A SIMPLE GUIDE TO

Retrieval Augmented Generation

Abhinav Kimothi

MEAP

MANNING

MEAP Edition
Manning Early Access Program

A Simple Guide to Retrieval Augmented Generation

Version 4

For more information on this and other Manning titles go to manning.com.

welcome

Thank you for purchasing the MEAP edition of *A Simple Guide to Retrieval Augmented Generation*.

Retrieval Augmented Generation, or RAG, has emerged as a pivotal technique in the realm of applied generative AI, offering reliability and trustworthiness in the outputs from Large Language Models (LLMs). This foundational guide is aimed at enthusiasts, practitioners and leaders who are looking for an easy yet comprehensive introduction to RAG. While prior exposure to the world of machine learning, generative AI and Large Language Models (LLMs) is always helpful, this book is a foundational guide and does not assume that you have a deep understanding of the concepts. You will also gain some perspective of the Large Language Models in the first chapter, itself.

I started working in the Natural Language Processing domain in 2016 and when OpenAI released GPT-3 in 2020, it was nothing short of magical. However, as a practitioner of AI and NLP, the limitations of the technology were evident. Hallucinations and memory limitations emerged to be two big hurdles in the implementation of LLMs. That is when I discovered RAG and it proved to be a game-changer. Since then, I have been tracking, experimenting with and building applications leveraging the advancements in RAG. Like with any new technique, RAG comes with a bit of a learning curve. With this book, I have tried to condensed my learnings of the last few years for you to get a solid foundation of RAG.

RAG, like the entire AI domain, is an evolving technique. With this book you can expect to –

- Develop a solid understanding of RAG fundamentals, the components of a RAG enabled system and its practical applications.
- Know what a non-parametric knowledge base for RAG means and how is it created.
- Gain knowledge about developing a RAG enabled system with details about the indexing pipeline and the generation pipeline.
- Gain deep insights into the evaluation of RAG enabled systems and modularised evaluation strategies
- Familiarize yourself with advanced RAG strategies and the evolving landscape
- Acquire knowledge of available tools, technologies and frameworks for building and deploying production grade RAG systems
- Get an understanding of the current limitations of RAG and an exposure to popular emerging techniques for further exploration

It is needless to express how valuable your feedback will be in shaping this book to be the best guide for you in your pursuit of this novel technique in Generative AI. I request you to please leave your comments, views, questions and complaints in the liveBook Discussion Forum. I will put in my best effort to improve the content.

Thanks again for your interest and for purchasing this MEAP edition.

Cheers,

—Abhinav Kimothi

brief contents

PART 1: FOUNDATIONS

1 Large Language Models and the Need for Retrieval Augmented Generation

2 RAG-enabled Systems and Their Design

PART 2: CREATING RAG ENABLED SYSTEMS

3 Indexing Pipeline: Creating a Knowledge Base for RAG-based Applications

4 Generation Pipeline: Generating Contextual LLM Responses

5 RAG Evaluation: Accuracy, Relevance, Faithfulness

PART 3: RAG IN PRODUCTION

6 Progression of RAG Systems: Naïve to Advanced, and Modular RAG

7 Evolving RAGOps Stack: Technologies that make RAG possible

PART 4: ADDITIONAL CONSIDERATIONS

8 Comparison with Fine-tuning, Multi-Modal and Agentic RAG

9 Cutting Edge: Areas of further exploration

Appendix A. LLM Lifecycle

Appendix B. LangChain & LlamaIndex

Appendix C. Caching & Guardrailing

1 Large Language Models and the Need for Retrieval Augmented Generation

This chapter covers

- What is Retrieval Augmented Generation?
- What are Large Language Models and How are they used?
- The challenges with Large Language Models and the need for RAG
- Popular use cases of RAG

In a short time, Large Language Models have found a wide applicability in modern language processing tasks and even paved the way for autonomous AI agents. There are high chances that you've heard about, if not personally used, ChatGPT, Claude, Bard and others. ChatGPT and the likes are powered by a generative AI technique called Large Language Models. Retrieval Augmented Generation, or RAG, plays a pivotal role in the application of Large Language Models by enhancing their memory and recall.

This book aims to demystify the idea and application of Retrieval Augmented Generation. Over the course of this book, you will be presented with the definition, the design, implementation, evaluation, and the evolution of this technique.

To kick things off, in this chapter, we will introduce the concepts behind Retrieval Augmented Generation and investigate its pressing need with the help of some examples. We will also take a brief look at Large Language Models and how one can interact with them. We will, further, discuss the challenges inherent to Large Language Models, how Retrieval Augmented Generation overcomes these challenges and, at the end, list down a few use cases that have been enabled by this technique.

By the end of this chapter, you will have gained a foundational knowledge to be ready for a deeper exploration of the components of a RAG-enabled system.

By the end of this chapter, you should –

- Have a strong hold on the definition of Retrieval Augmented Generation.
- Develop a basic level of familiarity with Large Language Models.
- Be able to appreciate the limitation of LLMs and the need for RAG.
- Be equipped to dive into the components of a RAG enabled system.

1.1 What is RAG?

Large Language Models, or LLMs, is a generative AI technology that has recently gained tremendous popularity. The most common example of the application of an LLM is ChatGPT by OpenAI. LLMs, like the one powering ChatGPT, have been shown to store knowledge in them. You can ask them questions and they tend to respond with answers that seem correct. However, despite their unprecedented ability to generate text, their responses are not always correct. Upon more careful observation, you may notice that LLM responses are plagued with sub-optimal information and inherent memory limitations. Retrieval Augmented Generation, or RAG, addresses these limitations of LLMs by providing them with information external to these models. Thereby, resulting in LLM responses that are more reliable and trustworthy.

To understand the basic concept of RAG, we will use a simple example. Those familiar with the wonderful sport of Cricket will recall that the Men's ODI Cricket World Cup tournament was held in 2023. The Australian cricket team emerged as the winner. Now, imagine you are interacting with ChatGPT, and you ask it a question, say, *"Who won the 2023 Cricket World Cup?"*. You are, in truth, interacting with GPT-3.5 or GPT-4, LLMs developed and maintained by OpenAI that power ChatGPT. In the first few sections of this chapter, we will use ChatGPT and LLMs interchangeably for simplicity. So, you ask the question and, most likely, you will observe a response like the one illustrated in figure 1.1 below.

Figure 1.1 ChatGPT response to the question, "Who won the 2023 cricket world cup?" (Variation 1), Source: Screenshot by author of his account on https://chat.openai.com

ChatGPT does not have any memory of the 2023 Cricket World Cup and it tells you to check the information from other sources. This is not ideal but, at least, ChatGPT is honest in its response. The same question asked again might also provide a factually inaccurate result. Look at the following illustration in figure 1.2. ChatGPT, falsely, responds that India was the winner of the tournament.

Figure 1.2 ChatGPT response to the question, "Who won the 2023 cricket world cup?" (Variation 2), Source: Screenshot by author of his account on https://chat.openai.com

This is problematic. Despite not having any memory of the 2023 Cricket World Cup, ChatGPT still generates the answer, in a seemingly confident tone, but does that inaccurately. This is what is called a "hallucination" and this has become a major point of criticism for LLMs.

What can be done to improve the response? The world, of course, has this knowledge about the 2023 Cricket World Cup. A simple Google Search will tell you about the winner of the 2023 Cricket World Cup, if you don't already know it. The Wikipedia article (figure 1.3) on the 2023 Cricket World Cup accurately provides this information in the opening section itself. If only, there was a way to tell the LLM about the 2023 Cricket World Cup.

Figure 1.3 Wikipedia Article on 2023 Cricket World Cup, Source : https://en.wikipedia.org/wiki/2023_Cricket_World_Cup

How can we give this information to ChatGPT? The answer is quite simple. We just add this piece of text to our input query (as seen in figure 1.4).

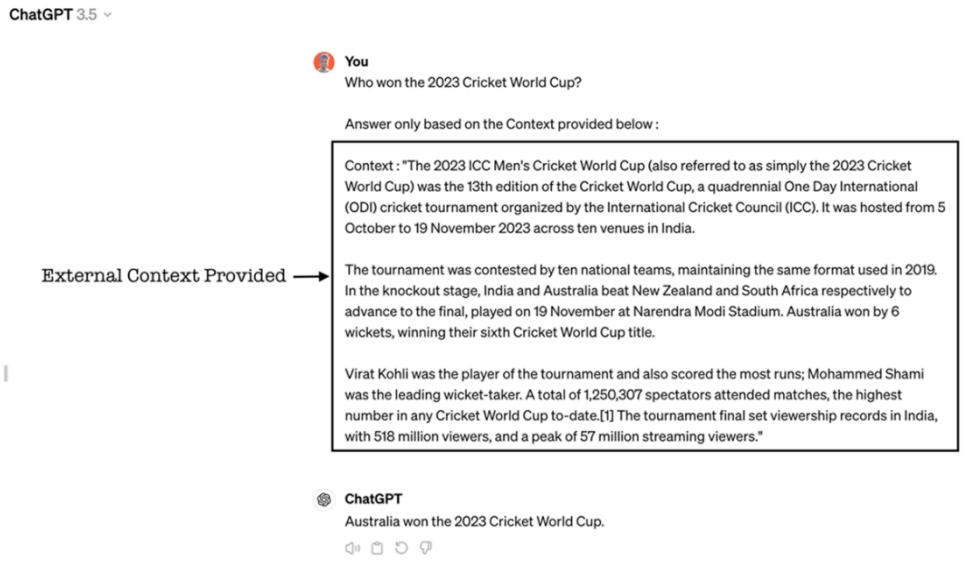

Figure 1.4 ChatGPT response to the question, augmented with external context, Source : Screenshot by author of his account on https://chat.openai.com

And there it is! ChatGPT, now, has responded with the correct answer. It was able to comprehend the piece of additional information we provided, distil the information about the winner of the tournament and respond with a precise and factually accurate answer.

In an oversimplified manner, this example illustrates the basic concept of Retrieval Augmented Generation. Let us look back at what we did here. We understood that the question is about the winner of the 2023 Cricket World Cup. We searched for information about the question and identified Wikipedia as a source of information. We then copied that information and passed it onto ChatGPT, and the LLM powering it, along with the original question. In a way, we added to ChatGPT's knowledge. Retrieval Augmented Generation, as a technique, does the same thing programmatically. It overcomes the limitations of LLMs by providing them with previously unknown information and, as a result, enhances the overall memory of the system.

As the name implies, Retrieval Augmented Generation, in three steps -

- **Retrieves** relevant information from a data source external to the LLMs (Wikipedia, in our example)
- **Augments** the input to the LLM with that external information
- Then, the LLM **Generates** a more accurate result.

A simple definition for RAG, also illustrated in figure 1.5 below, can therefore be as follows.

> The technique of retrieving relevant information from an external source, augmenting the input to the LLM with that external information, thereby enabling the LLM to generate an accurate response is called Retrieval Augmented Generation

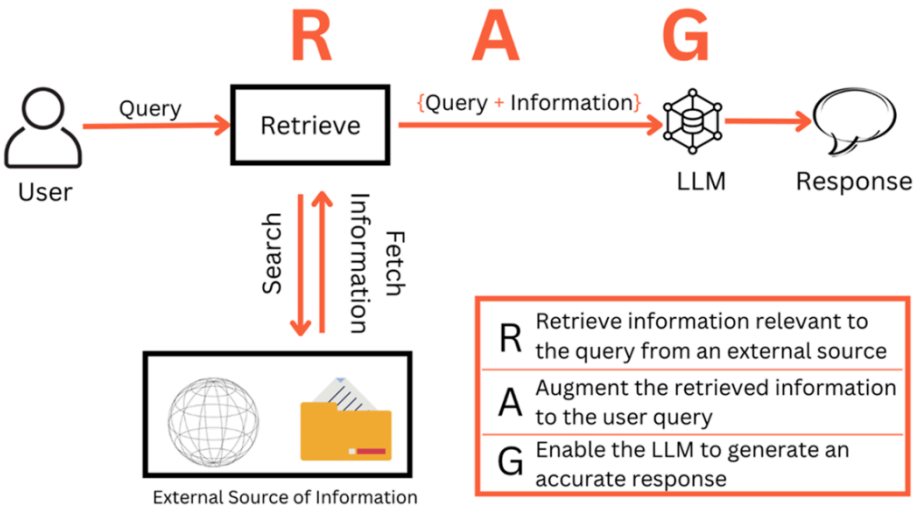

Figure 1.5 Retrieval Augmented Generation: A Simple Definition

The example that we have been looking at so far is an oversimplistic one. We manually searched for the external information and the search was for this one specific question only. In practice, all these processes are automated which allow the system to scale up to a diverse range of queries and data sources. This is what the subsequent chapters in the book will cover. But, before that, a brief understanding what LLMs are and how they can be leveraged will be helpful. We will understand what LLMs are, how they generate text and the concept of prompts. In case you are already familiar with these, you can skip this section and move to the next one.

1.2 What are Large Language Models?

30th November 2022 will be remembered as a watershed moment in the field of artificial intelligence. OpenAI released ChatGPT and the world was mesmerized. Interest in previously obscure terms like Generative AI and Large Language Models (LLMs), skyrocketed over the following 12 months (as seen in figure 1.6).

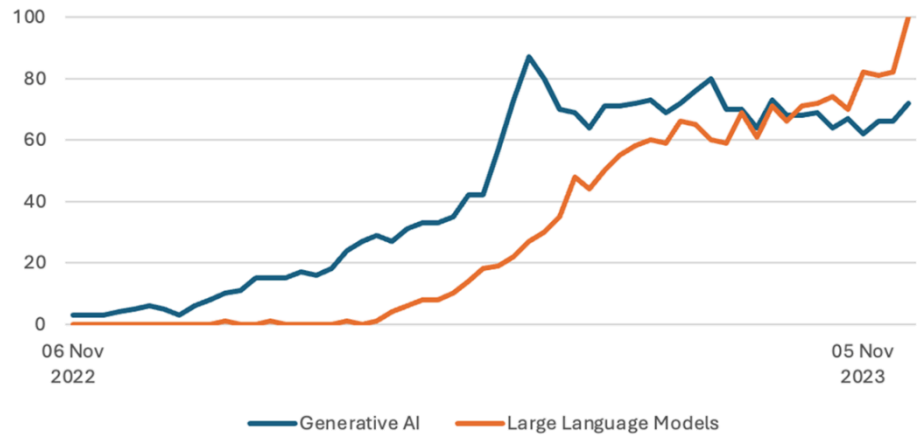

Figure 1.6 Google Trends of "Generative AI" and "Large Language Models" from Nov '22 to Nov '23

Generative AI, and Large Language Models (LLMs) specifically, is a general-purpose technology that is useful for a variety of applications. LLMs can be, generally, thought of as a next token (loosely, next word) prediction model. They are machine learning models that have learned from massive datasets of human-generated text, finding statistical patterns to replicate human-like language abilities.

Very simplistically, think of the model first being shown a sentence like "The teacher teaches the student" for training. Then we hide the last few words of this sentence, "The teacher _____ and ask the model what the next word should be. The model should learn to predict "teaches" as the next word, "the" as the word after that and so on. There are various methods of teaching the model like causal language modeling (CLM), masked language modeling (MLM), etc. Figure 1.7 show the idea behind these two techniques.

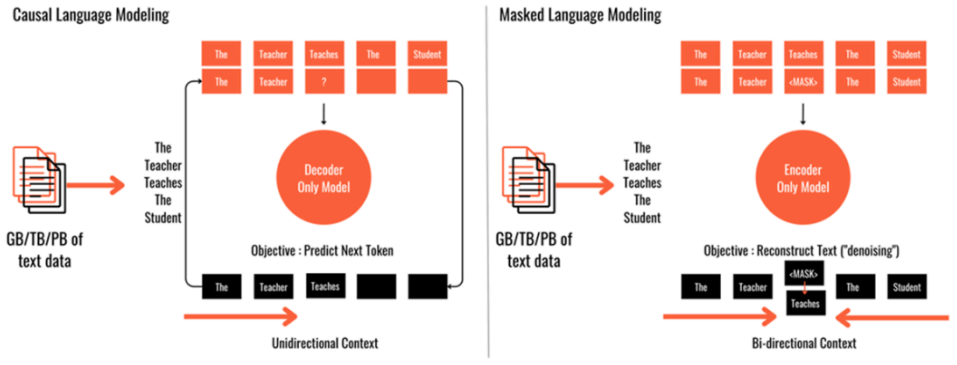

Figure 1.7 Two token prediction techniques – Causal Language Model & Masked Language Model

The training data can have billions of sentences of different kinds. The next token (or word) is chosen from a probability distribution observed in the training data. There are different means and method to choose the next token from the ones for which a probability has been calculated. In a crude manner, you can assume that a probability is calculated for all the words in the vocabulary and one amongst the high probability words is selected. For our example, "The teacher ____", figure 1.8 shows an illustration of the probability distribution. The word "teaches" is selected because it has the highest probability. Other words could also have been selected.

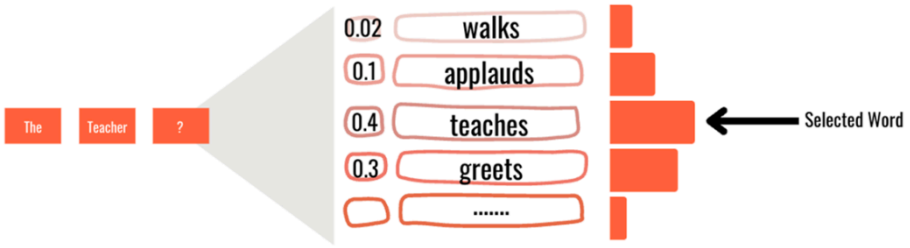

Figure 1.8 Illustrative probability distribution of words after "The Teacher"

From a deeper technical perspective, Large Language Models have been made possible by a simple network architecture based on attention mechanism known as 'transformers'. Prior to the introduction of transformers, tasks like language generation were accomplished using complex recurrent (RNNs) or convolutional neural networks (CNNs) in an encoder-decoder configuration. In their 2017 paper titled Attention Is All You Need (https://arxiv.org/abs/1706.03762), Vasvani et al, a part of the team at Google Research, introduced the transformers architecture and demonstrated remarkable efficacy in language translation tasks (Figure 1.9).

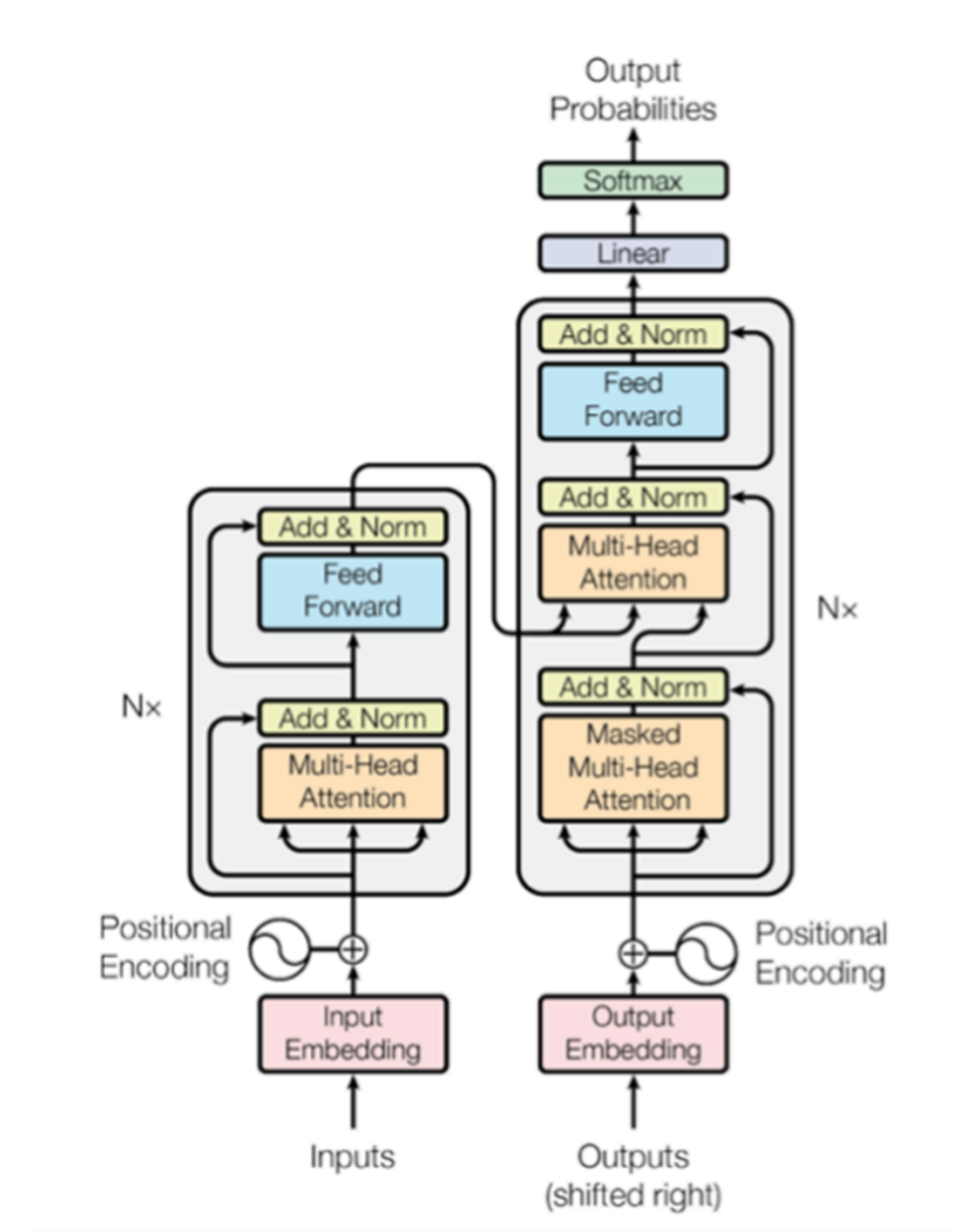

Figure 1.9 Transformer Architecture, Source: Attention is all you need, Vasvani et al.

The nuances of the transformers architecture and building LLMs from scratch is a wide area of study. In some use cases, building an LLM from scratch may be warranted but most applications rely on LLMs that have already been trained and available in the public domain. These models are called foundation models (or pretrained LLMs, or base models). They have been trained on trillions of words for weeks or months using extensive compute power.

WHEN IS TRAINING AN LLM FROM SCRATCH IS ADVISED? Generally available foundation models are mostly trained in commonly understood language. Public data available on the open internet is one of the major sources of data. Therefore, if your use case is in a domain where the vocabulary and the syntax of the language is very different from commonly spoken language then chances are that the available LLMs may not yield optimal results. Domains like healthcare prescription data where the vocabulary is very specific or legal domain where the meaning of words is very different from common language may require collection of domain specific data and training a language model.

The GPT (GPT 3.5, GPT 4) series of LLMs by OpenAI, Claude 3 and it's variants released by Anthropic, the Gemini series of models by Google AI, Command R/R+ by Cohere, as well as, open source models like Llama 2, Llama 3 by Meta AI, Mixtral by Mistral, Gemma 2, again, by Google AI are some popular foundation LLMs (as of April 2024) that are being used in a wide variety of AI powered applications.

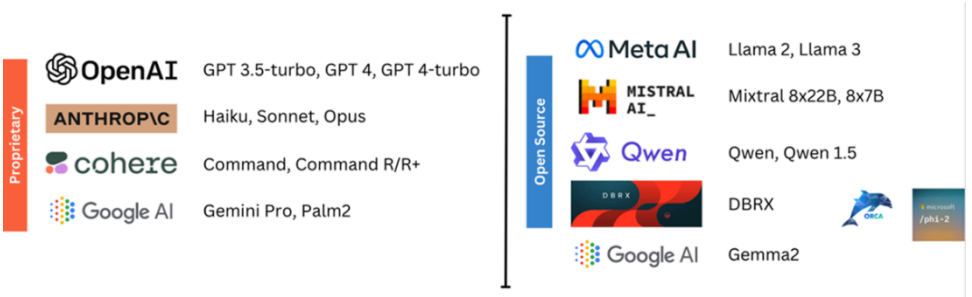

Figure 1.10 Popular proprietary and open source LLMs as of April 2024 (non-exhaustive list)

WHAT ARE MODEL PARAMETERS? You may have heard that Large Language Models have billions and trillions of parameters. GPT-4 has 1.76 trillion parameters. Meta's Llama models come in three different sizes and are denoted as 7B, 13B and 70B models. These are nothing but the number of parameters. So, what exactly are parameters? All machine learning models including LLMs are mathematical models of the form y=f(x) where y is the model and x are the features of the training data. Imagine an equation where $y= w + b_1x_1 + b_2x_2 + b_3x_3 +.... + b_nX_n$. Here w, b_1, b_2,...b_n are the values that the model adjusts or learns during training. These values are called model parameters. The larger the number of parameters, the bigger is the size of the model and the more computational resources are required. On the other hand, a large sized model is expected to perform better.

Large Language Models have found applicability in a wide variety of tasks because of their language understanding and text generation capabilities. Some of the application areas are -

- **Writing** – Generating pieces of content like blogs, reports, articles, posts, tweets etc.
- **Summarization** - Shortening long documents into a meaningful shorter length.
- **Language Translation** - Translating text from one language to the other.
- **Code Generation**- Writing code in a programming language given certain instructions.
- **Information Retrieval** - Extract specific information from text like names, locations, sentiment.
- **Classification** – Classifying pieces of text into groups.
- **Conversations**- Like question answering or chat.

LLMs is a rapidly evolving technology. Learning about LLMs and their architecture is a large area of study. Since, this book focusses on leveraging Retrieval Augmented Generation to use the available LLMs we will, therefore, not delve deep into the transformer architecture and the LLM pre-training process. We will, instead, spend some time in knowing how one interacts with the already available pretrained LLMs.

1.2.1 How do you work with Large Language Models?

Interacting with LLMs differs from traditional programming paradigms. Instead of formalized code syntax, you provide natural language (English, French, Hindi, etc.) input to the models. ChatGPT, as a widely popular example of an LLM powered application, demonstrates this. These inputs are called “prompts”. When you pass a prompt to the model, it predicts the next words and generates an output. This output is termed a “completion”. This entire process of passing a prompt to the LLM and receiving a completion is known as “inference”. Figure 1.11 illustrates the inferencing process.

Figure 1.11 Prompt, Completion, and Inference

Prompting an LLM may, at the first glance, seem like a simple task since the medium of prompting is commonly understood language like English. However, there's more nuance to prompting. The discipline of constructing effective prompts is called prompt engineering. Practitioners and researchers have discovered certain aspects of a prompt that assist in getting better responses from an LLM. For example –

- Defining a "Role" for the LLM like "You are a marketer who excels at creating digital marketing campaigns", or "You are a software engineer who is an expert in python" has been demonstrated to increase the quality of responses.
- Giving "examples" within the prompt has emerged to be one of the most effective techniques to guide the LLM responses. This is also known as Few Shot Prompting.
- It has also been observed that giving clear and detailed instructions helps in the adherence to the prompt.

Prompt engineering is an area of active research. Several nuanced prompting methodologies discovered by researchers have demonstrated the ability of LLMs to handle complex tasks. Chain Of Thought (CoT) prompting, Reason and Act (ReAct), Tree of Thought (ToT) and more prompt engineering frameworks are witnessing their use in several AI powered applications. We will refrain from going deeper into the discipline of prompt engineering right now but look at it, in the context of RAG, in chapter 4. However, an understanding of a few basic terms with respect to LLMs will be beneficial.

- **Context Window**: The nature of the underlying architecture puts a limit on the number of tokens that can be passed to the LLM. This limit is called the Context Window. It is a critical component of LLM usage as it will restrict the amount information that can be passed to the model and the amount of words the model will generate.
- **Temperature**: LLM outputs are based on the probability of generated token. Temperature controls the randomness of generation. The higher the temperature, the more random the output will be.
- **Few Shot Prompting**: LLMs have been observed to perform better if they are provided with certain examples of the desired output within the prompt. This technique of providing inputs is called Few Shot Prompting.
- **In-context Learning**: During inference, passing a prompt to an LLM does not alter the underlying model's memory. The model in its responses may still take into account the information that has been augmented in the prompt. This process, in which the model learns new information without changing any of the underlying parameters, is also called in-context learning.
- **Bias and Toxicity**: LLMs are trained on huge volumes of unstructured data. This data comes from various sources (predominantly, the open internet). The model may show favoritism or generate harmful content based on this training data.

- **Supervised Fine Tuning (SFT)**: For some tasks, prompt engineering alone does not yield satisfactory results. In such cases, the model is further trained by providing it with a set of examples. As opposed to in-context learning. This process changes the weights of the underlying model and consequently alters the memory of the model to suit the task at hand. This process is called Supervised Fine Tuning.
- **Small Language Models (SLMs)**: SLMs are like LLMs but with lesser number of trained parameters (therefore called "Small"). They are faster, require less memory and compute, but are not as adaptable and extensible as an LLM. Therefore, used for very specific tasks.

The domain of LLMs is expansive and an area of study in itself. You will come across the concepts like Reinforcement Learning from Human Feedback (RLHF), Parameter Efficient Fine Tuning (PEFT), various model deployment and monitoring techniques. We will discuss these concepts in the context of RAG in upcoming chapters.

LLMs have really captured the imagination of both researchers and practitioners. The world is now largely aware of the massive ability the LLMs hold. But, as is the case with any technology, LLMs also have their own set of limitations. While we touched upon these limitations in the first section, let us look at them in more detail and set the stage up for a deeper exploration of Retrieval Augmented Generation.

1.3 The Curse of the LLMs and the novelty of RAG

We discussed previously how ChatGPT got quite popular very soon. It became the fastest app ever to reach a million users. The usage exploded in a matter of days and, so did the expectations. Many users started using ChatGPT as a source of information, like an alternative to Google Search. They looked at LLMs for knowledge and wisdom, yet LLMs, as we now know, are just sophisticated predictors of what word comes next.

As a result, the users also started encountering prominent weaknesses of these models. There were questions around copyright, privacy, security, etc. But people also experienced the more concerning limitations of Large Language Models that raised questions around the general adoption and value of the technology.

KNOWLEDGE CUT-OFF DATE

Training an LLM is an expensive and time-consuming process. It takes massive volumes of data and several weeks, or even months, to train an LLM. The data that LLMs are trained on is therefore not always up to the current. e.g. The latest GPT4 Turbo model released by OpenAI on 9th April, 2024 has knowledge only till December 2023. Any event that happened after this knowledge cut-off date, the information of that event is not available to the model.

HALLUCINATIONS

Often, it is observed that LLMs provide responses that are factually incorrect (We saw this in the 2023 Cricket World Cup example at the beginning of this chapter). Despite being factually incorrect, the LLM responses sound extremely confident and legitimate. This characteristic of "lying with confidence", called hallucinations, has proved to be one of the biggest criticisms of LLMs.

KNOWLEDGE LIMITATION

LLMs, as we already read, have been trained on large volumes of data sourced from a variety of sources including the open internet. They do not have any knowledge of information that is not public. The LLMs have not been trained on non-public information like internal company documents, customer information, product documents, etc. So, LLMs cannot be expected to respond to any query about them.

These limitations are inherent to the nature of LLMs and their training process. While the weaknesses of LLMs were being discussed, a parallel discourse around providing additional context or knowledge to the LLMs models started. In essence, it meant creating a ChatGPT like system for proprietary or non-public data with three main objectives.

1. Make LLMs respond with up-to-date information.
2. Make LLMs respond with factually accurate information.
3. Make LLMs aware of proprietary information.

These objectives can be achieved through diverse techniques. A new LLM can be pre-trained from scratch that includes the new data. An existing model can also be fine-tuned with additional data. However, both the approaches require significant amount of data and computation resources. Also, updating the model at a regular frequency with new information is equally costly. In majority of the use-cases, these costs turn out to be prohibitive. Enter, Retrieval Augmented Generation, a cheaper, more effective and dynamic technique to attain the three objectives.

1.3.1 The Discovery of Retrieval Augmented Generation

In May 2020, Lewis et al in their paper, **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks** (https://arxiv.org/abs/2005.11401), explored the recipe for RAG - models which combine pre-trained 'parametric and 'non-parametric' memory for language generation. Let us pay some attention to these terms 'parametric' and 'non-parametric'.

Parameters in machine learning parlance refer to the model weights or variables that the model learns during the training process. In simple terms, they are settings or configurations that the model adjusts in order to perform the assigned task. For language generation, LLMs are trained with billions of parameters (GPT 4 model has 1.76 trillion parameters and the largest Llama 3 model has 80 billion parameters). The ability of an LLM to retain information that it has been trained on is solely reliant on its parameters. It can therefore be said that LLMs store factual information in their parameters. This memory that is internally present in the LLM can be referred to as the parametric memory. This parametric memory is limited. It depends upon the number of parameters and is a factor of the data on which the LLM has been trained on.

Conversely, we can provide information to an LLM that it does not have in its parametric memory. We saw in the example of the Cricket World Cup that when we provided information from an external source to ChatGPT, it was able to get rid of the hallucination. This information that is external to the LLM, but can be provided to the LLM is termed "non-parametric". If we can gather information from external sources as and when desired and use it with the LLM, it forms the "non-parametric" memory of the system. In the aforementioned paper, Lewis et al, stored Wikipedia data and used a retriever to access the information. They demonstrated that this RAG approach outperformed parametric-only baseline in generating more specific, diverse and factual language. We will discuss vector databases and retrievers in chapter 3 and chapter 4.

In 2024, RAG has become one of the most used techniques in the domain of Large Language Models. With the addition of a "non-parametric" memory, the LLM responses are more grounded and factual. Let us discuss the advantages of RAG.

1.3.2 How does RAG help?

With the introduction of 'non-parametric' memory, the LLM does not remain limited to its internal knowledge. We can, at least theoretically, conclude that this non-parametric memory can be extended as much as we want. It can store any volume of proprietary documents or data and have access to all sorts of sources like the intranet and the open internet. In a way, through RAG, we open up the possibility of embellishing the LLM with unlimited knowledge. There will always be some effort required to create this non-parametric memory or the knowledge base and we will look at it in detail later. Chapter 3 in this book is dedicated to the creation of the non-parametric knowledge base.

As a consequence of overcoming the challenge of limited parametric memory, RAG also builds user confidence in the LLM responses.

- The added information assists the LLM in generating responses that are contextually appropriate and the users can be relatively more assured. For example, if the non-parametric memory contains information about a particular company's products, users can be assured that the LLM will generate responses about those products from the provided sources and not from elsewhere.
- In addition to being context aware, because the information is being fetched from a known source, these sources can be cited in the response. This makes the responses more reliable since the users have the choice of validating the information from the source.
- With contextual awareness, the tendency of LLM responses to be factually inaccurate is greatly reduced. The LLMs hallucinate less in RAG enabled systems.

We started with a simple definition for RAG at the beginning of this chapter. Let us now try and expand that definition.

> The technique of enhancing the parametric memory of an LLM by creating access to an explicit non-parametric memory, from which a retriever can fetch relevant information, augment that information to the prompt, pass the prompt to an LLM to enable the LLM to generate a response that is contextual, reliable, and factually accurate is called Retrieval Augmented Generation

This definition is illustrated in the figure 1.12 below.

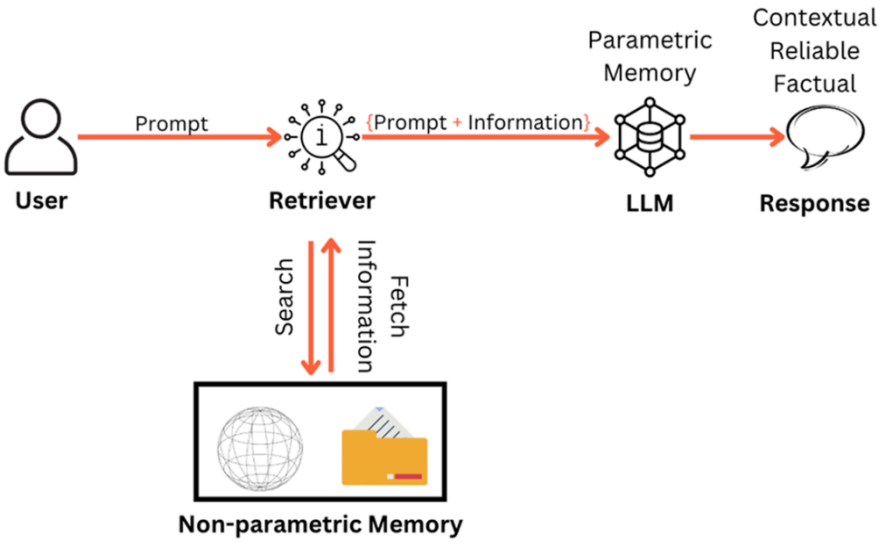

Figure 1.12 RAG enhances the parametric memory of an LLM by creating access to non-parametric memory

Retrieval Augmented Generation has acted as a catalyst in the propagation and acceptance of LLM powered applications. Before concluding this chapter and getting into the design of RAG enabled systems, let us look at some popular use cases where RAG is being adopted.

1.4 Popular RAG use cases

RAG is not just a theoretical concept but a technique that is as popular as the LLM technology itself. Software developers started leveraging language models as soon as Google released BERT in 2018. Today, there are thousands of applications that leverage LLMs to solve language intensive tasks. Whenever you come across an application using LLMs, more often than naught, it will have an internal RAG system in some shape and form. Common applications include –

1. **Search Engine Experience**: Conventional search results are shown as a list of page links ordered by relevance. More recently, Google Search, Perplexity, You have used RAG to present a coherent piece of text, in natural language, with source citation. As a matter of fact, search engine companies are now building LLM first search engines where RAG is the cornerstone of the algorithm.

2. **Personalized Marketing Content Generation**: The widest use of LLMs has probably been in content generation. Using RAG, the content can be personalized to readers, incorporate real-time trends and be contextually appropriate. Yarnit, Jasper, Simplified are some of the platforms that assist in marketing content generation like blogs, emails, social media content, digital advertisements etc.
3. **Real-time Event Commentary**: Imagine an event like a sport or a news event. A retriever can connect to real-time updates/data via APIs and pass this information to the LLM to create a virtual commentator. These can further be augmented with Text-To-Speech models. IBM leveraged technology for commentary during the 2023 US Open Tennis tournament.
4. **Conversational agents**: LLMs can be customized to product/service manuals, domain knowledge, guidelines, etc. using RAG and serve as support agents resolving user complaints and issues. These agents can also route users to more specialized agents depending on the nature of the query. Almost all LLM based chatbots on websites or as internal tools use RAG.
5. **Document Question Answering Systems**: As we have discussed, one of the limitations of LLMs is that they don't have access to proprietary non-public information like product documents, customer profiles etc. specific to an organization. With access to such proprietary documents, a RAG enabled system becomes an intelligent AI system that can answer all questions about the organization.
6. **Virtual Assistants**: Virtual personal assistants like Siri, Alexa and others are in plans to use LLMs to enhance the user's experience. Coupled with more context on user behavior using RAG, these assistants are set to become more personalized.
7. **AI powered research**: AI agents are gaining traction in research intensive fields like law and finance. RAG is being extensively used to retrieve and analyze case law to assist lawyers. A lot of portfolio management companies are introducing RAG enabled systems to analyze scores of documents to research investment opportunities. ESGReveal is a framework developed by researchers at Alibaba Group that employs RAG to extract and evaluate Environmental, Social, and Governance (ESG) data from corporate reports.

This introductory chapter dealt with the concept of Retrieval Augmented Generation. We also got a brief overview of Large Language Models and how one interacts with them. Overcoming the limitations of LLMs, RAG addresses these challenges by providing access to a non-parametric knowledge base to the system. Finally, we looked at some use cases of RAG.

With this foundational understanding of RAG, in the next chapter we will take the first step towards understanding how RAG enabled systems are built by looking at the different components of their design.

1.5 Summary

- RAG enhances the memory of LLMs by creating access to external information.

- LLMs are next word, (or token) prediction models that have been trained on massive amounts of text data to generate human-like text.
- Interaction with LLMs is carried out using natural language prompts and prompt engineering is an important discipline.
- LLMs face challenges of having a knowledge cut-off date and being trained only on public data. They are also prone to generating factually incorrect information (hallucinations).
- RAG overcomes the limitations of the LLMs by incorporating non-parametric memory and increases the context awareness and reliability in the responses.
- Popular use cases of RAG are search engines, document question answering systems, conversational agents, personalized content generation, virtual assistants among others.

2 RAG-enabled Systems and Their Design

This chapter covers

- Concept & Design of a RAG-enabled system
- Overview of the Indexing Pipeline
- Overview of the Generation Pipeline
- Overview of RAG Evaluation
- Overview of LLMOps Service Infrastructure

In the previous chapter, we explored the core principles behind Retrieval Augmented Generation and the challenges faced by Large Language Models that RAG addresses. To construct a RAG-enabled system there are several components that need to be assembled. This includes creation and maintenance of the non-parametric memory, or a knowledge base, for the system. Another pipeline facilitates real-time interaction by sending the prompts to and accepting the response from the LLM, with retrieval and augmentation steps in the middle. Evaluation is yet another critical component, ensuring the effectiveness and accuracy of the system. All these components of the system are supported by a robust service infrastructure.

In this chapter, we will detail out the design of a RAG-enabled system, examining the steps involved and the need for two different pipelines. We will call the pipeline that creates the knowledge base as the indexing pipeline. The other pipeline that allows real time interaction with the LLM will be referred to as the generation pipeline. We will discuss the individual components of these like data loading, embeddings, vector stores, retrievers and more. Additionally, we will get an understanding of how the evaluation of RAG enabled systems is conducted and introduce the service infrastructure that powers such systems.

This chapter is an introduction to various components that will be discussed in detail in the coming chapters. By the end of this chapter, you will have acquired a deep understanding of the components of a RAG-enabled system and be ready to deep dive into the different components.

By the end of the chapter, you should -

- Be able to understand the several components of the RAG system design
- Set yourself up for a deeper exploration of the indexing pipeline, the generation pipelines, RAG evaluation methods and the service infrastructure.

2.1 RAG-enabled Systems

By now, we have come to know that RAG is a vital component of the systems that leverage Large Language Models to solve their use cases. But, how does such a system look like? To illustrate this, we will revisit the example we used at the beginning of Chapter 1 ("Who won the 2023 Cricket World Cup?") and lay out the steps we undertook to enable ChatGPT to provide us with the accurate response.

The initial step was asking the question itself - "Who won the 2023 Cricket World Cup?". Following this, we manually searched for sources on the internet which might have information regarding the answer to the question. We found one (Wikipedia, in our example) and extracted a relevant paragraph from the source. Subsequently, we added the relevant paragraph to our original question, passed the question and the retrieved paragraph, together, in the prompt to ChatGPT and got a factually correct response – "Australia won the 2023 Cricket World Cup".

This can be distilled into a five-step process. Our system needs to facilitate all the five steps.

- **Step 1**: User asks a question to our system
- **Step 2**: The system searches for information relevant to the input question
- **Step 3**: The information relevant to the input question is fetched, or retrieved, and added to the input question
- **Step 4**: This question + information is passed to an LLM
- **Step 5**: The LLM responds with a contextual answer

If you recall, we have already drawn out this process in Chapter 1. Let us visualize it in the context of these five steps as shown in the figure 2.1 below. This workflow will be called the **Generation Pipeline** since it is the one generating the answer.

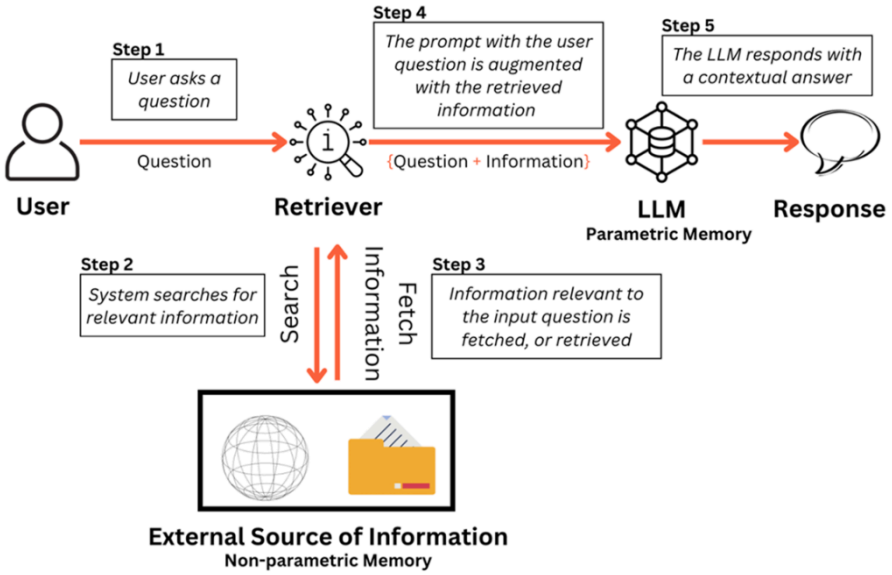

Figure 2.1 Generation Pipeline covering the five steps of Retrieval Augmented Generation

This pipeline is what enables the real-time contextual interaction with the LLM. There are, of course, several intricacies in each of the five steps needed to create the Generation Pipeline. There are decisions that need to be made around the design of retriever and choice of the LLM. The construction of prompts will also affect the quality of the response. These will be discussed down the line. We must, first, address a critical pre-requisite step, before this Generation Pipeline can be put in place. For that, key questions regarding the external source of information need to be answered.

- What is the location of the external source of information?
  - *Is it the open internet? Or are there some documents in the company's internal data storage? Is the information present in some third party databases? Are there multiple sources we want to use.*
  - Why is this important?
    We will need to know, in advance, where to look. We will also need to establish connections to all these disparate sources.

- What is the nature of the information at source?
  - *Are these word documents, pdf files? Is the information accessed via an API and response is in json format? Will we find answers to a question in one document or is the information distributed in multiple documents?*
  - Why is this important?
    We will also need to know the format and nature of data storage to be able to extract the information from the source files.

When data is stored across multiple sources, such as the internet and an internal data lake, the system must connect to each source, search for relevant information in various formats, and organize it according to the original query. Every time a question is asked, this process of connecting, extracting, and parsing will have to be repeated. Information from different sources may lead to factual inconsistencies which will have to be resolved in real time. Searching through all the information might be prohibitively time consuming. This will, therefore, prove to be a highly sub-optimal, unscalable process that may not yield the desired results. A RAG enabled system will work best if the information from different sources –

- Collected in a single location.
- Stored in a single format.
- Broken down into small pieces of information.

The need for a consolidated knowledge base arises from the disparate nature of external data sources. To address this, we need to undertake a series of steps to create and maintain a well-structured knowledge base. This again is a five step process as shown below.

- **Step 1 :** Connect to previously identified external sources
- **Step 2 :** Extract documents and parse text from these documents
- **Step 3 :** Break down long pieces of text into smaller manageable pieces
- **Step 4 :** Convert these small pieces into a suitable format
- **Step 5 :** Store this information

These steps that facilitate the creation of this knowledge base form the **Indexing Pipeline.** Indexing pipeline is shown below in figure 2.2.

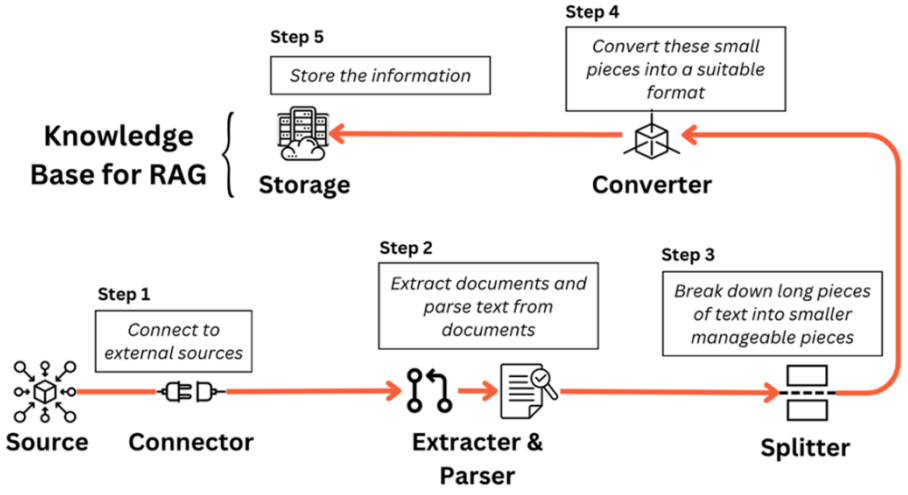

Figure 2.2 Indexing Pipeline covering the steps to create the Knowledge Base for RAG

In addition to creating the knowledge base, the indexing pipeline plays a crucial role in maintaining and updating it with the latest information to ensure its relevance and accuracy. Before the knowledge base is created by the indexing pipeline, there is nowhere for the Generation Pipeline to search information from. It is the indexing pipeline that lays the foundation for the subsequent operation of the Generation Pipeline. Therefore, setting up the Indexing Pipeline comes before the Generation Pipeline can be activated.

Together, these pipelines form the backbone of a RAG-enabled system, enabling seamless interaction with users and delivering contextually relevant responses. The figure 2.3, below shows the indexing and the generation pipeline working together to form the skeleton of a RAG-enabled system.

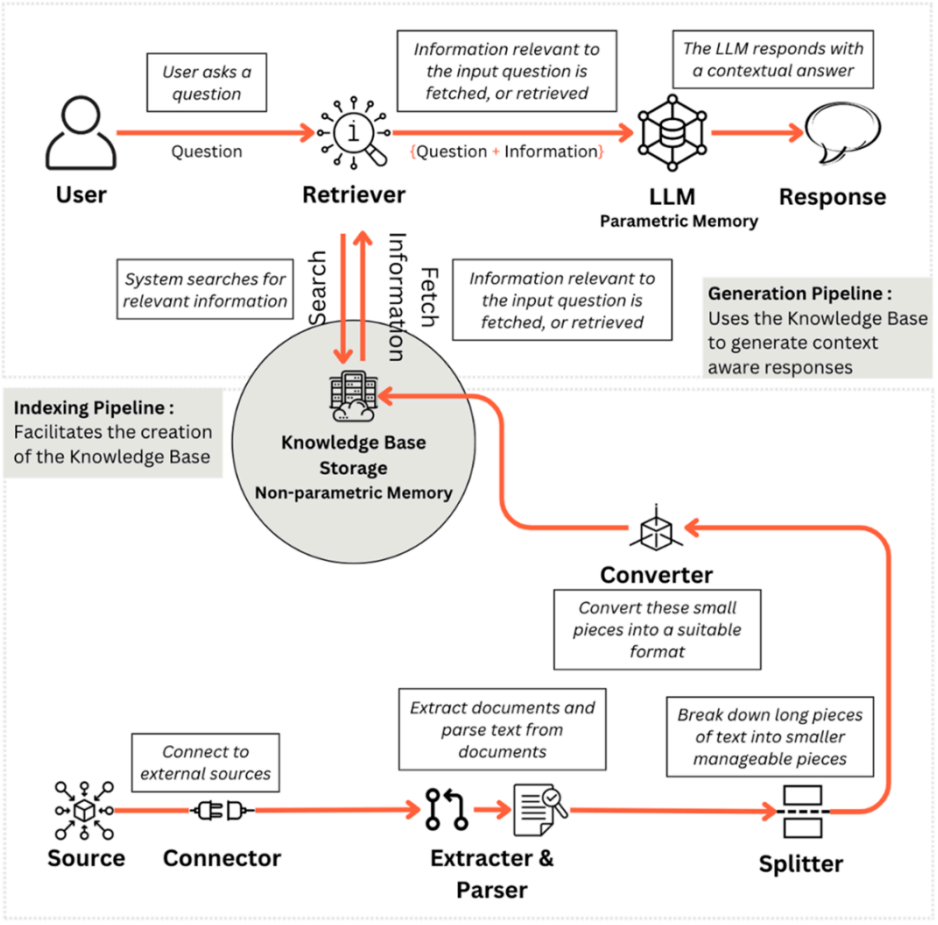

Figure 2.3 The Indexing Pipeline and the Generation Pipeline together make a RAG-enabled system.

We have established the flow of a RAG-enabled system that includes two pipelines. The indexing pipeline is responsible for creation and management of the knowledge base for RAG. The generation pipeline searches for information from the knowledge base to facilitate contextual responses to user queries. Conceptually, this is the complete flow. However, to build such systems to be used in the real world, there are more components that are required. The next section reimagines this flow along with other considerations and creates a design for RAG enabled systems.

2.2 Design of RAG-enabled Systems

We saw above how RAG enabled systems are created by the indexing pipeline and the generation pipeline. These two pipelines themselves include several parts. Like all software applications, production-ready RAG-enabled systems require more than just the basic components. We need to think about accuracy, observability, scalability and other important factors. In this book, we will discuss some of these components at length. In the figure 2.4 below, we will attempt to draw a rough layout of a RAG-enabled system. Apart from the indexing and generation component, we'll add layers for infrastructure, security, evaluation, etc.

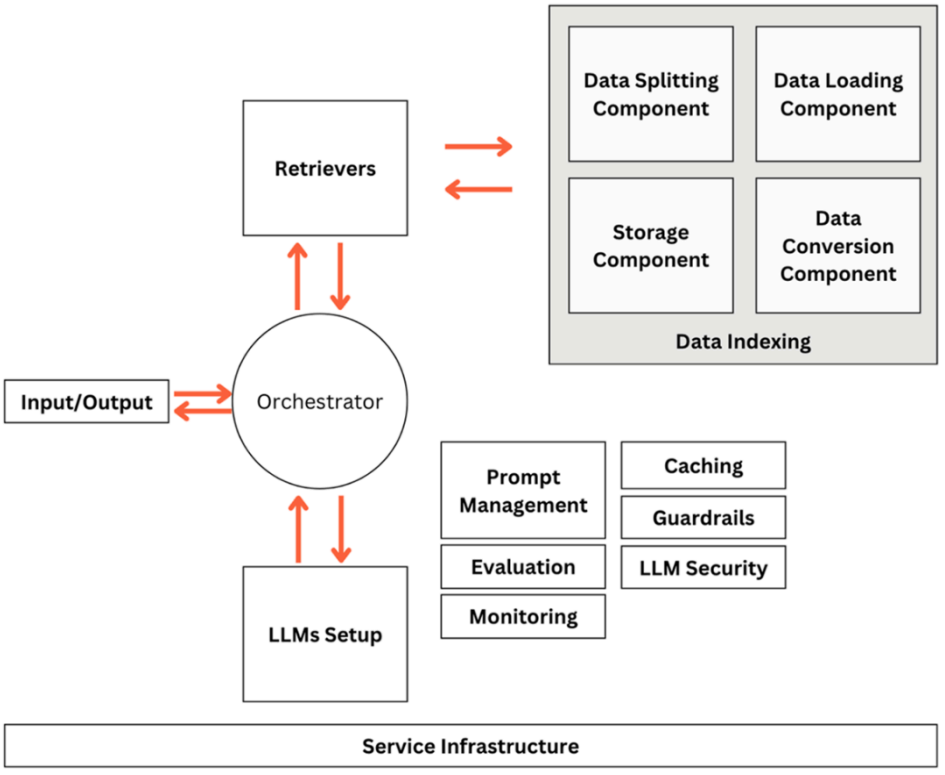

Figure 2.4 Components of a production ready RAG-enabled system

Let us look at the main components of a RAG enabled system -

- **Data Loading** component : connects to external sources, extracts and parses data
- **Data Splitting** component : breaks down large pieces of text into smaller manageable parts
- **Data Conversion** component : converts text data into a more suitable format
- **Storage** component : stores the data to create a knowledge base for the system

These first four components complete the indexing pipeline

- **Retrievers** : are responsible for searching and fetching information from the Storage
- **LLM Setup** : is responsible for generating the response to the input
- **Prompt Management** : enables the augmentation of the retrieved information to the original input

These next three components complete the generation pipeline

- **Evaluation** component : measures the accuracy and reliability of the system before and after deployment
- **Monitoring** : tracks the performance of the RAG-enabled system and helps detect failures
- The **Service Infrastructure** : in addition to facilitating deployment and maintenance, ensures a seamless integration of various system components for optimal performance.

Other components include **caching** which helps store previously generated responses to expedite retrieval for similar queries, **guardrails** to ensure compliance with policy, regulation and social responsibility, and **security** to protect LLMs against breaches like prompt injection, data poisoning etc.

All these components are managed and controlled by a central **orchestration layer** which is responsible for the interaction and sequencing of these components. It provides a unified interface for managing and monitoring the workflows and processes.

The following sections will provide an overview of these components before we have more of a deep dive into them in subsequent chapters.

2.3 Indexing Pipeline

We discussed how the indexing pipeline facilitates the creation of the knowledge base that is used in the real-time generation pipeline. For practical purposes, the indexing pipeline is an offline or asynchronous pipeline. What this means is that the indexing pipeline is not activated in real-time when the user is asking a question – rather, it creates the knowledge base in advance and updates it at pre-defined intervals. The indexing pipeline comprises four main components as seen in the figure 2.5 below.

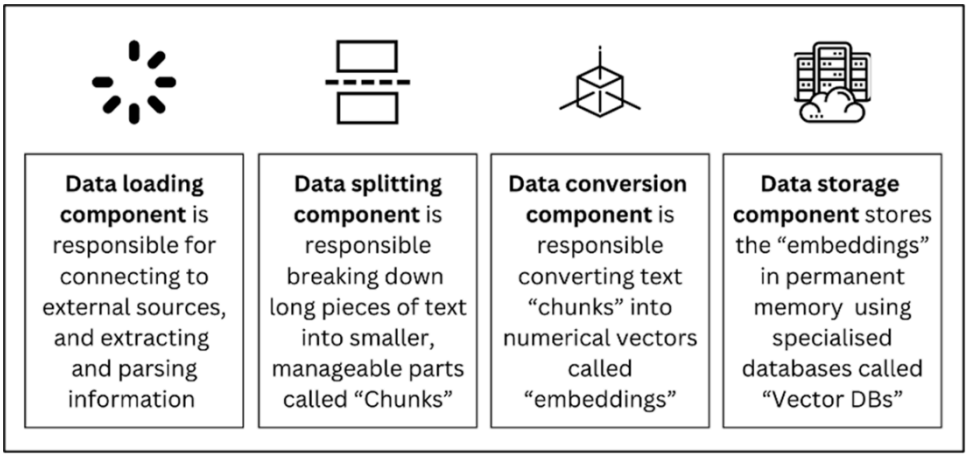

Figure 2.5 Four Components of the indexing pipeline facilitate the creation of the knowledge base

Let us delve deeper into each of these

1. **Data Loading** : This component is responsible for connecting to different sources where data is present, then be able to read the files in these external sources and, finally, extract and parse the text from these files. These external sources can be File Systems, Data Lakes, Content Management Systems, etc.. The files received from the sources can be in various formats like pdf, docs, json, HTML and more. This component, therefore, comprises of several connectors (for different external sources), extractors and parsers (for different file types). In chapter 3, we will look at several examples of such loaders.
2. **Data Splitting (Text Splitting)** : Breaking down text into smaller segments enhances the system's ability to process and analyse information efficiently. These smaller pieces in Natural Language Processing (NLP) parlance are commonly referred to as “chunks”. This process of splitting large text documents into smaller chunks is also called “chunking”. We will discuss the need for chunking and various chunking strategies in chapter 3.
3. **Data Conversion (Embeddings)** : Textual data must be converted into a numerical format for search and retrieval computations in RAG-enabled systems. There are different ways of doing this conversion. For all practical purposes, a data format called “embeddings” works the best for search and retrieval. We will take a look at what embeddings are and the different embeddings models in chapter 3.
4. **Data Storage** : Once the data is ready in the desired format (embeddings) it needs to be stored in persistent (permanent) memory so that the real-time Generation Pipeline can access the data whenever a user asks a question. Data is stored in specialised databases known as “vector databases” which are best-suited for search and retrieval of embeddings. In Chapter 3, will explore various vector databases and factors influencing their suitability for RAG-enabled systems.

DO YOU ALWAYS NEED AN INDEXING PIPELINE?

Offline Indexing pipelines are typically used when a knowledge base with large amount of data is being built for repeated usage e.g. a number of enterprise documents, manuals etc.

However, there are some cases in which the Generation Pipeline connects to a third-party API to receive information related to the user question. For example, imagine an application built for users seeking travel advice based on the weather forecast. An important component of this application will be fetching the weather details for the users' location. Suppose the system uses a third-party API service which can responds with a location's weather details when provided with the location in the input. This weather information is then passed to the LLM to generate the advice.

This application can also be thought of as a RAG-enabled system. But there is a difference. This system has outsourced the search and retrieval operation to the third-party API. It is the third party that maintains the data. For such systems, the indexing pipeline is not required to be built since the search and retrieval happens outside the system. Another example is applications that ask the user for inputting external information, like document summarisers. The search operation here is outsourced to the user.

Therefore, systems that use augment external information to the prompts but do not necessarily search and retrieve information themselves, do not warrant the creation of a knowledge base and therefore do not have an indexing pipeline. Some will argue that such systems are not RAG-enabled systems in the first place.

2.4 Generation Pipeline

Building upon the foundation established by the indexing pipeline, the generation pipeline facilitates real-time interactions in RAG-enabled systems. It is the generation pipeline that facilitates Retrieval, Augmentation and Generation in the system. When a user asks a question, the generation pipeline processes the query, retrieves relevant information, and generates a response—all without the user directly interacting with the underlying indexing pipeline. The generation pipeline is enabled by three components, as seen in figure 2.6 below.

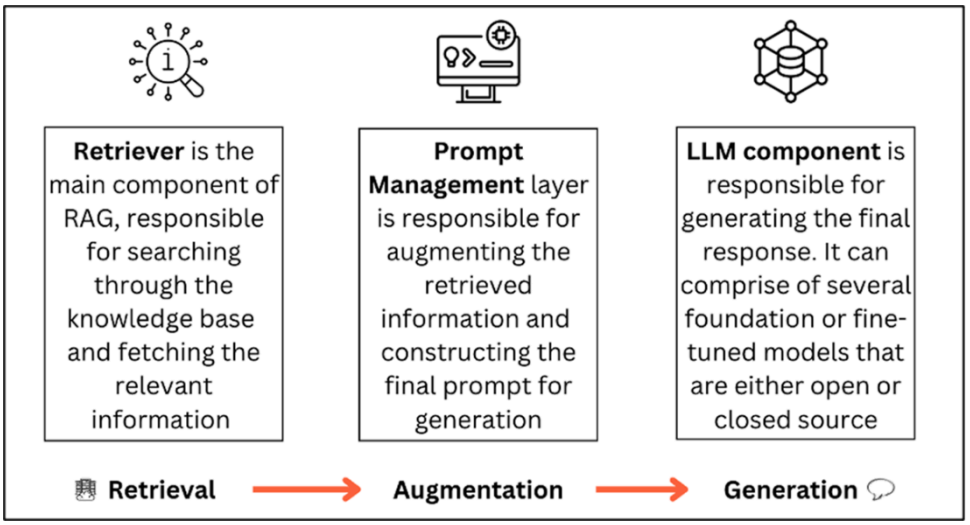

Figure 2.6 Three Components of the generation pipeline enable the real-time query-response process of a RAG enabled system

Let us consider each of these in some more detail

1. **The Retriever** : This is arguably the most critical component of the entire system. Using advanced search algorithms, the retriever scans the knowledge base to identify and retrieve the most relevant information based on the user's query. The overall effectiveness of the entire system relies heavily on the accuracy of the retriever. Also, search is a computationally heavy operation and may take time. Therefore, the retriever, also contributes heavily to the overall latency of the system. We will discuss different retrievers and retrieval strategies in chapter 4.
2. **Prompt Management**: Once the relevant information is retrieved by the retriever, it needs to be combined, or augmented, with the original user query. Now, this may seem a simple task at the first glance. However, the construction of the prompt makes significant difference to the quality of the generated response. This component also falls in the gambit of prompt engineering. We will explore different prompting and prompt management strategies in chapter 4.
3. **LLM Setup**: At the end, Large Language Models (LLMs) are responsible for generating the final response. A RAG enabled system may rely on more than one LLM. The LLMs can be the foundation (base) models that have been pre-trained and generally available either open source, like those by Meta or Mistral, or through a managed service, like OpenAI or Anthropic. LLMs can also be fine-tuned for specific tasks. Fine-tuning involves training pre-existing LLMs on specific datasets or tasks to improve performance and adaptability for specialized applications. In rare cases, the developer may decide to train their own LLMs. We will discuss on LLMs, in depth, in chapter 4.

2.5 Evaluation and Monitoring

Indexing and Generation pipelines complete the system from a usage perspective. With these two pipelines in place, at least in theory, a user can start interacting with the system and get responses. However, with just these two pipelines, we have no measure of the quality of the system. Is the system performing accurately or is it still prone to hallucinations? Is the information that is being fetched by the retriever the most relevant to the query? To answer these questions, we have to put in place an evaluation framework. This framework helps in evaluating the quality of the system before it is released and then for continuous monitoring and improvement.

Building on the advancements of large language models, Retrieval Augmented Generation (RAG) represents a recent innovation in natural language processing. Metrics such as relevance scores, recall, and pricision are commonly used to evaluate the effectiveness of RAG-enabled systems. One framework that intuitively guides a comprehensive evaluation is the triad of RAG metrics proposed by TruEra (https://truera.com/ai-quality-education/generative-ai-rags/what-is-the-rag-triad/). It looks at the RAG evaluation in three dimensions, as you can see in the figure 2.7 below.

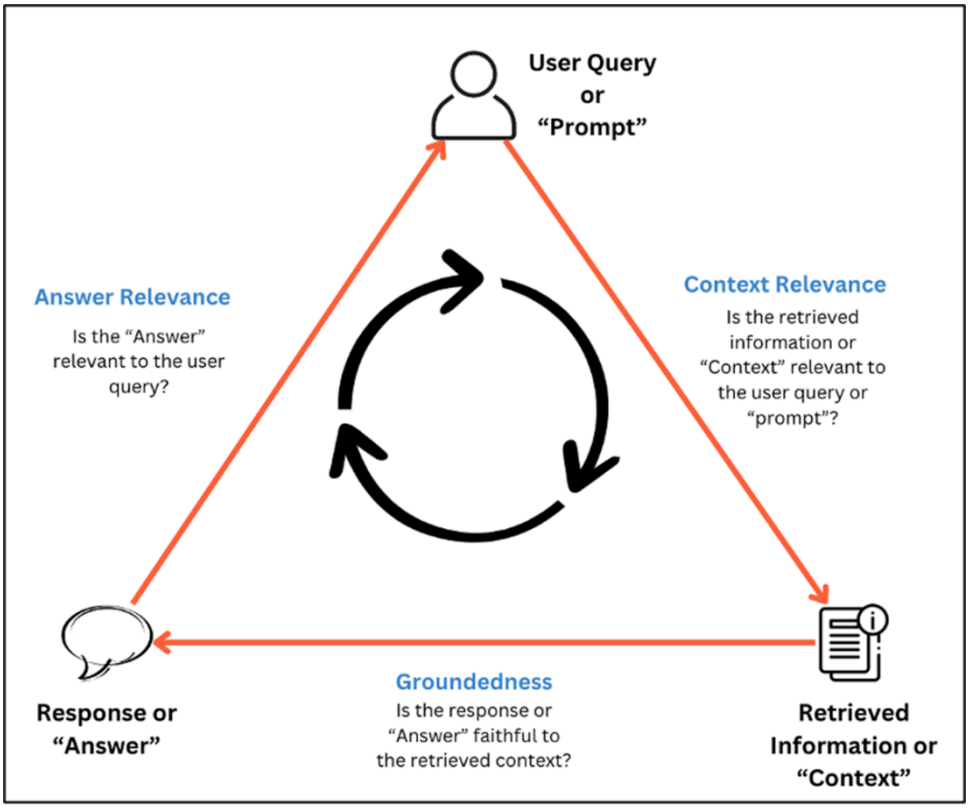

Figure 2.7 The triad of RAG evaluation proposed by TruEra

As you can see, the workflow involved checks in between each of the steps of Prompt, Context, and Answer. Let's take a closer look at each of these.

1. **Between the retrieved information (context) and the user query (prompt)** – Is the information that is being searched and retrieved by the retriever the most relevant to the question that the user has asked? The consequence of irrelevant information being retrieved is that no matter how good the LLM is, if the information being augmented is not good, the response will be sub-optimal
2. **Between the final response (answer) and the retrieved information (context)** – Does the LLM take into account all the retrieved information while generating responses or not? Even though RAG is aimed at reducing hallucinations, the system might still ignore the retrieved information. There are several reasons for it which we will discuss in subsequent chapters.
3. **Between the final response (answer) and the user query (prompt)** – Is the final response in line with the question that the user had originally asked? To assess the overall effectiveness of the system, the relevance of the final response to the original question is necessary.

There are several metrics that help assess each of these three dimensions. For some of the metrics a "ground truth" dataset is warranted. Ground truth datasets provide a benchmark for evaluating the accuracy and effectiveness of RAG-enabled systems by comparing generated responses to manually curated references. We will take a deeper look at these metrics and the ground truth dataset in chapter 5.

Continuous evaluation of metrics during live operation can identify the types of queries the system struggles to answer accurately. Qualitative feedback can also be collected from the user on the generated responses.

2.6 Service Infrastructure

RAG enabled, and LLM based apps, in general, are being powered by an evolving LLMOps stack. Various providers offer infrastructure components such as data storage platforms, model hosting services, and application orchestration frameworks. The infrastructure can be understood in several layers.

1. **Data Layer** : Tools and Platforms used to process and store data in form of embeddings
2. **Model Layer** : Providers of proprietary or open-source LLMs.
3. **Prompt Layer** : Tools offering maintenance and evaluation of prompts
4. **Evaluation Layer** : Tools and frameworks providing evaluation metrics for RAG
5. **App Orchestration** : Frameworks that facilitate invocation of different components of the system
6. **Deployment Layer** : Cloud providers and platforms for deploying RAG enabled LLM based apps
7. **Application Layer** : Hosting services for RAG enabled LLM based apps
8. **Monitoring Layer** : Platforms offering continuous monitoring of RAG enabled LLM based apps

In chapter 6, we will explore the various layers of infrastructure that support RAG-enabled systems.

2.7 Caching, Guardrails and Security

Finally, there are certain other components that are used frequently in RAG enabled systems. These components address the issues of system latency, regulatory and ethical compliances among other aspects.

- **Caching** : Caching is the process in which certain data is stored in cache memory for faster retrieval. LLM caching is slightly different from regular caching. The LLM responses to queries are stored in a semantic cache. Next time, a similar query is asked, the response from the cache is retrieved instead of sending the query through the complete RAG pipeline. This improves the performance of the system by reducing the time it takes to respond, by reducing the cost of LLM inferencing and by reducing the load on the LLM service.
- **Guardrails** : For several use cases, in practice, there will be a set of boundaries within which the output needs to be generated. Guardrails are pre-defined set of rules that are added in the system to comply with policies, regulations and ethical guidelines.
- **Security** : LLMs and RAG enabled LLM based applications have observed new kind of threats like prompt injections, data poisoning, sensitive information disclosure and others. With evolving threats, the security infrastructure also needs to evolve to address concerns around security and data privacy of RAG enabled systems.

This chapter provided an overview of the key components of RAG-enabled systems, including the indexing and generation pipelines, evaluation and monitoring, and service infrastructure. By understanding these components, you are now equipped to delve deeper into each of these components and the intricacies of RAG-enabled systems in subsequent chapters.

2.8 Summary

- A Retrieval Augmented Generation (RAG) enabled system consists of two main pipelines: the Indexing Pipeline and the Generation Pipeline.
- The Indexing Pipeline is responsible for creating and maintaining the knowledge base, which involves data loading, text splitting, data conversion (embeddings), and data storage in a vector database.
- The Generation Pipeline manages real-time interactions by retrieving information, augmenting queries, and generating responses using a Large Language Model (LLM).
- Evaluation and monitoring are crucial components to assess system performance, covering the relevance between the retrieved information and query, the final response and retrieved information, and the final response and the original query.

- The service infrastructure for RAG-enabled systems includes layers for data, models, prompts, evaluation, app orchestration, deployment, application hosting, and monitoring.
- Additional components like caching, guardrails, and security measures are often employed to improve performance, ensure compliance, and address potential threats.

3 Indexing Pipeline: Creating a Knowledge Base for RAG-based Applications

This chapter covers

- The four components of Indexing Pipeline
- Data Loading
- Text Splitting or Chunking
- Converting Text to Embeddings
- Storing Embeddings in Vector Databases
- Examples in Python using LangChain.

In Chapter 2, we discussed the main components of RAG-based system design. You may recall that the Indexing Pipeline creates the knowledge base or the non-parametric memory of RAG-based applications. Indexing Pipeline needs to be set-up before the real-time user interaction with the LLM can begin.

In this chapter, we will elaborate the four components of Indexing Pipeline. We will begin by discussing Data Loading which involves connecting to source, extracting files and parsing text. At this stage, we will introduce a framework called LangChain, that is fast becoming popular in the LLM app developer community. We will then elaborate on the need for data splitting or chunking and discuss chunking strategies. Embeddings is an important design pattern in the world of AI & ML. We will detail out what embeddings are and how they are relevant in the context of RAG. Finally, we will look at a new storage technique called Vector Storage and the databases that facilitate this storage.

By the end of this chapter, you will have a solid understanding of how a knowledge base, or the non-parametric memory of a RAG-based application is created. We will also embellish this chapter with snippets of python code so those of you who are so inclined, can try out a hands-on development of the indexing pipeline.

By the end of this chapter, you should –

- Know how to extract data from sources.
- Get a deeper understanding of text chunking strategies.
- Learn what embeddings are and how they are used.
- Gain the knowledge of vector storage and vector databases.
- Have an end-to-end knowledge of setting up the indexing pipeline.

3.1 Data Loading

In this section we will focus on the first stage of the indexing pipeline. We will read about data loaders, metadata information and data transformers.

The first step towards building a knowledge base (or non-parametric memory) of a RAG-enabled system is to source data from its original location. This data may be in the form of word documents, pdf files, csv, HTML etc. Further, the data may be stored in file, block or object stores, in data lakes, data warehouses or even in third party sources that can be accessed via the open internet. This process of sourcing data from its original location is called **Data Loading**. Loading documents from a list of sources may turn out to be a complicated process. It is advisable to document all the sources and the file formats in advance.

Before going too deep, let's begin with a simple example. If you recall, in Chapter 1, we used Wikipedia as a source of information about the 2023 Cricket World Cup. At that time, we copied the opening paragraph of the article and pasted it in the ChatGPT prompt window. Instead of doing it manually, we will now **connect** to Wikipedia and **extract** the data programmatically, using a very popular framework called LangChain. The code in this chapter and in the book can be run on python notebooks and is available in the Github repository of this book - https://github.com/abhinav-kimothi/A-Simple-Introduction-to-RAG

WHAT IS LANGCHAIN LangChain is an open-source framework, written in Python and JavaScript, developed by Harrison Chase to designed for building applications using Large Language Models. It was launched in October 2022. Apart from being suitable for Retrieval Augmented Generation, LangChain is suitable for building application use cases like Chatbots, Document Summarizers, Synthetic Data Generation and more. Over time, LangChain has built integrations with LLM providers like OpenAI, Anthropic, HuggingFace and more, a variety of vector store providers, cloud storage systems like AWS, Google and Azure, SQL and NoSQL databases, APIs for news, weather etc.

INSTALLING LANGCHAIN

To install LangChain (we'll use the version 0.1.20 in this chapter) using pip, run:

```
%pip install langchain==0.1.20
```

The langchain-community package contains third-party integrations. It is automatically installed by langchain but, in case it does not, you can also install it separately using pip:

```
%pip install langchain-community
```

Now that you have installed LangChain, we will now use it to connect to Wikipedia and extract data from the page about the 2023 Cricket World Cup. For this we will use the *AsyncHtmlLoader* function from the *document_loaders* library in the *langchain-community* package. To run *AsyncHtmlLoader* we will have to install another python package called bs4.

```
#Installing bs4 package
%pip install bs4==0.0.2 --quiet
```

#Importing the AsyncHtmlLoader

```
from langchain_community.document_loaders import AsyncHtmlLoader

#This is the url of the wikipedia page on the 2023 Cricket World Cup
url=”https://en.wikipedia.org/wiki/2023_Cricket_World_Cup”

#Invoking the AsyncHtmlLoader
loader = AsyncHtmlLoader (url)

#Loading the extracted information
data = loader.load()
```

The data variable in the code above now stores the information from the Wikipedia page.

```
print(data)
```

Below is the output (A large section of the text is replaced with … in order to save space)

```
>>[Document(page_content='<!DOCTYPE html>\n<html class="client-nojs vector-feature-
language-in-header-enabled……of In the knockout
stage, India and Australia beat New Zealand and South Africa respectively to advance to
the final, played on 19 November at <a href="/wiki/Narendra_Modi_Stadium"
title="Narendra Modi Stadium">Narendra Modi Stadium</a>. Australia won by 6 wickets,
winning their sixth Cricket World Cup title…… "datePublished":"2013-06-
29T19:20:08Z","dateModified":"2024-05-
01T05:16:34Z","image":"https:\\/\\/upload.wikimedia.org\\/wikipedia\\/en\\/e\\/eb\\/2023_
edition of the premier international cricket competition"}</script>\n</body>\n</html>',
metadata={'source': 'https://en.wikipedia.org/wiki/2023_Cricket_World_Cup', 'title':
'2023 Cricket World Cup - Wikipedia', 'language': 'en'})]
```

The variable data is a list of Documents which has two elements – page_content and metadata. page_content contains the text sourced from the URL. You will notice that the text along with the relevant information also has newline characters (\n) and other HTML tags. metadata on the other hand contains another important aspect of data.

Metadata is information about the data like its type, origin, purpose etc. This can include a summary of the data, how was the data created, who created it and why, when was it created, the size, quality, and condition of the data. Metadata information comes in extremely handy in the retrieval stage. Also, it can be used to resolve conflicting information that can arise due to chronology or origin. In our example above, while extracting the data from the URL, Wikipedia has already provided 'source', 'title' and 'language' in the metadata information. For many data sources, you will have to add metadata.

More often that naught, a **cleaning** of the source data is required. The data in our example above, has a lot of new line characters and HTML tags. This requires a certain level of cleanup.

We will attempt to clean up the webpage data that we extracted leveraging *Html2TextTransformer* function from the *document_transformers* library in the *langchain-community* package. For *Html2TextTransformer* we will also have to install another package called *html2text*.

```
#Install html2text
%pip install html2text==2024.2.26 –quiet

#Import Html2TextTransformer
from langchain_community.document_transformers import Html2TextTransformer

#Assign the Html2TextTransformer function
html2text = Html2TextTransformer()

#Call transform_documents
data_transformed = html2text.transform_documents(data)

print(data_transformed[0].page_content)
```

The output of the page_content is now free of any HTML tags and contains only the text from the webpage.

```
>>Jump to content  Main menu  Main menu  move to sidebar hide Navigation    * Main page
* Contents   * Current events   * Random article   * About Wikipedia   * Contact us   *
Donate  Contribute…….In the knockout stage, India and Australia beat New Zealand and
South Africa respectively to advance to the final, played on 19 November at Narendra
Modi Stadium. Australia won by 6 wickets, winning their sixth Cricket World Cup title……
* This page was last edited on 1 May 2024, at 05:16 (UTC).   * Text is available under
the Creative Commons Attribution-ShareAlike License 4.0; additional terms may apply. By
using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a
registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.    *
Privacy policy   * About Wikipedia   * Disclaimers   * Contact Wikipedia   * Code of
Conduct * Developers   * Statistics   * Cookie statement   * Mobile view    *
```

You can see above that now the text is more coherent since we have removed the HTML part of the data. There can be further cleanup like removing special characters and other unnecessary information. Data cleaning also becomes important to remove duplication. Yet another step to include in the data loading stage can be masking of sensitive information like PII or company secrets. In some cases, a fact checking may also be required.

The source for our data above was Wikipedia (more precisely, a web address pointing to a Wikipedia page) and the format was HTML. The source can also be other storage locations like AWS S3, SQL/NoSQL databases, Google Drive, GitHub, even WhatsApp, YouTube, and other social media sites. Likewise, the data formats can be .doc, .pdf, .csv, .ppt, .eml, etc. Most of the time, you will be able to leverage frameworks like LangChain that have integrations for the sources and the formats already built in. Sometimes, you may have to build custom connectors and loaders.

Though data loading may seem simplistic in the first glance, after all it's just connecting to a source and extract data, the nuances of adding metadata, document transformation, masking etc. add complexity to this step. Advanced planning of the sources, a review of the formats and curation of metadata information are advised for best results.

We have now taken the first step towards building our RAG-enabled system. Data Loading process can be further broken down into four sub steps as shown in figure 3.1 below.

1. Connect to the source of data.
2. Extract text from the file.
3. Review and update metadata information.
4. Clean or transform the data.

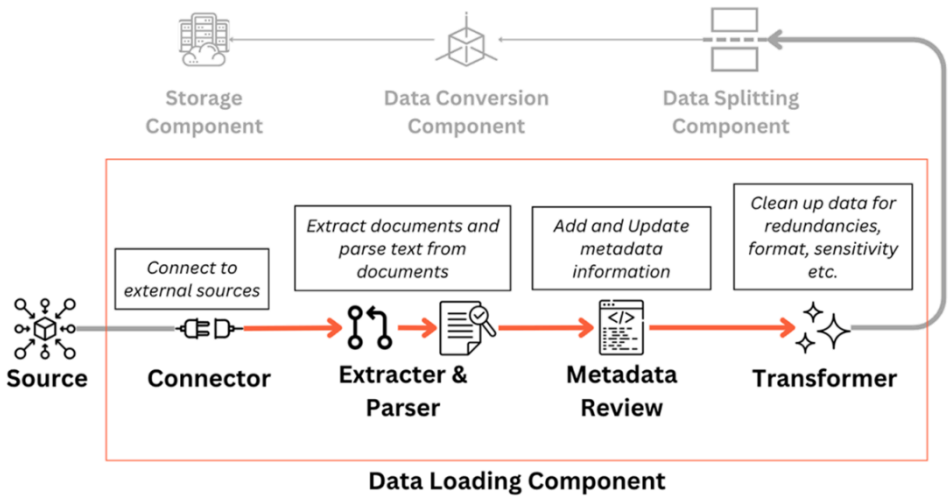

Figure 3.1 Four sub-steps of the Data Loading component of the Indexing Pipeline

We now have obtained data from the source and cleaned it to an extent. This Wikipedia page that we have loaded, alone, has more than 8,000 words. Imagine the number of words if we have multiple documents. For efficient management of information, we employ something called Data Splitting which we will discuss in the next section.

3.2 Data Splitting (Chunking)

Breaking down long pieces of text into manageable sizes is called **Data Splitting** or **Chunking**. In this section, we will discuss why chunking is necessary and the different strategies. We will also use functions from LangChain to illustrate a few examples.

3.2.1 Advantages of Chunking

In the previous section, we loaded the data from a URL (a Wikipedia page) and extracted the text out of it. It was a long piece of text of about 8,000 words. There are three advantages that chunking displays in overcoming the major limitations of using long pieces of text in LLM applications.

1. **Context Window of LLMs**: LLMs, due to the inherent nature of the technology, have a limit on the number of tokens (loosely, words) they can work with at a time. This includes both the number of tokens in the prompt (or the input) and the number of tokens in the completion (or the output). The limit on the total number of tokens that an LLM can process in one go is called the context window size. If we pass an input that is longer than the context window size, the LLM chooses to ignore all text beyond the size. It becomes very important to be careful with the amount to text that is being passed to the LLM. In our example, a text of 50,000 words will not work well with LLMs that have a smaller context window. The way to address this issue is to break the text down into smaller chunks.

2. **Lost in the middle problem**: Even in those LLMs which have a long context window (Claude 3 by Anthropic has a context window of up to 200,000 tokens), an issue with accurately reading the information has been observed. It has been noticed that accuracy declines dramatically if the relevant information is somewhere in the middle of the prompt. This problem can be addressed by passing only the relevant information to the LLM instead of the entire document.
3. **Ease of Search**: This is not a problem with the LLM per se, but it has been observed that large chunks of text are harder to search over. When we use a retriever (recall the generation pipeline introduced in Chapter 2) it is more efficient to search over smaller pieces of text.

WHAT ARE TOKENS? Tokens are the fundamental semantic units used in Natural Language Processing (NLP) tasks. Tokens can be assumed to be words but, sometimes, a single word can be made up of more than one token. OpenAI suggests one token to be made of four characters or 0.75 words. Tokens are important as most proprietary LLMs are priced based on token usage.

3.2.2 Chunking Process

The process of chunking can be understood in three steps as illustrated in figure 3.2.

1. Divide the longer text into compact, meaningful units like sentences or paragraphs.
2. Merge the smaller units into larger chunks until a specific size is achieved. Once the size is achieved, this chunk is treated as an independent segment of text.
3. When a new chunk is being created include a part of the previous chunk at the start of the new chunk. This overlap is necessary to maintain contextual continuity.

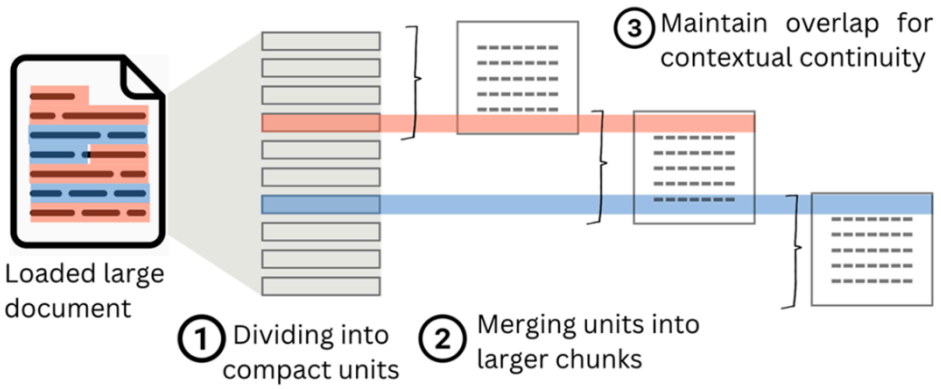

Figure 3.2 Data chunking process

3.2.3 Chunking Methods

While splitting documents into chunks might sound a simple concept, there multiple methods that can be employed to execute chunking. There are two aspects that vary across the chunking methodologies –

1. How is the text splitting done?
2. How is the size of the chunk measured?

FIXED SIZE CHUNKING

A very common approach is to pre-determine the size of the chunk and the amount of overlap between the chunks. Two of these methods that fall under the **Fixed Size Chunking** category are -

1. **Split by Character**: In this method, we specify a certain character like a newline character '\n' or a special character '*' to determine how the text should be split. Whenever this character is encounters, the text is split into a unit. The chunk size is measured in number of characters. We must choose the chunk size or the number of characters we need in each chunk. We can also choose the number of characters we need to overlap between two chunks. We will look at an example and demonstrate this method using ***CharacterTextSplitter*** from ***langchain_text_splitters***. For this we will take the same document that we loaded and transformed in the previous section from Wikipedia and stored in the variable data_transformed.

```
#Install langchain-text-splitters
pip install -qU langchain-text-splitters

#import libraries
from langchain_text_splitters import CharacterTextSplitter
#Set the CharacterTextSplitter parameters
text_splitter = CharacterTextSplitter(
    separator="\n",    #The character that should be used to split
    chunk_size=1000,   #Number of characters in each chunk
    chunk_overlap=200, #Number of overlapping characters between chunks
)

#Create Chunks
chunks=text_splitter.create_documents([data_transformed[0].page_content])

#Show the number of chunks created
print(f"The number of chunks created : {len(chunks)}")

>>The number of chunks created : 64
```

In all, this method created 64 chunks. But what about the overlap. Let us check two chunks at random, say, chunk 4 and chunk 5. We will compare the last 200 characters of chunk 4 with the first 200 characters of chunk 5.

```
chunks[4].page_content[-200:]

>> 'on was to be played from 9 February to 26 March\n2023.[3][4] In July 2020 it was
announced that due to the disruption of the\nqualification schedule by the COVID-19
pandemic, the start of the tournament'

chunks[5].page_content[:200]

>> '2023.[3][4] In July 2020 it was announced that due to the disruption of
the\nqualification schedule by the COVID-19 pandemic, the start of the tournament\nwould
be delayed to October.[5][6] The ICC rele'
```

Comparing the two outputs, we can observe that there is an overlap between the two consecutive chunks.
Splitting by Character is a simple and effective way to create chunks. It is the first chunking method that anyone should try. However, sometimes it may not be feasible to create chunks within the specified length. This is because the sequential occurrence of the character on which text needs to be split is far apart. You may see a message like Created a chunk of size 3799, which is longer than the specified 1000.
To address this issue, a recursive approach is employed.

2. **Recursively Split by Character:** This method is quite like the Split by Character but instead of specifying a single character for splitting, we specify a list of characters. The approach initially tries creating chunks based on the first character. In case it is not able to create a chunk of the specified size using the first character, it then uses the next character to further break down chunks to the required size. This method ensures, that chunks are largely created within the specified size. This method is recommended for generic texts. You can use *RecursiveCharacterTextSplitter* from LangChain to use this method. The only difference in RecursiveCharacterTextSplitter is that instead of passing a single character in the separator parameter separator="\n" we will need to pass a list separators= [“\n\n”,”\n”, ”.”, ” ”].

Another perspective to consider with fixed sized chunking is the use of tokens. Tokens, as we introduced in the beginning of this section, are the fundamental units of natural language processing. They can loosely be understood as a proxy for words. All LLMs process text in form of tokens. So, it would also make sense to use tokens to determine the size of the chunks. This method is called the **Split by Token** method. In Split by Token method, the splitting still happens based on a character, but the size of the chunk and the overlap is determined by the number of tokens instead of number of characters.

> **COMMON TOKENIZERS** Tokenizers are used to create tokens from a piece of text. Tokens are slightly different from words. A phrase like "I'd like that!" has three words, however, in NLP this text may be parsed as five tokens i.e. "I", "'d", "like", "that", "!". Different LLMs use different methods for creating these tokens. OpenAI uses a tokenizer called **tiktoken** for GPT3.5, GPT4 models, Llama2 by Meta uses **LLamaTokenizer** that is available in the transformers library by HuggingFace. You can also explore other tokenizers on HuggingFace. NLTK, spaCy are some other popular libraries that can be used as tokenizers.

To use the split by token method, you can use specific methods within the *RecursiveCharacterTextSplitter* and *CharacterTextSplitter* classes, like RecursiveCharacterTextSplitter.from_tiktoken_encoder (encoding="cl100k_base", chunk_size=100, chunk_overlap=10) for creating chunks of 100 tokens with an overlap of 10 tokens using OpenAI's tiktoken tokenizer or CharacterTextSplitter.from_huggingface_tokenizer(tokenizer, chunk_size=100, chunk_overlap=10)for creating the same sized chunk using another tokenizer from HuggingFace.

The limitation of Fixed Size Chunking is that it doesn't consider the semantic integrity of the text. In other words, the meaning of the text is ignored. It works best in scenarios where data is inherently uniform like genetic sequences, service manuals or uniformly structured reports like survey responses.

SPECIALIZED CHUNKING

The aim of chunking is to keep meaningful data together. If we are dealing with data in form of HTML, Markdown, JSON or even computer code, it makes more sense to split the data based on the structure rather than a fixed size. Another approach for chunking is to take into consideration the format of the extracted and loaded data. A markdown file, for example is organized by headers, a code written in a programming language like python or java is organized by classes and functions and HTML, likewise, is organized in headers and sections. For such formats a specialized chunking approach can be employed. LangChain offers classes like ***MarkdownHeaderTextSplitter***, ***HTMLHeaderTextSplitter***, ***RecursiveJsonSplitter*** amongst others for these formats.

Below is a simple example of a code that splits an HTML document using *HTMLHeaderTextSplitter*. We are using the same Wikipedia article to source the HTML page. We first split the input data basis on the headers. Headers in HTML are tagged as <h1>, <h2>, <h3> and so on. It can be assumed that a well-structured HTML document will have similar information in a particular header. This helps us in creating chunks that have similar information. To use the *HTMLHeaderTextSplitter* library we must install another python package called lxml.

```
#Installing lxml
%pip install lxml==5.2.2 --quiet

# Import the HTMLHeaderTextSplitter library
from langchain_text_splitters import HTMLHeaderTextSplitter

# Set url as the Wikipedia page link
url="https://en.wikipedia.org/wiki/2023_Cricket_World_Cup"

# Specify the header tags on which splits should be made
headers_to_split_on=[
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("h4", "Header 4")
]

# Create the HTMLHeaderTextSplitter function
html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)

# Create splits in text obtained from the url
html_header_splits = html_splitter.split_text_from_url(url)
```

The advantage of specialized chunking is that chunk sizes are no longer limited by a fixed width. It helps in preserving the inherent structure of the data. Because the size of the chunks changes depending on the structure, this method is also sometimes called **Adaptive Chunking**. Specialized Chunking works well in structured scenarios like customer reviews or patient records where data can be of different lengths but should ideally be in the same chunk.

In the previous example, let us see how many chunks have been created?

```
len(html_header_splits)

>> 26
```

This method has given us 26 chunks from the URL. Chunking methods do not have to be exclusive. We can further chunk these 26 chunks using a fixed size chunking method like *RecursiveCharacterTextSplitter.*

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200
)

chunks = text_splitter.split_documents(html_header_splits)
```

Let us look at how many chunks were created by this combination of techniques.

```
len(chunks)

>> 67
```

67 chunks were created by first splitting the HTML data from the URL using a specialized chunking method followed by a fixed size method. This gave us more chunks than using the fixed size method alone, that we saw in the previous section (Split by Character gave us 64 chunks).

You may be wondering about the advantages of having more chunks and the optimal number of chunks. Unfortunately, there's no straightforward answer to that. Having many chunks (consequently smaller sized chunks) means that the information in the chunks is precise. This is advantageous in providing the LLM with accurate information. On the other hand, by chunking into small sizes you may lose the overall themes, ideas, and coherence of the larger document. The task here is to strike a balance. We will discuss more on chunking strategies after we take a cursory look at a novel method that looks at the meaning of the text to perform chunking and aims to create chunks that are super-contextual.

SEMANTIC CHUNKING

This idea proposed by Greg Kamradt questions two aspects of the previous chunking methods.

1. Why should we have a pre-defined fixed size of chunks?
2. Why don't chunking methods take into consideration the actual meaning of content?

To address these issues, a method that looks at semantic similarity (or similarity in the meaning) between sentences is called semantic chunking. It first creates groups of three sentences and then merges groups that are similar in meaning. To find out the similarity in meaning, this method uses Embeddings (We will discuss Embeddings in the next section 3.3). This is still an experimental chunking technique. In LangChain, you can use the class *SemanticChunker* from the *langchain_experimental.text_splitter library*.

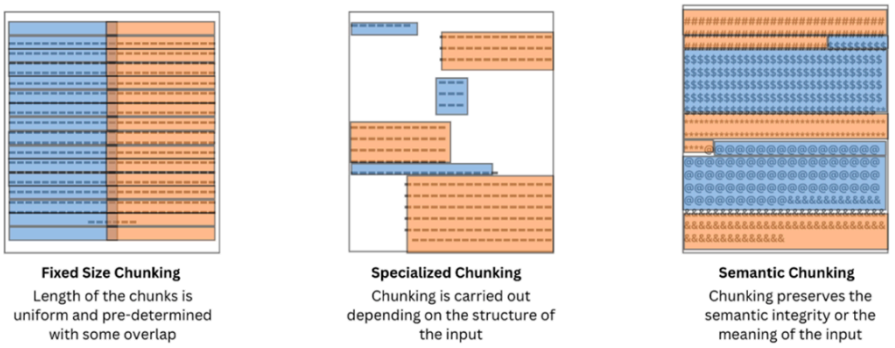

Figure 3.3 Chunking Methods.

As the Large Language Model and the Generative AI space is a fast evolving one, chunking methods are also becoming more sophisticated. Simple chunking methods predetermine the size of the chunks and a split by characters. A slightly more nuanced technique is to split the data by tokens. Specialized methods are more suitable for different formats of data. Experimental techniques like semantic chunking and agentic chunking are spearheading the advancements in the chunking space. Now, let us come to the important question of how to choose a chunking method.

3.2.4 Choice of Chunking Strategy

We have seen that there are many chunking methods available to us. Which chunking method to use, whether to use a single method or multiple methods are questions that come up during the creation of the indexing pipeline. There are no guidelines or rules to answer these questions. However, certain features of the application that you're developing can guide you towards an effective strategy.

NATURE OF THE CONTENT

The type of data that you're dealing with can be a guide for the chunking strategy. If your application uses a data in a specific format like code or HTML, a specialized chunking method is recommended. Not only that, whether you're working with long documents like whitepapers and reports or short form content like social media posts, tweets, etc. can also guide the chunk size and overlap limits. If you're using a diverse set of information sources, then you might have to use different methods for different sources.

EXPECTED LENGTH AND COMPLEXITY OF USER QUERY

The nature of the query that your RAG enabled system is likely to receive is also a determinant of the chunking strategy. If your system expects a short and straightforward query, then the size of your chunks should be different when compared to a long and complex query. Matching long queries to short chunks may prove inefficient in certain cases. Similarly, short queries matching with large chunks may yield partially irrelevant results.

APPLICATION & USE CASE REQUIREMENTS

The nature of the use case you're addressing may also determine the optimal chunking strategy. For a direct question answering system, it is likely that shorter chunks are used for precise results, while for summarization tasks, longer chunks may make more sense. If the results of your system need to serve as an input to another downstream application, that may also influence the choice of the chunking strategy.

EMBEDDINGS MODEL BEING USED

We are going to discuss embeddings in the next section. For now, you can make a note that certain embeddings models perform better with chunks of specific sizes.

We have discussed chunking at length in this section. From understanding the need and advantages of chunking to different chunking methods and the choice of chunking strategies, you are now equipped to load data from different sources and split them into optimal sizes. Remember, chunking is not an overcomplicated task and most chunking methods will work. You will have to evaluate and improve your chunking strategy depending on the results you observe.

Now that data has been split into manageable sizes, we need to store it so that it can be fetched later to be used in the generation pipeline. We need to ensure that these chunks can be effectively searched over to match the user query. Turns out that one data pattern is the most efficient for such tasks. This pattern is called embeddings. Let us explore embeddings and their use in RAG enabled systems in the next section.

3.3 Data Conversion (Embeddings)

Computers, at the very core, do mathematical calculations. Mathematical calculations are done on numbers. Therefore, for a computer to process any kind of non-numeric data like text or image, it must be first converted into a numerical form.

3.3.1 What are Embeddings?

Embeddings is a design pattern that is extremely helpful in the fields of data science, machine learning and artificial intelligence. Embeddings are vector representations of data. As a general definition, embeddings are data that has been transformed into n-dimensional matrices. A word embedding is a vector representation of words. We will understand embeddings by taking an example of three words – Dog, Bark and Fly.

WHAT ARE VECTORS? In physics and mathematics, vector is an object that has a magnitude and a direction – like an arrow in space. The length of the arrow is the magnitude of the quantity and the direction that the arrow points to is the direction of the quantity. Examples of such quantities in physics are velocity, force, acceleration etc. In computer science and machine learning, the idea of a vector is an abstract representation of data, and the representation is an array or list of numbers. These numbers represent the features or attribute of the data. In NLP, a vector can represent a document, a sentence or even a word. The length of the array or list is the number of dimensions in the vector. A two-dimensional vector will have two numbers, a three-dimensional vector has three numbers, and an n-dimensional vector will have 'n' numbers.

Let us understand embeddings by assigning a number to the three words – Dog = 1, Bark = 2 and Fly = 6, as shown in figure 3.4. We chose these numbers because the word 'Dog' is closer to the word 'Bark' and farther from the word 'Fly'.

Figure 3.4 Words in a unidimensional vector

Unidimensional vectors are not great representations because we can't accurately plot unrelated words accurately. In our example, we can plot that words Fly and Bark which are verbs are far from each other and Bark is closer to a Dog because, dogs can bark. But how do we plot a word like Love or Red. To accurately represent all the words, we need to increase the number of dimensions.

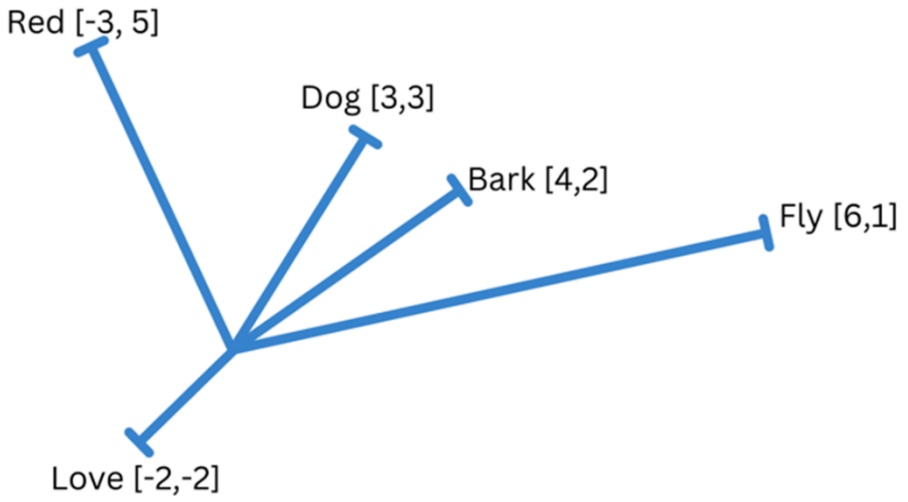

Figure 3.5 Words in a two-dimensional vector space

The goal of an embedding model is to convert words (or sentences/paragraphs) into n-dimensional vectors, such that the words (or sentences/paragraphs) that are like each other in meaning, lie close to each other in the vector space.

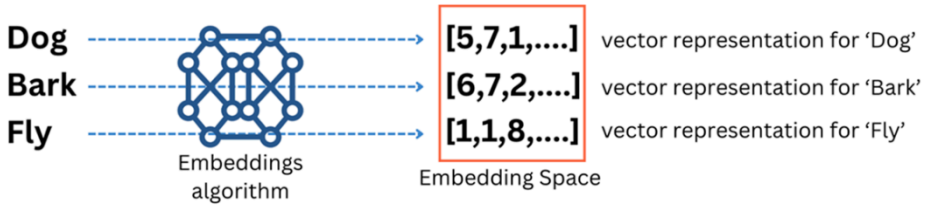

Figure 3.6 The process of embedding transforms data (like text) into vectors, compresses the input information resulting in an embedding space specific to the training data.

An embeddings model can be trained on a corpus of preprocessed text data using an embedding algorithm like Word2Vec, GloVe, FastText or BERT.

Popular Embeddings Algorithms

1. **Word2Vec**: Word2Vec is a shallow neural network-based model for learning word embeddings developed by researchers at Google. It is one of the earliest embedding techniques.
2. **GloVe**: Global Vectors for Word Representations is an unsupervised learning technique developed by researchers at Stanford University.
3. **FastText**: FastText is an extension of Word2Vec developed by Facebook AI Research. It is particularly useful for handling misspellings and rare words.
4. **ELMo**: Embeddings from Language Models was developed by researchers at Allen Institute for AI. ELMo embeddings have been shown to improve performance on question answering and sentiment analysis tasks.
5. **BERT**: Bidirectional Encoder Representations from Transformers, developed by researchers at Google, is a Transformers architecture-based model. It provides contextualized word embeddings by considering bidirectional context, achieving state-of-the-art performance on various natural language processing tasks.

Training a custom embeddings model can prove to be beneficial in some use cases where the scope is limited. Training an embeddings model that generalizes well can be a laborious exercise. Collection and pre-processing text data can be cumbersome. The training process can turn out to be computationally expensive too.

3.3.2 Common Pre-trained Embeddings Models

The good news for anyone building RAG enabled systems is that embeddings once created can also generalize across tasks and domains. There are a variety of proprietary and open-source pre-trained embeddings models that are available to use. This is also one of the reasons why the usage of embeddings has exploded in popularity across machine learning applications.

1. **Embeddings Models by OpenAI**
   OpenAI, the company behind ChatGPT and GPT series of Large Language Models also provide three Embeddings Models.
   a. **text-embedding-ada-002** was released in December 2022. It has a dimension of 1536 meaning that it converts text into a vector of 1536 dimensions.
   b. **text-embedding-3-small** is the latest small embedding model of 1536 dimensions released in January 2024. The flexibility it provides over ada-002 model is that users can adjust the size of the dimensions according to their needs.
   c. **text-embedding-3-large** is a large embedding model of 3072 dimensions released together with the text-embedding-3-small model. It is the best performing model released by OpenAI yet.
   OpenAI models are closed source and can be accessed using the OpenAI API and are priced based on the number of input tokens for which embeddings are desired.

2. **Gemini Embeddings Model by Google**
   text-embedding-004 (last updated in April 2024) is the model offered by Google Gemini. It offers elastic embeddings size up to 768 dimensions and can be accessed via the Gemini API
3. **Voyage AI**
   Voyage AI embeddings models are recommended by Anthropic, the providers of Claude series of Large Language Models. Voyage offers several embeddings models like –
   a. **voyage-large-2-instruct** is a 1024-dimension embeddings model that has become a leader in embeddings models.
   b. **voyage-law-2** is a 1024-dimension model that has been optimized for legal documents.
   c. **voyage-code-2** is a 1536-dimension model that has been optimized for code retrieval.
   d. **voyage-large-2** is a 1536-dimension general purpose model optimized for retrieval.
   Voyage AI offers several free tokens before charging for using the embeddings models.

4. **Mistral AI Embeddings**

Mistral is the company behind LLMs like Mistral and Mixtral. They offer a 1024-dimension embeddings model by the name of **mistral-embed**. This is an open-source embeddings model.

5. **Cohere Embeddings**
   Cohere, the developers of Command, Command R and Command R+ LLMs also offer a variety of embeddings models. Some of these are-
   a. **embed-english-v3.0** is a 1024-dimension model that works on embeddings for English only.
   b. **embed-english-light-v3.0** is a lighter version of embed-english model that has 384 dimensions.
   c. **embed-multilingual-v3.0** offers multilingual support for over 100 languages.
   Cohere embeddings can be accessed via cohere API.

These five models are in no way recommendations but just a list of the popular embeddings models. Apart from these providers, almost all LLM developers like Meta, TII, LMSYS also offer pre-trained embeddings models. One place to check out all the popular embeddings models is the MTEB (Massive Text Embedding Benchmark) Leaderboard on HuggingFace (https://huggingface.co/spaces/mteb/leaderboard). The MTEB benchmark compares the embeddings models on tasks like classification, retrieval, clustering and more. We now know what embeddings are, but why are they useful? Let us discuss that now with some examples of use cases.

3.3.3 Embeddings Use Cases

The reason why embeddings are popular is because they help in establishing semantic relationship between words, phrases, and documents. In the simplest methods of searching or text matching, we use keywords and if the keywords match, we can show the matching documents as results of the search. However, this approach fails to consider the semantic relationships or the meanings of the words while searching. This challenge is overcome by using embeddings.

HOW IS SIMILARITY CALCULATED

We discussed that embeddings are vector representations of words or sentences. Similar pieces of text lie close to each other. Closeness to each other is calculated by the distance between the points in the vector space. One of the most common measures of similarity is **Cosine Similarity**. Cosine similarity is calculated as the cosine value of the angle between the two vectors. Recall from trigonometry that cosine of parallel lines i.e. angle=0^o is 1 and cosine of a right angle i.e. 90^o is 0. On the other end, the cosine of opposite lines i.e. angle =180^o is -1. Therefore, the cosine similarity lies between -1 and 1 where unrelated terms have a value close to 0, and related terms have a value close to 1. Terms that are opposite in meaning have a value of -1.

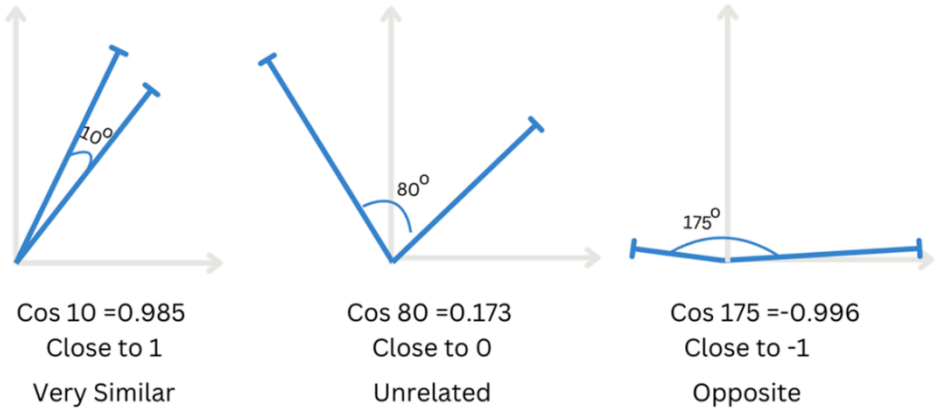

Figure 3.7 Cosine similarity of vectors in two-dimensional vector space.

Yet another measure of similarity is the **Euclidian distance** between two vectors. Vectors that are close have a small Euclidian distance. It can be calculated using the formula -

Distance (A, B) = $\text{sqrt}((A_i - B_i)^2)$

where i is the i-th dimension of the n-dimensional vectors

DIFFERENT USE CASES OF EMBEDDINGS

Text Search: Searching through the knowledge base for the right document chunk is a key component of RAG enabled systems. Embeddings are used to calculate similarity between the user query and the stored documents.

Clustering: Categorizing similar data together to find themes and groups in the data can result in valuable insights. Embeddings are used to group similar pieces of text together to find out, for example, the common themes in customer reviews.

Machine Learning: Advanced machine learning techniques can be used for different problems like classification and regression. To convert text data into numerical features, embeddings prove to be a valuable technique.

Recommendation Engines: Shorter distances between product features means greater similarity. Using embeddings for product and user features can be used to recommend similar products.

Since we are focusing on RAG enabled systems, we will focus on using embeddings for text search – to find the document chunks that are closest to the user's query. Let us continue with our example of the Wikipedia page on the 2023 Cricket World Cup. In the last section, we created 67 chunks using a combination of specialized and fixed width chunking. Now we will see how to create embeddings for each of the chunks. We will see how to use an open source as well as a proprietary embeddings model.

Below is the code example for creating embeddings using an open-source embeddings model all-MiniLM-l6-v2 via HuggingFace. We will have to install the sentence_transformers library.

```
# Install the Sentence Transformers library
%pip install sentence_transformers ==2.7.0 --quiet

# Import HuggingFaceEmbeddings from embeddings library
from langchain_community.embeddings import HuggingFaceEmbeddings

# Instantiate the embeddings model. The embeddings model_name can be changed as desired
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-l6-v2")

# Create embeddings for all chunk
chunk_embedding = embeddings.embed_documents([chunk.page_content for chunk in chunks])

#Check the length(dimension) of the embedding
len(chunk_embedding[0])
>> 384
```

This model creates embeddings of dimension 384. The list chunk_embedding is made up of 67 lists, one list each of 384 numbers for each of the chunk. Figure 3.8 shows the embeddings space of all the chunks.

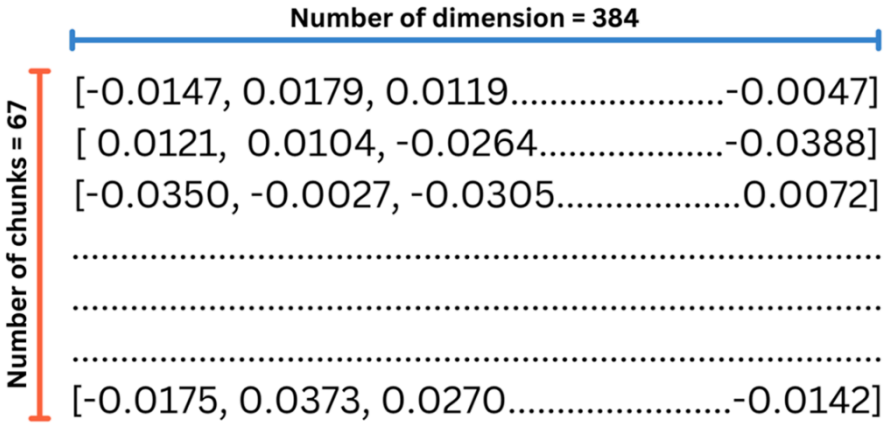

Figure 3.8 Embeddings created for chunks of Wikipedia page using all-MiniLM-l6-v2 model.

Similarly, we can use a proprietary model like the text-embedding-3-large model hosted by OpenAI. The only pre-requisite is obtaining an API key and setting up a billing account with OpenAI.

```
# Install the langchain openai library
%pip install langchain-openai==0.1.6 --quiet

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

# Set the OPENAI_API_KEY as the environment variable
import os
os.environ["OPENAI_API_KEY"] = <YOUR_API_KEY>

# Instantiate the embeddings object
embeddings = OpenAIEmbeddings(model="text-embedding-3-large")

# Create embeddings for all chunks
chunk_embedding = embeddings.embed_documents([chunk.page_content for chunk in chunks])

#Check the length(dimension) of the embedding
len(chunk_embedding[0])

>> 3072
```

This text-embedding-3-large model creates embeddings for the same chunks of dimension 3072.

There are several embeddings models available for use and new ones get added every day. The choice of embeddings can be dictated by certain factors. Let us look at a few factors.

3.3.4 How to choose embeddings?

There are a few major factors that will impact your choice of embeddings.

USE CASE

Your application use case may determine your choice of embeddings. The MTEB leaderboard scores each of the embeddings models across seven use cases – Classification, Clustering, Pair Classification, Reranking, Retrieval, Semantic Text Similarity and Summarization. At the time of writing this book, SFR-Embedding-Mistral model developed by Salesforce performs the best for retrieval tasks.

COST

Cost is another important factor to consider. To create the knowledge base, you may have to create embeddings for thousands of documents running into millions of tokens.

Embeddings are powerful data patterns that are most effective in finding similarity between text. In RAG enabled systems, embeddings play a critical role in search and retrieval of data relevant to the user query. Once the embeddings have been created, they need to be stored in persistent memory for real time access. To store embeddings a new kind of database called **Vector Database** has gained popularity.

3.4 Storage (Vector Databases)

Now we are at the last step of the indexing pipeline. The data has been loaded, split, and converted into embeddings. For us to use this information repeatedly, we need to store it in memory so that it can be accessed on demand.

3.4.1 What are vector databases?

The evolution of databases can be traced back to early days of computing. Databases are organized collection of data, designed to be easily accessed, managed, and updated. Relational databases like MySQL organize structured data into rows and columns. NoSQL Databases, like MongoDB, specialize in handling unstructured and semi-structured data. Graph databases, like Neo4j, are optimized for querying graph data. In the same manner, Vector Databases are built to handle high dimensional vectors. These databases specialize in indexing and storing vector embeddings for fast semantic search and retrieval.

Apart from efficiently storing high dimensional vector data, modern vector databases offer traditional features like scalability, security, multi-tenancy, versioning & management, etc. However, vector databases are unique in offering similarity search based on Euclidian Distance or Cosine Similarity. They also employ specialized indexing techniques.

3.4.2 Types of vector databases

Vector databases started as specialized database offering but propelled by the growth in demand for storing vector data, all major database providers have added vector indexing capability. We can categorize the popular vector databases available today into six broad categories.

Vector Indices: These are libraries that focus on the core features of indexing and search. They do not support data management, query processing, interfaces etc. They can be considered a bare bones vector database. Examples of vector indices are Facebook AI Similarity Search (FAISS), Non-Metric Space Library (NMSLIB), Approximate Nearest Neighbors Oh Yeah (ANNOY), Scalable Nearest Neighbors (ScaNN), etc.

Specialized Vector Databases: These databases are focused on the core feature of high-dimensional vector support, indexing, search, and retrieval, like vector indices, but also offer database features like data management, extensibility, security, scalability, non-vector data support etc. Examples of specialized vector DBs are Pinecone, ChromaDB, Milvus, Qdrant, Weaviate, Vald, LanceDB, Vespa, Marqo, etc.

Search Platforms: Solr, Elastic Search, Open Search, Apache Lucene etc. are traditional text search platforms and engines built for full text search. They have now added vector similarity search capabilities to their existing search capabilities.

Vector Capabilities for SQL Databases with: Azure SQL, Postgres SQL(pgvector), SingleStore, CloudSQL, etc. are traditional SQL databases that have now added vector data handling capabilities

Vector Capabilities for NoSQL Databases: Like SQL DBs, NoSQL DBs like MongoDB have also added vector search capabilities.

Graph Databases with Vector Capabilities: Graph DBs like Neo4j adding vector capabilities has also opened new possibilities.

Using a vector Index like FAISS is supported by LangChain. To use FAISS we will first have to install the faiss-cpu library. We will use the already created chunks in section 3.2 and the OpenAI embeddings that we used in section 3.3.

```
# Install FAISS-CPU
%pip install faiss-cpu==1.8.0 --quiet

# Import FAISS class from vectorstore library
from langchain_community.vectorstores import FAISS

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

# Set the OPENAI_API_KEY as the environment variable
import os
os.environ["OPENAI_API_KEY"] = <YOUR_API_KEY>

# Chunks from Section 3.3
chunks=chunks

# Instantiate the embeddings object
embeddings=OpenAIEmbeddings(model="text-embedding-3-large")

# Create the database
db=FAISS.from_documents(chunks,embeddings)

# Check the number of chunks that have been indexed
db.index.ntotal

>> 67
```

With the above code, the 67 chunks of data have been converted to vector embeddings and these embeddings are stored in a FAISS vector index. The FAISS vector index can also be saved to memory using the db.save_local(folder_path) and FAISS.load_local(folder_path) functions. Let us now take a cursory look at how a vector store can be used. We will take our original question that we have been asking since the beginning of this book – "Who won the 2023 Cricket World Cup?"

```
# Original Question
query = "Who won the 2023 Cricket World Cup?"

# Ranking the chunks in descending order of similarity
docs = db.similarity_search(query)

# Printing one of the top ranked chunk
print(docs[1].page_content)

>> 13th edition of ICC Cricket World Cup
It takes One Day
India
 2019
2027 
```

The 2023 ICC Men's Cricket World Cup (also referred to as simply the 2023 Cricket World Cup) was the 13th edition of the Cricket World Cup, a quadrennial One Day International (ODI) cricket tournament organized by the International Cricket Council (ICC). It was hosted from 5 October to 19 November 2023 across ten venues in India.

The tournament was contested by ten national teams, maintaining the same format used in 2019. In the knockout stage, India and Australia beat New Zealand and South Africa respectively to advance to the final, played on 19 November at Narendra Modi Stadium. Australia won by 6 wickets, winning their sixth Cricket World Cup title.

Similarity Search orders the chunks in descending order of similarity, meaning that the most similar chunks to the query, are ranked on top. In the example above, we can observe that the chunk that speaks about the world cup final has been ranked on top.

FAISS is a strip down high-performance vector index that can work for many applications. ChromaDB is another user-friendly vector DB that has gained popularity. Pinecone offers managed services and customization. Milvus claims higher performance on similarity search while Qdrant boasts of an advanced filtering system. We will now discuss some points on how to choose a vector database that works for your requirements.

3.4.3 Choosing a Vector Database

All vector databases offer the same basic capabilities, but each one of them also claims a differentiated value. Your choice should be influenced by the nuance of your use case matching with the value proposition of the database. A few things to consider while evaluating and implementing a vector database –

1. **Accuracy vs Speed**: Certain algorithms are more accurate but slower. A balance between search accuracy and query speed must be achieved based on application needs. It will become important to evaluate vector DBs on these parameters.

2. **Flexibility vs Performance**: Vector DBs provide customizations to the user. While it may help you in tailoring the DB to your specific requirements, more customizations can add overhead and slow systems down.
3. **Local vs Cloud Storage**: Assess tradeoffs between local storage speed and access vs cloud storage benefits like security, redundancy, and scalability.
4. **Direct Access vs API**: Determine if tight integration control via direct libraries is required or if ease- of-use abstractions like APIs better suit your use case.
5. **Simplicity vs Advanced Features**: Compare advanced algorithm optimizations, query features, and indexing vs how much complexity your use case necessitates vs needs for simplicity.
6. **Cost**: While you many incur regular cost in a fully managed solution, a self-hosted one might prove costlier if not managed well.

We have now completed an end-to-end indexing of a document. We continued with the same question – "Who won the 2023 Cricket World Cup?" and the same external source – Wikipedia page of the 2023 Cricket World Cup (https://en.wikipedia.org/wiki/2023_Cricket_World_Cup). In this chapter, we started with programmatic loading of this Wikipedia page and extracting the HTML document and then parsing the HTML document to extract text. Thereafter, we divided the text into small sized chunks using a specialized and a fixed width chunking method. We converted these chunks into embeddings using OpenAI's text-embedding-003-large model. Finally, we stored the embeddings into a FAISS vector index. We also saw how using similarity search on this vector index, we were able to retrieve relevant chunks.

When several such documents, in different formats from different sources, are indexed using a combination of methods and strategies, we can store all the information in form of vector embeddings creating a non-parametric knowledge base for our RAG enabled system (shown in figure 3.9).

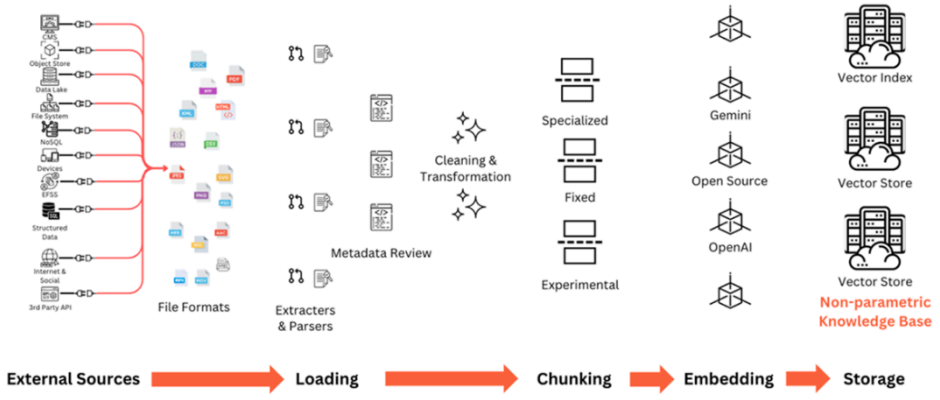

Figure 3.9 Creating the knowledge base for RAG enabled system using a combination of strategies.

This concludes our discussion on the indexing pipeline. By now, you must have built a solid understanding of the four components of the indexing pipeline and should be ready to build a knowledge base for a RAG enabled system.

In the next chapter we will use this knowledge base to generate real time responses to user queries through the generation pipeline.

3.5 Summary

DATA LOADING

- The process of sourcing data from its original location is called **Data Loading**.
- Data Loading is comprised of four steps-
    - Connecting to the source.
    - Extracting and parsing text.
    - Reviewing and updating metadata.
    - Cleaning and transforming data.

- Loading documents from a list of sources may turn out to be a complicated process. Make sure to plan for all the sources and loaders in advance.
- A variety of data loaders from LangChain can be leveraged.

DATA SPLITTING

- Breaking down long pieces of text into manageable sizes is called **Data Splitting** or **Chunking.**
- Chunking addresses context window limits of LLMs, mitigates the "lost in the middle" problem for long prompts, and enables easier search and retrieval.
- The chunking process involves dividing longer texts into small units, merging small units into chunks, and including an overlap between chunks to preserve contextual continuity.
- Chunking can be fixed size, specialized (or adaptive) and semantic. Newer chunking methods are constantly getting introduced.
- Your choice of the chunking strategy should be based on the nature of the content, expected length and complexity of user query, application use case and the embeddings model being used.
- Chunking strategy can include multiple methods working together.

DATA CONVERSION

- For processing, text needs to be converted into a numerical format.
- Embeddings are vector representations of data (words, sentences, documents etc.).
- The goal of an embedding algorithm is to position similar datapoints close to each other in a vector space.

- Several pre-trained, open source and proprietary, embedding models are available for use.
- Embeddings models enable similarity search.
- Embeddings can be used for text search, clustering, machine learning models and recommendation engines.
- The choice of embeddings is largely based on the use case and the cost implications.

DATA STORAGE

- Vector databases designed to efficiently store and retrieve high-dimensional vector data like embeddings.
- Vector databases provide similarity search based on distance metrics like cosine similarity.
- Apart from the similarity search, vector databases offer traditional services like scalability, security, versioning etc.
- Vector capabilities can be offered by standalone vector indices, specialized vector databases or, legacy offerings like search platforms, SQL, and NoSQL databases with added vector capabilities.
- Accuracy, speed, flexibility, storage, performance, simplicity, access, and cost are some of the factors that can influence the choice of a vector database.

4 Generation Pipeline: Generating Contextual LLM Responses

This chapter covers

- Retrievers and Retrieval Methodologies
- Augmentation with Prompt Engineering Techniques
- Generation using Large Language Models
- Basic implementation of the RAG pipeline in python

In Chapter 3, we discussed the creation of the knowledge base or the non-parametric memory of RAG-based applications via the Indexing Pipeline. To leverage this knowledge base for accurate and contextual responses a Generation Pipeline including the steps of retrieval, augmentation and generation is created.

In this chapter, we will elaborate the three components of the Generation Pipeline. We will begin by discussing the retrieval process which primarily involves searching through the embeddings stored in vector databases of the knowledge base and returning a list of documents that closely match the input query of the user. We will understand the concept of retrievers and a few retrieval algorithms. We will then move to the augmentation step. At this point, it will also be worthwhile understanding the different prompt engineering frameworks that are used with RAG. Finally, as part of the generation step, we will discuss a few stages of the LLM lifecycle like using foundation models vs supervised fine-tuning, models of different sizes and open-source vs proprietary models in the context of RAG. In each of these steps, we will also highlight the benefits and drawbacks of different methods.

By the end of this chapter, you will be equipped with the understanding of the two foundational pipelines of a RAG-enabled system. In a way, you will be ready to build a basic RAG enabled system.

By the end of this chapter, you should –

- Know several retrievers used in RAG.
- Get an understanding augmentation using prompt engineering.
- Learn some details about how Large Language Models are used in the context of RAG.
- Have an end-to-end knowledge of setting up a basic RAG-enabled system.

Let's get started with an overview of the generation pipeline before diving into each of the components.

4.1 Generation Pipeline Overview

Recall the generation pipeline that was introduced in Chapter 2. When a user provides an input, the generation pipeline is responsible for providing the contextual response. The retriever searches for the most appropriate information from the knowledge base. The user question is augmented with this information and passed as input to the LLM for generating the final response. This is illustrated again in figure 4.1 below.

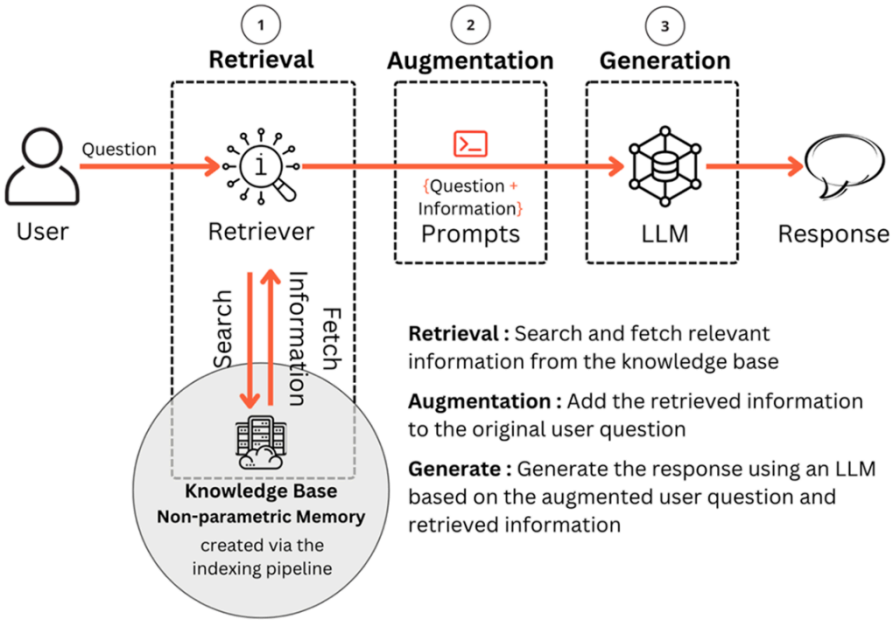

Figure 4.1 Generation Pipeline Overview with the three components i.e. retrieval, augmentation and generation

The generation pipeline can be thought of as three processes. Retrieval, Augmentation and Generation. The retrieval process is responsible for fetching the information relevant to the user query from the knowledge base. Augmentation is the process of combining the fetched information to the user query. Generation is the last step where the LLM generates a response based on the augmented prompt. This chapter will discuss these three processes in detail.

4.2 Retrieval

Retrieval refers to the process of finding and extracting relevant pieces of information from a large corpus or knowledge base. We saw, in chapter 3, that information from various sources is parsed, chunked and stored as embeddings in vector databases. These stored embeddings are also sometimes referred as documents and the knowledge base consists of several volumes of documents.

Retrieval, essentially, is a search problem with the objective of finding the documents that best match the input query.

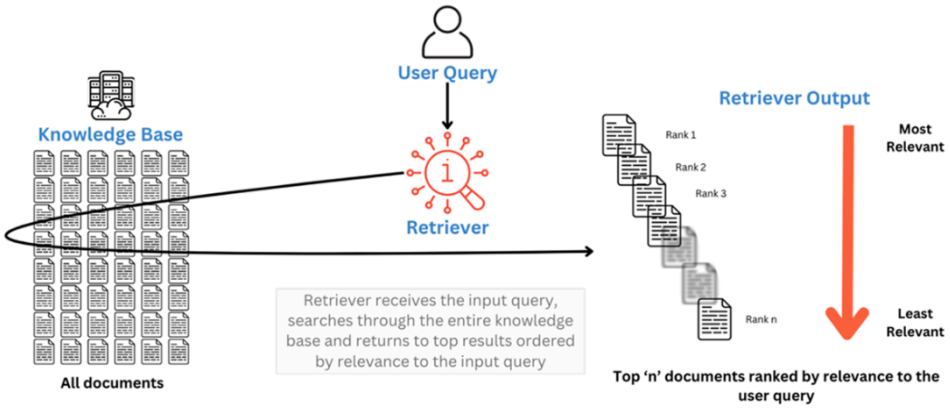

Figure 4.2 A Retriever searches through the knowledge base and returns the most relevant documents

Searching through the knowledge base and retrieving the right documents is done by a component called the *retriever*. In simple terms, retrievers accept a query as input and return a list of matching documents as output. This is illustrated in figure 4.2. You can imagine that retrieval is an extremely crucial step since the quality of the retrieved information directly impacts the quality of the output that will be generated.

We have already discussed embeddings in Chapter 3 while building the indexing pipeline. Using embeddings, we can find documents that match the user query. Embeddings is one method in which retrieval can happen. There are other methods too and it is worth spending some time understanding different types of retrieval methods and the way they calculate the results.

In this section on retrievers, we will first discuss different retrieval algorithms and their significance in the context of RAG. In RAG based systems, one or more retrieval methods can be used to build the retriever component. Then we will look at a few examples of pre-built retrievers that can directly be used using a framework like LangChain. These are integrated with services like databases, cloud providers or third-party information sources. Finally, we will close this section by building a very simple retriever using in LangChain using python. We will continue with this example to demonstrate the augmentation and generation steps too, so that by the end of this chapter, we have a full implementation of the generation pipeline.

INDEXING AND RETRIEVAL ARE TIGHTLY COUPLED WITH EACH OTHER In Chapter 3, we discussed indexing and how to convert and store data in a numerical form that can be used to retrieve information later. You may recall we discussed embeddings at length in section 3.3. It should be intuitive that since we stored the data in form of embeddings, to fetch this data, we will also have to work on the search using embeddings. Therefore, the retrieval process is tightly coupled with the indexing process. Whatever we use to index we will have to use to retrieve.

4.2.1 Progression of Retrieval Methods

Information Retrieval or IR is the science of searching. Whether you are searching for information in a document, or searching for document themselves, it falls under the gamut of Information Retrieval. IR has a rich history in computing starting from Joseph Marie Jacquard inventing the Jacquard Loom, the first device that could read punched cards, back in the early 19th century. Since then, IR has evolved leaps and bounds from simple to highly sophisticated search and retrieval. **Boolean retrieval** is a simple keyword-based search (like the one you encounter when you press CTRL/CMD+F on your browser or word processor) where Boolean logic is used to match documents with queries based on absence or presence of the words. Documents are retrieved if they contain the exact terms in the query, often combined with AND, NOT and OR operators. **Bag of Words (BoW)** was used quite often in the early days of NLP, Bag of Words creates a vocabulary of all the words in the documents as a vector indicating the presence or absence of each word. Consider two sentences, "The cat sat on the mat" and "The cat in the hat". The vocabulary is ["the", "cat", "in", "hat", "on", "mat"] and the first sentence is represented as vector [2, 1, 1, 1, 0, 0] while the second sentence is [2, 1, 0, 0, 1, 1]. While it is simple, it ignores the context, meaning and the order of words.

Some of these, though popular in ML and IR space, don't make sense in the context of RAG for a variety of reasons. For our purpose, we will focus on a few of the popular retrieval techniques that have been used in RAG.

TF-IDF (TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY)

TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus). It assigns higher weights to words that appear frequently in a document but infrequently across the corpus. Figure 4.3 illustrates how TF-IDF is calculated for a unigram search term.

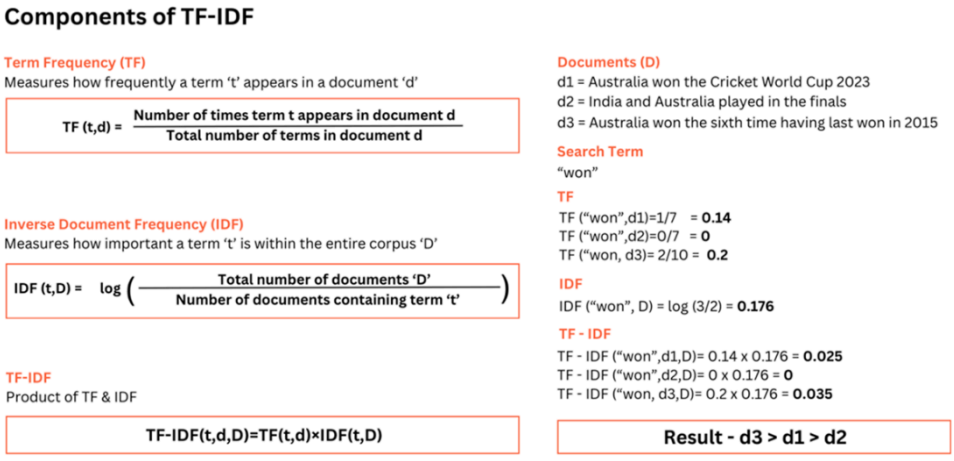

Figure 4.3 Calculating TF-IDF to rank documents based on search terms

LangChain also provides an abstract implementation of TF-IDF using retrievers from langchain_community which, in turn, leverages scikit-learn

```
# Install or Upgrade Scikit-learn
%pip install --upgrade scikit-learn

# Import TFIDFRetriever class from retrievers library
from langchain_community.retrievers import TFIDFRetriever

# Create instance of the TFIDFRetriever with texts
retriever = TFIDFRetriever.from_texts(
["Australia won the Cricket World Cup 2023",
 "India and Australia played in the finals",
 "Australia won the sixth time having last won in 2015"]
)

# Use the retriever using the invoke method
result=retriever.invoke("won")

# Print the results
print(result)
```

TF-IDF can not only be used for unigrams but also for phrases (n-grams). Even though, TF-IDF improves upon simpler search methods by emphasizing unique words, but it still lacks context and word order consideration, making it less suitable for complex tasks like RAG.

BM25 (BEST MATCH 25)

BM25 is an advanced probabilistic model used to rank documents based on the query terms appearing in each document. It is part of the family of probabilistic information retrieval models and is considered an advancement over the classic TF-IDF model. The improvement that BM25 brings is that it adjusts for the length of the documents so that longer documents do not unfairly get higher scores. Figure 4.4 illustrates shows the BM25 calculation.

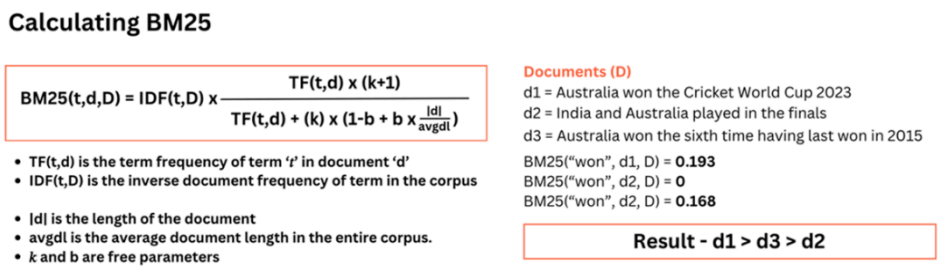

Figure 4.4 BM25 also considers the length of the documents

Like TF-IDF, LangChain also has an abstract implementation of BM25 (Okapi BM25, specifically) leveraging the rank_bm25 package.

```
# Install or Upgrade rank_bm25
%pip install --upgrade rank_bm25

# Import BM25Retriever class from retrievers library
from langchain_community.retrievers import BM25Retriever

# Create instance of the TFIDFRetriever with texts
retriever = BM25Retriever.from_texts(
["Australia won the Cricket World Cup 2023",
 "India and Australia played in the finals",
 "Australia won the sixth time having last won in 2015"]
)

# Use the retriever using the invoke method
result=retriever.invoke("Who won the 2023 Cricket World Cup?")

# Print the results
print(result)
```

For long queries instead of single keywords, the BM25 value is calculated for each word in the query and the final BM25 value for the query is a summation of the values for all the words. BM25 is a powerful tool in traditional IR, it still doesn't capture the full semantic meaning of queries and documents required for RAG applications. BM25 is generally used in RAG for quick initial retrieval and then a more powerful retriever is used to re-rank the results. We will learn about reranking later in this book, in Chapter 6, when we discuss advanced strategies for RAG.

STATIC WORD EMBEDDINGS

Static embeddings like Word2Vec and GloVe represent words as dense vectors in a continuous vector space, capturing semantic relationships based on context. For instance, "king" - "man" + "woman" approximates "queen". These embeddings can capture nuances like similarity and analogy, which BoW, TF-IDF and BM25 miss. However, while they provide a richer representation, they still lack full contextual understanding and are limited in handling polysemy (words with multiple meanings). The term “static” here highlights that the vector representation of words does not change with the context of the word in the input query.

CONTEXTUAL EMBEDDINGS

Contextual embeddings, generated by models such as BERT or OpenAI's text embeddings, produce high-dimensional, context-aware representations for queries and documents. These models, based on transformers, capture deep semantic meanings and relationships. For example, a query about "apple" will retrieve documents discussing apple the fruit or apple the technology company depending on the input query. Figure 4.4 illustrates the difference between Static and Contextual embeddings. Contextual embeddings represent a significant advancement in IR, providing the context and understanding necessary for RAG tasks. Despite being computationally intensive, contextual embeddings are the most widely used retrievers in RAG. Examples of embedding models discussed in section 3.3.2 of chapter 3 are contextual embeddings.

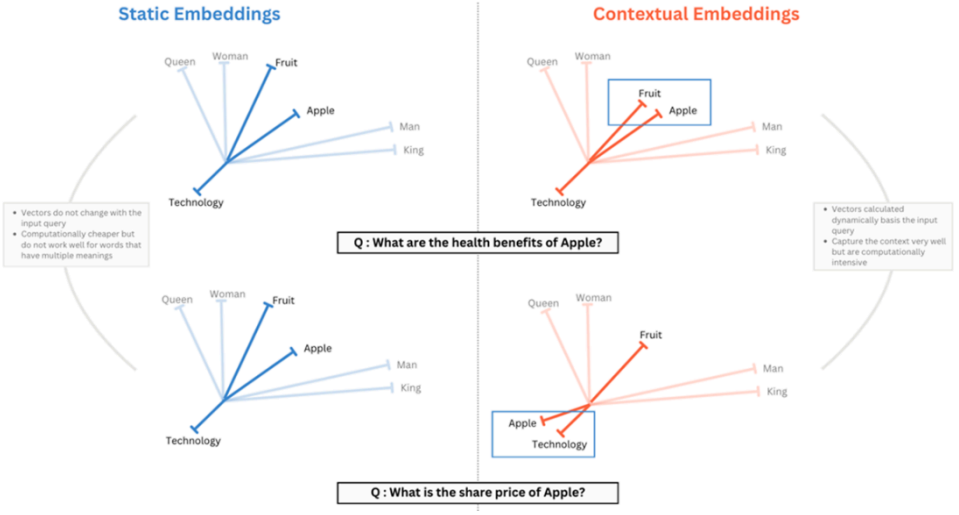

Figure 4.5 Static Embeddings vs Contextual Embeddings

Methods like TF-IDF and BM25 use frequency-based calculations to rank documents. In Embeddings (both static and contextual) ranking is done basis a similarity score. Similarity is popularly calculated using cosine of the angle between document vectors. We had discussed cosine similarity calculation in section 3.3.3 of chapter 3. Figure 4.6 illustrates the process of retrieval using embeddings.

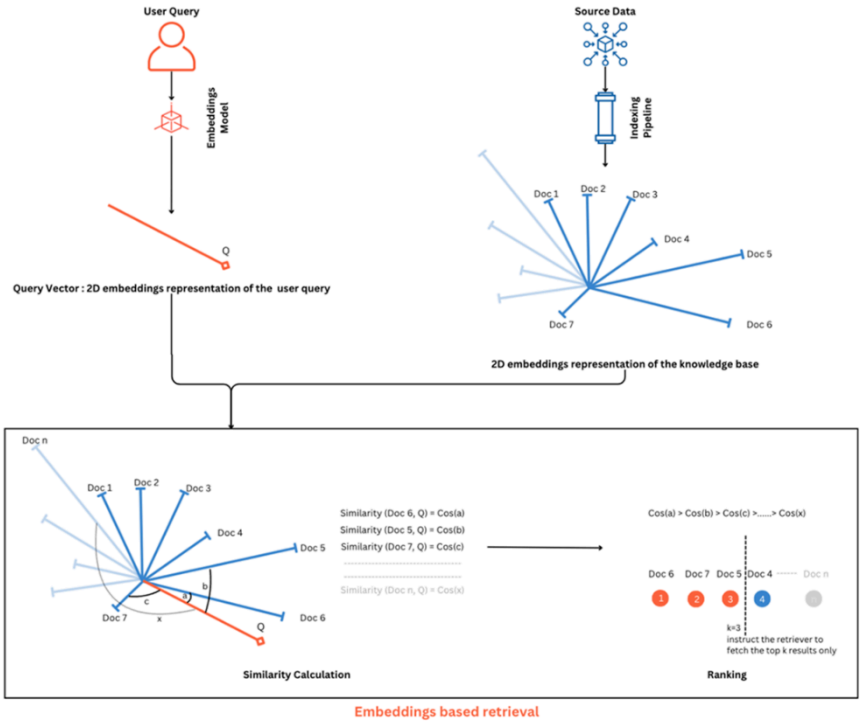

Figure 4.6 Similarity calculation and results ranking in embeddings-based retrieval technique

OTHER RETRIEVAL METHODS

While the ones discussed above are most popular in the discourse, there are other methods that can also be explored. These methods represent more recent developments and specialized approaches and are good to refer to if you want to dive deeper into the world of information retrieval –

- **Learned Sparse Retrieval**: Generate sparse, interpretable representations using neural networks. Examples: SPLADE, DeepCT, DocT5Query
- **Dense Retrieval**: Encode queries and documents as dense vectors for semantic matching. Examples: DPR (Dense Passage Retriever), ANCE, RepBERT
- **Hybrid Retrieval**: Combine sparse and dense methods for balanced efficiency and effectiveness. Examples: ColBERT, COIL
- **Cross-Encoder Retrieval**: Directly compare query-document pairs using transformer models. Example: BERT-based rerankers
- **Graph-based Retrieval**: Leverage graph structures to model relationships between documents. Examples: TextGraphs, Graph Neural Networks for IR
- **Quantum-inspired Retrieval**: Apply quantum computing principles to information retrieval. Example: Quantum Language Models (QLM)

- **Neural IR models**: Encompass various neural network-based approaches to information retrieval. Examples: NPRF (Neural PRF), KNRM (Kernel-based Neural Ranking Model)

Table 4.1 Comparison of different retrieval techniques for RAG

Technique	Key Feature	Strengths	Weaknesses	Suitability for RAG
Boolean Retrieval	Exact matching with logical operators	Simple, fast, precise	Limited relevance ranking, no partial matching	Low - too rigid
Bag of Words (BoW)	Unordered word frequency counts	Simple, intuitive	Ignores word order and context	Low - lacks semantic understanding
TF-IDF	Term weighting based on document and corpus frequency	Improved relevance ranking over BoW	Still ignores semantics and word relationships	Low-Medium - better than BoW, but limited. Used in Hybrid retrieval.
BM25	Advanced ranking function with length normalization	Robust performance, industry standard	Limited semantic understanding	Medium - good baseline for simple RAG. Used in Hybrid retrieval.
Static Embeddings	Fixed dense vector representations	Captures some semantic relationships	Context-independent, limited in polysemy handling	Medium - introduces basic semantics
Contextual Embeddings	Context-aware dense representations	Rich semantic understanding, handles polysemy	Computationally intensive	High - excellent semantic capture
Learned Sparse Retrievers	Neural-network generated sparse representations	Efficient, interpretable, some semantic understanding	May miss some semantic relationships	High - balances efficiency and semantics
Dense Retrievers	Dense vector matching for queries and documents	Strong semantic matching	Computationally intensive, less interpretable	High - excellent for semantic search in RAG
Hybrid Retrievers	Combination of sparse and dense methods	Balances efficiency and effectiveness	Complex to implement and tune	High - versatile for various RAG needs
Cross-Encoder Retrievers	Direct query-document comparison	Very accurate relevance assessment	Extremely computationally expensive	Medium-High - great for reranking in RAG
Graph-based Retrievers	Graph structure for document relationships	Captures complex relationships in data	Can be complex to construct and query	Medium-High - good for structured data in RAG

Quantum-inspired Retrievers	Quantum computing concepts in IR	Potential for handling complex queries	Emerging field, practical benefits not fully proven	Low-Medium - potentially promising but not mature
Neural IR models	Various neural network approaches to IR	Flexible, can capture complex patterns	Often require large training data, can be black-box	High - adaptable to various RAG scenarios

Table 4.1 notes the weaknesses and strengths of different retrievers. While Contextual Embeddings are the only ones that you need to know to get started with RAG, it is worthwhile understanding these other retrievers for further exploration and for cases where you want to improve the performance of the retriever. Like we discussed above, the implementation of TF-IDF using scikit-learn retriever and BM25 using rank_bm25 retriever in LangChain, there are many retrievers available that use one of the above-mentioned methodologies. We will look at some of the popular ones in the next section.

4.2.2 Popular Retrievers

Developers can build their own retrievers based on one or a combination of multiple retrieval methodologies. Retrievers are used not just in RAG but a variety of search related tasks.

For RAG, LangChain provides many integrations where the algorithms such as TF-IDF, Embeddings & similarity search, BM25 have been abstracted as retrievers for developers to use. We have already seen the ones for TF-IDF and BM25. Some of the other popular retrievers are –

VECTOR STORES AND DATABASES AS RETRIEVERS

Vector Stores can act as the retrievers taking away the responsibility from the developer to convert the query vector into embeddings and calculating similarity and ranking the results. FAISS uses a contextual embeddings model for retrieval. Other vector DBs like PineCone, Milvus and Weviate provide Hybrid Search functionality combining dense retrieval methods like embeddings and sparse methods like ANN, BM25, SPLADE etc.

CLOUD PROVIDERS

Cloud providers Azure, AWS, Google etc. also provide their retrievers. Integration with Amazon Kendra, Azure AI Search, AWS Bedrock, Google Drive, Google Vertex AI Search provide gives developers infrastructure, APIs, and tools for information retrieval of vector, keyword, and hybrid queries at scale.

WEB INFORMATION RESOURCES

Connections to information resources like Wikipedia, Arxiv, AskNews provide optimized search and retrieval from these sources.

You can check these retrievers and more in the official LangChain documentation (https://python.langchain.com/v0.2/docs/integrations/retrievers/)

This was a brief introduction to the world of retrievers. If you found the information slightly complex, you can always revisit. At this stage, the understanding of contextual embeddings will suffice. Contextual embeddings are the most popular technique for basic RAG pipelines, and we will now create a simple retriever using OpenAI embeddings.

4.2.3 A simple retriever implementation

Before we move to the next step of the generation pipeline, let us look at a simple example of a retriever. In Chapter 3, we were working on indexing the Wikipedia page for the 2023 cricket world cup. If you recall we had used embeddings from OpenAI to encode the text and used FAISS as the vector index to store the embeddings. We also stored the FAISS index in a local directory. Let's reuse this index.

```
# Install the langchain openai library
%pip install langchain-openai==0.1.6
```

```
# Import FAISS class from vectorstore library
from langchain_community.vectorstores import FAISS

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

# Set the OPENAI_API_KEY as the environment variable
import os
os.environ["OPENAI_API_KEY"] = <YOUR_API_KEY>

# Instantiate the embeddings object
embeddings=OpenAIEmbeddings(model="text-embedding-3-large")

# Load the database stored in the local directory
db=FAISS.load_local("../../Assets/Data", embeddings,
allow_dangerous_deserialization=True)

# Original Question
query = "Who won the 2023 Cricket World Cup?"

# Ranking the chunks in descending order of similarity
docs = db.similarity_search(query)
```

This similarity_search() function returns a list of matching documents ordered by a score. This score is a quantification of the similarity between the query and the document and is hence called the similarity score. In this example, the vector index's inbuilt similarity search feature was used for retrieval. As one of the retrievers we discussed in section 4.2.2, the vector store itself acted as the retriever. This is the most basic implementation of a retriever in the generation pipeline of a RAG-enabled system. This method of retrieval is enabled by embeddings. We used the text-embedding-3-large from OpenAI. FAISS calculated the similarity score based on these embeddings.

Retrievers are the backbone of RAG enabled systems. The quality of the retriever has a great bearing on the quality of generated output. In this section we learnt about vanilla retrieval methods. There are multiple strategies that are used when designing production grade systems. We will read about these advanced strategies in Chapter 6. Now that we have gained an understanding of the retrievers, we will move on to the next important step – augmentation.

4.3 Augmentation

A retriever fetches the information (or documents) that are most relevant to the user query. But, what next? How do we use this information? The answer is quite intuitive. If you recall the discussion in Chapter 1, the input to an LLM is a natural language prompt. This information fetched by the retriever should also be sent to the LLM in form of a natural language prompt. This process of combining the user query and the retrieved information is called *augmentation*.

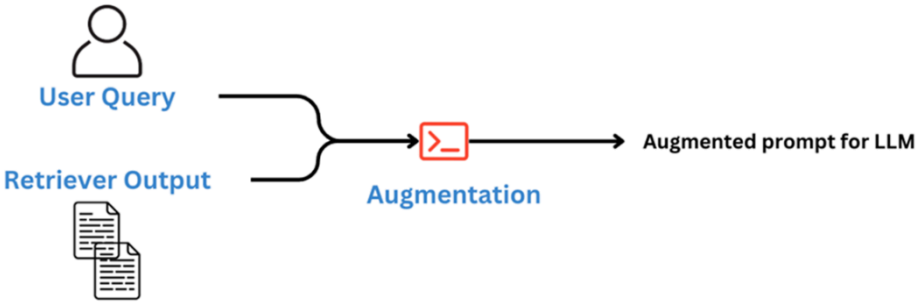

Figure 4.7 Simple augmentation combines the user query with retrieved documents to send to the LLM

The step of Augmentation in RAG largely falls under the discipline of prompt engineering. Prompt engineering can be defined as the technique of giving instructions to an LLM to attain a desired outcome. The goal of Prompt Engineering is to construct the prompts to achieve accuracy and relevance in the LLM responses with respect to the desired outcome(s). At the first glance, augmentation is quite simple – just add the retrieved information to the query. However, there are some nuanced augmentation techniques that help improve the quality of the generated results.

4.3.1 RAG Prompt Engineering Techniques

Prompt engineering as a discipline has, sometimes, been dismissed as being too simplistic to be called engineering. You may have heard the phrase, "English is the new programming language". It is true that interaction with LLMs is in natural language. However, what is also true is that the principles of programming are not the language in which code is written but the logic in which the machine is instructed. With that in mind, let us dive deep into different logical approaches that can be taken to augment the user query with the retrieved information.

CONTEXTUAL PROMPTING

To understand a simple augmentation technique, let us revisit what we did in Chapter 1. Recall our example of "Who won the 2023 Cricket World Cup?". We copied an excerpt from the Wikipedia article. This excerpt is the retrieved information. We then added this information to the prompt and provided an extra instruction – "Answer only based on the context provided below". Figure 4.8 illustrates this example.

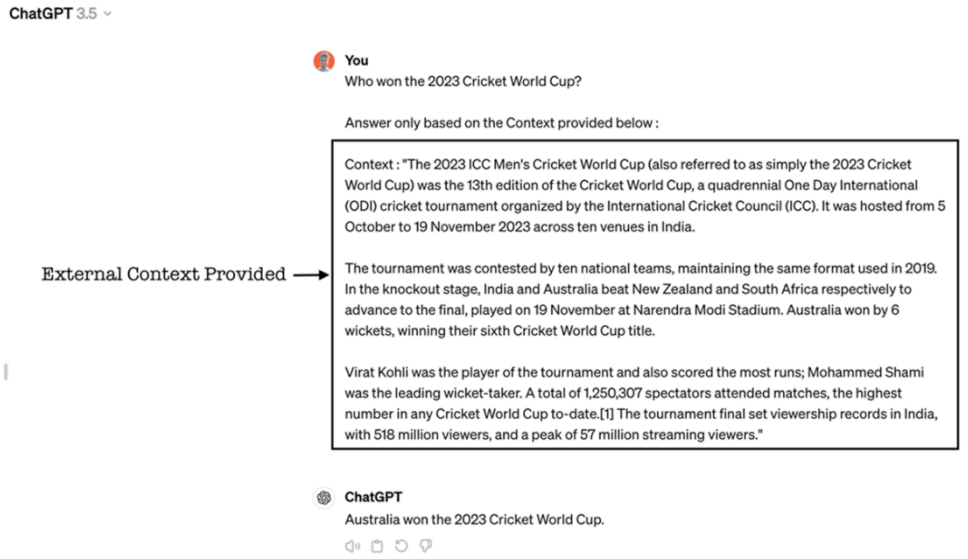

Figure 4.8 Information augmented to the original question with an added instruction

By adding this instruction, we have set up our generation to focus only on the provided information and not from LLM's internal knowledge (or parametric knowledge). This is a simple augmentation technique which is also referred to as **Contextual Prompting**. Please note that the instruction can be given in any linguistic construct. For example, we could have added the instruction at the beginning of the prompt as – "Given the context below answer the question – Who won the 2023 Cricket World Cup. Information: <Wikipedia excerpt>". We can also reiterate the instruction at the end of the prompt – "Remember to answer only based on the context provided and not from any other source".

CONTROLLED GENERATION PROMPTING

Sometimes, the information might not be present in the retrieved document. This happens when the documents in the knowledge base do not have any information relevant to the user query. The retriever might still fetch some documents that are the closest to the user query. In these cases, the chances of hallucination increase because the LLM will still try to follow the instruction of answering the question. To avoid this scenario an additional instruction is added which tells the LLM to not give an answer if the retrieved document does not have proper information to answer the user question. Something like – "If the question cannot be answered based on the provided context, say I don't know." In the context of RAG, this technique is particularly valuable because it ensures that the model's responses are grounded in the retrieved information. If the relevant information hasn't been retrieved or isn't present in the knowledge base, the model is instructed to acknowledge this lack of information rather than attempting to generate a potentially incorrect answer.

FEW SHOT PROMPTING

It has been observed that while generating responses, LLMs adhere quite well to examples provided in the prompt. If you want the generation to be in a certain format or style, it is recommended to provide a few examples. In RAG, while providing the retrieved information in the prompt, we can also specify certain examples to help guide the generation in the way we need the retrieved information to be used. This technique is called **Few Shot Prompting**. Here "shot" refers to the examples given in the prompt. Figure 4.9 illustrates a prompt that includes two examples with the question.

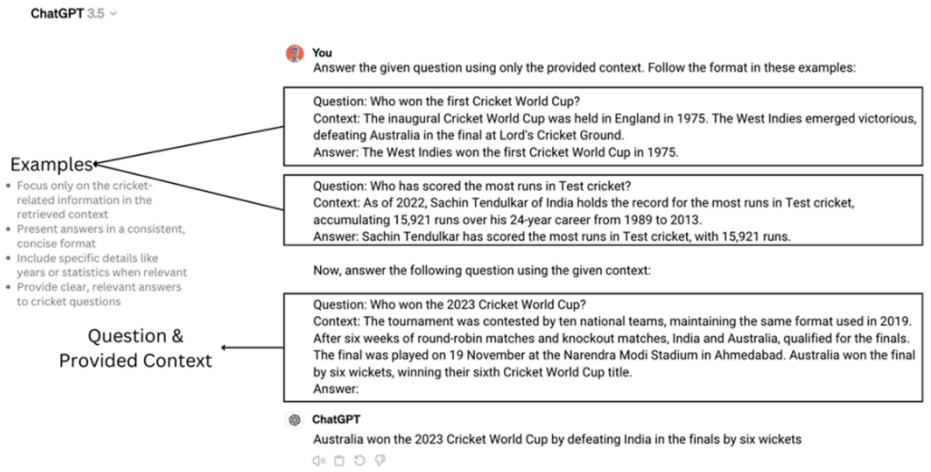

Figure 4.9 Example of Few Shot Prompting in the context of RAG

You might come across terms like one shot prompting or two shot prompting which replaces the term "Few" with the number of examples given. Conversely, when no example is given and the LLM is expected to answer correctly, the technique is also called Zero Shot Prompting.

CHAIN OF THOUGHT PROMPTING

It has been observed that the introduction of intermediate "reasoning" steps, improves the performance of LLMs in tasks that require complex reasoning like arithmetic, common sense, and symbolic reasoning. The same can be applied in the context of RAG. This is called Chain of Thought or CoT prompting. In figure 4.10, we ask ChatGPT to analyze the performance of two teams based on the retrieved information.

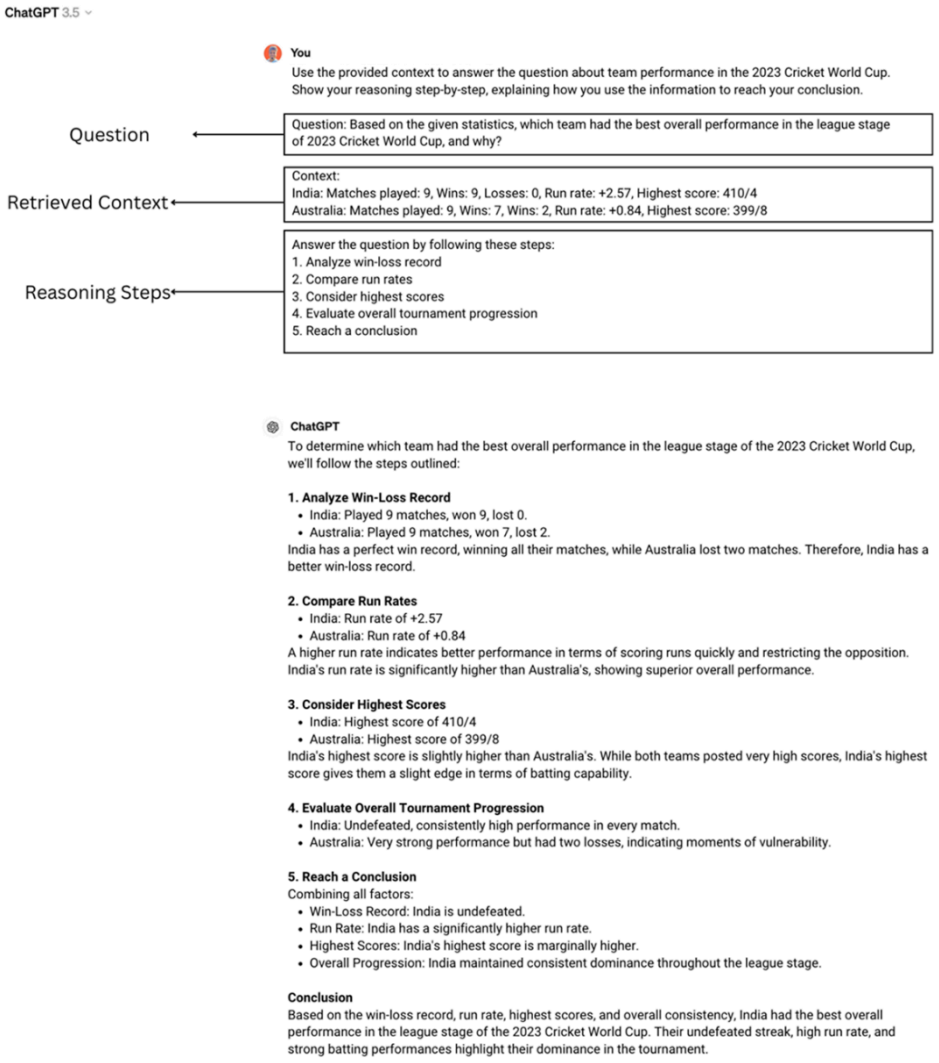

Figure 4.10 Chain-of-Thought (CoT) prompting for reasoning tasks

The CoT prompting approach can also be combined with Few Shot Prompting technique where a few examples of reasoning are provided before the final question. Creating these examples is a manually intensive task. In Auto-CoT, the examples are also created using an LLM.

OTHER ADVANCED PROMPTING TECHNIQUES

Prompt engineering is becoming an intricate discipline. Ongoing research constantly presents new improvements in prompting techniques. To dive deeper into prompt engineering, some of the following techniques can be referred to –

- **Self Consistency**
  While CoT uses a single Reasoning Chain in Chain of Thought prompting, Self-Consistency aims to sample multiple diverse reasoning paths and use their respective generations to arrive at the most consistent answer
- **Generated Knowledge Prompting**
  This technique explores the idea of prompt-based knowledge generation by dynamically constructing relevant knowledge chains, leveraging models' latent knowledge to strengthen reasoning.
- **Tree of Thoughts Prompting**
  This technique maintains an explorable tree structure of coherent intermediate thought steps aimed at solving problems.
- **Automatic Reasoning and Tool-use (ART)**
  ART framework automatically interleaves model generations with tool use for complex reasoning tasks. ART leverages demonstrations to decompose problems and integrate tools without task-specific scripting.
- **Automatic Prompt Engineer (APE)**
  The APE framework automatically generates and selects optimal instructions to guide models. It leverages a large language model to synthesize candidate prompt solutions for a task based on output demonstrations.
- **Active Prompt**
  Active-Prompt improves Chain-of-thought methods by dynamically adapting Language Models to task-specific prompts through a process involving query, uncertainty analysis, human annotation, and enhanced inference.
- **ReAct Prompting**
  ReAct integrates LLMs for concurrent reasoning traces and task-specific actions, improving performance by interacting with external tools for information retrieval. When combined with CoT, it optimally utilizes internal knowledge and external information, enhancing interpretability and trustworthiness of LLMs.
- **Recursive Prompting**
  Recursive prompting breaks down complex problems into sub-problems, solving them sequentially using prompts. This method aids compositional generalization in tasks like math problems or question answering, with the model building on solutions from previous steps.

Table 4.2 summarizes the different prompting techniques mentioned above. Prompt Engineering for augmentation is an evolving discipline. It is important to note that there is a lot of scope for creativity in writing prompts for RAG applications. Efficient prompting has a significant impact on the generated output. The kind of prompts you use will depend a lot on your use case and the nature of information in the knowledge base.

Table 4.2 Comparison of prompting techniques for augmentation

Technique	Description	Key Advantage	Best Use Case	Complexity
Contextual Prompting	Adds retrieved information to the prompt with instructions to focus on provided context	Ensures focus on relevant information	General RAG queries	Low
Controlled Generation Prompting	Instructs the model to say "I don't know" when information is not available	Reduces hallucination risk	When accuracy is critical	Low
Few Shot Prompting	Provides examples in the prompt to guide response format and style	Improves output consistency and format adherence	When specific output format is required	Medium
Chain of Thought (CoT) Prompting	Introduces intermediate reasoning steps	Improves performance on complex reasoning tasks	Complex queries requiring step-by-step analysis	Medium
Self Consistency	Samples multiple diverse reasoning paths	Improves answer consistency and accuracy	Tasks with multiple possible reasoning approaches	High
Generated Knowledge Prompting	Dynamically constructs relevant knowledge chains	Leverages model's latent knowledge	Tasks requiring broad knowledge application	High
Tree of Thoughts Prompting	Maintains an explorable tree structure of thought steps	Allows for more comprehensive problem-solving	Complex, multi-step problem solving	High
Automatic Reasoning and Tool-use (ART)	Interleaves model generations with tool use	Enhances problem decomposition and tool integration	Tasks requiring external tool use	Very High
Automatic Prompt Engineer (APE)	Automatically generates and selects optimal instructions	Optimizes prompts for specific tasks	Prompt optimization for complex tasks	Very High
Active Prompt	Dynamically adapts LMs to task-specific prompts	Improves task-specific performance	Tasks requiring adaptive prompting	High
ReAct Prompting	Integrates reasoning traces with task-specific actions	Improves performance and interpretability	Tasks requiring both reasoning and action	High

Recursive Prompting	Breaks down complex problems into sub-problems	Aids in compositional generalization	Complex, multi-step problems	High

We have already built a simple retriever in the previous section. We will now execute augmentation with a simple contextual prompt with controlled generation.

4.3.2 A simple augmentation prompt creation

In section 4.2.3, we were able to implement a FAISS based retriever using OpenAI embeddings. We will now make use of this retriever and create the augmentation prompt.

```
# Import FAISS class from vectorstore library
from langchain_community.vectorstores import FAISS

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

# Set the OPENAI_API_KEY as the environment variable
import os
os.environ["OPENAI_API_KEY"] = <YOUR_API_KEY>

# Instantiate the embeddings object
embeddings=OpenAIEmbeddings(model="text-embedding-3-large")

# Load the database stored in the local directory
db=FAISS.load_local("../../Assets/Data", embeddings,
allow_dangerous_deserialization=True)

# Original Question
query = "Who won the 2023 Cricket World Cup?"

# Ranking the chunks in descending order of similarity
docs = db.similarity_search(query)

# Selecting first chunk as the retrieved information
retrieved_context=docs[0]

# Creating the prompt
augmented_prompt=f""""

Given the context below answer the question.

Question: {query}
```

```
Context : {retrieved_context}

Remember to answer only based on the context provided and not from any other source.

If the question cannot be answered based on the provided context, say I don’t know.

”””
```

With the augmentation step complete, we are now ready to send the prompt to the LLM for the generation of the desired outcome. We will now understand how Large Language Models generate text and the nuances of generation.

4.4 Generation

Generation is the final step of this pipeline. While LLMs may be used in any of the previous steps in the pipeline, the generation step is completely reliant on the LLM. The most popular LLMs are the ones being developed by OpenAI, Anthropic, Meta, Google, Microsoft and Mistral amongst other developers. While text generation is the core capability of LLMs we are now seeing multimodal models that can handle images and audio along with text. At the same time, researchers are developing faster and smaller models.

In this section, we will discuss the factors that can be helpful in choosing a language model for your RAG based system. We will then continue with our example of the retriever and augmented prompt that we have built so far and complete it by adding the generation step.

4.4.1 Categorization of LLMs and suitability for RAG

As of June 2024, there are over a hundred LLMs available to use and new ones are coming out every week. How then do we decide which LLM to choose for our RAG enabled system. To do this, we will discuss three themes under which we can broadly categorize LLMs. –

1. Based on how they have been trained
2. Based on how they can be accessed
3. Based on their size

We will discuss the LLMs under these themes and understand the factors that may influence the choice of the LLM for RAG.

ORIGINAL MODELS VS FINE-TUNED MODELS

Training a large language model takes massive amounts of data and computational resources. LLMs training is done through an unsupervised learning process. All modern LLMs are autoregressive models and are trained to generate the next token in a sequence. These massive pre-trained LLMs are also called **foundation models**.

The question that you may ask is that if LLMs just predict the next tokens in a sequence, how are we able to ask questions and chat with these models. The answer lies in what we call **supervised fine-tuning or SFT**

Supervised fine-tuning is a process used to adapt a pre-trained language model for specific tasks or behaviors like question-answering or chat. It involves further training a pre-trained foundation model on a labeled dataset, where the model learns to map inputs to specific desired outputs. You start with a pre-trained model, prepare a labelled dataset for target task and train the model on this dataset which adjusts the model parameters to perform better on the target task. Figure 4.11 gives an overview of the supervised fine-tuning process.

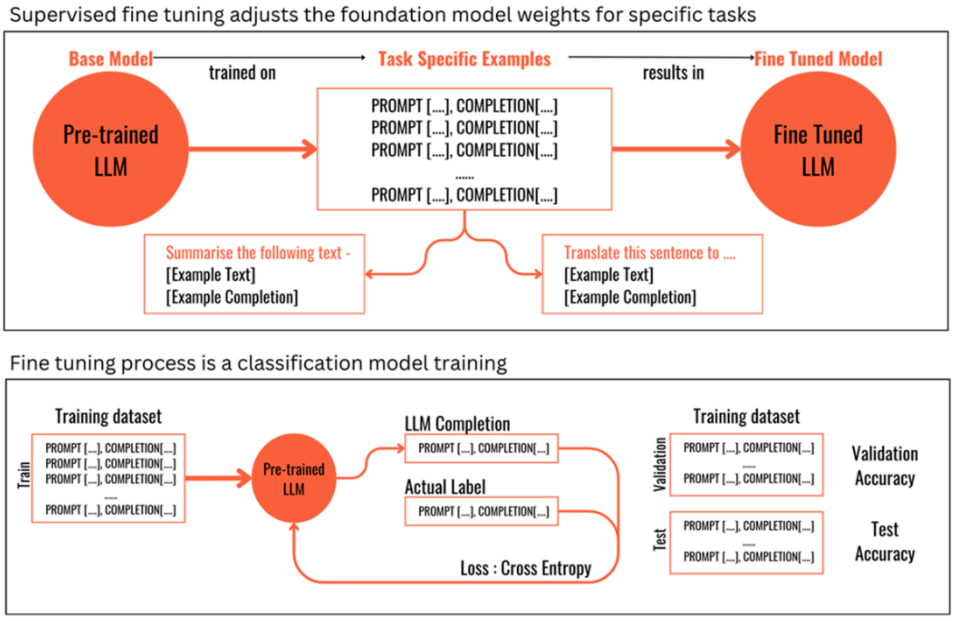

Figure 4.11 Supervised fine tuning is a classification model training process

While foundation models generalize well for a wide array of tasks, there are several use cases where the need for a fine-tuned model arises. Domain adaptation for specialized fields like law and healthcare, task specific optimization like classification and NER, conversational AI, personalization are some use cases where you may observe a fine-tuned model performing better.

Specifically, in the context of RAG, there are some criteria that should be considered while choosing between a foundation model and fine-tuning one -

- **Domain Specificity**: Foundation models have broader knowledge and can handle a wider range of topics and queries for general purpose RAG systems. If your RAG application is a specialized one say, dealing with patient records or instruction manuals for heavy machinery, you may find that fine-tuning the model for specific domains may lead to a better performance.

- **Retrieval Integration**: If you observe that a foundation model that you are using is not integrating the retrieved information well, a fine-tuned model trained to better utilize information can result in better quality of generations.
- **Deployment Speed**: A foundation model can be quickly deployed since there is no additional training required. For fine-tuning a model, you will need to spend time in gathering training data and the actual training of the model.
- **Customization of Responses**: For generating results in a specific format or custom style elements like tone or vocabulary, a fine-tuned model may result in better adherence to the requirements compared to a foundation models.
- **Resource efficiency**: Fine-tuning a model requires more storage and computational resources. Depending on the scale of deployment, the costs may be higher for a fine-tuned model.
- **Ethical alignment**: A fine-tuned model allows for better control over the responses in adherence to ethical guidelines and even certain privacy aspects.

A summary of the criteria discussed above is presented in table 4.3 below.

Table 4.3 Criteria for choosing between foundation and fine-tuned models

Criteria	Better Suitability	Explanation
Domain Specificity	Fine-Tuned Models	Better performance for specialized applications (e.g., patient records, instruction manuals)
Retrieval Integration	Fine-Tuned Models	Can be trained to better utilize retrieved information
Deployment Speed	Foundation Models	Quicker deployment with no additional training required
Customization of Responses	Fine-Tuned Models	Better adherence to specific format, style, tone, or vocabulary requirements
Resource Efficiency	Foundation Models	Requires less storage and computational resources
Ethical Alignment	Fine-Tuned Models	Allows better control over responses for ethical guidelines and privacy

Fine-tuned models give you better control over you RAG systems but are costly. There's also a risk of over-reliance on retrieval and a potential trade-off between RAG performance and inherent LLM language abilities. Therefore, whether to use a foundation model or fine-tuning one should be dependent on the improvements you are targeting, availability of data, cost and other trade-offs. The general recommendation is to start experimenting with a foundation model and then progress to supervised fine-tuning for improvement in performance.

OPEN-SOURCE VS PROPRIETARY MODELS

Software development and distribution is represented by two fundamentally different approaches – Open-Source vs Proprietary Software. The world of LLMs is no different. Some LLM developers like Meta and Mistral have made the model weights public for fostering collaboration and community driven innovation. On the other hand, pioneers like OpenAI, Anthropic and Google have kept the models closed offering support, managed services and better user experience.

For RAG enabled systems, open-source models give you the flexibility of customization, deployment method, transparency but warrant the need of the necessary infrastructure to maintain the models. Proprietary model providers might be costlier for high volumes but provide regular updates, ease of use, scalability, faster development amongst other things. Some proprietary model providers like OpenAI have pre-built RAG capabilities. Your choice of which type of model you choose may depend on some of the following criteria –

- **Customization**: Open-Source LLMs are generally considered better for customizations like deep integration with custom retrieval mechanisms. A better control over fine-tuning is also something that Open Source LLMs allow for. Customization of proprietary models is limited to API capabilities.
- **Ease of use**: Foundation models, on the other hand, are much easier to use. Some of the models like OpenAI, Cohere etc. offer optimized, pre-build RAG solutions.
- **Deployment Flexibility**: Open-Source models can be deployed according to your preference (private cloud, on-prem) while proprietary models are managed by the providers. This also has a bearing on data security and privacy. Most proprietary model providers are also, now, offering multiple deployment options.
- **Cost**: Open-Source LLMs may come with upfront infrastructure costs while proprietary models are priced based on usage. Long term costs and query volumes are considerations to choose between open-source and proprietary models. Large scale deployments may favor the usage of open-source models.

The choice between open-source and proprietary models for RAG depends on factors such as the scale of deployment, specific domain requirements, integration needs, and the importance of customization in the retrieval and generation process. Apart from these, the need for knowledge update, transparency, scalability, structure of data, compliance, etc. will determine the choice of the model. A summary of the discussion is presented in table 4.4

Table 4.4 Criteria for choosing between open-source and proprietary models

Criteria	Better Suitability	Explanation
Customization	Open-Source	Allows deeper integration with custom retrieval mechanisms and better control over fine-tuning
Ease of Use	Proprietary	Offer optimized, pre-built RAG solutions and are generally easier to use
Deployment Flexibility	Open-Source	Can be deployed on private cloud or on-premises, offering more options
Cost for Large-Scale Deployment	Open-Source	May be more cost-effective for large-scale deployments despite upfront infrastructure costs
Data Security and Privacy	Open-Source	Offers more control over data, though some private models now offer various deployment options
Regular Updates and Support	Proprietary	Typically provide regular updates and better support

A hybrid approach is also not ruled out. At a PoC stage, a proprietary model may make sense for quick experimentation.

Examples of popular proprietary models –

- GPT series by OpenAI (https://platform.openai.com/docs/models)
- Claude series by Anthropic (https://www.anthropic.com/claude)
- Gemini series by Google (https://ai.google.dev/gemini-api/docs/models/gemini)
- Command R series by Cohere (https://cohere.com/command)

Examples of popular open-source models –

- Llama series by Meta (https://llama.meta.com/)
- Mistral (https://docs.mistral.ai/getting-started/models/)

MODEL SIZES

LLMs come in various sizes that is typically measured by the number of parameters they contain. The size of the model greatly impacts the capabilities along with the resource requirements.

Larger models have several billion and even trillions of parameters. These models exhibit superior performance in reasoning abilities, language understanding and have broader knowledge. They can generate more coherent text and their responses are contextually more accurate. However, these larger models have significantly high computation, storage and energy requirements.

Smaller models with parameter sizes in millions or a few billion offer benefits such as faster inference times, lower resource usage and easier deployment on edge devices or resource constrained environments. Researchers and developers continue to explore methods to achieve large model performance with smaller and more efficient architectures.

For a RAG enabled system, the following are few considerations that should be assessed -

- **Resource Constraints**: Small models have a much lower resource usage. Lightweight RAG applications with faster inference can be built with smaller models.
- **Reasoning Capability**: On the other spectrum of resource constraints is the language processing ability of the model. Large models are more suited for complex reasoning tasks and can deal with ambiguity in the retrieved information. Smaller models, therefore, will rely heavily on the quality of retrieved information.
- **Deployment Options**: The size of large models makes it difficult to deploy on edge devices. This is a flexibility that smaller models provide bring RAG applications to a wide range of devices and environments.
- **Context handling**: Large models may be better at integrating multiple pieces of retrieved information in RAG systems since they have longer context windows. Large models are also better at handling diverse queries while small models struggle with out-of-domain queries. Large models might perform better in RAG systems with diverse or unpredictable query types.

In practice, most RAG applications are being built on large models. However, smaller models make more sense in the long-term adoptability and application of the technology. The factors discussed above are summarized in table 4.5

Table 4.5 Criteria for choosing between small and large models

Criteria	Better Suitability	Explanation
Resource Constraints	Small Models	Lower resource usage, suitable for lightweight RAG applications
Reasoning Capability	Large Models	Better for complex reasoning tasks and handling ambiguity in retrieved information
Deployment Options	Small Models	More flexible, can be deployed on edge devices and resource-constrained environments
Context Handling	Large Models	Better at integrating multiple pieces of retrieved information, longer context windows
Query Diversity	Large Models	Handle diverse and unpredictable query types better
Inference Speed	Small Models	Faster inference times, suitable for applications requiring quick responses

Examples of popular Small Language Models –

- Phi-3 by Microsoft (https://azure.microsoft.com/en-us/products/phi-3)
- Gemma by Google (https://ai.google.dev/gemma)

The choice of the Large Language Model is a core consideration in your RAG enabled system. This requires careful consideration and iterations. The performance of your system may need you to experiment and adapt your choice of the LLM.

The list of LLMs is fast becoming endless. What this means for developers and businesses is that the technology has truly been democratized. While all LLMs have their unique propositions and architecture, for practical applications there are a wide array of choices available to us. While simple RAG applications may rely on a single LLM provider, for more complex applications a multi-LLM strategy may be beneficial.

We have implemented a simple retriever and created an augmented prompt. In the last section of this chapter, we will now round up the pipeline by creating the generation step.

4.4.2 Completing the RAG pipeline: Generating using LLMs

We have built a simple retriever using FAISS and OpenAI embeddings and, we created a simple augmented prompt. Now we will use OpenAI's latest model, GPT-4o, to generate the response.

```
# Import FAISS class from vectorstore library
from langchain_community.vectorstores import FAISS

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

# Set the OPENAI_API_KEY as the environment variable
import os
os.environ["OPENAI_API_KEY"] = <YOUR_API_KEY>

# Instantiate the embeddings object
embeddings=OpenAIEmbeddings(model="text-embedding-3-large")

# Load the database stored in the local directory
db=FAISS.load_local("../../Assets/Data", embeddings,
allow_dangerous_deserialization=True)

# Original Question
query = "Who won the 2023 Cricket World Cup?"

# Ranking the chunks in descending order of similarity
docs = db.similarity_search(query)

# Selecting first chunk as the retrieved information
retrieved_context=docs[0]

# Creating the prompt
augmented_prompt=f"""
```

```
Given the context below answer the question.

Question: {query}

Context : {retrieved_context}

Remember to answer only based on the context provided and not from any other source.

If the question cannot be answered based on the provided context, say I don't know.

"""
# Importing the OpenAI library
from openai import OpenAI

# Instantiate the OpenAI client
client = OpenAI()

# Make the API call passing the augmented prompt to the LLM
response = client.chat.completions.create(
  model="gpt-4o",
  messages= [
    {"role": "user", "content": augmented_prompt}
            ]
)

# Extract the answer from the response object
answer=response.choices[0].message.content

print(answer)
```

And there it is. We have built a generation pipeline, albeit a very simple one. It can now fetch information from the knowledge base and generate an answer that is pertinent to the question asked and rooted in the knowledge base. Try asking a different question to see how well the pipeline generalizes.

We have now covered all three - Retrieval, Augmentation and Generation – steps of the Generation pipeline. With the knowledge of the Indexing pipeline (covered in chapter 3) and the Generation pipeline, you are now all set to create a basic RAG enabled system. What we have discussed so far can be termed a naïve implementation of RAG. Naïve RAG can be marred by inaccuracies. It can be inefficient in retrieving and ranking information correctly. The LLM can ignore the retrieved information and still hallucinate. To discuss and address these challenges, in chapter 6, we will discuss advanced strategies that allow for more complex and better performing RAG systems.

But before that, the question of evaluating the system arises. Is it generating the responses on the expected lines? Is the LLM still hallucinating? Before trying to improve the performance of the system we need to be able to measure and benchmark it. That is what we will do in chapter 5. We will look at the evaluation metrics and the popular benchmarks for RAG.

4.5 Summary

RETRIEVAL

- Retrieval is the process of finding relevant information from the knowledge base based on a user query. Retrieval is a search problem to match documents with input queries.
- The popular retrieval methods for RAG include:
    - TF-IDF (Term Frequency-Inverse Document Frequency):
      Statistical measure of word importance in a document relative to a corpus
      Can be implemented using LangChain's TFIDFRetriever
    - BM25 (Best Match 25):
      Advanced probabilistic model, improvement over TF-IDF
      Adjusts for document length
      Can be implemented using LangChain's BM25Retriever
    - Static Word Embeddings:
      Represent words as dense vectors (e.g., Word2Vec, GloVe)
      Capture semantic relationships but lack full contextual understanding
    - Contextual Embeddings:
      Produced by models like BERT or OpenAI's text embeddings
      Provide context-aware representations
      Most widely used in RAG despite being computationally intensive
    - Advanced retrieval methods like Learned Sparse Retrieval, Dense Retrieval, Hybrid Retrieval, Cross-Encoder Retrieval, Graph-based Retrieval, Quantum-inspired Retrieval, Neural IR models can be explored for further deep dive

- Most advanced implementations will include a hybrid approach of methods
- Vector Stores and Databases (e.g., FAISS, PineCone, Milvus, Weviate), Cloud Provider Solutions (e.g., Amazon Kendra, Azure AI Search, Google Vertex AI Search), Web Information Resources (e.g., Wikipedia, Arxiv, AskNews) are some of the popular retriever integrations provided by LangChain
- The choice of retriever depends on factors like accuracy, speed, and compatibility with the indexing method.

AUGMENTATION

- Augmentation combines the user query with retrieved information to create a prompt for the LLM.
- Prompt engineering is crucial for effective augmentation, aiming for accuracy and relevance in LLM responses.
- Key prompt engineering techniques for RAG include:
    - Contextual Prompting: Adding retrieved information with instructions to focus on provided context.
    - Controlled Generation Prompting: Instructing the LLM to admit lack of knowledge when information is insufficient.
    - Few Shot Prompting: Providing examples to guide the LLM's response format or style.
    - Chain of Thought (CoT) Prompting: Introducing intermediate reasoning steps for complex tasks.
    - Advanced techniques like Self Consistency, Generated Knowledge Prompting, and Tree of Thought.
- The choice of augmentation technique depends on the task complexity, desired output format, and LLM capabilities.

GENERATION

- Generation is the final step where the LLM produces the response based on the augmented prompt.
- LLMs can be categorized based on how they've been trained, how they can be accessed and the number of parameters they have
- Supervised Fine Tuning or SFT improves context use, domain optimization, enhances coherence and enables source attribution but comes with challenges like cost, risk of over-reliance on retrieval and potential trade-offs with inherent LLM abilities
- Choice between open source and proprietary LLMs depends on customization needs, long-term costs and data sensitivity
- Larger Models come with superior reasoning, language understanding, broader knowledge, and generate more coherent and contextually accurate responses but come with high computational and resource requirements. Smaller models allow faster inference, lower resource usage and are easier deployment on edge devices or resource-constrained environments but do not have the same language understanding abilities as large models
- Popular LLMs include offerings from OpenAI, Anthropic, Google, etc. and open-source models are available through platforms like HuggingFace.
- The choice of LLM depends on factors like performance requirements, resource constraints, deployment environment, and data sensitivity.
- The choice of LLM for RAG systems requires careful consideration, experimentation, and potential adaptation based on performance

5 RAG Evaluation: Accuracy, Relevance, Faithfulness

This chapter covers

- The need and requirements for evaluating RAG pipelines
- Metrics, Frameworks and Benchmarks for RAG evaluation
- Current limitations and future course of RAG evaluation

In Chapter 3 & 4, we discussed the development of RAG systems via the Indexing and the Generation pipeline. The promise of RAG is to reduce hallucinations and ground the LLM responses in provided context. This is done by creating a non-parametric memory or knowledge base for the system and then retrieving information from this knowledge base.

In this chapter, we will cover the methods to evaluate how well the RAG system is functioning. We need to make sure that the components of the two RAG pipelines are performing in accordance with the expectations. At a high level, we need to make sure that the information that is being retrieved is relevant to the input query and that the LLM is generating responses that are grounded in the retrieved context. To this end there have been several frameworks that have been developed over time. We will discuss some popular frameworks and the metrics that they calculate.

There is a second aspect to evaluation. While the frameworks allow for the calculation of metrics, how do you make sure that your RAG pipelines are working better than pipelines developed by other developers. The evaluations cannot be done in isolation. For this purpose, several benchmarks have been established. These benchmarks evaluate the RAG systems on preset data, like question answer sets, for accurate comparison of different RAG pipelines. These benchmarks help developers evaluate the performance of their systems vis-à-vis those developed by other developers.

Finally, like RAG technique, the research on RAG evaluations is still in-progress. There are some limitations that still exist in the current set of evaluation parameters. We will discuss these limitations and some ideas on the way forward for RAG evaluations.

By the end of this chapter, you should –

- Know the fundamentals of RAG evaluations.
- Be aware of the popular frameworks, metrics and benchmarks for RAG evaluation.
- Understand the limitations and best practices.
- Be able to evaluate the RAG pipeline in python.

For RAG to live up to the promise of grounding the LLM responses in data, you will need to go beyond the simple implementation of indexing, retrieval, augmentation and generation. We will discuss these advanced strategies in Chapter 6. However, to improve something, you need to first measure the performance. RAG evaluations help in setting up the baseline of your RAG system performance for you to then improve it. We will first look at the fundamental aspects of evaluating RAG systems.

5.1 Key Aspects of RAG evaluation

Building a PoC RAG pipeline is not overtly complex. It is achievable through brief training and verification on a limited set of examples. However, to enhance its robustness, thorough testing on a dataset that accurately mirrors the production use case is imperative. RAG pipelines can suffer from hallucinations of their own. This can be because -

- The retriever fails to retrieve the entire context or retrieves irrelevant context
- The LLM, despite being provided the context, does not consider it
- The LLM instead of answering the query picks irrelevant information from the context

Retrieval and Generation are two processes that need special focus from an evaluation perspective. This is because these two steps produce outputs that can be evaluated. (While indexing and augmentation will have a bearing on the outputs, they themselves do not produce measurable outcomes) We will now ask a few questions of these two processes –

Retrieval –

- How good is the retrieval of the context from the knowledge base?
- Is it relevant to the query?
- How much noise (irrelevant information) is present?

Generation –

- How good is the generated response?
- Is the response grounded in the provided context?
- Is the response relevant to the query?

You can ask many more questions like these to assess the performance of your RAG system. Contemporary research has discovered certain scores to assess the quality and abilities of a RAG system. We will discuss three predominant quality scores and four main abilities in this section.

5.1.1 Quality Scores

There are three quality score dimensions prevalent in the discourse on RAG evaluation. These quality scores measure the quality of retrieval and the quality of generation.

1. **Context Relevance**: This dimension evaluates how relevant the retrieved information or context is to the user query. It calculates metrics like the precision and recall with which context is retrieved from the knowledge base.
2. **Answer Faithfulness** (also called groundedness): This dimension evaluates if the answer generated by the system is using the retrieved information or not.
3. **Answer Relevance**: This dimension evaluates how relevant the answer generated by the system is to the original user query.

We will discuss how these scores are calculated in section 5.2

5.1.2 Required Abilities

The quality scores are important to measure how well the retrieval and the generation components of the RAG system are performing. If you look at the RAG system at an overall level, there are certain critical abilities that the system should possess.

- **Noise Robustness**: It is impractical to assume that the information stored in the knowledge base for RAG systems is perfectly curated to answer the questions that can be potentially asked of the system. It is very probable that a document is related to the user query but does not have any meaningful information to answer the query. The ability of the RAG system to separate these noisy documents from the relevant ones is Noise Robustness.
- **Negative Rejection**: By nature, Large Language Models always generate text. It is possible that there is absolutely no information about the user query in the documents in the knowledge base. The ability of the RAG system to not give an answer when there is no relevant information is Negative Rejection.
- **Information Integration**: It is also very likely that to answer a user query comprehensively the information must be retrieved from multiple documents. This ability of the system to assimilate information from multiple documents is Information Integration.
- **Counterfactual Robustness**: Sometimes the information in the knowledge base might itself be inaccurate. A high-quality RAG system should be able to address this and reject known inaccuracies in the retrieved information. This ability is Counterfactual Robustness

Noise robustness in an ability that the retrieval component should possess, and other abilities are largely related to the generation component.

Apart from these another capability that often finds a mention is the **latency**. Latency, though it is a non-functional requirement, is quite critical in generative AI applications. Latency is the delay that happens between the user query and the response. You may have observed that LLMs themselves have considerable latency before the final response is generated. Add to it the task of retrieval and augmentation, the latency is bound to increase. Therefore, it is also important to monitor how much time your RAG system takes from user input to response.

Ethical considerations are also at the forefront of generative AI adoption. For some RAG applications, it is important to measure the degree of **bias** and **toxicity** in the system responses. This is also influenced by the underlying data in the knowledge base. While it is not specific to RAG, it is important to evaluate the outputs for bias and toxicity.

Another aspect to check is the **robustness** of the system i.e. it's ability to handle different types of queries. Some queries may be simple, while others may involve complex reasoning. Some queries may require comparing two pieces of information, while others may involve complex post processing like mathematics calculations. We will look at some types of queries when we discuss CRAG, a benchmark, in section 5.4.

Finally, it is important to mention that these are scores and abilities that approach RAG at core technique level. RAG, after all, is a means to solving the end use-case. Therefore, you may have to build a **use-case specific** evaluation criteria for your RAG system. For example, a question-answering system may use an exact match (EM) or F1 score as a metric and a summarization service may use ROUGE scores. Modern search engines using RAG may look at user interaction metrics, accuracy of source attribution etc.

This is the main idea behind evaluating RAG pipelines. The quality scores and the abilities that we discussed above need to be measured and benchmarked. There are two critical enablers of RAG evaluations – Frameworks and Benchmarks.

Frameworks are tools designed to facilitate evaluation offering automation of the evaluation process and data generation. They are used to streamline the evaluation process by providing a structured environment for testing different aspects of a RAG systems. They are flexible and can be adapted to different datasets and metrics. We will discuss the popular evaluation frameworks in section 5.3.

Benchmarks are standardized datasets and their evaluation metrics used to measure the performance of RAG systems. Benchmarks provide a common ground for comparing different RAG approaches. Benchmarks ensure consistency across the evaluations by considering a fixed set of tasks and their evaluation criteria. For example, HotpotQA focusses on multi-hop reasoning and retrieval capabilities using metrics like Exact Match and F1 scores.

Benchmarks are used to establish a baseline for performance and identify strengths/weaknesses is specific tasks or domains. We will discuss a few benchmarks and their characteristics in section 5.4

Developers can use frameworks to integrate evaluation in their development process and use benchmarks to compare their development with established standards. The frameworks and benchmarks both calculate **metrics** that focus on retrieval and the RAG quality scores. We will begin our discussion with the metrics in the next section before moving on to the popular benchmarks and frameworks.

5.2 Evaluation Metrics

Metrics quantify the assessment of the RAG system performance. We will classify the evaluation metrics into two broad groups –

1. Retrieval metrics that are commonly used in information retrieval tasks
2. RAG specific metrics that have evolved as RAG has found more application

It is noteworthy that there are natural language generation specific metrics like BLEU, ROUGE, METEOR, etc. that focus on the fluency and measure relevance and semantic similarity. They play an important role in analyzing and benchmarking the performance of Large Language Models. In this book we will discuss metrics specific to retrieval and RAG. You can find more details about the generation metrics in the Appendix.

5.2.1 Retrieval Metrics

The retrieval component of RAG can be evaluated independently to determine how well the retrievers are satisfying the user query. The primary retrieval evaluation metrics include accuracy, precision, recall, F1-score, mean reciprocal rank (MRR), mean average precision (MAP), and normalized discounted cumulative gain (nDCG).

ACCURACY

Accuracy is typically defined as the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined. In the context of information retrieval, it could be interpreted as –

$$\frac{\text{Number of relevant documents retrieved} + \text{Number of irrelevant documents not retrieved}}{\text{Total number of documents in the knowledge base}}$$

Even though accuracy is a simple, intuitive metric, it is not the primary metric for retrieval. In a large knowledge base, majority of documents are usually irrelevant to any given query, which can lead to misleadingly high accuracy scores. It does not consider ranking of the retrieved results.

PRECISION

Precision focusses on the quality of the retrieved results. It measures the proportion of retrieved documents that are relevant to the user query. It answers the question, "Of all the documents that were retrieved, how many were actually relevant?"

$$\text{Precision} = \frac{Number\ of\ Relevant\ Documents\ Retrieved}{Total\ Number\ of\ Documents\ Retrieved}$$

A higher precision will mean that the retriever is performing well and retrieving mostly relevant documents.

PRECISION@K

Precision@k is a variation of precision that measures the proportion of relevant documents amongst the top 'k' retrieved results. It is particularly important because it focusses on the top results rather than all the retrieved documents. For RAG it is important because only the top results are most likely to be used for augmentation. For example, if you restrict your RAG system to use only the top 5 retrieved documents for context augmentation, Precision@5 will be the metric to calculate.

$$Precision@k = \frac{Number\ of\ Relevant\ Documents\ Retrieved\ in\ Top\ 'k'\ Documents}{Top\ 'k'\ Documents\ Retrieved}$$

$$where\ 'k'\ is\ a\ chosen\ cut-off\ point$$

A precision@5 of .8 will mean that out of the top 5 retrieved documents, 4 were relevant.

Precision@k is also useful to compare systems when the total number of results retrieved may be different in different systems. However, the limitation is that the choice of 'k' can be arbitrary, and this metric doesn't look beyond the chosen 'k'.

RECALL

Recall focusses on the coverage that the retriever provides. It measures the proportion of the relevant documents retrieved from all the relevant documents in the corpus. It answers the question, "Of all the relevant documents, how many were actually retrieved?"

$$Recall = \frac{Number\ of\ relevant\ documents\ retrieved}{Total\ number\ of\ relevant\ documents\ in\ the\ knowledge\ base}$$

Note that, unlike precision, calculation of recall requires a prior knowledge of the total number of relevant documents. This can become challenging in large scale systems which have many documents in the knowledge base.

Like precision, recall also doesn't consider the ranking of the retrieved documents. It can also be misleading as retrieving all documents in the knowledge base will result in a perfect recall value.

Figure 5.1 visualizes various precision and recall scenarios.

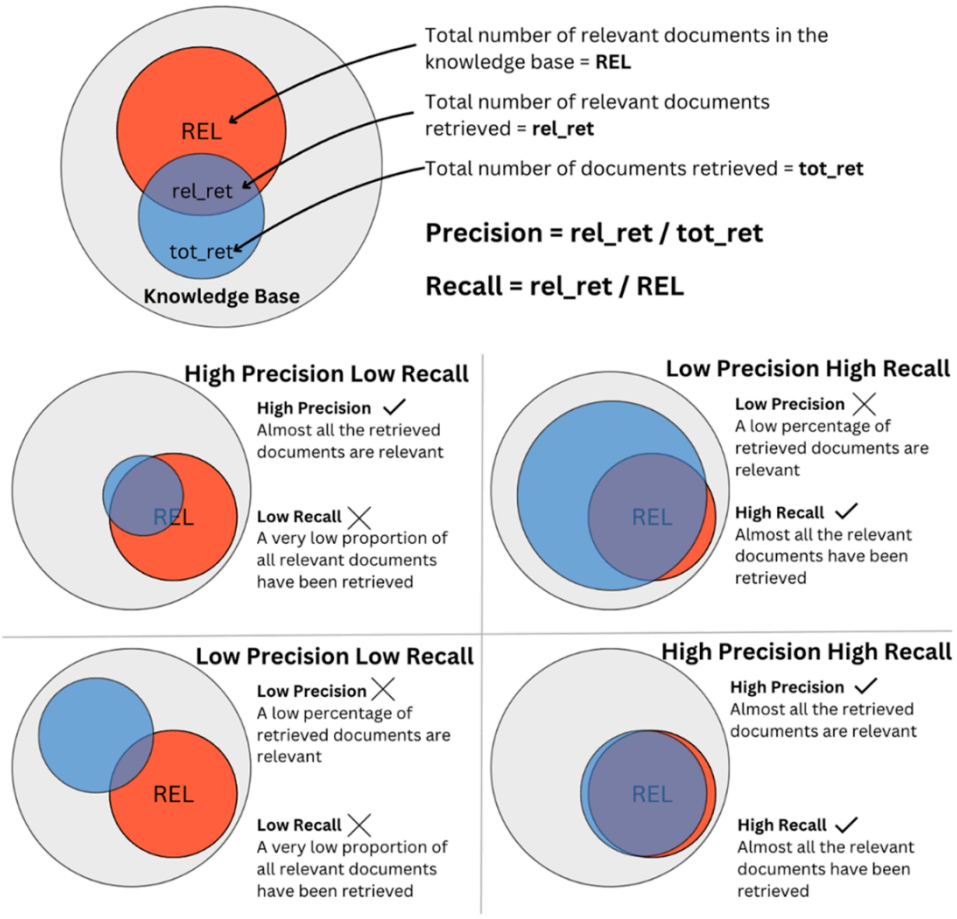

Figure 5.1 Precision and Recall

F1-SCORE

F1-score is the harmonic mean of precision and recall. It provides a single metric that balances both the quality and coverage of the retriever.

$$F1 - Score = 2\,x\,\frac{Precision\,x\,Recall}{Precision\,+\,Recall}$$

The equation is such that F1-score penalizes either variable having a low score; a High F1 score is only possible when both recall and precision values are high. This means that the score cannot be positively skewed by a single variable. Figure 5.2 illustrates how F1-score balances precision and recall.

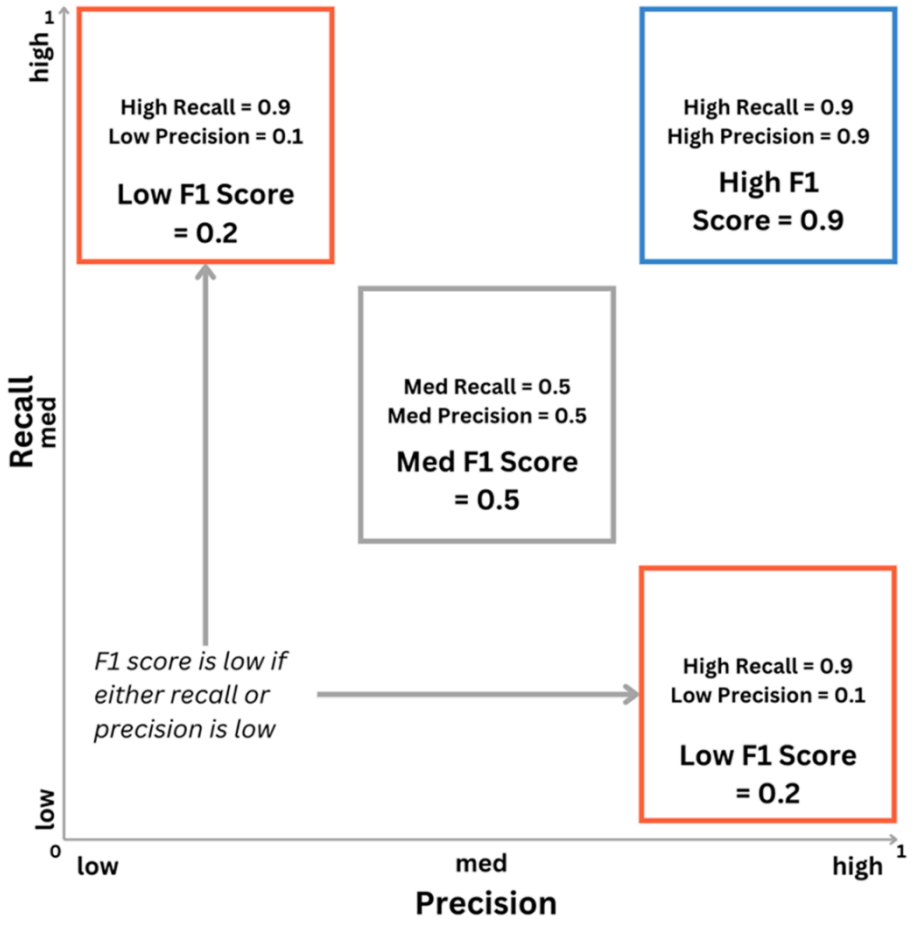

Figure 5.2 f1-score balances precision and recall. A medium value of both precision and recall gets a higher f1-score than if one value is very high and the other is very low.

F1-score provides and single, balanced measure that can be used to easily compare different systems. However, it does not take ranking into account and gives equal weightage to precision and recall which might not always be ideal.

MEAN RECIPROCAL RANK (MRR)

Mean Reciprocal Rank or MRR, is particularly useful in evaluating the rank of the relevant document. It measures the reciprocal of the ranks of the first relevant document in the list of results. MRR is calculated over a set of queries.

$$MRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$$

where N is the total number of queries and $rank^i$ is the rank of the first relevant document of the i-th query

MRR is particularly useful when you're interested in how quickly the system can find a relevant document and considers the ranking of the results. However, since it doesn't look at anything beyond the first relevant result, it may not be useful when multiple relevant results are important. Figure 5.3 shows how mean reciprocal rank is calculated.

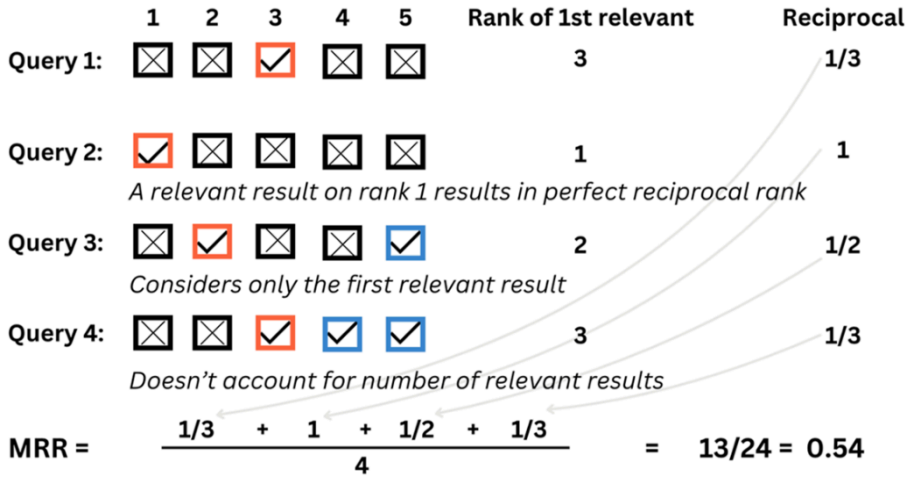

Figure 5.3 MRR considers the ranking but doesn't consider all the documents

MEAN AVERAGE PRECISION (MAP)

Mean Average Precision or MAP is a metric that combines precision and recall at different cut-off levels of 'k' i.e. the cut-off number for the top results. It calculates a measure called Average Precision and then averages it across all queries.

$$Averageprecisionforasinglequery(i) = \frac{1}{R_i}\sum_{k=1}^{n} Precision@k \; x \; rel@k$$

R^i is the number of relevant documents for query i

Precision@k is the precision at cut-off 'k'

rel@k is a binary flag indicating the relevance of the document at rank k

Mean Average Precision is the mean of the average precision (shown above) over all the 'N' queries

$$MAP = \frac{1}{N}\sum_{i=1}^{N} Average\ precision(i)$$

MAP provides a single measure of quality across recall levels. It is quite suitable when result ranking is important, but it is complex to calculate. Let us look at an example MAP calculation in figure 5.4 below

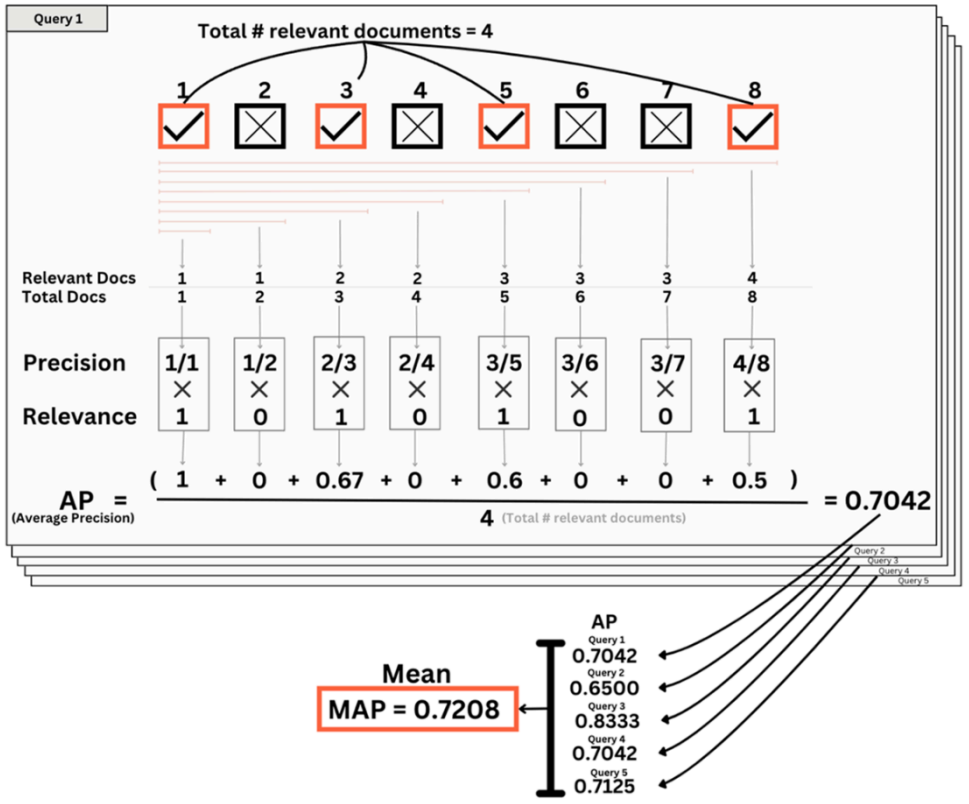

Figure 5.4 MAP considers all the retrieved documents and gives a higher score for better ranking

NORMALIZED DISCOUNTED CUMULATIVE GAIN (NDCG)

nDCG evaluates the ranking quality by considering the position of relevant documents in the result list and assigning higher scores to relevant documents appearing earlier. It is particularly effective for scenarios where documents have varying degrees of relevance. To calculate discounted cumulative gain (DCG), each document in the retrieved list is assigned a relevance score, rel and a discount factor reduces the weight of documents as their rank position increases.

$$DCG = \sum_{i=1}^{n} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

Here rel^i is the graded relevance of document at position i. IDCG is the ideal DCG which is the DCG for perfect ranking.

nDCG is calculated as the ratio between actual discounted cumulative gain (DCG) and the ideal discounted cumulative gain (IDCG)

$$nDCG = \frac{DCG}{IDCG}$$

Figure 5.5 shows an example of nDCG calculation.

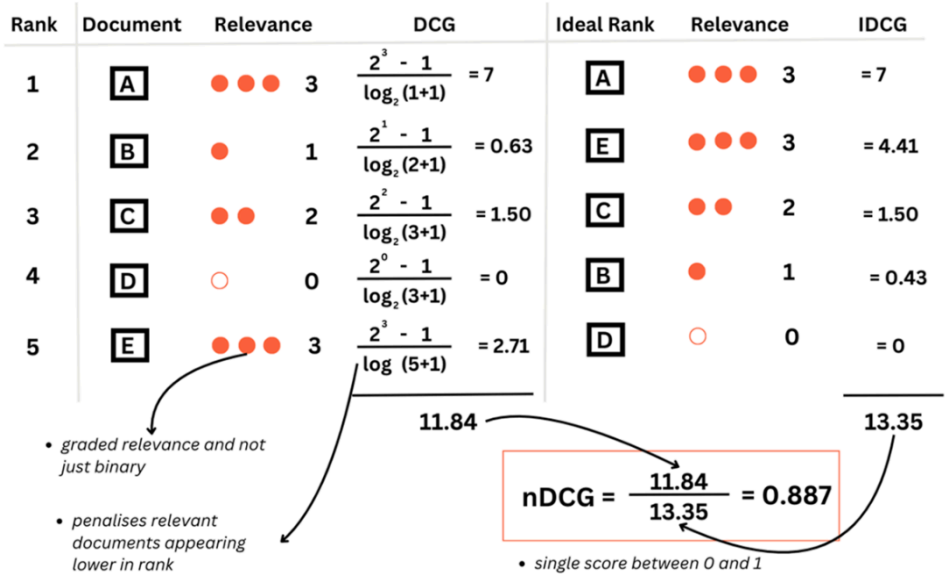

Figure 5.5 nDCG addresses degrees of relevance in documents and penalizes incorrect ranking

NDCG is quite a complex metric to calculate. It requires documents to have a relevance score which may lead to subjectivity and the choice of the discount factor affects the values significantly, but it accounts for varying degrees of relevance in documents and gives more weightage to higher ranked items.

Retrieval systems are not just used in RAG but in a variety of other application areas like web and enterprise search engines, e-commerce product search and personalized recommendations, social media ad retrieval, archival systems, databases, virtual assistants and more. The retrieval metrics help in assessing and improving the performance to effectively meet user needs. Table 5.1 will summarize the above discussion on different retrieval metrics.

Table 5.1 Retrieval metrics

Metric	What it Measures	Strengths	Use Cases	Considerations
Accuracy	Overall correctness of retrieval	Simple to understand, includes true negatives	General performance in balanced datasets	Can be misleading in imbalanced datasets, doesn't consider ranking
Precision	Quality of retrieved results	Easy to understand and calculate	General retrieval quality assessment	Doesn't consider ranking or completeness of retrieval
Precision@k	Quality of top k retrieved results	Focuses on most relevant results for RAG	When only top k results are used for augmentation	Choose k based on your RAG system's usage
Recall	Coverage of relevant documents	Measures completeness of retrieval	Assessing if important information is missed	Requires knowing all relevant documents in corpus
F1-Score	Balance between precision and recall	Single metric combining quality and coverage	Overall retrieval performance	May obscure trade-offs between precision and recall
Mean Reciprocal Rank (MRR)	How quickly a relevant document is found	Emphasizes finding at least one relevant result quickly	When finding one good result is sufficient	Less useful when multiple relevant results are needed
Mean Average Precision (MAP)	Precision at different recall levels	Considers both precision and ranking	Comprehensive evaluation of ranked retrieval results	More complex to calculate and interpret
Normalized Discounted Cumulative Gain (nDCG)	Ranking quality with graded relevance	Accounts for varying degrees of relevance and ranking	When documents have different levels of relevance	Requires relevance scoring for documents

Not all retrieval metrics are popular for evaluations. Often, the more complex metrics are overlooked for the sake of explainability. The usage of the above metrics depends on the stage of improvement in the evolution of system performance you are. For example, to start with you may just be trying to improve precision, while at an evolved stage you may be looking more for better ranking.

While the above metrics focus on retrieval in general, there are metrics that have been created specifically for RAG applications. These metrics focus on the three quality scores that we discussed in section 5.1.

5.2.2 RAG specific metrics

The three quality scores that are used to evaluate RAG applications are context relevance, answer relevance and answer faithfulness. These scores specifically answer three questions –

- Is the information retrieval relevant to the user query?
- Is the generated answer rooted in the retrieved information?
- Is the generated answer relevant to the user query?

Let us look at each of these scores –

CONTEXT RELEVANCE

Context relevance evaluates how well the retrieved documents relate to the original query. The key aspects are topical alignment, information usefulness and redundancy. There are human evaluation methods as well as semantic similarity measures to calculate context relevance.

One such measure is employed by the Retrieval Augmented Generation Assessment (RAGAs) framework (discussed further in Section 5.3). The retrieved context should contain information only relevant to the query or the prompt. For context relevance, a metric 'S' is estimated. 'S' is the number of sentences in the retrieved context that are relevant for responding to the query or the prompt.

$$Context\ Relevance = \frac{Number\ of\ relevant\ sentences\ in\ the\ retrieved\ context\ (S)}{Total\ number\ of\ sentences\ in\ the\ retrieved\ context}$$

Figure 5.6 is an illustrative example of high and low context relevance.

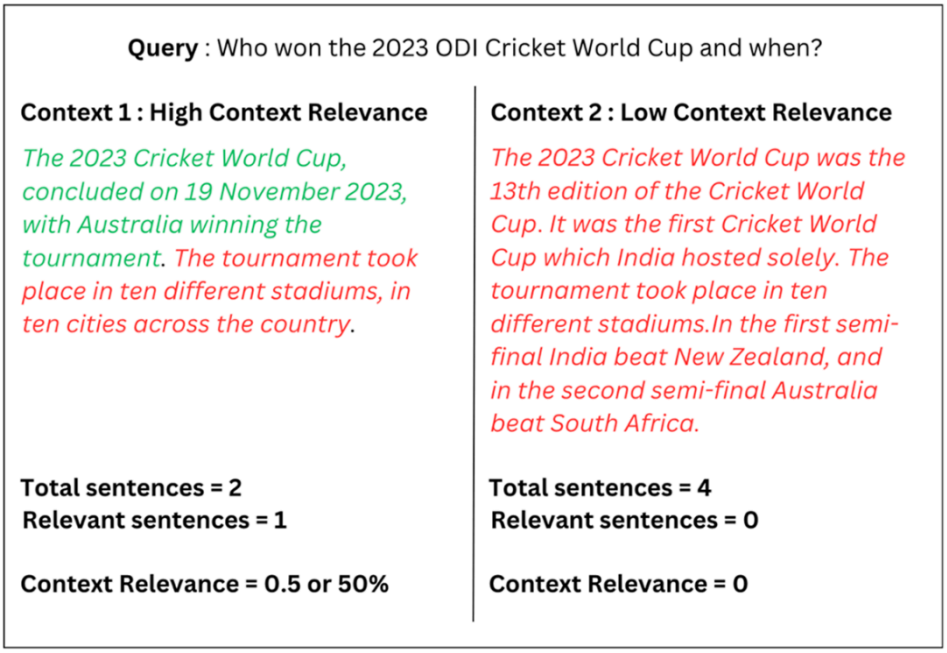

Figure 5.6 Context relevance evaluates the degree to which the retrieved information is relevant to the query

The number of relevant sentences is also sometimes customized to the sum of similarity scores of each of the sentences with the query. Context relevance ensures that the generation component has access to appropriate information.

ANSWER FAITHFULNESS

Faithfulness is the measure of the extent to which the response is factually grounded in the retrieved context. Faithfulness ensures that the facts in the response do not contradict the context and can be traced back to the source. It also ensures that the LLM is not hallucinating. In RAGAs framework, faithfulness first identifies the number of "claims" made in the response and calculates the proportion of those "claims" present in the context.

$$\text{Answer Faithfulness} = \frac{\text{Number of generated claims present in the context}}{\text{Total number of claims made in the generated response}}$$

Let us look at an example in figure 5.7

Figure 5.7 Answer Faithfulness evaluates the closeness of the generated response to the retrieved context

Faithfulness is not a complete measure of factual accuracy but only evaluates the groundedness to the context.

An inverse metric for faithfulness is also **Hallucination Rate** which can calculate the proportion of generated claims in the response that are not present in the retrieved context.

Another related metrics to faithfulness is **Coverage**. Coverage measures the number of relevant claims in the context and calculates the proportion of relevant claims present in the generated response. This measures how much of the relevant information from the retrieved passages is included in the generated answer.

$$Coverage = \frac{Number\ of\ relevant\ claims\ from\ the\ context\ in\ the\ generated\ response}{Total\ number\ of\ relevant\ claims\ in\ the\ context}$$

ANSWER RELEVANCE

Like context relevance measures the relevance of the retrieved context to the query, answer relevance is the measure of the extent to which the response is relevant to the query. This metric focusses on key aspects such as system's ability to comprehend the query, response being pertinent to the query and the completeness of the response.

In RAGAs, for this metric, a response is generated for the initial query or prompt. To compute the score, the LLM is then prompted to generate questions for the generated response several times. The mean cosine similarity between these questions and the original one is then calculated. The concept is that if the answer correctly addresses the initial question, the LLM should generate questions from it that match the original question.

$$\text{Answer Relevance} = \frac{1}{\mathrm{N}}\sum_{i=1}^{N} Similarity(UserQuery, LLMGeneratedQuery[i])$$

where N is the number of queries generated by the LLM.

It is important to note that answer relevance is not a measure of truthfulness but only of relevance. The response may or may not be factually accurate but may be relevant. In figure 5.8 we see an illustration of the answer relevance calculation. Can you find the reason why the relevance is not very high? (Hint: The answer may have some irrelevant facts). Answer relevance ensures that the RAG system provides useful and appropriate responses, enhancing user satisfaction and the system's practical utility.

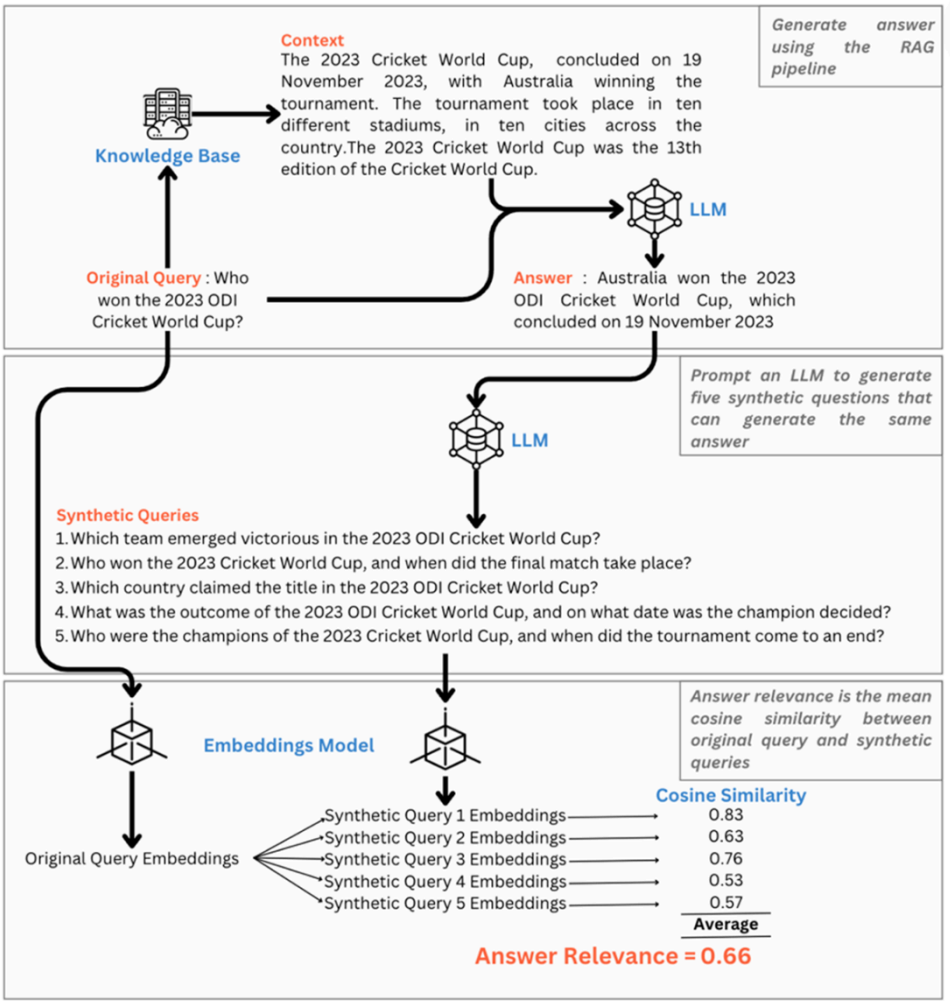

Figure 5.8 Answer relevance is calculated as mean of cosine similarity between original and synthetic questions.

TRADE-OFFS AND OTHER CONSIDERATIONS

These three metrics and their derivatives form the core of RAG quality evaluation. These three metrics are interconnected and sometimes involve trade-offs. High context relevance usually leads to better faithfulness, as the system has access to more pertinent information. However, high faithfulness doesn't always guarantee high answer relevance. A system might faithfully reproduce information from the retrieved passages but fail to directly address the query. Optimizing for answer relevance without considering faithfulness might lead to responses that seem appropriate but contain hallucinated or incorrect information.

HUMAN EVALUATIONS & GROUND TRUTH DATA

Most of the metrics we discussed talk about a concept of relevant documents. For example, precision is calculated as the number of relevant documents retrieved divided by the total number of retrieved documents. The question that arises is - How does one establish that a document is relevant?

The simple answer is a human evaluation approach. A subject matter expert looks at the documents and determines the relevance. Human evaluation brings in subjectivity and, therefore, human evaluations are done by a panel of experts rather than an individual. But human evaluations are restrictive from a scale and a cost perspective.

Any data that can reliably establish relevance, consequently, becomes extremely useful. Ground truth is information that is known to be real or true. In RAG, and Generative AI domain in general, Ground Truth is a prepared set of Prompt-Context-Response or Question-Context-Response example, akin to labelled data in Supervised Machine Learning parlance. Ground truth data that is created for your knowledge base can be used for evaluation of your RAG system.

How does one go about creating the ground truth data? It can be viewed at as a one-time exercise where a group of experts creates this data. However, generating hundreds of QCA (Question-Context-Answer) samples from documents manually can be a time-consuming and labor-intensive task. Additionally, if the knowledge base is dynamic, the ground truth data will also need updates. Questions created by humans may face challenges in achieving the necessary level of complexity for a comprehensive evaluation, potentially affecting the overall quality of the assessment.

Large Language Models can be used to address these challenges. Synthetic Data Generation uses LLMs to generate diverse questions and answers from the documents in the knowledge base. LLMs can be prompted to create questions like simple questions, multi-context questions, conditional questions, reasoning questions etc. using the documents from the knowledge base as context.

We have discussed quite a few metrics in this section. Effective interpretation of these metrics is crucial for performance improvement. As creators of RAG systems, you should use these metrics to compare with similar systems. You can also look at consistent trends to identify strengths and weaknesses of your system. A low precision-high recall system may indicate that your system is retrieving a lot of documents, and you may need to make your retriever more selective. A low precision-low recall system points out fundamental issues with retrieval and you may need to reassess the indexing pipeline itself. Same issue may be indicated by a low MAP or a low context relevance score. Similarly, a low MRR or a low nDCG value may indicate a problem with ranking algorithm of the retriever. To address, low answer faithfulness or low answer relevance you may need to improve your prompts or fine-tune the LLM.

There may also exist some trade-offs that you will need to balance. Improving precision often reduces recall and vice-versa. Highly relevant but brief contexts may lead to incomplete answers and high answer faithfulness may sometimes come at the cost of answer relevance.

The relative importance of each metric will depend on your use case and user requirements. You may need to include other metrics that are specific to your downstream use case like summarization may want to measure conciseness and chatbots may want to emphasize on conversation coherence.

Developers can code these metrics from scratch and integrate them in the development and deployment process of their RAG system. However, you'll find evaluation frameworks, that are readily available, quite handy. We will discuss three popular frameworks in section 5.3.

5.3 Frameworks

Frameworks provide a structured approach to RAG evaluations. They can be used to automate the evaluation process. Some go beyond and assist in the synthetic ground truth data generation. While new evaluation frameworks will continue to be introduced, there are two popular ones that we will discuss in this section.

- RAGAs (Retrieval Augmented Generation Assessment)
- ARES (Automated RAG Evaluation System)

5.3.1 RAGAs

Retrieval Augmented Generation Assessment or RAGAs is a framework developed by Exploding Gradients that assesses the retrieval and generation components of RAG systems without relying on extensive human annotations. RAGAs helps in -

- Synthetically generate a test dataset that can be used to evaluate a RAG pipeline
- Use metrics to measure the performance of the pipeline
- Monitor the quality of the application in production

We will continue with our example of the Wikipedia page of the 2023 Cricket World Cup. We will first create a synthetic test dataset using ragas and then use the RAGAs metrics to evaluate the performance of the RAG pipeline we have created in Chapter 3 and Chapter 4.

SYNTHETIC TEST DATASET GENERATION (GROUND TRUTHS)

We discussed in section 5.2 that Ground Truths data is necessary to calculate evaluation metrics for assessing the quality of RAG pipelines. While this data can be manually curated, RAGAs provides the functionality of generating this dataset from the documents in the knowledge base.

RAGAs does this using an LLM. It analyses the documents in the knowledge base and uses an LLM to generate seed questions from chunks in the knowledge base. These questions are based on the document chunks from the knowledge base. These chunks act as the context for the questions. Another LLM is used to generate answer to these questions. This is how it generates a Question-Context-Answer data based on the documents in the knowledge base. RAGAs also has an evolver module that creates more difficult questions like multi-context, reasoning, conditional, etc. for a more comprehensive evaluation. Figure 5.9 illustrates the process of synthetic data generation with RAGAs.

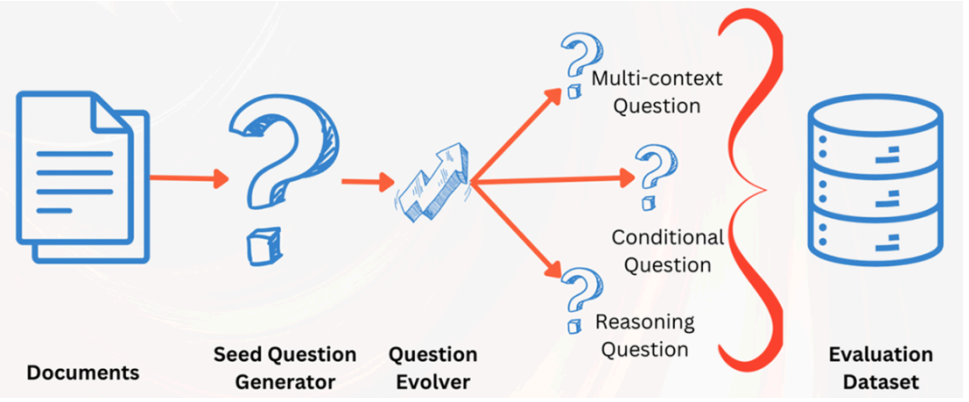

Figure 5.9 Synthetic ground truths data generation using RAGAs

To evaluate our RAG pipeline let us recreate the documents from the Wikipedia page like we did in chapter 3. Note that we will have to install the packages that we have in the previous chapters to continue with the code below.

```
#Importing the AsyncHtmlLoader
from langchain_community.document_loaders import AsyncHtmlLoader

#This is the url of the wikipedia page on the 2023 Cricket World Cup
url="https://en.wikipedia.org/wiki/2023_Cricket_World_Cup"

#Instantiating the AsyncHtmlLoader
loader = AsyncHtmlLoader (url)

#Loading the extracted information
data = loader.load()

from langchain_community.document_transformers import Html2TextTransformer

#Instantiate the Html2TextTransformer function
html2text = Html2TextTransformer()


#Call transform_documents
data_transformed = html2text.transform_documents(data)
```

The data_transformed contains the necessary document format of the Wikipedia page. We will use RAGAs library to generate the dataset from these documents. For that we will first need to install the ragas library.

```
%pip install ragas==0.1.11

# Import necessary libraries
from ragas.testset.generator import TestsetGenerator
from ragas.testset.evolutions import simple, reasoning, multi_context
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

# Instantiate the models
generator_llm = ChatOpenAI(model="gpt-4o-mini")
critic_llm = ChatOpenAI(model="gpt-4o-mini")
embeddings = OpenAIEmbeddings()

# Create the TestsetGenerator
generator = TestsetGenerator.from_langchain(
    generator_llm,
    critic_llm,
    embeddings
)

# Call the generator
testset = generator.generate_with_langchain_docs(
          data_transformed,
          test_size=20,
          distributions={
                    simple: 0.5,
          reasoning: 0.25,
          multi_context: 0.25}
          )
```

The testset that we have created above contains 20 questions based on our document along with the chunk of the document that the question was based on and the ground truth answer. 10 questions are simple, and 5 questions each are reasoning and multi-context. A screenshot of the dataset is shown in figure 5.10

	question	contexts	ground_truth	evolution_type	metadata	episode_done
0	What was the outcome of the match between Paki...	[3 overs) \n---|---|--- \n|	\n**New Zea...	India won by 7 wickets against Pakistan on 14 ...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
1	What was the outcome of the warm-up match held...	[to 3 October 2023 at Rajiv Gandhi\nInternati...	The warm-up match held at Greenfield Internati...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
2	What was the outcome of the match between Sout...	[aram Stadium, Chennai \n--- \n \n19 Octobe...	South Africa won by 229 runs against England o...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
3	What was the outcome of the warm-up match held...	[to 3 October 2023 at Rajiv Gandhi\nInternati...	The warm-up match held at Greenfield Internati...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
4	What broadcasting rights were associated with ...	[Opening batsman / wicket-keeper \nRohit Sh...	Disney Star served as the host broadcaster of ...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
5	What was the outcome of the match between Afgh...	[3 overs) \n---|---|--- \n|	\n**New Zea...	New Zealand won by 149 runs against Afghanista...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
6	What was the outcome of the match between Aust...	[v**	**New Zealand** \n383/9 (50 overs) \n...	Australia won by 33 runs against England on No...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
7	What were the results of the matches played in...	[aram Stadium, Chennai \n--- \n \n19 Octobe...	From October 19 to October 28, 2023, the resul...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
8	What notable matches took place at the Assam C...	[Hasan 74 (89) \nReece Topley 3/23 (5 overs)...	Notable matches that took place at the Assam C...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
9	What broadcasting rights were associated with ...	[Opening batsman / wicket-keeper \nRohit Sh...	Disney Star served as the host broadcaster of ...	simple	[{'source': 'https://en.wikipedia.org/wiki/202...	True
10	What's the winning margin for Pak vs NZ on Nov...	[v**	**New Zealand** \n383/9 (50 overs) \n...	Pakistan won by 21 runs (DLS method).	reasoning	[{'source': 'https://en.wikipedia.org/wiki/202...	True
11	What's the winning margin for Aus vs Eng on No...	[v**	**New Zealand** \n383/9 (50 overs) \n...	Australia won by 33 runs against England on No...	reasoning	[{'source': 'https://en.wikipedia.org/wiki/202...	True
12	Which broadcaster teamed up with ICC for the 2...	[Opening batsman / wicket-keeper \nRohit Sh...	Disney Star served as the host broadcaster of ...	reasoning	[{'source': 'https://en.wikipedia.org/wiki/202...	True

Figure 5.10 Synthetic test data generated using RAGAs

We will use this dataset to evaluate our RAG pipeline.

RECREATING RAG PIPELINE

From the test dataset created above, we will use the question and the ground_truth information. We will pass the questions to our RAG pipeline and generate answers. We will compare these answers with the ground_truth to calculate the evaluation metrics. First, we will recreate our RAG pipeline. Again, it is important to note that we will have to install the packages that we have in the previous chapters to continue with the code below.

Retrieval Function

```
# Import FAISS class from vectorstore library
from langchain_community.vectorstores import FAISS

# Import OpenAIEmbeddings from the library
from langchain_openai import OpenAIEmbeddings

def retrieve_context(query, db_path):
    embeddings=OpenAIEmbeddings(model="text-embedding-3-large")

    # Load the database stored in the local directory
    db=FAISS.load_local(db_path, embeddings, allow_dangerous_deserialization=True)

    # Ranking the chunks in descending order of similarity
    docs = db.similarity_search(query)
    # Selecting first chunk as the retrieved information
    retrieved_context=docs[0].page_content

    return str(retrieved_context)
```

Augmentation Function

```
def create_augmeted(query, db_path):

    retrieved_context=retrieve_context(query,db_path)

    # Creating the prompt
    augmented_prompt=f"""

    Given the context below answer the question.

    Question: {query}

    Context : {retrieved_context}

    Remember to answer only based on the context provided and not from any other source.

    If the question cannot be answered based on the provided context, say I don’t know.

    """

    return retrieved_context, str(augmented_prompt)
```

RAG function

```
# Importing the OpenAI library
from openai import OpenAI

def create_rag(query, db_path):

    augmented_prompt=create_augmeted(query,db_path)



    # Instantiate the OpenAI client
    client = OpenAI()

    # Make the API call passing the augmented prompt to the LLM
    response = client.chat.completions.create(
    model="gpt-4o",
    messages=    [
        {"role": "user", "content": augmented_prompt}
          ]
    )

    # Extract the answer from the response object
    answer=response.choices[0].message.content

    return retrieved_context, answer
```

We can try this pipeline to generate answers

```
# Location of the stored vector index created by the indexing pipeline
db_path='../../Assets/Data'

# User Question
query="Who won the 2023 cricket world cup?"

# Calling the RAG function
create_rag(query, db_path)
```

Now that we have this RAG pipeline function, we can evaluate this pipeline on the questions that have been synthetically generated.

EVALUATIONS

We will first generate answers to the questions in the synthetic test data using our RAG pipeline. We will then compare the answers to the ground truth answers. We will first generate the answers.

```
# Create Lists for Questions and Ground Truths from testset
questions_list=testset.to_pandas().question.to_list()
gt_list=testset.to_pandas().ground_truth.to_list()

answer_list=[]
context_list=[]

# Iterate through the testset to generate response for questions
for record in testset.test_data:

     # Call the RAG function
     rag_context, rag_answer=create_rag(record.question,db_path)
     ground_truth=record.ground_truth
     answer_list.append(rag_answer)
     context_list.append([rag_context])

# Create dictionary of question, answer, context and ground truth
data_samples={
    'question':questions_list,
    'answer':answer_list,
    'contexts': context_list,
    'ground_truth':gt_list
}
```

For RAGAs, the evaluation set needs to be in the Dataset format. Datasets is a lightweight library from HuggingFace.

```
# Install the datasets package
%pip install datasets==2.20.0

# Import the Datasets library
from datasets import Dataset

# Create Dataset from the dictionary
dataset = Dataset.from_dict(data_samples)
```

Now that we have the complete evaluation dataset, we can invoke the metrics.

```
#Import all the libraries
from ragas import evaluate

from ragas.metrics import (
```

```
    answer_relevancy,
    faithfulness,
    context_recall,
    context_precision,
    context_entity_recall,
    answer_similarity,
    answer_correctness
)

from ragas.metrics.critique import (
    harmfulness,
    maliciousness,
    coherence,
    correctness,
    conciseness
)

# Calculate the metrics for the dataset

result = evaluate(
    dataset,
    metrics=[
        context_precision,
        faithfulness,
        answer_relevancy,
        context_recall,
        context_entity_recall,
        answer_similarity,
        answer_correctness,
        harmfulness,
        maliciousness,
        coherence,
        correctness,
        conciseness

    ],
)
```

The result will look something like below

```
{
    "context_precision": 0.749999999925,
    "faithfulness": 0.39583333333333337,
    "answer_relevancy": 0.5376135644777853,
    "context_recall": 0.6,
    "context_entity_recall": 0.4677380943262032,
    "answer_similarity": 0.8603128301847682,
    "answer_correctness": 0.5283911977374367,
    "harmfulness": 0.0,
    "maliciousness": 0.1,
    "coherence": 0.5,
    "correctness": 0.55,
    "conciseness": 0.55
}
```


You can also visit the official documentation of RAGAs for more information (https://docs.ragas.io/en/stable/). RAGAs calculates a bunch of metrics that are useful for assessing the quality of the RAG pipeline. RAGAs uses an LLM to do this, somewhat subjective, task. For example, to calculate faithfulness for a given question-context-answer record, RAGAs first breaks down the answer into simple statements. Then, for each statement, it asks the LLM whether the statement can be inferred from the context. The LLM provides a 0 or 1 response along with a reason. This process is repeated a couple of times. Finally, faithfulness is calculated as the proportion of statements judged by the LLM as faithful (i.e. 1). Several other metrics are calculated using this LLM based approach. This approach where an LLM is used in evaluating a task is also popularly called **LLM as a judge** approach. An important point to note here is that the accuracy of this evaluation is also dependent on the quality of the LLM that is being used as the judge.

5.3.2 ARES

Automated RAG evaluation system, or ARES, is a framework developed by researchers at Stanford University and Databricks. Like RAGAs, ARES uses an LLM as a judge approach for evaluations. Both request a language model to classify answer relevance, context relevance and faithfulness for a given query. However, there are some differences.

- RAGAs relies on heuristically written prompts that are sent to the LLM for evaluation. ARES, on the other hand, trains a classifier using a language model.
- RAGAs aggregates the responses from the LLM to arrive at a score. ARES provides confidence intervals for the scores leveraging a framework called Prediction-Powered Inference (PPI)
- RAGAs generates a simple synthetic question-context-answer dataset for evaluation from the documents. ARES generate synthetic datasets comprising both positive and negative examples of query-passage-answer triples.

ARES requires more data than we saw above with RAGAs. To leverage ARES, you need three datasets –

- **In-Domain Passage Set**: This is a collection of passages relevant to the specific domain being evaluated. The passages should be suitable for generating queries and answers. In our case, it will be the documents that we created from the Wikipedia article.
- **Human Preference Validation Set**: A minimum of approximately 150 annotated data points is required. This set is used to validate the preferences of human annotators regarding the relevance of the generated query-passage-answer triples.
- **Few-Shot Examples**: At least five examples of in-domain queries and answers are needed. These examples help prompt the large language models (LLMs) during the synthetic data generation process.

The need for a human-preference validation set and fine-tuning of language models for classification makes applying ARES more complex than RAGAs. We will keep the application of ARES out of scope of this book. However, ARES is a robust framework. It provides a detailed analysis of system performance with statistical confidence intervals, making it suitable for in-depth RAG system evaluations. RAGAs promises a faster evaluation cycle without reliance on human annotations. More details on ARES application can be found in the official github repository (https://github.com/stanford-futuredata/ARES)

While RAGAs and ARES have gained popularity, there are other frameworks, like TruLens, DeepEval, RAGChecker, etc., that have also gotten acceptance amongst RAG developers. There is more information on additional frameworks in the Appendix.

Frameworks provide a standardized method of automating evaluation of your RAG pipelines. Your choice of the evaluation framework should depend on the requirements of your use-case. For quick and easy evaluations that are widely understood, RAGAs may be your choice. For robustness across diverse domains and question types, ARES might suit better. Most of the proprietary service providers (Vector DBs, LLMs, etc.) have their own evaluation features that you may use. You can also develop your own metrics.

Next, we look at benchmarks. Benchmarks are used to compare competing RAG systems with one another.

5.4 Benchmarks

Benchmarks provide a standard point of reference to evaluate the quality and performance of a system. RAG benchmarks are a set of standardized tasks, and a dataset used to compare the efficiency of different RAG system in retrieving relevant information and generating accurate responses. There has been a surge in creating benchmarks since 2023 when RAG started gaining popularity but there have been benchmarks on question answering tasks that were introduced before that. Benchmarks like Stanford Question Answering Dataset (SQuAD), WikiQA, Natural Question (NQ), HotpotQA, are open domain question answering datasets that evaluate, primarily, the retriever component using metrics like Exact Match (EM) and F1-score. BEIR or Benchmarking Information Retrieval (figure 5.11) is a comprehensive, heterogeneous benchmark that is based on 9 IR tasks and 19 Question-Answer datasets. More details about these benchmarks are in the Appendix. We will discuss three of the popular RAG specific benchmarks and their evaluation in this section.

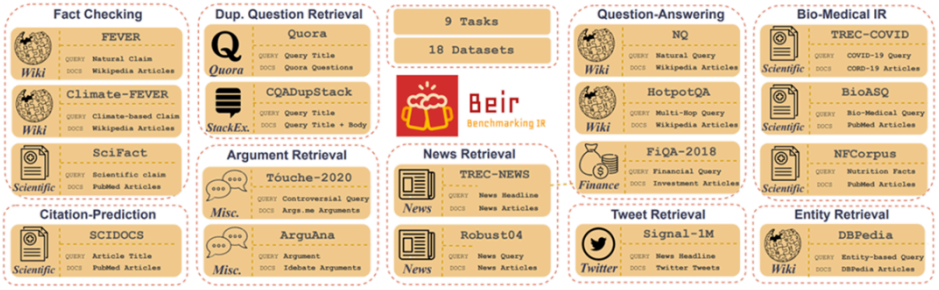

Figure 5.11 BEIR – 9 tasks and 18 (of 19) datasets (Source: BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models https://arxiv.org/pdf/2104.08663v4)

RGB

Retrieval Augmented Generation Benchmark (RGB) was introduced in a Dec 2023 paper (https://arxiv.org/pdf/2309.01431). It comprises 600 base questions and 400 additional questions, evenly split between English and Chinese. The corpus was constructed using a multi-step process that involved collecting recent news articles, generating questions and answers using ChatGPT, retrieving relevant web pages through Google's API, and selecting the most pertinent text chunks using a dense retrieval model. It is a benchmark that focusses on four key abilities of a RAG system – Noise Robustness, Negative Rejection, Information Integration and Counterfactual Robustness as illustrated in figure 5.12

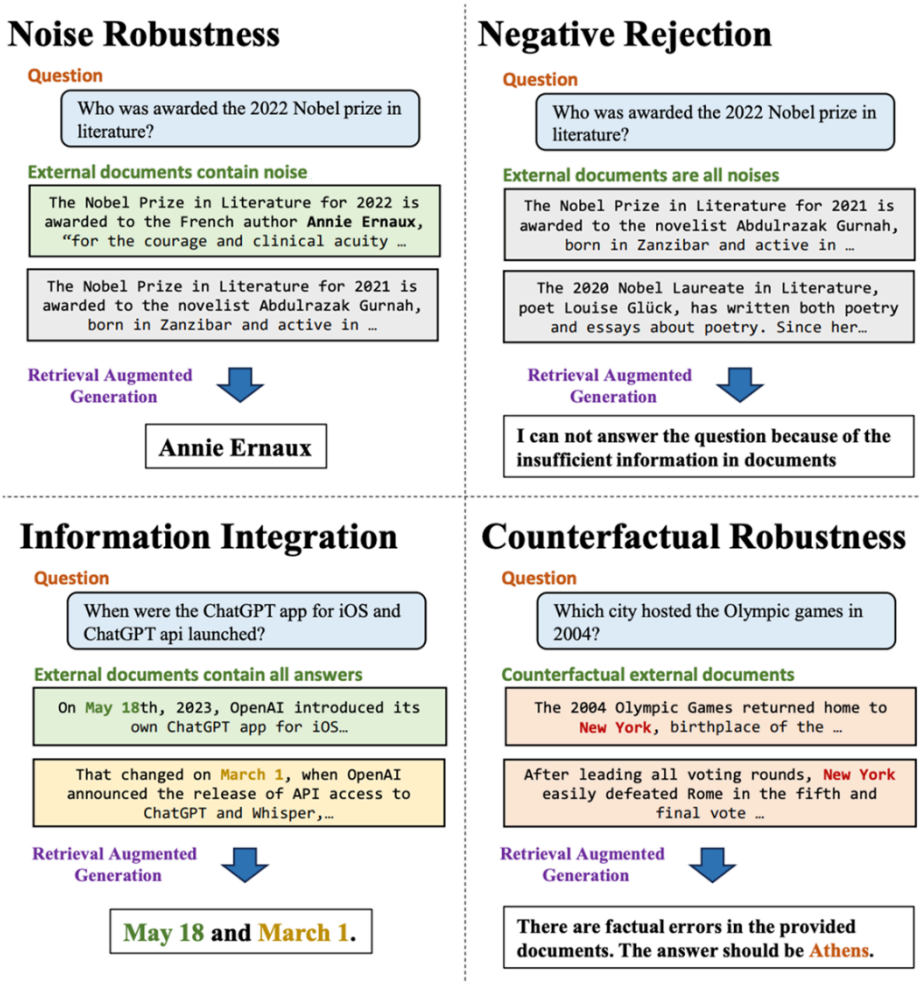

Figure 5.12 Four abilities required of RAG systems (Source: Benchmarking Large Language Models in Retrieval-Augmented Generation, Chen et al - https://arxiv.org/pdf/2309.0143)

It focusses on the following metrics for evaluation

- Accuracy:
  - Used for noise robustness and information integration
  - Based on exact matching of the generated text with the correct answer

- Rejection Rate:
  - Used for negative rejection
  - Measured by exact matching of the model's output with a specific rejection phrase

    - Also evaluated using ChatGPT to determine if responses contain rejection information

- Error Detection Rate:
    - Used for counterfactual robustness
    - Measured by exact matching of the model's output with a specific error detection phrase
    - Also evaluated using ChatGPT

- Error Correction Rate:
    - Used for counterfactual robustness
    - Measures whether the model can provide the correct answer after identifying errors

You can use the GitHub repository to implement RGB - https://github.com/chen700564/RGB

MULTIHOP RAG

Curated by researchers at HKUST, Multihop RAG contains 2556 queries, with evidence for each query distributed across 2 to 4 documents. The queries also involve document metadata, reflecting complex scenarios commonly found in real-world RAG applications. It contains four types of queries.

- Inference: Synthesizing information across multiple sources
    - g. Which report discusses the supply chain risk of Apple, the 2019 annual report or the 2020 annual report?

- Comparison: Comparing facts from different sources
    - g. Did Netflix or Google report higher revenue for the year 2023?

- Temporal: Analyzing temporal ordering of events
    - g. Did Apple introduce the AirTag tracking device before or after the launch of the 5th generation iPad Pro?

- Null: Queries not answerable from the knowledge base

Full implementation code can be found at https://github.com/yixuantt/MultiHop-RAG

CRAG

Comprehensive RAG benchmark, curated by Meta and HKUST is a factual question answering benchmark of 4,409 question-answer pairs and mock APIs to simulate web and Knowledge Graph (KG) search. It contains 8 types (Simple, Conditions, Comparison questions, Aggregation questions, multi-hop questions, Set queries, Post-processing-heavy questions, and False-premise questions as illustrated in figure 5.13) of queries across 5 domains (Finance, Sports, Music, Movie, and Open domain).

Question type	Definition
Simple	Questions asking for simple facts that are unlikely to change overtime, such as the birth date of a person and the authors of a book.
Simple w. Condition	Questions asking for simple facts with some given conditions, such as stock prices on a certain date and a director's recent movies in a certain genre.
Set	Questions that expect a set of entities or objects as the answer (e.g., "*what are the continents in the southern hemisphere?*").
Comparison	Questions that compare two entities (e.g., "*who started performing earlier, Adele or Ed Sheeran?*").
Aggregation	Questions that require aggregation of retrieval results to answer (e.g., "*how many Oscar awards did Meryl Streep win?*").
Multi-hop	Questions that require chaining multiple pieces of information to compose the answer (e.g., "*who acted in Ang Lee's latest movie?*").
Post-processing heavy	Questions that need reasoning or processing of the retrieved information to obtain the answer (e.g., "*how many days did Thurgood Marshall serve as a Supreme Court justice?*").
False Premise	Questions that have a false preposition or assumption (e.g., "*What's the name of Taylor Swift's rap album before she transitioned to pop?*" (Taylor Swift has not yet released any rap album)).

Figure 5.13 8 question types in CRAG.

For each question in the evaluation set, CRAG labels the answer with one of four classes –

1. **Perfect**: The response correctly answers the user's question and contains no hallucinated content [Scored as +1]
2. **Acceptable**: The response provides a useful answer to the user's question but may contain minor errors that do not harm the usefulness of the answer [Scored as +0.5]
3. **Missing**: The response is "I don't know", "I'm sorry I can't find …", a system error such as an empty response, or a request from the system to clarify the original question [Scored as 0]
4. **Incorrect**: The response provides wrong or irrelevant information to answer the user's question [Scored as -1]

For automatic evaluation, CRAG classifies an answer as perfect if it exactly matches the ground truth. If not, then it asks an LLM to do the classification. It uses two LLM evaluators. You can read more about CRAG here - https://arxiv.org/pdf/2406.04744

Other noteworthy benchmark datasets are MedRAG (https://github.com/Teddy-XiongGZ/MedRAG) which focusses on Medical Information, CRUD-RAG (https://arxiv.org/pdf/2401.17043) which focusses on Chinese language and FeB4RAG (https://arxiv.org/abs/2402.11891) focusing on federated search. If you're developing an LLM application which has accurate and contextual generation as its core proposition, you will be able to communicate the quality of your application by showing how it performs on different benchmarks. Table 5.2 compares the different benchmarks.

Table 5.2 RAG Benchmarks

Benchmark	Dataset	Task	Metrics	Applicability
SQuAD	Stanford Question Answering Dataset	Open-domain QA	Exact Match (EM), F1-score	General QA tasks, model evaluation on comprehension accuracy
Natural Questions	Real Google Search Queries	Open-domain QA	F1-score	Real-world QA, information retrieval from large corpora
HotpotQA	Wikipedia-based QA	Multi-hop QA	EM, F1-score	QA involving multiple documents, complex reasoning tasks
BEIR	Multiple Datasets	Information Retrieval	nDCG@10	Comprehensive IR model evaluation across multiple domains
RGB	News Articles, ChatGPT-generated Q&A	Robust QA	Accuracy, Rejection Rate, Error Detection Rate, Error Correction Rate	Robustness and reliability of RAG systems
Multihop RAG	HKUST-curated queries	Complex QA	Various	RAG applications requiring multi-source synthesis
CRAG	Multiple sources (Finance, Sports, Music, etc.)	Factual QA	Four-class Evaluation (Perfect, Acceptable, Missing, Incorrect)	Evaluating factual QA with diverse question types

BENCHMARKING USING LANGCHAIN LangChain also offers benchmarking on different datasets (https://langchain-ai.github.io/langchain-benchmarks/). A notebook with the benchmarking of our RAG pipeline is present in the GitHub repository of this book (https://github.com/abhinav-kimothi/A-Simple-Introduction-to-RAG)

We have looked frameworks that help in automating the calculation of evaluation metrics and benchmarks that enable comparisons across different implementations and approaches. Frameworks will help you in improving the performance of your system and benchmarks will help comparing it with other systems available in the market.

However, as with any evolving field, there are limitations and challenges to consider. In the next section, we'll examine these limitations and discuss best practices that have emerged to address them, ensuring a more holistic and nuanced approach to RAG evaluation.

5.5 Limitations and Best Practices

There has been a lot of progress made in the frameworks and benchmarks used for evaluating RAG. The complexity in evaluation arises due to the interplay between the retrieval and generation components. In practice, there's a significant reliance on human judgements which are subjective and difficult to scale. Below are a few common challenges and some guidelines to navigate them.

LACK OF STANDARDIZED METRICS

There's no consensus on what the best metrics are to evaluate RAG systems. Precision, recall and F1-score are commonly measured for retrieval but do not fully capture the nuances of generative response. Similarly, commonly used generation metrics like BLEU, ROUGE, etc. do not fully capture the context awareness required for RAG. Using RAG specific metrics like answer relevance, context relevance and faithfulness for evaluation brings in the necessary nuance required for RAG evaluation. However, even for these metrics, there's no standard way of calculation and each framework brings in its own methodology.

Best Practice: Compare the results on RAG specific metrics from different frameworks. Sometimes, it may be warranted to change the calculation method with respect to the use case.

OVER-RELIANCE ON LLM AS A JUDGE

The evaluation of RAG specific metrics (in RAGAs, ARES, etc.) relies on using an LLM as a judge. An LLM is prompted or fine-tuned to classify a response as relevant or not. This adds the complexity of the LLMs ability to do this task. It may be possible that for your specific documents and knowledge bases, the LLM is not very accurate in judging. Another problem that arises is that of self-reference. It is possible that if the judge LLM is same as the generation LLM in your system, you will get a more favorable evaluation.

Best Practice: Sample a few results from the judge LLM and evaluate if the results are in-line with commonly understood business practice. To avoid the self-reference problem, make sure to use a judge LLM different from the generation LLM. It may also help if you use multiple judge LLMs and aggregate their results.

LACK OF USE-CASE SUBJECTIVITY

Most frameworks have a generalized approach toward evaluation. They may not capture the subjective nature of the task relevant to your use-case (content generation vs chatbot vs question-answering, etc.)

Best Practice: Focus on use case specific metrics to assess quality, coherence, usefulness etc. Incorporate human judgements in your workflow with techniques like user feedback, crowd-sourcing or expert ratings.

BENCHMARKS ARE STATIC

Most benchmarks are static and do not account for the evolving nature of information. RAG systems need to adapt to real-time information changes, which is not currently tested effectively. There is a lack of evaluation for how well RAG models learn and adapt from new data over time. Most benchmarks are domain-agnostic, which may not reflect the performance of RAG systems in your specific domain.

Best Practice: Use a benchmark that is tailored to your domain. The static nature of benchmarks is limiting. Do not overly rely on benchmarks and augment the use of benchmarks with regularly updating data.

SCALABILITY AND COST

Evaluating large-scale RAG systems is more complex than evaluating basic RAG pipelines. It requires significant computational resources. Benchmarks and frameworks also do not, generally, account for metrics like latency and efficiency which are critical for real world applications.

Best Practice: Employ careful sampling of test cases for evaluation. Incorporate workflows to measure latency and efficiency.

Apart from these, you should also carefully consider the aspects of bias and toxicity, focus on information integration and negative rejection which the frameworks do not evaluate well. It is also important to keep an eye on how these evaluation frameworks and benchmarks evolve.

In this chapter we have looked comprehensively at the evaluation metrics, frameworks and benchmarks that will help you evaluate your RAG pipelines. We used RAGAS to evaluate the pipeline that we have been building.

Till now we have looked at building and evaluating a simple RAG system. This also marks the end of part 2 of this book. You are now familiar with the creation of the RAG knowledge brain using the indexing pipeline, enabling real-time interaction using the generation pipeline and evaluating your RAG system using frameworks and benchmarks.

In the next part, part 3, we will move towards discussing the production aspects of RAG systems. In chapter 6, we will look at strategies and advanced techniques to improve our RAG pipeline which should also reflect in better evaluation metrics. In chapter 7, we will look at the LLMOps stack that enables RAG in production.

5.6 Summary

RAG EVALUATION FUNDAMENTALS

- RAG evaluation assesses how well systems reduce hallucinations and ground responses in provided context.
- Three key quality scores for RAG evaluation are Context Relevance, Answer Faithfulness, and Answer Relevance.
- Four critical abilities required of RAG systems include Noise Robustness, Negative Rejection, Information Integration, and Counterfactual Robustness.
- Additional considerations include latency, robustness, bias, and toxicity of responses.
- Custom use-case specific metrics should be developed to evaluate performance

EVALUATION METRICS

- Retrieval metrics include precision, recall, F1-score, Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), and Normalized Discounted Cumulative Gain (nDCG).
- Accuracy, precision, recall and F1-score do not consider the ranking order of the results
- RAG-specific metrics focus on context relevance, answer faithfulness, and answer relevance.
- Human evaluations and ground truth data play a crucial role in RAG assessment.

EVALUATION FRAMEWORKS

- Frameworks like RAGAs and ARES automate the evaluation process and assist in synthetic data generation.
- RAGAs is an easy to implement framework which can be used for quick evaluation of RAG pipelines.
- ARES uses a more complex approach, including classifier training and confidence interval calculations.

BENCHMARKS

- Benchmarks provide standardized datasets and metrics for comparing different RAG implementations on specific tasks.
- Popular benchmarks likw SQuAD, Natural Questions, HotpotQA, BEIR, etc. focus on retrieval quality
- Recent benchmarks like RGB, Multihop RAG and CRAG are more holistic from a RAG perspective.
- Benchmarks focus on different aspects of RAG performance, such as multi-hop reasoning or specific domains.

LIMITATIONS AND BEST PRACTICES

- Challenges in RAG evaluation include lack of standardized metrics, over-reliance on LLMs as judges, and static nature of benchmarks.
- Best practices include using multiple frameworks, incorporating use-case specific metrics, and regularly updating evaluation data.
- Balancing automated metrics with human judgment and considering use-case specific requirements is crucial.
- The field of RAG evaluation is evolving, with new frameworks and benchmarks continually emerging.
- Developers should stay informed about new developments and adapt their evaluation strategies accordingly.

6 Progression of RAG Systems: Naïve to Advanced, and Modular RAG

This chapter covers

- Limitations of Naïve RAG approach
- Advanced RAG strategies and techniques
- Modular patterns in RAG

In the first two parts of this book, we have, so far, familiarized ourselves with the utility of RAG along with the development and evaluation of a basic RAG system. The basic, or the Naïve RAG approach that we have discussed is, generally, inadequate when it comes to production-grade systems. The next two chapters will focus on more advanced concepts in RAG and the technologies that make RAG possible in production.

In this chapter, we will begin by revisiting the limitations and the points of failure of the Naïve RAG approach. We will discuss the failures at the retrieval, augmentation, and generation stages. Advanced strategies and techniques to address these points of failure will be understood in distinct phases of the RAG pipeline.

Better indexing of the knowledge base leads to better RAG outcomes. We will look at a few data indexing strategies that will build upon the Naïve indexing pipeline to improve the searchability of the knowledge base.

In the generation pipeline, improvements will be examined in three stages – pre-retrieval stage, retrieval stage and post-retrieval stage. Pre-retrieval techniques focus on manipulating and improving the input user query. Retrieval strategies focus on better matching of the user query to the documents in the knowledge base. Lastly, in the post-retrieval stage, the focus is on aligning the retrieved context with the desired result and making it suitable for generation.

In the last part of this chapter, we will discuss a modular approach to RAG that is emerging to find applicability in RAG systems. The modular approach is an architectural enhancement to the basic RAG system.

Note that the strategies and techniques for RAG improvement are expansive, and this chapter will highlight a few of the popular ones. The chapter is interspersed with code examples but for a more exhaustive supporting code, it is advised to check out the source code repository of this book.

By the end of this chapter, you should –

- Understand why the Naïve approach to RAG is not suitable for production
- Be aware of indexing strategies that make the RAG knowledge base more efficient
- Know some of the popular pre-retrieval, retrieval and post-retrieval techniques.
- Be familiar with the modular approach to RAG

RAG powers a variety of AI applications. However, there is a certain aspect of uncertainty when it comes to the outcomes. Inaccuracies in retrieval, disjointed context and incoherence in the LLM outputs need to be addressed before taking RAG to production. In a very short time, researchers and practitioners have experimented with innovative techniques to improve the relevance and faithfulness of RAG systems. Before we look at these techniques, it is important to note why a Naïve RAG approach often doesn't find its way into a production environment.

6.1 Limitations of Naïve RAG

Naïve RAG can be thought of as the earliest form of RAG that gained popularity after the release of ChatGPT and the rise of LLM technology. As we have seen so far in this book, it follows a linear process of indexing, retrieving, augmenting and generation. This process falls in a "retrieve then read" framework - which means that there's a retriever that is retrieving information and then there's an LLM that is reading this information to generate the results. We see this in Figure 6.1 below.

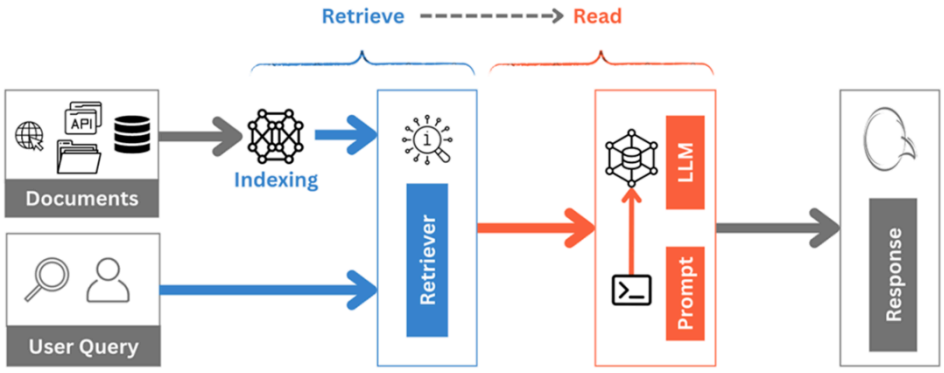

Figure 6.1 Naïve RAG is a sequential "Retrieve then Read" process.

The Naïve RAG approach is marred with drawbacks at each of the three stages of retrieval, augmentation and generation as shown in figure 6.2 below.

- **Retrieval**: Naïve retrieval is often observed to have low precision that leads to irrelevant information being retrieved. It also has a low recall which means that relevant information is missed leading to incomplete results.
- **Augmentation**: There is a real possibility of redundancy and repetition when multiple retrieved documents have similar information. Also, when information is sourced from different documents, the context becomes disjointed. There's also the issue of context length of the LLMs which has an impact on the volume of retrieved context that can be passed onto the LLM for generation.
- **Generation**: With the inadequacies of the upstream processes, the generation suffers from hallucination and lack of groundedness of the generated content. The LLM faces challenge in reconciling information. The challenges of toxicity and bias also persist. It is also noticed, sometimes, that the LLM becomes over-reliant on the retrieved context and forgets to draw from its own parametric memory.

Figure 6.2, which follows, summarizes these drawbacks.

Figure 6.2 Drawbacks of Naïve RAG at each stage of the process

In the last few years, a lot of research and experimentation has been done to address these drawbacks. Early approaches involved pre-training language models. Techniques involving fine-tuning of the LLMs, embeddings models and retrievers have also been tried. These techniques require training data and re-computation of model weights, generally, using supervised learning techniques. Since this is a foundational guide, we will not go into these complex techniques.

This chapter will cover some interventions, techniques and strategies that are used at different stages of the two RAG pipelines – the indexing pipeline and the generation pipeline. Though, the array of such interventions is endless, we will highlight some of the more popular ones in the subsequent sections of this chapter. We will divide them into three stages – pre-retrieval stage, retrieval stage, and post-retrieval stage.

6.2 Advanced RAG techniques

Advanced techniques in RAG have continued to emerge since the earliest experiments with Naïve RAG. There are three stages in which we can discuss these techniques –

1. **Pre-retrieval Stage**: Like the name suggests, there are certain interventions that can be employed before the retriever comes into action. This broadly covers two aspects
    a. *Index Optimization* – The way documents are stored in the knowledge base
    b. *Query Optimization* – Optimizing the user query so it aligns better to the retrieval and generation tasks
2. **Retrieval Stage**: Certain strategies can improve the recall and precision of the retrieval process. This goes beyond the capability of the underlying retrieval algorithms that we discussed in Chapter 4.
3. **Post-retrieval Stage**: Once the information has been retrieved, the context can be further optimized to better align with the generation task and the downstream LLM.

With techniques employed at these three stages, the Advanced RAG process follows a 'Rewrite then Retrieve then Re-rank then Read' frameworks. Two additional components of Rewrite and Re-rank are added, and the Retrieve component is enhanced in comparison with Naïve RAG. This can be seen in the figure 6.3 below.

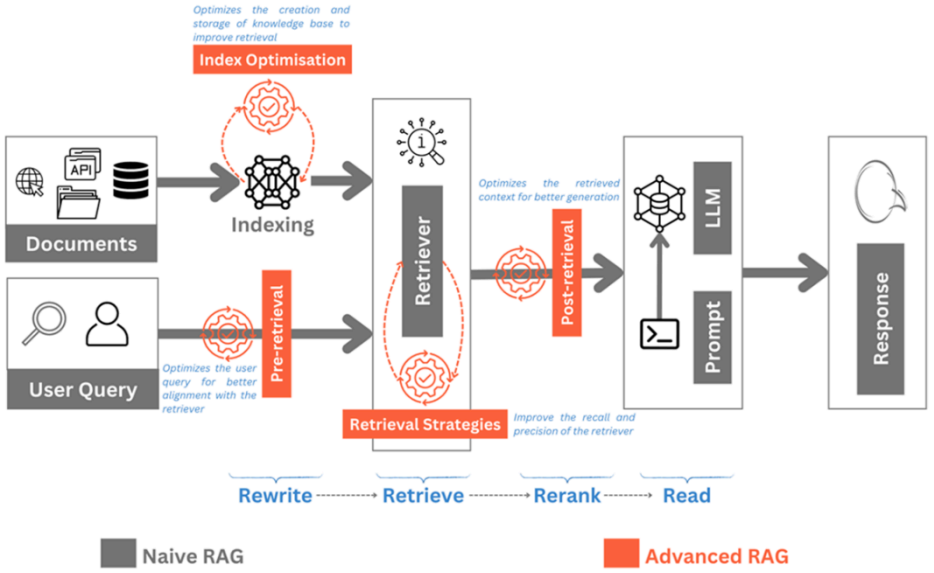

Figure 6.3 Advanced RAG is a Rewrite-Retrieve-Rerank-Read process as compared to a Retrieve-Read Naïve RAG process

We will now explore these components one by one beginning with the pre-retrieval stage.

6.2.1 Pre-retrieval Techniques

The primary objective of employing pre-retrieval techniques is to facilitate better retrieval. We have noted that the retrieval stage of Naïve RAG suffers from low recall and low precision –irrelevant information is retrieved and not all relevant information is retrieved. This can happen, largely because of two reasons –

a. **Knowledge Base is not suited for retrieval**: If the information in the knowledge base is not stored in a manner that is easy to search through, then the quality of retrieval will remain suboptimal. To address this, *Index Optimization* is done in the indexing pipeline for more efficient storage of the knowledge base.

b. **Retriever doesn't completely understand the input query**: In generative AI applications, the control over the user query is, generally, limited. The level of details a user provides is subjective. The retriever sometimes may misunderstand or not completely understand the context of the user query. *Query Optimization* addresses this aspect of the challenge with the Naïve RAG.

Both index Optimization and query Optimization are carried out before the retriever is invoked. This is the only stage that recommends interventions both in the indexing pipeline and the generation pipeline. We will look at a few techniques in each of these.

INDEX OPTIMIZATION

Index Optimization, like discussed above, is employed in the indexing pipeline. The objective of index Optimization is to set up the knowledge base for better retrieval. Some of the popular strategies are as follows.

Chunk Optimization

We discussed in Chapter 3, the significance of chunking in the indexing pipeline. Chunking large documents into smaller segments plays a crucial role in retrieval and handling the context length limits of LLMs. There are certain techniques that aim for better chunking and efficient retrieval of the chunks like –

- **Chunk Size Optimization**: The size of the chunks can have a significant impact on the quality of the RAG system. While large sized chunks provide better context, they also carry a lot of noise. Smaller chunks, on the other hand, have precise information but they might miss important information. For instance, consider a legal document that's 10,000 words long. If we chunk it into 1,000-word segments, each chunk might contain multiple legal clauses, making it hard to retrieve specific information. Conversely, chunking it into 200-word segments allows for more precise retrieval of individual clauses, but may lose the context provided by surrounding clauses. Experimenting with chunk sizes can help find the optimal balance for accurate retrieval. The processing time also depends on the size of the chunk. Chunk size, therefore, has a significant impact on retrieval accuracy, processing speed and storage efficiency. The ideal chunk size varies with the use case and depends on balancing factors like document types and structure, complexity of user query and the desired response time. There is no one size fits all approach towards optimizing chunk sizes. Experimentation and evaluation of different chunk sizes on metrics like faithfulness, relevance, response time (as discussed in chapter 5) can help in identifying the optimal chunk size for the RAG system. Chunk size optimization may require periodic reassessment as data, or requirements change.
- **Context-Enriched Chunking**: This method adds the summary of the larger document to each chunk to enrich the context of the smaller chunk. This makes more context available to the LLM without adding too much noise. It also improves the retrieval accuracy and maintains semantic coherence across chunks. This is particularly useful in scenarios where a more holistic view of the information is crucial. While this approach enhances the understanding of the broader context, it adds a level of complexity and comes at the cost of higher computational requirements, increased storage needs and possible latency in retrieval. Below is an example of how context enrichment can be done using GPT-4o-mini, OpenAI embeddings and FAISS.

```
from langchain_community.document_loaders import AsyncHtmlLoader #A
from langchain_community.document_transformers import Html2TextTransformer #A
url=https://en.wikipedia.org/wiki/2023_Cricket_World_Cup #A
loader = AsyncHtmlLoader (url) #A
data = loader.load() #A
html2text = Html2TextTransformer() #A
document_text=data_transformed[0].page_content #A


summary_prompt = f"Summarize the given document in a single paragraph\ndocument:
{document_text}" #B
from openai import OpenAI #B
client = OpenAI() #B

response = client.chat.completions.create( #B
  model="gpt-4o-mini", #B
  messages= [ #B
    {"role": "user", "content": summary_prompt} #B
      ] #B
) #B

summary=response.choices[0].message.content #B

from langchain_text_splitters import RecursiveCharacterTextSplitter #C
text_splitter = RecursiveCharacterTextSplitter( #C
chunk_size=1000, #C
chunk_overlap=200) #C
chunks=text_splitter.split_text(data_transformed[0].page_content) #C

context_enriched_chunks = [answer + "\n" + chunk for chunk in chunks] #D

embedding = OpenAIEmbeddings(openai_api_key=api_key) #E
vector_store = FAISS.from_texts(context_enriched_chunks, embedding) #E
```

#A Loading text from Wikipedia page
#B Generating summary of the text using GPT-4o-mini model
#C Creating Chunks using Recursive Character Splitter
#D Enriching Chunks with Summary Data
#E Creating embeddings and storing in FAISS index

- **Fetch Surrounding Chunks**: In this technique, chunks are created at a granular level, say, at a sentence level and when a relevant chunk of text is found in response to a query, the system retrieves not only that chunk but also the surrounding chunks. This makes the search granular but also performs contextual expansion by retrieving adjacent chunks. It is useful in long form content like books and reports where information flows across paragraphs and sections. This technique also adds a layer of processing cost and latency to the system. Apart from that, there is a possibility of diluting the relevance as the neighboring chunks may contain noise.

Chunk optimization is an effective step towards better RAG systems. Though it presents challenges such as managing the costs, system latency and storage efficiency, optimizing chunking can fundamentally improve the retrieval and generation process of the RAG system.

Metadata Enhancements

A common way of defining metadata is “data about data”. Metadata describes other data. It can provide information like a description of the data, time of creation, author, etc. While metadata is useful for managing and organizing data, in the context of RAG, metadata enhances the searchability of data. A few ways in which metadata is crucial in improving RAG systems are -

- **Metadata filtering**: Adding metadata like timestamp, author, category, etc. can enhance the chunks. While retrieving, chunks can first be filtered by relevant metadata information before doing a similarity search. This improves retrieval efficiency and reduces noise in the system. For example, using the timestamp filters can help avoid outdated information in the knowledge base. If a user searches for 'latest COVID-19 travel guidelines,' metadata filtering by timestamp ensures that only the most recent guidelines are retrieved, avoiding outdated information.
- **Metadata enrichment**: Time stamp, author, category, chapter, page number etc. are common metadata elements that can be extracted from documents. However, even more valuable metadata items can be constructed. This can be a summary of the chunk, extracting tags from the chunk. One particularly useful technique is Reverse Hypothetical Document Embeddings. It involves using a language model to generate potential queries that could be answered by each document or chunk. These synthetic queries are then added to the metadata. During retrieval, the system compares the user's query with these synthetic queries to find the most relevant chunks.

Metadata is a great tool in your repertoire for improving the accuracy of the retrieval system. However, a degree of caution must be exercised while adding metadata to the chunks. Designing the metadata schema is important to avoid redundancies and managing processing and storage costs. Providing improved relevance and accuracy, metadata enhancement has become extremely popular in contemporary RAG systems.

Index Structures

Another important aspect of the knowledge base is also how well the information is structured. In Naïve RAG approach, there is no structural order to documents/chunks. However, for a more efficient retrieval, a few indexing structures have become popular and effective.

- **Parent-child document structure**: In a parent-child document structure, documents are organized hierarchically. The parent document contains overarching themes or summaries, while child documents delve into specific details. During retrieval, the system can first locate the most relevant child documents and then refer to the parent documents for additional context if needed. This approach enhances the precision of retrieval while maintaining the broader context. At the same time, this hierarchical structure can present challenges in terms of memory requirements and computational load.
- **Knowledge Graph index**: Knowledge graphs organize data in a structured manner as entities and relationships. Using knowledge graph structures not only increases the contextual understanding but also equips the system with enhanced reasoning capabilities and improved explainability. Knowledge Graph creation and maintenance, however, is an expensive process. Knowledge Graph powered RAG, also called GraphRAG, is an emerging Advanced RAG pattern that has demonstrated significant improvements in RAG performance. We will discuss GraphRAG in detail in chapter 8.

Index structure, perhaps, has the biggest impact on index Optimization for retrieval. It, however, introduces storage and memory burden on the system and impacts search time performance. Index structure Optimization is therefore advised in large scale systems where the true potential of concepts like GraphRAG and hierarchical index can be realized.

> **FINETUNING EMBEDDINGS** We have discussed in the previous chapters how embeddings are a crucial component of RAG. They are used to calculate the semantic similarity between the user query and the documents stored in the knowledge base. Generally available embeddings models have been trained on commonly spoken language. When dealing with domain-specific or specialized content, these models may not yield good results. Fine-tuning embedding models allows you to optimize vector representations for your specific domain or task, leading to more accurate retrieval of relevant context. Fine-tuning is a slightly complex process since it requires curation of training dataset and resources for recalculating the embeddings model. In case you're dealing with highly specialized domains where the vocabulary is different from commonly spoken languages, you should consider fine-tuning the embedding model for your domain.

Index Optimization like the indexing pipeline is a periodic process and does not happen real-time. The objective of index Optimization is to set up the knowledge base for better retrieval. One must also be mindful of the added complexity that leads to an increase in computational, memory and storage requirements. Figure 6.4 is an illustrative workflow of an index optimized knowledge base.

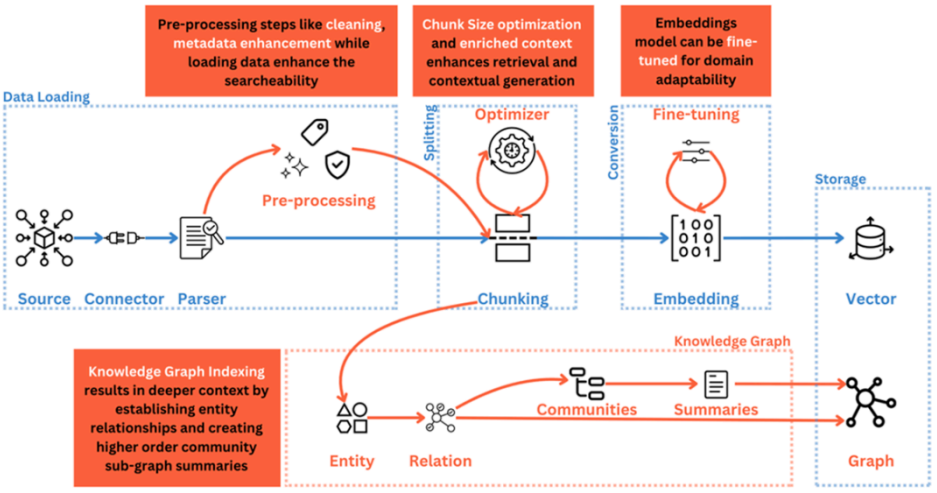

Figure 6.4 An illustration of an index optimized knowledge base

QUERY OPTIMIZATION

The second stage of pre-retrieval techniques is a part of the generation pipeline. The objective of this stage is to optimize the input user query in a manner that makes it better suited for the retrieval tasks. Some of the popular query optimization strategies are listed below.

Query Expansion

In query expansion, the original user query is enriched with the aim of retrieving more relevant information. This helps in increasing the recall of the system and overcomes the challenge of incomplete or very brief user queries. Some of the techniques that expand user queries are -

- **Multi-query expansion**: In this approach, multiple variations of the original query are generated using an LLM and each variant query is used to search and retrieve chunks from the knowledge base. For a query 'How does climate change affect polar bears?', multi-query expansion might generate 'Impact of global warming on polar bears', 'What are the consequences of climate change for polar bear habitats?'. Let us look at a simple example of multi-query generation using GPT 4o-mini model

```
original_query="How does climate change affect polar bears?"
num=5

expansion_prompt=f"Generate {num} variations of the following query: {original_query}.
Respond in JSON format." #A

from openai import OpenAI                                          #B
client = OpenAI()                                                  #B
response = client.chat.completions.create(                         #B
  model="gpt-4o-mini",                                             #B
  messages= [                                                      #B
    {"role": "user", "content": expansion_prompt}                  #B
      ],                                                           #B
          response_format={ "type": "json_object" }                #B
)                                                                  #B

expanded_queries=response.choices[0].message.content               #C
```

#A Craft the prompt for query expansion
#B Using GPT 4o-mini to generate expanded queries
#C Extract the text from the response object

- **Sub-query expansion**: Sub-query approach is quite like the multi-query approach. In this approach instead of generating variations of the original query, a complex query is broken down into simpler sub-queries. This approach is inspired from the least to most prompting technique where complex problems are broken down into simpler subproblems and are solved one by one. A sub-query expansion on the same query as above, 'How does climate change affect polar bears?', may generate 'How does melting sea ice influence polar bear hunting and feeding behaviors?' and 'What are the physiological and health impacts of climate change on polar bears?. The approach to sub-query is similar to multi-query except the changes to the prompt.

```
sub_query_expansion_prompt=f"Break down the following query into {num} sub-queries
targeting different aspects of the query: {original_query}.Respond in JSON format. "
```

- **Step back expansion**: The term comes from the step-back prompting approach where the original query is abstracted to a higher-level conceptual query. During retrieval, both the original query and the abstracted query are used to fetch chunks. Similar to above example, an abstracted step back query may be "What are the ecological impacts of climate change on arctic ecosystems?". Below is an example of the prompt that can be used.

```
step_back_expansion_prompt = f"Given the query: {original_query}, generate a more
abstract, higher-level conceptual query. "
```

While multi-query expansion generates various rephrasings or synonyms of the original query to cast a wider net during retrieval, sub-query expansion breaks down a complex query into simpler, component queries to target specific pieces of information and step back expansion abstracts the query to a higher-level concept to capture broader context.

Query expansion also presents its own set of challenges that need to be considered while implementing this strategy. While query expansion may increase recall by matching more documents, it may reduce the precision. The expansion terms need to be carefully selected to avoid contextual drift from the original query. Overexpansion can dilute the focus from the original query. Despite the challenges, query expansion has proved to be an effective technique to improve the recall of retrieval and generating more context aware responses.

Query Transformation

Compared to query expansion, in query transformation, instead of the original user query retrieval happens on a transformed query which is more suitable for the retriever.

- **Rewrite**: Queries are rewritten from the input. The input in quite a few real-world applications may not be a direct query or a query suited for retrieval. Based on the input a language model can be trained to transform the input into a query which can be used for retrieval. A user's statement like 'I can't send emails from my phone' can be rewritten as 'Troubleshooting steps for resolving email sending issues on smartphones,' making it more suitable for retrieval.
- **HyDE**: Hypothetical document embedding or HyDE is a technique where the language model first generates a hypothetical answer to the user's query without accessing the knowledge base. This generated answer is then used to perform a similarity search against the document embeddings in the knowledge base, effectively retrieving documents that are similar to the hypothetical answer rather than the query itself. Below is an example that generates hypothetical document embeddings.

```
# Original Query
original_query="How does climate change affect polar bears?" #A

# Prompts for generating HyDE
system_prompt="You are an expert in climate change and arctic life." #B
hyde_prompt=f"Generate an answer to the question: {original_query}" #B

# Using OpenAI to generate hypothetical answer

from openai import OpenAI #C
client = OpenAI() #C
response = client.chat.completions.create( #C
  model="gpt-4o-mini", #C
  messages= [ #C
    {"role": "system", "content": system_prompt}, #C
    {"role": "user", "content": hyde_prompt} #C
  ] #C
) #C

hy_answer=response.choices[0].message.content #C

# Using OpenAI Embeddings to convert hyde into embeddings
embeddings = OpenAIEmbeddings(model="text-embedding-3-large") #D
hyde = embeddings.embed_query(hy_answer) #D
```

#A Original Query
#B Prompts for generating HyDE
#C Using OpenAI to generate hypothetical answer
#D Using OpenAI Embeddings to convert hyde into embeddings

Challenges similar to query expansion like drift from original query and maintaining intent also persist in query transformation strategies. Effective rewriting and transformation of the query results in enhancing the context awareness of the system.

Query Routing

Different queries can demand different retrieval methods. Based on criteria like intent, domain, language, complexity, source of information etc., queries are needed to be classified so that they can follow the appropriate retrieval method. This is the idea behind optimizing the user query by routing it to the appropriate workflow. Types of routing techniques include –

- **Intent Classification**: A pre-trained classification model is used to classify the intent of the user query to select the appropriate retrieval method. A modification to this technique is prompt-based classification where instead of a pre-trained classifier, an LLM is prompted to categorize the query into an intent.

- **Metadata Routing**: In this approach, keywords and tags are extracted from the user query and then a filtering is done on the chunk metadata to narrow down the scope of search.
- **Semantic Routing**: In this approach, the user query is matched with pre-defined set of queries for each retrieval method. Wherever the similarity between the user query and pre-defined queries is the highest, that retrieval method is invoked.

In customer support chatbots, query routing ensures that technical queries are directed to databases with troubleshooting guides, while billing questions are routed to account information, enhancing user satisfaction.

Implementing query routing takes effort and skill. It introduces a whole new predictive component bringing uncertainty to the process. It must therefore be carefully crafted. Query routing is a must when dealing with variability in source data and query types.

Though the universe of pre-retrieval strategies and techniques is expansive and ever evolving, we have looked at few of the most popular and effective techniques in this section. Bear in mind that the applicability of the strategies will depend on the nature of the content in the knowledge base and the use case. However, using each of these strategies will result in incremental gain in the RAG system performance. Now that we have set up the knowledge base and the user query for better retrieval, let us discuss important retrieval strategies in the next section.

6.2.2 Retrieval Strategies

Interventions in the pre-retrieval stage can bring significant improvements in the performance of the RAG system if the query and the knowledge base becomes well aligned with the retrieval algorithm. We have discussed quite a few retrieval algorithms in chapter 4. In this section, we will focus on strategies that can be employed for better retrieval.

HYBRID RETRIEVAL

Hybrid retrieval strategy is an essential component of production-grade RAG systems. It involves combining retrieval methods for improved retrieval accuracy. This can mean simply using a keyword-based search along with semantic similarity. It can also mean combining all sparse embedding, dense embedding vector and knowledge graph-based search. The retrieval can be a union or an intersection of all these methods depending on the requirements of precision and recall. It generally follows a weighted approach to retrieval. Figure 6.5 shows the hybrid retriever querying graph and vector storage.

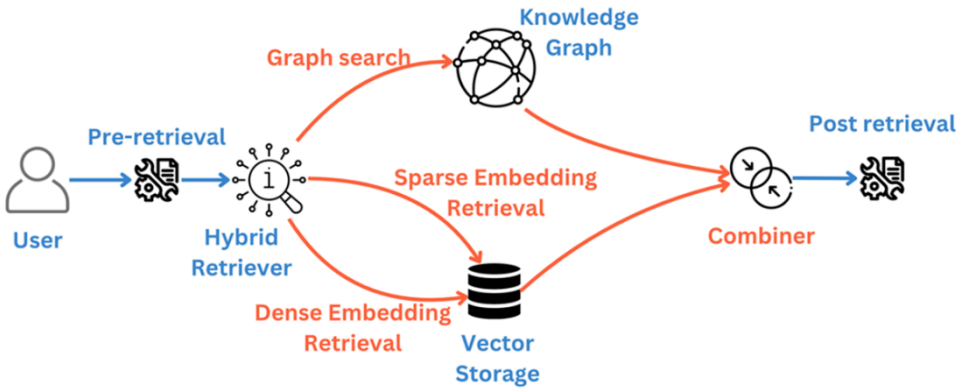

Figure 6.5 Hybrid retriever employs multiple querying techniques and combines the results

ITERATIVE RETRIEVAL

Instead of using a retrieve-generate linear process, iterative retrieval strategy searches the knowledge base repeatedly based on the original query and the generated text. This allows the system to gather more information by refining search based on initial results. It is useful when solving multi-hop or complex queries. While effective, iterative retrieval can lead to longer processing times and may introduce challenges in managing larger amounts of retrieved information. There are examples of iterative retrieval that have demonstrated remarkably improved performance like Iter-RetGen which is an iterative approach that alternates between retrieval and generation steps.

RECURSIVE RETRIEVAL

Recursive retrieval strategy builds upon the idea of iterative retrieval by transforming the query iteratively depending on the results obtained. While the initial query is used to retrieve the chunks, new focused queries are generated based on these chunks. It, therefore, leads to a better ability of finding scattered information across document chunks and a more coherent and contextual response. Iterative Retrieval Chain-of-Thought (IRCoT) is a recursive retrieval technique which combines iterative retrieval with chain-of-thought prompting.

ADAPTIVE RETRIEVAL

Adaptive retrieval also follows the approach of repeated retrieval cycles. In adaptive retrieval strategies, an LLM is enabled to determine the most appropriate moment and content for retrieval. The objective of adaptive retrieval is to make the retrieval process more personalized to users and context. It is applied in areas like adapting queries depending on user behavior or adjusting retrieval based on user performance. FLARE and Self-RAG are two popular examples of adaptive retrieval. Self-RAG introduces 'reflection tokens' that enable the model to introspect and decide when additional retrieval is necessary. FLARE (Forward-Looking Active Retrieval Augmented Generation) predicts future content needs based on the current generation and retrieves relevant information proactively. Adaptive Retrieval is a part of a broader trend of Agentic AI. Agentic AI refers to AI systems that can make autonomous decisions during tasks, adapting their actions based on the context. In the context of RAG, Agentic RAG involves AI agents that dynamically decide when and how to retrieve information, enhancing the flexibility and efficiency of the retrieval process. Agentic AI is an important emerging RAG pattern. We will discuss Agentic RAG in detail in Chapter 8.

Figure 6.6 compares the three retrieval strategies that focus on repeated retrieval cycles. While recursive and iterative approaches need a threshold to break out of the iterations, in the adaptive approach a judge model decides on-demand retrieval and generation steps.

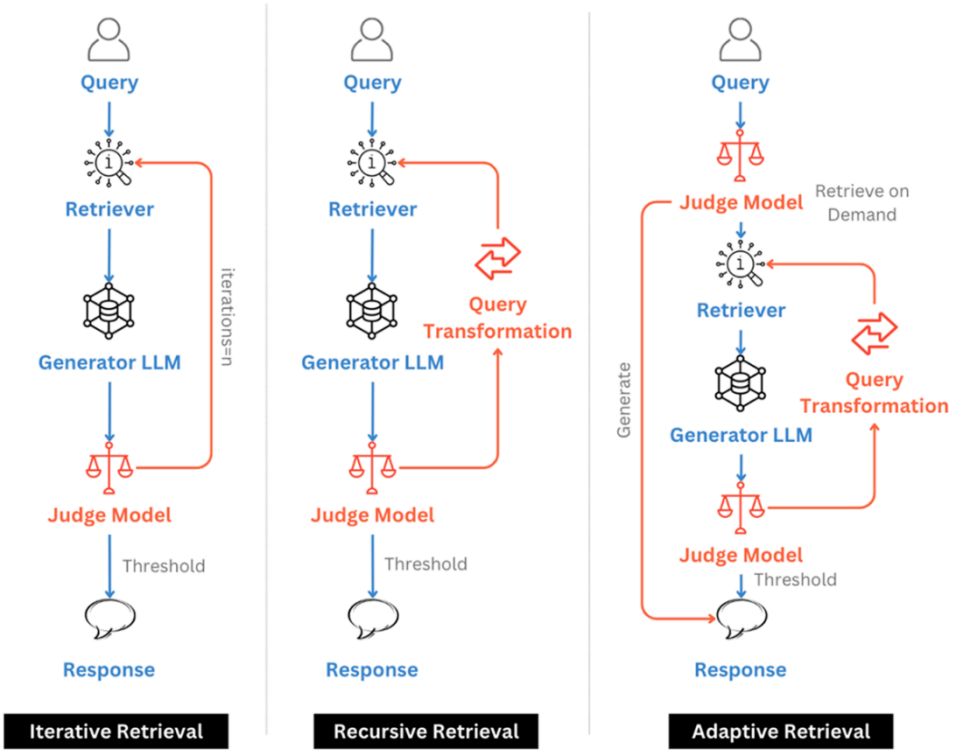

Figure 6.6 Iterative, Recursive and Adaptive retrieval incorporate repeated retrieval cycles. (Source – Adapted from Retrieval-Augmented Generation for Large Language Models: A Survey, Gao et al)

All the advanced retrieval strategies introduce overheads in terms of computational complexity and therefore the accuracy must be balanced against the cost and latency of the system.

By employing advanced pre-retrieval techniques and a suitable retrieval strategy, we can expect that richer, deeper and more relevant context is being retrieved from the knowledge base. Even when relevant context is retrieved, the LLM may struggle to assimilate all the information. To address this issue, in the next section, we will discuss a couple of post-retrieval strategies that help curate the context before augmenting the prompt with the necessary information.

6.2.3 Post-Retrieval Techniques

Even if the retrieval of the chunks happens in an expected manner, there is still a point of failure that remains. The LLM might not be able to process all the information. This may be due to redundancies or disjointed nature of the context amongst many other reasons. At the post-retrieval stage the approaches of reranking and compression help in providing better context to the LLM for generation.

COMPRESSION

Excessively long context has the potential of introducing noise into the system. This diminishes the LLM's capability to process information. As a result, hallucinations and irrelevant responses to the query may persist. In prompt compression, language models are used to detect and remove unimportant and irrelevant tokens. Apart from making the context more relevant, prompt compression also has a positive influence on the cost and efficiency. Another advantage of prompt compression is to be able to reduce the size of the prompt so that it can fit into the context window of the LLM. COCOM is a context compression method which compresses contexts into a small number of context embeddings. Similarly, xRAG is a method that uses document embeddings as features. Compression can lead to loss of information and therefore a balance needs to be achieved between compression and performance. A very simplistic prompt to compress a large retrieved context can be as follows –

```
compress_prompt = f"Compress the following document into a shorter version, retaining
only the essential information:\n\n{document}"
```

RERANKING

Reordering all the retrieved documents ensures that the most relevant information is prioritized for the generation step. It refines retrieval results by prioritizing documents that are more contextually appropriate for the query, improving the overall quality and accuracy of information used for generation. This also addresses the question of prioritization when a hybrid approach to retrieval is employed and improves the overall response quality. There are commonly available rerankers like multi-vector, Learning to Rank (LTR), BERT based and even hybrid rerankers that can be employed. Specialized APIs like Cohere Rerank offer pre-trained models for efficient reranking integration.

In this section, we discussed some of the popular Advanced RAG strategies and techniques that are employed at different stages of the RAG pipeline. It is important to also consider the trade-offs that come with these techniques. Almost any advanced technique will introduce overheads to the system. These can be in the form of computational load, latency in the system and increased storage and memory requirements. Therefore, these techniques warrant a performance versus overhead assessment catered to specific use cases. Table 6.1 provides a summary of the twelve strategies that we have discussed so far.

Table 6.1 Advanced RAG strategies with their benefits and limitations

Strategy	Description	Benefits	Challenges
Chunk Optimization	Adjusting document chunks for optimal size and context.	Improves retrieval accuracy, processing speed, and storage.	Requires experimentation; optimal chunk varies by use case.
Metadata Enhancements	Enriching chunks with additional metadata for better filtering and searchability.	Improves retrieval efficiency; reduces noise.	Requires careful schema design; manages processing costs.
Index Structures	Organizing data in structured formats for efficient retrieval.	Enhances accuracy and context in retrieval.	Increases memory and computational load.
Query Expansion	Enriching the user query to retrieve more relevant information.	Increases recall; overcomes brief queries.	May reduce precision; risk of contextual drift.
Query Transformation	Modifying the user query for better retrieval suitability.	Enhances context awareness; maintains intent.	Potential for misinterpretation; drift from original query.
Query Routing	Directing queries to appropriate retrieval methods based on classification.	Enhances retrieval by matching method to query type.	Introduces uncertainty; requires careful crafting.
Hybrid Retrieval	Combining multiple retrieval methods (e.g., keyword and semantic).	Improves retrieval accuracy and robustness.	Increased complexity; requires method weighting.
Iterative Retrieval	Repeatedly searching based on initial results and query refinement.	Gathers more comprehensive information; refines search.	Longer processing times; managing more data.
Recursive Retrieval	Iteratively transforming the query based on obtained results.	Finds scattered information; provides coherent responses.	Similar to iterative retrieval; potential for increased load.
Adaptive Retrieval	LLM decides when and what to retrieve during generation.	Personalized and context-aware retrieval; dynamic adaptation.	Increased computational complexity; part of agentic AI.
Compression	Reducing context length by removing irrelevant information.	Fits within LLM context window; reduces noise and costs.	Potential loss of important information; needs balance.

Reranking	Reordering retrieved documents to prioritize relevance.	Enhances response quality; ensures most relevant info is used.	Requires additional models; may introduce overhead.

Figure 6.7 is an illustrative example of what a generation pipeline looks like after incorporating advanced techniques.

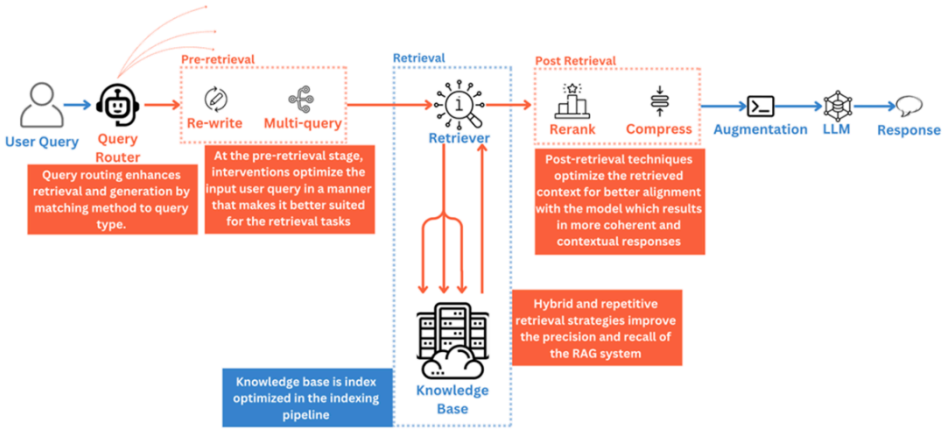

Figure 6.7 Illustrative example of advanced generation pipeline.

While these advanced strategies and techniques discussed above are extremely useful in improving performance, a RAG system also needs to allow for customization and flexibility. This is because you may need to quickly adopt different techniques as the nature of data and queries evolve. A Modular RAG approach that we will discuss in the next section aims to provide greater architectural flexibility over the traditional RAG system.

6.3 Modular RAG

AI systems are becoming increasingly complex and demand more customizable, flexible, and scalable RAG architectures. The emergence of Modular RAG is a leap forward in the evolution of RAG systems. Modular RAG breaks down the traditional monolithic RAG structure into interchangeable components. This allows for tailoring of the system to specific use cases. Modular approach brings modularity to RAG components like retrievers, indexing, and generation, while also adding more modules like search, memory, and fusion. We can think of the Modular RAG approach in two parts –

1. Core components of RAG developed as flexible, interchangeable modules
2. Specialized modules to enhance the core features of retrieval, augmentation and generation

6.3.1 Core Modules

The core components of the RAG system i.e. indexing, retrieval, augmentation and generation along with the advanced pre and post-retrieval techniques are composed as flexible, interchangeable modules in the Modular RAG framework.

- **Indexing Module**: The Indexing Module serves as the foundation for building the knowledge base. By modularizing this component, developers can choose from various embedding models for advanced semantic understanding. Vector stores can be interchanged based on scalability and performance needs. Additionally, chunking methods can be adapted to the data structure, whether it's text, code, or multimedia content, ensuring optimal indexing for retrieval.
- **Retrieval Module**: The Retrieval Module enables the use of diverse retrieval algorithms. For instance, developers can switch between semantic similarity search using dense embeddings and traditional keyword-based search like BM25. This flexibility allows for tailoring retrieval methods to the specific requirements of the application, such as prioritizing speed, accuracy, or resource utilization. For example, a customer support chatbot might use semantic search during off-peak hours for higher accuracy and switch to keyword search during peak hours to handle increased load. The modular retrieval component allows this dynamic interchange of retrieval strategies based on real-time needs.
- **Generation Module**: In the Generation Module, the choice of Language Model (LLM) is modular. Developers can select from models like GPT-4 for complex language generation or smaller models for cost efficiency. This module also handles prompt engineering for augmentation to guide the LLM in generating accurate and relevant responses.
- **Pre-retrieval Module**: Allows for flexibility of pre-retrieval techniques to improve quality of indexed content and user query
- **Post-retrieval Module**: Like the pre-retrieval module, this module allows for flexible implementation of post-retrieval techniques to refine and optimize the retrieved context

You may note that the first three modules complete the Naïve RAG approach, and the addition of the pre-retrieval and post-retrieval module enhance the Naïve RAG into an Advanced RAG implementation. It can also be said that Naïve RAG is a special (and limited) case of Advanced RAG.

6.3.2 New Modules

The Modular RAG framework has introduced several new components to enhance the retrieval and generation capabilities of Naïve and Advanced RAG approaches. Some of these components/modules are -

- **Search**: The search module is aimed at performing search on different data sources. It is customized to different data sources and aimed at increasing the source data for better response generation

- **Fusion**: RAG-Fusion improves traditional search systems by overcoming their limitations through a multi-query approach. The Fusion Module enhances retrieval by expanding the user's query into multiple, diverse perspectives using an LLM. It then conducts parallel searches for these expanded queries, fuses the results by re-ranking and selecting the most relevant information, and presents a comprehensive answer. This approach captures both explicit and implicit information, uncovering deeper insights that might be missed with a single query.
- **Memory**: The Memory Module leverages the inherent 'memory' of the LLM, meaning the knowledge encoded within its parameters from pre-training. This module uses the LLM to recall information without explicit retrieval, guiding the system on when to retrieve additional data and when to rely on the LLM's internal knowledge. It can involve techniques like using reflection tokens or prompts that encourage the model to introspect and decide if more information is needed. For example, when answering a query about historical events, the Memory Module can decide to rely on the LLM's knowledge about World War II to provide context, only retrieving specific dates or figures as needed. This reduces unnecessary retrieval and leverages the model's pre-trained knowledge.
- **Routing**: Routing in the RAG system navigates through diverse data sources, selecting the optimal pathway for a query, whether it involves summarization, specific database searches, or merging different information streams.
- **Task Adapter**: This module makes RAG adaptable to various downstream tasks allowing the development of task-specific end-to-end retrievers with minimal examples, demonstrating flexibility in handling different tasks. The Task Adapter Module allows the RAG system to be fine-tuned for specific tasks like summarization, translation, or sentiment analysis. By incorporating a small number of task-specific examples or prompts, the module adjusts the retrieval and generation components to produce outputs tailored to the desired task, enhancing versatility without extensive retraining.

You may observe that Advanced RAG is a special case within the Modular RAG framework. We also saw earlier how Naïve RAG is a special case of Advanced RAG. This means that the RAG approaches i.e., Naïve, Advanced, and Modular are not competing but are progressive in nature. You may start out by trying out a Naïve implementation of RAG and move to a more modular approach. Figure 6.8 shows the progression of RAG systems.

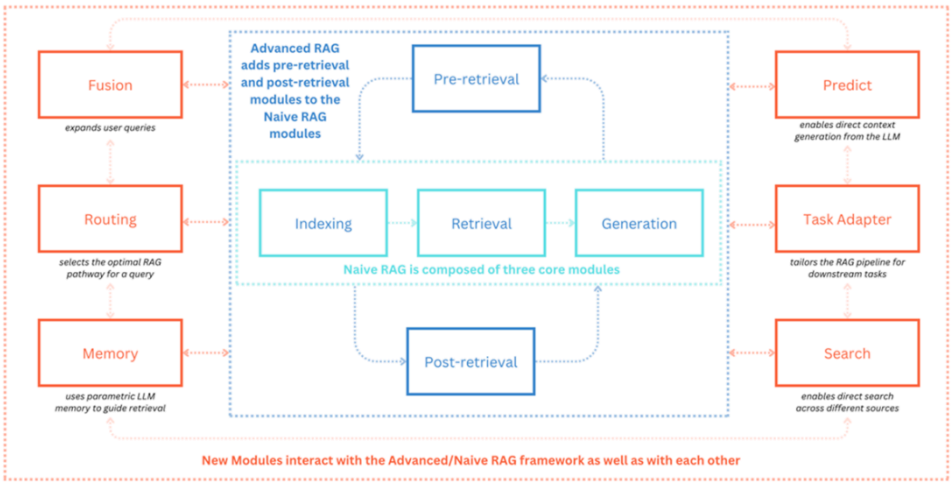

Figure 6.8 Naïve, Advanced and Modular approaches to RAG are progressive in nature. Naïve RAG is a sub-component of Advanced RAG which is a sub-component of Modular RAG

While building a Modular RAG system, remember that each module should be designed to work independently. This will require defining clear inputs and outputs. Along with the independent modules, the orchestration layer should be flexible to allow mixing and matching of modules. One should also bear in mind that a modular approach introduces complexity in the process. Managing interfaces, dependencies, configurations and versions of modules can be complex. Ensuring compatibility and consistency between modules can be challenging. Testing each module independently and collectively requires a robust evaluation strategy. Extra modules may also add latency and inference cost to the system.

Despite the added complexities, the modular approach towards RAG is the state-of-the-art in large-scale RAG systems. It enables rapid experimentation, efficient optimization, and seamless integration of new technologies as they emerge. By offering the ability to mix and match different modules, Modular RAG empowers you to build more robust, accurate, and versatile AI solutions. It also facilitates easier maintenance, updates, and scalability, making it an ideal choice for managing complex, evolving knowledge bases.

This concludes the discussion on improving RAG performance using advanced techniques and a modular framework. Interventions can be employed at different stages of the indexing and generation pipelines. Modular approaches to RAG provide for rapid experimentation, flexibility and a scalable architecture. You will need to experiment to figure out the techniques that help in improving RAG for specific use cases. It is also important to be mindful of the trade-offs. Advanced techniques introduce complexities that have an impact on computation, memory and storage requirements.

This is one aspect of putting RAG in production. Advanced techniques are necessary for RAG systems to achieve acceptable accuracy and efficiency. The other enabler for RAG systems in production are the tools and technologies that form the backbone of the RAG stack. In the next chapter we will look at this technology infrastructure that enables RAG systems.

6.4 Summary

LIMITATIONS OF NAÏVE RAG

- Naïve RAG follows a simple "Retrieve-then-Read" process.
- This approach suffers from low precision and incomplete retrieval.
- Retrieval often misses relevant information and pulls in irrelevant content.
- In the augmentation stage, there is often redundancy from similar retrieved documents.
- Context can become disjointed when sourced from multiple documents.
- The generation stage faces hallucinations and biased outputs.
- The model can overly rely on retrieved data and ignore its internal knowledge.

ADVANCED RAG TECHNIQUES

- The Advanced RAG process follows a 'Rewrite then Retrieve then Re-rank then Read' framework, where the query is optimized through rewriting, retrieval is enhanced for better precision, results are re-ranked to prioritize relevance, and the most relevant information is used for generating the final response.
- Pre-retrieval Techniques:
    - Index Optimization: Improves document storage for better searchability.
    - Chunk Optimization: Balances chunk sizes to avoid losing context or introducing noise.
    - Context-enriched Chunking: Adds summaries to each chunk to improve retrieval.
    - Metadata Enhancements: Adds tags and metadata like timestamps or categories for better filtering.
    - Query Optimization: Expands or rewrites user queries for improved retrieval accuracy.

- Retrieval Techniques:
    - Hybrid Retrieval: Combines keyword-based and semantic searches.
    - Iterative Retrieval: Refines searches by repeatedly querying based on initial results.
    - Recursive Retrieval: Generates new queries based on retrieved chunks to gather more relevant information.

- Post-retrieval Techniques:
    - Compression: Reduces unnecessary context to remove noise and fit within the model's context window.
    - Reranking: Reorders retrieved documents to prioritize the most relevant ones.

MODULAR RAG FRAMEWORK

- Core Modules:
    - Indexing Module: Allows flexible embedding models and vector store options.
    - Retrieval Module: Supports switching between dense and keyword-based retrieval methods.
    - Generation Module: Offers flexibility in selecting language models based on complexity and cost.

- New Modules:
    - Search Module: Tailors searches to specific data sources for better results.
    - Fusion Module: Expands user queries into multiple forms and combines retrieved results for deeper insights.
    - Memory Module: Uses the model's internal knowledge to reduce unnecessary retrieval, retrieving only when needed.
    - Routing Module: Dynamically selects the best path for handling different types of queries.
    - Task Adapter Module: Adapts the system for different downstream tasks like summarization or translation.

TRADE-OFFS AND BEST PRACTICES

- Advanced techniques improve RAG accuracy but add complexity.
- Techniques like hybrid retrieval or reranking can increase computational costs and latency.
- Modular RAG offers flexibility but requires careful management of interfaces and module compatibility.
- Testing each module independently and as a whole is important to ensure system stability and performance.
- Trade-offs between performance, cost, and system complexity should be carefully assessed.