


# RAG is Hard Until I Know these 12 Techniques → RAG Pipeline to 99% Accuracy

 [medium.com/@simranjeetsingh1497/rag-is-hard-until-i-know-these-12-techniques-rag-pipeline-to-99-accuracy-0100d9cb969b](https://medium.com/@simranjeetsingh1497/rag-is-hard-until-i-know-these-12-techniques-rag-pipeline-to-99-accuracy-0100d9cb969b)

Simranjeet Singh

September 27, 2025

| I thought the magic was just in LLMs — I was wrong.

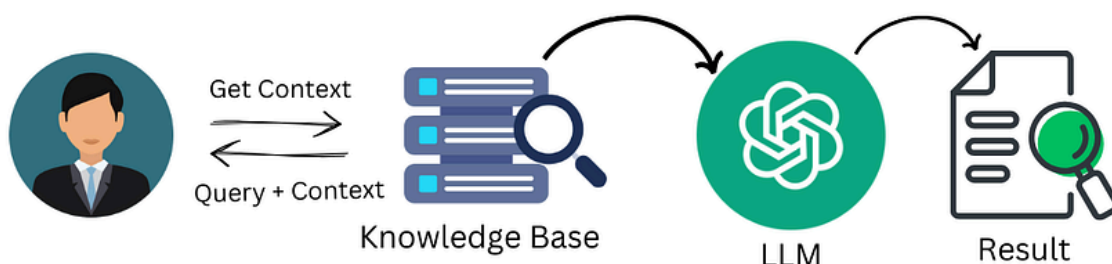
When I first started building **Retrieval-Augmented Generation (RAG)** systems, I was convinced the key to success was simply using a bigger, more powerful LLM. I thought a better model would solve all my problems. **I was wrong.**

After months of building and iterating, **I realized the true magic of RAG isn't about the model itself.** It's about the intelligent art of **how you retrieve, rank, and reason with information.**

Top 12 Techniques



# RAG is Hard Until I Know these Techniques RAG to 99% Accuracy



Read this before your Interview

RAG is Hard Until I Know these Techniques → RAG Pipeline to 99% Accuracy

Today, I'm able to scale my RAG systems to an accuracy rate of 98% — a level I once thought was impossible. The difference wasn't the model; it was the specific techniques I learned along the way. I'm excited to share the exact methods that changed everything for me.

- Why early RAG implementations failed for me.
- Real breakthrough comes from .
- By the end, readers will understand the exact techniques that improved my RAG pipelines to 99% accuracy.

🔔 Follow Me: [Need Help in Career](#) | [Medium](#) | [GitHub](#) | [Linkedin](#) | [Telegram](#)

## Table of Contents

---

1. Why Basic RAG Fails?
2. 12 Core Techniques for High-Performance RAG
3. Measuring Success: The Right Metrics
4. Additional Advanced Techniques
5. Conclusion / Call to Action



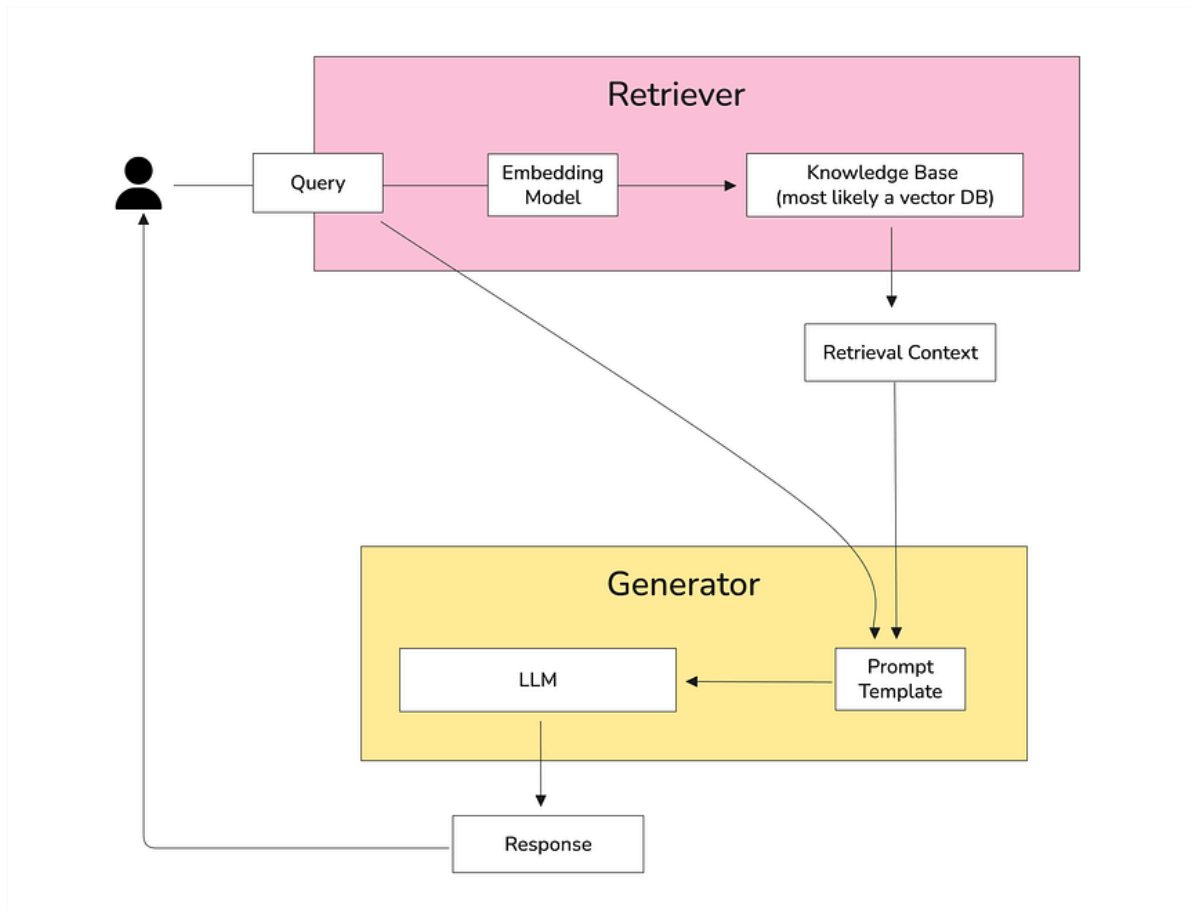
Remember what Eddard Stark Says.

## Why Basic RAG Fails?

---

“Most RAG systems are broken before they even start.”

When I first started building RAG applications, I followed the classic pipeline everyone talks about: **chunk documents** → **embed them** → **store in a vector database** → **retrieve top-k similar chunks**. On paper, it sounds solid — but in real-world use, this approach often falls flat.



Basic RAG Architecture

Here's why:

- Simple similarity search struggles when the answer requires combining information from multiple chunks.
- Traditional chunking breaks natural document flow, causing the model to lose important context.
- A vague or multi-part query often retrieves irrelevant chunks, leaving the LLM guessing.
- When answers require connecting facts across different documents or sources, basic vector search can't handle relationships effectively.

**Real-world examples:** Companies like **DoorDash**, **LinkedIn**, and **Amazon** have realized these limitations and moved far beyond simple vector similarity. They implement advanced retrieval strategies that adapt to query complexity, document structure, and multi-source reasoning, allowing their RAG systems to perform at production scale.

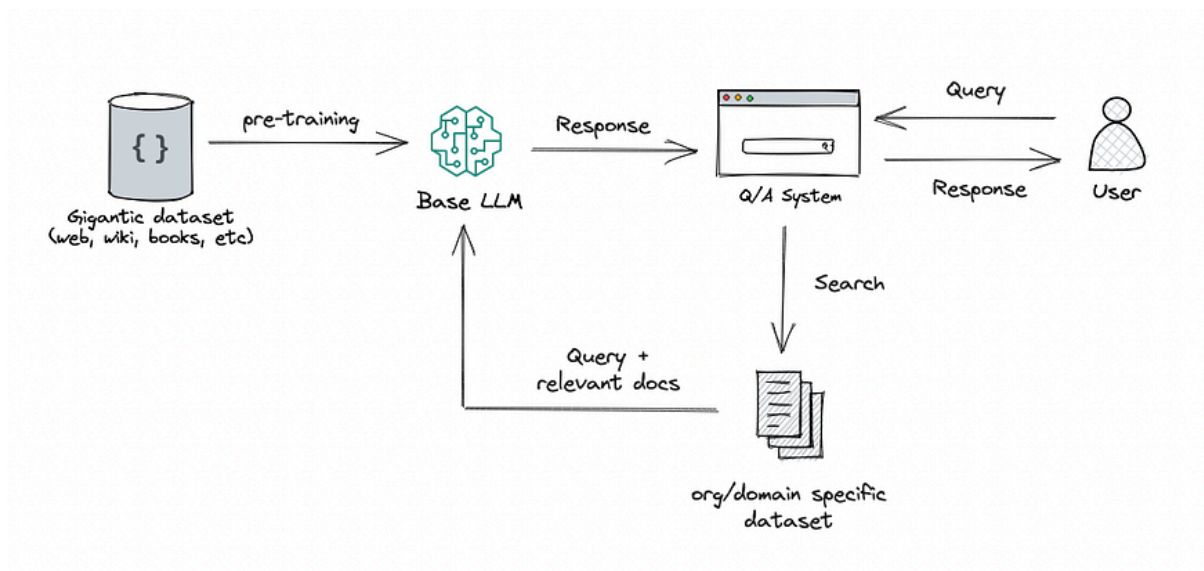
This is why understanding **techniques beyond basic retrieval** is critical to building high-accuracy RAG applications.

## Core Techniques for High-Performance RAG

“The model isn't the magic — these techniques are.”

When I first built RAG systems, I assumed that a bigger LLM would solve every problem. I was wrong. The real difference comes from **how you retrieve, rank, and structure your knowledge**.

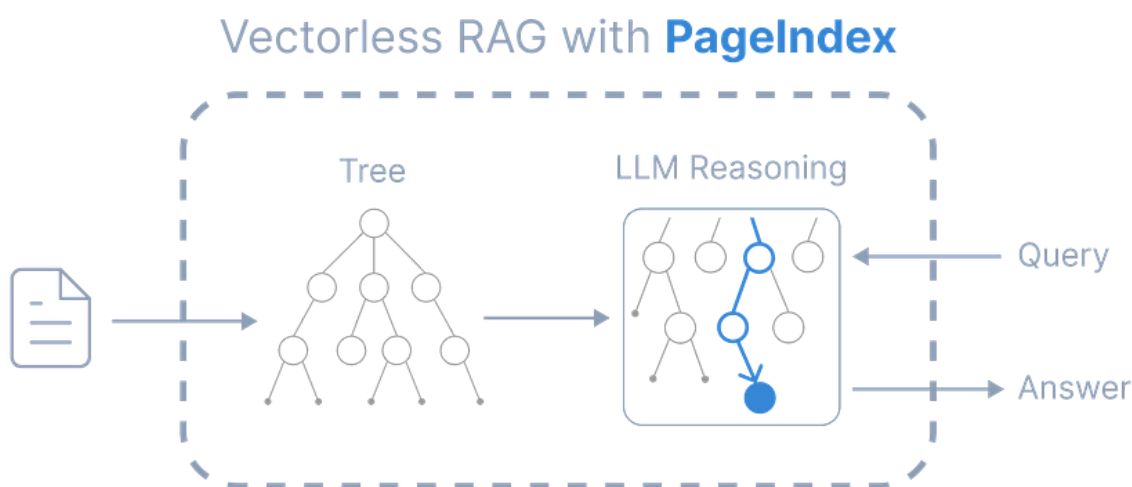
Let's dive into the techniques that boosted my RAG pipeline to **99% accuracy on FinanceBench**.



High-Performance RAG

## 1. PageIndex: Human-like Document Navigation

Instead of treating documents as isolated chunks, [PageIndex](#) builds a **hierarchical tree** of the document — much like a human skim-reading a report.



PageIndex, Source: <https://github.com/VectifyAI/PageIndex>

### Why it works:

- can reason over sections instead of scanning irrelevant chunks.
- No artificial chunking → preserves natural structure.
- you can trace which nodes were used in the reasoning process.

## Why to use it?

It solves the fundamental problem of “**chunking**.” Instead of retrieving disconnected text snippets, it allows the **LLM to navigate and reason** over entire documents and their structure, just as a human would. This eliminates hallucination and **improves accuracy by over 40%** on complex datasets, as the model has full context.

## Disadvantages:

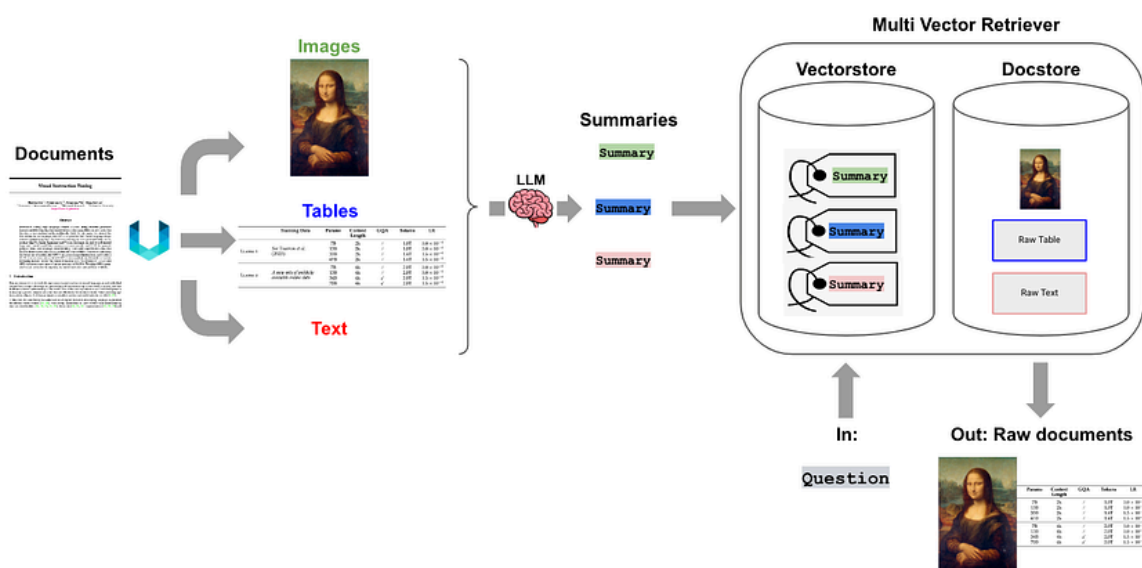
- Building and maintaining a hierarchical tree is complex and resource-intensive.
- Reasoning over a tree can be slower than a simple vector search.

**Fact:** On FinanceBench, vector RAG achieved ~50% accuracy, while PageIndex hit **98.7%**.

```
langchain.indexes PageIndex langchain.llms OpenAIDocument_tree =  
PageIndex.from_documents(documents)llm = OpenAI(temperature=)query = response =  
document_tree.query(query, llm=llm)(response)
```

## 2. Multivector Retrieval

Instead of a single embedding per chunk, generate **multiple embeddings** representing different aspects: summary, keywords, potential questions, and full text.



Multivector Retrieval, Source: <https://blog.langchain.com/semi-structured-multi-modal-rag/>

## Why it works:

- One embedding can't capture all relevance angles.
- Multiple embeddings allow matching content to queries in .

## Why to use it?

It overcomes the “**single perspective**” **limitation** of a single vector embedding. By capturing multiple facets of a document (e.g., summary, keywords, questions), it drastically improves the chances of a successful and relevant retrieval, especially for complex or multi-faceted queries.

#### Disadvantages:

- Storing multiple embeddings per chunk significantly increases storage requirements.
- The indexing process is more complex and time-consuming.

```
langchain.vectorstores.FAISS sentence_transformers SentenceTransformerModel =  
SentenceTransformer() embeddings = [ model.encode(chunk.text),  
model.encode(chunk.summary), model.encode(chunk.keywords)  
] faiss_index = FAISS.from_embeddings(embeddings)
```

### 3. Metadata Augmentation

---

Metadata enriches each chunk with **extra context**: source, author, creation date, quality scores, related entities.

#### Benefits:

- Improves retrieval precision.
- Helps models prioritize .

#### Why to use it?

It **adds crucial context that vector** search alone misses. By including information like source, date, and author, it helps the **LLM make more informed and trustworthy decisions**, leading to higher-quality answers and better citation recall.

#### Disadvantages:

- Manually adding metadata can be time-consuming and prone to human error.
- The quality of the retrieval is highly dependent on the quality of the metadata.

```
chunk = {      : ,      : ,      : ,      : [, ],      : }  
vectorstore.add_texts([chunk[]],  
metadatas=[chunk])
```

### 4. CAG (Cache-Augmented Generation)

---

CAG reduces latency for **frequently-accessed static data** by preloading it into the model’s **key-value cache**.

- Ideal for compliance rules, product manuals, or internal guidelines.
- When combined with RAG:

#### Why to use it?



It dramatically **reduces latency and computational cost** for frequently asked questions or static information. By **pre-loading data into the cache**, it bypasses the entire RAG pipeline, **providing instant, low-cost answers** for common queries.

#### Disadvantages:

- Caching is only effective for high-frequency, static data; it's not useful for dynamic or new information.
- Preloading data into the KV cache consumes significant memory.

```
kv_cache = CAGCache()kv_cache.preload(static_documents)response = llm.generate(query, context=kv_cache)
```

**Fact:** Companies using CAG report **50–70% faster response times** for high-traffic queries.

## 5. Contextual Retrieval

---

This technique ensures **each chunk carries enough context** before embedding. Inspired by Anthropic's research, it prevents LLMs from misinterpreting isolated chunks.

```
():  
    embeddings = model.encode([contextualize(c)  c  chunks])
```

- Alone, improves retrieval accuracy by ~49%.
- Combined with reranking → up to .

#### Why to use it?

It directly **addresses “retrieval failures” caused by ambiguous** or complex queries. By adding a concise, model-generated context, it guides the **search and ensures the most relevant information is retrieved** on the first attempt, improving efficiency and accuracy.

#### Disadvantages:

- The process of adding context before embedding can introduce a small amount of latency.
- Effectiveness relies on well-designed prompts to add useful context.

## 6. Reranking

---

“The first retrieval isn't enough — you need a second opinion.”

Reranking adds a **secondary model** to evaluate the relevance of initially retrieved documents. This step ensures that the LLM only sees the **most contextually appropriate chunks**, rather than blindly trusting vector similarity.

#### Why to use it?

It elevates the quality of retrieved results from “potentially relevant” to “highly relevant.” A simple semantic search often returns a list of related documents, but a **reranker sorts them by true relevance**, ensuring the LLM only reasons over the best possible evidence.

**Fact:** Reranking can improve accuracy by **up to 60%** over simple semantic similarity.

```
transformers.AutoModelForSequenceClassification, AutoTokenizer, tokenizer =
AutoTokenizer.from_pretrained(model =
AutoModelForSequenceClassification.from_pretrained()) inputs =
tokenizer([query]*(candidates), candidates, return_tensors='pt', padding=True) scores =
model(**inputs).logits.squeeze().detach().numpy() ranked = [c, c * ((scores,
candidates), reverse=True)] ranked_top_candidates = rerank(ranked, retrieved_docs)
```

### Disadvantages:

- Running a second model for reranking adds computational time and latency to the retrieval process.
- Reranking with a more sophisticated model can increase costs.

## 7. Hybrid RAG

---

“One method isn’t enough — combine strengths.”

Hybrid RAG combines **vector-based semantic search** with **graph traversal** to handle:

- Semantic similarity (vector search).
- Entity relationships and dependencies (graph).

**Why it matters:** Ideal for **multi-hop reasoning** where answers require connecting facts across documents.

### Why to use it?

It **combines the strengths of two different retrieval methods**. Vector search excels at finding semantically similar information, while graph traversal is **perfect for multi-hop reasoning and connecting entities**. This allows the system to handle both simple and complex, interconnected data queries effectively.

```
vector_candidates = vectorstore.similarity_search(query)graph_candidates =
knowledge_graph.query_related_entities(query)hybrid_candidates = rerank(query,
vector_candidates + graph_candidates)
```

### Disadvantages:

- Combining vector search with a graph traversal system is technically challenging to build and maintain.
- Requires managing two distinct search methods, increasing infrastructure and maintenance costs.



## 8. Self-Reasoning

---

“Make your LLM an active quality inspector.”

Instead of trusting retrieved chunks blindly, the **LLM evaluates its own inputs** in multiple stages:

- Assesses retrieval quality.
- Selects key sentences with justification.
- Synthesizes reasoning paths into final answers.

### Why to use it?

It transforms the RAG system from a passive retriever into an active, intelligent agent. By having the LLM evaluate the retrieved chunks itself, **it can correct its own mistakes, refine its search**, and significantly reduce hallucination and improve citation accuracy.

**Fact:** Self-reasoning achieves **83.9% accuracy** versus 72.1% for standard RAG and improves **citation recall**.

### Disadvantages:

- The multi-stage reasoning process is computationally expensive and slow.
- The performance is highly dependent on the LLM's ability to reason, which isn't always reliable.

```
chunk    retrieved_chunks:    relevance_score = llm.evaluate_relevance(chunk,
query)    relevance_score > :    evidence_chunks.append(chunk) answer =
llm.synthesize_answer(evidence_chunks)
```

## 9. Iterative / Adaptive RAG

---

“Not all queries are equal — treat them differently.”

Adaptive RAG **routes queries** based on complexity:

- Simple factual queries → single-step retrieval.
- Complex analytical queries → multi-step iterative retrieval.

### Why to use it?

It creates a **more efficient and responsive system**. Instead of using a one-size-fits-all approach, it matches the **retrieval strategy to the query's complexity**, saving time and resources for simple questions while ensuring thoroughness for complex ones.

### Disadvantages:

- The logic for routing queries based on complexity is difficult to implement and fine-tune.

- Building a system with different retrieval paths adds significant development and testing effort.

**Benefit:** Optimizes accuracy **and cost**, focusing compute where it's needed.

```
complexity = query_classifier.predict(query) complexity == :    answer =
simple_retrieve(query):    answer = iterative_retrieve(query)
```

## 10. Graph RAG

---

“Turn documents into a knowledge network.”

Graph RAG builds **knowledge graphs** from documents, connecting entities, concepts, and facts:

- Supports multi-hop reasoning.
- Reduces hallucinations.
- Example integration:

### Why to use it?

It provides a robust **solution for multi-hop reasoning** and knowledge-intensive tasks. By creating a structured knowledge graph, it allows the **LLM to follow relationships between entities**, leading to highly accurate and hallucination-free answers for complex, interconnected queries.

### Disadvantages:

- Creating and maintaining a knowledge graph from unstructured text is a complex and time-consuming task.
- Scaling the graph and graph traversal to a massive number of documents can be challenging.

```
langchain.graphs KnowledgeGraphkg =
KnowledgeGraph()kg.add_entities_from_documents(documents)kg.add_edges_from_relation
= kg.query_entities(, depth=)
```

## 11. Query Rewriting

---

“Don’t make your system guess — clarify intent first.”

- Transforms ambiguous or poorly structured queries into .
- Breaks complex questions into sub-queries.
- Adds for precision.

### Why to use it?

It directly **improves the user experience** by making the system more **robust to ambiguous or poorly formulated questions**. By automatically clarifying or breaking down queries, it ensures the search is successful **even when the user's initial prompt is not perfect**.

#### Disadvantages:

- The LLM's rewriting process could misinterpret the user's intent, leading to incorrect results.
- Rewriting the query before search adds an extra step and latency.

```
():  
    clean_query = rewrite_query()
```

## 12. BM25 Integration

---

| *"Combine semantic intelligence with exact matching."*

BM25 integration blends **semantic vector search** with **keyword-based ranking**:

- Runs both searches in parallel.
- Merges results using weighted scoring.
- Applies reranking to pick the most relevant candidates.

#### Why to use it?

It adds a crucial layer of **lexical precision to semantic search**. By including keyword matching, it ensures that the system can **find exact terms, names, or codes** that a purely semantic search might miss, creating a more comprehensive and accurate result.

#### Disadvantages:

- Weighting the results between vector and keyword search requires careful and complex tuning.
- BM25 alone lacks semantic understanding, which can impact the quality of the overall result if not balanced properly.

```
rank_bm25 BM25OkapiBM25 = BM25Okapi([doc.split() doc docs])bm25_scores =  
bm25.get_scores(query.split())final_scores = * vector_scores + *  
bm25_scoresranked_docs = [doc _, doc ((final_scores, docs), reverse=)]
```

These advanced techniques form the **backbone of high-accuracy, production-ready RAG systems**. When I applied them systematically, my pipelines went from basic ~50–60% accuracy to **99% on FinanceBench**, with faster, more human-like retrieval and minimal hallucinations.

## Implementation Strategy: Phase-Wise

---

| *"Techniques are useless without a systematic plan."*

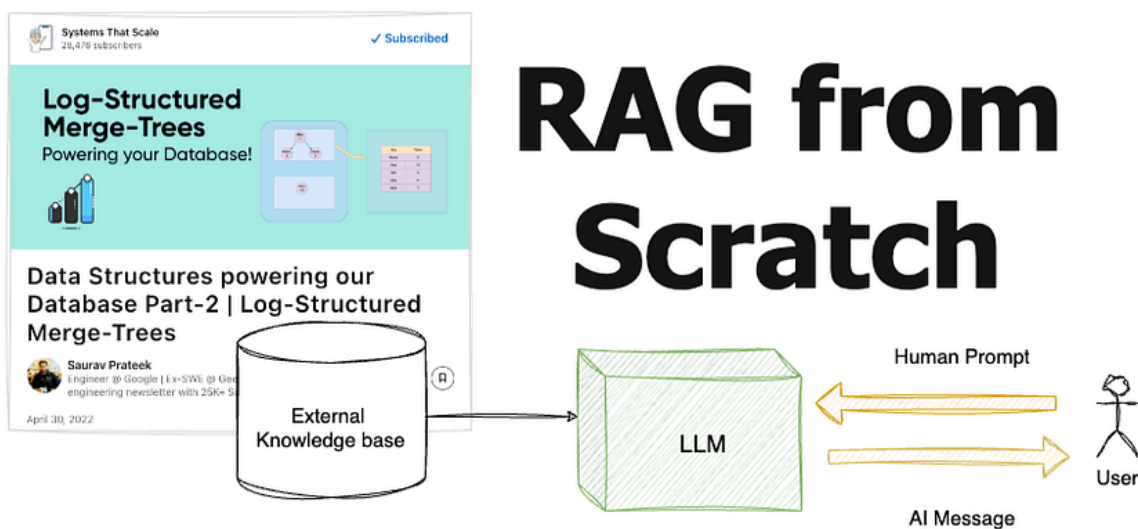
All the RAG techniques we discussed are powerful — but without a **structured implementation strategy**, even the best methods fail to deliver consistent results. Here's a **phase-wise approach** that worked for me and can be applied to most RAG systems:

## Phase 1: Foundation

---

**Goal:** Build a solid base for your retrieval system.

- Clean, normalize, and structure your documents (remove noise, handle tables, HTML parsing if needed).
- Avoid arbitrary splitting; preserve semantic boundaries so that each chunk retains meaningful context.



Build Good Knowledge Base [VectorDB]

```
():\n    clean_text = clean_html(doc)\n    split_into_semantic_chunks(clean_text)\nchunks = [preprocess_document(d) for d in documents]
```

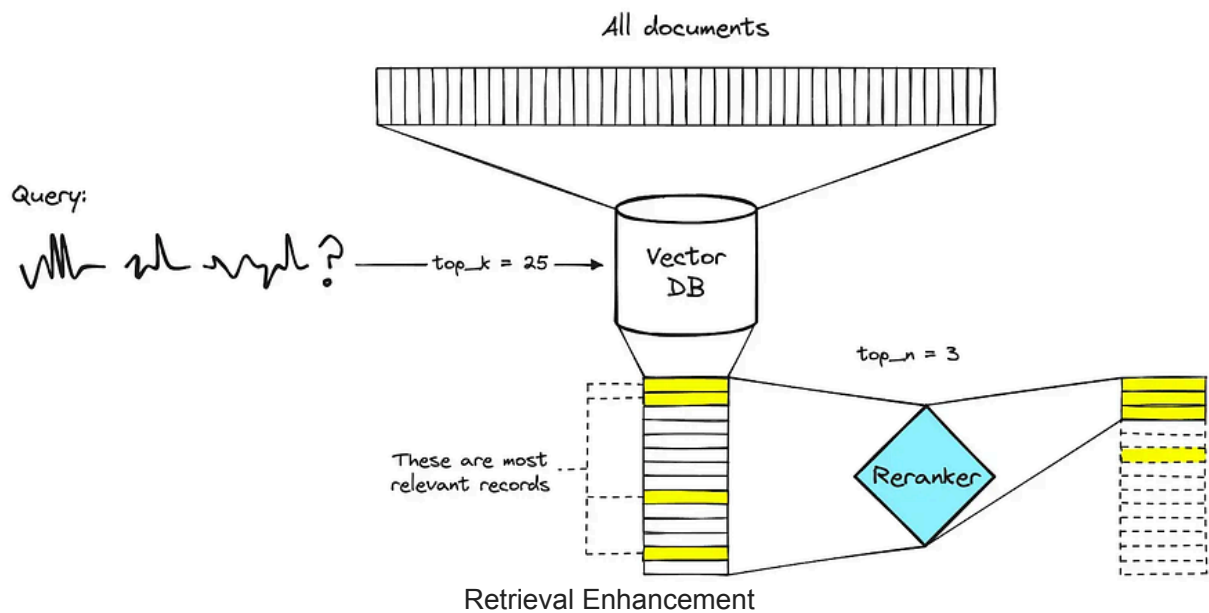
**Impact:** Proper foundation often delivers **the largest initial gains** in retrieval accuracy.

## Phase 2: Retrieval Enhancement

---

**Goal:** Improve the relevance of the documents your model sees.

- Add a secondary model to prioritize the most contextually relevant chunks.
- Combine semantic vector search with exact keyword matching to capture precise terms.



### Example:

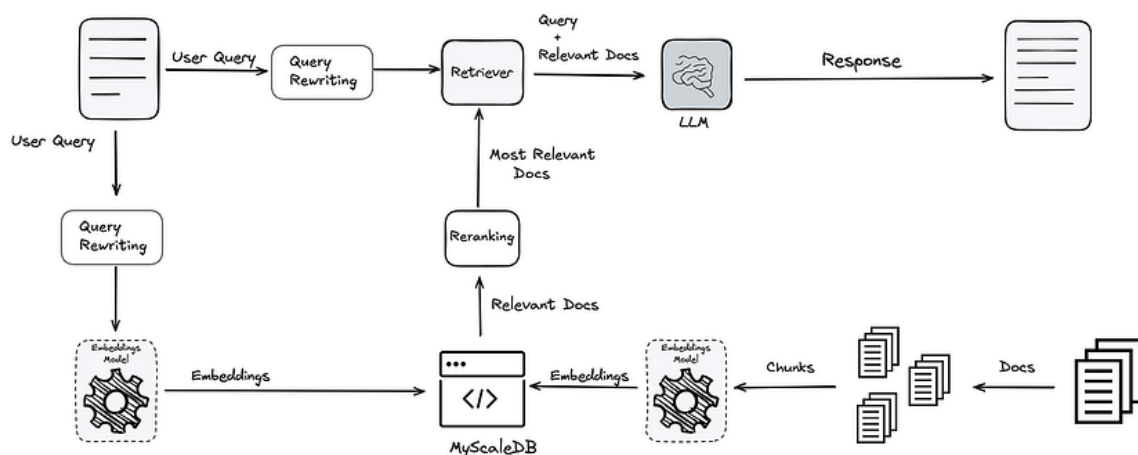
```
final_scores = * vector_scores + * bm25_scores
ranked_docs = [doc _, doc _]
((final_scores, docs), reverse=)]
```

**Impact:** Improves retrieval precision and reduces irrelevant information feeding into the LLM.

## Phase 3: Intelligence Layers

**Goal:** Make your system query-aware and adaptive.

- Clarify user intent before retrieval to reduce ambiguity.
- Route queries based on complexity; simple queries get fast single-step answers, complex queries get multi-step reasoning.



Intelligence Layers in RAG and LLM based Architecture

```
query_classifier.predict(query) == :    answer = iterative_retrieve(query):
answer = simple_retrieve(query)
```

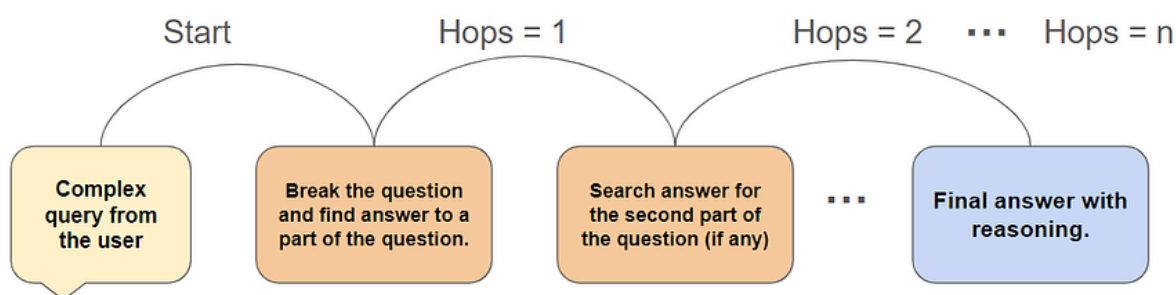
**Impact:** Optimizes both **accuracy** and **latency**, focusing resources where they are needed most.

## Phase 4: Advanced Techniques

---

**Goal:** Handle complex, multi-hop, or connected data effectively.

- Capture relationships between entities and concepts across documents.
- Have the LLM evaluate chunk relevance and synthesize reasoning paths.
- Use multiple embeddings per chunk for different aspects (summary, keywords, questions).



### Multi-Hop Retrieval

Multi-Hop and Handling Complex Queries

#### Example:

```
related_entities = knowledge_graph.query_entities(, depth=)
evidence_chunks = [c c
retrieved_chunks llm.evaluate_relevance(c, query) > ]
answer = llm.synthesize_answer(evidence_chunks)
```

**Impact:** Maximizes **accuracy**, **context-awareness**, and **reliability** for production-level RAG systems.

Implementing RAG is not just about picking techniques — it's about **layering them strategically**. By following this phase-wise plan, you ensure that each technique compounds the effectiveness of the previous one, leading to **high-accuracy, scalable RAG pipelines**.

## Measuring Success: The Right Metrics

---

“If you can't measure it, you can't improve it.”

Implementing a high-performance RAG system is only half the battle. The other half is **measuring how well it actually works**. Without the right metrics, you won't know which techniques are helping and which aren't.

Here are the **key metrics for evaluating RAG systems**:



## 1. Faithfulness

---

**Definition:** Measures how well the model's response aligns with the retrieved context.

- Ensures that the answer isn't hallucinating facts.
- Critical for domains like finance, healthcare, or legal where accuracy is non-negotiable.

**Example Concept:**

```
faithfulness_score = llm_judge.evaluate_faithfulness(response, retrieved_chunks)
```

## 2. Answer Relevance

---

**Definition:** Does the response actually answer the user's query?

- High retrieval accuracy doesn't guarantee the LLM produces relevant answers.
- Evaluated by both automated judges and human reviewers.

**Example:**

```
relevance_score = llm_judge.evaluate_relevance(response, query)
```

## 3. Context Precision & Recall

---

**Context Precision:** Are the most relevant documents ranked at the top?

**Context Recall:** Are all relevant documents retrieved?

- Precision ensures LLM sees .
- Recall ensures , especially for multi-hop or connected queries.

**Example Concept:**

```
precision = compute_precision(retrieved_docs, ground_truth_docs)
recall = compute_recall(retrieved_docs, ground_truth_docs)
```

## 4. Continuous Evaluation: LLM-as-a-Judge + Human Validation

---

Modern production RAG systems often use **LLM-as-a-judge** frameworks to automatically evaluate metrics at scale:

- Judges score outputs on faithfulness, relevance, and format compliance.
- Human reviewers validate samples to ensure .

```
alignment_rate = llm_judge.calculate_human_alignment(sample_responses,
human_labels)
```

**Fact:** LLM judges can align with human reviewers **85% of the time**, often exceeding human inter-rater agreement (81%).

Using these metrics systematically ensures that your RAG system isn't just "working" — it's **accurate, reliable, and continuously improving**.

## Additional Advanced Techniques

---

*"Once you master the basics, these advanced techniques take your RAG system to the next level."*

Even with a strong foundation and phase-wise implementation, high-performance RAG systems benefit from **specialized optimizations and advanced methods**. Here are some techniques that helped me push accuracy and scalability to production-ready levels:

### 6.1 Embedding Model Optimization

---

**Definition:** Using **domain-specific embeddings** instead of generic ones to better capture nuances in your data.

- Generic embeddings often miss domain jargon, abbreviations, or structured content.
- Switching to embeddings trained on finance, medical, or legal data can dramatically improve retrieval accuracy.

**Example:**

```
sentence_transformers SentenceTransformerModel = SentenceTransformer()  
embeddings = model.encode(documents)
```

### 6.2 Late Chunking

---

**Definition:** Generate embeddings for the **entire document first**, then split into chunks afterward.

- Preserves full context within each chunk.
- Especially useful for long documents where traditional chunking breaks context.

**Example Concept:**

```
full_embedding = model.encode(full_document)chunks_embeddings =  
split_embedding_into_chunks(full_embedding, chunk_size=)
```

### 6.3 Advanced Document Preprocessing

---

**Definition:** Handle complex document structures beyond plain text:

- Preserves headings, tables, and hierarchy.
- Extracts structured data accurately.
- Handles images, charts, or PDFs with embedded text.

**Example Concept:**

```
bs4 BeautifulSoupsoup = BeautifulSoup(html_content, )text_chunks =
[section.get_text() section soup.find_all()]
```

## 6.4 Fusion-in-Decoder (FiD)

---

**Definition:** Process **question-document pairs in parallel** before generation, allowing the model to consider multiple sources simultaneously.

- Improves for complex queries.
- Better than sequential token-by-token generation when synthesizing information.

### Example Concept:

```
responses = fid_model.generate(query, context_documents)final_answer =
combine_responses(responses)
```

## 6.5 RAG-Token vs RAG-Sequence

---

**Definition:** Two strategies for integrating retrieved content during generation:

- Chooses different documents for each generated token → fine-grained integration.
- Selects the best overall document for the entire answer → simpler, more stable output.

### Use Case:

- RAG-Token → great for multi-source synthesis with high detail.
- RAG-Sequence → better for concise, single-source answers.

### Example Concept:

```
answer = rag_sequence_model.generate(query, retrieved_documents)answer =
rag_token_model.generate(query, retrieved_documents)
```

These advanced techniques are optional but **highly impactful** for production-grade RAG applications, especially when dealing with **long documents, complex queries, or domain-specific datasets**.

## Conclusion / Call to Action

---

“RAG is evolving — your systems must too.”

Most engineers start by thinking the **LLM is the magic**. But as we’ve seen, real performance comes from the **retrieval and reasoning strategies** you build around it. Accuracy, scalability, and reliability don’t come from bigger models — they come from smarter systems.

If you’re building RAG for production, remember:

- → it’s about PageIndex, reranking, metadata, and adaptive pipelines.

- → it's about CAG, contextual retrieval, and caching the right data.
- → it's about measuring the right metrics and systematically applying advanced techniques.

Don't rely on one-off hacks. **Adopt a systematic, phase-wise improvement strategy** to take your RAG applications from prototype to production-ready.

### 👉 Your next step:

- Subscribe for more deep-dive posts on .
- Implement one technique from this guide in your current pipeline.
- Share your results — I'd love to see what accuracy gains you achieve.

The future of RAG isn't just about retrieval. It's about **intelligent, adaptive information systems** — and you have the chance to build them today.

## Problems RAG solves



**Hallucinations**



**Out-of-date info**

RAG solves the basic LLMM problems

**Read my Google Interview Experience and Joining as AI/ML Engineer**

**[TCS to Google | Manual Tester to AI/ML Engineer | Tier 3 to FAANG | Interview and Learning...](#)**

[generativeai.pub](https://generativeai.pub)

If my story inspired you, I invite you to **follow my journey** and stay connected:


I've also compiled **learning resources and guides** that helped me at every stage — from Data Science fundamentals to advanced GenAI and LLM projects. Links to these resources are available in the description and posts.

Remember, your dream is closer than you think. Start small, stay consistent, and never give up. The next success story could be yours.

**Keep learning. Keep building. Keep believing.**

Looking ahead, I'm excited to share you my **75 Hard GenAI Challenge** in which you learn GenAI for Free from Scratch.



 Agentic AI 14+ Projects- <https://www.youtube.com/playlist?list=PLYIE4hvbWhsAkn8VzMWbMOxetpaGp-p4k>

 Learn RAG from Scratch — <https://www.youtube.com/playlist?list=PLYIE4hvbWhsAKSZVAn5oX1k0oGQ6Mnf1d>

 Complete Source Code of all 75 Day Hard

 GitHub — [https://github.com/simranjeet97/75DayHard\\_GenAI\\_LLM\\_Challenge](https://github.com/simranjeet97/75DayHard_GenAI_LLM_Challenge)

 Kaggle Notebook — <https://www.kaggle.com/simranjeetsingh1430>



 Learn GenAI for Free [Free Courses and Study Material with Daily Updates and Learning's Uploaded] Join Telegram  — <https://t.me/genaiwithsimran>


 Exclusive End to End Projects on GenAI or Deep Learning or Machine Learning in a Domain Specific way — <https://www.youtube.com/@freebirdscrew2023>

**You can also schedule a meeting with me here.**


Link — <https://topmate.io/simranjeet97>





If you like the article and would like to support me make sure to:


 Clap for the story (100 Claps) and follow me  Simranjeet Singh

 View more content on my Medium Profile

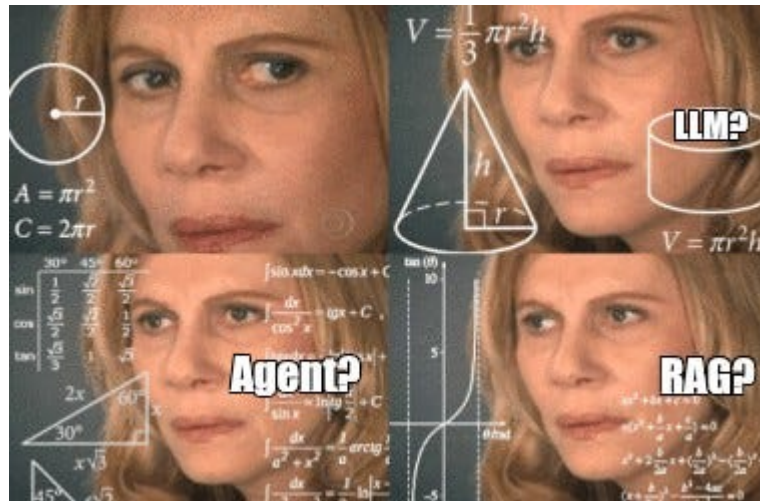
 Follow Me: LinkedIn | [Medium](#) | [GitHub](#) | [Linkedin](#) | [Telegram](#)

 Help me in reaching to a wider audience by sharing my content with your friends and colleagues.

 Do Donate  or Give me a Tip  If you really like my blogs. Click Here to Donate or Tip  — <https://bit.ly/3oTHiz3>

 If you want to start a career in Data Science and Artificial Intelligence you do not know how? I offer data science and AI mentoring sessions and long-term career guidance.

 17 1:1 Guidance — About Python, Data Science, and Machine Learning



Confused? Schedule call with me on Topmate Now? <https://topmate.io/simranjeet97>