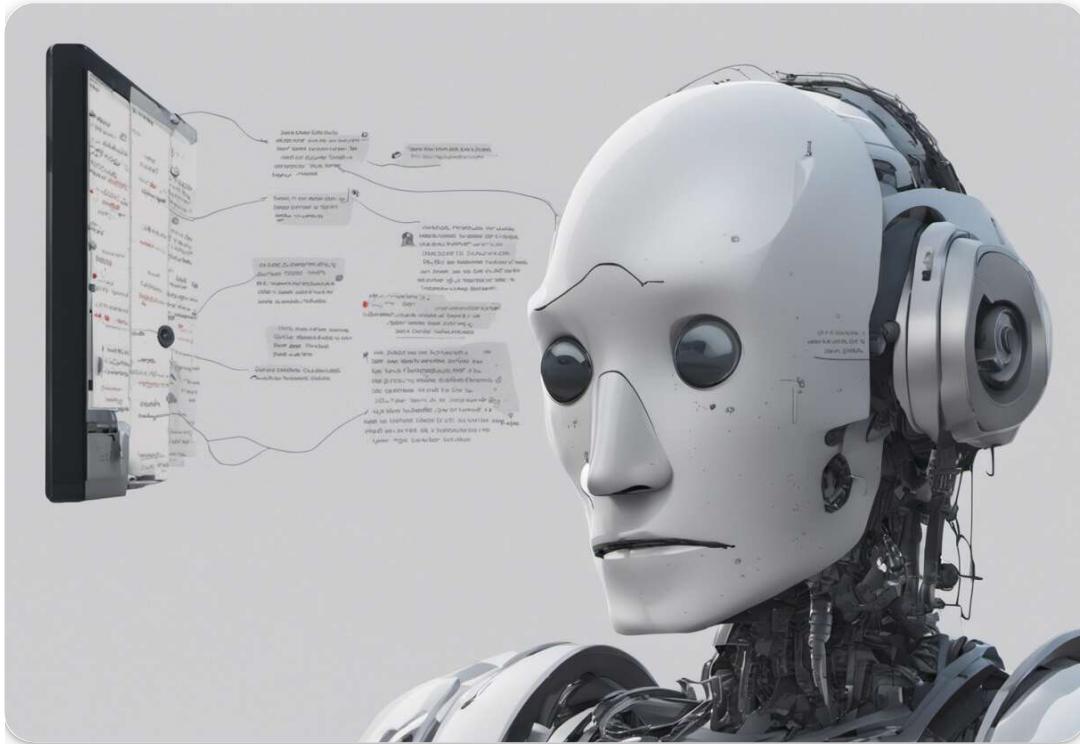


Blog Post



Data ingestion

Retrieval Augmented Generation (RAG) Done Right: Chunking

How to properly chunk your text data to optimize performance of retrieval augmented generation...

September 20, 2023 by Ofer Mendelevitch



Ofer Mendelevitch

Ofer Mendelevitch leads developer relations at Vectara. He has extensive hands-on experience in machine learning, data science and big data systems across multiple industries, and has focused on developing products using large language models since 2019. Prior to Vectara he built and led data science teams at Syntegra, Helix, Lendup, Hortonworks and Yahoo! Ofer holds a B.Sc. in computer science from Technion and M.Sc. in EE from Tel Aviv university, and is the author of "Practical data science with Hadoop" (Addison Wesley).

Share



Grounded Generation (Vectara's original version of Retrieval)

Augmented Generation) remains the most common category of GenAI applications, with use cases like question answering, chatbot, knowledge search, and many others.

As described in the [Grounded Generation reference architecture blog article](#), building such applications on your own requires careful integration of many components and systems, and with that comes many design decisions.

One such design decision is "How to chunk the text data?"

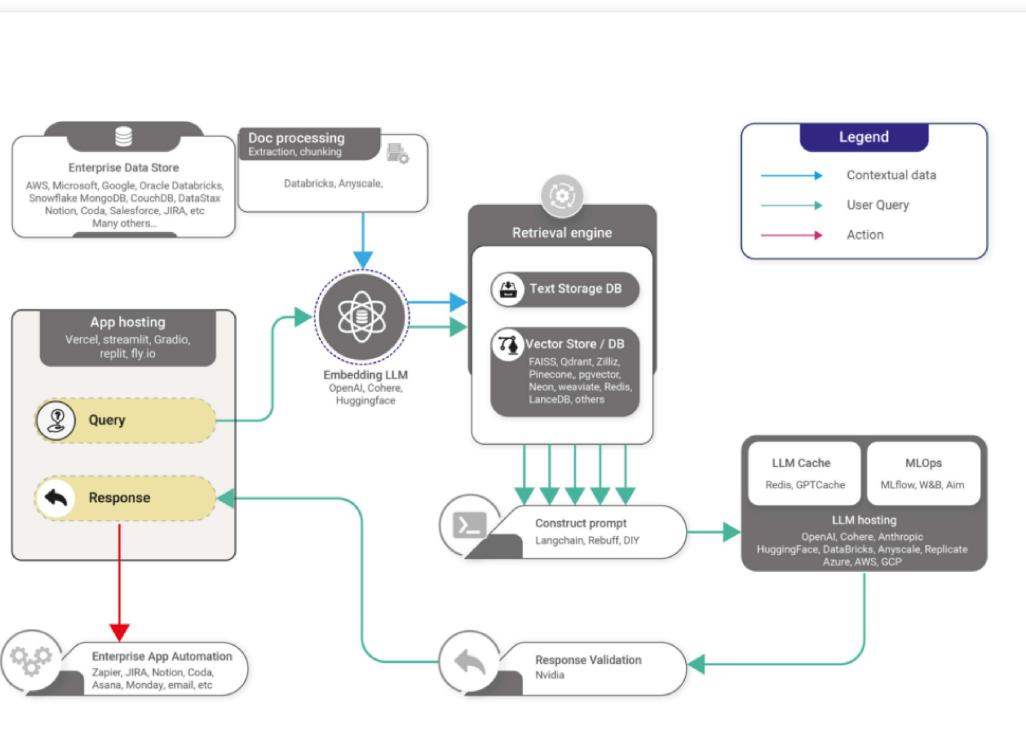


Figure 1: Grounded Generation reference architecture

The box titled "Doc Processing" in Figure 1 depicts the step where documents are pre-processed before the embeddings can be calculated. This involves two main processing steps:

1. Translating the raw input into text. For example, if the input is a PDF file, we must extract the text from the (binary) PDF file.



THIS RAISES AN IMPORTANT DESIGN QUESTION: WHAT CHUNK SIZE IS OPTIMAL?

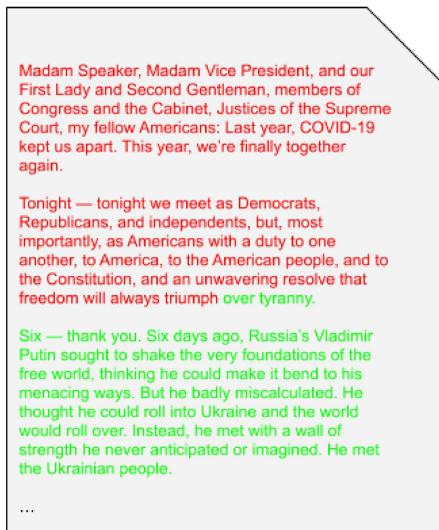
As it turns out, your chunking strategy may be more important than you might think.

Let's dig in.

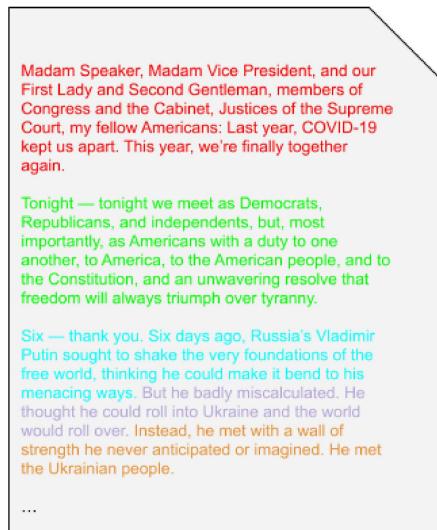
What is Chunking?

In the context of Grounded Generation, chunking is the process of breaking down the input text into smaller segments or chunks. A chunk could be defined simply by its size (i.e., number of characters) or can be determined by analyzing the text to find natural segments like complete sentences or paragraphs.

To demonstrate this, we took a small piece from President Biden's "State of the Union" speech in 2022. Figure 2 shows how fixed-chunking (by size) vs. NLP chunking (by sentences) works.



Fixed Chunking



NLP Chunking



Why does chunking matter so much, you may ask?

We encode each chunk into an embedding vector and use that for retrieval (learn more about [how vector search works](#)). If the chunk is small enough it allows for a more granular match between the user query and the content, whereas larger chunks result in additional noise in the text, reducing the accuracy of the retrieval step.

Chunking with LangChain and LlamaIndex

Most recently the discussion around the importance of selecting a good chunking strategy has received more focus. [Jerry Liu](#) from LlamaIndex commented that a good strategy is to “use smaller chunks for embedding and an expanded window for the LLM,” using LlamaIndex’s `SentenceWindowNodeParser` and the `window` field. Similarly, [Harrison Chase](#) from LangChain presented the `ParentDocumentRetriever` approach that powers a similar capability.

What are some common chunking strategies that are available today?

The simplest approach is to use fixed-size chunking. With [LangChain](#), this can be implemented as follows:

```
from langchain.text_splitter import CharacterTextSplitter

with open('state_of_the_union.txt') as f:
    text = f.read()
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 1000,
    chunk_overlap = 100,
    length_function = len,
    is_separator_regex = False)
docs = text_splitter.create_documents([text])
```



Another strategy is to split recursively by the number of characters until the resulting chunks are small enough. For example:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 100, chunk_overlap = 20, length_function = len,
    is_separator_regex = False)
docs = text_splitter.create_documents([text])
```

For both of these strategies, however, you need to determine the chunk size, and it's often unclear how to do that in a way that results in the best outcome.

One recent addition to LangChain is the [ParentDocumentRetriever](#), which can help with challenges imposed by chucking strategies such as fixed-size or recursive splitting.

When using [ParentDocumentRetriever](#), the chunking is performed with smaller chunks, but the link to the original (parent) document is maintained, so that larger chunks surrounding the matching piece of text are available as input to the LLM summarization step.

LangChain is not the only one providing different chunking strategies, and many options are similarly available with LlamaIndex.

With [LlamaIndex](#), given a list of documents a NodeParser chunks them into segments called "Node objects," such that each node is of a specific size.

```
from llama_index.node_parser import SimpleNodeParser
from llama_index import Document

node_parser = SimpleNodeParser.from_defaults(chunk_size=1000,
                                             chunk_overlap=100)
```



There are additional splitter types like `SentenceSplitter` and `CodeSplitter`, and of course the `SentenceWindowNodeParser` mentioned above. With

[SentenceWindowNodeParser](#)

each document is split into individual sentences, but the resulting nodes maintain the surrounding “window” of sentences around each node in the metadata. This can help achieve the same goal as LangChain’s `ParentDocumentRetriever` – you can replace the sentence with its surrounding context before sending the node content to the LLM. You can see an end-to-end example of using `SentenceWindowNodeParser` [here](#).

Vectara’s approach: Chunking with NLP

Vectara provides a GenAI platform that simplifies the creation and management of Grounded Generation pipelines with a simple API, as you can see in Figure 3:



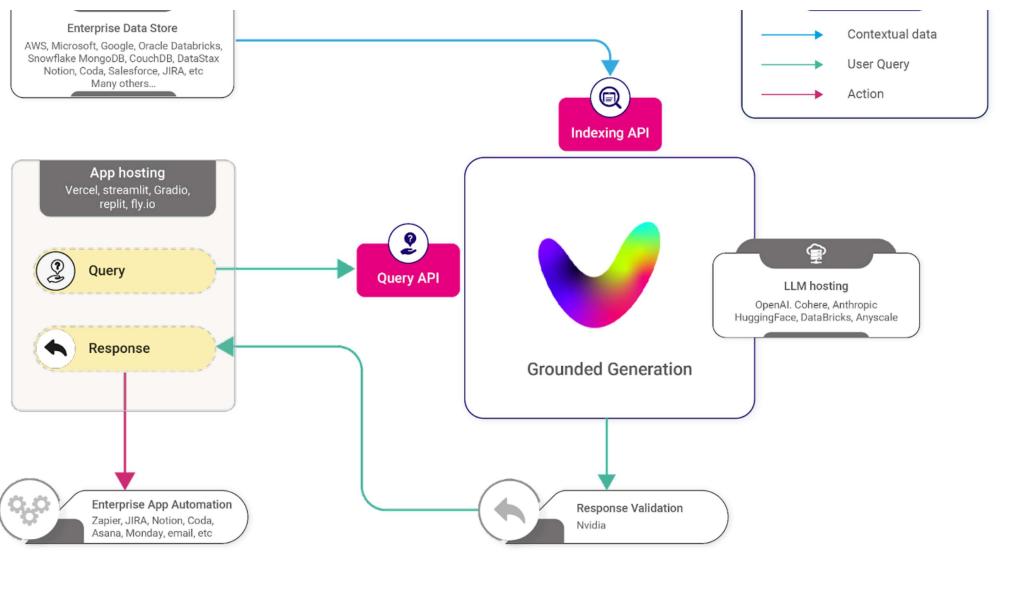


Figure 3: Grounded Generation architecture with Vectara

So how is “chunking” done with Vectara?

The full document text is sent via the indexing API, and chunking is done automatically for you on the backend. Specifically, Vectara uses advanced natural language processing techniques to split documents into small enough chunks that capture a clean signal of semantic meaning. During query time, the user can specify how much context is included in the response around the small chunk via the **sentences_before** and **sentences_after** variables (see query API [docs](#) for more details).

This has been our approach at Vectara since day one, allowing our Grounded Generation flow to achieve better performance in summarization, and it’s great to see the open-source community catch up to the same methodology with LangChain’s `ParentDocumentRetriever` and LLamaIndex’s `SentenceWindowNodeParser`.

Let’s see how this works using Vectara [integration](#) with LangChain: all you have to do is construct the Vectara object:



```
VECTARA = vectara.from_documents(urls, embedding=None, doc_metadata={  
    "speech": "state-of-the-union"})
```

Vectara's `from_documents()` function will treat the "state of the union" text as a single text document and upload it to the Vectara platform, where it will be processed and chunked properly.

Comparing Chunking Strategies

So let's do some experimentation. We take the newly released "Llama2" [paper](#) and compare the performance of various chunking strategies to answer 2 questions:

Question 1: "What is shown in Figure 16?"

Question 2: "Is GPT-4 better than Llama2?"

Our goal here is not to identify which chunking strategy is "best" but rather to demonstrate how various choices of chunking may have a non-trivial impact on the ultimate outcome from the retrieval-augmented-generation (Grounded Generation) pipeline.

The results are shown in Table 1 below, and you can find the full code in this [notebook](#).

Experiment	Figure 16 shows the impact of context distillation and context distillation with answer templates on the safety RM scores. It compares the distribution of safety RM scores from the base model when adding a generic preprompt, and when adding a preprompt based on the risk category with a tailored answer	Is GPT-4 better than Llama2?
		

	<p>preprompt with a tailored answer template helps even more.</p>	
Fixed 1000, overlap 0	I'm sorry, but I don't have access to the specific figures mentioned in the context.	<p>Based on the provided context, there is a significant gap in performance between Llama 2 70B and GPT-4. The context does not provide specific details about the performance of GPT-4, so it is unclear whether GPT-4 is better than Llama 2.</p>
Fixed 1000, overlap 100	I'm sorry, but I don't have access to any figures or images.	<p>According to the provided context, there is a significant gap in performance between Llama 2 70B and GPT-4. Therefore, it can be inferred that GPT-4 is better than Llama 2.</p>
Fixed 2000, overlap 0	I'm sorry, but I don't have any information about Figure 16.	<p>According to the provided context, there is a significant gap in performance between Llama 2 70B and GPT-4. Therefore, it can be inferred that GPT-4 is better than Llama 2.</p>
Fixed 2000, overlap 100	I'm sorry, but I don't have access to any figures or images.	<p>According to the provided context, there is a significant gap in</p>



		inferred that GPT-4 is better than Llama 2 70B.
Recursive 1000, overlap 0	I'm sorry, but I don't have access to any figures or images.	Yes, according to the provided information, there is still a large gap in performance between Llama 2 70B and GPT-4.
Recursive 1000, overlap 100	I'm sorry, but I don't have access to Figure 16.	Yes, according to the provided information, there is still a large gap in performance between Llama 2 70B and GPT-4.
Recursive 2000, overlap 0	I'm sorry, but I don't have access to any figures or images.	Yes, according to the provided information, there is still a large gap in performance between Llama 2 70B and GPT-4.
Recursive 2000, overlap 100	Figure 16 is not described in the given context.	Yes, according to the provided information, there is still a large gap in performance between Llama 2 70B and GPT-4.
Vectara chunking	Figure 16 shows the impact of context distillation and context distillation with answer templates on the safety RM scores. It compares the distribution of safety RM scores from the base model when adding a generic preprompt and when adding a preprompt based on the risk category with a tailored answer	Based on the given context, it is mentioned that there is still a large gap in performance between Llama 2 70B and GPT-4. Therefore, it can be inferred that GPT-4 is considered to be better than Llama 2.



preprompt with a tailored answer

template helps even more.

Table 1: comparison of results with various chunking methods. We try fixed-chunking with chunk sizes of 1000 and 2000, with overlap sizes of 0 or 100. We compare that to Recursive chunking with the same parameters and, lastly to Vectara's chunking.

As you can see, the various chunking strategies (fixed and recursive) resulted in a good response for “Is GPT-4 better than Llama2” in almost all cases. However, when asked “what is shown in figure 16” – only Vectara chunking responded with the right answer, and other chunking strategies failed.

Summary

Retrieval-augmented-generation, also known as Grounded Generation, continues to be a major architectural pattern for most enterprise GenAI applications.

When processing data in a GG flow, splitting the source documents into chunks requires care and expertise to ensure the resulting chunks are small enough to be effective during fact retrieval but not too small so that enough context is provided during summarization.

In this blog post we have seen how different chunking strategies result in different responses to a user query, and it’s often unclear which strategy to choose for a given use case.

Vectara’s NLP-powered chunking strategy, coupled with the ability to include a broader context (more sentences before/after the matching sentence) around matching chunks, provides a robust solution that works well in most applications and seems to be where the open source community (LangChain and LlamalIndex) is also headed.



Search, AI, and Generative AI

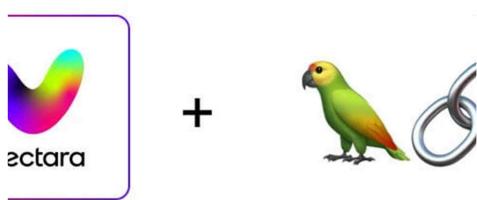
Recommended Content



Research

A Reference Architecture for Grounded Generation

August 8, 2023 by Ofer Mendelevitch & Tallat Shafaat | 7 min Read



Application Development

5 Reasons to Use Vectara's LangChain Integration

June 15, 2023 by Ofer Mendelevitch | 6 min Read



Hallucination

Avoiding hallucinations in LLM-powered Applications

May 2, 2023 by Ofer Mendelevitch | 8 min Read



Vectara: Hybrid Search and Beyond [PDF]

In the AI era, how people interact with information has changed. Users expect relevant answers to questions in natural language, not a shopping list of hit or miss search results. They expect the best semantic or exact matches regardless of typos, colloquialisms, or context. Additionally, it is Vectara's mission to remove language as a barrier by allowing cross-language hybrid search that delivers summarized answers in the language of your choice. The Internet, mobile, and AI have made information accessible, now Vectara helps you find meaning quickly through the most relevant answers. Get to know Vectara, and if you have a question, just ask.

[Get Introduced to Vectara →](#)



[Platform](#)[Solutions](#)[Pricing](#)[Developers](#)[Resources](#)[Get Started Free](#)[Breakthrough
Relevance](#)[API First](#)[Language
Agnostic](#)[Secure and
Reliable](#)[Global Site
Search](#)[Workplace
Search](#)[Developers](#)[Marketers](#)[IT](#)[e-Books and
Guides](#)[Webinars and
Events](#)[Pinecone.io Vs.
Vectara](#)[Algolia Vs.
Vectara](#)[FAQ](#)[Community
Help Center](#)

Trust and Security

[Privacy Policy](#)[Status](#)[Terms](#)

© 2023 Vectara, Inc. All rights reserved.

