15 Advanced RAG Techniques from Pre-Retrieval to Generation

ZAKEY FAIEQ

WILLOWTREE®
a TELUS International Company

March 2024

Table of Contents

The Author

Zakey Faieq

Connect with WillowTree's Data and AI Research Team (DART):

Michelle Avery | Group VP, AI
WillowTree, a TELUS International Company
michelle.avery@willowtreeapps.com

Let's connect

ai@willowtreeapps.com
1-888-329-9875
Boston | Charlottesville | Columbus | Durham
Lisbon | Porto Alegre | São Paulo | Vancouver

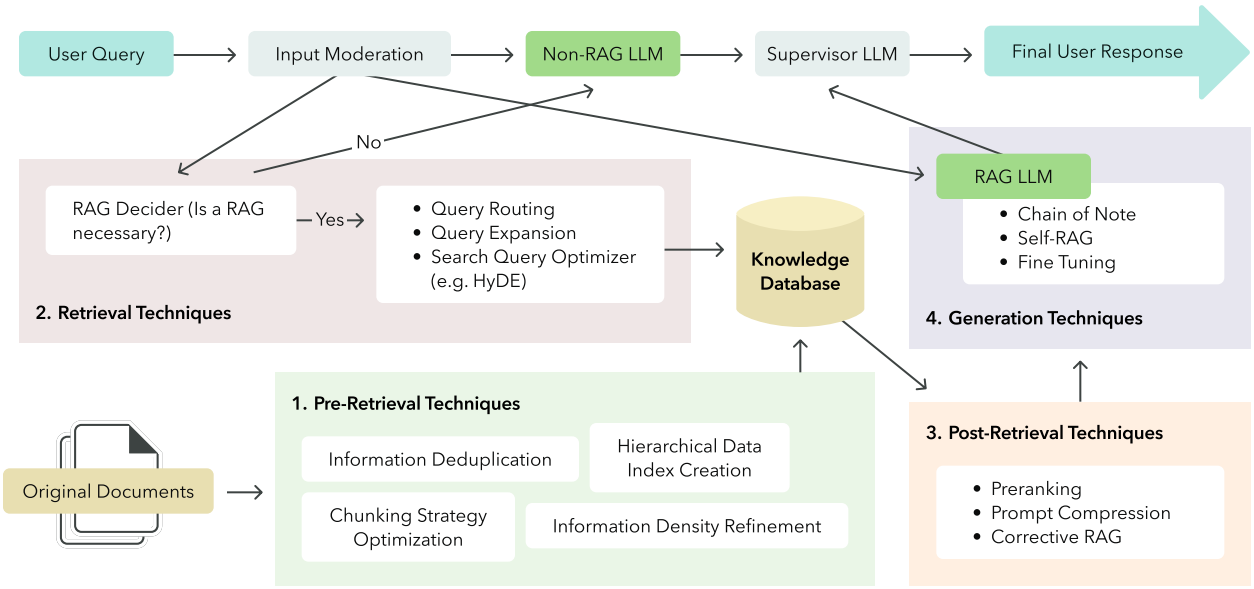
WILLOWTREE®
a TELUS International Company

Abstract

Our introductory article on retrieval augmented generation (RAG) introduced key concepts and looked at how RAG systems work. In this whitepaper, we explore 15 advanced RAG techniques for improving a generative AI system's output quality and overall performance robustness.

These advanced RAG techniques greatly expand your team's options to fine-tune system performance. For example, we experimented with most of the advanced RAG techniques explored in this whitepaper when building a safe conversational AI assistant for a major financial services firm. Doing so allowed us to test and identify the proper optimizations, from pre-retrieval through generation, using our automated RAG evaluation pipeline (see image below).

Note that most of the following techniques are discussed within the context of a conversational AI assistant powered by RAG.



1. Pre-Retrieval and Data-Indexing Techniques

Pre-retrieval optimizations primarily consist of improving the quality and retrievability of the information in your data index or knowledge database. The techniques and level of effort needed here largely depend on the nature, sources, and size of your data.

For example, optimizing information density improves user experience and lowers costs by producing more accurate responses with fewer tokens. But how would we optimize changes from system to system? The optimizations that will enhance retrieval for a travel industry chatbot could backfire spectacularly in a financial AI assistant because each depends on information of a different nature and distinct regulatory frameworks.

LLMs offer us many ways to optimize information during pre-retrieval, allowing us to test and fine-tune different approaches depending on our goals. Here are five LLM-based advanced RAG techniques worth exploring at the pre-retrieval stage, including several variations and improvements on the primary Technique 1.

Technique 1: Increase information density using LLMs

You can significantly improve the performance of your RAG system by using LLMs to process, clean, and label data before storage. This improvement is due to the fact that unstructured data from heterogeneous data sources (e.g., PDFs, scraped web data, audio transcripts) is not necessarily built for RAG systems, creating issues such as:

- low information density
- irrelevant information and/or noise
- information duplication

Low information density forces RAG systems to insert more chunks into the LLM context window to correctly answer a user query, increasing token usage and cost. Furthermore, low information density dilutes relevant information to the point an LLM could respond incorrectly. GPT-4 appears relatively resistant to this problem when using less than 70,000 tokens, but other models may not be as robust.

Here's a recent scenario we encountered that could easily occur when working on a RAG system: We scraped hundreds of web pages as our primary data source, but the original HTML contained a significant amount of irrelevant information (e.g., CSS classes, header/

footer navigations, HTML tags, redundant information between pages). Even after stripping the CSS and HTML programmatically, the information density was still low.

So, to improve the information density in our chunks, we experimented with using GPT-4 as a fact extractor that gleans relevant information from a document. After stripping CSS and HTML tags, we used an LLM call resembling the one below to process every scraped web page before chunking and inserting them into our knowledge base:

```
fact_extracted_output = openai.ChatCompletion.create(
model="gpt-4",
                messages=[
                    {
                        "role": "system",
"content": "You are a data processing assistant. Your task is to
extract  meaningful information from a scraped web page from XYZ
Corp. This information will serve as a knowledge base for further
customer inquiries. Be sure to include all possible relevant
information that could be queried by XYZ Corp's customers. The output
should be text-only (no lists) separated by paragraphs.",
                    },
                    {"role": "user", "content":  <scraped web page>},
                ],
                temperature=0)
```

Here's an anonymized example of scraped web content that went through our pipeline. Starting with the left column, we see that useful information density is low for the raw HTML snippet.

Raw HTML

<html lang="en-US" xml:lang="en-US"><head class="at-element-marker">

<script async="" src="https://cdn.branch.io/branch-latest.min.js"></script><script id="launch" data-launcher="success">
document.addEventListener("at-library-loaded", (function(e) {
document.getElementById('launch').setAttribute('data-launcher','success')
}
));
</script>

<meta charset="UTF-8">
<title>High Interest IRA Savings Account | Example Bank Select Savings IRA</title>

<meta name="description" content="Learn about Example Bank's Preferred Savings IRA, a high interest IRA savings account with higher rates when you bundle your Example accounts. Visit a Example Bank to open an account now.">
<meta name="template" content="example-generic-page-template">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="https://www.example.com/us/en/personal-banking/ira/savings" rel="canonical">
<link rel="stylesheet" href="/etc.clientlibs/examplesite/clientlibs/clientlib-base.lc-1681110478373-lc.min.css" type="text/css">
<script type="text/javascript">
var province_list = "{\x22provinces\x22:[]}";
var provinceCityPage = "false";
var ratesAPIInfo =

HTML Stripped of CSS/JS/HTML Tags

High Interest Retirement Account Savings | Example Bank Selective Savings Retirement Account
Example Savings Retirement Account
1
Increase your earnings by maintaining a higher balance and connecting a qualifying Example account
2
$10,000
Initial deposit requirement to open
Rate enhancement
Superior rates when you connect a qualifying Example account
2
No monthly charges
No monthly maintenance charges
Secure
Initiate in a physical location
Compare
Benefits you receive
Boost your earnings by maintaining a higher balance and connecting a qualifying account.
2
Retirement savings
An Example Savings Retirement Account could be an optimal choice for those seeking a superior interest rate with the flexibility to add funds anytime
$10,000 initial deposit
Permits additional contributions
location
You are examining info for _____
Modify your location
location icon
NA,NA
Region
Choose a region
Connecticut
Delaware

GPT-4 Fact Extraction

GPT-4 Fact-Extracted Content

Example Bank offers a high-yield retirement savings account titled the Example Savings Retirement Account. This account enables you to enhance your earnings by keeping a larger balance and connecting a qualifying Example account. To initiate this account, a minimum of $10,000 is required. Additionally, this account is exempt from monthly maintenance fees.

The Example Savings Retirement Account is an excellent option for those seeking a competitive interest rate alongside the ability to contribute funds at any moment. Your interest rate escalates as your balance increases, aiding in maximizing your savings. An extra rate increase is available when you link a qualifying Example Bank mortgage, home equity loan, credit card, or an operational personal or small business checking account.

This account promotes the acceleration of your retirement savings with its tiered interest rates. It also provides tax advantages. For Traditional IRAs, earnings accumulate tax-deferred, and upon withdrawal, earnings are taxed as income. Contributions might be tax-deductible. For Roth IRAs, earnings accumulate tax-free, and earnings withdrawn are not subject to income tax if they meet the criteria for a qualified distribution.

The Example Savings Retirement Account does not charge a monthly maintenance fee, earns interest at tiered rates, and permits additional contributions. It offers a rate increase upon linking a qualifying account and supports retirement savings with a competitive interest rate and the flexibility for anytime contributions.

For specific tax advice or to determine the most suitable type of IRA for your situation, it's recommended to consult with a tax advisor or refer to IRS Publication 590. This publication is available by contacting the IRS at 1-800-829-3676 or via their website at www.irs.gov.

Things are a little better in the central snippet with programmatically stripped CSS, JS, and HTML, but it still doesn't contain high-quality information. **Now, look at the right-hand snippet with fact-extracted content processed by GPT-4 and notice the leap in information density.**

That's what we used in our chunking and embedding process. Notably, the token counts for the text from a single webpage at the various stages were:

- Raw HTML: ~55,000 tokens
- Stripped HTML: 1,500 tokens
- **GPT-4 processed HTML: 330 tokens**

While the most significant reduction in token count (20x) occurred with the programmatic stripping of CSS, JS, and HTML, **the GPT-4 fact-extraction step applied to the stripped HTML consistently reduced token counts further by a factor of 500%**.

We tried a pipeline where we kept HTML tags in case there was some **semantic meaning** inherent to the structure of the HTML, but in our particular case, our RAG evaluation metrics showed improved performance with the tags left out.

Caveat: Risk of Information Loss

The risk of using LLMs to increase information density is that critical information could be lost. One strategy to mitigate this is ensuring that the maximum size of the fact-extraction LLM output is a) not capped or b) lower than the size of the input content, in case the input content was already information-dense and 100% useful.

Technique 2: Apply hierarchical index retrieval

Search can be made more efficient via multi-layer retrieval systems utilizing LLM-generated summaries. The practice of hierarchical index retrieval uses document summaries to streamline the identification of relevant information for response generation.

The previous section focused on improving information density without losing relevant information, similar to lossless compression. However, in generating summaries of documents, LLMs do something more analogous to lossy compression.

These summarized documents support the efficient search of large databases. Instead of only creating a single data index consisting of document chunks, an additional data index composed of document summaries creates a first-layer filtering mechanism that excludes document chunks from documents with summaries irrelevant to the search query.

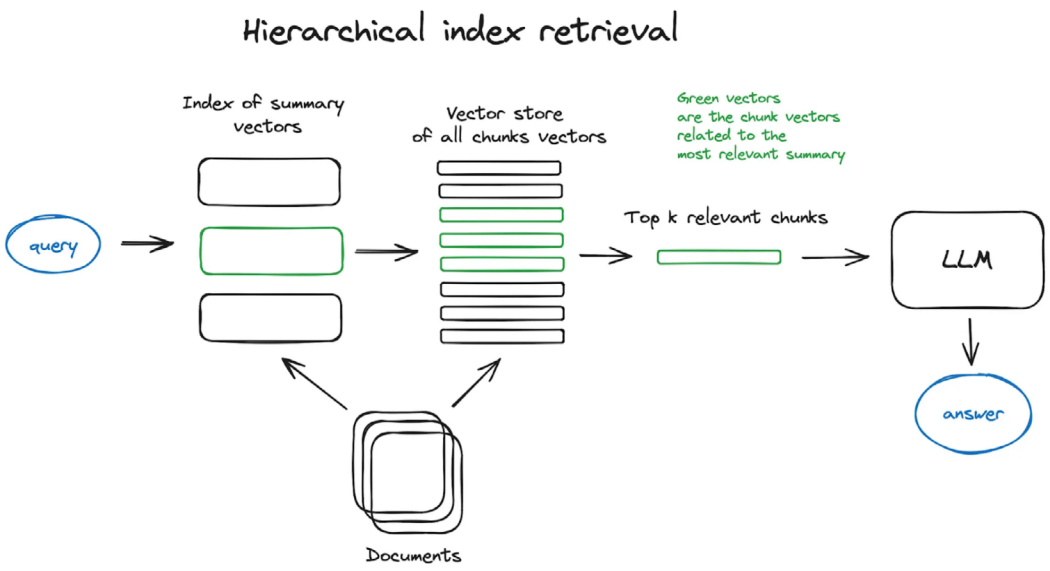


Image credit: towardsai.net

Technique 3: Improve retrieval symmetry with a hypothetical question index

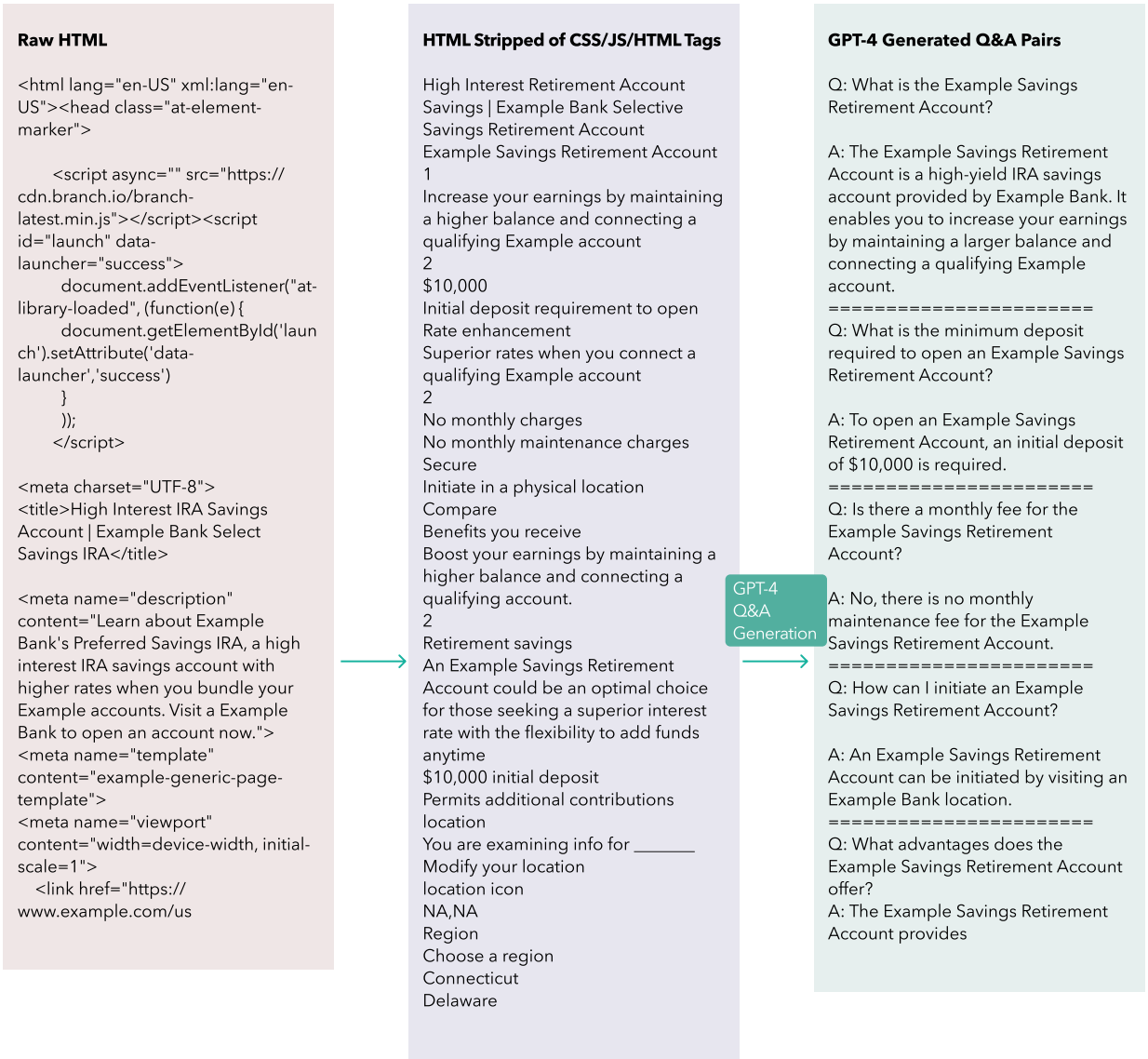
LLMs can also transform documents into a format optimal for both the embedding model and queries used in the RAG system. One method is to use GPT-4 to generate a list of hypothetical/possible question-and-answer pairs for each document, then use the generated questions as the chunks to be embedded for retrieval.

At retrieval time, the system will retrieve both the question and its corresponding answer and provide them to the LLM. **Thus, the embeddings for the queries likely have a much greater cosine similarity to the embeddings of the generated questions**. This similarity reduces the risk of losing relevant context in the chunking process. Each Q&A pair is thus self-contained and will theoretically contain all the required context.

Asymmetry between the query and the documents used for retrieval is a common issue in RAG systems. Queries are typically short questions like, "What is the best travel credit card offered by XYZ financial institution?" However, the relevant document chunks for that query are much longer (e.g., extensive paragraphs containing breakdowns of all the credit cards offered by XYZ financial institution).

This financial services example poses a problem for semantic search: if the query is markedly different from the document chunks (i.e., too much asymmetry), semantic similarity may be low, which could yield poor search results and bias the system toward the wrong information.

Here's a diagram illustrating how we used a hypothetical question index:



And here's the prompt we used:

```
generated_question_answer_pairs = openai.ChatCompletion.create(
                model="gpt-4",
                messages=[
                    {
                        "role": "system",
"content": "Analyze the provided text or html from Example bank's
```

```
website and create questions an Example bank customer could ask a
chatbot about the information in the text. You should not create
a question if it does not have a useful/informative answer to it
that would be helpful for a customer. For every question, please
formulate answers based strictly on the information in the text.
Use Q: for questions and A: for answers. Do not write any other
commentary. Questions should not reference html sections or links.
Create as many useful Q&A pairings as possible.",
                    },
                    {"role": "user", "content":  <scraped web page>},
                ],
                temperature=0)
```

Using LLMs to generate Q&A pairs can also be helpful for RAG benchmarking and evaluation. The Q&A pairs can serve as the gold standard dataset for questions and expected responses that the RAG system as a whole should be able to answer.

As for decreasing chunk size, that approach would only go so far, as chunks must be a minimum size to maintain enough context information to be useful. **Even with larger chunk sizes, there's always a risk of losing critical context information in the chunking process**. This risk can be mitigated (but not eliminated) by experimenting based on chunking considerations such as size and overlap ratio.

Caveat: Risks and alternatives to a hypothetical question index

Information loss is still a risk with this advanced RAG technique. For highly information-dense documents, LLMs may not be able to generate enough Q&A pairs to cover the range of queries users may have relative to the information in a document.

Additionally, depending on your document store's size, using an LLM to process and transform every document to alleviate query-document asymmetry may be cost-prohibitive.

Finally, depending on the traffic of your RAG system, a more effective solution may be a reverse approach called hypothetical document embeddings (HyDE) to transform user queries instead of documents. We discuss HyDE further in the Retrieval Techniques section below.

Technique 4: Deduplicate information in your data index using LLMs

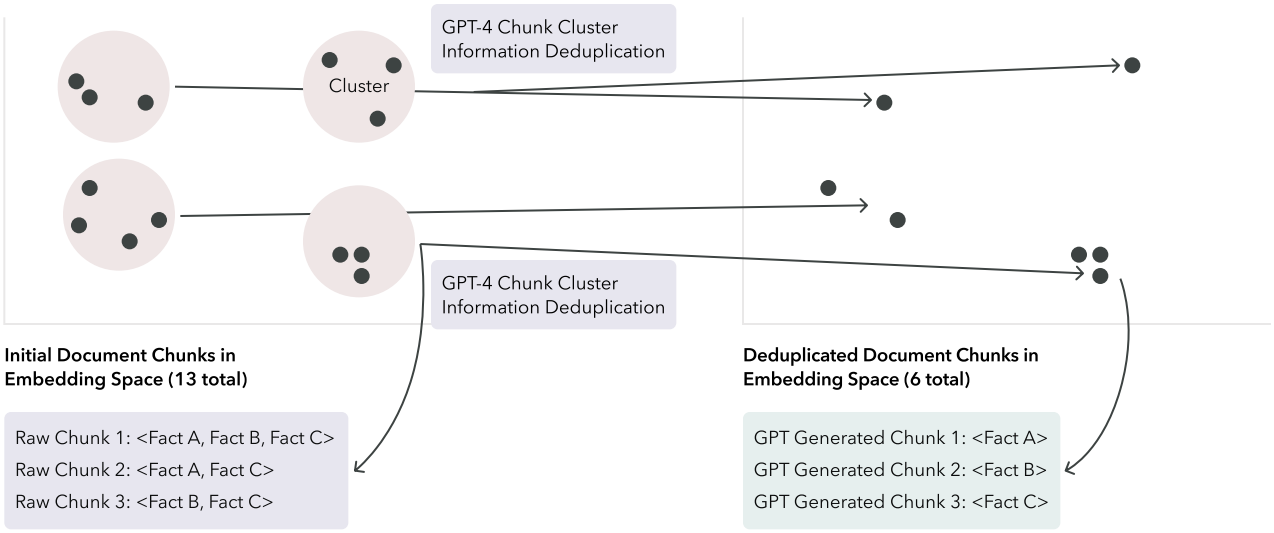
Using an LLM as an information deduplicator can improve the quality of your data index. The LLM deduplicates information by distilling the chunks into fewer chunks, improving the odds of a desirable response.

This is a valuable technique because, depending on the circumstances, information duplication in your data index may help or hinder your RAG system's output. On the one hand, if the correct information needed to answer a query is duplicated within the context window of the LLM generating a response, it increases the likelihood the LLM will respond desirably.

Conversely, suppose the degree of duplication dilutes or even entirely crowds out the desired information from the LLM's context window. In that case, users may receive a non-answer – or worse, an [AI hallucination] from the RAG system.

We can deduplicate information by k-means clustering the chunks in our embeddings space so the aggregate token count in each cluster of chunks fits within an LLM's effective context window. From here, we can task the LLM to simply output a distilled set of new chunks from the original cluster that removes duplicated information. If a given cluster contains N chunks, we should expect this deduplication prompt to output <= N new chunks where any redundant information is removed in the new chunks.

The image below shows chunks in an embedding space before and after information deduplication.



NOTE: This process makes it more difficult to include any citation system in your RAG system, as the new chunks may contain information from multiple documents. Also, the risk of information loss is still present, as is the risk of de-emphasizing information that may have benefited from being duplicated in the retrieval system.

Technique 5: Test and optimize your chunking strategy

The techniques above highlight the importance of chunking strategy. **But the optimal chunking strategy is specific to the use case, and a multitude of factors influence it**. The only way to find the optimal chunking strategy is to extensively A/B test your RAG system. Here are some of the most important factors to consider when testing.

Embedding model

Different embedding models have different performance characteristics at different input sizes. For instance, embedding models from sentence transformers excel with embedding single sentences, whereas text-embedding-ada-002 can handle much larger inputs. Chunk sizes should ideally be tailored to the specific embedding model used, or vice versa.

Nature of content to be embedded

Depending on the information density, format, and complexity of your documents, chunks may need to be a certain minimum size to contain enough context to be useful for the LLM. However, this is a balancing act. If the chunks are too large, they may dilute relevant information in the embedding, reducing retrieval odds for that chunk during semantic search.

If your documents don't contain natural breakpoints (e.g., a textbook chapter sectioned with subheaders) and documents are being chunked based on an arbitrary character limit (e.g., 500 characters), there's a risk of critical context information being split apart. In this case, an overlap should be considered. For instance, a chunking strategy with a 50% overlap ratio would mean that two adjacent 500-character chunks from a document will overlap each other by 250 characters. Information duplication and the cost of embedding more data should be taken into consideration when deciding on the overlap ratio.

Complexity or type of queries to be embedded

If your RAG system handles queries made in large paragraphs, then chunking your data up into large paragraphs makes sense. However, large chunk sizes may not facilitate optimal information retrieval if queries are only a few words long.

LLM capabilities, context window, and cost

While GPT-4 seems capable of handling many large chunks, smaller generative models may not perform as well. Also, running inference on many large chunks can be costly.

Amount of data to be embedded

Embeddings must be stored somewhere, and smaller chunk sizes result in more embeddings for the same amount of data, meaning increased storage requirements and costs. More embeddings may also increase the computational resources required for a semantic search, depending on how your semantic search is implemented (exact nearest neighbor vs. approximate nearest neighbor).

Our experience

By conducting extensive A/B testing with the help of our LLM-powered RAG evaluation system, we can evaluate the best chunking strategy for each of our use cases.

We tested the following chunking strategies, mainly on the improved information-dense documents processed by GPT-4:

- 1,000-character chunks with 200-character overlap
- 500-character chunks with 100-character overlap
- Paragraphs (paragraph breaks existed in the processed docs)
- Sentences (split using spaCy)
- Hypothetical questions (embedding questions from the generated hypothetical question index detailed above)

When building our financial services AI assistant, we found that the choice of chunking strategy wasn't very impactful (see results in the table below). The 1,000-character chunks with a 200-character overlap strategy performed marginally better than the other strategies.

Chunking Strategy (with 3,500 tokens' worth of chunks inserted into GPT-4 prompt)	Results (GPT-4 evaluation of truthfulness compared to gold standard response | Score between 1–5)
1,000-character chunks with 200-character overlap	4.34
Hypothetical questions (embedding questions from the generated hypothetical question index detailed above)	4.23
Paragraphs (paragraph breaks existed in the processed docs)	4.2
500-character chunks with 100-character overlap	4.19
Sentences (split using spaCy)	3.96

Our hypotheses for why the chunking strategy did significantly impact our use case include:

- the domain we built our RAG application for wasn't too complex
- the dataset was too small for the chunking strategy to impact it much
- the processing we did with GPT-4 to improve information density made the biggest impact

However, with just one change to our application — such as building for a domain with greater complexity — and we'd likely see different results.

Other pre-retrieval optimization techniques

There are more advanced RAG techniques for pre-retrieval and data indexing that we're eager to test. These include **recursive retrieval**, which supports complex, multi-step queries by iterating the output from one retrieval step and using it as the input for another step. **Context enrichment** via sentence window and parent-child chunk retrieval also interests us as a way to improve search and enrich LLM responses.

2. Retrieval Techniques

Retrieval optimizations cover advanced RAG techniques and strategies targeted at inference time with the goal of improving search performance and retrieval results before retrieval occurs.

Technique 6: Optimize search queries using LLMs

Search systems tend to work optimally when search queries are presented in a specific format. LLMs are a powerful tool for tailoring or optimizing user search queries for a specific search system. To illustrate this, let's look at two examples: optimizing a simple search query and a query for a conversational system.

Simple search query optimization

Let's say a user wants to search for all news articles from example-news-site.com about Bill Gates or Steve Jobs. They might type something like this into Google:

```
(Suboptimal Google Search Query):
Articles about Bill Gates or Steve Jobs from example-news-site
```

An LLM can be used to optimize this search query specifically for Google by utilizing a few advanced search features Google provides.

```
(Optimal Google Search Query):
“Bill Gates” OR “Steve Jobs” -site:example-news-site.com
```

This approach works fine for simple queries, but for conversational systems, we need to step things up.

Search query optimization for conversational AI systems

Though the simple search query optimization detailed above can be viewed as an enhancement, we've found that using LLMs to optimize search queries for RAG in conversational systems is critical.

For a simple Q&A bot capable of only having single-turn exchanges, the user's question doubles as the search query to retrieve information for augmenting an LLM's knowledge. But things get a little more tricky in conversational systems. Take the following example conversation:

```
Customer: " What are the interest rates for your CDs?"
Assistant: "Our interest rates are XYZ."
Customer: "Which credit card is good for travel?"
Assistant: "The XYZ credit card is good for travel for ABC reasons"
Customer: "Tell me more about the interest rate for that"
```

To answer the user's last question, a semantic search likely needs to occur to retrieve information regarding the specific XYZ travel credit card. What, then, should the search query be? Merely using the last user message is not specific enough since a financial institution likely has many products that yield interest. In that case, the semantic search would yield lots of potentially irrelevant information that could crowd out actual relevant information in the LLM's context window.

What about using the entire conversation transcript as a semantic search query? That may yield better results, but they'll still likely include information about CDs which is not relevant to the user's latest question in the conversation.

The best technique we've found so far is to use an LLM to generate the optimal search query given a conversation as input. For the example conversation above, the prompting looked something like:

```
const messages = [...]

const systemPrompt = `You are examining a conversation between
a customer of Example bank and an Example bank chatbot. A
```

```
documentation lookup of Example bank's policies, products, or
services is necessary for the chatbot to respond to the customer.
Please construct a search query that will be used to retrieve the
relevant documentation that can be used to respond to the user.`


    let optimizedSearchQuery = await this.textCompletionEngine.
complete(
            [
                { role: 'system', content: systemPrompt },
                { role: 'user', content:
stringifyChatConversation(messages) }],
            'gpt-4',
            {
                temperature: 0,
                maxTokens: 100,
            },
        );
```

A variation of this technique is query expansion, where multiple subsearch queries are generated by an LLM. This variation is especially useful in RAG systems with a hybrid retrieval system that combines search results from multiple data stores with different structures (e.g., SQL database + separate vector DB). Other prompt engineering techniques like **step-back prompting** and HyDE (discussed in the next section) can also be combined with this approach.

Technique 7: Fix query-document asymmetry with hypothetical document embeddings (HyDE)

As mentioned in our Pre-retrieval Techniques section, we can leverage LLMs to resolve query-document asymmetry and improve retrieval results. We can also achieve greater semantic similarity during the retrieval stage by applying HyDE.

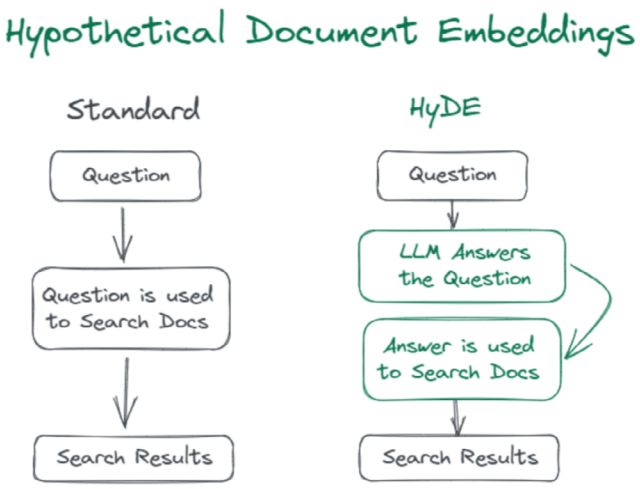


Image credit: [wfhbrian.com]

We do this by asking an LLM at inference time before retrieval has occurred to generate a hypothetical document or document chunk that answers a query. Here's an example prompt that we used along with the 1000-character chunking strategy to generate hypothetical documents for semantic search:

```
prompt = 'Please generate a 1000 character chunk of text that
hypothetically could be found on Example banks website that can help
the customer answer their question.';
```

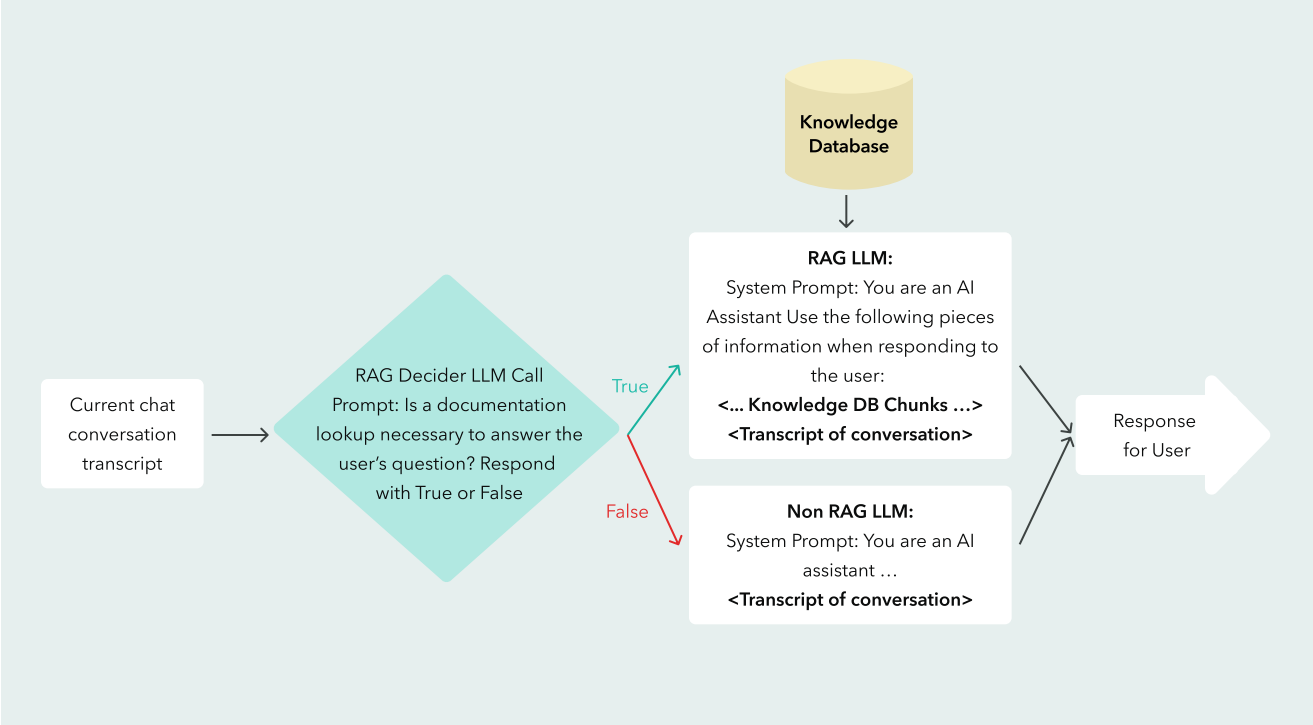
This hypothetical document or chunk is then embedded and used instead of the user query to conduct the semantic search. The idea is the hypothetical document or chunk will have greater semantic similarity to the desired chunk than the user query itself in RAG systems with query-document asymmetry.

Technique 8: Implement query routing or a RAG decider pattern

The query router is one of the more popular advanced RAG techniques we've seen. The idea is to use LLMs to route search queries to the appropriate database when a RAG system uses multiple data sources. This involves predefining the routing decision options in a prompt and parsing the LLM's routing decision output so it can be used in code.

To help lower costs and improve quality in our RAG, **we developed a variation of this technique we call the RAG decider pattern.**

Not all queries require a RAG lookup. So, identifying when an LLM can answer a query independently without external knowledge results in greater efficiency. A basic example is when a user asks a simple question that does not require RAG information, such as: "Hi, is this customer support?"



A less obvious example is when all the information needed to answer a user query already exists in recent conversation history. In this case, the LLM simply needs to repeat or slightly rephrase what it said previously. For instance, "Can you please translate your last message into Spanish?" or "Please explain that last message like I'm five years old." Both queries do not require a new retrieval to occur, as an LLM can answer the queries simply using its built-in capabilities.

In our case, when the RAG decider has made the decision that a full RAG lookup is unnecessary for a given user query, we substitute the typical RAG prompt with a prompt that doesn't mention anything about RAG results or documents.

An alternative method may be to enable a singular LLM agent to decide whether or not to conduct a retrieval step via function calling or some other mechanism (e.g., reflection tokens as introduced in **Self-RAG**) instead of delegating to a separate LLM call.

3. Post-Retrieval Techniques

Post-retrieval optimizations cover strategies or techniques employed after retrieval has occurred but before final response generation.

A critical consideration at this point: **Even if all of the pre-retrieval and retrieval strategies above have been deployed, there's still no guarantee that our retrieved documents will contain all the relevant information needed for the LLM to answer a query**.

That's because retrieved documents can be a mixture of any or all of the following categories:

- Relevant documents (i.e., documents containing the information needed to answer a user query)
- Related but irrelevant documents
- Irrelevant and unrelated documents
- Counterfactual documents (i.e., documents that contradict the correct relevant documents)

Research from Cuconasu et al. indicates that related but irrelevant documents are the most harmful to RAG systems. The researchers found "accuracy deteriorating by more than -67% in some cases. Even more importantly, adding just one related document causes a sharp reduction in accuracy, with peaks of -25% … empirical analysis suggests that introducing semantically aligned yet non-relevant documents adds a layer of complexity, potentially misleading LLMs away from the correct response."

Perhaps even more surprisingly, the same researchers discovered that **documents that are both irrelevant and unrelated to the query "are actually helpful in driving up the accuracy of these systems when placed correctly."** Those who build RAG systems should keep an eye on this kind of research but also conduct thorough A/B testing on their own systems to confirm whether the research findings apply to their system.

Thus far, we've been fortunate not to need post-retrieval optimizations in our clients' RAG systems. That said, the following post-retrieval techniques interest us and should be on the radar of all RAG practitioners.

Technique 9: Prioritize search results with reranking

The research by Cuconasu et al. **also demonstrates that positioning the most relevant documents closest to the query in a prompt improves RAG performance**.

Rerank models optimize the priority of chunk search results for a given query. This technique works well when combined with hybrid retrieval systems and query expansion.

Technique 10: Optimize search results with contextual prompt compression

LLMs can process the information in each chunk to filter, reformat, or even compress the final bits of information that make it to the generation prompt.

LLMLingua **is a promising framework for this approach. LLMLingua uses a small language model, such as GPT2-small or LLaMA-7B, to detect and remove unimportant tokens in prompts. It also enables inference with the compressed prompt in black-box LLMs, achieving up to 20x compression with minimal performance loss**. LongLLMLingua takes this further and makes it applicable to RAG systems by considering the input query when conducting compression to remove tokens that are unimportant in general and unimportant to the query in question.

What's remarkable is that in addition to fully understanding and using compressed prompts to answer queries (e.g., as part of RAG), even though the prompts are not human-readable, GPT-4 can also be used to reverse or uncompress inputs.

Technique 11: Score and filter retrieved documents with corrective RAG

Corrective RAG is an approach first introduced by Yan et al. whereby a T5-Large model is trained to identify RAG results as correct/relevant, ambiguous, or incorrect for a given question before providing the results to an LLM for final response generation. RAG results that do not pass the threshold for being classified as correct/relevant or ambiguous are discarded.

Compared to the critique approach used by a fine-tuned Llama-2 7B model with Self-RAG (see Technique 13 below), using a T5-Large model is much more lightweight and can be combined with any large language model.

4. Generation Techniques

Generation optimizations cover improving the LLM call(s) that generate the final user response. The lowest hanging fruit here would be to iterate on the prompt and determine the optimal number of chunks to insert into the generation prompt.

We A/B tested this using 1,000, 3,500, and 7,000 tokens' worth of retrieved context/chunks with GPT-4. We found that inserting 3,500 tokens' worth of retrieved context into the RAG prompt was marginally better than the other options. **We suspect this finding is not universally applicable and that every use case will have a different optimum number.**

At this point, one can consider evaluating and improving an LLM's capabilities for appropriately handling the different kinds of documents it may receive (relevant, related, irrelevant, etc.). Ideally, we create what Yoran et al. call a "retrieval-robust LLM." This means that:

- when relevant, retrieved context should improve model performance
- when irrelevant or even counterfactual, the retrieved context should not hurt model performance

In their article on chain-of-noting (CoN), researchers Wenhao Yu et al. add another characteristic of retrieval-robustness:

"Unknown Robustness: The capacity of a [retrieval-augmented language model] to acknowledge its limitations by responding with 'unknown' when given a query it does not have the corresponding knowledge to answer, and the relevant information is not found within the retrieved documents."

Here are some generation techniques to improve retrieval robustness we've come across but have not yet implemented.

Technique 12: Tune out noise with chain-of-thought prompting

Chain-of-thought (CoT) prompting increases the likelihood that the LLM will arrive at the correct response in the presence of noisy or irrelevant context through reasoning.

Researchers Wenhao Yu et al. took this idea further with chain-of-noting where they fine-tuned a model to generate "sequential reading notes for each retrieved document. This process allows for an in-depth assessment of document relevance to the posed question and aids in synthesizing this information to craft the final answer." The fine-tuned model was LLaMA-7B, and the training data was created using ChatGPT.

Technique 13: Make your system self-reflective with Self-RAG

Self-RAG is another fine-tuning-based approach where a language model is trained to output special reflection tokens during generation. Reflection tokens are either retrieval or critique tokens. Researchers Asai et al. describe their approach in more detail:

> *"Given an input prompt and preceding generations, Self-RAG first determines if augmenting the continued generation with retrieved passages would be helpful. If so, it outputs a retrieval token that calls a retriever model on demand. Subsequently, Self-RAG concurrently processes multiple retrieved passages, evaluating their relevance and then generating corresponding task outputs. It then generates critique tokens to criticize its own output and choose the best one in terms of factuality and overall quality."*

Technique 14: Ignore irrelevant context through fine-tuning

Given that LLMs aren't usually explicitly trained or tuned for RAG, it stands to reason that fine-tuning a model for this use case can conceivably improve a model's ability to ignore irrelevant context. Yoran et al. empirically showed that even 1,000 examples suffice to train the model to be robust to irrelevant contexts while maintaining high performance on examples with relevant ones.

Technique 15: Use natural language inference to make LLMs robust against irrelevant context

Yoran et al. also explored using a natural language inference (NLI) model for irrelevant context identification. **As the researchers point out, there are scenarios where irrelevant RAG context negatively impacts LLM performance.** NLI models can be used to filter out irrelevant context. This technique works by using the retrieved context only when the hypothesis (i.e., the use question and LLM generated answer) is classified as entailed by the premise (i.e., the retrieved context or RAG results).

5. Other Considerations for Advanced RAG Techniques

As this whitepaper shows, there are many techniques available to optimize your RAG system, and undoubtedly more will be discovered as businesses continue pushing the capabilities of their generative AI applications. As you research potential techniques, keep the following considerations in mind.

Input and output guardrails

Mitigating for off-topic or malicious jailbreak inputs and outputs in RAG systems can be difficult, but it's critically important, especially in highly regulated industries such as financial services or healthcare. We believe the best approach is **adopting a defense-in-depth strategy** hardened by rigorous red-teaming. We've written at length about some strategies we've employed in the following posts:

- deploying LLM moderation toward the output of the main RAG chatbot LLM
- using **partial intent classification** not only for regular queries, but also to catch potential jailbreak attempts

Evaluating your RAG system

Having a scalable, automated method of evaluating the quality and accuracy of your RAG system is critical for:

- ensuring your system is improving over time and preventing regressions
- conducting A/B tests (e.g., for prompt changes)

We've written extensively about evaluation strategy, most prominently in our blog on **RAG benchmarks**.

Hallucination rates

AI hallucination is still a significant risk with RAG systems, which means detecting, measuring, and mitigating for hallucinations is another crucial part of building safe and secure RAG systems. Get started by learning **how to detect AI hallucinations**.

Other potential improvements

The literature on RAG is vast and continually expanding. Some topics we've not explored yet but excite us a great deal include:

- Fine-tuning the embeddings model
- Using knowledge graphs (i.e. **GraphRAG**)
- Using long-context LLMs (e.g. Gemini 1.5, or GPT-4 128k) instead of chunking and retrieval

We'll test and report on these and other advanced RAG techniques as use cases permit.

Conclusion: Everything You Need to Build and Fine-Tune Your RAG System

As the landscape of retrieval augmented generation systems rapidly evolves, it presents many opportunities for enhancing conversational AI and other generative AI applications. Our experimentation and research alone highlight the potential advanced RAG techniques to improve:

- information density
- retrieval accuracy
- user response quality

Correctly implemented, these techniques drive greater cost efficiencies for businesses and a better customer experience. But to keep pace with rapidly emerging best practices, software engineers and data scientists need timely, trustworthy resources to turn to.

> Check out the latest in RAG, LLM, and generative AI research on **WillowTree's Data & AI Knowledge Hub**, where we publish all our leading-edge AI content, from research papers to case studies.

WILLOWTREE®

a TELUS International Company

ai@willowtreeapps.com | 1-888-329-9875

willowtreeapps.com