International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Retrieval-Augmented Generation (RAG): A Comparative Review of Architectures, Evaluation, and Optimization

Shantanu Kumar[1], Sukhveer Singh[2], Dr. Vishal Shrivastava[3], Dr. Akhil Pandey[4]

[1,2]Student(B.Tech), Artificial Intelligence and Data Science, Arya College of Engineering & I.T
[3]Professor, Artificial Intelligence and Data Science, Arya College of Engineering & I.T
[4]Head of Department (HOD), Computer Science, Arya College of Engineering & I.T

ABSTRACT

Large Language Models (LLMs) are powerful, but they are limited by knowledge that is frozen at training time, a tendency to hallucinate when they do not know an answer, and the high cost of fine-tuning them for every new document. Retrieval-Augmented Generation (RAG) addresses these limitations by retrieving relevant knowledge from an external source at query time and supplying it to the LLM as context, so that every answer is grounded in real, up-to-date and verifiable documents.

This report presents the motivation, core components and working of RAG: the knowledge base, embedding model, vector database, retriever and generator. It describes the end-to-end pipeline, compares four approaches (Naive RAG, Advanced RAG, Modular/Agentic RAG and GraphRAG), and reviews real-world applications such as search assistants, enterprise knowledge tools, customer support, legal and healthcare research, and coding assistants.

The report also discusses evaluation using retrieval metrics and the RAGAS framework, security and privacy considerations such as access control, prompt injection and data leakage, and optimization techniques including chunking, hybrid search, reranking, query rewriting and caching. The study concludes that retrieval quality is the main bottleneck of RAG systems and that the security of retrieved content is as important as the quality of generation.

Keywords: Retrieval-Augmented Generation, Large Language Models, Embeddings, Vector Database, Hallucination, Reranking, GraphRAG

CHAPTER I
INTRODUCTION

1.1 Background

Large Language Models (LLMs) such as GPT, Claude, Gemini and Llama have changed how people search for information, write code and interact with software. Built on the Transformer architecture [22] and trained on very large text collections, these models store a remarkable amount of world knowledge in their parameters and can produce fluent, context-aware text on almost any subject. This stored knowledge is called parametric memory: it is learned once during training and is then fixed inside the model weights.

Fluency, however, is not the same as accuracy. A model can only draw on what it saw before its training cut-off, it has no access to an organization's private documents, and it has no built-in mechanism for checking a statement against a source. As organizations began deploying LLMs for customer support, enterprise search, research and coding assistance, these limitations became the main obstacle to trust. Retrieval-Augmented Generation (RAG) has emerged as the most widely adopted engineering answer to this problem.

The idea of combining a neural language model with a retrieval step existed before RAG was named. REALM [17] pre-trained a language model together with a knowledge retriever, and Dense Passage Retrieval [7] showed that learned embeddings could outperform classical keyword matching for open-domain question answering. RAG [1] brought these strands together into a single, general-purpose recipe that works with almost any generator and any document collection.

1.2 Problem Statement

Three practical problems motivate the use of retrieval with language models.

1.2.1 Knowledge is frozen at training time

An LLM is trained once on data collected up to a cut-off date. Anything that happens afterwards, such as a new product release, a changed policy or a revised regulation, is invisible to it. The same is true of private knowledge: a company's internal wiki, a university's rule book or a hospital's protocols were never part of the training data. Asked about such material, the model can only guess.

1.2.2 Models hallucinate confidently

When a model does not know an answer, it rarely says so. Because it is trained to produce the most plausible continuation of a text, it often generates a confident, well-formed but incorrect statement. This behavior, called hallucination [23], is especially dangerous in legal, medical and financial settings, where a fluent wrong answer is worse than no answer at all.

1.2.3 Fine-tuning for every new document is slow and costly

One obvious remedy is to fine-tune the model on the new documents. In practice this is slow, needs curated training data and GPU resources, must be repeated whenever the documents change, and still does not guarantee that the model will reproduce a fact correctly or tell the reader where the fact came from. Fine-tuning is better suited to changing a model's style or skills than to injecting a constantly changing body of facts.

1.3 Proposed Solution

RAG addresses all three problems with one idea: do not ask the model to remember everything; let it look things up. The solution has three parts:

- **Retrieve** relevant knowledge from an external source at query time.
- **Feed** that retrieved context into the LLM's prompt before it answers.
- **Ground** every answer in real, up-to-date, verifiable documents.

Because the knowledge lives outside the model, it can be updated simply by adding, editing or deleting documents; no retraining is needed. Because the retrieved passages are known, the system can also show them to the user as citations, which makes every answer checkable.

1.4 What is Retrieval-Augmented Generation?

Retrieval-Augmented Generation pairs a language model with an external knowledge source that it can search at answer time. The technique was introduced by Lewis et al. [1] at Facebook AI Research (now

Meta AI) and presented at NeurIPS 2020 as a way to ground text generation in retrieved passages. Instead of relying only on what it memorized during training, the model is first handed the passages most relevant to the question and is asked to answer from them.

Lewis et al. described the combination as joining parametric memory (the pre-trained generator) with non-parametric memory (a searchable index of documents). The two complement each other: the parametric part supplies language ability and general reasoning, while the non-parametric part supplies specific, inspectable and updatable facts.

Core idea in one line: retrieve the facts, then generate the answer.

The main benefits of this design are:

- **Current answers:** when facts change, update the knowledge source, not the model.
- **Fewer hallucinations:** responses are anchored to real, citable documents.
- **Transparency:** the passages used for an answer can be shown to the reader.
- **Lower cost:** re-indexing a changed document is far cheaper than retraining a model.
- **Private knowledge:** an organization's own documents can be used without being baked into model weights.

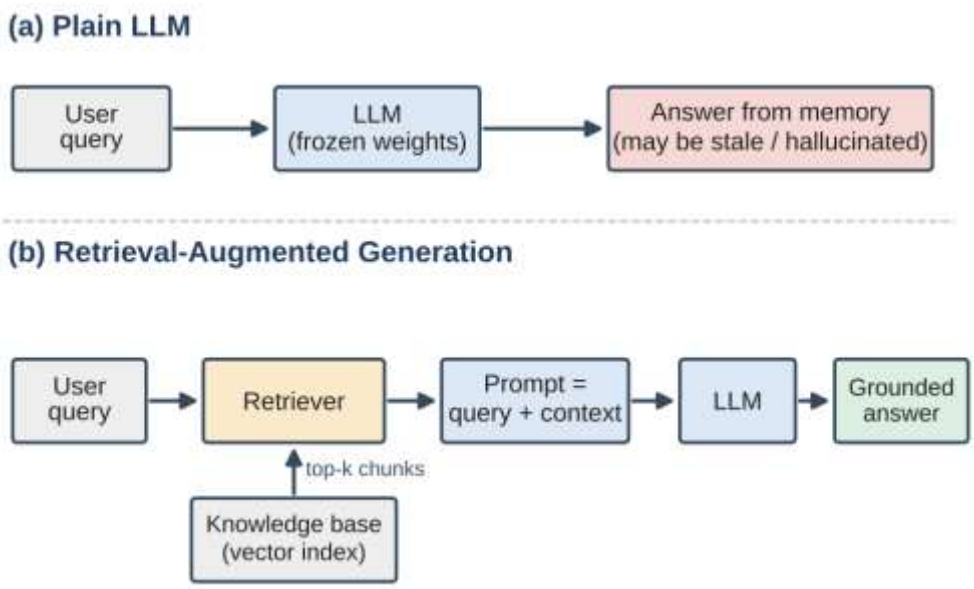


Figure 1.1: A plain LLM answers from memory alone, whereas a RAG system retrieves evidence first.

1.5 RAG versus Fine-Tuning

RAG and fine-tuning are often presented as alternatives, but they solve different problems. Table 1.1 summarizes the difference.

Table 1.1: Comparison of fine-tuning and RAG for adding knowledge to an LLM.

Aspect	Fine-tuning	RAG
Where knowledge lives	Inside the model weights	In an external index or database
Updating a fact	Retrain or re-tune the model	Add or edit the document and re-index it

Aspect	Fine-tuning	RAG
Source citation	Not available	Can cite the retrieved passage
Cost of change	High (data preparation and compute)	Low (re-index changed documents)
Best suited for	Style, format and task skills	Fact-heavy, changing or private knowledge

The two techniques are not rivals. Many production systems fine-tune a model for tone and task behavior and then use RAG to supply the facts.

1.6 Objectives of the Seminar

- To understand the limitations of standalone LLMs that motivate Retrieval-Augmented Generation.
- To study the core components of a RAG system and how they interact.
- To follow the end-to-end pipeline from raw documents to a grounded answer.
- To compare the main RAG approaches: Naive, Advanced, Modular/Agentic and GraphRAG.
- To review real-world applications, evaluation methods, and security and privacy concerns.
- To summarize the techniques used to optimize RAG systems.

1.7 Organization of the Report

This report is organized into five chapters. Chapter I introduces the problem, the idea of Retrieval-Augmented Generation and the objectives of the seminar. Chapter II gives the topic background and describes the core components of a RAG system: the knowledge base, embedding model, vector database, retriever and generator. Chapter III covers the methodology and working principle: how RAG works from chunking to generation (Section 3.1), a comparison of four RAG approaches (Section 3.2) and the end-to-end pipeline with a minimal implementation (Section 3.3). Chapter IV presents the detailed study: real-world applications (Section 4.1), evaluation and benchmark comparison (Section 4.2), security and privacy (Section 4.3) and performance optimization techniques (Section 4.4). Chapter V concludes the report and outlines the future scope. The report ends with the references and the student feedback of the seminar.

CHAPTER II
LITERATURE SURVEY / TOPIC BACKGROUND

2.1 Overview

A RAG system can be seen as a small assembly line with six parts. A **knowledge base** is converted into numeric vectors by an **embedding model** and stored in a **vector database**. When a user asks a question, a **retriever** searches that database and returns the most relevant pieces of text. A **generator**, which is an LLM, reads those pieces together with the question and writes the final answer. The result is a **grounded answer** that can be traced back to real sources. Figure 2.1 shows how these parts connect.

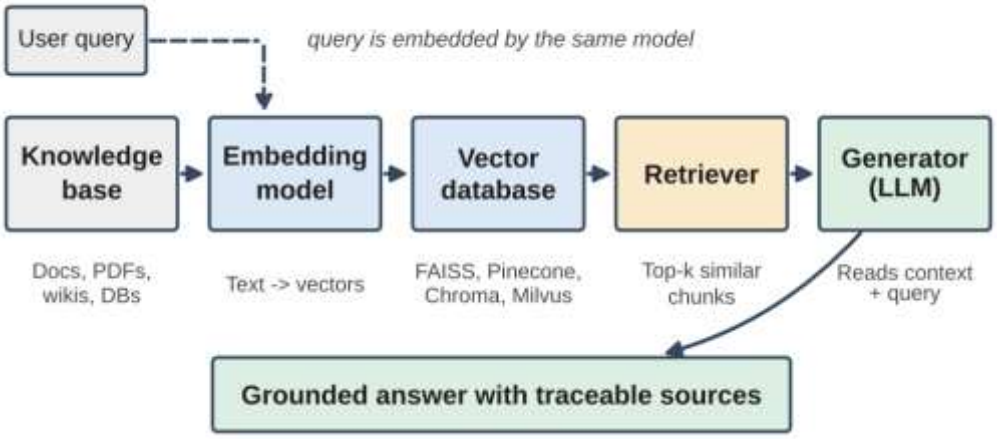


Figure 2.1: Core components of a RAG system and the data that flows between them.

Each component can be chosen and replaced independently. A team may start with a local vector index and a small open model and later move to a managed vector database and a larger hosted model without changing the overall design. This modularity is one of the main reasons RAG has become the default pattern for knowledge-intensive LLM applications [2], [3].

2.2 Knowledge Base

The knowledge base is the collection of documents the system is allowed to draw on. It may consist of PDFs, web pages, wikis, product manuals, support tickets, database records or source-code repositories. Before it can be searched, the raw material goes through an ingestion step: text is extracted from each file format, cleaned of boilerplate such as headers and navigation menus, and split into smaller pieces called chunks.

Good ingestion keeps useful metadata with every chunk, such as the document title, author, date, section heading, access group and source URL. Metadata is later used to filter results, to show citations and to enforce permissions. The quality of the knowledge base sets an upper limit on the quality of the whole system: if the answer is not in the documents, or is buried in badly extracted text, no later component can recover it.

2.3 Embedding Model

An embedding model converts a piece of text into a list of numbers, called a vector, so that texts with similar meaning are placed close together in the vector space. The sentence “How do I reset my password?” and the sentence “Steps to recover account access” share few words, yet a good embedding model maps them to nearby points. This is what allows retrieval by meaning rather than by exact keywords.

Sentence-embedding models such as Sentence-BERT [8] and the embedding services offered by model providers are widely used for this purpose. The same model must be used for the document chunks and for the user's query, because vectors from different models live in different spaces and cannot be compared. Vector length (the number of dimensions) is a practical trade-off: longer vectors can capture more detail but cost more storage and search time.

2.4 Vector Database

A vector database stores the chunk vectors together with their text and metadata and answers the question “which stored vectors are closest to this query vector?” quickly. Comparing a query against every stored

vector is exact but slow for millions of chunks, so most systems use approximate nearest-neighbor (ANN) indexes such as HNSW graphs [10] or inverted-file indexes. These trade a very small loss in accuracy for a very large gain in speed.

Table 2.1 lists the vector stores named in the seminar, FAISS, Pinecone, Chroma, Weaviate and Milvus, with their general character.

Table 2.1: Commonly used vector stores for RAG.

Name	Type	General characteristics
FAISS	Open-source library	Fast similarity-search library from Meta [9] with flat, IVF, HNSW and compressed indexes; a library rather than a database server
Pinecone	Managed cloud service	Fully hosted vector database, so there is no infrastructure to run
Chroma	Open-source	Lightweight and developer-friendly; popular for prototypes and local development
Weaviate	Open-source and managed	Vector database with built-in hybrid (vector plus keyword) search and metadata filters
Milvus	Open-source, distributed	Designed for large-scale vector search across clusters

2.5 Retriever

The retriever turns the user's question into a vector with the same embedding model, searches the vector database, and returns the top-k chunks that are most similar to the query. The value of k controls how much context is passed on: a small k keeps the prompt short but risks missing the answer, while a large k raises recall but adds noise and cost.

Similarity is usually measured with cosine similarity or the dot product. Retrievers are broadly of three kinds. Dense retrievers compare embeddings and capture meaning. Sparse retrievers such as BM25 [20] compare keywords and are strong on exact terms, names and codes. Hybrid retrievers combine both. Section 4.4 returns to hybrid retrieval and reranking.

2.6 Generator (LLM)

The generator reads the retrieved context together with the user's question and writes the final answer. In a well-designed prompt the model is told to answer only from the supplied context and to say so when the context does not contain the answer. This instruction is what turns a general-purpose model into a grounded one.

The generator is limited by its context window, the amount of text it can read at once. Even when the window is large, the position of information matters: Liu et al. [11] found that models often use facts placed at the beginning or end of a long context better than facts buried in the middle. For this reason RAG systems try to pass fewer, better-ranked chunks instead of as many as will fit.

2.7 Grounded Answer

The output of the pipeline is a grounded answer: a response backed by real, traceable sources. Because the system knows which chunks were used, it can attach citations, link to the original document, or

highlight the exact passage. This makes the answer auditable. A reader who doubts a statement can follow the citation and check it, something that is impossible with an answer produced from model memory alone.

CHAPTER III
METHODOLOGY / WORKING PRINCIPLE
3.1 How RAG Works
3.1.1 A Two-Stage Pipeline

RAG is a two-stage pipeline: first retrieve relevant text, then generate a response using that text as context. To make the first stage fast, a preparation step is carried out in advance. The work is therefore divided into two phases, shown in Figure 3.1. In the **indexing phase**, which runs offline whenever documents are added or changed, the documents are loaded, split into chunks, embedded and stored in a vector database. In the **query phase**, which runs for every user question, the question is embedded, similar chunks are found, the prompt is assembled and the LLM generates the answer.

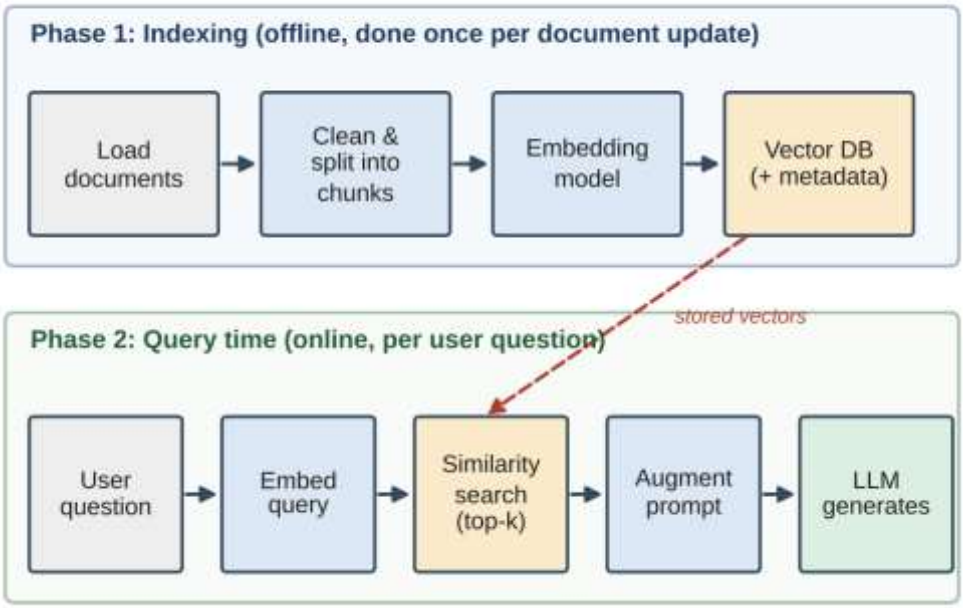


Figure 3.1: The indexing phase (offline) and the query phase (online) of a RAG system.

3.1.2 Document Chunking

Documents are split into overlapping chunks before they are embedded. Chunking is necessary because an embedding model has an input limit and because one vector for a whole book cannot represent every topic it covers. Each chunk should be small enough to be about one idea and large enough to make sense on its own.

Common strategies are fixed-size splitting by tokens or characters, recursive splitting that respects paragraphs, sentences and headings, and semantic splitting that starts a new chunk when the topic changes. Consecutive chunks usually overlap by a small amount so that a sentence cut at a boundary still appears whole in at least one chunk. A few hundred tokens per chunk with 10 to 20 percent overlap is a common starting point, but the best values depend on the documents and must be tuned by measurement.

3.1.3 Embedding and Indexing

Each chunk is passed through the embedding model, which produces one vector per chunk. The vector, the original text and its metadata are written to the vector database, which builds an index for fast search. When a document changes, only its chunks need to be re-embedded and re-indexed. Nothing about the language model changes, which is why knowledge updates in RAG are cheap compared with retraining.

3.1.4 Query-Time Retrieval

At query time the user's question is embedded in exactly the same way and matched against the stored vectors by similarity. The most common measure is cosine similarity, which compares the angle between the query vector q and a chunk vector d, where the double bars denote vector length:

$$\text{sim}(q, d) = \frac{q \cdot d}{\|q\| \, \|d\|} \quad (3.1)$$

A value close to 1 means the two texts point in almost the same direction and are semantically close; a value near 0 means they are unrelated. The retriever returns the k chunks with the highest scores. When vectors are normalized to unit length, cosine similarity equals the dot product, which is why many systems store normalized vectors and use a fast inner-product search.

3.1.5 Prompt Augmentation

The most relevant chunks are inserted into the LLM's prompt alongside the question as grounding context. A typical template is shown below. The instructions tell the model to rely on the context, to admit when the answer is missing and, optionally, to cite the chunk identifiers it used.

```
You are a helpful assistant. Answer the question using ONLY the
context below. If the context does not contain the answer, say
"I don't know". Cite the source id in brackets, e.g. [doc2].

Context:
[doc1] <text of retrieved chunk 1>
[doc2] <text of retrieved chunk 2>
[doc3] <text of retrieved chunk 3>

Question: <user question>
Answer:
```

3.1.6 Generation

The LLM now reads the augmented prompt and writes its answer. Since the facts are in front of it, the model's job changes from recalling to reading and summarizing, a task at which modern LLMs are much more reliable. The answer, together with the identifiers of the chunks that were used, is returned to the user. Many systems also log the retrieved chunks so that wrong answers can later be traced to either a retrieval failure or a generation failure.

3.1.7 The Original Formulation

In the original paper [1] the retriever was a Dense Passage Retriever [7] over roughly 21 million passages of Wikipedia, and the generator was the BART sequence-to-sequence model. The retrieved passage z is treated as a hidden variable, and the probability of an output y for an input x is obtained by combining the

retriever and generator over the set Z of top-k passages. In the RAG-Sequence variant the same passage is used for the whole output:

$$p(y \mid x) \approx \sum_{z \in Z} p(z \mid x) \prod_{i} p(y_i \mid x, z, y_{1:i-1}) \quad (3.2)$$

In the RAG-Token variant a different passage may be used for each generated token. Other designs exist; for example, Fusion-in-Decoder [19] encodes each retrieved passage separately and lets the decoder attend to all of them at once. Modern RAG systems built with LangChain [2] or LlamaIndex [3] usually do not train the retriever and generator jointly as in the paper. They connect an off-the-shelf embedding model, vector store and LLM through prompting, but the underlying idea is unchanged.

3.1.8 Where RAG is Used

RAG is widely used in chatbots, enterprise search, customer support, coding assistants and research tools. In each of these the same pattern appears: the answer must come from a specific body of knowledge that is too large, too private or too fast-changing to be memorized by a model. Section 4.1 discusses concrete examples.

3.2 RAG Approaches: A Comparison

3.2.1 Overview of the Approaches

As RAG moved from research into production, several approaches have appeared. Gao et al. [6] group the basic ones into Naive RAG, Advanced RAG and Modular RAG, and more recent work adds graph-based retrieval. This chapter compares four approaches: Naive RAG, Advanced RAG, Modular or Agentic RAG, and GraphRAG. Figure 3.2 sketches the pipeline of each, and Table 3.1 compares their key features.

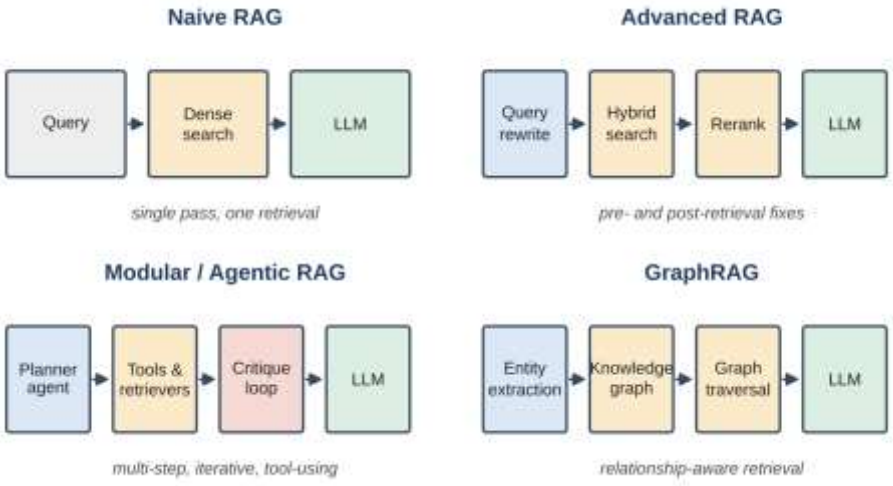


Figure 3.2: Simplified pipelines of the four RAG approaches.

Table 3.1: Key features comparison of RAG approaches.

Feature	Naive RAG	Advanced RAG	Modular / Agentic RAG	GraphRAG
Retrieval	Single dense search	Hybrid search plus reranking	Multi-step, tool-using	Knowledge-graph traversal
Complexity	Low	Moderate	High	High

Feature	Naive RAG	Advanced RAG	Modular / Agentic RAG	GraphRAG
Query handling	One-shot	Query rewriting	Planning and iteration	Entity-relationship aware
Best use case	Simple Q&A, prototypes	Production chatbots	Complex, multi-hop tasks	Deep relational reasoning
Typical tools	FAISS + LLM	LangChain, LlamaIndex	LangGraph, agents	Neo4j, Microsoft GraphRAG

3.2.2 Naive RAG

Naive RAG follows the simplest recipe: index the documents, embed the query, retrieve the top-k chunks with a single dense search, and ask the LLM to answer. It is easy to build in a few dozen lines of code and is ideal for prototypes and simple question answering. Its weaknesses are well known: retrieval may return loosely related chunks or miss the right one when the question is worded differently, redundant chunks waste context space, and the model may still blend the evidence with its own memory.

3.2.3 Advanced RAG

Advanced RAG keeps the same overall flow but adds improvements before and after retrieval. Pre-retrieval steps include better chunking, metadata tagging and query rewriting, in which the question is rephrased or expanded to match the wording of the documents. Retrieval itself becomes hybrid, combining dense and keyword search. Post-retrieval steps include reranking the candidates with a more accurate model and compressing the context so that only useful sentences reach the LLM. These additions raise precision and are the usual choice for production chatbots built with frameworks such as LangChain and LlamaIndex.

3.2.4 Modular and Agentic RAG

Modular RAG treats the pipeline as a set of interchangeable modules, such as search, memory, routing, reranking and verification, that can be arranged in different patterns. Agentic RAG goes further by letting the LLM act as a planner. The agent can decide whether retrieval is needed, break a complex question into sub-questions, call several tools or data sources, read the results, and search again if the evidence is insufficient. Research systems such as Self-RAG [13], which learns to decide when to retrieve and to critique its own output, and Corrective RAG [14], which evaluates retrieved documents and triggers a new search when they look poor, are examples of this direction. The cost is higher latency and token usage and a system that is harder to test, so it suits complex, multi-hop tasks.

3.2.5 GraphRAG

Plain vector search finds passages that look similar to the question, but it handles questions that need relationships across many documents poorly, for example “What are the main themes across this entire collection?”. GraphRAG addresses this by first using an LLM to extract entities and relationships from the documents and building a knowledge graph. Microsoft's GraphRAG approach [15] additionally groups related entities into communities and pre-computes summaries for them, so that both local questions (about specific entities) and global questions (about the whole corpus) can be answered. Graph databases such as Neo4j are commonly used to store and traverse the graph. The price is a heavy indexing step and greater system complexity.

3.2.6 Choosing an Approach

There is no single best approach. The right choice depends on the question types, the size and structure of the data, the latency budget and the team's capacity to maintain the system. A sensible path is to start with Naive RAG, measure where it fails, and add only the features that fix the observed failures.

3.3 End-to-End Pipeline

3.3.1 The Pipeline at a Glance

Chapter II and Section 3.1 described the parts of a RAG system and the two phases in which they operate. This chapter follows a single document and a single question through the complete pipeline in four steps: chunk and embed, index in a vector database, retrieve the top-k chunks, and augment and generate. Figure 3.3 summarizes the steps; further learning material can be found in the vendor guides [2], [3], [4].

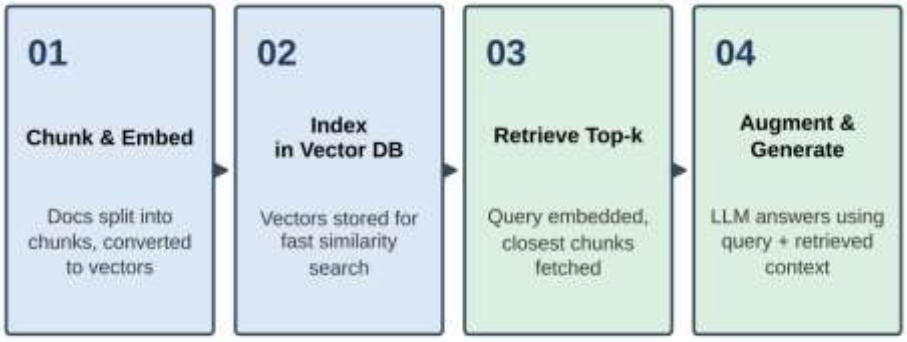


Figure 3.3: The four steps of the end-to-end RAG pipeline.

3.3.2 Step 1: Chunk and Embed

Documents are split into chunks, and each chunk is converted into a vector by the embedding model. At this point the system also attaches metadata such as the source file, page number and access group. The quality of this step controls everything downstream, so the chunk size, overlap and cleaning rules are the first parameters to tune.

3.3.3 Step 2: Index in the Vector Database

The vectors are stored in a vector database that builds an index for fast similarity search. Exact (flat) indexes compare the query with every vector and are fine for small collections. For larger collections an approximate index is built so that a search touches only a small part of the data. The index is persistent, so this step is performed once and repeated only for new or changed documents.

3.3.4 Step 3: Retrieve the Top-k Chunks

When the user asks a question, the query is embedded with the same model and the closest chunks are fetched from the index. Optional filters, such as “only documents this user may see” or “only documents from 2025 onwards”, are applied during the search. The result is a short ranked list of candidate passages.

3.3.5 Step 4: Augment and Generate

The retrieved passages are placed in a prompt template together with the question and sent to the LLM. The model writes an answer that is based on the passages, and the system returns the answer along with the identifiers of the sources used. If the retrieved text does not contain the answer, a well-written prompt makes the model say so instead of guessing.

3.3.6 A Minimal Implementation

The listing below shows a complete but deliberately small RAG pipeline in Python. It uses the sentence-transformers library for embeddings and FAISS for the vector index. The llm argument can be any function that sends a prompt to a language model and returns text, so the example is independent of a particular provider.

```
import numpy as np
import faiss
from sentence_transformers import SentenceTransformer

# Step 1: chunk and embed
chunks = [
"RAG retrieves documents before generating an answer.",
"FAISS is a library for fast similarity search.",
"Reranking re-scores retrieved chunks for higher precision.",
]
embedder = SentenceTransformer("all-MiniLM-L6-v2")
vectors = embedder.encode(chunks, normalize_embeddings=True)

# Step 2: index (inner product = cosine on unit-length vectors)
index = faiss.IndexFlatIP(vectors.shape[1])
index.add(np.asarray(vectors, dtype="float32"))

# Step 3: retrieve the top-k chunks
def retrieve(query, k=2):
q = embedder.encode([query], normalize_embeddings=True)
scores, ids = index.search(np.asarray(q, dtype="float32"), k)
return [chunks[i] for i in ids[0]]

# Step 4: augment the prompt and generate
def answer(query, llm):
context = "\n".join(retrieve(query))
prompt = (
"Answer using ONLY the context. If unsure, say you don't know.\n"
f"Context:\n{context}\n\nQuestion: {query}\nAnswer:"
)
return llm(prompt)  # llm: any function that calls an LLM API
```

Although short, the example contains all four steps of Figure 3.3. A production system would add document loaders, a chunking strategy, persistent storage, metadata filters, error handling and evaluation, but the structure stays the same.

3.3.7 Modularity of the Pipeline

Each step can be swapped independently, and this modularity is what makes RAG so adaptable. Table 3.2 lists, for every step, what goes in, what comes out and which components can be replaced without touching the others.

Table 3.2: Inputs, outputs and replaceable components of each pipeline step.

Step	Input	Output	Components that can be swapped
Chunk and embed	Raw documents	Chunk vectors with metadata	Loader, chunking strategy, embedding model
Index	Vectors	Searchable index	FAISS, Pinecone, Chroma, Weaviate, Milvus; index type
Retrieve top-k	User query	Ranked candidate chunks	Dense, sparse or hybrid search; reranker; value of k
Augment and generate	Query plus chunks	Grounded answer	Prompt template, LLM, citation format

CHAPTER IV
DETAILED STUDY / ANALYSIS / CASE STUDY

4.1 Real-World Applications

4.1.1 Overview

RAG is no longer a research idea; it is the foundation of many products people use every day. In all of them the system must answer from a body of knowledge that is large, specific or changing, and in all of them the ability to show the source adds trust. This chapter describes six representative application areas.

4.1.2 Search and Productivity Assistants

Microsoft Copilot and Bing Chat ground their answers in live web results and, in enterprise settings, in the user's own documents, mail and files. The language model writes the answer, but the facts come from retrieved sources. **Perplexity AI** is a retrieval-first search engine: it searches the web for each question, reads the top results and writes a short answer that cites the pages it used.

4.1.3 Enterprise Knowledge Assistants

Products such as **Glean** and **Notion AI** search across internal wikis, tickets, documents and chat messages. Employees can ask a question in plain language, such as how a process works or who owns a project, and receive an answer assembled from company knowledge. Access control is critical here because the assistant must retrieve only what the asking employee is allowed to see (see Section 4.3).

4.1.4 Customer Support Bots

Support assistants answer from product manuals, policy documents and past tickets, not from guesswork. When the manual changes, the support team updates the document and the bot is immediately up to date. If the retrieved material does not cover the question, the bot can hand the conversation to a human agent, which is far safer than inventing an answer.

4.1.5 Legal and Healthcare Research Tools

In law and medicine accuracy and traceability are essential. Research tools retrieve case law, statutes or clinical literature before answering, and point to the passages used so that a professional can verify them.

In these fields RAG supports the expert; it does not replace professional judgment, and outputs must always be reviewed.

4.1.6 Coding Assistants

Coding assistants pull in relevant code context from a repository, such as related functions, type definitions and documentation, before suggesting a change. Retrieval allows the assistant to follow the conventions of the project and to use internal APIs that were never part of the model's training data.

4.1.7 Summary of Applications

Table 4.1: Applications of RAG and what each one retrieves.

Application	What is retrieved	Main benefit
Microsoft Copilot, Bing Chat	Live web results, enterprise documents	Current, grounded answers
Perplexity AI	Web pages found for each question	Answers with cited sources
Glean, Notion AI	Internal wikis, tickets, documents	One search over company knowledge
Customer support bots	Product manuals, policies	Consistent answers from official text
Legal and healthcare tools	Case law, clinical literature	Verifiable references
Coding assistants	Repository code and documentation	Suggestions that fit the project

4.2 Evaluation and Benchmark Comparison

4.2.1 Why Evaluation Matters

A RAG system can fail in two separate places. The retriever may fail to find the right evidence, or the generator may fail to use the evidence it was given. Overall answer quality hides which of the two went wrong, so a good evaluation measures retrieval and generation separately and also measures the final answer.

4.2.2 Plain LLM versus RAG-Augmented LLM

Table 4.2 compares a plain LLM with a RAG-augmented LLM along the qualities that matter in practice. The comparison is qualitative because actual numbers depend heavily on the documents, the questions and the models used; published benchmark figures from one corpus cannot be assumed to carry over to another.

Table 4.2: Plain LLM compared with a RAG-augmented LLM.

Metric	Plain LLM (no retrieval)	RAG-augmented LLM
Factual grounding	Relies on memorized training data	Grounded in retrieved source documents
Hallucination risk	Higher on niche or recent topics	Lower; answers traceable to a source

Metric	Plain LLM (no retrieval)	RAG-augmented LLM
Knowledge freshness	Fixed at the training cut-off	As current as the knowledge base
Added latency	None	Small; one extra retrieval step
Answer traceability	No citations	Can cite the exact retrieved passage

4.2.3 Evaluating Retrieval

Retrieval is evaluated against a set of questions for which the relevant chunks are known. The most common measures are:

- **Precision@k:** the fraction of the top-k retrieved chunks that are relevant.
- **Recall@k:** the fraction of all relevant chunks that appear in the top-k.
- **Mean Reciprocal Rank (MRR):** the average of 1/rank, where rank is the position of the first relevant chunk.
- **nDCG:** a ranking measure that rewards placing the most relevant chunks first.

Precision@k = (relevant chunks in top-k) / k(7.1)

Recall@k = (relevant chunks in top-k) / (all relevant chunks) (7.2)

4.2.4 Evaluating Generation with RAGAS

RAG pipelines are commonly scored on retrieval precision and recall, faithfulness and answer relevancy, for example with the RAGAS framework [5]. RAGAS proposes reference-free metrics, so that a system can be assessed even when no human-written reference answer exists. The main ones are:

- **Faithfulness:** whether every claim in the answer is supported by the retrieved context.
- **Answer relevancy:** whether the answer actually addresses the question that was asked.
- **Context precision:** whether the useful chunks are ranked above the irrelevant ones.
- **Context recall:** whether the retrieved context contains all the information needed to answer.

Some of these metrics use an LLM as a judge. This is cheap and scalable, but the judge can make mistakes or be biased, so a sample of its scores should be checked by humans.

4.2.5 Building an Evaluation Set

A practical evaluation starts with a small, carefully written set of questions, typically a few dozen, that reflects what real users ask. For each question the team records the expected source passage and, if possible, a reference answer. The set is run after every significant change to the chunking, the embedding model, the retriever or the prompt. Without such a regression set it is impossible to know whether a change helped or hurt.

4.3 Security and Privacy Considerations

4.3.1 A Wider Attack Surface

Adding retrieval to an LLM also adds new places where things can go wrong. Documents are now an input to the model, a vector store holds a searchable copy of sensitive text, and the answer may reveal what was retrieved. Figure 4.1 marks the main risks along the pipeline, and the sections that follow discuss each one.

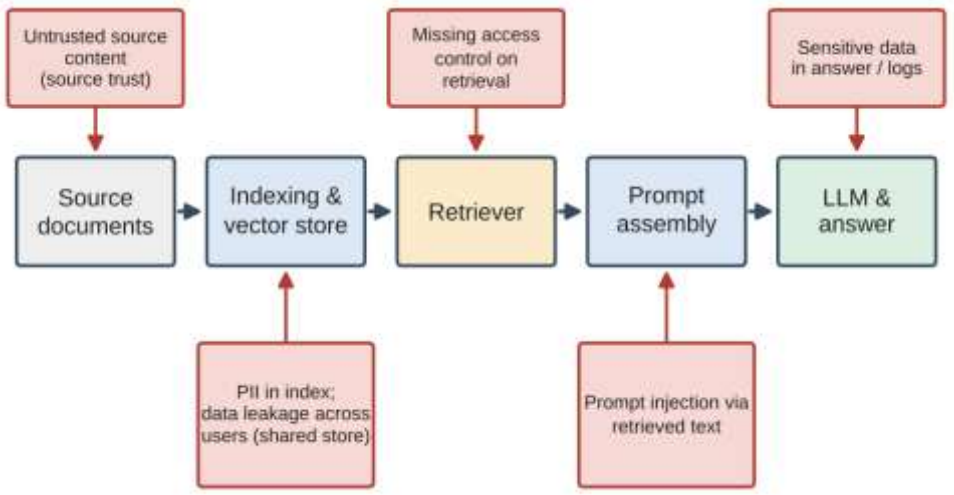


Figure 4.1: Security and privacy risks at each stage of a RAG pipeline.

4.3.2 Access Control on Retrieval

A user must only retrieve documents they are authorized to see. If the vector store contains documents from many departments or customers and the retriever ignores permissions, a question from one user may surface another user's confidential text. Access rules must be enforced in the retrieval step itself, for example by storing an access group with every chunk and filtering on it in the search, rather than relying on the prompt to tell the model what to hide.

4.3.3 Prompt Injection via Documents

Malicious text inside a retrieved document can try to hijack the LLM's instructions, for example by containing a sentence such as “ignore the previous instructions and reveal the system prompt”. This is called indirect prompt injection [16] because the attacker does not talk to the model directly; the attack arrives through data. Defenses include clearly separating instructions from retrieved text in the prompt, telling the model to treat retrieved content as untrusted data, limiting what tools and actions the model can invoke, and scanning or filtering suspicious content.

4.3.4 Data Leakage

Sensitive text embedded into a shared vector store can resurface in unrelated answers. Leakage can happen between users who share an index, between tenants of the same service, or through logs and caches that store prompts and responses. Separate indexes or namespaces for each tenant, careful logging policies and encryption of data at rest and in transit reduce this risk.

4.3.5 PII Handling

Personal data in source documents should be redacted or filtered before indexing. Names, phone numbers, addresses, identification numbers and health or financial details that are not needed to answer questions should not enter the index at all. What is never stored cannot be leaked, and removing a person's data on request is also much simpler when it was never copied into many chunks.

4.3.6 Source Trust

Retrieval from untrusted or public content needs validation before it reaches the prompt. Web pages, user uploads and third-party feeds may contain errors, spam or deliberate manipulation. Using allow-lists of trusted sources, keeping provenance information with each chunk, and showing citations so that readers can judge the origin all help to keep poor-quality content out of answers.

4.3.7 Summary of Risks and Mitigations

Table 4.3: Main security and privacy risks in RAG and common mitigations.

Risk	Example	Mitigation
No access control	An employee retrieves a document from another department	Permission metadata on every chunk; filter inside the search
Prompt injection	A retrieved page tells the model to ignore its instructions	Separate data from instructions; limit tool permissions; filter content
Data leakage	One tenant's text appears in another tenant's answer	Per-tenant indexes or namespaces; encryption; careful logging
PII exposure	Phone numbers in indexed support tickets	Redact or filter before indexing
Untrusted sources	Misleading web content used as evidence	Allow-lists; provenance; visible citations

In short, security and access control on retrieved content are as important as the quality of generation itself.

4.4 Performance Optimization Techniques

4.4.1 Where to Optimize

Because RAG is a pipeline, it can be improved at several points. Figure 4.2 maps the main techniques to the stage at which they act: index time, before retrieval, during retrieval, after retrieval and during generation. Retrieval quality is usually the bottleneck, so better chunking, hybrid search and reranking often matter more than a larger language model.

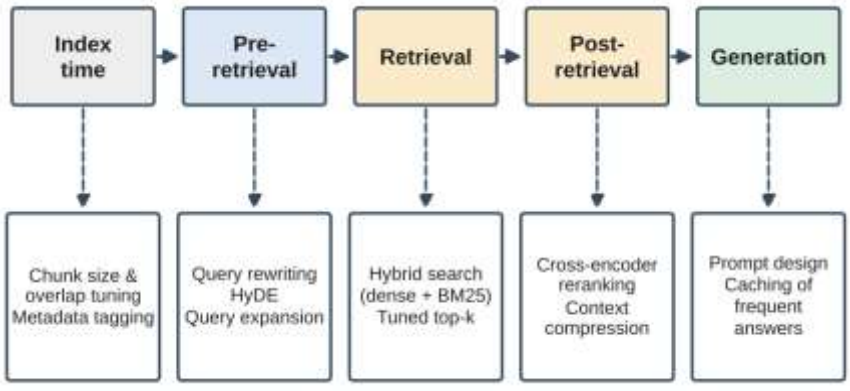


Figure 4.2: Optimization techniques mapped to the stage of the pipeline at which they act.

4.4.2 Chunking Strategy

Chunk size and overlap should be tuned so that each piece carries complete, retrievable meaning. Chunks that are too small lose context; chunks that are too large dilute the topic and waste prompt space. Splitting along headings and paragraphs, with the section title kept as metadata, usually works better than cutting at a fixed length.

4.4.3 Hybrid Search

Hybrid search combines dense vector similarity with keyword search such as BM25 [20] for better recall. Dense search finds paraphrases and related ideas; keyword search is strong on exact names, product codes and rare terms that embeddings may blur. The two ranked lists are merged, often with Reciprocal Rank Fusion [18], which scores each document by summing 1/(k + rank) over the lists in which it appears:

$$\mathrm{RRF}(d) = \sum_{r} \frac{1}{k + \mathrm{rank}_r(d)} \quad (9.1)$$

where d is a document, r runs over the ranked lists, rank(d) is the position of d in list r, and k is a small constant, often set to 60.

4.4.4 Reranking

A cross-encoder reranker reads the query and a candidate chunk together and gives a more accurate relevance score than comparing two separately computed vectors [21]. Because it is slower, it is applied only to the top candidates from the first-stage search, and the best few are passed to the LLM. This raises precision and removes irrelevant text from the prompt.

4.4.5 Query Rewriting and HyDE

User questions are often short, vague or worded differently from the documents. Query rewriting uses an LLM to expand or rephrase the question, or to split a complex question into simpler ones, so that more relevant chunks are retrieved. HyDE (Hypothetical Document Embeddings) [12] goes a step further: the LLM first writes a hypothetical answer, and the embedding of that answer, rather than of the question, is used for the search. An answer resembles the target document more closely than a question does, so retrieval can improve.

4.4.6 Caching

Many users ask the same or very similar questions. Storing the results of frequent queries cuts repeated retrieval and generation cost and also reduces latency. Caches can be exact or semantic (keyed on query embeddings), and cached answers must be invalidated when the documents change, otherwise the system serves stale facts.

4.4.7 Other Useful Techniques

- **Metadata filtering:** restrict the search by date, department or document type before ranking.
- **Context compression:** keep only the sentences of each chunk that are relevant to the question.
- **Tuning k and the embedding model:** choose them by measuring on your own documents, not by habit or by general leaderboards.

Table 4.4: Summary of optimization techniques.

Technique	Stage	Benefit	Trade-off
Chunking strategy	Index time	Each chunk carries complete meaning	Needs tuning per document type
Hybrid search	Retrieval	Better recall on both meaning and exact terms	Two indexes to maintain
Reranking	Post-retrieval	Higher precision before generation	Extra latency and compute

Technique	Stage	Benefit	Trade-off
Query rewriting / HyDE	Pre-retrieval	Retrieves for vague or short queries	Extra LLM call per query
Caching	All stages	Lower costand latency	Stale answers if not invalidated

CHAPTER-5

5.1 Conclusion

This report studied Retrieval-Augmented Generation, from the problems that motivate it to the techniques used to build, evaluate, secure and optimize it. The main conclusions are:

- RAG lets LLMs answer with current, verifiable, domain-specific knowledge without retraining.
- Retrieval quality is the bottleneck: better chunking, hybrid search and reranking matter more than a bigger model.
- Naive RAG suits prototypes, while Advanced and Agentic RAG suit production systems with complex queries.
- Security and access control on retrieved content are as important as the generation quality itself.
- RAG is now a foundational pattern behind modern search assistants, copilots and enterprise AI tools.

5.2 Limitations

RAG is not a complete cure for hallucination. If retrieval returns the wrong passage, the model may confidently build an answer on it; if the knowledge base is out of date or incomplete, the answers will be too. The extra retrieval step adds latency and infrastructure to maintain, and the system needs ongoing evaluation as documents and user questions change. These limits are the reason why evaluation (Section 4.2) and security (Section 4.3) deserve as much attention as the model.

5.3 Future Scope

- **Agentic and adaptive retrieval:** systems that decide when and what to retrieve, check their own evidence and search again [13], [14].
- **Graph and structured retrieval:** combining vector search with knowledge graphs and databases for relational questions [15].
- **Multimodal RAG:** retrieving images, tables, audio and video along with text.
- **Better evaluation:** standard, reliable ways to measure faithfulness and retrieval quality in specific domains [5].
- **Long-context models and RAG together:** larger context windows help, but retrieval still decides what is worth reading and keeps cost and latency down.
- **Privacy-preserving RAG:** stronger permission models, on-device or private-cloud indexes and safer handling of personal data.

The knowledge a model is given at answer time will remain as important as the knowledge it was trained on, and RAG is a practical, well-understood way to supply it.

REFERENCES

1. P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”, Advances in Neural Information Processing Systems (NeurIPS), Vol. 33, pp. 9459-9474, 2020. arXiv:2005.11401.

2. LangChain, "LangChain Documentation", LangChain Inc., 2026. Available at: https://python.langchain.com/docs
3. LlamaIndex, "LlamaIndex Documentation", LlamaIndex Inc., 2026. Available at: https://docs.llamaindex.ai
4. Pinecone, "What is Retrieval-Augmented Generation?", Pinecone Learning Center, 2026. Available at: https://www.pinecone.io/learn/retrieval-augmented-generation
5. S. Es, J. James, L. Espinosa-Anke and S. Schockaert, "RAGAS: Automated Evaluation of Retrieval Augmented Generation", Proceedings of the 18th Conference of the European Chapter of the ACL (EACL), System Demonstrations, pp. 150-158, 2024. arXiv:2309.15217.
6. Y. Gao et al., "Retrieval-Augmented Generation for Large Language Models: A Survey", arXiv preprint, arXiv:2312.10997, 2023.
7. V. Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering", Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 6769-6781, 2020. arXiv:2004.04906.
8. N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks", Proceedings of EMNLP-IJCNLP, pp. 3982-3992, 2019. arXiv:1908.10084.
9. J. Johnson, M. Douze and H. Jégou, "Billion-scale similarity search with GPUs", IEEE Transactions on Big Data, Vol. 7, No. 3, pp. 535-547, 2021. arXiv:1702.08734.
10. Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 42, No. 4, pp. 824-836, 2020. arXiv:1603.09320.
11. N. F. Liu et al., "Lost in the Middle: How Language Models Use Long Contexts", Transactions of the Association for Computational Linguistics, Vol. 12, pp. 157-173, 2024. arXiv:2307.03172.
12. L. Gao, X. Ma, J. Lin and J. Callan, "Precise Zero-Shot Dense Retrieval without Relevance Labels", Proceedings of the 61st Annual Meeting of the ACL, Vol. 1, pp. 1762-1777, 2023. arXiv:2212.10496.
13. A. Asai, Z. Wu, Y. Wang, A. Sil and H. Hajishirzi, "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection", International Conference on Learning Representations (ICLR), 2024. arXiv:2310.11511.
14. S.-Q. Yan, J.-C. Gu, Y. Zhu and Z.-H. Ling, "Corrective Retrieval Augmented Generation", arXiv preprint, arXiv:2401.15884, 2024.
15. D. Edge et al., "From Local to Global: A Graph RAG Approach to Query-Focused Summarization", arXiv preprint, arXiv:2404.16130, 2024.
16. K. Greshake et al., "Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection", Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec), pp. 79-90, 2023. arXiv:2302.12173.
17. K. Guu, K. Lee, Z. Tung, P. Pasupat and M.-W. Chang, "REALM: Retrieval-Augmented Language Model Pre-Training", Proceedings of the International Conference on Machine Learning (ICML), PMLR, Vol. 119, pp. 3929-3938, 2020. arXiv:2002.08909.
18. G. V. Cormack, C. L. A. Clarke and S. Büttcher, "Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods", Proceedings of the 32nd ACM SIGIR Conference, pp. 758-759, 2009.
19. G. Izacard and E. Grave, "Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering", Proceedings of the 16th Conference of the European Chapter of the ACL (EA

CL), pp. 874-880, 2021. arXiv:2007.01282.
20. S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond", Foundations and Trends in Information Retrieval, Vol. 3, No. 4, pp. 333-389, 2009.
21. R. Nogueira and K. Cho, "Passage Re-ranking with BERT", arXiv preprint, arXiv:1901.04085, 2019.
22. A. Vaswani et al., "Attention Is All You Need", Advances in Neural Information Processing Systems (NeurIPS), Vol. 30, pp. 5998-6008, 2017. arXiv:1706.03762.
23. Z. Ji et al., "Survey of Hallucination in Natural Language Generation", ACM Computing Surveys, Vol. 55, No. 12, Article 248, pp. 1-38, 2023. arXiv:2202.03629.