Advanced Retrieval Augmented Generation (RAG) Techniques

# Objective:

The aim of this document is to provide a one stop shop for questions that revolve around implementation of production ready RAG pipelines. It focuses on key considerations, optimization and enhancements required to build a robust advanced RAG pipeline.

# Scope:

The naive RAG models are the basic version of RAG that we can build. However, moving from POC to production demands fairly more structured / reusable and optimized RAG models with a scope for continuous advancements and a supportable framework. We will be focusing on:

1. Factors to consider when choosing the components that lay the foundation of a RAG pipeline
2. Advantages and Limitations of open source RAG components and tools, naïve RAG models
3. Advanced RAG approaches pre/ post retrieval enhancements that can help build a robust framework,
4. RAG evaluation metrics
5. Touch upon modular RAG concepts and Agentic AI

# Choosing LLMs and Embeddings

## Factors:

1. Identify your business objectives and use cases
   1. Understand your business's present and future objectives and ensure your chosen LLM can fulfill those needs.
   2. Determine the LLM's specific use case, such as content generation, sentiment analysis, customer support, or fraud detection.
2. Evaluate popular LLMs and their capabilities
   1. Each model has unique strengths and weaknesses.
   2. Familiarize yourself with as many LLMs as you possibly can. Test them for accuracy, relevance and quality of generated text. Even the best LLMs may sometimes provide inaccurate information, so aim to find one that does so less frequently.
   3. Understand each model's strengths and weaknesses, such as advanced coding, complex reasoning, commonsense reasoning, or efficiency in building AI assistants.
   4. Consider the input size the LLM can handle. Larger context windows allow more input and context to be passed to the model.
3. License
   1. Different providers may have varying restrictions on how their models can be used.
   2. Some licenses may limit the use of the LLM or embeddings to non-commercial applications, require attribution, or restrict the types of applications for which the model can be used.
   3. Make sure you inspect the license and ensure that your project complies with the terms.
4. Consider language support and multilingual capabilities
   1. If your project requires support for multiple languages, ensure the LLM and embedding provider you choose can handle the languages you need.
   2. Evaluate the quality of the model's output in each language and its ability to understand and generate text in a culturally appropriate manner.
5. Evaluate security and privacy compliance
   1. Assess the security measures in place to protect your data and the privacy of your users.
   2. Look for features like data encryption, secure data handling, and compliance with relevant privacy regulations (e.g., GDPR, HIPAA).
6. Assess cost and performance
   1. Understand the cost structure of each LLM, including any usage-based pricing.
   2. Evaluate the performance of different models in terms of accuracy, speed, and response times.
7. Ensure long-term relevance and support
   1. Investigate the LLM provider's future plans and how regularly the model receives updates.
   2. Consider the vendor's reputation, the LLM's ability to seamlessly integrate with your existing infrastructure, and its ability to scale to address your business's growing demands.
8. Assess community support and resources
   1. Look for an active community of developers and users around the LLM and embedding provider you're considering.
   2. A strong community can provide valuable resources, such as tutorials, forums, and open-source projects, which can help you get started and troubleshoot issues more quickly.

## LLM and Embeddings Provider

You can choose from companies that build and serve their own LLMs, like:

1. [OpenAI] - (https://platform.openai.com/docs/models)
2. [Anthropic] - (https://docs.anthropic.com/claude/docs/models-overview)
3. [Cohere] - (https://docs.cohere.com/docs/the-cohere-platform)
4. [Mistral] - (https://docs.mistral.ai/platform/pricing/)
5. [Google Gemini] - (https://ai.google.dev/)

Or you can choose from companies that host and serve open-source models via an API, like:

1. [Fireworks AI] - (https://fireworks.ai/models)
2. [Together AI] - (https://www.together.ai/pricing)
3. [Predibase] - (https://docs.predibase.com/user-guide/inference/models)
4. [Hugging Face] - (https://huggingface.co/docs/text-generation-inference/en/supported\_models)
5. [Basten] - (https://www.baseten.co/library/)
6. [Replicate] - (https://replicate.com/collections/language-models)
7. [Lepton AI] - (https://www.lepton.ai/docs)
8. [Clarifai] - (https://clarifai.com/explore/models)

# Choosing Vector Database

Without a doubt, vector databases are an essential part of any RAG system.

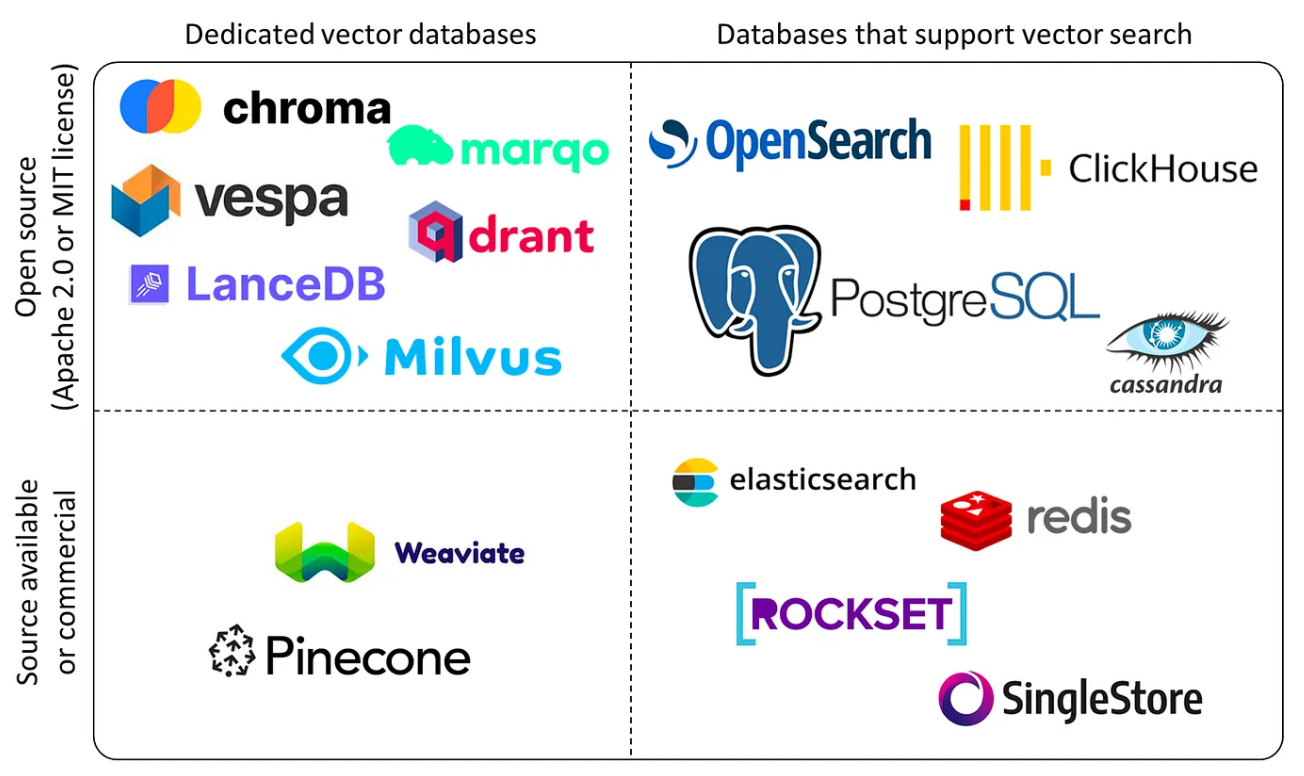
## Factors:

1. Similarity search performance
   1. RAG relies heavily on efficient similarity search to retrieve relevant documents or passages. The vector database should provide fast and accurate similarity search capabilities, such as cosine similarity or Euclidean distance, to quickly retrieve relevant information.
2. Compatibility with datastore
3. Scalability
   1. As the amount of data grows, the vector database should be able to scale horizontally and handle large-scale indexing and querying. It should efficiently store and manage high-dimensional vectors and support distributed search across multiple nodes if necessary.
4. Integration with LLM frameworks
   1. The vector database should integrate well with popular LLM orchestration frameworks like LlamaInde, LangChain, or Instructor. This integration allows seamless interaction between the RAG model and the vector database, enabling efficient retrieval and generation.
5. Support for various data types
   1. RAG applications may deal with different data types, such as text, images, or audio. The vector database should support storing and indexing vectors derived from various data types, allowing flexibility in the data types that can be retrieved.
6. Indexing and updating capabilities
   1. The vector database should provide efficient indexing mechanisms to quickly build and update the index as new data is added or modified. It should handle incremental updates and support real-time indexing if required by the application.
7. Retrieval flexibility
   1. The vector database should offer flexibility in retrieval options, such as specifying the number of nearest neighbors to retrieve, setting similarity thresholds, or applying filters based on metadata. This flexibility allows fine-tuning the retrieval process based on the specific requirements of the RAG application.
8. Data persistence and reliability
   1. The vector database should ensure data persistence and provide data backup and recovery mechanisms. It should be reliable and able to handle potential failures or data loss scenarios.
9. Community support and documentation
   1. Consider the level of community support and documentation available for the vector database. An active community and comprehensive documentation can greatly assist in troubleshooting, optimizing performance, and staying updated with the latest features and best practices.
10. Ease of use and deployment
    1. The vector database should be easy to set up, configure, and deploy. It should provide clear APIs or client libraries for integration with the RAG application and have straightforward deployment options, whether on-premises or in the cloud.
11. Cost and licensing
    1. Consider the vector database's cost and licensing model. Consider pricing, scalability costs, and any limitations or restrictions imposed by the licensing terms.

## Recommendation:

Considering the above reasons, QDrant seems to check all these boxes they also have a way to store data in more than one form – Graph, vectors etc which helps in multiple use cases.

1. Their documentation is a breath of fresh air – clear, to the point, and without any fluff. It makes getting up and running a breeze if you want to go deeper than the LlamaIndex abstractions.
2. The development is open, and the team behind Qdrant is technically savvy. It's reassuring to see the level of transparency and expertise.
3. They've recently added built-in authentication to the dev version. It's a game changer if you're looking for that extra layer of security.
4. They offer an extremely generous free tier via their hosted cloud, making it easy to test drive Qdrant and see if it fits your needs without any commitment.
5. OpenAI is rumoured to use Qdrant as an embedding vector database, according to a [Reddit post](<https://www.reddit.com/r/ChatGPT/comments/17plmqj/openai_is_using_qdrant_as_a_vector_database/>).
6. X AI also uses Qdrant, as [evidenced by the fork in their GitHub](<https://github.com/xai-org>).



# RAG Overview

## Components and purpose

1. LLM - This is the system's brain, which is responsible for taking the augmented prompt and generating text from it.
2. Prompt - Every interaction starts with a user's query or statement. The Prompt captures this initial input, setting the stage for the retrieval and generation processes.
3. Document Loader - The Document Loader imports and reads documents, preparing them for chunking and embedding.
4. Document Chunker - A document chunker breaks documents into smaller, more digestible pieces to make the data more manageable and efficient for retrieval.
5. Embedding Model - Before storing or retrieving data, we need to convert textual information into a format the system can understand. The Embedder takes on this role, transforming text into vector representations.
6. Vector Store - This storage system houses embeddings from their corresponding textual data.
7. Vector Store Retriever - The Vector Store Retriever fetches relevant documents from the vector store by comparing vector similarities, ensuring that the most pertinent information is always available.
8. User Input - Last but not least, the User Input tool captures the query or statement provided by the end user, initiating the entire RAG process.

## Subsystems

Each component fits within one of the following subsystems:

1. Index
2. Retrieval
3. Augment

These work together as an orchestrated flow, transforming a user's query into a contextually rich and accurate response.

INDEX System

This subsystem prepares and organizes the data for efficient retrieval. Here are the steps of the Index system

1. Load Documents (Document loader): This process imports and reads the data the system will use.
2. Chunk Documents (Document chunker): This step breaks down the loaded documents into smaller, more manageable chunks for more efficient retrieval.
3. Embed Documents (embedding model): The text chunks are converted into vector representations, making them searchable within the system.
4. Store Embeddings (Vector Store): Stores the vector embeddings for future retrieval.

RETRIEVAL System

This subsystem fetches the most relevant information based on the user's query. Here are the steps in the Retrieval system

1. Obtain User Query (User Input): Captures the user's question or statement.
2. Embed User Query (Embedding model): Transforms the user's query into a vector format, similar to the indexed documents.
3. Vector Search (Vector Store Retriever): Search the Vector Store for documents with embeddings that closely match the embedded user query.
4. Return Relevant Documents: The system returns the top matches, ensuring that the most pertinent information is always provided.

AUGMENT System

This subsystem enhances the LLM's input prompt with the retrieved context, ensuring the model has all the necessary information to generate a comprehensive response.

1. Create Initial Prompt (Prompt): Starts with the original user query or statement.
2. Augment Prompt with Retrieved Context: Merges the initial prompt with the context retrieved from the Vector Store, creating an enriched input for the LLM.
3. Send Augmented Prompt to LLM: The enhanced prompt is fed to the LLM.
4. Receive LLM's Response: After processing the augmented prompt, it generates and returns its response.

These subsystems make up the whole RAG system, which will produce more accurate, credible, and contextually relevant outputs.

## Ingestion Pipeline

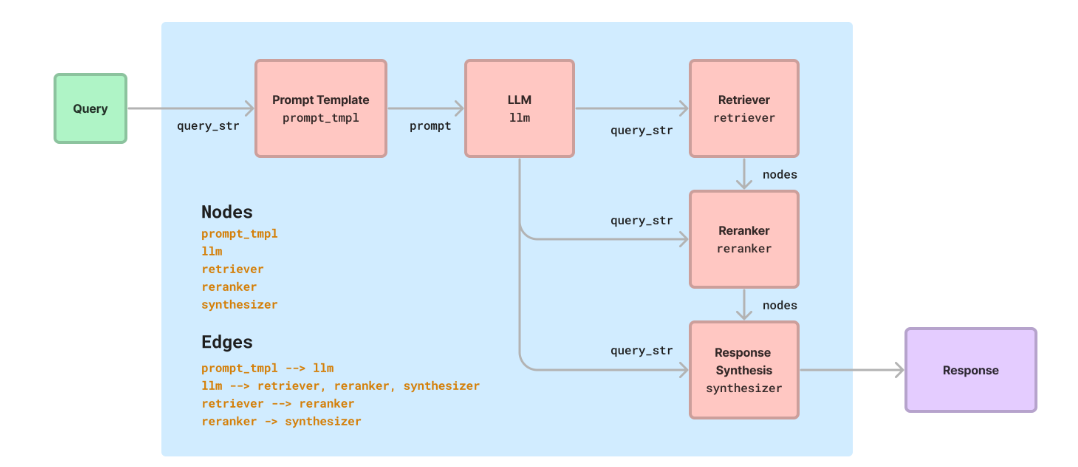
Llama Index offers ingestion pipeline which can be very useful to make the ingestion process robust and streamline.

1. Utilizes `Transformations` applied to input data, modifying data into nodes, which are returned or inserted to a vector database.
2. Caching Mechanism : Each node+transformation pair is cached, enhancing efficiency for identical subsequent operations by utilizing cached results.

## Query Pipeline

LlamaIndex offers a query API for chaining modules to manage data workflows easily. It revolves around the QueryPipeline, where you link various modules like LLMs, prompts, and retrievers in a sequence or DAG for end-to-end execution.

You can streamline workflows efficiently using QueryPipeline, reducing code complexity and enhancing readability. Additionally, a declarative interface ensures easy serialization of pipeline components for portability and deployment across systems in the future.



## Prompt template and Response modes

Llama has various prompt template and a default template that can be used. However, it is very important to tweak the default template for better and a custom response structure.

In LlamaIndex, [response modes] (https://docs.llamaindex.ai/en/stable/module\_guides/deploying/query\_engine/response\_modes/) are used to determine how the system should process and return the results of a query.  Each response mode is designed to handle different types of queries and use cases, providing flexibility and customization in how you interact with your data.

1. Refine
   1. Refine is an iterative method to generate a response.
   2. Initially, we use the context in the first node and the query to create a basic answer. Then, we refine this answer by inputting it, along with the query and context of the second node, into a "refine prompt" to generate an improved answer.
   3. This refinement process continues through N-1 nodes, with N being the total number of nodes. It makes a separate LLM call per Node/retrieved chunk. This mode is good for generating more detailed answers.
2. Compact
   1. Compact and refine mode first combine text chunks into larger consolidated chunks that more fully utilize the available context window, then refine answers across them. This mode is faster than refine since we make fewer calls to the LLM.
   2. This mode is useful when you want to reduce the number of LLM calls while still refining the answer.
3. Simple summarize
   1. Merge all text chunks into one and make a large language model call. The call will fail if the merged text chunk exceeds the context window size.
   2. It's good for quick summarization purposes, but may lose detail due to truncation.
4. Tree summarize
   1. Construct a tree index for the candidate nodes in a bottom-up manner then use a summary prompt based on the query. Return the root node as the final response. When this mode is set, the system is instructed to iterate through many, if not all, documents in order to synthesize an answer, which can lead to better summarization results.
   2. This mode is particularly useful for summarization queries, where the goal is to provide a comprehensive summary of a collection of text or a specific topic.
5. Accumulate
   1. This mode applies the query to each text chunk while accumulating the responses into an array. It returns a concatenated string of all responses.
   2. This mode is good for when you need to run the same query separately against each text chunk.
6. Compact accumulate
   1. In the compact and accumulate mode, text chunks are combined into larger chunks to utilize the context window better. Answers are then accumulated for each chunk and returned as a concatenation. This mode is faster than accumulate as it reduces calls to the LLM.

## Data preparation

Your RAG system is only as good as the data you retrieve. That's why data preparation and cleaning are important steps to ensure high-quality results. Here are some considerations for data prep.

1. Document Content : Utilize text from documents for keyword searches or to find similar content in RAG applications.
2. Document Elements : Break down documents into fundamental parts to assist in RAG tasks like filtering and segmenting, like:
   1. Titles
   2. Narrative text
   3. List items
   4. Tables
   5. Images
3. Element Metadata : Provide additional details for each document element to support hybrid search and track information origin, such as:
   1. Filename
   2. Filetype
   3. Page number
   4. Section
4. Summary : Explains document preprocessing for retrieval systems, focusing on transforming documents into searchable elements and metadata.

**Challenges with Complex PDFs and Documents**

1. Formatting inconsistencies : PDFs and other documents can have varying layouts, fonts, and styles, making it difficult to extract text consistently.
2. Images and graphics : Documents may contain images, charts, and other visual elements that need to be handled separately or extracted using Optical Character Recognition (OCR) techniques.
3. Tables and structured data : Extracting information from tables and structured data within documents can be challenging and may require specialized tools or techniques.
4. Metadata and noise : Documents may include metadata, headers, footers, and other noise that needs to be handled before processing.

While this course won't cover these complex scenarios in depth, it's essential to understand the potential challenges and the need for more advanced data preparation and cleaning techniques when working with diverse document types.

**Options for parsing complex pdfs**

**General PDFs**

1. LlamaParse <https://docs.llamaindex.ai/en/stable/module_guides/loading/connector/llama_parse/> - LlamaParse is an API created by LlamaIndex to efficiently parse and represent files for efficient retrieval and context augmentation using LlamaIndex frameworks.
2. pdfminer.six <https://pdfminersix.readthedocs.io/en/latest/> - A tool for extracting information from PDF documents. It focuses on getting and analyzing text data.
3. pdfplumber <https://github.com/jsvine/pdfplumber> - Gives you detailed information about each text character, rectangle, and line. Plus: Table extraction and visual debugging.
4. Pypdf <https://pypdf.readthedocs.io/en/latest/> - Capable of splitting, merging, cropping, and transforming the pages of PDF files.
5. PyMuPDF <https://pymupdf.readthedocs.io/en/latest/> - A high-performance Python library for data extraction, analysis, conversion & manipulation of PDF (and other) documents.
6. Camelot <https://camelot-py.readthedocs.io/en/master/> - This library is specifically for extracting data from tables in PDFs. This repo also has a [nice comparison](https://github.com/camelot-dev/camelot/wiki/Comparison-with-other-PDF-Table-Extraction-libraries-and-tools) of other table extraction libraries.
7. LLMSherpa <https://github.com/nlmatics/llmsherpa> - The main class here is the `LayoutPDFReader`, and a good read about the problem and proposed solution is [here](<https://ambikasukla.substack.com/p/efficient-rag-with-document-layout>)
8. Unstructured <https://github.com/Unstructured-IO/unstructured> - This has components for ingesting and pre-processing images and text documents, such as PDFs, HTML, Word docs, and many more.
9. Table Transformer <https://github.com/microsoft/table-transformer> - A deep learning model for extracting tables from unstructured documents (PDFs and images)
10. Layout Parser <https://github.com/Layout-Parser/layout-parser> - This is a unified toolkit for deep learning based document image analysis which has a rich repository of deep learning models for layout detection, as well as a set of unified APIs for using them.
11. Marker <https://github.com/VikParuchuri/marker> - Converts PDF to markdown quickly with high accuracy.
12. surya <https://github.com/VikParuchuri/surya> - A document OCR toolkit for accurate OCR in 90+ languages, line-level text detection in any language, layout analysis (table, image, header, etc detection) in any language.

# Drawbacks of Naive RAG

While the naive RAG workflow provides a foundation for enhancing LLMs with external knowledge, it also presents several challenges:

1. Indexing Issues
   1. Incomplete or inaccurate information extraction during the indexing process.
   2. Suboptimal chunking of data, leading to inefficient retrieval.
   3. Inefficient indexing techniques that hinder quick and accurate retrieval.
   4. Poor semantic representation of the indexed data, limiting understanding of its meaning and context.
2. Retrieval Difficulties
   1. Low Precision: Not all retrieved data matches the query, leading to potential errors and inaccuracies.
   2. Low Recall: Failure to retrieve all relevant data limits the context available for response generation.

Users query effectiveness in fetching the most appropriate information.

Retrieval of redundant or overlapping information, leading to inefficient use of computational resources.

1. Generation Problems
   1. Hallucination: The model may generate fictitious responses not grounded in retrieved data, or poor merging of retrieved data can result in incoherent or disjointed outputs.
      1. Hard to ascertain the relative relevance of retrieved context for the generation task.
      2. Risk of overly relying on external information, or merely echo the retrieved data without adding new insights, leading to responses that lack originality or fail to capture the nuances of the user's query.
      3. Potential for generating biased, inconsistent, or irrelevant responses due to the limitations of the retrieved data or the generation process itself.

These challenges must be addressed to build sophisticated and reliable RAG systems that can effectively leverage external knowledge to enhance the performance of language models.

# RAG Evaluation

Core Metrics for RAG Evaluation:

1. Faithfulness:
   1. Ensures answers generated by the model remain true to the given context
   2. Answers must be consistent with the context information and not deviate or contradict it
   3. Crucial for addressing illusions in large models
2. Answer Relevance:
   1. Generated answers need to be directly related to the posed question
   2. Ensures the model stays on topic and provides pertinent information
3. Context Relevance:
   1. Retrieved contextual information should be accurate and targeted
   2. Avoids irrelevant content that can reduce the efficiency of LLMs in utilizing context
   3. Processing long texts is costly for LLMs, so minimizing irrelevant information is essential
4. Mean Reciprocal Rank (MRR):
   1. Statistic measure for evaluating processes that produce a list of possible responses to a sample of queries, ordered by probability of correctness
   2. Reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer
   3. MRR is the average of the reciprocal ranks of results for a sample of queries

LLM-as-a-Judge:

Faithfulness, context relevancy, and answer relevancy use the [LLM-as-a-Judge](<https://arxiv.org/abs/2306.05685>) paradigm for evaluation.

LLM-as-a-Judge is a technique that uses an LLM to automatically evaluate the outputs of other LLMs in a way that aims to approximate human judgment. The key idea is to leverage strong LLMs to assess generated text that usually requires human evaluation, such as the abovementioned metrics. Using LLMs as judges is a scalable and cost-effective way to evaluate RAG systems. It enables faster benchmarking and iteration cycles during AI development.

However, LLM judges have some limitations and biases that need to be carefully addressed, such as:

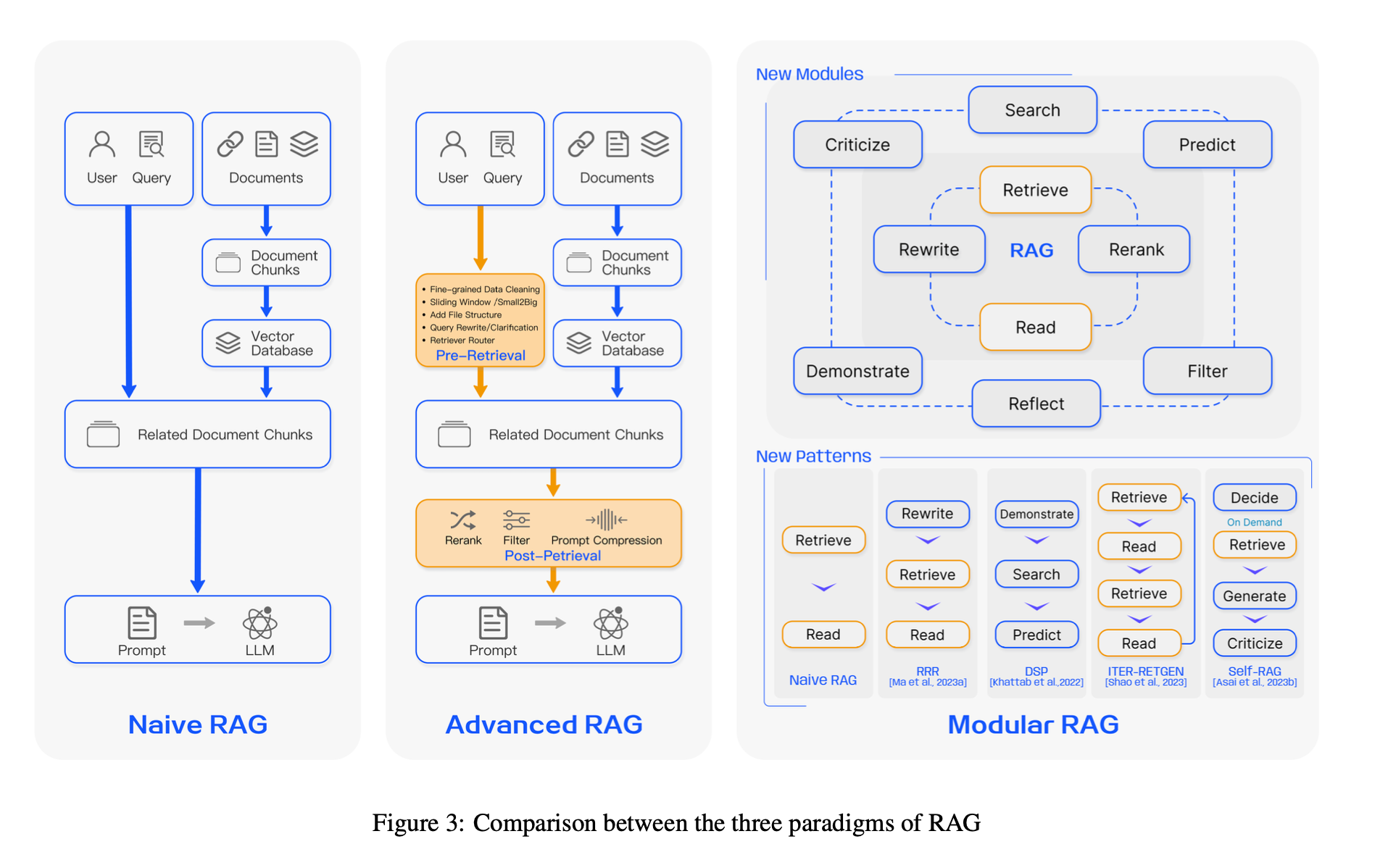
- Position bias: Favoring certain positions in a text

- Verbosity bias: Preferring longer outputs

- Limited reasoning capabilities for complex topics like math

Despite these challenges, studies have shown that with proper prompt engineering and debiasing techniques, strong LLM judges like GPT-4 can achieve high agreement (>80%) with human preferences and expert ratings across different benchmarks. LLM-as-a-Judge is a promising complement to human evaluation for assessing open-ended AI tasks at scale.

# Advanced RAG



## RAG Paradigm and Key Techniques

Naive RAG

1. Indexing: Data cleansing, extraction, chunking, and vectorization
2. Retrieval: Similarity comparison between query and indexed chunks
3. Generation: Synthesizing query and retrieved documents into a prompt for LLM response

Advanced RAG

Pre-Retrieval Process:

1. Optimizing data indexing (enhancing granularity, structure, metadata, alignment, mixed retrieval
2. Fine-tuning embedding models for domain-specific relevance
3. Using dynamic embeddings

OPTIMIZATION MEHODS:

1. **Enhanced Data Granularity** - Improves text standardization and factual accuracy, removes ambiguities, and incorporates domain-specific annotations.
2. **Optimize Index Structures** - Utilizes varied chunk sizes and multiple index paths, and introduces graph structures for better accuracy.
3. **Metadata Utilization** - Incorporates metadata like dates and chapters into chunks to improve filtering and relevance.
4. **Alignment Optimization** - Aligns hypothetical questions with document content to reduce discrepancies.
5. **Mixed Retrieval Techniques** - Combine keyword, semantic, and vector searches to ensure relevant and rich information retrieval.

Post-Retrieval Process:

1. Re-ranking retrieved information (diversity ranking, alternating best document placement, recalculating semantic similarity
2. Prompt compression (importance estimation, granular compressors, summarization)

OPTIMIZATION MEHODS:

1. **ReRank Strategies** - Prioritizes the most relevant documents using advanced frameworks and re-ranking methods.
2. **Prompt Compression** - Reduces noise by compressing irrelevant context and highlighting crucial information, improving focus and response quality.

Modular RAG

1. Integrating new modules (e.g., search module for specific scenarios, memory module for retrieval guidance)
2. Allowing serialized pipeline or end-to-end training across modules
3. Inheriting and building upon Advanced RAG techniques

## Pre-retrieval Optimization

### Optimizing Chunk Size

When documents are ingested into an index, `LlamaIndex` splits them into smaller pieces called "chunks." This process is known as chunking. By default, LlamaIndex uses a \*chunk size\* of 1024 and a \*chunk overlap\* of 20.

**Chunk size:**

The chunk size determines the maximum number of tokens (roughly equivalent to words) that each chunk will contain. With a default chunk size of 1024, `LlamaIndex` will split your documents into chunks that are no longer than 1024 tokens each.

Smaller Chunk Size

1. More precise and focused embeddings
2. Beneficial for retrieving specific information

Larger Chunk Size

1. More general embeddings with broader context
2. Useful for document overviews, but may miss details

Chunk Overlap

1. Shared tokens between adjacent chunks (default: 20)
2. Maintains context and prevents information loss

Relevance and Granularity

1. Smaller chunks (e.g., 128) offer granularity but risk missing vital information, or lack sufficient context.
2. Larger chunks (e.g., 512) are more likely to capture necessary context, but also run the risk of including irrelevant information.
3. Faithfulness and Relevancy metrics help assess response quality.

Chunk Size and Use Case

1. Question Answering: Shorter, specific chunks for precise answers.
2. Summarization: Longer chunks to capture the overall context.

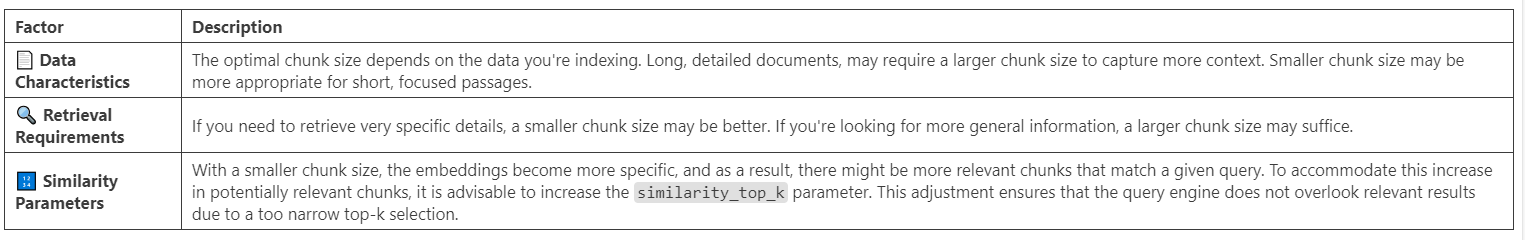
Response Generation Time

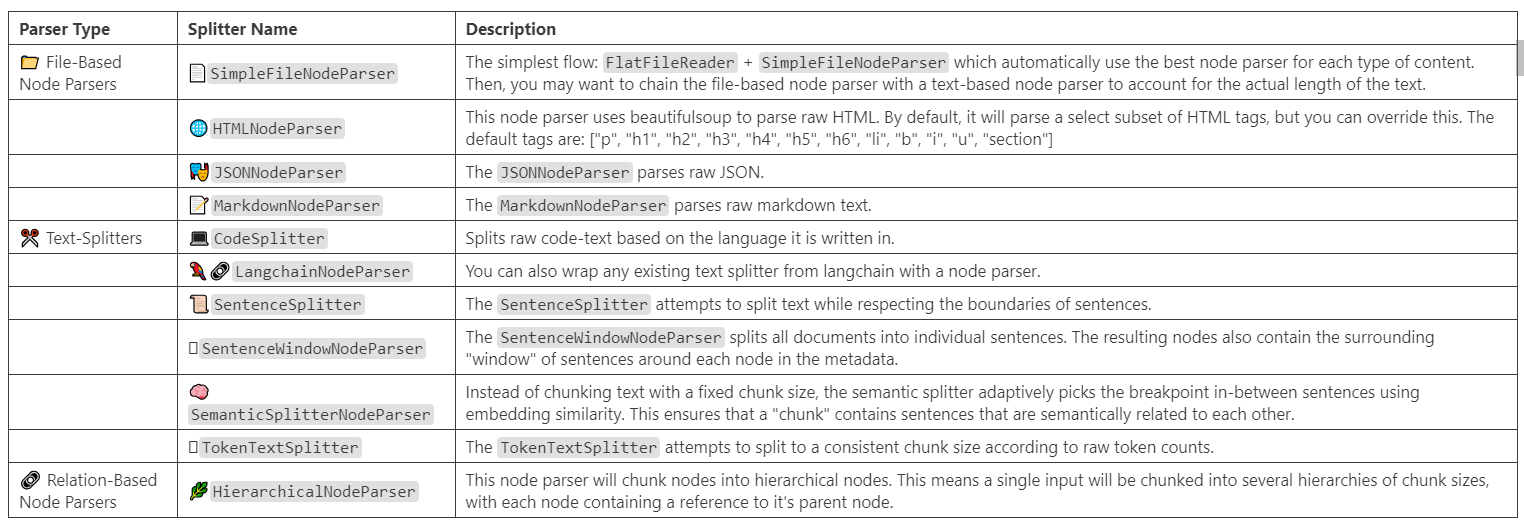
1. Larger chunks provide more context but may slow down the system.
2. Balancing comprehensiveness with speed is crucial.

Finding the Optimal Size

1. Testing various chunk sizes is essential for specific use cases and datasets.
2. Balancing information capture with efficiency is key.

**Considerations when customizing Chunk Size:**





[**Link to various chunking techniques**](https://docs.llamaindex.ai/en/stable/module_guides/loading/node_parsers/modules/)

### Recursive Retrieval (Small to big retrieval)

The concept of small to big retrieval, also known as recursive retrieval, is a key part of LlamaIndex. And, in order to use this, we need to define how to efficiently retrieve relevant context from an index based on a query. That means defining a recursive retrieval strategy, post processing the nodes once they've been retrieved and synthesizing the responses.

1. Small Chunks (Child Chunks) : Initially retrieves smaller, query-specific chunks of data.
2. Big Chunks (Parent Chunks) : Follows references to larger, contextual chunks related to the smaller chunks. Retains context within each chunk.

### Semantic Chunking

Here's the gist of what semantic chunking does:

1. Uses sentence embeddings to find breakpoints based on semantic similarity
2. Keeps related sentences together in the same chunk
3. Dynamically determines chunk size, no fixed length needed

One of the most widely used library is **SemanticSplitterNodeParser.** Here’s how it works:

1. Split document into sentences
2. Index sentences by position
3. Choose buffer size (sentences on either side to keep)
4. Measure similarity in embedding space
   * Keep similar sentences together
   * Split dissimilar sentences apart
5. Merge groups based on similarity threshold
6. Create nodes and metadata nodes

### Metadata Extraction

Metadata provides additional context or information about the nodes. During retrieval we can leverage this additional context and information, for more precise and relevant retrieval. However, the effectiveness of this approach depends on the quality and relevance of the metadata tags used.

Metadata extraction in LlamaIndex is a process that helps to disambiguate similar-looking passages of text, especially in long documents. This is achieved by using LLMs to extract contextual information relevant to the document. This information aids the retrieval and language models in distinguishing between similar passages.

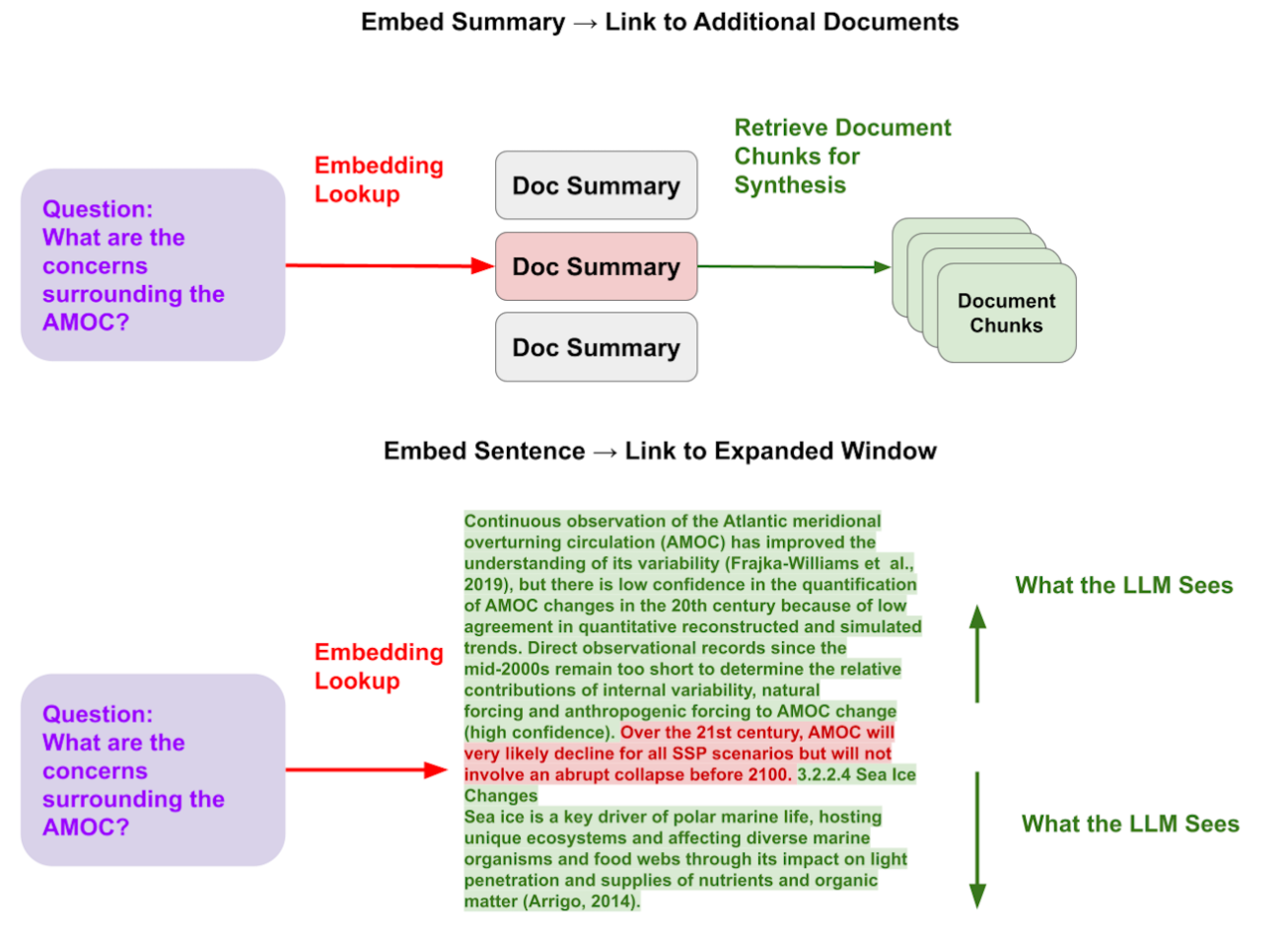
In LlamaIndex, metadata extraction is performed using various feature extractors within the **MetadataExtractor** class. [link](https://github.com/run-llama/llama_index/tree/954398e1957027a364d0d332fee61733ad322f8b/llama-index-core/llama_index/core/extractors)

These extractors include:

1. **SummaryExtractor** : This extractor automatically generates a summary over a set of Nodes.
2. **QuestionsAnsweredExtractor** : This extractor identifies a set of questions that each Node can answer.
3. **TitleExtractor** : This extractor identifies a title over the context of each Node.
4. **KeywordExtractor** : Keywords that uniquely identify the node
5. **Filenameextractor**?

### Document Summary Index

You can use the **DocumentSummaryIndex** library fromm Llama for this.



This method extracts summaries for each document to improve retrieval performance over traditional semantic search on text chunks alone. It uses concise summaries and LLM reasoning capabilities to enhance retrieval before synthesis over retrieved chunks.

**Limitations of chunk-based retrieval**

1. Chunks lack global context
2. Careful tuning of similarity thresholds required
3. Embeddings may not capture relevance well
4. Keyword filtering has its own challenges

**The Document Summary Index stores**

1. A summary extracted by an LLM for each document
2. The document split into text chunks
3. Mapping between summaries and source documents/chunks

**Retrieval approaches**

1. LLM-based: LLM scores relevance of document summaries. **DocumentSummaryIndexLLMRetriever**
2. Embedding-based: Retrieve based on summary embedding similarity. **DocumentSummaryIndexEmbeddingRetriever**

**Advantages**

1. Summaries provide more context than chunks alone
2. LLM can reason over summaries before full documents
3. Different optimal representations for retrieval vs. synthesis

**Key techniques**

1. Embed summaries linked to document chunks
2. Retrieve summaries, replace with full document content

### Query Transformation

We can achieve query transformation in atleast 3 different ways. They are focussed on rewriting the user original query in multiple ways which can then be sent for retrieval and generation.

1. Prompt definitions
2. SubquestionQuery Engine
3. Hypothetical Document Embeddings (HyDE)

**Prompt Definitions**

You can change the default prompt to suggest to the LLM to create alternate queries that are more robust

**SubquestionQuery Engine**

This is a library that helps break down complex query into simpler sub-questions (with each potentially targeting a specific data source) Here’s how it works:

1. The `**SubQuestionQueryEngine**` receives a complex query.
2. It then decomposes this query into several sub-questions. Each sub-question is designed to extract specific information from a particular data source.
3. The engine then sends these sub-questions to their respective data sources and gathers the responses.
4. Finally, it synthesizes all the intermediate responses to form a final comprehensive answer to the original complex query.

This process makes the `**SubQuestionQueryEngine**` particularly useful for handling compare/contrast queries across documents, as well as queries pertaining to a specific document. It's also well-suited for multi-document queries and can execute any number of sub-queries against any subset of query engine tools before synthesizing the final answer.

**Hypothetical Document Embeddings (HyDE)**

At a high level, [HyDE](https://arxiv.org/pdf/2212.10496.pdf) is an embedding technique that takes queries, generates a hypothetical answer, and then embeds that generated document and uses that as the final example.

Problem Tackled : Addresses the struggle of creating fully zero-shot dense retrieval systems without relevance labels.

Traditional Methods: Rely on relevance labels for document retrieval based on semantic similarities.

Zero-Shot Challenge: Especially tough without a large dataset for training.

What is HyDE?

Given a query, `HyDE` instructs a language model to generate a hypothetical document. This document captures relevance patterns but might contain inaccuracies or false details. After generating the hypothetical document, an unsupervised contrastively learned encoder encodes the document into an embedding vector. This vector identifies a neighborhood in the corpus embedding space, where similar real documents are retrieved based on vector similarity.

How Does HyDE Work?

The process starts by feeding a query to a generative model with the instruction to "write a document that answers the question". This generates a hypothetical document that captures the essence of relevance.

* 1. Generates an embedding vector for a "fake" document
  2. It does not generate any actual text content for the document
  3. The embedding is solely for reserving space in the vectorstore index
  4. There is no full hypothetical document text you can access later

This vector is used to search against the corpus embeddings, and the most similar real documents are retrieved. The idea is that a hypothetical answer to a question is more semantically similar to the real answer than the question is. In practice this means that your search would use GPT to generate a hypothetical answer, then embed that and use it for search.

Key advantages of HyDE:

1. Zero-shot, no labeled data or fine-tuning needed
2. Performs comparably to fine-tuned retrievers across tasks/languages
3. Grounds the query in real data via generated hypothetical documents

## Post-retrieval Optimization

### Node Post Processing

A postprocessor is a tool that applies some additional processing or filtering to a list of nodes returned by a query and returns the final results. Node postprocessors are modules that take a set of nodes, apply some kind of transformation or filtering, and return them.

In LlamaIndex, node postprocessors are commonly used within a query engine after the node retrieval step and before the response synthesis step. Here are a few node post processors that might be worthy looking into:

1. SimilarityPostProcessor
2. KeywordNodePostProcessor
3. MetadataReplacementPostProcessor
4. LongContextReorder
5. SentenceEmbeddingOptimizer

They are all part of the BaseNodePostProcessor class.

### Reranking

In LlamaIndex, reranking and post-processing are two different steps in the query pipeline.

Reranking is a process that takes the initial set of retrieved nodes (documents or pieces of information) and reorders them based on some criteria. This could be based on a model's prediction of relevance, a time-based factor, or any other custom criteria.

The goal of reranking is to bring the most relevant or useful nodes to the top of the list. On the other hand, post-processing is a step that happens after the retrieval and reranking steps. It involves applying transformations or filters to the set of nodes. This could include filtering out nodes below a certain similarity score, applying a time decay factor, or any other custom transformation. The goal of post-processing is to further refine the set of nodes before they are used to synthesize the final response. Both reranking and post-processing involve manipulating the set of retrieved nodes, they serve different purposes and occur at different stages in the query pipeline.

Reranking is about ordering the nodes, while post-processing is about transforming or filtering the nodes. By far, the most popular reranking technique is using Cohere's Rerank model. The two most popular ones include:

1. Colbert Rerank
2. Flag Embedding Reranker

### Forward Looking Active Retrieval augmented generation (FLARE)

FLARE is a promising approach to enhance the factual accuracy of LLMs by retrieving relevant information from external knowledge sources throughout the generation process.

FLARE (Forward-Looking Active REtrieval augmented generation) is a novel ARAG method that actively decides when and what to retrieve, leading to improved performance in long-form knowledge-intensive generation tasks.

The Limitations of Single-Retrieval Approaches

1. LLMs often hallucinate and generate factually inaccurate output
2. Existing retrieval-augmented LMs mostly retrieve information only once based on the input
3. Single retrieval is insufficient for generating long texts, where continually gathering information is essential

Actively Retrieving Information as Needed

1. FLARE iteratively predicts the upcoming sentence to anticipate future content
2. The predicted sentence is used as a query to retrieve relevant documents
3. If the predicted sentence contains low-confidence tokens, FLARE regenerates it using the retrieved documents
4. This process continues until the entire response is generated

Two Variants of FLARE

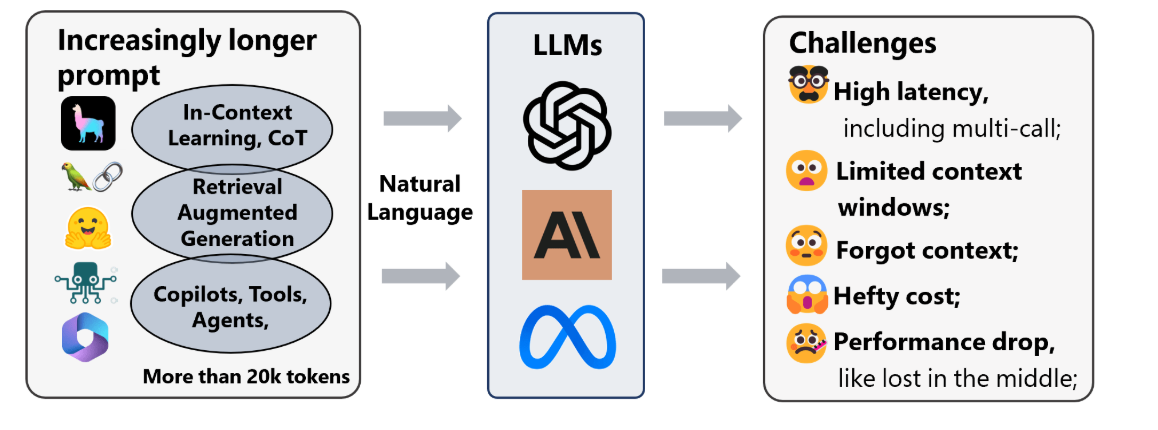
1. `FLAREinstruct`: Prompts the LM to generate retrieval queries when necessary using retrieval-encouraging instructions
2. `FLAREdirect`: Directly uses the LM's generated sentence as the retrieval query if it contains uncertain tokens

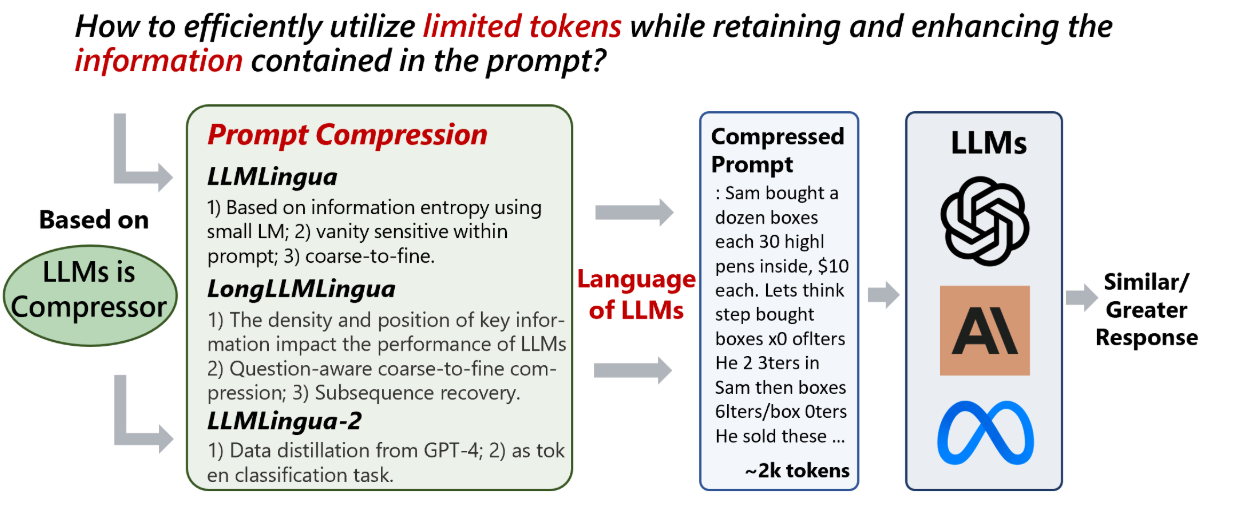
Confidence-Based Retrieval and Query Formulation

1. FLARE employs confidence-based active retrieval, triggering document retrieval only when the LM lacks necessary knowledge
2. Confidence-based query formulation methods include using masked sentences as implicit queries and generating questions as explicit queries

### Prompt Compression

LongLLMLingua (<https://llmlingua.com/>) is a method for improving the performance and efficiency of RAG and long context scenarios by compressing prompts. It addresses issues such as the "lost in the middle" problem, high costs, and context window limitations in RAG.





The key components of LongLLMLingua are:

1. Question-aware Coarse-Grained prompt compression: Evaluates the relevance between the context and the question based on the perplexity corresponding to the question.

2. Question-aware Fine-grained Prompt Compression: Uses contrastive perplexity to extract key tokens from documents that are relevant to the question.

3. Adaptive granular control during compression: Dynamically allocates different compression ratios to different documents based on the rank information obtained from the coarse-grained compression.

4. Subsequence recovery: Recovers the original prompt content by establishing the mapping relationship between the response subsequence that appears in the compressed prompt and the subsequence of the original prompt.

Experiments show that LongLLMLingua can improve performance by up to 21.4 points at a 4x compression rate in RAG scenarios, and it outperforms retrieval-based and compression-based methods in long context benchmarks like LongBench and ZeroScrolls.

The **LongLLMLinguaPostprocessor** is a postprocessor that optimizes the nodes by compressing the context using the LongLLMLingua method described. It aims to shorten the node text based on the given query to improve efficiency and reduce computational costs.

### Self-Correcting Query Engines

Self-correcting query engines in LlamaIndex evaluate their own output and then self-correct to provide better responses. They are designed to improve the quality of responses from a base query engine.

There are a few types of self-correcting query engines:

1. Retry Query Engine: This engine uses an evaluator to improve the response from a base query engine. It first queries the base query engine, then uses the evaluator to decide if the response passes. If the response passes, it returns the response. Otherwise, it transforms the original query with the evaluation result into a new query and repeats the process up to a maximum number of retries.
2. Retry Source Query Engine: This engine modifies the query source nodes by filtering the existing source nodes for the query based on LLM node evaluation.
3. Retry Guideline Query Engine: This engine uses guidelines to direct the evaluator's behaviour. It can be customized with your own guidelines. The engine evaluates the response against the guidelines, and if the response doesn't meet the guidelines, it transforms the query and retries.

# Modular RAG

## Vector Store Query mode

The `vector\_store\_query\_mode` in LlamaIndex determines the type of search to be performed. Here's a brief description of each mode:

1. `default`: This mode performs a vector search. It retrieves the most similar vectors based on the query vector. They create a numerical representation of a piece of text, represented as a long list of numbers. These dense vectors can capture rich semantics across the entire piece of text. `alpha=0.75` is used by default.
2. `hybrid`: This mode performs a hybrid search. It combines vector search with traditional search methods. `alpha` parameter determines weighting (`alpha = 0` -> bm25, `alpha = 1` -> vector search).
3. `semantic\_hybrid`: Semantic hybrid search combines text search with vector embeddings. Text search provides keyword matching and lexical retrieval. Vector embeddings allow finding documents with similar meaning, even if they don't contain exact keyword matches. This mode incorporates semantic reranking to hybrid search results to improve search relevance.
4. `sparse`: Most of the elements in a sparse vector are zero, with only a few key values being non-zero. These sparse vectors are great at capturing specific keywords and similar small details. You need to use a specialized embedding model to create sparse vectors.
5. `FastEmbed` has a few choices for sparse text embedding models, for example you can pass in `prithvida/Splade\_PP\_en\_v1` as the model name when you run `setup\_embed\_model` if you want to use them.
6. `text\_search`: Text search looks for exact keyword matches between the query and documents.
7. `similarity\_top\_k`: controls the final number of returned nodes. A fusion algorithm is applied to rank and order the nodes from different vector spaces, `similarity\_top\_k=2` means the top two nodes after fusion are returned.
8. `hybrid\_top\_k`: return top k results from `hybrid` search. `similarity\_top\_k` is used for dense search top k

## Hybrid Fusion Retriever

The system follows a three-step process:

1. Query Generation/Rewriting: It creates multiple queries from the original user query to better match the user's intent and improve the precision and recall of the retrieved results.
2. Retrieval: It performs the retrieval for each query over an ensemble of retrievers.
3. Reranking/Fusion: It combines the results from all queries and applies a reranking step to fuse the top relevant results.

## Agentic RAGs

Data Agents are LLM-powered knowledge workers in LlamaIndex that can intelligently perform various tasks over your data, in both a "read" and "write" function.

They are capable of:

1. Performing automated search and retrieval over different types of data - unstructured, semi-structured, and structured.
2. Calling any external service API in a structured fashion, and processing the response + storing it for later.

In that sense, agents are a step beyond our query engines in that they can not only "read" from a static source of data, but can dynamically ingest and modify data from a variety of different tools.

Building a data agent requires the following core components:

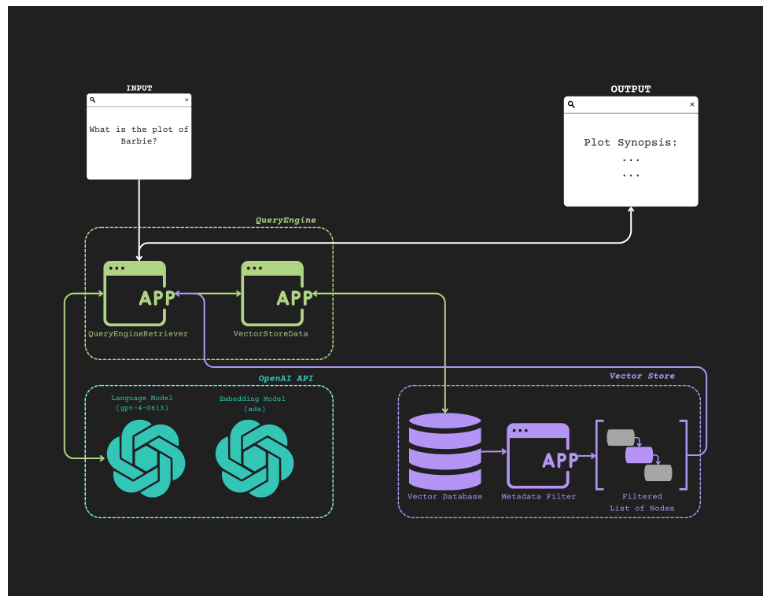
1. A reasoning loop

2. Tool abstractions

A data agent is initialized with set of APIs, or Tools, to interact with; these APIs can be called by the agent to return information or modify state. Given an input task, the data agent uses a reasoning loop to decide which tools to use, in which sequence, and the parameters to call each tool.

How does the agent work?

A user's query is processed through a query engine, transformed, and then filtered and retrieved from a vector database to provide a relevant response.



1. User Query: A user inputs a query.

2. Query Engine: The query is first sent to the `QueryEngineRetriever` application, which is part of the Query Engine system. The `QueryEngineRetriever` then interacts with the `VectorStoreData` application to process the query further.

3. OpenAI API: The `VectorStoreData` communicates with two models from the OpenAI API:

1. Language Model: Understands and generate coherent responses based on the query.
2. Embedding Model: Converts the query and data into vector representations for easier processing and retrieval.

4. Vector Store:

1. The processed information is sent to a `Vector Database` within the Vector Store system.
2. The `Vector Database` stores the vectorized data and can filter it based on metadata through the `Metadata Filter` application.
3. This filtering process results in a `Filtered List of Nodes`, which represents the most relevant pieces of information related to the query.

5. Result Compilation:

1. The filtered information is sent back to the `VectorStoreData` application, which compiles the final response.
2. This compiled response is then sent back to the `QueryEngineRetriever` application.

6. Response Display: Finally, the processed and relevant information is displayed to the user as the plot synopsis in the top right box.