

# Building Performant RAG Applications for Production

Prototyping a RAG application is easy, but making it performant, robust, and scalable to a large knowledge corpus is hard.

This guide contains a variety of tips and tricks to improve the performance of your RAG pipeline. We first outline some general techniques - they are loosely ordered in terms of most straightforward to most challenging. We then dive a bit more deeply into each technique, the use cases that it solves, and how to implement it with LlamaIndex!

The end goal is to optimize your retrieval and generation performance to answer more queries over more complex datasets accurately and without hallucinations.

## General Techniques for Building Production-Grade RAG

Here are some top Considerations for Building Production-Grade RAG

- Decoupling chunks used for retrieval vs. chunks used for synthesis
- Structured Retrieval for Larger Document Sets
- Dynamically Retrieve Chunks Depending on your Task
- Optimize context embeddings

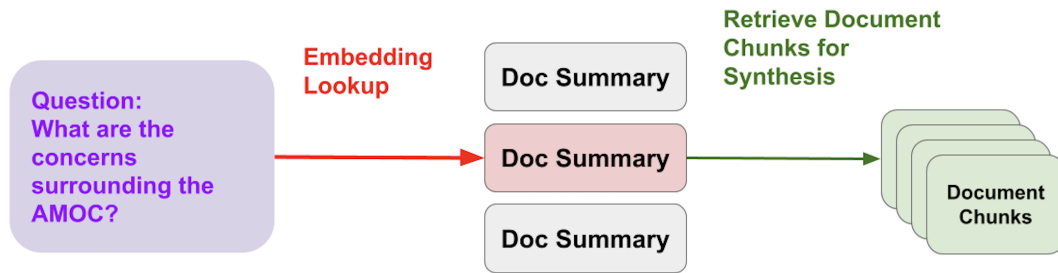
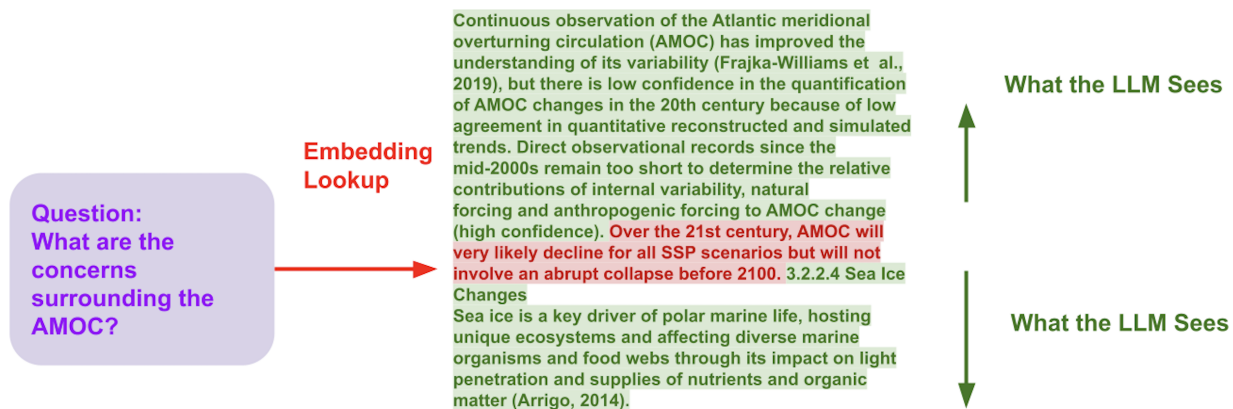
We discussed this and more during our [Production RAG Webinar](#). Check out [this Tweet thread](#) for more synthesized details.

## Decoupling Chunks Used for Retrieval vs. Chunks Used for Synthesis

A key technique for better retrieval is to decouple chunks used for retrieval with those that are used for synthesis.



CTRL + K

**Embed Summary → Link to Additional Documents****Embed Sentence → Link to Expanded Window**

## Motivation

The optimal chunk representation for retrieval might be different than the optimal consideration used for synthesis. For instance, a raw text chunk may contain needed details for the LLM to synthesize a more detailed answer given a query. However, it may contain filler words/info that may bias the embedding representation, or it may lack global context and not be retrieved at all when a relevant query comes in.

## Key Techniques

There's two main ways to take advantage of this idea:

### 1. Embed a document summary, which links to chunks associated with the document.

This can help retrieve relevant documents at a high-level before retrieving chunks vs. retrieving chunks directly (that might be in irrelevant documents).

Resources:

- [Recursive Retriever + Query Engine Demo](#)
- [Document Summary Index](#)



### 2. Embed a sentence, which then links to a window around the sentence.

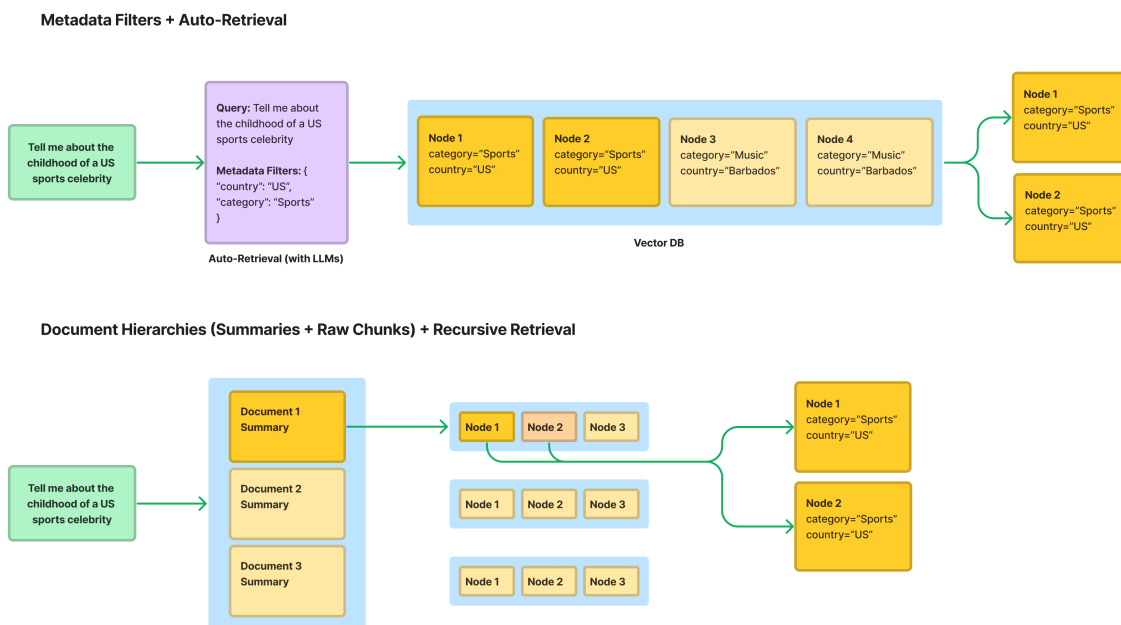
CTRL + K

This allows for finer-grained retrieval of relevant context (embedding giant chunks leads to “lost in the middle” problems), but also ensures enough context for LLM synthesis.

Resources:

- [Metadata Replacement + Node Sentence Window](#)

## Structured Retrieval for Larger Document Sets



## Motivation



A big issue with the standard RAG stack (top-k retrieval + basic text splitting) is that it doesn't do well as the number of documents scales up - e.g. if you have 100 different PDFs. In this setting, given a query you may want to use structured information to help with more precise retrieval; for instance, if you ask a question that's only relevant to two PDFs, using structured information to ensure those two PDFs get returned beyond raw embedding similarity with chunks.

## Key Techniques



There's a few ways of performing more structured tagging/retrieval for production RAG systems, each with their own pros/cons.

1. **Metadata Filters + Auto Retrieval** Tag each document with metadata and the CTRL + K a vector database. During inference time, use the LLM to infer the right metadata

query the vector db in addition to the semantic query string.

- Pros : Supported in major vector dbs. Can filter document via multiple dimensions.
- Cons : Can be hard to define the right tags. Tags may not contain enough relevant information for more precise retrieval. Also tags represent keyword search at the document-level, doesn't allow for semantic lookups.

Resources: **2. Store Document Hierarchies (summaries -> raw chunks) + Recursive Retrieval** Embed document summaries and map to chunks per document. Fetch at the document-level first before chunk level.

- Pros : Allows for semantic lookups at the document level.
- Cons : Doesn't allow for keyword lookups by structured tags (can be more precise than semantic search). Also autogenerating summaries can be expensive.

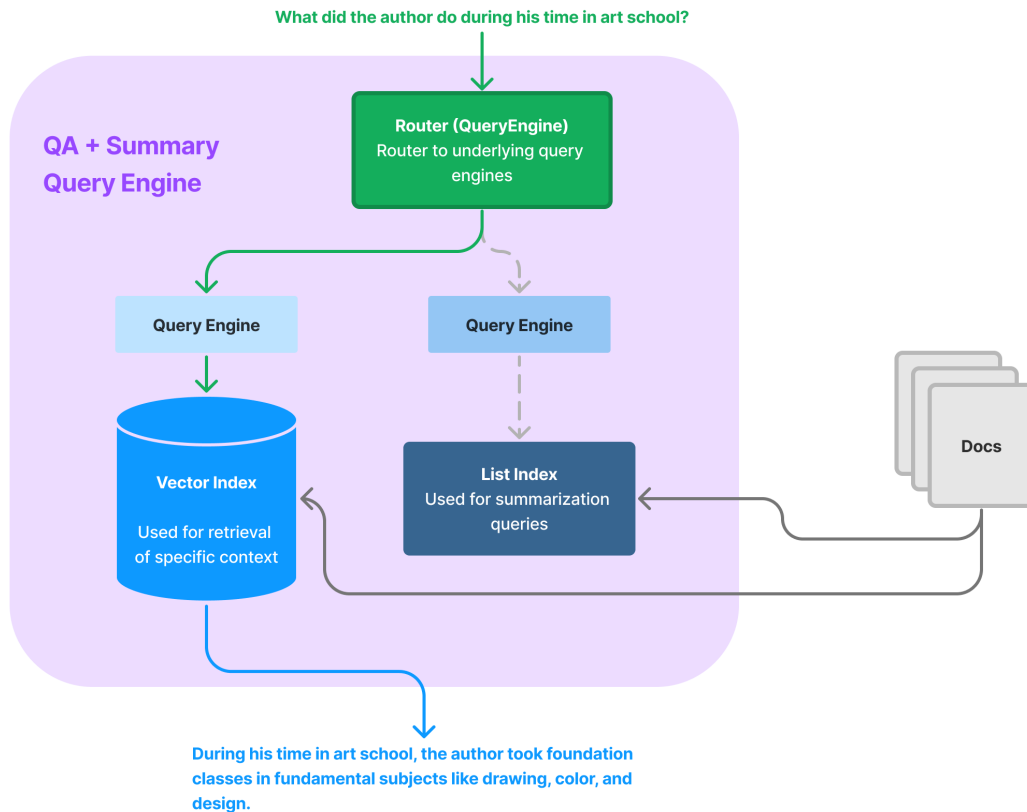
## Resources

- [Auto-Retrieval from a Vector Database](#)
- [Document Summary Index](#)
- [Recursive Retriever + Document Agents](#)
- [Comparing Methods for Structured Retrieval \(Auto-Retrieval vs. Recursive Retrieval\)](#)



CTRL + K

# Dynamically Retrieve Chunks Depending on your Task



## Motivation

RAG isn't just about question-answering about specific facts, which top-k similarity is optimized for. There can be a broad range of queries that a user might ask. Queries that are handled by naive RAG stacks include ones that ask about specific facts e.g. "Tell me about the D&I initiatives for this company in 2023" or "What did the narrator do during his time at Google". But queries can also include summarization e.g. "Can you give me a high-level overview of this document", or comparisons "Can you compare/contrast X and Y". All of these use cases may require different retrieval techniques.

## Key Techniques

LlamaIndex provides some core abstractions to help you do task-specific queries. This includes our [router](#) module as well as our [data agent](#) module. This also includes



CTRL + K

advanced query engine modules. This also include other modules that join structured and unstructured data.

You can use these modules to do joint question-answering and summarization, or even combine structured queries with unstructured queries.

### Core Module Resources

- [Query engine](#)
- [Agents](#)
- [Router](#)

### Detailed Guide Resources

- [Sub Question Query Engine](#)
- [Joint QA Summary Query Engine](#)
- [Recursive Retriever + Document Agents](#)
- [Router Query Engine](#)
- [OpenAI Agent + Query Engine Experimental Cookbook](#)
- [OpenAI Agent Query Planning](#)

# Optimize Context Embeddings

## Motivation

This is related to the motivation described above in “decoupling chunks used for retrieval vs. synthesis”. We want to make sure that the embeddings are optimized for better retrieval over your specific data corpus. Pre-trained models may not capture the salient properties of the data relevant to your use case.

## Key Techniques

Beyond some of the techniques listed above, we can also try finetuning the embedding model. We can actually do this over an unstructured text corpus, in a label-free way.

Check out our guides here:

- [Embedding Fine-tuning Guide](#)