Optimizing RAG

This chapter covers

- Strategies for optimizing document extractions: OCR, and text-to-markdown conversion
- Using vector databases for storing/retrieving embeddings
- Advanced retrieval: re-ranking, hybrid search, query writing
- Advanced generation: prompt engineering, chain of thought, LLM selection
- Evaluating RAG applications with and without ground truth

In the previous chapter, you learned how to build your first RAG application. While this worked well for answering questions from text in a sample document, you noticed that the performance on answering questions from tables was not great. Sometimes the application answered a related question, other times it gave a wrong answer. In this chapter, we are going to learn how to optimize your RAG application for better performance through better document parsing and chunking.

Remember that an important component to RAG is the retrieval of relevant context, for responding to a user. Obtaining embedding vectors from text is an important step prior to retrieval, as it enables vector based lookups like cosine similarity, which we discussed. In the previous chapter for simplicity, we used OpenAI's default embedding model (text-embedding-ada-002) on text, but this model might not be ideal, depending on your use-case. In this chapter, we will discuss how to choose the right embedding model, and how a dedicated vector database can improve operational performance by speeding up the retrieval process.

There has been some recent research on retrieval strategies such as re-ranking, self-RAG, HyDE, and more that we will discuss. We will also discuss optimizing generation through prompt engineering, and choosing the LLM for generation. For this chapter, we will be using LlamaIndex, a popular open-source framework for building RAG applications.

Finally, choosing from multiple options requires measurement of performance. And your measurement is as good as your yardstick. We will discuss measurement metrics and how to evaluate RAG performance across relevant metrics.

3.1 Optimizing Document Parsing and Chunking

In Chapter 2, you saw that the PyMUPDF parser with fixed size chunking worked well for text, but when extracting information from tables, the table structure was not obvious. Of course, this is problematic since you are likely going to need to extract data from various embedded tables. Let's take a look at some different document parsing and chunking strategies to help overcome this issue and extract data from embedded tables.
First, let's modify the chunking strategy. Previously, we used fixed size chunking of 1024 characters. The issue with this however, is that this might lead to cases where important contents such as tables are broken up into 2 separate chunks. An easy resolution to this, is now chunking pages as a whole - and only breaking them up into smaller chunks if the threshold is exceeded. LlamaIndex allows this easily as below. Let's again use PyMUPDF, with chunking pages as a whole and breaking them up with a sentence splitter if the chunks exceed 2048 tokens.

Listing 3.1 PyMUPDF document parsing using LlamaIndex

```
from llama_index.core.schema import TextNode
from llama_index.core.node_parser import SentenceSplitter

file_path =
"/content/Q1-2023-Amazon-Earnings-Release.pdf"
doc = fitz.open(file_path) # Loading the file
#Next, we instantiate the sentence splitter, with a
maximum chunk size of 2048 tokens.
text_parser = SentenceSplitter(
    chunk_size=2048,
)
text_chunks = [] #Parsing pages
for doc_idx, page in enumerate(doc):
  page_text = page.get_text("text")
  cur_text_chunks = text_parser.split_text(page_text)
  text_chunks.extend(cur_text_chunks)
nodes = [] #Adding text chunks to LlamaIndex nodes
for idx, text_chunk in enumerate(text_chunks):
  node = TextNode(
```

```
        text=text_chunk,
    )
    nodes.append(node)
```

Since each page is less than 2048 tokens, here the number of nodes corresponds to the number of pages in the document, which is 15. Let's print the 9th page for example, and visualize it alongside the actual table in figure 3.1:

Consolidated Statements of Comprehensive Income (Loss)
(in millions)
(unaudited)

	Three Months Ended March 31,	
	2022	2023
Net income (loss)	$ (3,844)	$ 3,172
Other comprehensive income (loss):		
Foreign currency translation adjustments, net of tax of $(16) and $(10)	(333)	386
Net change in unrealized gains (losses) on available-for-sale debt securities:		
Unrealized gains (losses), net of tax of $1 and $(29)	(662)	95
Reclassification adjustment for losses (gains) included in "Other income (expense), net," net of tax of $0 and $(10)	6	33
Net unrealized gains (losses) on available-for-sale debt securities	(656)	128
Total other comprehensive income (loss)	(989)	514
Comprehensive income (loss)	$ (4,833)	$ 3,686

Figure 3.1 Page 9 Table From Amazon Q1 2023 Document

As you can see, this table contains text organized in a specific format. It is important to capture all the text, as well as the relative positions and header formats in order to accurately capture the table contents.
As you can see in figure 3.2, PyMUPDf is able to extract the text from the table, but it does not capture the structure of the table. We seem to lose information about what text reflects what aspects e.g. it is unclear what the comprehensive income loss values correspond to.

```
AMAZON.COM, INC.
Consolidated Statements of Comprehensive Income (Loss)
(in millions)
(unaudited)

Three Months Ended
March 31,

2022
2023
Net income (loss)
$
(3,844) $
3,172
Other comprehensive income (loss):
Foreign currency translation adjustments, net of tax of $(16) and $(10)

(333)
386
Net change in unrealized gains (losses) on available-for-sale debt securities:
Unrealized gains (losses), net of tax of $1 and $(29)

(662)
95
Reclassification adjustment for losses (gains) included in “Other income (expense),
net,” net of tax of $0 and $(10)

6
33
Net unrealized gains (losses) on available-for-sale debt securities

(656)
128
Total other comprehensive income (loss)

(989)
514
Comprehensive income (loss)
$
(4,833) $
3,686
```

Figure 3.2 extracted Figure 9 table using PyMUPDF

In the section below, we will look at advanced document parsing techniques like optical character recognition (OCR) and text to markdown conversion, to help identify the structure of tables.

3.1.1 Document Parsing Using OCR

Optical character recognition (OCR) is a way to convert images to text, widely used for data entry from bank statements, passports, invoices, and other suitable documents. OCR also enables the extraction of structured data from documents, containing information such as the location of content blocks, and text. Tesseract, originally developed by Hewlett Packard, is an open-source OCR engine that is widely considered one of the most accurate open source OCR engines. AWS Textract, is a closed source OCR engine that is widely used. The advantage of parsing document information from images using OCR, over text parsing is that:

1. Text parsing from PDFs can only recognize text, whereas OCR can recognize text from multiple formats, after converting to images
2. OCR tools like Textract enable the extraction of structured data from documents, and parsing into JSON format, containing information such as the location of content blocks, and text contained within the blocks.

Here's an example of using pytesseract, an open-source python wrapper for Google's Tesseract-OCR Engine for parsing the PDF. First we convert PDF pages into images, and next parse images into text using pytesseract.

Listing 3.2 Document Parsing With OCR

```
from PIL import Image #Importing dependencies
import pytesseract
import sys
from pdf2image import convert_from_path
import os
PDF_file =
"/content/Q1-2023-Amazon-Earnings-Release.pdf"
pages = convert_from_path(PDF_file)#Converting PDf to
images
i=8
filename = "page"+str(i)+".jpg"
pages[i].save(filename, 'JPEG')
outfile =  "page"+str(i)+"_text.txt"
f = open(outfile, "a")
text=
str(((pytesseract.image_to_string(Image.open(filename))
```

```
))#Loading an image for pytesseract to parse into a
string
text = text.replace('-\n', '')
f.write(text)
f.close()
print(text)
```

In figure 3.3, you can see the parsed pdf table using OCR.

```
AMAZON.COM, INC.
Consolidated Statements of Comprehensive Income (Loss)
(in millions)

(unaudited)
Three Months Ended
March 31,
2022 2023
Net income (loss) $ (3,844) $ 3,172
Other comprehensive income (loss):
Foreign currency translation adjustments, net of tax of $(16) and $(10) (333) 386
Net change in unrealized gains (losses) on available-for-sale debt securities:
Unrealized gains (losses), net of tax of $1 and $(29) (662) 95
Reclassification adjustment for losses (gains) included in “Other income (expense),
net,” net of tax of $0 and $(10) 6 33
Net unrealized gains (losses) on available-for-sale debt securities (656) 128
Total other comprehensive income (loss) (989) 514
Comprehensive income (loss) $ (4,833) $ 3,686
```

Figure 3.3 Parsed PDF Table Using OCR

3.1.2 Intelligent Document Parsing (IDP)

Both PDF to text and image to text approaches discussed above have their own limitations. PDF to text extracts all the text from the document, but this text might not be well structured, e.g. tables are notoriously hard to extract. On the other hand, OCR techniques can be tailored to extract and parse structured formats like receipts and cheques, but need to be customized based on document template. OCR can also be inaccurate. It can misread or skip letters, combine text incorrectly, and have trouble with handwritten text, symbols, and non-standard fonts.

Due to the growing popularity of RAG, there are multiple new PDF extractors that combine components like PDF to text extractors, Optical character recognition (OCR) engines like Tesseract, and even smaller LLMs for parsing text into structured formats to produce high fidelity renderings of PDF as LLM readable text, in a structured format. Structured formats include saving

metadata e.g. positions and relative headers, and also preserving structure in markdown formats. This is called intelligent document parsing (IDP).

As an example, here is a sample table image and the corresponding markdown below:

Month	Earnings	Savings
January	$500	$200
February	$1,000	$500
March	$6,000	$1,000
April	$10,000	$2,000

```
| Month    | Earnings | Savings |
|----------|----------|---------|
| January  | $500     | $200    |
| February | $1,000   | $500    |
| March    | $6,000   | $1,000  |
| April    | $10,000  | $2,000  |
```

Figure 3.4 Sample Table (top) and markdown text (below)

Newer intelligent document parsers like pymupdf4llm, unstructured.io, and LlamaParse (from LlamaIndex) include multiple state of the art techniques like OCR, extraction to markdown, and LLM based extraction. Let's look at using LlamaParse for document parsing to markdown that preserves document structure as below:

Listing 3.3 LlamaParse Intelligent Document Parsing

```
from llama_parse import LlamaParse
documents =
LlamaParse(result_type="markdown").load_data("/content/Q1-2023-Amazon-
Earnings-Release.pdf") #Configuring LlamaParse to convert text to
markdown, to keep doc structures
def get_page_nodes(docs, separator="\n---\n"): #Getting page texts
    """Split each document into page node, by separator."""
    nodes = [] #Adding chunks to llamaindex nodes
    for doc in docs:
        doc_chunks = doc.text.split(separator)
        for doc_chunk in doc_chunks:
            node = TextNode(
                text=doc_chunk,
```

```
            metadata=deepcopy(doc.metadata),
        )
        nodes.append(node)

    return nodes


nodes = get_page_nodes(documents)
```

Here, you can see the markdown returned, corresponding to page 9. You can see that the formatting seems to retain more structure. Headers are marked by '#', and columns are marked by | delimiters.

```
# AMAZON.COM, INC.

# Consolidated Statements of Comprehensive Income (Loss)

| |Three Months Ended March 31, 2022|Three Months Ended March 31, 2023|
|---|---|---|
|Net income (loss)|$ (3,844)|$ 3,172|
|Other comprehensive income (loss):| | |
|Foreign currency translation adjustments, net of tax of $(16) and $(10)|(333)|386|
|Net change in unrealized gains (losses) on available-for-sale debt securities:| | |
|Unrealized gains (losses), net of tax of $1 and $(29)|(662)|95|
|Reclassification adjustment for losses (gains) included in "Other income (expense), net," net of tax of $0 and $(10)|6|33|
|Net unrealized gains (losses) on available-for-sale debt securities|(656)|128|
|Total other comprehensive income (loss)|(989)|514|
|Comprehensive income (loss)|$ (4,833)|$ 3,686|
```

Markdown is a standard format to represent rich text formats. Using a standard online markdown visualization tool, you can see just how good of a representation it is of the actual table, shown below in figure 3.5.

AMAZON.COM, INC.

Consolidated Statements of Comprehensive Income (Loss)

	Three Months Ended March 31, 2022	Three Months Ended March 31, 2023
Net income (loss)	$ (3,844)	$ 3,172
Other comprehensive income (loss):		
Foreign currency translation adjustments, net of tax of $(16) and $(10)	(333)	386
Net change in unrealized gains (losses) on available-for-sale debt securities:		
Unrealized gains (losses), net of tax of $1 and $(29)	(662)	95
Reclassification adjustment for losses (gains) included in "Other income (expense), net," net of tax of $0 and $(10)	6	33
Net unrealized gains (losses) on available-for-sale debt securities	(656)	128
Total other comprehensive income (loss)	(989)	514
Comprehensive income (loss)	$ (4,833)	$ 3,686

Figure 3.5 Table parsed into markdown format using LlamaParse

3.1.3 Choosing a document parsing strategy

We've looked at three types of document extraction strategies - extracting text from documents, converting documents to images and extracting text using OCR, and intelligent document parsing. The pros and cons of each method are listed below:

Table 3.1 Document parsing strategies

Methodology	**Tools**	**Pros**	**Cons**
Extracting Text from documents	PyMUPDF, PyPDF, PDFMiner	Relatively mature, low latency	Does not capture complex structures well, e.g. tables, images, layouts
Converting to images and extracting text using OCR	Tesseract, Textract	Relatively mature, captures document structure and able to extract text from images	High latency, use-case dependent
Intelligent document parsing	LlamaParse, Unstructured.io, docsumo, Azure Doc Intelligence	Able to capture complex document structure, conversion to markdown	Less mature, lack of scalability currently, high latency

Which method you choose ultimately depends on the use case. If the information you wish to parse is stored in easily recognizable text, PDF parsers like PyMUPDF could be a good option. But if you wanted to parse text from images, you would need more advanced OCR or intelligent document parsers. We will discuss evaluating RAG applications and choosing the right document parser model for your application.
After document parsing and chunking, the next step in the RAG pipeline is embedding documents. We will discuss how to make embedding model choices in the coming section.

3.2 Embeddings and Vector Databases

At the heart of effective RAG systems lie two critical components: embeddings and vector databases. This section will explore how these technologies work together to dramatically improve the performance and efficiency of RAG applications. By converting information into dense numerical vectors, embeddings capture complex relationships and similarities that go beyond simple keyword matching. In the context of RAG, embeddings

enable the system to find the most relevant information from a vast knowledge base, even when the exact wording differs from the user's query. Vector databases, on the other hand, are the engines that make rapid retrieval of this embedded information possible at scale. They're designed to efficiently store, index, and search through millions or even billions of high-dimensional vectors. This capability is crucial for RAG systems that need to quickly sift through large amounts of data to find the most pertinent information for generating responses.

In the following subsections, we'll dive deep into the practical aspects of implementing embeddings and vector databases for RAG:

1. We'll explore how to choose the right embedding model for your specific needs, balancing factors like accuracy, computational resources, and cost.
2. You'll learn about various vector database options and how to integrate them into your RAG pipeline.
3. We'll walk through concrete code examples that demonstrate how to embed documents, store them in a vector database, and perform efficient similarity searches.

3.2.1 Choosing the right embedding model

While OpenAI's text-embedding-ada-002 (and other similar models) embedding models are easy to get started with and offer good quality embeddings, there are cons associated with them such as pay as you go embedding models. The ada v2 model costs $0.10/1 Million tokens. For context, the first Harry Potter book has approximately 100k tokens. Tokenizing this book is relatively cheap - just a cent. This sounds cheap, but now consider the case where you are processing a million records a day - quite regular for an ecommerce or social website that gets millions of daily interactions. If each record is roughly 1000 tokens, this would cost 0.1*1000 or 100$/day. If you process 10 million records a day, this comes to 1000$/day or 365,000$ a year which is now not small change anymore. In addition, since these embedding models are processing high volumes of daily data, latency is important. Anecdotally, these embedding API models have latencies of 250ms up to 30s, as well as outages, making them less reliable. Also, closed-source embedding model APIs can be deprecated or changed, meaning that you need to again embed historical documents, which can be additionally expensive.

Hosting your own embedding models can improve performance and reliability. The Massive Text Embedding Benchmark (MTEB) Leaderboard has a

constantly updated list of embedding models and their performance, as shown in figure 3.6.

Rank	Model	Model Size (Million Parameters)	Memory Usage (GB, fp32)	Average
1	gte-Qwen2-7B-instruct	7613	28.36	60.25
2	Linq-Embed-Mistral	7111	26.49	60.19
3	SFR-Embedding-2_R	7111	26.49	60.18
4	NV-Embed-v1	7851	29.25	59.36
5	SFR-Embedding-Mistral	7111	26.49	59
6	voyage-large-2-instruct			58.28
7	gte-large-en-v1.5	434	1.62	57.91
8	GritLM-7B	7242	26.98	57.41
9	TDTE			57.05
10	e5-mistral-7b-instruct	7111	26.49	56.89

Figure 3.6 MTEB Leaderboard On Huggingface

You can see the top ten models for document retrieval on the Leaderboard above. The things to look out for while choosing an embedding model is the average score (the rightmost column), as well as memory usage, which is closely related to model size. Usually there is a tradeoff between quality and resource requirements/latency. Larger models require more resources and have higher latency, whereas smaller models are low latency, but lower quality.

3.2.2 Vector Databases

A vector database makes it efficient to retrieve and rank contexts, based on user input embeddings, and document embeddings. In the previous chapter, we manually calculated the cosine similarities between each document embedding, and user input embedding - which was the question asked by the user; and chose the top-k relevant documents based on highest cosine

similarity. However, this is quite inefficient, and becomes time consuming as the number of documents increases.
Vector databases are specialized systems designed to store, index, and retrieve vector embeddings efficiently. These embeddings are numerical representations of data that capture semantic meaning, allowing for similarity-based searches. These rely on Approximate Nearest Neighbor (ANN) algorithms to locate the closest vectors, ensuring low latency in query responses. Examples include open-source options like Qdrant, Chroma, Milvus, Redis, Weaviate, pgvector, as well as closed-source platforms like Pinecone and Databricks Vector Search. Let's look at embedding the document chunks obtained in the previous section, using LlamaIndex, and adding to a vector database. We will make use of Qdrant vector DB in this example. First, let's load the required repositories, and instantiate the embedding model (text-embedding-3-small):

```
from llama_index.llms.openai import OpenAI
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core import VectorStoreIndex
from llama_index.core import Settings
from llama_index.vector_stores.pinecone import
PineconeVectorStore
embed_model =
OpenAIEmbedding(model="text-embedding-3-small") #A
llm = OpenAI(model="gpt-3.5-turbo-0125")
import qdrant_client
os.environ["QDRANT_API_KEY"] = getpass.getpass()
os.environ["QDRANT_URL"] = getpass.getpass()

Settings.llm = llm
Settings.embed_model = embed_model
```

 # Instantiating the embedding model

Next, let's import Qdrant, as well as creating an index.

```
client = qdrant_client.QdrantClient(
#you can use :memory: mode for fast and light-weight
experiments. It does not require to have Qdrant deployed
anywhere, but requires qdrant-client >= 1.1.1. Otherwise
set Qdrant instance address with:
```

```
    url=os.environ["QDRANT_URL"],
#otherwise set Qdrant instance with host and port:
host="localhost",port=6333


    api_key=os.environ["QDRANT_API_KEY"],
)

vector_store = QdrantVectorStore(client=client,
collection_name="01_Basic_RAG")
```

Next, we get all the node embeddings as below, and add them to the vector store.

```
for node in nodes:
    node_embedding = embed_model.get_text_embedding(
        node.get_content(metadata_mode="all")
    )
    node.embedding = node_embedding
vector_store.add(nodes)
```

Finally, we can query our Vector DB to retrieve the top-k relevant contexts to a query as below. In this example, top_k is 5 - or 5 retrieved contexts used for responding to the query:

```
from llama_index.core import VectorStoreIndex
from llama_index.core import StorageContext
index = VectorStoreIndex.from_vector_store(vector_store)
query_engine = index.as_query_engine(top_k = 5)
query = """What was the Comprehensive income (loss) for Amazon for the
Three Months Ended March 31, 2022?"""
response = query_engine.query(query)
```

In the above code snippet, the query_engine is making a call to an LLM (gpt-3.5-turbo-0125 here) - including the query and retrieved context as the prompt input. First, you import the necessary classes from llama_index.core. You create a VectorStoreIndex from an existing vector store. You set up a query engine configured to return the top 5 results. You define a specific query about Amazon's comprehensive income as a string. You execute the query using the query engine. Finally, you store the response in a variable, allowing you to efficiently retrieve information from a pre-indexed dataset based on similarity.

In comparing the response from LlamaParse vs PyMUPDF discussed in the section above on optimizing document parsing, we find both PyMUPDF parser and LlamaParse parsing resulted in the right answer ('The Comprehensive income (loss) for Amazon for the Three Months Ended March 31, 2022, was $(4,833) million.').

3.3 Optimizing Retrieval

In RAG, optimizing retrieval is a crucial step to ensure that the most relevant and informative contexts are provided to the language model. This section delves into various strategies and techniques designed to enhance the retrieval process, thereby improving the overall accuracy and efficiency of RAG applications. By refining how documents are retrieved, reordered, and processed, these methods aim to bridge the gap between user queries and the vast amount of information available in document collections. Below, we explore some of the key techniques for optimizing retrieval.

3.3.1 Reranking

Re-ranking is the process of refining and reordering the initial set of documents retrieved.This ensures that the most relevant documents are prioritized for the language model to use in generating responses. The importance of re-ranking in RAG stems from several key benefits:
Improved relevance: Re-ranking helps identify and prioritize the most relevant documents for a given query, ensuring that the language model has access to the most informative context for generating accurate responses.
Enhanced retrieval quality: By reordering and filtering documents, re-ranking places the most relevant information at the forefront, thereby improving the overall effectiveness of the RAG process.
Cost efficiency: By prioritizing the most relevant documents, re-ranking can potentially reduce the amount of irrelevant information processed by the language model, leading to cost savings in the RAG process
Through re-ranking retrieved contexts by relevance, this technique is used to improve the quality of generated outputs by making it more likely for the LLM to receive the most relevant context corresponding to the user query.
Let's use the LLM Reranker available in LlamaIndex to see how this works on our example document, using the same query as above.

Here, we import the LLMRerank postprocessor and configure it to process the top 5 retrieved documents, selecting the 3 most relevant. We set up our

query engine with a similarity_top_k of 5 and apply the LLMRerank postprocessor. When we execute our query about Amazon's comprehensive income, the engine first retrieves the 5 most similar documents based on vector similarity. Then, it uses the LLMRerank to reassess these documents, likely improving the relevance of the results. The final response we receive contains information from the 3 most pertinent documents, as determined by both vector similarity and LLM-based relevance assessment. This two-step process helps ensure we get more relevant information in response to our financial query.

Listing 3.4 LLM Reranker

```
from llama_index.core.postprocessor import LLMRerank
#Loading LLM re-ranker
query_engine = index.as_query_engine(
    similarity_top_k=5,
    node_postprocessors=[
        LLMRerank(
            choice_batch_size=5,
            top_n=3, #filtering top-5 after re-ranking

    ],
)
response = query_engine.query(
    "What was the Comprehensive income (loss) for Amazon
for the Three Months Ended March 31, 2022?",
)
```

Whether to use this or any other optimization strategy ultimately depends on whether or not there is a significant improvement in the metrics you care about. For example, if you have a test set of questions and answers, evaluating a variation of RAG with and without re-ranking will show you how much of a performance change there is. We will discuss RAG evaluation in depth - in Section 3.5.

Ultimately, evaluation results will tell you whether using this or any other retrieval strategy results in a significant improvement in the metrics you care about.

There are multiple reranking models such as cross-encoder models like BGE-Reranker, and multi-vector solutions like ColBERT. Choosing a re-ranker should be closely related to application requirements including latency, performance on evaluation, and generalization ability.

3.3.2 Hybrid Search

Let’s look at another popular retrieval strategy. Hybrid search combines the strengths of different search methods to improve the relevance and accuracy of retrieved information. It typically integrates vector embeddings search with keyword-based search to achieve better results. While embeddings based search has unlocked multiple use-cases, it is not without downsides, compared to typical keyword based search, that has been around for decades. In some use-cases it could be beneficial to combine simple keyword based search with vector based search - for example in cases where we require the extraction of precise fields (like the comprehensive income discussed above) and texts containing those specific keywords. The combination is often controlled by a parameter called "alpha," which determines the weight given to each search method.
Let’s look at an example of hybrid search using LlamaIndex and Qdrant. First, a QdrantVectorStore is initialized with enable_hybrid=True. This is crucial for enabling hybrid search capabilities. Next, Data (nodes) is added to the vector store. An index is created from the vector store, which will be used for querying.
A query engine is created from the index with specific parameters:
similarity_top_k = 5: This limits the results to the top 5 most similar matches.
vector_store_query_mode="hybrid": This explicitly sets the query mode to hybrid.
Finally, you can query the same document as above.
Here is the code:

```
vector_store = QdrantVectorStore(client=client,
collection_name="04_Hybrid_search",enable_hybrid=True)
#Needed for hybrid search
vector_store.add(nodes)
index = VectorStoreIndex.from_vector_store(vector_store)
query_engine_4 = index.as_query_engine(similarity_top_k =
5, vector_store_query_mode="hybrid")
query = """What was the Comprehensive income (loss) for
Amazon for the Three Months Ended March 31, 2022?"""
response = query_engine_4.query(query)
str(response)
```

3.3.3 Query Rewriting

Query rewriting is a process of transforming or expanding the original user query into multiple related queries or a more effective form. This optimization strategy aims to bridge the semantic gap between user queries and document content, ultimately enhancing the overall performance of RAG applications. Here are the 3 query rewriting strategies:

Routing: Use the query to identify downstream tools for further processing. An example could be to run the query against a summarization index vs a regular index for documents.

Query-Rewriting: Transform the query to enhance retrieval quality.

Sub-Queries: Decompose queries into multiple sub-queries.

HyDE, or Hypothetical Document Embeddings, is a specific query rewriting strategy that aims to bridge the semantic gap between queries and documents. The process works as follows:

1. Generate a hypothetical document: Using the original query, an AI model creates a hypothetical document that would ideally answer the query.
2. Embed the hypothetical document: The generated document is then embedded using the same embedding model used for the actual documents in the database.
3. Retrieve similar documents: The embedding of the hypothetical document is used to find similar actual documents in the database.

The advantage of HyDE is that it can potentially capture more nuanced semantic relationships between the query and the documents, leading to more relevant retrievals. However, it's important to note that these techniques should be applied judiciously.Blindly modifying user queries can sometimes lead to unintended consequences, such as diluting the original intent or introducing unnecessary complexity. The decision to implement query rewriting techniques like HyDE should be based on the specific needs of the application and the nature of the document collection. If the queries are complex or the documents contain diverse language and semantics, HyDE can enhance retrieval accuracy and relevance. However, it's essential to weigh the computational overhead and performance implications against the potential benefits to ensure that the implementation aligns with user needs and expectations.

3.3.3 Other strategies

With the boom of RAG use-cases, there have been many other innovative strategies. In Self-RAG, the authors develop a way for a fine-tuned LM (Llama2–7B and 13B) to output special tokens [Retrieval], [No Retrieval], [Relevant], [Irrelevant], [No support / Contradictory], [Partially supported], [Utility], etc. appended to LLM generations to decide whether or not a context is relevant/irrelevant, the LLM generated text from the context is supported or not, and the utility of the generation. Based on the tokens, retrieval is repeated until all relevant documents are found.
S2A — released by META tries to solve the problem of spurious context with a little more finesse. Instead of marking contexts as relevant/irrelevant as in self-RAG or re-ranking, in S2A context is regenerated to remove noise and ensure relevant information remains. This approach aims to improve the quality and accuracy of LLM responses by focusing on the most relevant parts of the input context. The disadvantages of S2A include computational overhead, and potential for information loss. S2A works through a specific instruction that requires the LLM to regenerate the context, extracting the part that is beneficial for providing relevant context for a given query as below.

> Given the following text by a user, extract the part that is unbiased and not their opinion, so that using that text alone would be good context for providing an unbiased answer to the question portion of the text.
>
> Please include the actual question or query that the user is asking. Separate this into two categories labeled with "Unbiased text context (includes all content except user's bias):" and "Question/Query (does not include user bias/preference):".
>
> Text by User: [ORIGINAL INPUT PROMPT]

Fig. 3.7 S2A Implementation

Corrective RAG (CRAG) aims to be an all encompassing strategy for retrieval, including external knowledge searches. In CRAG, when a query is received, relevant documents are retrieved from a knowledge base, where these documents undergo rigorous evaluation. A retrieval evaluator assesses their relevance, factuality, and quality, filtering out low-quality or irrelevant information. This process may involve real-time corrections, factual accuracy checks, and in some implementations, neural clustering of semantically related passages. The resulting high-quality, refined information is then used

to guide the language model's response generation, leading to more accurate, contextually appropriate, and reliable outputs.

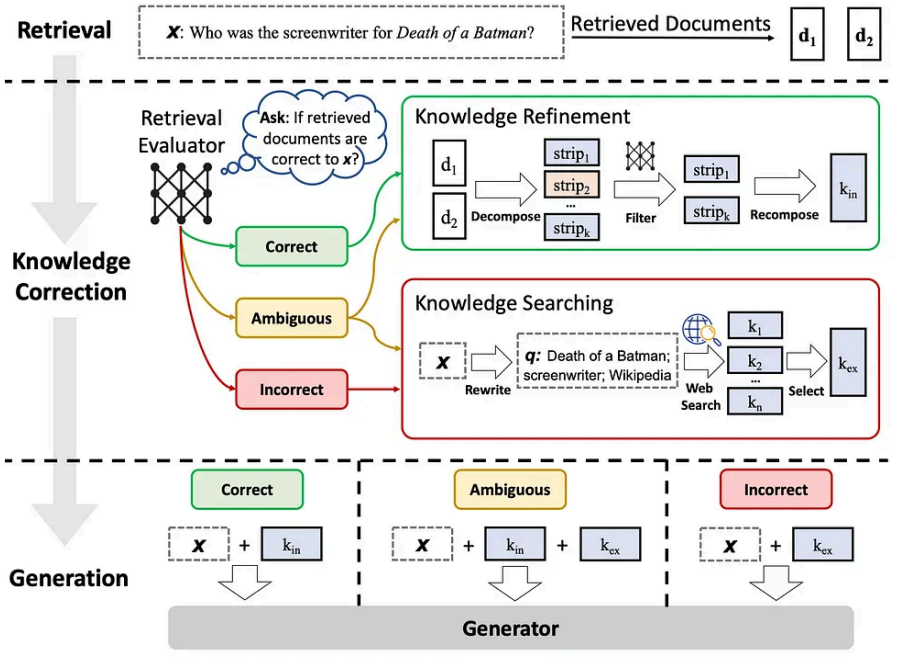

Fig. 3.8 Corrective RAG

These are not a comprehensive list of all retrieval techniques, but the more popular ones. As RAG gains popularity, it is likely more such retrieval techniques will emerge.

3.4 Optimizing Generation

After retrieving relevant context, the next crucial step in a RAG system is to generate high-quality, accurate responses. This section explores various techniques to optimize the generation process, enhancing the overall performance and reliability of your RAG application.

We'll delve into prompt engineering strategies, discuss the importance of system instructions, and explore methods like few-shot prompting and Chain of Thought (CoT) reasoning. These techniques not only improve the quality of generated responses but also help in maintaining consistency, adhering to specific output formats, and handling complex queries that require multi-step reasoning.

By implementing these optimization techniques, you can enhance the capabilities of your RAG system, ensuring it provides more accurate,

relevant, and well-structured responses to user queries. Let's begin by examining the nuances of prompt engineering and how it can be leveraged to improve your RAG application's output.

3.4.1 Prompt Engineering

We briefly discussed incorporating retrieved contexts into the LLM prompt for text generation in chapters. 1 and 2. It looks something like below - where the two placeholder fields are the user provided query, and retrieved context using embeddings or keyword search algorithms discussed earlier:

```
Answer the query based on the context provided.
    Query:
    ```{query}.```

    Context:
    ```{context}```
```

However, there are several ways this could be optimized. First, in most RAG cases we would like the user to be returned answers specific to the context. If a user is asking a question about something unrelated (e.g. asking an airline agent RAG application for restaurant recommendations), the results could be misused, or the company could be held accountable for wrong information. It is a good practice to add some instructions in the prompt for this such as return “Sorry I cannot answer that" if the context is not relevant to the user query like below:

```
Answer the query based on the context provided. If no
relevant context is found, answer with “Sorry I cannot
answer that”.
    Query:
    ```{query}.```

    Context:
    ```{context}```
```

Here’s how to change the default prompt in the LlamaIndex query engine:

```
qa_prompt_tmpl_str = (
    Answer the query based on the context provided. If no
relevant context is found, answer with “Sorry I cannot
answer that”
)
qa_prompt_tmpl = PromptTemplate(qa_prompt_tmpl_str)
```

```
query_engine.update_prompts(
    {"response_synthesizer:text_qa_template":
qa_prompt_tmpl}
)
```

Another good practice is to add system instructions. Typically, system instructions contain specific information to help guide and add guardrails to the application such as requiring responses to not contain profanity. An example of a system instruction is:

```
system_instructions = “You are a helpful financial
analyst, and an expert in extracting financial
information from documents. Your goal is to give a
response to the user query, based on the relevant
context. If no context is provided, respond with ‘Sorry I
cannot answer that’. Make sure to follow the following
instructions in addition:
   1. Do not provide any general information
   2. Do not respond with profanity ”
```

Providing examples of inputs and outputs is a good practice, especially if outputs are expected in a certain format, such as json. Few-shot prompting is a way to ensure outputs adhere to these specific requirements. You can add that to the above prompt as below. Make sure to use delimiters like ``` where it makes sense, to make instructions clear.

```
system_instructions = “You are a helpful financial
analyst, and an expert in extracting financial
information from documents. Your goal is to give a
response to the user query, based on the relevant
context. YOur output should be json formatted, with the
json key being the user query and the value being the
numerical dollar amount, like {query: value}. Here is an
example:
```
```
Example Input:
“What was Amazon’s net sales in Q1 2023?”
Example Output:
{“Amazon net sales in Q1 2023”: “127358000000”}
```
```

If no context is provided, respond with ‘Sorry I cannot
answer that’as the value for the json key.
```

```
Make sure to follow the following instructions in
addition:
   3. Do not provide any general information
   4. Do not respond with profanity ”
```

Another common pattern is to use Chain Of Thought (CoT) to provide higher quality answers. Chain of thought prompting is a technique in prompting LLMs where the LLM is guided to break down its reasoning process into a series of logical steps. This approach encourages the model to "think aloud" and show its work, rather than just providing a final answer. Chain of thought prompting is particularly effective for tasks that require multi-step reasoning, such as mathematical problem-solving, logical deductions, or complex decision-making processes.

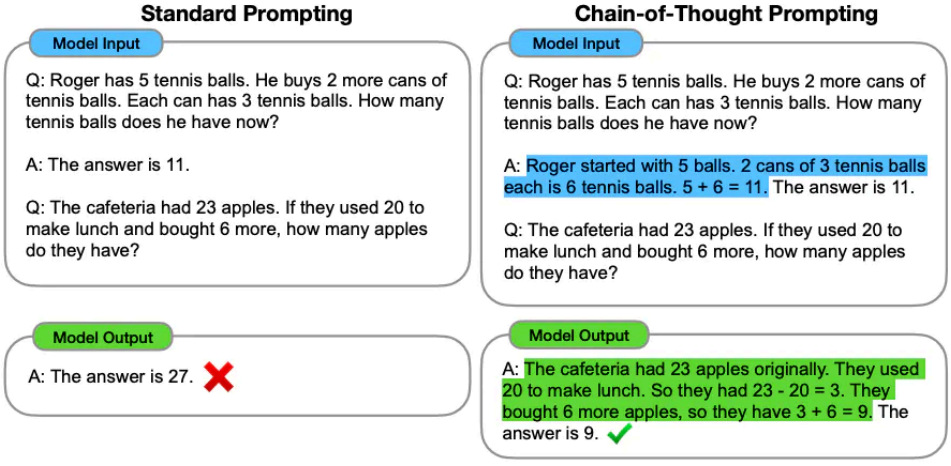

Fig. 3.9 Chain Of Thought Prompting For Better Outputs

As you can see above, asking the LLM to think step by step, and only then output an answer, works better for complex applications. This might be helpful, or even essential if you are combining multiple pieces of information for your RAG application.

3.4.2 Choosing Your LLM

When implementing a Retrieval Augmented Generation (RAG) system, selecting the right Language Model (LLM) is crucial for optimal performance. The choice between open-source and closed-source models can significantly impact your RAG application's effectiveness, cost, and scalability.

Closed-Source Models for RAG

Popular closed-source models like OpenAI's GPT series, Anthropic's Claude, and Google's Gemini offer high performance and are often preferred for RAG applications due to their advanced capabilities. These models excel at understanding context and generating coherent responses, which is essential for effective RAG implementations.

Advantages for RAG:

- Minimal infrastructure setup required
- High-quality context understanding and response generation
- Regular updates and improvements

Considerations:

- Usage-based pricing can become expensive for high-volume RAG applications
- Limited customization options

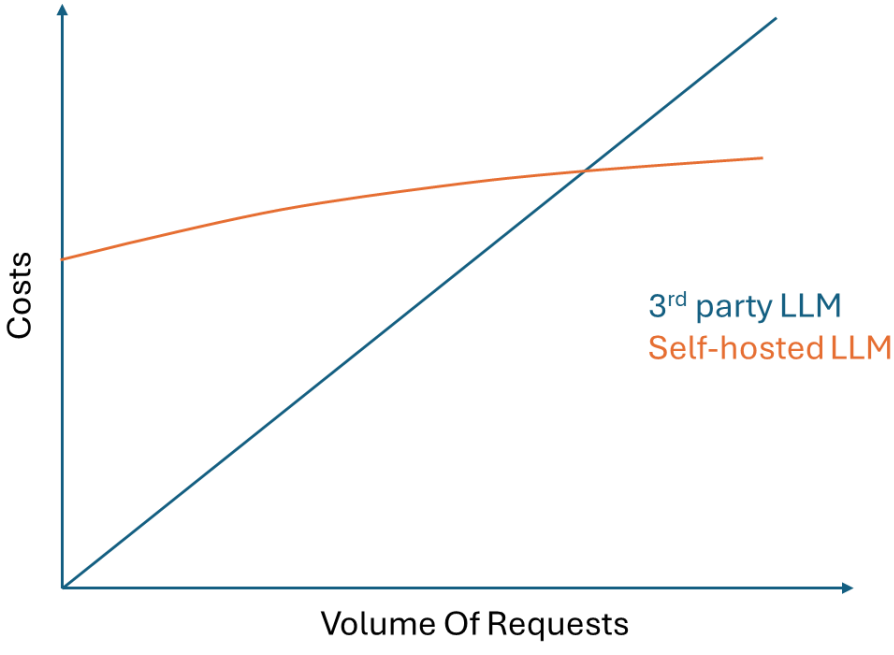

Figure 3.10 Costs for self-hosted vs 3rd party hosted LLMs vs request volume

Open-Source Models for RAG

Open-source models such as Meta's Llama series, Mistral, Falcon, and Google's Gemma are increasingly viable options for RAG systems. While traditionally less performant than closed-source alternatives, recent advancements have narrowed this gap.

Advantages for RAG:

- Customizable through fine-tuning for specific domains or tasks
- Cost-effective for high-volume applications
- Full control over the model and data

Considerations:

- Requires more infrastructure and technical expertise to deploy
- May need additional optimization for RAG-specific tasks

Optimizing Open-Source Models for RAG

Recent progress in model optimization techniques has made open-source models more attractive for RAG applications:

- Quantization: This technique reduces model size and memory requirements, making it feasible to run RAG systems on consumer-grade hardware. For example, an 8-bit quantized model requires 4x less memory than its 32-bit counterpart.
- Fine-tuning: Open-source models can be fine-tuned on domain-specific data, potentially improving RAG performance for specialized applications.
- Hugging Face Integration: LlamaIndex provides wrappers for Hugging Face models, making it easier to integrate various open-source LLMs into your RAG pipeline.

Choosing the Right Model for Your RAG Application

When selecting an LLM for your RAG system, consider:

1. Application Scale: For smaller-scale or prototype RAG applications, closed-source models may be more convenient. For large-scale deployments, open-source models could be more cost-effective.
2. Domain Specificity: If your RAG application requires domain-specific knowledge, an open-source model that can be fine-tuned might be preferable.
3. Infrastructure Capabilities: Assess your team's ability to manage and optimize open-source models for RAG tasks.
4. Cost Projections: Estimate long-term costs based on expected usage and compare closed-source API fees with the infrastructure costs of self-hosting.
5. Performance Requirements: Evaluate the model's ability to understand and generate responses based on retrieved context, which is crucial for RAG applications.
6. Privacy requirements: Certain sensitive industries like Healthcare and financial domains with access to sensitive customer data might have additional restrictions that require on-premise hosting, in which case

self-hosted open-source models or managed on-premises hosting of closed-source models are mandatory.

By carefully considering these factors, you can select the most suitable LLM for your RAG application, balancing performance, cost, and scalability to meet your specific needs.

3.5 Evaluating Your RAG Application

As we've explored various techniques to optimize retrieval and generation in RAG systems, it's crucial to understand how these optimizations impact the overall performance of your application. Evaluation is the key to measuring this impact and ensuring that your RAG system is delivering high-quality, accurate, and relevant responses to user queries.

Getting a good sense of how well your RAG application is going to perform, ahead of serving to customers is critical. You want to make sure that the experience is smooth, and customers are wowed. For this, it is critical to develop a set of metrics for evaluating your application.

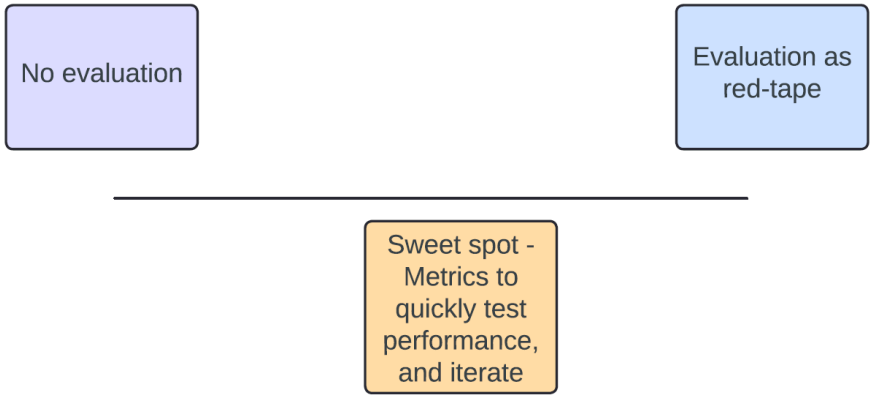

Figure 3.11 Evaluation Scale

Evaluating a RAG application is not just about assessing its final output; it's about understanding how each component of the system contributes to its overall effectiveness. This process allows you to:

- Identify bottlenecks in your retrieval or generation pipeline
- Measure the impact of different optimization techniques
- Ensure that your system is providing accurate and contextually relevant information

- Detect and address potential issues before they affect end-users
- Make data-driven decisions on further optimizations or trade-offs

However, it's important to strike a balance between thorough evaluation and rapid development. Over-emphasis on evaluation can slow down the deployment process (the right side of Fig. 3.6), while insufficient evaluation can lead to poor user experiences and potential misinformation (left of Fig. 3.6).

In this section, we'll explore various approaches to evaluating RAG applications, focusing on both output quality and component-level performance. We'll discuss metrics that are particularly relevant to RAG systems, such as retrieval accuracy, answer relevance, and faithfulness to the retrieved context. Additionally, we'll examine the challenges of evaluating LLM-based systems where traditional ground truth comparisons may not always be applicable.

By the end of this section, you'll have a comprehensive understanding of how to effectively evaluate your RAG application.

3.5.1 Ground Truth Evaluation Metrics

Depending on the language task — there are many standard evaluation metrics. For extractive tasks like extracting the right answer from the context, for a question — exact match and F1 are used for comparing the string output by the model and the “gold standard” answer.

For example, the gold standard answer for the question *“To whom did the Virgin Mary allegedly appear in 1858 in Lourdes, France?”*

with context:

“Architecturally, the school has a Catholic character. Atop the Main Building’s gold dome is a golden statue of the Virgin Mary. Immediately in front of the Main Building and facing it, is a copper statue of Christ with arms upraised with the legend “Venite Ad Me Omnes”. Next to the Main Building is the Basilica of the Sacred Heart. Immediately behind the basilica is the Grotto, a Marian place of prayer and reflection. It is a replica of the grotto at Lourdes, France where the Virgin Mary reputedly appeared to Saint Bernadette Soubirous in 1858. At the end of the main drive (and in a direct line that connects through 3 statues and the Gold Dome), is a simple, modern stone statue of Mary.”

Is: *“Saint Bernadette Soubirous.”*

Let’s say the answer given by the LLM based application is “*to Saint Bernadette Soubirous*”.

From an exact match metric, the score of this answer would either be 1 or 0–in this case it is 0 because the model output didn't exactly match the gold standard answer.
Let's take another common metric, F1 score. F1 = 2*precision*recall/(precision+recall) where precision is the ratio of the number of words shared to the total number of words in the prediction. Recall is the ratio of the number of words shared to the total number of words in the gold standard. This would have an F1 score of 0.75.
For generative tasks like summarization, more advanced evaluation methods are necessary while computing similarity scores between abstractive, longer pieces of text. The ROUGE score measures the similarity between overlapping n-grams. The ROUGE-1 amounts to the overlap of unigrams — and is similar to the individual word scores discussed for Q&A above, whereas ROUGE-2 evaluates the overlap between bigrams in the reference and predicted summaries. The ROUGE-L metric is slightly different — where it finds the longest common sequence between reference and prediction. ROUGE-L takes into account sentence-level structure similarity naturally and identifies longest co-occurring in sequence n-grams automatically. The BLEU score is another metric to evaluate long outputs — typically useful for tasks like language translation. It's been shown that BLEU scores correlate well to human translation quality assessments.
In addition to the above, LLM based ground truth metrics are becoming increasingly relevant. The idea is to design a prompt specifically for judging if an answer is correct or not (typically a 1/0 label), while also providing the LLM the ground truth answer. The usefulness of this metric is that this is a reasonably simple task for modern LLMs, and especially useful for natural language answers such as summaries, where traditional metrics like F1 score or Rouge score, are more weighted towards keyword similarity and not semantic similarity. If LLM based ground truth comparisons are too expensive or high latency, an alternative is to embed both the ground truth result and LLM generated response, and compare these embeddings using vector similarity metrics like cosine similarity (that we discuss in Ch. 2).

3.5.2 Evaluation Without Ground-Truth

In a majority of LLM applications I have encountered ground truths were not present. For example, let's say you want a RAG application to summarize documents. You might not have pre-existing summaries to compare with and so this makes it non-trivial to judge LLM outputs. There are several emerging patterns to judge LLM outputs in the absence of ground truths.

Judging the quality of outputs from LLM applications can be highly use-case dependent. Let's consider an application that creates LinkedIn posts. If the goal is to automate content creation for users and create posts in a similar voice as the customer has done previously, good quality content could be judged based on how similar (or not) this content is to previous content. You might also check for hallucinations. For example, if the user wishes to write a post wishing their user base "A happy new year," and the post returned contains the wrong year (e.g. 2022 instead of 2025), this should be rated a lower quality.

There needs to be some creativity involved in developing specific metrics for evaluation without ground truth. An emerging pattern here is to create LLM judge prompts specific to use-case. In addition to including specific judging criteria, it is a good practice to include human labeled scores along with respective inputs as few shot additions. Here is an example judge prompt:

```
Prompt = f""" You are an expert evaluator of LinkedIn
marketing content. Judge the outputs returned from their
respective inputs based on the following criteria:

Judge criteria:
5 - excellent because…
4 - good enough because…
3 - average, needs some tweaking bcause…
2 - not good, because…
1 - generation failure, because…

Here is a sample input containing model input, output,
and your response:

## Inputs:
----------
Model input:
…
Model output:
…
----------
## Outputs:
score:...
"""
```

There are also initiatives to develop RAG specific metrics. For example, the RAG Triad is an evaluation framework to assess the reliability and contextual

integrity of Language Model (LLM) responses, which consists of three key metrics:

1. **Context Relevance:** This metric evaluates how well the retrieved context aligns with the user's query
2. **Faithfulness:** This assessment checks if the LLM's response is accurately based on the provided context
3. **Answer Relevance**: This metric examines how effectively the generated response addresses the original user query

3.5.3 Evaluating Our Financial Q&A RAG Application

Now that we've explored various RAG optimization techniques, including document parsing, advanced retrieval methods, prompt engineering, LLM selection, and evaluation strategies, let's put these concepts into practice. We'll examine the performance of our RAG application using four different strategies, each incorporating some of the optimization techniques we've discussed. This comparison will help illustrate how different approaches can impact the overall effectiveness of a RAG system.

Let’s take a look at the performance of our RAG application, with 4 RAG strategies:

1. Simple RAG with PyMUPDF document parsing (top-k = 5)
2. RAG with LlamaParse LLM based document extraction (top-k = 5)
3. RAG with LLM re-ranking (top-k = 3)
4. RAG with hybrid search (top-k = 5)

For this, we are going to make use of a sample of 57 questions and labeled answers, or 57 ground truths. For judging performance, we will look at ground truth evaluation, using an LLM. These ground truths were generated by the Claude 3.5 Sonnet LLM - after providing the PDF as context. Generating high-quality ground truths for RAG evaluation is crucial and can be accomplished through LLM-generated or hand-labeled methods, each with its own advantages. LLM-generated ground truths, created by feeding documents into advanced models like Claude 3.5, offer efficiency and consistency but may lack nuance. Hand-generated labels, crafted by domain experts, provide high-quality, nuanced questions and answers but are time-consuming and potentially expensive. A hybrid approach, combining LLM generation with human refinement, often yields the best results by balancing efficiency and accuracy.

The system template below defines the role of the judge - to respond with 1/0 depending on whether the reference answer and generated answer are the

same. The prompt template takes 3 inputs, the user query, reference answer, and generated answer. Next, we define the functions required to run the above prompt. For the evaluation of LLM, we will use GPT-4.
Let's define a system template and user template as below:

```
CORRECTNESS_SYS_TMPL = """
You are an expert evaluation system for a financial
document q&a tool.

You are given the following information:
- a user query,
- a reference answer, and
- a generated answer.

Your job is to judge the correctness of the generated
answer.
Output a score that represents if the answer is correct
(1) or wrong (0).

If the generated answer is basically the same (e.g. if
the reference answer is 54,330 million whereas the
generated answer is 54 billion,
give a score of 1. Only give 0 if the answer is
completely wrong.)

But if the answer is completely wrong (e.g. reference
answer is 21 million, but generated answer is 54 billion,
give 0.)

You must return your response in a line with only the
score.
Do not return answers in any other format.
On a separate line provide your reasoning for the score
as well.

"""

CORRECTNESS_USER_TMPL = """
## User Query
{query}
```

```
## Reference Answer
{reference_answer}

## Generated Answer
{generated_answer}
"""
```

Now let's incorporate this template into a pipeline for running evaluations.

```
llm = OpenAI(model="gpt-4") #Setting the Eval LLM
eval_chat_template = ChatPromptTemplate( # Instantiating
the chat template
    message_templates=[
        ChatMessage(role=MessageRole.SYSTEM,
content=CORRECTNESS_SYS_TMPL),
        ChatMessage(role=MessageRole.USER,
content=CORRECTNESS_USER_TMPL),
    ]
)
from llama_index.llms.openai import OpenAI


def run_correctness_eval( # Eval function
    query_str: str,
    reference_answer: str,
    generated_answer: str,
    llm: OpenAI,
    threshold: float = 1.0,
) -> Dict:
    """Run correctness eval."""
    fmt_messages = eval_chat_template.format_messages(
        llm=llm,
        query=query_str,
        reference_answer=reference_answer,
        generated_answer=generated_answer,
    )
    chat_response = llm.chat(fmt_messages)
    raw_output = chat_response.message.content

    # Extract from response
    score_str, reasoning_str = raw_output.split("\n", 1)
```

```
    score = float(score_str)
    reasoning = reasoning_str.lstrip("\n")

    return {"passing": score >= threshold, "score":
score, "reason": reasoning}
```

Let's now look at an example, running the logic with a sample query:

```
query = data_eval[0]['question']
reference_answer = data_eval[0]['answer']
generated_answer = str(query_engine1.query(query))

eval_results = run_correctness_eval(
    query_str, reference_answer, generated_answer,
llm=llm, threshold=4.0
)
display(eval_results)
```

The evaluation response corresponding to the question "What was Amazon's operating cash flow for the trailing twelve months (TTM) in Q1 2023?" is:
{ 'score': 0.0,
 'reason': "The generated answer is incorrect because it does not match the reference answer. The reference answer states that Amazon's operating cash flow for the trailing twelve months (TTM) in Q1 2023 was $54,330 million, but the generated answer states it was $21,446 million."}
As you can see, the LLM response is returned as a json object with 2 keys: score, and reason. We can now run this over the 3 models discussed, and parse the output jsons to get the mean scores after running through all the examples:

```
# Instantiating all the scores and reasons
score_1=[]
reason_1=[]
ans_1=[]
score_2=[]
reason_2=[]
ans_2=[]
score_3=[]
reason_3=[]
ans_3=[]
score_4=[]
```

```
reason_4=[]
ans_4=[]


#Looping over all
score_1=[]
reason_1=[]
ans_1=[]
score_2=[]
reason_2=[]
ans_2=[]
score_3=[]
reason_3=[]
ans_3=[]
score_4=[]
reason_4=[]
ans_4=[]


for i in range(0,len(data_eval)):
    query = data_eval[i]['question']
    reference_answer = data_eval[i]['answer']
    generated_answer = str(query_engine_1.query(query)) #
Basic RAG
    ans_1.append(generated_answer)

    eval_results = run_correctness_eval(
        query, reference_answer, generated_answer,
llm=llm, threshold=1.0
    )

    score_1.append(eval_results['score'])
    reason_1.append(eval_results['reason'])

    generated_answer = str(query_engine_2.query(query))
#llamaparse extractor
    ans_2.append(generated_answer)

    eval_results = run_correctness_eval(
```

```
        query, reference_answer, generated_answer,
llm=llm, threshold=1.0
    )

    score_2.append(eval_results['score'])
    reason_2.append(eval_results['reason'])


    generated_answer = str(query_engine_3.query(query))
#re-ranker
    ans_3.append(generated_answer)

    eval_results = run_correctness_eval(
        query, reference_answer, generated_answer,
llm=llm, threshold=1.0
    )

    score_3.append(eval_results['score'])
    reason_3.append(eval_results['reason'])

    generated_answer = str(query_engine_4.query(query))
#Hybrid search
    ans_4.append(generated_answer)

    eval_results = run_correctness_eval(
        query, reference_answer, generated_answer,
llm=llm, threshold=1.0
    )

    score_4.append(eval_results['score'])
    reason_4.append(eval_results['reason'])
    print(i)
```

We find that the evaluation scores are 0.8, 0.74, 0.77, 0.89 respectively corresponding to the four models. The hybrid search model performs the best. It is also interesting that the basic PyMUPDF extraction model performs better, without re-ranking. Note that re-ranking considered here is only for top-3 retrieved contexts, whereas the LlamaParse without re-ranking retrieves all 5 contexts. Also keep in mind that this is just for one PDF

document. In order to comprehensively evaluate our app and understand which approach works best, we would do evaluations over multiple representative documents.

Now, let's look at evaluating our applications in the absence of ground truths, using the three RAG triad metrics defined above, using deepeval.

```
# importing dependencies

from deepeval.metrics import FaithfulnessMetric
from deepeval.test_case import LLMTestCase
from deepeval.metrics import ContextualRelevancyMetric
from deepeval.metrics import AnswerRelevancyMetric
import nest_asyncio
nest_asyncio.apply()
import os
import pandas as pd
df=pd.read_csv('/content/eval_ch3_3.csv')

#defining the 3 metrics

def
get_faithfulness_metric(user_query,final_answer,formatted_top_r
esults):
  metric = FaithfulnessMetric(
  threshold=0.7,
  model="gpt-4o",
  include_reason=True
  )
  test_case = LLMTestCase(
  input= user_query,
  actual_output=final_answer,
  retrieval_context=formatted_top_results
  )
  metric.measure(test_case)
  score = metric.score
  reason = metric.reason
  return score, reason

def
get_context_relevancy_metric(user_query,final_answer,formatted_
top_results):

  metric = ContextualRelevancyMetric(
```

```
  threshold=0.7,
  model="gpt-4o",
  include_reason=True
  )
  test_case = LLMTestCase(
  input= user_query,
  actual_output=final_answer,
  retrieval_context=formatted_top_results
  )
  metric.measure(test_case)
  score = metric.score
  reason = metric.reason
  return score, reason

def get_answer_relevancy_metric(user_query,final_answer):
  metric = AnswerRelevancyMetric(
  threshold=0.7,
  model="gpt-4o",
  include_reason=True
  )
  test_case = LLMTestCase(
  input= user_query,
  actual_output=final_answer,
  # retrieval_context=formatted_top_results_hypoth
  )
  metric.measure(test_case)
  score = metric.score
  reason = metric.reason
  return score, reason
```

Next, lets run a few examples in a loop and get the 3 metrics:

```
f_1, f_2, f_3, f_4 = [], [], [], []
cr_1, cr_2, cr_3, cr_4 = [], [], [], []
ar_1, ar_2, ar_3, ar_4 = [], [], [], []
f_1_reason, f_2_reason, f_3_reason, f_4_reason = [], [], [], []
cr_1_reason, cr_2_reason, cr_3_reason, cr_4_reason = [], [],
[], []
ar_1_reason, ar_2_reason, ar_3_reason, ar_4_reason = [], [],
[], []

for i in range(0,6):
  for j in range(1, 5):  # Loop for answers 1 to 4
    ans_key = f'ans_{j}'
```

```
    context_key = f'context_{j}'

    f_metric = get_faithfulness_metric(df.iloc[i]['question'],
df.iloc[i][ans_key], [df.iloc[i][context_key]])
    cr_metric =
get_context_relevancy_metric(df.iloc[i]['question'],
df.iloc[i][ans_key], [df.iloc[i][context_key]])
    ar_metric =
get_answer_relevancy_metric(df.iloc[i]['question'],
df.iloc[i][ans_key])

    globals()[f'f_{j}'].append(f_metric[0])
    globals()[f'cr_{j}'].append(cr_metric[0])
    globals()[f'ar_{j}'].append(ar_metric[0])
    globals()[f'f_{j}_reason'].append(f_metric[1])
    globals()[f'cr_{j}_reason'].append(cr_metric[1])
    globals()[f'ar_{j}_reason'].append(ar_metric[1])
  print(i)
#comparing faithfulness scores
print(np.mean(f_1), np.mean(f_2), np.mean(f_3), np.mean(f_4))
#1.0 0.8333333333333334 1.0 1.0

#comparing context relevancy scores
print(np.mean(cr_1), np.mean(cr_2),np.mean(cr_3),
np.mean(cr_4))
#1.0 1.0 1.0 1.0

#comparing answer relevancy scores
print(np.mean(ar_1), np.mean(ar_2), np.mean(ar_3),
np.mean(ar_4))
1.0 1.0 1.0 1.0
```

As you can see above, the scores returned are 1.0s overall. Only the faithfulness score for the 2nd RAG model has an average score of 0.833. Let's investigate this in more detail and look at the outputs:

['The score is 1.00 because there are no contradictions, indicating the actual output is perfectly aligned with the retrieval context. Great job!',

'The score is 1.00 because there are no contradictions. Great job on maintaining perfect alignment with the retrieval context!',
'The score is 1.00 because there are no contradictions. Excellent job maintaining faithfulness to the retrieval context!',
'The score is 0.00 because the actual output claims the operating income for Amazon Web Services (AWS) in Q1 2023 was $5.1 billion, which contradicts the retrieval context stating it was $5.123 billion.',
'The score is 1.00 because there are no contradictions. Great job maintaining accuracy and faithfulness to the retrieval context!',
'The score is 1.00 because there are no contradictions. Great job!']

As you can see from above, all scores are 1.0 but there's only one score of 0.0 for a slight rounding difference. This shows the issues of blindly trusting general non-ground truth RAG metrics (or LLM metrics for that matter). It is likely that evaluation metrics need to be more customized for the use-case.

In the next chapter, we will learn how to scale our application and deploy it for multiple customers to access.

Summary

- Document parsing can be improved using OCR, converting to markdown, and specialized AI-based extractors like LlamaParse.
- Vector databases like Qdrant efficiently store and retrieve embeddings, improving performance for large document collections.
- Advanced retrieval strategies include reranking, hybrid search (combining vector and keyword search), and query rewriting techniques like HyDE.
- Generation can be optimized through prompt engineering, including system instructions and chain-of-thought prompting.
- The choice between open-source and closed-source LLMs depends on factors like performance needs, cost considerations, and infrastructure requirements.
- Evaluating RAG applications involves both ground truth metrics (when available) and techniques for evaluation without ground truth, such as LLM-based judging.
- Be careful while using non-ground truth metrics out of the box, as these could yield irrelevant results. More than likely, these metrics need to be customized based on the use-case.