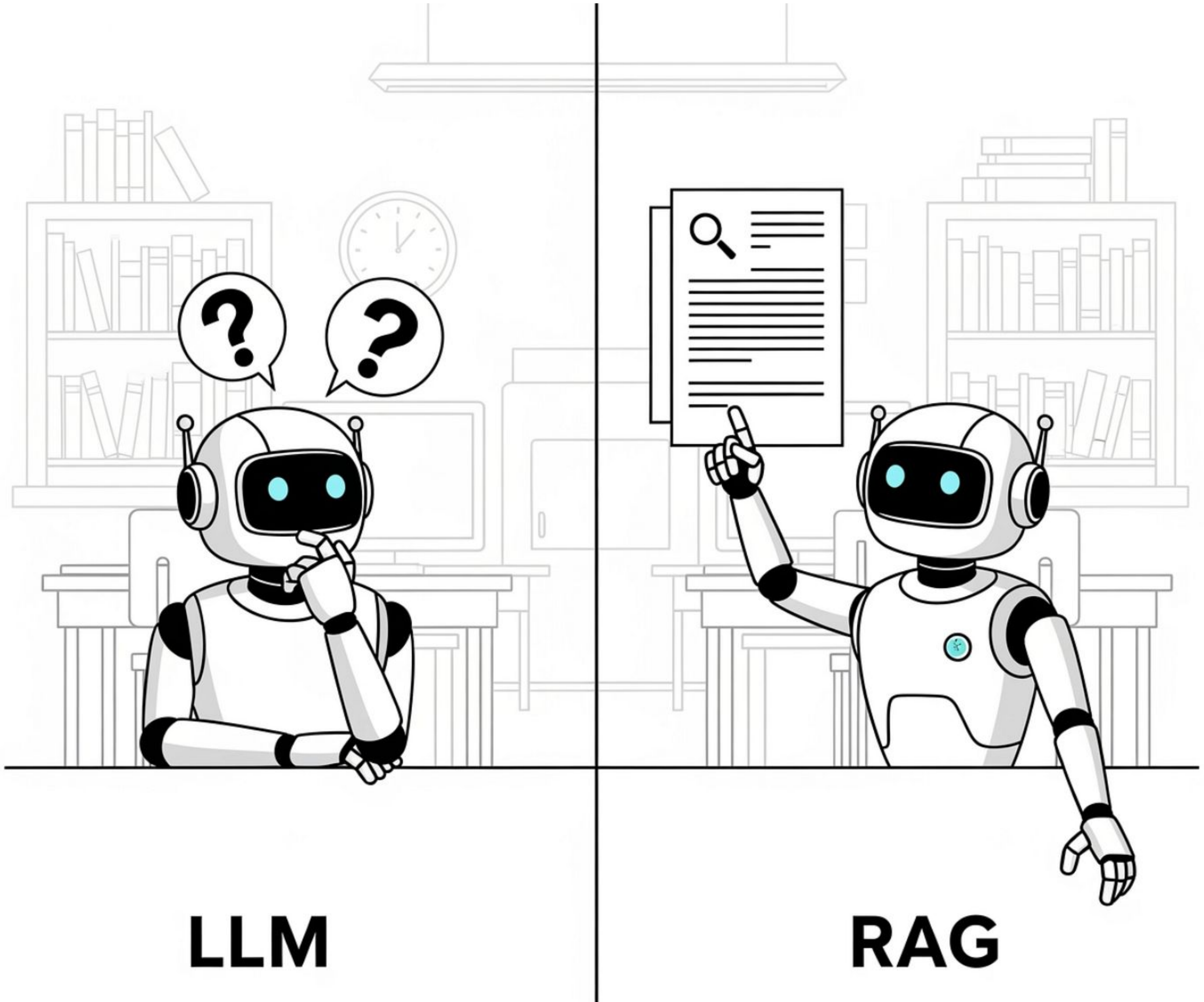
RAG: Retrieval Augmented Generation

Presenter: Ali Faheem

LLM's and It's Limitations

- **Outdated Information:** Generative AI uses large language models trained on data up to a certain point. Requests for information beyond that date may yield inaccurate results.
- **Hallucinations:** AI can produce outputs that are factually incorrect or nonsensical but appear coherent. This can mislead and impact business decisions.
- **Domain-Specific Accuracy:** AI often lacks precise information for specific domains. For example, it may not accurately address organizational HR policies tailored to specific employees, tending towards generic responses.
- **Source Citations:** Generative AI often doesn't specify sources, making citations difficult and potentially unethical as proper credit may not be given.
- **Training Time:** Updating models with new information requires significant resources and time, as training is computationally intensive.

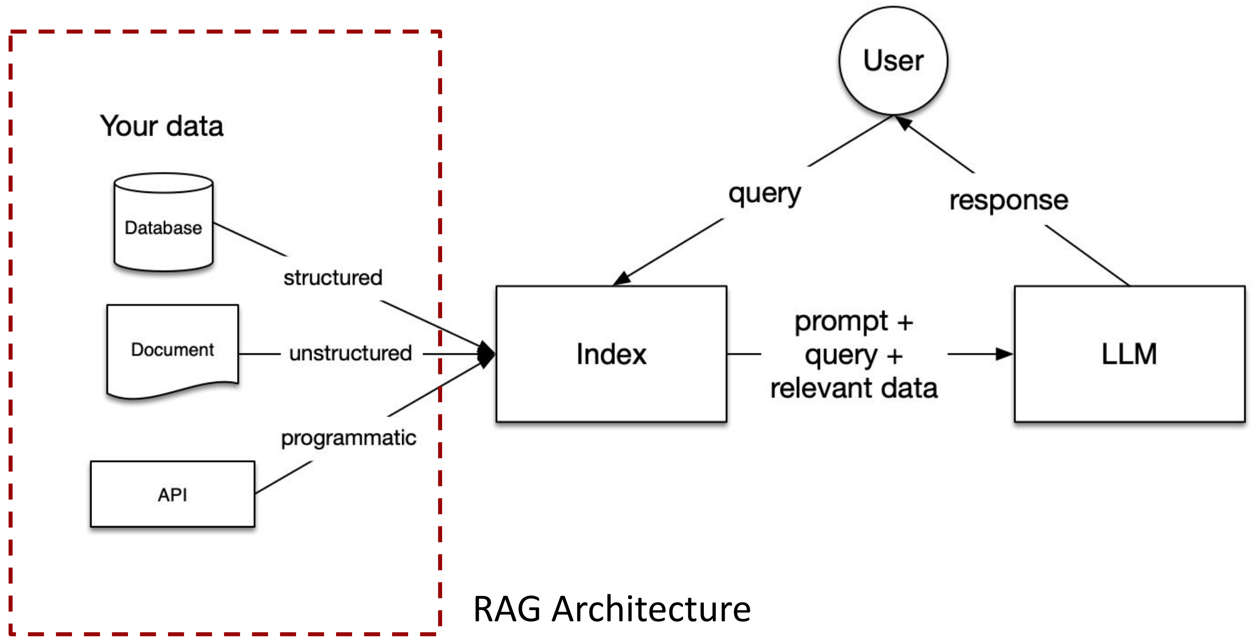

LLM
RAG

What is RAG?

- RAG stands for Retrieval-Augmented Generation. It's an advanced technique used in Large Language Models (LLMs)
- RAG **combines retrieval and generation** processes to enhance the capabilities of LLMs
- In RAG, the model retrieves relevant **information from a knowledge base** or external sources
- This retrieved information is then used in conjunction with the model's internal knowledge to generate coherent and **contextually relevant responses**
- RAG enables LLMs to produce higher-quality and more **context-aware outputs** compared to traditional generation methods

Stages within RAG

- **Loading Data:** Import data from various sources like text files, PDFs, websites, databases, or APIs into your pipeline
- **Indexing:** Create data structures, typically vector embeddings and metadata, to enable efficient and contextually relevant queries.
- **Querying:** Utilize various querying strategies, including sub-queries, multi-step queries, and hybrid methods.



RAG Architecture

Data Loading

The process of importing data from various sources into a system for processing, analysis, or storage. In RAG (Retrieval-Augmented Generation), it’s the first step for preparing data for indexing and retrieval.

Types of Data Sources

1. **Unstructured Documents**
   - Text files, PDFs, Word docs.
   - *Examples*: Research papers, emails, social media posts.
   - *Challenges*: Requires preprocessing (text extraction, chunking).

2. **Structured Databases**
   - Tables with rows/columns (e.g., MySQL, PostgreSQL).
   - *Examples*: Spreadsheets, CSV files.
   - *Challenges*: Needs schema mapping and transformation.

3. **APIs**
   - Data from external services (e.g., weather, stock market, Slack APIs).
   - *Challenges*: Retrieving and mapping the required data.

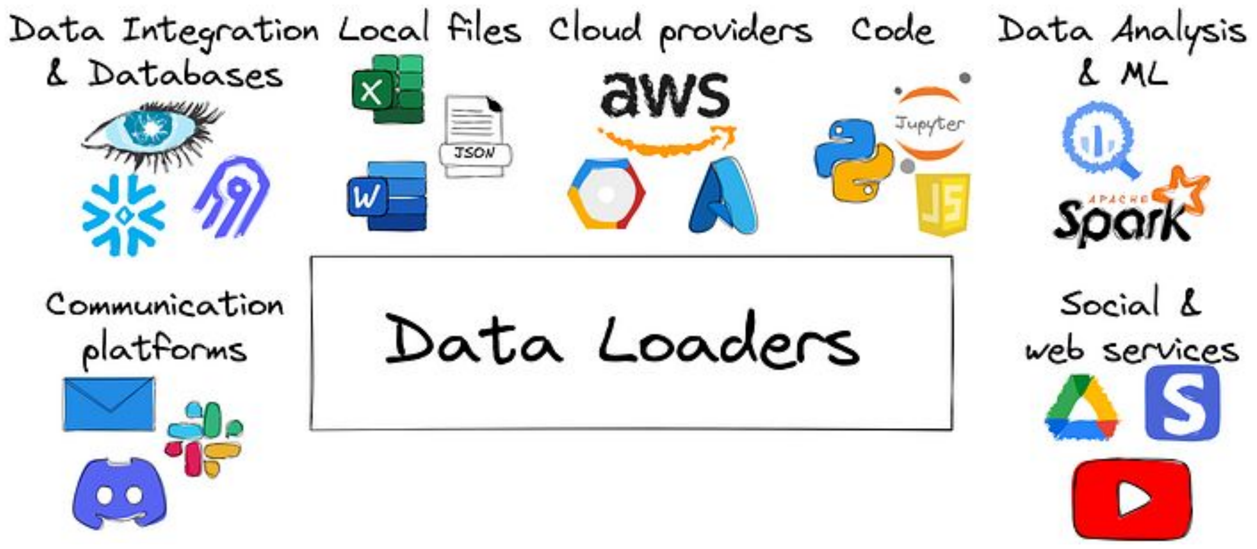
Data Loading

Llama Index & LangChain

- offers flexible data loader pipelines for unstructured and structured data.
- Supports integration with external APIs, documents, and databases.

Challenge

- Custom data (e.g., proprietary formats, unique APIs) often requires tailored solutions. Involves building custom scripts, ETL pipelines, or using specialized libraries.

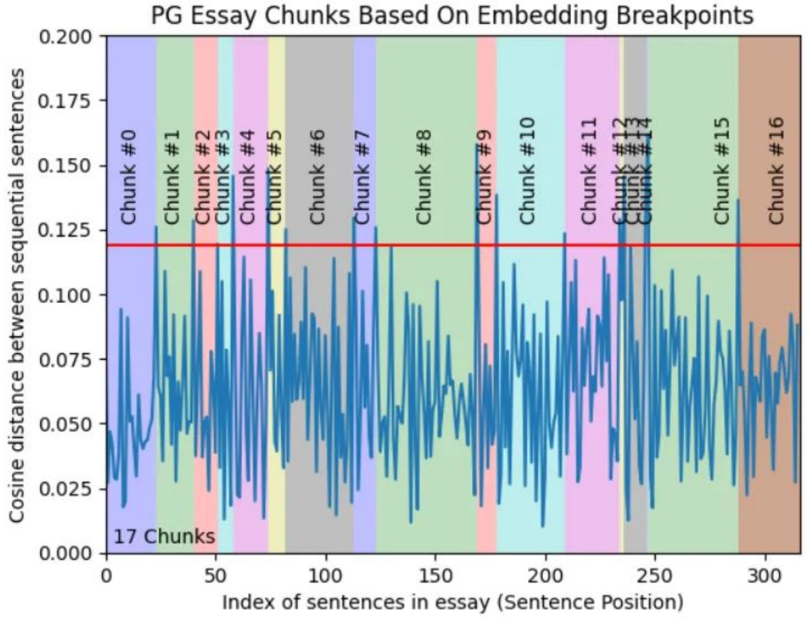


Data Chunking

Breaking down your large data files into more manageable segments.
Following are the 5 levels of chunking strategies based on the complexity and effectiveness.

- Level 1: Fixed Size Chunking
- Level 2: Recursive Chunking
- Level 3 : Document Based Chunking
  - custom approach specific to document
- Level 4: Semantic Chunking
  - Grouping similar semantic sentences based on embedding similarity
- Level 5: Agentic Chunking
  - Leveraging LLM to identify the breakpoints in documents.

Semantic Chunking

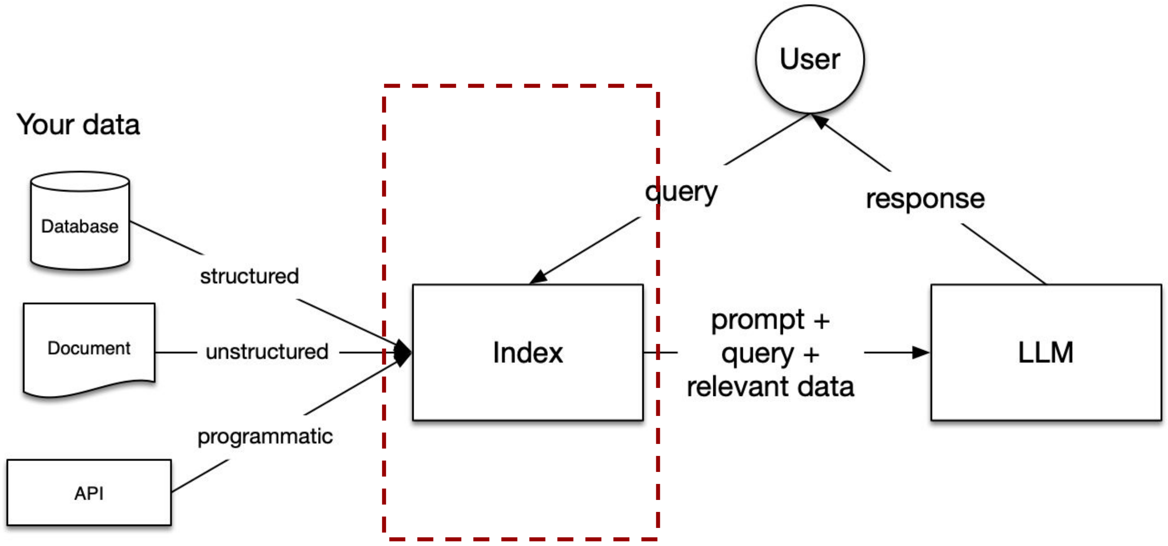


Example: `Greg went to the park. He likes walking` > `['Greg went to the park.', 'Greg likes walking']`

Example of Agentic Chunking

Stages within RAG

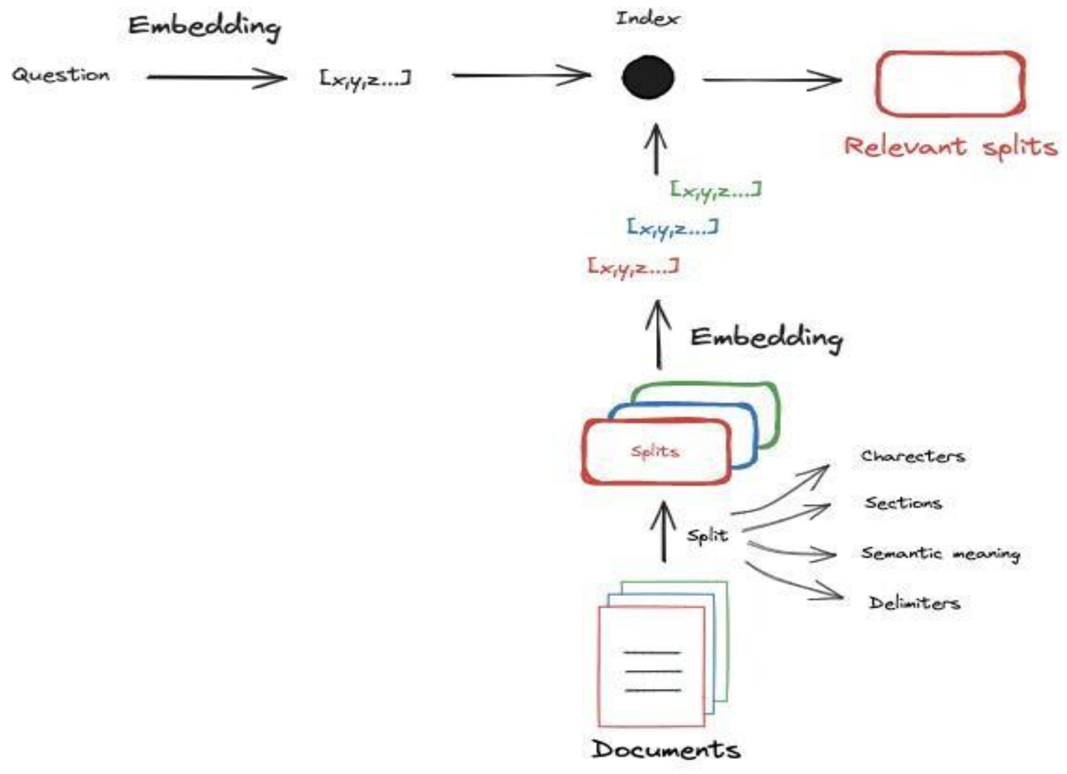
- **Loading Data:** Import data from various sources like text files, PDFs, websites, databases, or APIs into your pipeline
- **Indexing:** Create data structures, typically vector embeddings and metadata, to enable efficient and contextually relevant queries.
- **Querying:** Utilize various querying strategies, including sub-queries, multi-step queries, and hybrid methods.



RAG Architecture

Indexing

Indexing in terms of RAG is the process of organizing a vast amount of text data in a way that allows the RAG system to quickly find the most relevant pieces of information for a given query



Indexing Terminologies

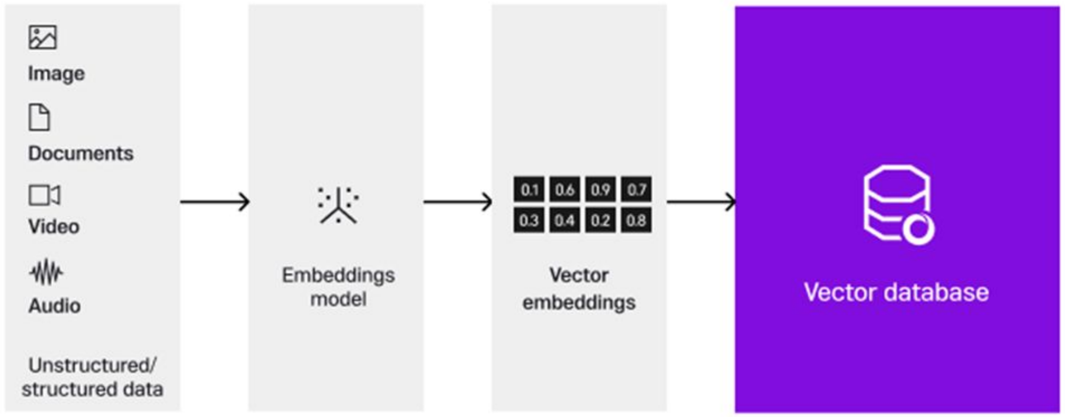
Node: A node represents a chunk or segment of data within the indexing structure. It can be a sentence, paragraph, or document

- Relationship between nodes (parent-child) help in in structuring the data

Metadata: Metadata refers to additional information attached to nodes or documents, such as keywords, tags, timestamps, Questions, Summary or source information

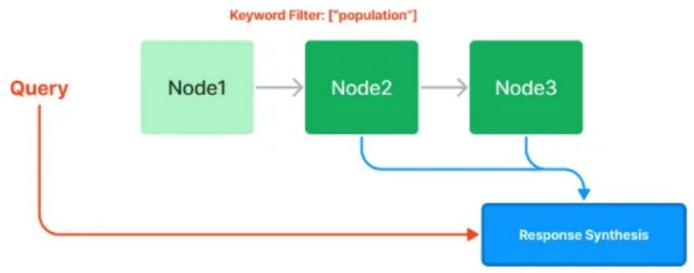
Vector Store: A database or storage system that stores embeddings (vector representations) of text data. Optimized for similarity search

- Example: ChromaDB, FAISS, Pinecone

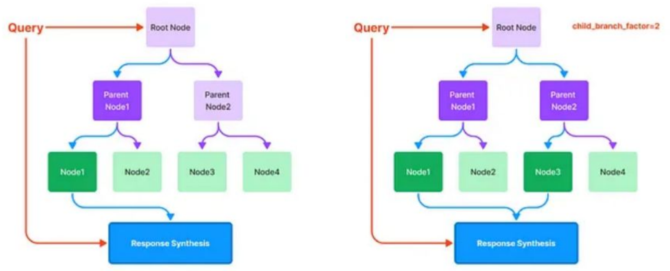


Indexes Types

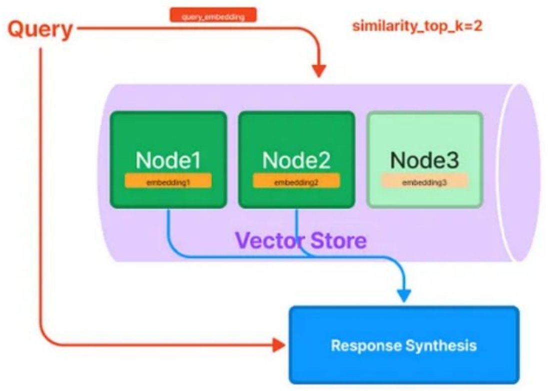
List Index



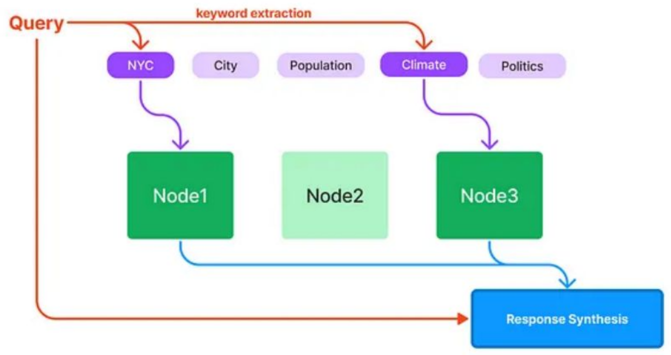
Tree Index



Vector Store Index



Keyword Table Index



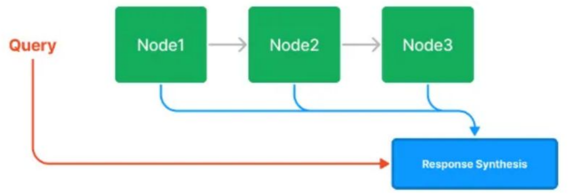
Indexes Types

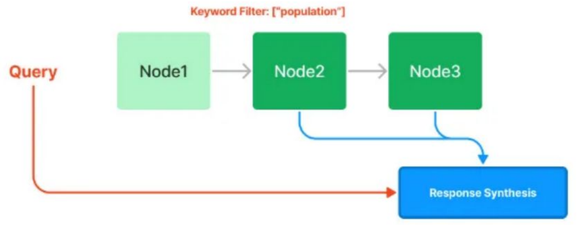
List Index

stores documents as a sequence of nodes. During index construction, the document texts are chunked up, converted to nodes

Advantages

- Simplicity: simple to implement and use
- Efficiency: efficient for retrieving documents that contain a particular keyword or phrase
- Flexibility: The List Index can be used with any type of document data.





Disadvantages

- Scalability: The List Index can become inefficient for large datasets.

Indexes Types

Vector Store Index

stores documents as Embedding vectors

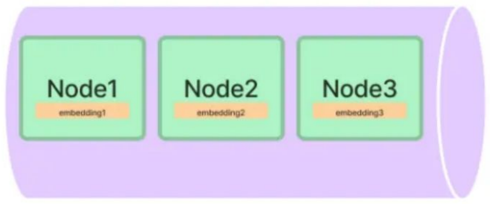
Advantages

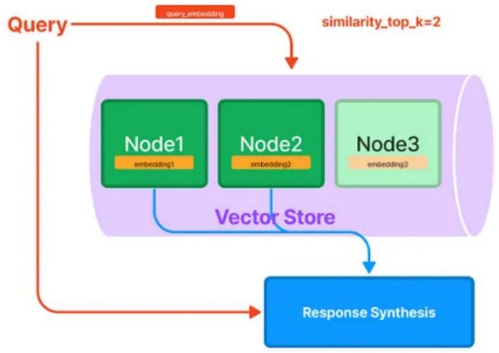
- Scalable: can be used to index large datasets

- Efficient: efficient for finding similar documents. No text processing required

Disadvantages

- Can be expensive and computationally expensive to use, based on the embedding model choices.





Indexes Types

Tree Index

stores the text of documents in a tree-like structure. Each node in the tree represents a summary of the documents that are its children

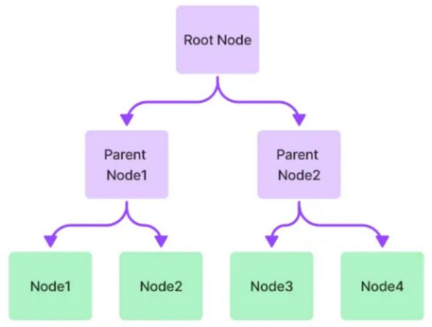
For each query we identify the keywords and look for those keywords in the node summaries

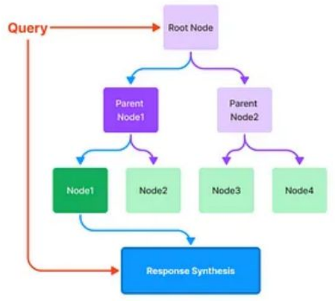
Advantages

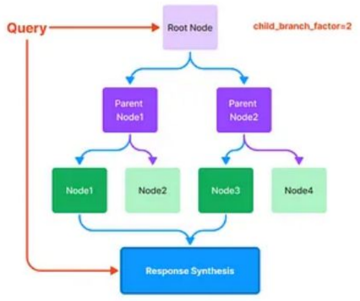
- Scalable: can be used to index large datasets
- Efficient: efficient for finding related documents because of tree structure

Disadvantages

- difficult to create than other types of indexes







Indexes Types

Keyword Table Index

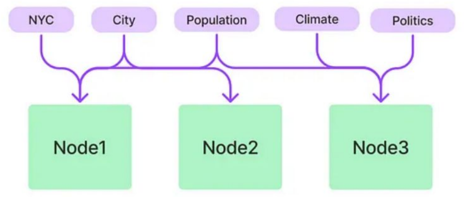
Stores the keywords of documents in a table. Each row in the table represents a keyword, and each column represents a document.

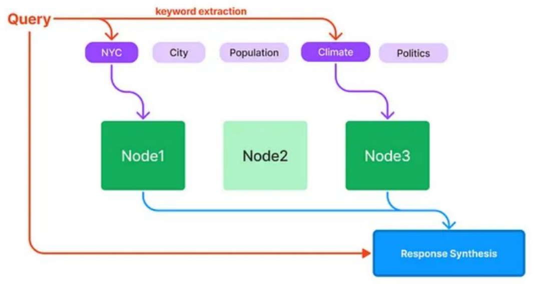
Advantages

- Keyword table indexes are also very scalable.
- efficient for finding documents that contain a specific keyword

Disadvantages

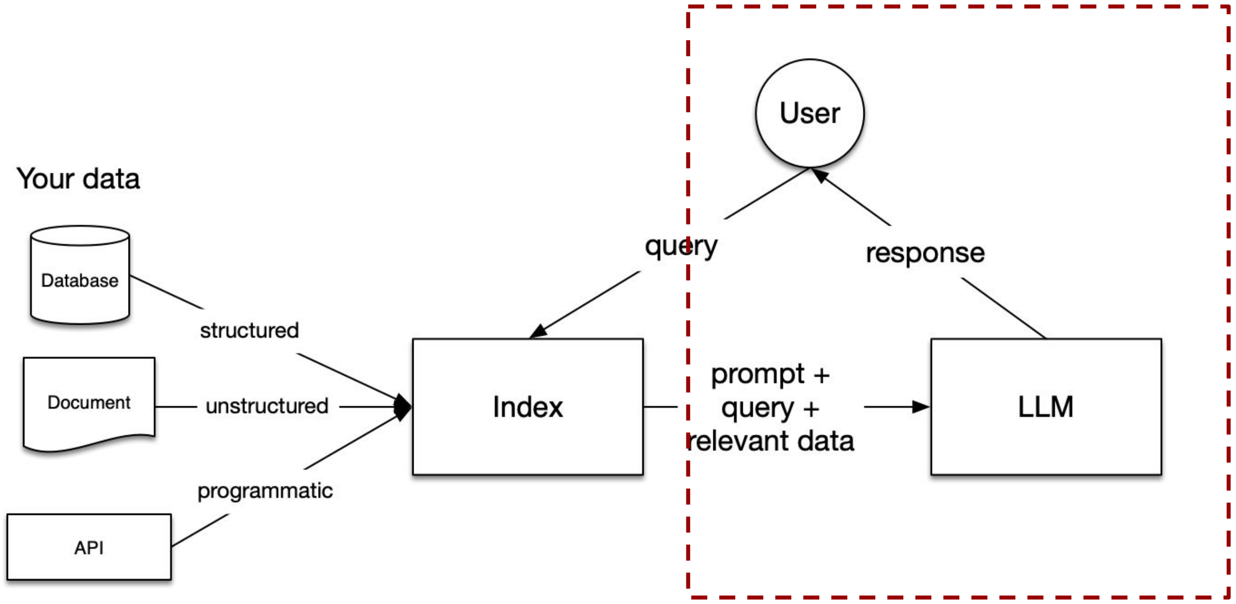
- Keyword table indexes can be less accurate for documents with synonyms or related keywords.





Stages within RAG

- **Loading Data:** Import data from various sources like text files, PDFs, websites, databases, or APIs into your pipeline
- **Indexing:** Create data structures, typically vector embeddings and metadata, to enable efficient and contextually relevant queries.
- **Querying:** Utilize various querying strategies, including sub-queries, multi-step queries, and hybrid methods.



RAG Architecture

Querying

Querying is most important part of LLM application. At its simplest, querying is just a prompt call to an LLM: it can be a question and get an answer, or a request for summarization, or a much more complex instruction.

Stages of querying

Querying consists of three distinct stages:

- **Retrieval** is when you find and return the most relevant documents for your query from your Index. As previously discussed in indexing, the most common type of retrieval is "top-k" semantic retrieval.
- **Postprocessing** is when the Nodes (chunks) retrieved are optionally transformed, or filtered, for instance by requiring that they have specific metadata such as keywords attached.
- **Response synthesis** is when your query, your most-relevant data and your prompt are combined and sent to your LLM to return a response.

Response Synthesis

In Llama Index, multiple strategies can be adopted for response synthesis including

- **Compact:** Add as many text chunks that can fit within the context window
- **Refine:**create and refine an answer by sequentially going through each retrieved text chunk. This makes a separate LLM call per Node/retrieved chunk.
- **Tree_summarize**: Given a set of Node objects query as many times as needed so that all chunks have been queried, resulting in as many answers that are themselves recursively used as chunks
- **Accumulate**: Given a set of Node objects and the query, apply the query to each Node text chunk while accumulating the responses into an array. Returns a concatenated string of all responses.

Challenges in Naive RAG

Relevance of Retrieved Information

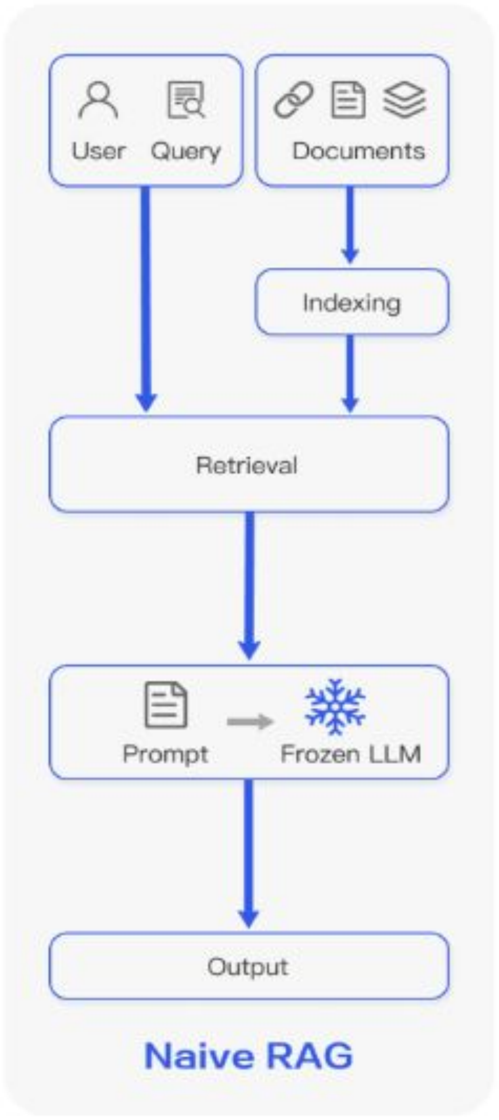
- Difficulty ensuring the retrieved information is highly relevant to the query.
- Potential for retrieving unrelated chunks, reducing the quality of the generated output.

Information Redundancy:

- Risk of retrieving redundant or overlapping information chunks, leading to repetitive generated content.

Complex Query:

- Naive RAG fails on complex query where in depth analysis is required to answer the query
- A query may require scattered information from different sources



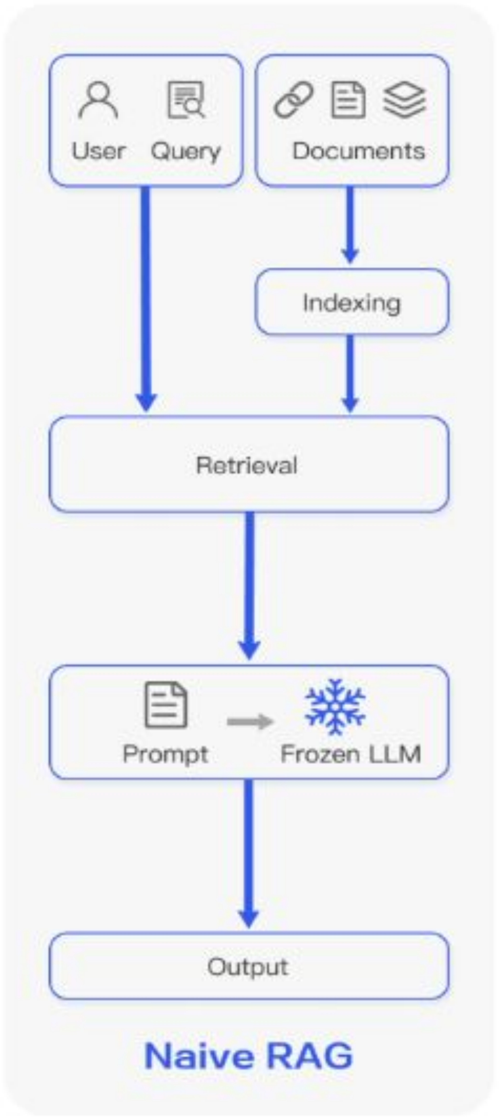
Challenges in Naive RAG (cont)

Lack of Deep Reasoning

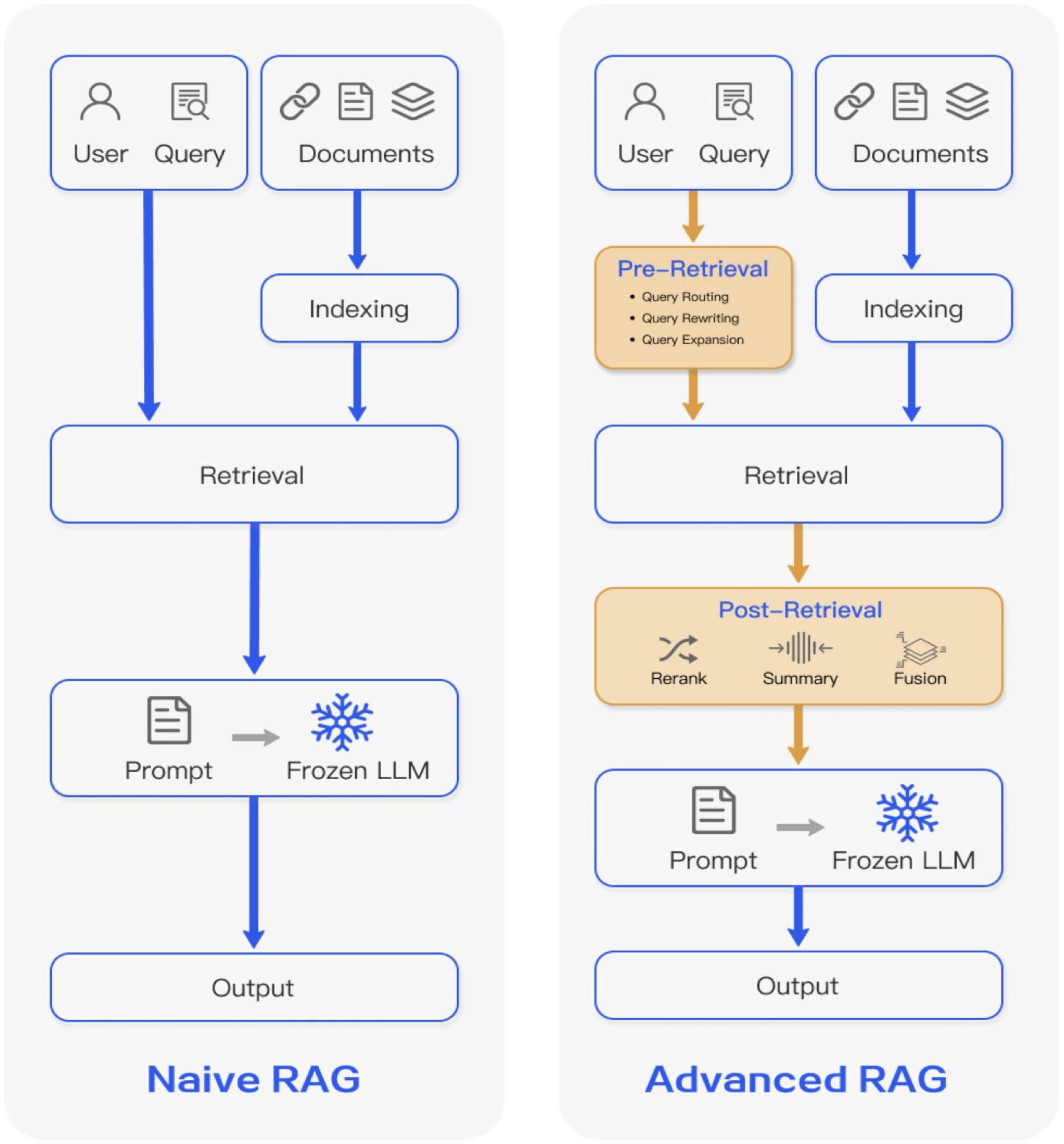
- Naive RAG systems primarily rely on pattern-matching and retrieval, which restricts their ability to perform deep reasoning or complex analysis.

Difficulty Integrating Information from Multiple Sources

- Naive RAG systems may struggle to effectively combine information from multiple diverse data sources



Naive vs Advanced RAG



Pre Retrieval vs Post Retrieval Optimizations

Pre-Retrieval Optimizations:

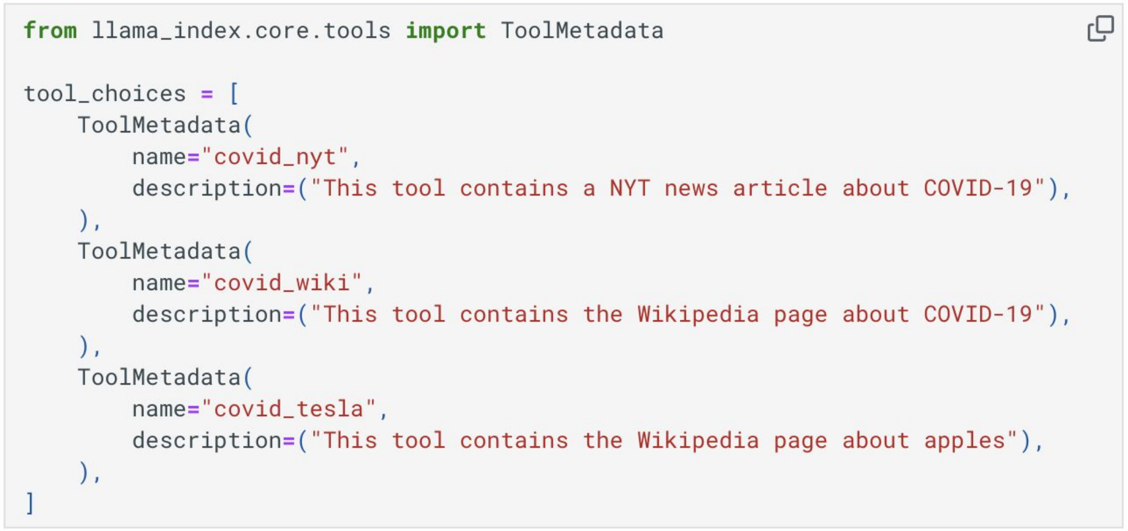
- **Query Reformulation**: Modifying the query to improve the chances of retrieving relevant documents (e.g., adding synonyms, alternative phrasings).
- **Query Expansion**: Expanding the query by adding relevant keywords to enhance retrieval performance.
- **Routing to Relevant Data Sources:** Directing the query to specific, relevant data sources before retrieval to optimize the search and ensure higher-quality, focused results.

Post-Retrieval Optimizations:

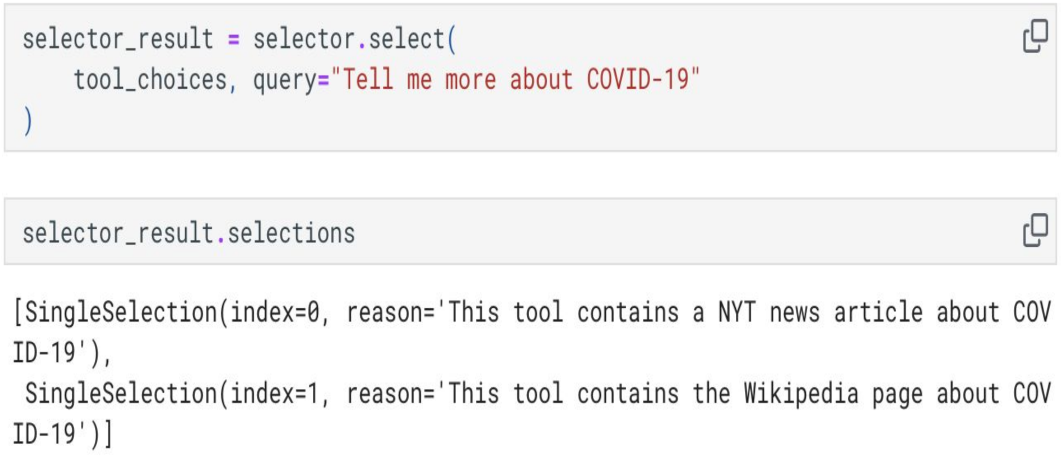
- **Ranking & Re-ranking**: After documents are retrieved, applying algorithms to rank them based on relevance, ensuring the top results are the most useful.
- **Document Filtering**: Removing irrelevant or low-quality documents from the retrieved set before passing them to the generator.
- **Summarization**: condensing multiple retrieved documents into a more concise and relevant form before passing them to the generator.
- **Fusion** refers to the process of combining or merging multiple retrieved documents or information from different sources into a single coherent response.

Pre-Retrieval Optimization Techniques

- **Query Routing**: Keep the query, but identify the relevant subset of tools that the query applies to. Output those tools (indexes, dbs) as the relevant choices.



```python
from llama_index.core.tools import ToolMetadata

tool_choices = [
    ToolMetadata(
        name="covid_nyt",
        description=("This tool contains a NYT news article about COVID-19"),
    ),
    ToolMetadata(
        name="covid_wiki",
        description=("This tool contains the Wikipedia page about COVID-19"),
    ),
    ToolMetadata(
        name="covid_tesla",
        description=("This tool contains the Wikipedia page about apples"),
    ),
]
```



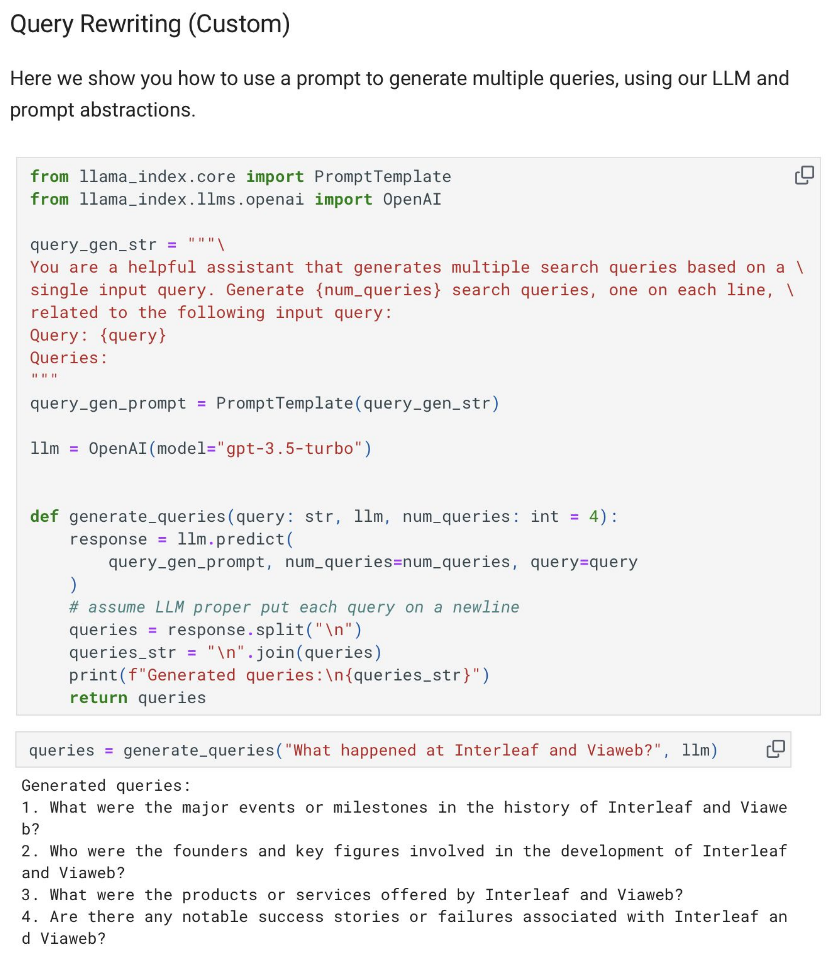
```python
selector_result = selector.select(
    tool_choices, query="Tell me more about COVID-19"
)
```

```python
selector_result.selections
```

```
[SingleSelection(index=0, reason='This tool contains a NYT news article about COV
ID-19'),
 SingleSelection(index=1, reason='This tool contains the Wikipedia page about COV
ID-19')]
```

Pre-Retrieval Optimization Techniques

- **Query-Rewriting:** generate multiple queries , leading to higher-quality retrieved results.



Query Rewriting (Custom)

Here we show you how to use a prompt to generate multiple queries, using our LLM and prompt abstractions.

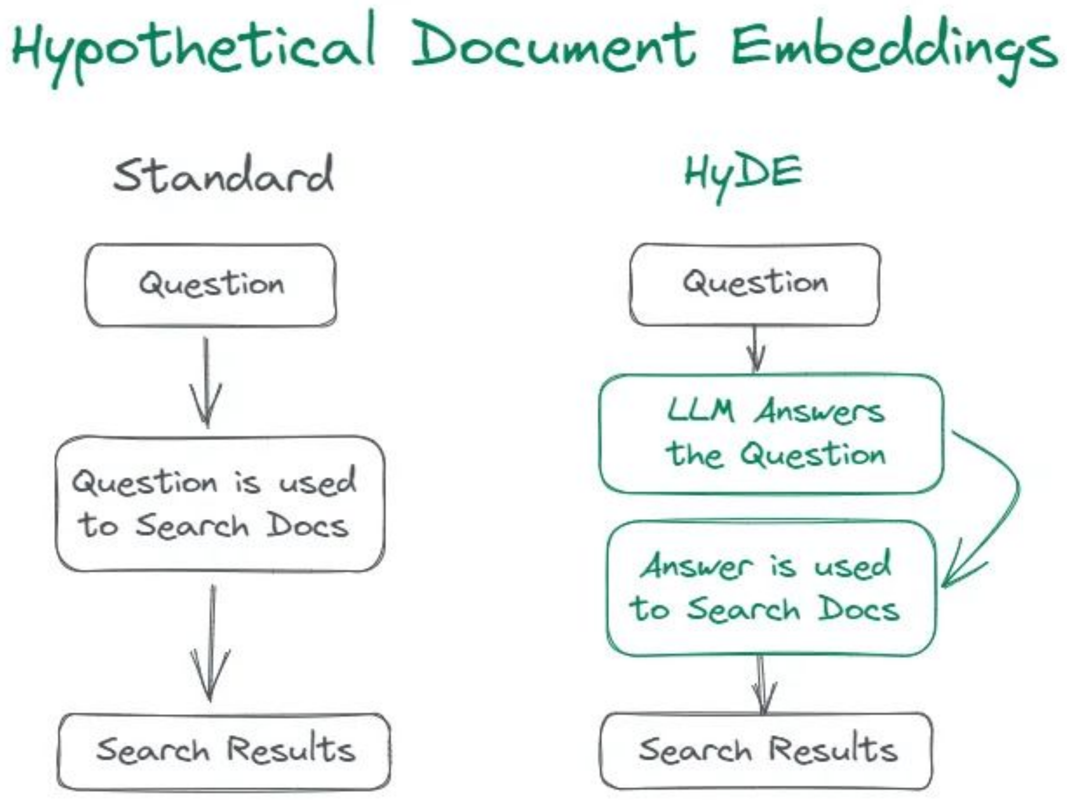
```python
from llama_index.core import PromptTemplate
from llama_index.llms.openai import OpenAI

query_gen_str = """\
You are a helpful assistant that generates multiple search queries based on a \
single input query. Generate {num_queries} search queries, one on each line, \
related to the following input query:
Query: {query}
Queries:
"""
query_gen_prompt = PromptTemplate(query_gen_str)

llm = OpenAI(model="gpt-3.5-turbo")


def generate_queries(query: str, llm, num_queries: int = 4):
    response = llm.predict(
        query_gen_prompt, num_queries=num_queries, query=query
    )
    # assume LLM proper put each query on a newline
    queries = response.split("\n")
    queries_str = "\n".join(queries)
    print(f"Generated queries:\n{queries_str}")
    return queries
```

```python
queries = generate_queries("What happened at Interleaf and Viaweb?", llm)
```

```
Generated queries:
1. What were the major events or milestones in the history of Interleaf and Viawe
b?
2. Who were the founders and key figures involved in the development of Interleaf
and Viaweb?
3. What were the products or services offered by Interleaf and Viaweb?
4. Are there any notable success stories or failures associated with Interleaf an
d Viaweb?
```

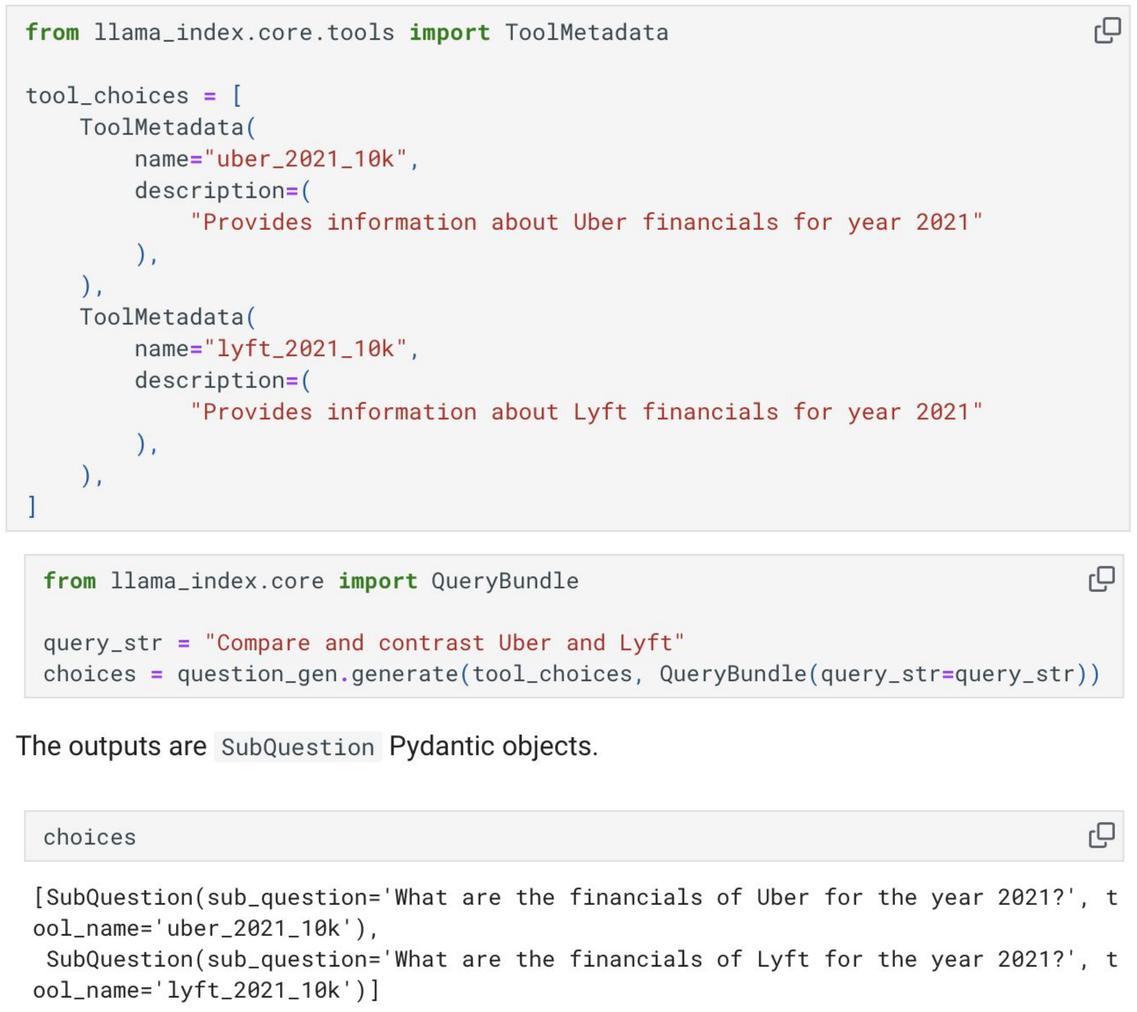
Pre-Retrieval Optimization Techniques

- **HyDE Query-Rewriting:** HyDE uses a Language Learning Model to create a hypothetical document when responding to a query.
  - Rather than seeking embedding similarity for questions or queries, it focuses on answer-to-answer embedding similarity.


Hypothetical Document Embeddings
Standard
Question
Question is used to Search Docs
Search Results
HyDE
Question
LLM Answers the Question
Answer is used to Search Docs
Search Results

Pre-Retrieval Optimization Techniques

- **Sub Question Query:** Given a set of tools/data-sources and a user query, decide both the 1) set of sub-questions to generate, and 2) the tools that each sub-question should run over.



```python
from llama_index.core.tools import ToolMetadata

tool_choices = [
    ToolMetadata(
        name="uber_2021_10k",
        description=(
            "Provides information about Uber financials for year 2021"
        ),
    ),
    ToolMetadata(
        name="lyft_2021_10k",
        description=(
            "Provides information about Lyft financials for year 2021"
        ),
    ),
]
```

```python
from llama_index.core import QueryBundle

query_str = "Compare and contrast Uber and Lyft"
choices = question_gen.generate(tool_choices, QueryBundle(query_str=query_str))
```

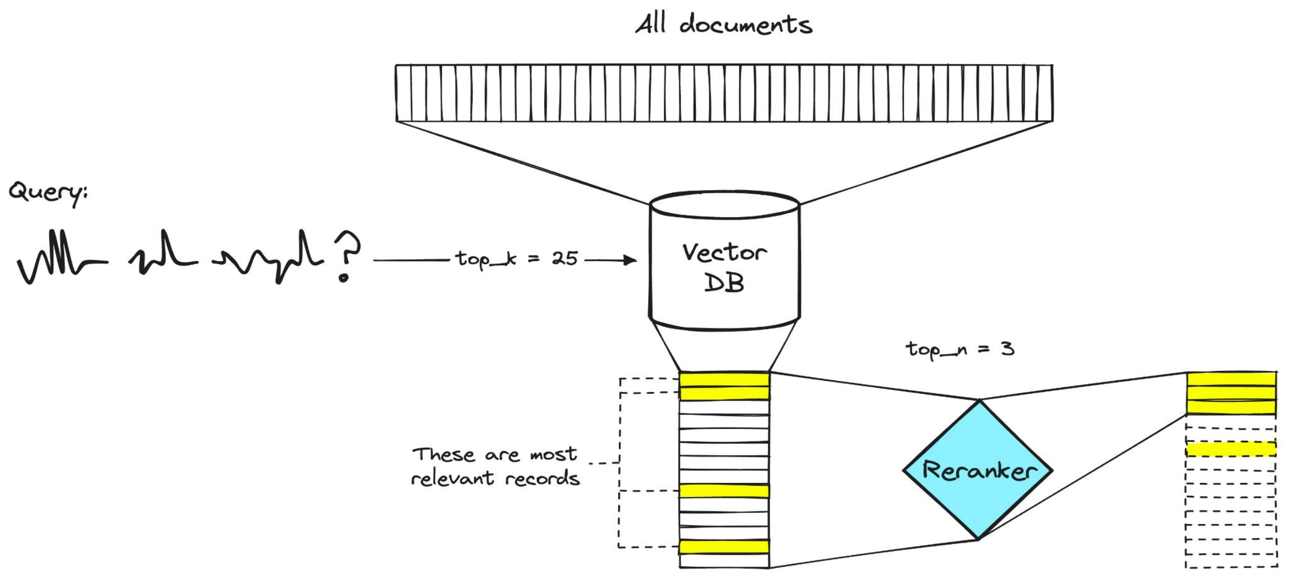
The outputs are `SubQuestion` Pydantic objects.

```python
choices
```

```
[SubQuestion(sub_question='What are the financials of Uber for the year 2021?', t
ool_name='uber_2021_10k'),
 SubQuestion(sub_question='What are the financials of Lyft for the year 2021?', t
ool_name='lyft_2021_10k')]
```

Post-Retrieval Optimization Techniques

- **Reranking**
  - RAG enables semantic searches across vast text collections.
  - Documents range from thousands to billions.
  - We use vector search for fast results. There is some information loss because we're compressing this information into a single vector.
  - we need to keep smaller number of text chunks as LLMs have limited context window
  - Re-ranking the retrieved information to relocate the most relevant content to the edges of the prompt is a key strategy



Limitations of Advanced RAG

- **Fixed Pipeline** - Rigid structure (**Retrieve** → **Read** → **Generate**) limits adaptability.
- **Inefficient Complex Query Handling** - Struggles with **multi-hop reasoning** and **iterative retrieval**.
- **Context Overload** - Excessive retrieved documents introduce **noise**, reducing LLM performance.
- **Lack of Feedback Integration** - Does not leverage **LLM-generated feedback** for improving retrieval.

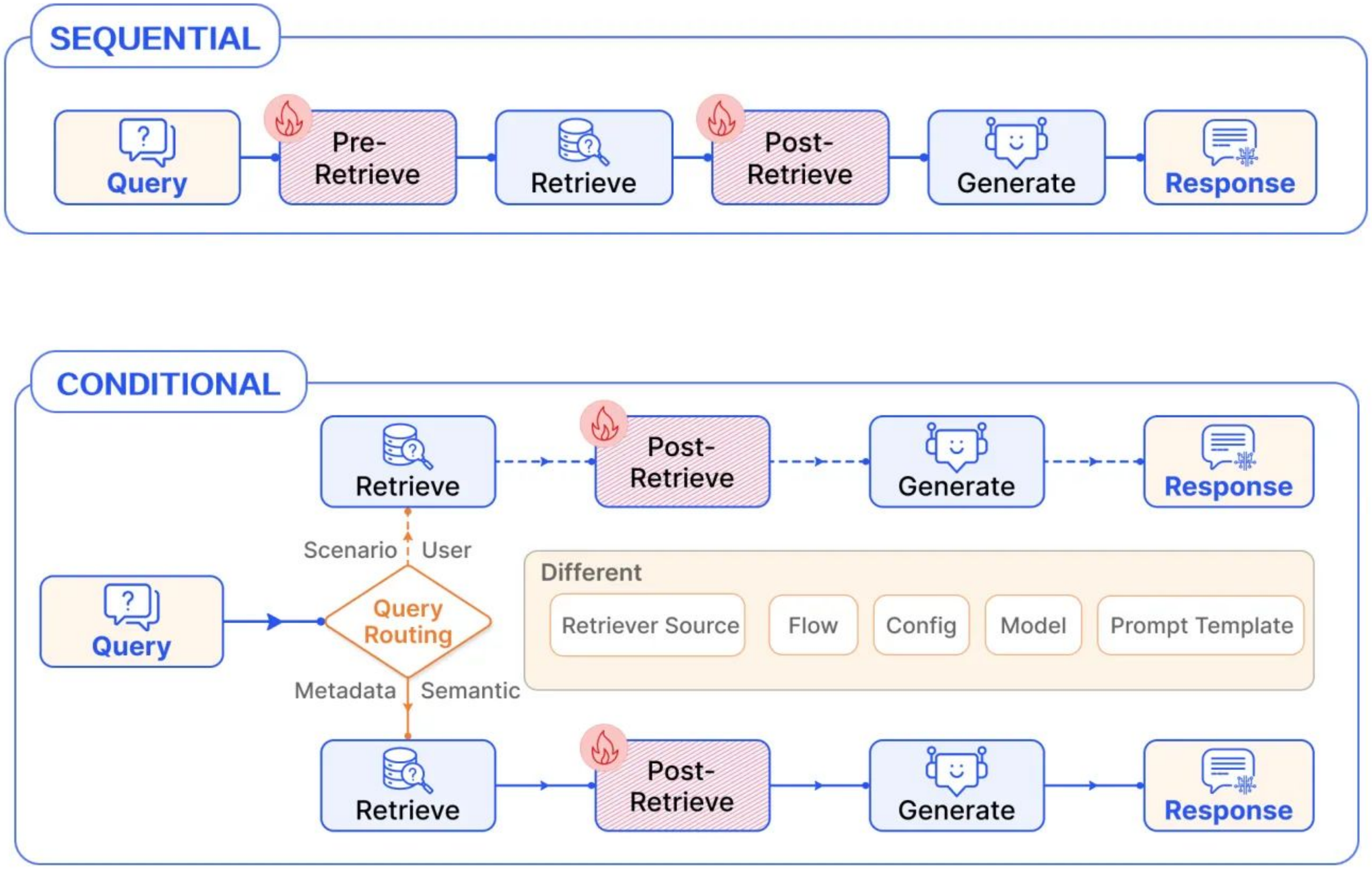
Solution: Modular RAG

- Modular RAG architecture that treats retrieval and generation as a flexible system of interchangeable, specialized modules rather than a fixed pipeline, enabling dynamic routing, iterative processing, and easier experimentation.

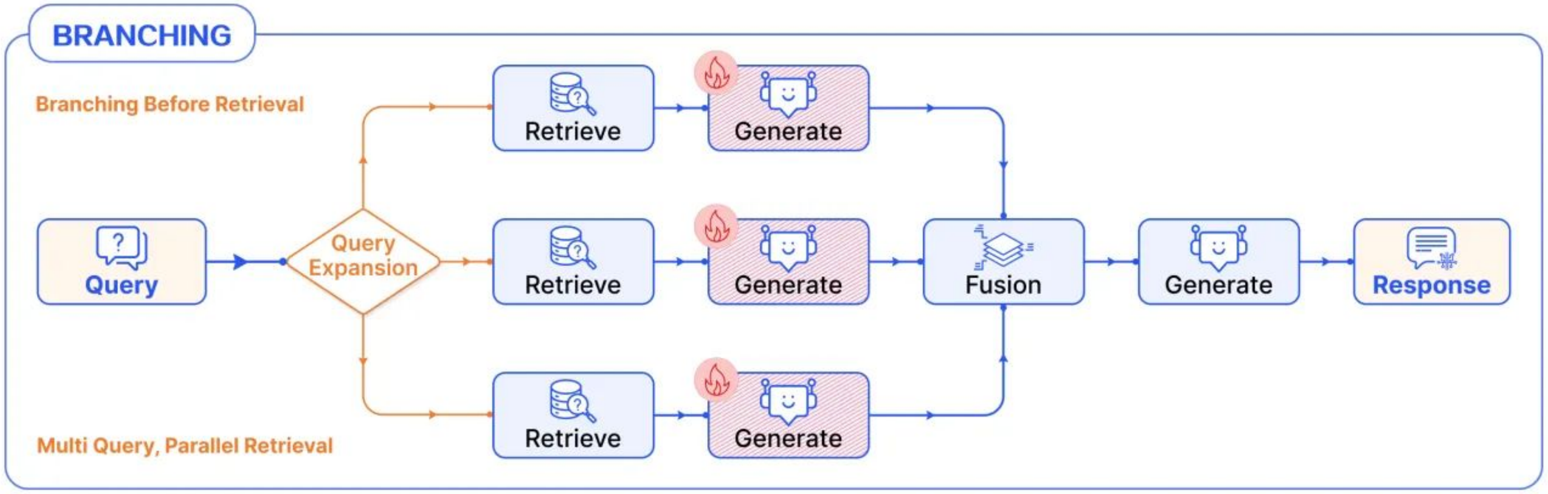
- Modular RAG provides flexibility and scalability to support a variety of RAG flow patterns.

    1. Linear Flow
    2. The Conditional Pattern
    3. The Branching Pattern
    4. The Looping Pattern

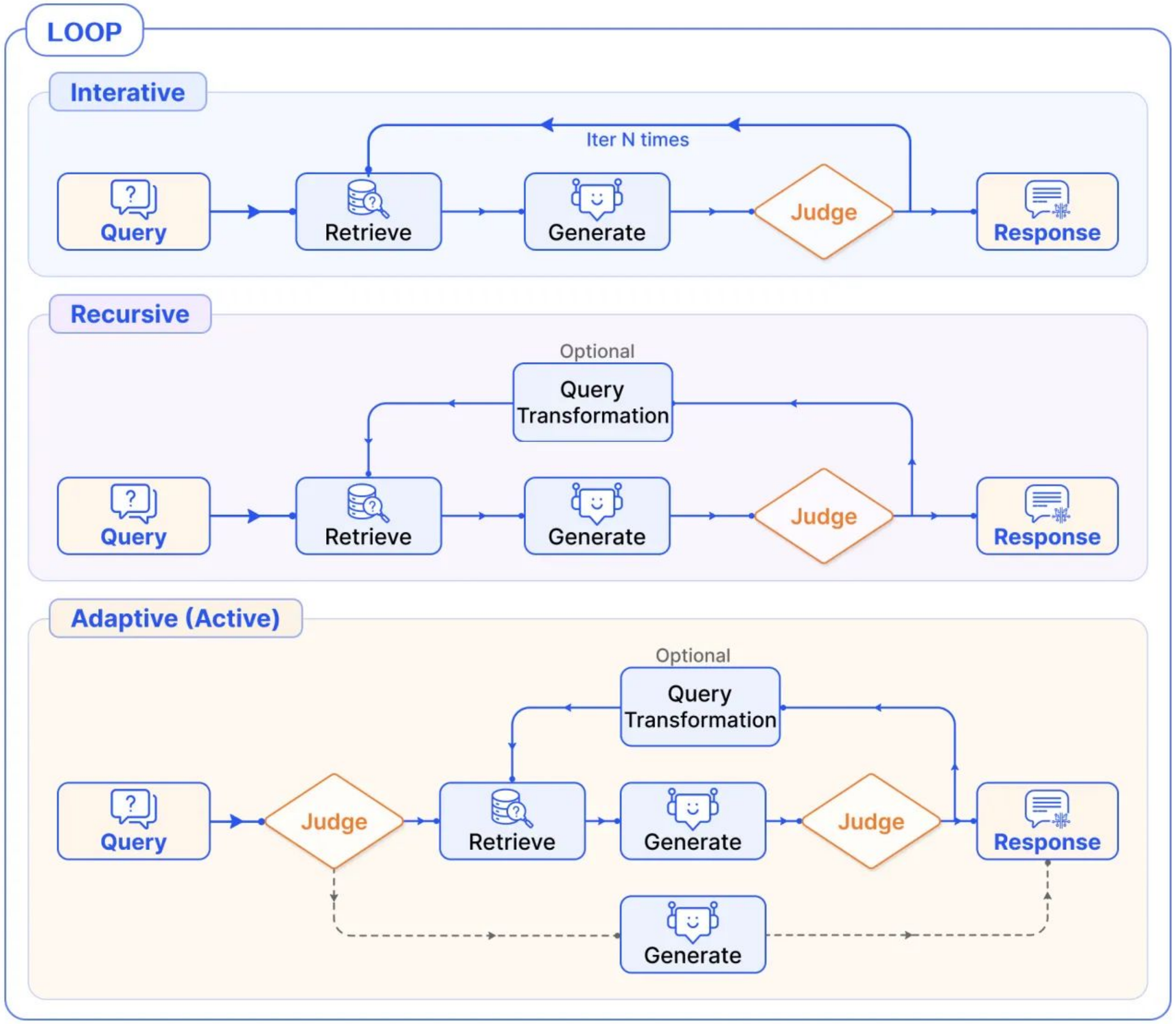
Linear Flow and Conditional Pattern



Branching Pattern



Looping Pattern



Evaluation

Critical for ensuring that RAG systems meet performance goals and provide accurate information.

RAG combines a **Retriever** (searching for relevant documents) and a **Generator** (producing coherent responses) for improved NLP tasks.

Evaluating the pipeline involves assessing both the retrieval and generation components.

Key Metrics

Retrieval Evaluation:

- **Hit Rate**: Hit Rate refers to the proportion of times a relevant item appears in the retrieved set
- **Top-K Precision**: Are the best answers retrieved within the top K results?

Generation Evaluation:

- **Fluency**: How natural and coherent is the generated response?
- **Relevance**: Ensures the generated output is contextually relevant to the query.
- **Factuality**: Measures correctness of the generated content.
- **ROUGE Score**: Evaluates recall-based metrics, especially for summarization tasks.

RAG Evaluation Framework

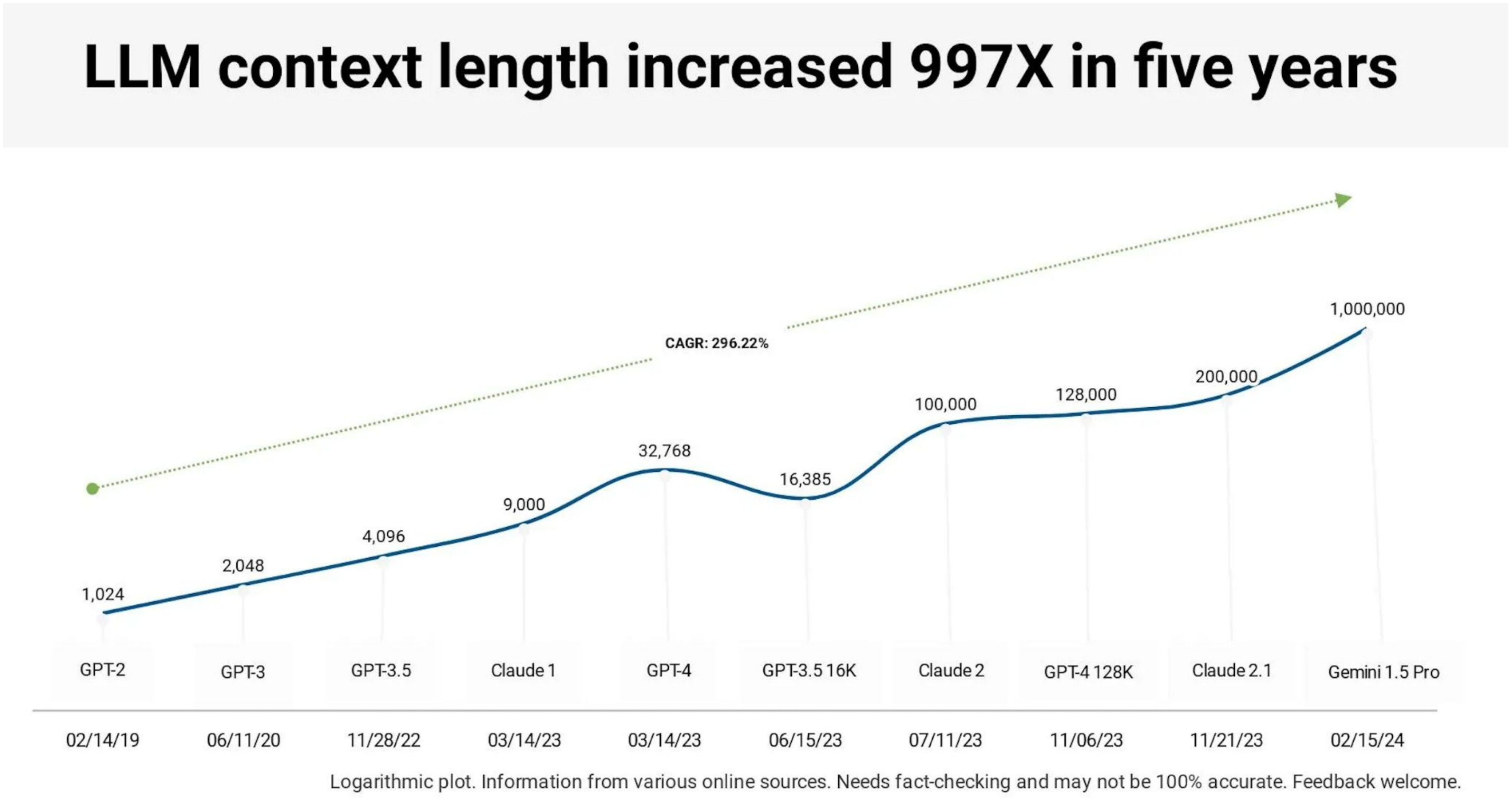
RAGAS (Retrieval Augmented Generation Assessment)
a framework for reference-free evaluation of Retrieval Augmented Generation (RAG) pipelines

- **Faithfulness:** Answer should be grounded in the given context
  - LLM breaks down the answers into atomic statements and validates if those statements exists in the context

- **Answer Relevance**: Generated answer should address the actual question that was provided
  - LLM generate several potential questions against answers and computes the semantic similarity of those questions with actual questions

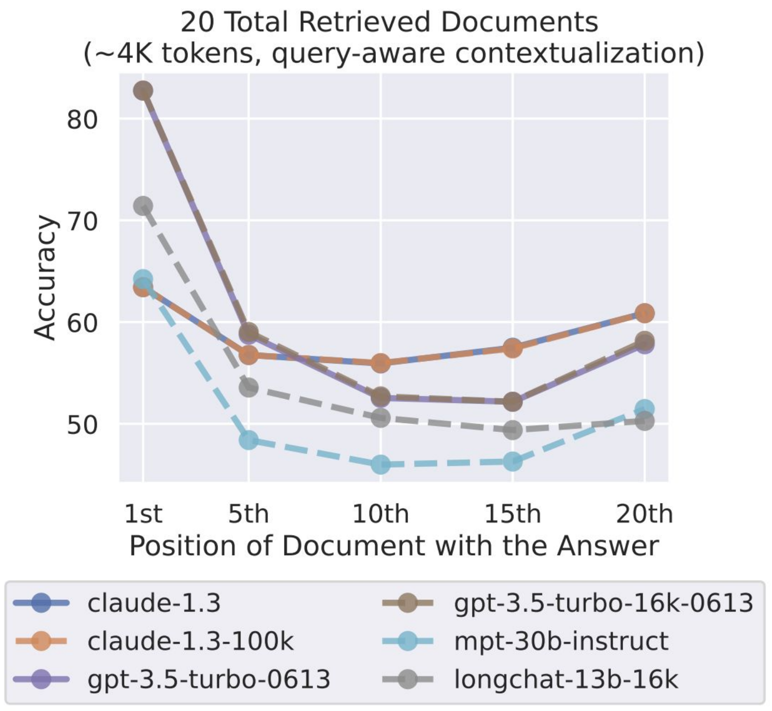
- **Context Relevance:** Retrieved context should be focused, containing as little irrelevant information as possible
  - LLM evaluates the chunks of context and evaluates how many contexts are relevant and required for the answer

Is RAG Dead?

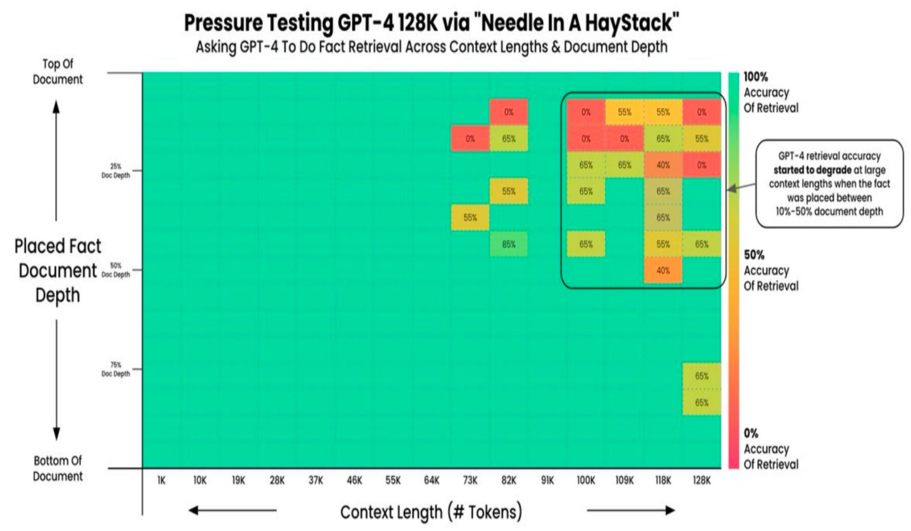

LLM context length increased 997X in five years
CAGR: 296.22%
1,024
2,048
4,096
9,000
32,768
16,385
100,000
128,000
200,000
1,000,000
GPT-2
GPT-3
GPT-3.5
Claude 1
GPT-4
GPT-3.5 16K
Claude 2
GPT-4 128K
Claude 2.1
Gemini 1.5 Pro
02/14/19
06/11/20
11/28/22
03/14/23
03/14/23
06/15/23
07/11/23
11/06/23
11/21/23
02/15/24
Logarithmic plot. Information from various online sources. Needs fact-checking and may not be 100% accurate. Feedback welcome.

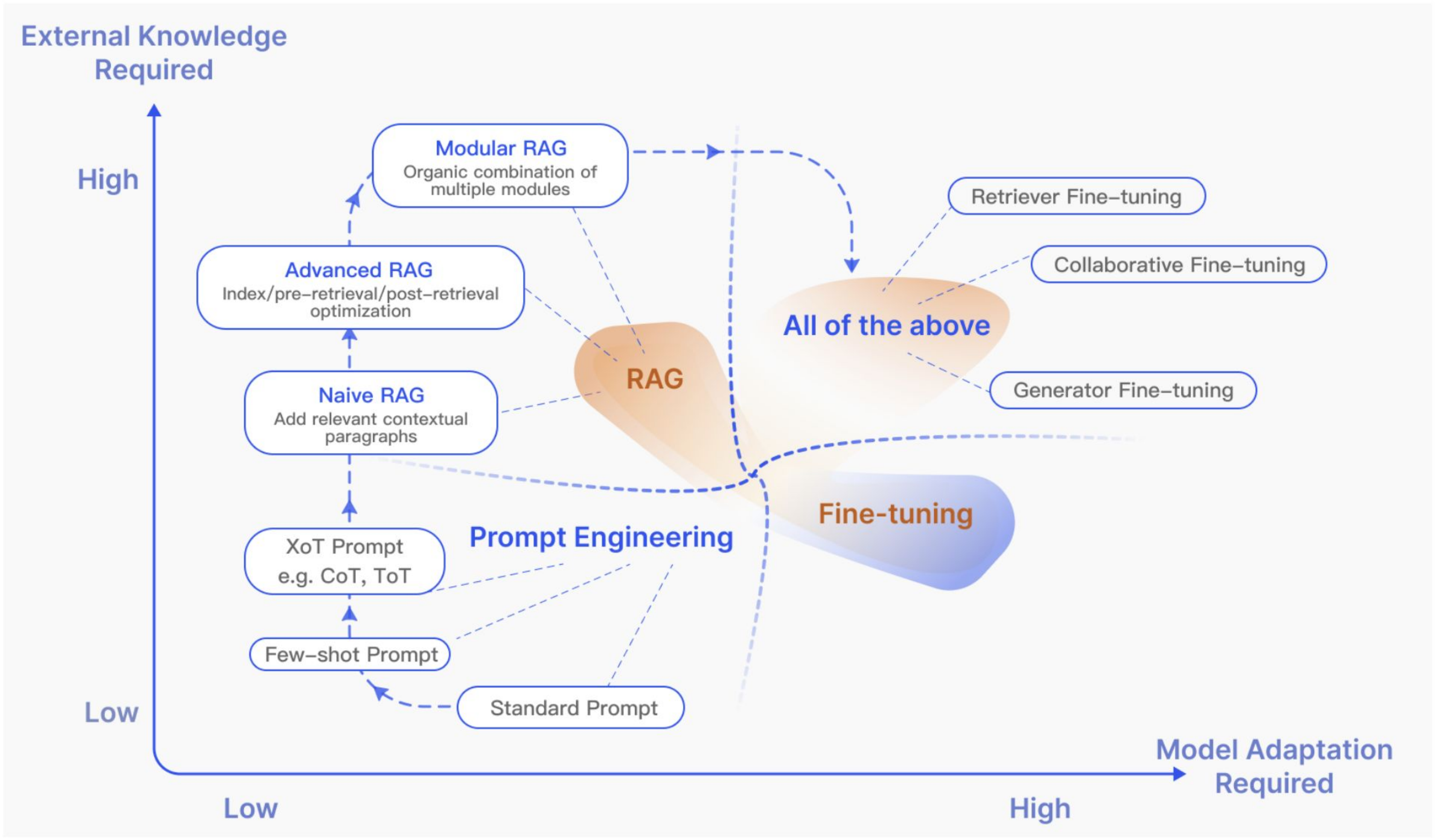
Is RAG Dead?

2023: Lost in the Middle



2024: Needle in haystack




External Knowledge Required
High
Low
Modular RAG
Organic combination of multiple modules
Advanced RAG
Index/pre-retrieval/post-retrieval optimization
Naive RAG
Add relevant contextual paragraphs
XoT Prompt
e.g. CoT, ToT
Few-shot Prompt
Standard Prompt
Prompt Engineering
RAG
All of the above
Fine-tuning
Retriever Fine-tuning
Collaborative Fine-tuning
Generator Fine-tuning
Model Adaptation Required
Low
High

Categorization of Large Language Model (LLM) performance techniques by the degree of External Knowledge Required (RAG/Knowledge Base) versus Model Adaptation Required (Prompt Engineering/Fine-tuning).

Relevant Links

Naive RAG -
https://medium.com/@drjulija/what-is-retrieval-augmented-generation-rag-938e4f6e03d1

Advanced RAG Techniques:
https://medium.com/@krtarunsingh/advanced-rag-techniques-unlocking-the-next-level-040c205b95bc

Modular RAG:
https://medium.com/@sahin.samia/modular-rag-using-llms-what-is-it-and-how-does-it-work-d482ebb3d372

RAG Playground:
https://ragplay.vercel.app/