]
1. New question entered
[Before] Prepare your data store
2. Collect similar concepts from your data store
3. Pass these concepts to the LLM in the form of:
- New question
- Plus extra context added
[End]
Receive the answer
pragmaticengineer.com

A RAG pipeline at work. The data store is ready and we just need to do steps 2 and 3

The work that's left to do:

Step 2: Collect similar concepts from the data store. We use a vector database to query the data.

In our code, the very bottom line does the retrieval:



```python
_TOP_K_RETRIEVAL = 20


def configure_query_pipeline(*, index: VectorStoreIndex, llm: OpenAI) -> QueryPipeline:
    """Configure and set up the query pipeline"""
    text_qa_chat_template = ChatPromptTemplate.from_messages(_chat_template_messages)
    query_pipeline = QueryPipeline()

    retriever = index.as_retriever(similarity_top_k=_TOP_K_RETRIEVAL)
```

Collecting similar concepts (chunks) from our stored data. See this line [in the example codebase]

The "retriever" variable now contains the 20 chunks in similarity (as the value of _TOP_K_RETRIEVAL is 20).

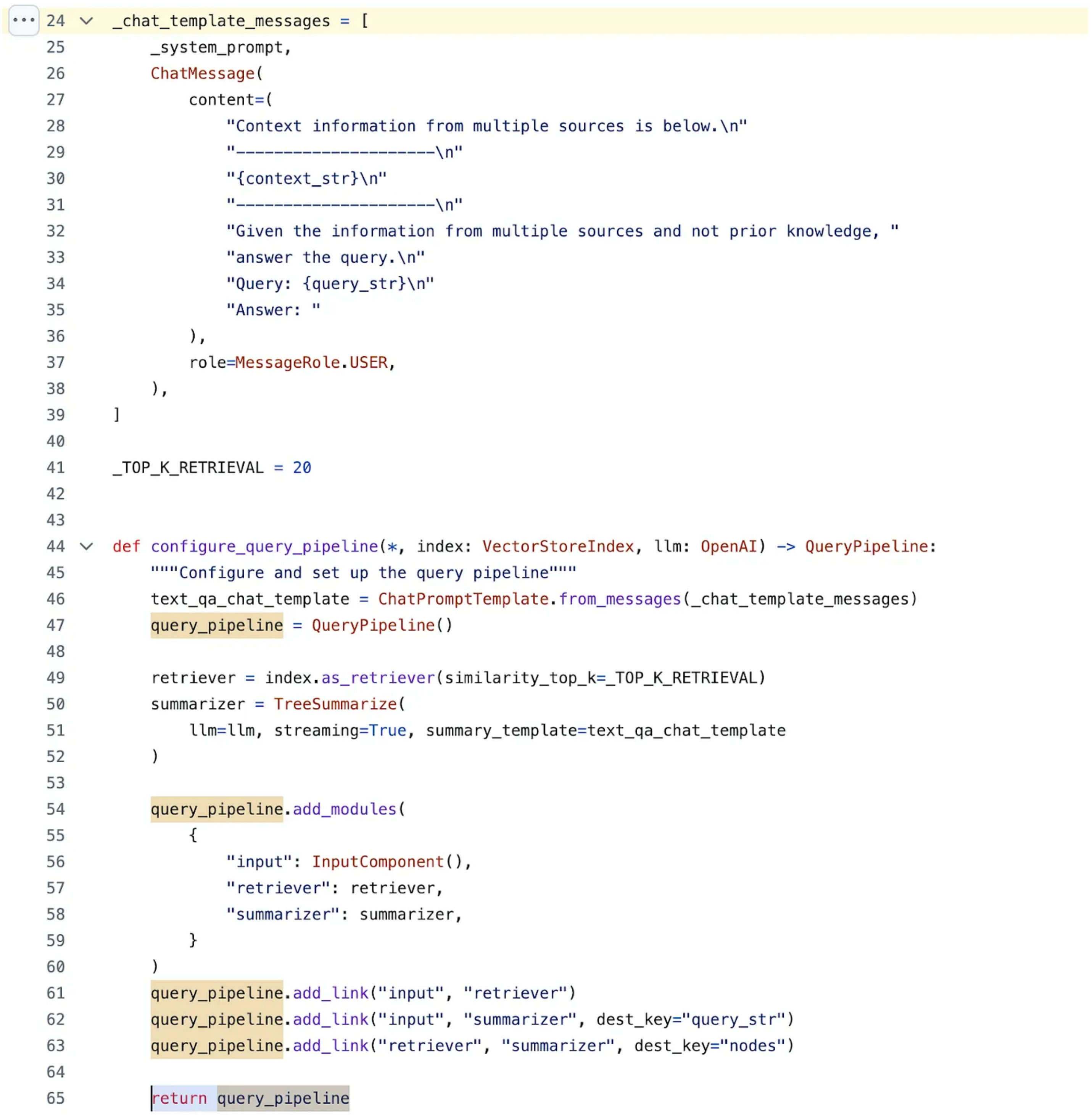
Step 3: Recombine these concepts with the original query to build a more relevant and authoritative answer.

With the related chunks available, we now create a new query, which is the updated query we want to pass into the LLM:
Context information from multiple sources is below.

> ---------------------
>
> {LIST OF THE 20 CHUNKS}
>
> ---------------------
>
> "Given the information from multiple sources and not prior knowledge, answer the

> query.
>
> Query: {ORIGINAL QUERY}
>
> Answer:

In our code, we create the above string by filling out the list of the 20 most relevant chunks (as *context_str*) and the original query (as *query_str*):



```python
_chat_template_messages = [
    _system_prompt,
    ChatMessage(
        content=(
            "Context information from multiple sources is below.\n"
            "---------------------\n"
            "{context_str}\n"
            "---------------------\n"
            "Given the information from multiple sources and not prior knowledge, "
            "answer the query.\n"
            "Query: {query_str}\n"
            "Answer: "
        ),
        role=MessageRole.USER,
    ),
]

_TOP_K_RETRIEVAL = 20


def configure_query_pipeline(*, index: VectorStoreIndex, llm: OpenAI) -> QueryPipeline:
    """Configure and set up the query pipeline"""
    text_qa_chat_template = ChatPromptTemplate.from_messages(_chat_template_messages)
    query_pipeline = QueryPipeline()

    retriever = index.as_retriever(similarity_top_k=_TOP_K_RETRIEVAL)
    summarizer = TreeSummarize(
        llm=llm, streaming=True, summary_template=text_qa_chat_template
    )

    query_pipeline.add_modules(
        {
            "input": InputComponent(),
            "retriever": retriever,
            "summarizer": summarizer,
        }
    )
    query_pipeline.add_link("input", "retriever")
    query_pipeline.add_link("input", "summarizer", dest_key="query_str")
    query_pipeline.add_link("retriever", "summarizer", dest_key="nodes")

    return query_pipeline
```

The code to generate our query pipeline. This is it! *Browse the code here.*

If you've followed this code-along, then congratulations! You now know how to code a simple RAG pipeline!

6. RAG limitations

RAG is a powerful set of tools that can help you focus AI onto your data. However, it's not a perfect fit for all use-cases, and there are some areas in which RAG does poorly.

Summarization

The RAG approach will not result in great output for summarizing. This is because with the RAG approach, documents are broken into many small sections, meaning that results will be poor if seeking insight that needs context from an *entire* document. For example, if asked: “give a summary of the key points in our contract with Microsoft,” an LLM that performs this well must process the entire contract, not just 2 or 3 chunks that are the assigned “key parts.”

For a model that performs well with summarization paths, consider an alternative approach of detecting use cases for which you route summarization queries to a different pipeline. For such cases, load the entire document that needs to be summarized. Doing so results in worse performance and higher cost, but it's the only way to get a more accurate response.

Multi-part, or hybrid questions

More complex questions often have an element of reasoning. Take the question:

“What percentage of agreements have their governing law in South Africa?”

This question needs to be broken down. Also, data needs to be collected from other data sources to determine the correct answer. For this specific question, to answer it correctly we need to follow these steps:

1. Retrieve all agreements with their law in South Africa.

2. Determine how many such agreements there are.
3. Process the math to recombine the output. It's *important to note that most LLMs are not good with math and you may want to not use an LLM for it!*

To do a good job with such a complex query, RAG alone is insufficient, although it's necessary at some steps. For the best results, build a higher-level orchestration layer, and coordinate other AI agents working in this pipeline to process complex queries.

7. Real-world learnings

We've only scratched the surface of RAG in building this simple pipeline. For the more technically-minded reader who wants to experiment in this space, below are seven things that we at Wordsmith wish we'd had known before, which would've saved time while building our AI solutions.

#1: Natural language is not always the best input for an LLM

Instead of using plain text, it is often better to use more structured input and output with LLMs; for example, JSON. A structured format simplifies parsing of the results, and can also increase the quality of an answer. This is because additional structured metadata can be passed along with the chunks of text to the prompt. The model can then be asked to produce additional outputs, and these extra outputs can help improve the answer itself.

This structured approach helps make your instructions highly targeted and very precise.

#2: The quality of your evaluation "evals" are critical for making reliable progress

"Evals" refers to a set of scenarios used to grade the quality of an agent's responses to questions. Each scenario has an input and an output which you can run as you go. LLMs have inherently unpredictable output, meaning there's a lot of trial and error. So it's essential to have a strong set of test cases for tracking your progress.

Invest time in defining these "evals" upfront. I've written more about how we approached our evaluation criteria.

#3: Improve performance by asking the LLM to do extra things

Here are two ways to significantly improve performance

1. Combine the context with the original question, then ask the LLM to rebuild a better answer.
2. Ask the LLM to capture the user's intent from the original question, and offer reasoning.

In my experience, both approaches improved the output's quality.

#4: Get the right token size as LLM context

If you feed too few chunks or too little context into the LLM, you'll get narrow, lightweight answers. Feed in too many chunks and too much context, and the model will start overlooking essential information and get confused. Experiment to get the right chunk size, and the correct number of tokens for better performance.

A simple reference point that seems to work well is passing about 16,000 tokens for GPT-4 Turbo.

#5: Chunking matters a lot – A LOT!

A way to improve performance is to blend multiple chunking strategies to create overlapping chunks, which can help build some resilience into the data to ensure you get the most relevant manifestation.

For example, create fixed-size chunks for 2,500 character size and 500 character size. Calculate the embeddings for both options, which means embedding the same data several times. During a search, your system will retrieve and use the best fitting chunk, which could be the shorter or longer one!

#6: Use suitable document parsers

There are so many open source solutions to help parse and pre-process documents, so take some time to research the best ones for your use case, whose output fits well.

Document parsers tend to struggle to handle certain formats, like nested numbered lists. But at Wordsmith, these are very important in contracts and legal documents! So we had to “hand-roll” our custom solution, after failing to find an open source document parser that did the job.

#7: Use output formatting which the model is comfortable with

Each foundation model has been trained on different source data. This means each model will work better or worse with certain input and output formats. For example, GPT4 and Mistral are efficient when using JSON and Markdown, suggesting they have been extensively trained with this kind of data. Meanwhile, Claude seems to work well when using Markdown, but less so with JSON. Experiment with models, learn which formats work better, and choose models based on their strengths.

Beyond RAGs

RAG is the foundation of nearly every LLM application, and this space is moving fast. I sense a “new dawn” is starting to break in multi-agent orchestration and interaction. These new architectures will give developers the ability to chain many agents together, with each one performing a specialist role in a pipeline. Such advanced pipelines will help progress beyond many of the constraints with which basic RAG approaches struggle.

Right now, the cost of running LLMs can still be pretty high; for example, our test suite cost $30 to execute on each run. However, the cost of running LLMs is falling quickly, and the performance of these tools is increasing just as fast.

It’s an exciting time to be building on Gen AI and LLMs. I hope this overview and code-along helps you get started!

A big thanks to Derek and Gigz for the effort they put into helping contribute to the hello_wordsmith repository!

Takeaways

Gergely again. Thanks very much Ross and the Wordsmith team, for this detailed walkthrough about building a RAG pipeline. You can follow Ross on LinkedIn, X or subscribe to his blog, where he writes on topics like defensible moats in the age of generative AI. Also, Wordsmith is hiring for product engineering and sales.

My takeaways from this deep dive:

Data is one of the biggest moats in most GenAI use cases. There are two types of AI startups:

1. Those building foundational models, of which there are a handful and among whom OpenAI is the best known, with its GPT models. There's also Anthropic (Claude,) Google (Gemini,) Meta (Llama,) and Mistral. These companies spend up to hundreds of millions of dollars on training these models, then offer them for use; sometimes for a fee, and sometimes for free.
2. Ones building applications on top of foundational models. The majority of startups utilize foundational models, and build creative use cases like professional headshots (Secta AI, as covered in the bootstrapped companies article,) or Wordsmith, which offers LLM-powered tools for legal professionals.

For the second category which includes most startups, the two biggest advantages are speed of execution, or access to unique data which competitors don't possess. Speed of execution can be a competitive advantage, but having access to data which competitors don't feels like the biggest, durable advantage for a startup.

RAG is one of the simplest ways to use "data as a moat" with AI models. For any company with a data moat, RAG is the simplest way to enhance any LLM model without exposing the underlying data to the outside world.

Sourcegraph has used RAG to produce standout code suggestions. Head of Engineering Steve Yegge wrote last December:

> "Cody's [Sourcegraph's AI coding assistant] secret sauce and differentiator has

> always been Sourcegraph's deep understanding of code bases, and Cody has tapped into that understanding to create the perfect RAG-based coding assistant, which by definition is the one that produces the best context for the LLM.
>
> That's what RAG (retrieval-augmented generation) is all about. You augment the LLM's generation by retrieving as much information as a human might need in order to perform some task, and feeding that to the LLM along with the task instructions. (...)
>
> Producing the perfect context is a dark art today, but I think Cody is likely the furthest along here from what I can see. Cody's Context has graduated from "hey we have vector embeddings" to "hey we have a whole squad of engines."

RAG is surprisingly easy to understand! At root, all RAG does is take an input query and add several sentences or paragraphs of additional context. Getting this additional context is an LLM search task, and performing this task involves preparing the "context" data, which is additional data which the LLM hasn't been trained on.

Unoptimized RAG is expensive! Running the example code, I saw a $1.38 charge after asking 5-10 questions from this model. I was wondering where the billing was coming from: the price of creating embeddings, or the cost of using GPT-4?

It turns out that all the cost was for using GPT-4, and the model charged $5 per 1M tokens. For each question I asked, the RAG pipeline added plenty of additional context, which made the query expensive in cost (and processing, I might add.) For a prototype approach, this cost is not a problem. However, for production use cases, heavy optimization would be needed, which could come from passing in more targeted – but less, overall – context. Or it could mean using a model that's cheaper to run, or operating the model ourselves for better cost efficiency.

A RAG pipeline is a basic building block of GenAI applications, so it's helpful to be familiar with it. One reason for this deep dive with Ross is that RAG pipelines are common at AI startups and products, but they remain a relatively new area, meaning that a lot of "build-it-yourself" takes place. It's easy enough to build a basic RAG pipeline, as Ross shows. The tricky part is optimizing things like chunking strategy, chunk size, and the context window.

If you need or want to build LLM applications, a RAG pipeline is a helpful early building block. I hope you enjoyed this deep dive into such an interesting, emerging area!

Update on 14 May 2024: updated token limits.

131 Likes · 5 Restacks

A guest post by

Ross McNairn

CEO @ https://www.wordsmith.ai/ ex CPTO @Travelperk, ex @skyscanner, @letgo. Exited Founder and lawyer. Discusses AI, tech, startups and venture.

Subscribe to Ross