

RAG: Retrieval, Embeddings & Evaluation

From frozen knowledge to grounded, cited answers.

The Problem: LLMs Have Frozen Knowledge

Why LLMs can't answer everything

KAUST Academy



Knowledge Cutoff

Training ends, knowledge freezes.
No awareness of anything new after the cutoff date.



No Private Data

Cannot access internal documents,
proprietary databases, or personal
notes.



No Verification

Everything recalled from compressed
statistical patterns. No way to double
check answers.

Try it: Ask an LLM "What is our company's refund policy?" — it has no idea.

When LLMs Don't Know, They Make Things Up

The hallucination problem

KAUST Academy

LLMs are trained to produce fluent, confident-sounding text. When they lack knowledge, they generate plausible but fabricated answers.

Factual Fabrication

Inventing facts, statistics, or citations that simply don't exist.

Outdated Information

Confidently stating something that used to be true but has since changed.

Source Confusion

Mixing up details from different entities, people, or events.

Hallucination Has Real Consequences

This isn't a theoretical problem

KAUST Academy

Mata v. Avianca (2023)

A lawyer used ChatGPT to draft a legal brief.

The model invented six fake court cases with fake citations that sounded completely real.

The lawyer submitted them to a federal judge — and was sanctioned.

Looked legitimate



Completely fabricated



Lawyer sanctioned

This is why grounding LLM outputs in real documents matters.

The RAG Solution

What is Retrieval Augmented Generation?

KAUST Academy

Retrieval Augmented Generation is a technique where, before generating a response, the system first **retrieves** relevant information from an external knowledge source and provides it to the LLM as context.

The LLM then **generates** an answer grounded in the retrieved documents, not just its training data.

Simple Analogy

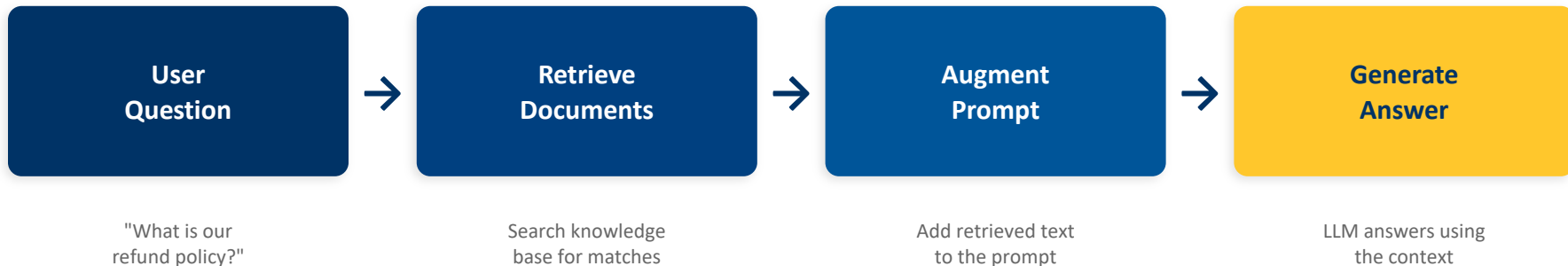
Vanilla LLM
= Closed-book exam

RAG
= Open-book exam

How RAG Works

Retrieve first, then generate

KAUST Academy



Without RAG:

"I don't have information about your specific company policies."

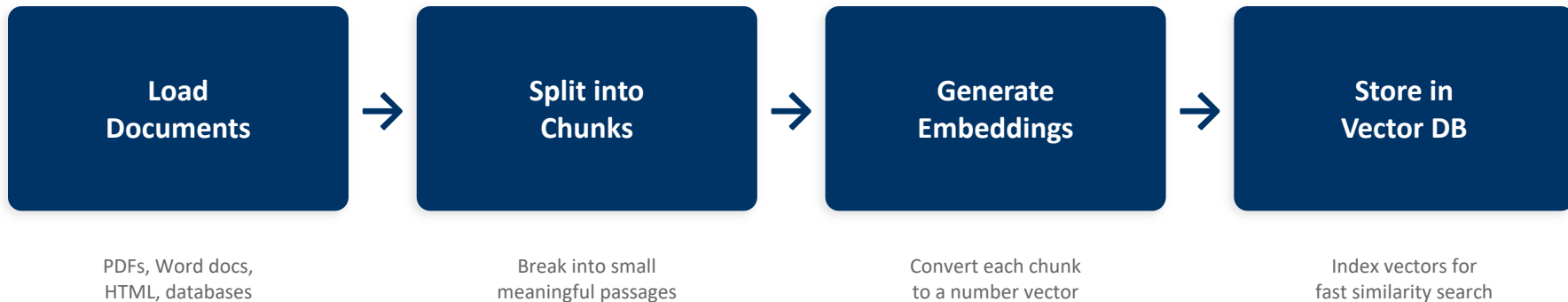
With RAG:

"Per our policy doc, returns are accepted within 30 days..."

RAG Architecture: Indexing Pipeline

Preparing your knowledge base (runs offline)

KAUST Academy

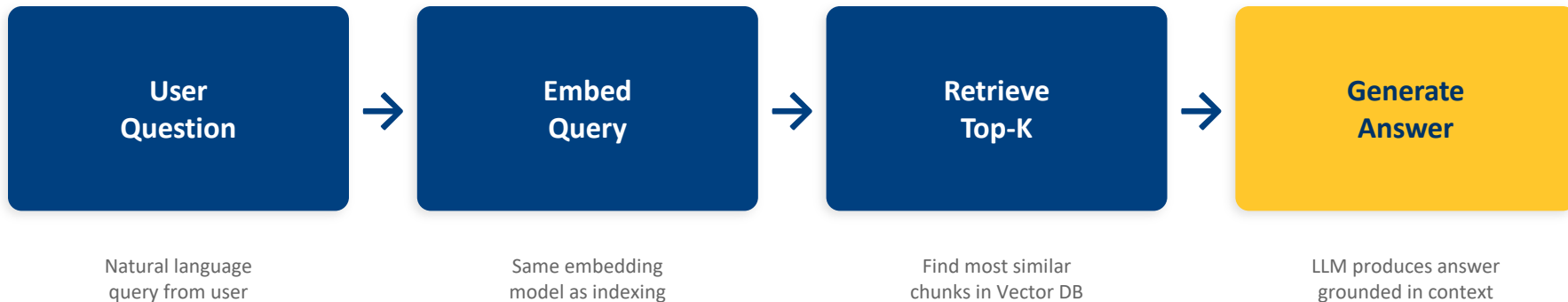


Key insight: The indexing pipeline runs once (or periodically when documents update). It prepares your knowledge base so it can be searched at query time.

RAG Architecture: Query Pipeline

Answering questions in real time

KAUST Academy



Critical rule: You must use the same embedding model for both the indexing and query pipelines. Different models produce incompatible vector spaces.

RAG vs Fine-Tuning vs Prompting

Choosing the right approach

KAUST Academy

	Prompting	Fine-Tuning	RAG
How	Manually paste relevant information into the prompt	Re-train model weights	Dynamically retrieve relevant documents at query time
Pros	Zero infra, instant	Learns style & behavior	Up-to-date, scalable, citable
Cons	Context window limit	Expensive, slow to update	Requires vector DB infra
Best for	Small fixed knowledge, prototyping	Tone, format, behavior	Factual Q&A, knowledge bases

These approaches are not mutually exclusive — many production systems combine all three.

Decision Guide

When should you use RAG?

KAUST Academy

Use RAG When...

- Your data changes frequently
- You need cited, document-grounded answers
- Too much data for a single prompt
- You need private or proprietary data access
- Accuracy and trust matter more than speed

Skip RAG When...

- The task is purely creative
- Built-in LLM knowledge is sufficient
- You need to change the model's behavior
- Your knowledge fits in one prompt and never changes

Rule of thumb: If a human would need to look something up to answer correctly, you probably need RAG.

RAG in the Real World

Who's using RAG today?

KAUST Academy



Customer Support

Klarna & Intercom power chatbots with help center articles and product docs.



Internal Knowledge Assistants

Glean & Notion AI search company wikis, Slack, and databases.



Legal Analysis

Law firms search thousands of case files and retrieve relevant precedents.



Medical Q&A

Healthcare systems ground answers in clinical guidelines and literature.

CHAPTER 2

Embeddings Deep Dive

How machines understand meaning through numbers

What are embeddings

Similarity search

Choosing models

Limitations

What Are Embeddings?

Turning text into meaningful numbers

KAUST Academy

An **embedding** is a dense vector — a list of floating-point numbers — that represents the **semantic meaning** of text.

You pass text through a specialized neural network (an embedding model), and it outputs a fixed-length vector, typically 384 to 3,072 dimensions.

The key idea: Similar meanings produce similar vectors. The model captures meaning, not just word overlap.

"The cat sat on the mat"

▼ Embedding Model

[0.12, -0.34, 0.56, ...]

"A feline rested on the rug"

▼ Embedding Model

[0.11, -0.33, 0.55, 0.79, ...]

▼ Calculate similarity

Almost identical!

Embeddings Capture Meaning, Not Words

Why this matters for search

KAUST Academy

Semantically Close

"Dog" ↔ "Puppy" **0.95**

"Car" ↔ "Automobile" **0.93**

"Happy" ↔ "Joyful" **0.91**

Semantically Far (when used in different sentences)

"Bank" (river) ↔ "Bank" (money) **0.34**

"Apple" (fruit) ↔ "Apple" (co.) **0.28**

"Cat on mat" ↔ "Stocks crashed" **0.07**

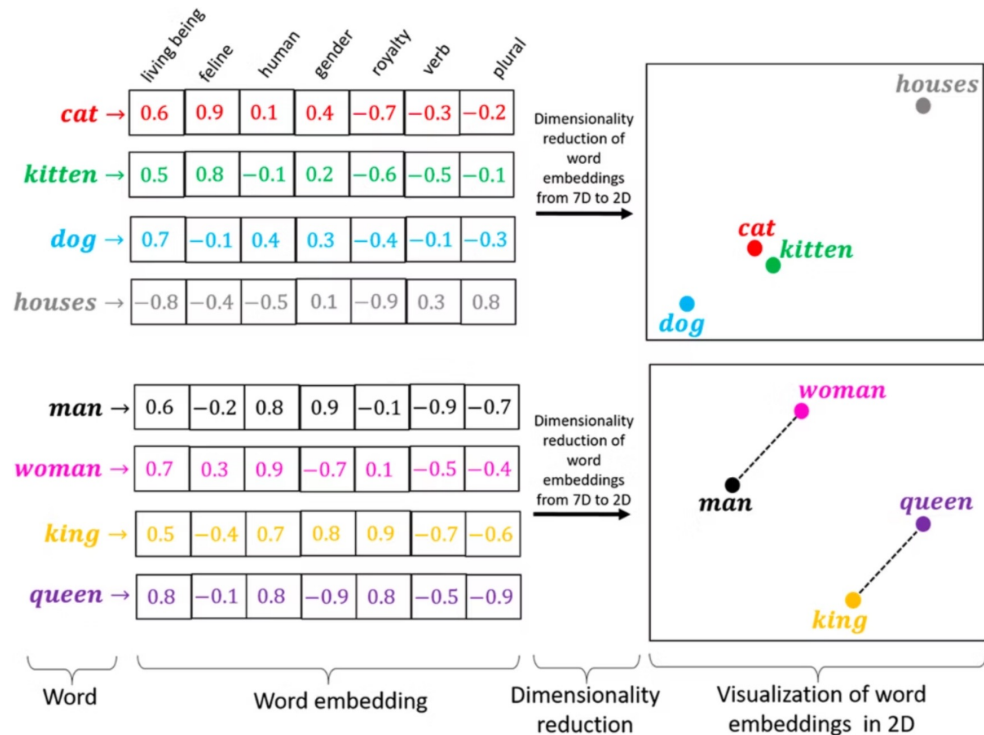
This is the magic: Keyword search for "puppy" misses documents about "dog." Embedding search finds them because the vectors are close.

Embeddings in Space

How does it look like?

KAUST Academy

Important caveat: Real embeddings live in 768+ dimensions, not 2. We use dimensionality reduction (t-SNE, UMAP) to visualize them in 2D. The actual high-dimensional space captures far more nuance than any 2D plot can show.



Finding Relevant Text with Vector Similarity

The retrieval mechanism

KAUST Academy



Embed Query

Convert the user's question into a vector using the same embedding model.



Compare Vectors

Compute cosine similarity between the query vector and every stored vector.



Return Top-K

Return the K most similar chunks as context for the LLM.

Choosing the Right Embedding Model

Popular options and trade-offs

KAUST Academy

Model	Dims	Type	Best For
text-embedding-3-small	1,536	OpenAI API	Most use cases, low cost
text-embedding-3-large	3,072	OpenAI API	When precision matters
all-MiniLM-L6-v2	384	Open source	Prototyping, very fast
BAAI/bge-large-en-v1.5	1,024	Open source	High quality, free
Cohere embed-v3	—	Cohere API	Multilingual support

Critical rule: You must use the same embedding model for indexing and querying. Mixing models means vector spaces don't align.

Limitations

Important limitations to keep in mind

KAUST Academy

Text Length Limits

Most models cap at 8,192 tokens. Longer text gets truncated — this is why chunking matters.

No Reasoning Ability

Embeddings measure similarity, not logic.
"Paris **is** the capital" and "Paris is **NOT** the capital" look similar.

Domain Mismatch

A model trained on Wikipedia may struggle with medical records, legal contracts, or specialized jargon.

Ambiguity Without Context

Short texts are harder to embed accurately. "Apple" alone — fruit or company?

Practical tip: Always test embedding quality on your data. Run sample queries and check if results make sense.

CHAPTER 3

RAG Pipeline Components

Indexing → Retrieval → Generation — the three pillars of every RAG system

Document loading

Chunking strategies

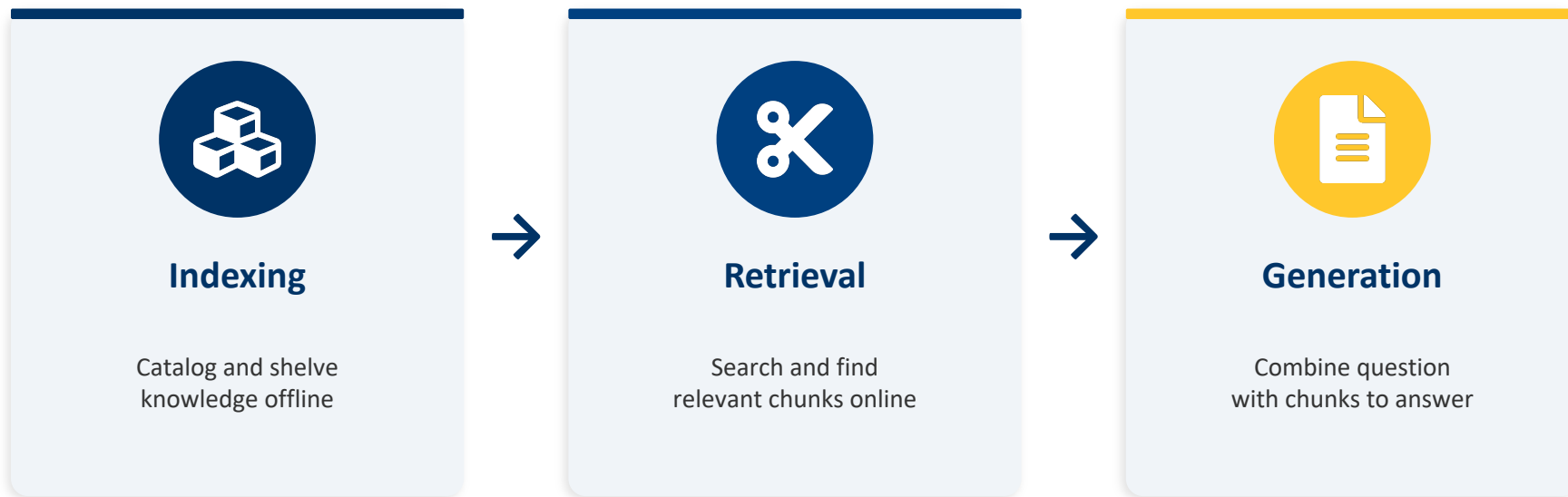
Embeddings

Vector storage

The Three Stages of RAG

Every RAG system follows this pattern

KAUST Academy



Document Loading

Getting your raw data into the pipeline

KAUST Academy

Your knowledge might live in many formats. The first step is ingesting all of it into a unified pipeline.

PDFs

Word docs

HTML pages

Databases

APIs

Notion / Confluence

Slack messages

Plain text

Common Tools

LangChain Document Loaders

160+ loaders for various formats

LlamaIndex Readers

Similar breadth of connectors

Unstructured.io

Specialized in parsing real-world documents

Best practice: Always inspect your loaded documents manually before proceeding. Garbage in = garbage out.

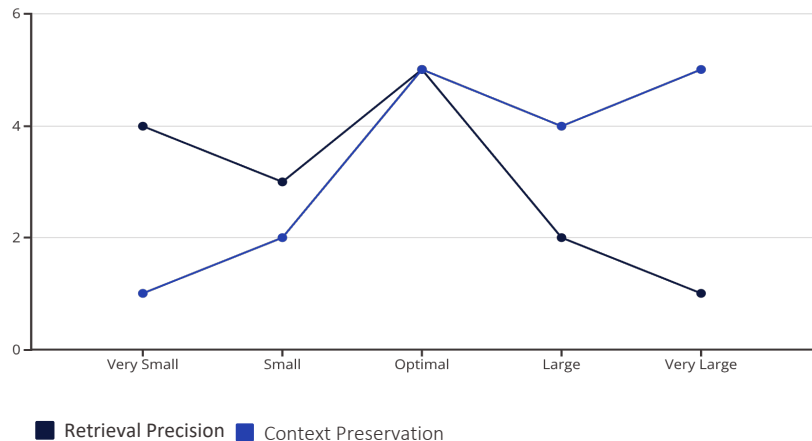
Chunking: Breaking Documents into Pieces

Why we need to split documents

KAUST Academy

Embedding models have token limits (max 8,192 tokens). You can't embed a 1,000-page PDF as one vector.

Retrieval precision: If you embed entire documents, search might return a 20-page doc when only one paragraph is relevant.



The Chunking Dilemma

Too small → Chunks lose context. A sentence alone may not carry enough meaning.

Too large → Chunks contain too much noise. The relevant sentence is buried.

Goal: Each chunk should be a self-contained, meaningful unit of information.

Chunk Overlap: Why It Matters

Preventing information loss at chunk boundaries

KAUST Academy

When you split a document, important info can be split at the boundary. **Overlap** ensures boundary content appears in both chunks.

Full Document (1,500 tokens)

Without Overlap

Chunk 1 (1–500)

Chunk 2 (501–1000)

Chunk 3 (1001–1500)

✗ Info at boundaries is lost ✗

With Overlap (50 tokens)

Chunk 1 (1–500)

Chunk 2 (451–950)

Chunk 3 (901–1400)

Overlap zones — boundary content appears in both chunks

Typical overlap: 50–200 tokens. Too much overlap wastes storage; too little defeats the purpose.

Chunking Strategies Compared

Three main approaches

KAUST Academy

	Fixed-Size	Recursive Splitting	Semantic Chunking
How	Split by token count with overlap	Split by \n\n, then \n, then sentence	Embed sentences, group by similarity
Pros	Simple, predictable sizes	Respects document structure	Thematically coherent chunks
Cons	Cuts mid-sentence blindly	Still size-based at core	Slow, complex, variable sizes
Best for	Prototyping, uniform text	Most use cases (default)	Multi-topic documents

Recommendation: Start with recursive splitting (chunk_size=500–1000, overlap=50–200). It's the default in LangChain.

Recursive Character Splitting

The recommended default strategy

KAUST Academy

01

First Priority

Split by double newline (`\n\n`) — paragraph breaks

02

Second Priority

If still too large, split by single newline (`\n`) — line breaks

03

Third Priority

If still too large, split by sentence (period + space)

04

Last Resort

Split by character

It respects document structure as much as possible while staying within the size limit.

Full section text embedded as one chunk

Using document structure as natural boundaries

KAUST Academy

The simplest form of semantic chunking: use **document structure** (headers, sections) as natural chunk boundaries.

Source: Employee Handbook

H1 Company Policies

H2 1.1 Refund Policy

Returns are accepted within 30 days...

H2 1.2 Leave Policy

Employees are entitled to 12 weeks...

H2 1.3 Remote Work

Employees may work remotely up to...



Resulting Chunks

Chunk 1

Section: Refund Policy

Returns are accepted within 30 days of purchase. Original receipt required. Refunds processed within 5-7 business days. Items must be in original packaging...

Chunk 2

Section: Leave Policy

Employees are entitled to 12 weeks of paid parental leave after 1 year of employment. Leave must be requested 30 days in advance...

Chunk 3

Section: Remote Work

Employees may work remotely up to 3 days per week with manager approval. A stable internet connection is required...

When to use: Works best for well-structured documents (handbooks, wikis, docs with clear headers).

Metadata: Don't Just Store Text

Attach context to every chunk

KAUST Academy

Every chunk should carry metadata — structured information about where it came from.

What to Attach

source	"company_handbook.pdf"
page	12
section_title	"Refund Policy"
date_updated	"2024-11-15"
author	"HR Department"
category	"policies"

Example: Stored Chunk

text: "Returns are accepted within 30 days..."

metadata: {

```
source: "handbook.pdf",  
page: 12,  
section: "Refund Policy",  
date: "2024-11-15"
```

}

Why it matters: Metadata lets you filter before searching. "Current refund policy" → filter source="policies", date ≥ 2024 → then vector search.

Contextual Chunk Enrichment

Prepend context before embedding for better retrieval

KAUST Academy

Problem: A raw chunk "Returns are accepted within 30 days" doesn't mention it's about refund policy. The embedding lacks that broader context.

Before Enrichment

"Returns are accepted within 30 days of purchase. Original receipt required."



After Enrichment

Document: Company Policy Manual
Section: Refund Policy

"Returns are accepted within 30 days of purchase. Original receipt required."

Why This Works

- The embedding now captures that this chunk is about refund policy.
- Queries like "What is the refund policy?" match better because of shared vocabulary in the prefix.
- All chunks from the same section get similar embeddings, improving coherence.

Document-Based Chunking

Handling special document types

KAUST Academy



Markdown

MarkdownTextSplitter splits using headers and sections as separators.



Python / JS Code

PythonCodeTextSplitter splits based on class, function, and logical boundaries.



Tables

Format as HTML/CSV for the LLM. Summarize tables and embed the summaries.



Images

Use multi-modal models (e.g. GPT-4V) to generate text summaries, then embed, or use a VLM

Generating Embeddings for Your Chunks

Converting text to vectors

KAUST Academy

Each text chunk is passed through an embedding model to produce a vector that captures semantic meaning.

Practical Considerations

Batch Processing

Send chunks in batches (e.g. 100 at a time) for efficiency.

Cost

If using OpenAI's API, you pay per token embedded. Calculate costs upfront.

Speed

Open-source models on a GPU can be faster and cheaper at scale.

Consistency

Use the exact same model version if you re-embed documents later.

Storing Vectors in a Vector Database

The end of the indexing pipeline

KAUST Academy

A **vector database** is a specialized database designed to store high-dimensional vectors and perform fast similarity searches.

What You Store Per Chunk

- The vector (the embedding)
- The original text (to show the LLM)
- Metadata (source file, page, date, author, category)

Example: Stored Chunk

Embedding: [0.12, -0.01 , 0.33, 3.2...]

text: "Returns are accepted within 30 days..."

```
metadata: {  
  source: "polices.pdf",  
}
```

Why not a regular database?

Traditional databases use exact matches (SQL WHERE).

Vector databases use approximate nearest neighbor (ANN) search — finding the most similar vectors quickly.

Chunking Best Practices — Summary

Putting it all together

KAUST Academy

1

Start with recursive splitting

chunk_size=500–1000, overlap=50–200. Default in LangChain.

2

Use structure when available

Headers/sections make natural chunk boundaries.

3

Always add metadata

Source, page, section title, date — for filtering and citation.

4

Enrich chunks before embedding

Prepend section title + doc name. Dramatically improves retrieval.

5

Use overlap

50–200 tokens prevents losing info at boundaries.

6

Test on real queries

Check top results manually. Move to semantic chunking only if needed.

CHAPTER 4

Retrieval

Finding the right information at query time

Semantic search

Keyword search

Hybrid search

Reranking

Semantic Search — Finding by Meaning

The primary retrieval method in RAG

KAUST Academy

Embed the user's query → compare against stored vectors → return closest matches. **It understands meaning, not keywords.**

Example

Query: "How do I return a product?"

Our refund policy allows returns within 30 days of purchase.

0.92

Shipping costs are non-refundable for returned items.

0.87

Product catalog updated quarterly with new items.

0.31

"return a product" matched "refund policy" and "returned items" — no keyword overlap needed!

Weakness: Struggles with exact identifiers. "error code XJ-4521" might not find the right doc because codes have no semantic meaning.

Cosine Similarity

Measuring how similar two vectors are

KAUST Academy

Cosine similarity measures the **angle** between two vectors, not the distance. Vectors pointing in the same direction are similar, regardless of length.



Identical Meaning

Vectors point in the same direction (0°)



Unrelated

Vectors are perpendicular (90°)



Opposite Meaning

Vectors point in opposite directions (180°)

Formula: $\cos(\theta) = (A \cdot B) / (||A|| \times ||B||)$

Range: -1 (opposite) to +1 (identical). In RAG, relevant matches typically score 0.6–0.95.

Approximate Nearest Neighbor (ANN)

How vector databases search at scale

KAUST Academy

Computing cosine similarity against every vector would be $O(n)$ — far too slow with millions of vectors. **Vector databases solve this with ANN algorithms.**

Brute Force (Exact)

Compare query against every single vector.

Perfect recall (100%)
But $O(n)$ complexity
1M vectors → seconds per query

ANN (e.g. HNSW)

Build an index structure. Search a smart subset.

~97–99% recall
 $O(\log n)$ complexity
1M vectors → milliseconds per query

How HNSW Works (Simplified)

Build a multi-layer graph. Top layer is sparse (long-range jumps). Bottom layer is dense (all data). To search: start at the top, greedily navigate to the closest node, drop down a layer, repeat. Like zooming into a map — first find the region, then the exact location.

"Approximate" = might miss the absolute best match, but finds very close ones orders of magnitude faster.

Top-K Selection: How Many Chunks?

Balancing completeness vs noise

KAUST Academy

After similarity search, you return the K most similar chunks. How do you choose K?

K Too Small (e.g. K=1)

- Might miss critical information
- If top result is wrong, no backup
- Incomplete answers

K Too Large (e.g. K=50)

- Irrelevant chunks dilute good ones
- Wastes context window and cost
- LLM may get confused by noise

Sweet spot: Start with K=3–5. Increase if answers feel incomplete, decrease if noisy. **With a reranker: retrieve K=20–50, then rerank down to 3–5.**

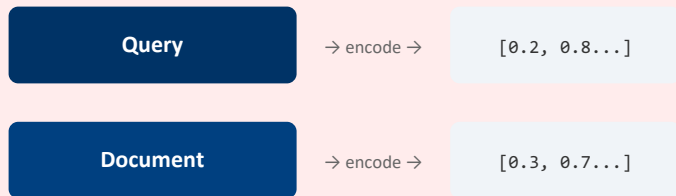
Also consider a similarity threshold (e.g. only return chunks with score > 0.7). Prevents returning irrelevant results even when K allows it.

Reranking: Why Initial Retrieval Isn't Enough

The bi-encoder vs cross-encoder problem

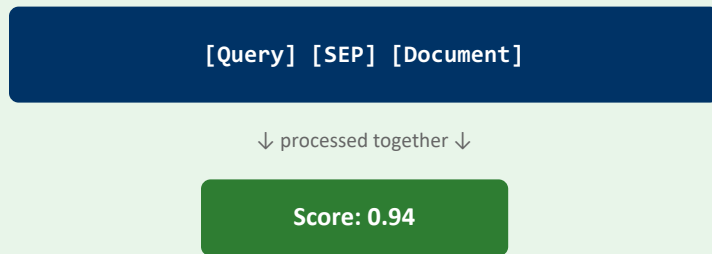
KAUST Academy

Bi-Encoder (Vector Search)



Encoded separately.
Only compared after.
Fast but shallow.

Cross-Encoder (Reranker)



Query and doc see each other.
Deep token-level interaction.
Slow but precise.

The strategy: Bi-encoder retrieves 20–50 candidates fast. Cross-encoder reranks them precisely. You get speed AND accuracy.

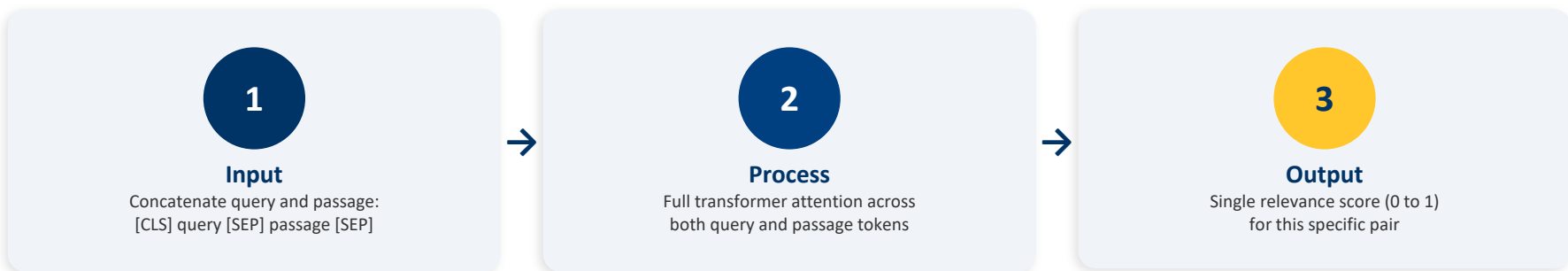
Reranker Models: A High-Level View

How cross-encoders are trained and what they do

KAUST Academy

A reranker is a cross-encoder model trained specifically to judge whether a (query, passage) pair is a good match.

What Happens Inside



Why This Is Better Than Bi-Encoders

Bi-encoder: Query and document never "see" each other during encoding. Similarity is a rough comparison of two independent summaries.

Cross-encoder: Every query token attends to every document token. The model understands the relationship directly.

Trade-off: Cross-encoders are ~100x slower than bi-encoders. That's why we only use them on 20–50 candidates, not millions.

Reranking: Worked Example

Query: "What is our refund policy?"

KAUST Academy

Step 1: Vector search returns Top-5 candidates (fast, imprecise)

#1	General company overview and history	Vec: 0.82	Cross: 0.12	×
#2	Refund policy: returns within 30 days with receipt	Vec: 0.79	Cross: 0.96	✓
#3	Q3 marketing campaign results	Vec: 0.77	Cross: 0.08	×
#4	Return shipping instructions and labels	Vec: 0.75	Cross: 0.89	✓
#5	Office holiday schedule 2024	Vec: 0.74	Cross: 0.05	×

Step 2: Cross-encoder rescores → reorder by cross-encoder score

#1	Refund policy: returns within 30 days with receipt	0.96
#2	Return shipping instructions and labels	0.89
#3	General company overview and history	0.12

Relevant results jumped from #2 and #4 to #1 and #2. Noise dropped out.

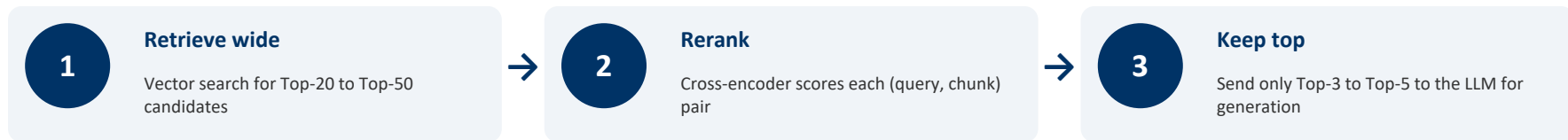
Reranking: Models & Practical Guidance

Choosing the right reranker

KAUST Academy

Model	Type	Latency	Best For
Cohere Rerank v3	API	~200ms	Production, easy setup
cross-encoder/ms-marco-MiniLM	Open source	~100ms	Self-hosted, English
bge-reranker-v2-m3	Open source	~150ms	Multilingual, high quality
Jina Reranker v2	API / Open	~180ms	Long context support

Practical Pattern



Impact: Reranking is one of the highest-leverage improvements you can make. Adds ~100–500ms but dramatically improves answer quality.

Keyword Search (BM25)

Classic information retrieval with exact matching

KAUST Academy

BM25 scores documents by how often query terms appear, adjusted for document length and term rarity. No embeddings needed.

Example

Query: "error code XJ-4521"

Error XJ-4521: Connection timeout on port 443. Reset the gateway.

8.7

Exact term match on "XJ-4521"

Error code reference table: XJ-series codes indicate network issues.

6.2

Partial match on "error code" + "XJ"

How to troubleshoot connection timeouts in production.

1.1

No keyword overlap with "XJ-4521"

BM25 found the exact error code that semantic search would miss — "XJ-4521" has no meaning, just characters.

Strengths: Exact keyword match. Fast. No model needed.

Weaknesses: "car" won't match "automobile". No meaning.

Hybrid Search

Combining semantic + keyword search

KAUST Academy

Neither semantic nor keyword search is perfect alone. Hybrid search runs both and merges the results.

Semantic search alone

Finds "refund policy" when you search "return a product"
But misses exact codes like "XJ-4521"

Keyword search alone

Finds exact "XJ-4521" matches perfectly
But misses synonyms — "car" won't find "automobile"

How Hybrid Search Works



Recommendation: Hybrid search is almost always better than either approach alone. Most modern vector DBs (Weaviate, Qdrant, Pinecone) support it natively.

Hybrid Search: Worked Example

Query: "error code XJ-4521 connection timeout"

KAUST Academy

Semantic Search Results

- | | | |
|----|-------------------------------------|----------------|
| #1 | Connection timeout troubleshooting | $1/61 = 0.016$ |
| #2 | Network connectivity best practices | $1/62 = 0.016$ |
| #3 | Server response optimization | $1/63 = 0.016$ |

BM25 Results

- | | | |
|----|------------------------------------|----------------|
| #1 | Error XJ-4521: Known issues log | $1/61 = 0.016$ |
| #2 | Connection timeout troubleshooting | $1/62 = 0.016$ |
| #3 | Error code reference table | $1/63 = 0.016$ |

▼ Merge with RRF: sum scores for results appearing in both lists

- | | | | |
|----|-------------------------------------|--------------|---|
| #1 | Connection timeout troubleshooting | 0.033 | <i>In BOTH lists → highest combined score</i> |
| #2 | Error XJ-4521: Known issues log | 0.016 | <i>BM25 caught the exact error code</i> |
| #3 | Network connectivity best practices | 0.016 | <i>Semantic found the meaning match</i> |

Metadata Filtering

Narrow the search space before comparing vectors

KAUST Academy

Before or during vector search, filter out irrelevant chunks based on metadata.

Why It Matters

100,000 chunks across 5 years and 10 departments. No reason to search financial reports from 2020 when asking about 2024 HR policies.

Examples

"Current refund policy"

→ `filter: source="policies", date ≥ 2024`

"Python code examples"

→ `filter: category="code", lang="Python"`

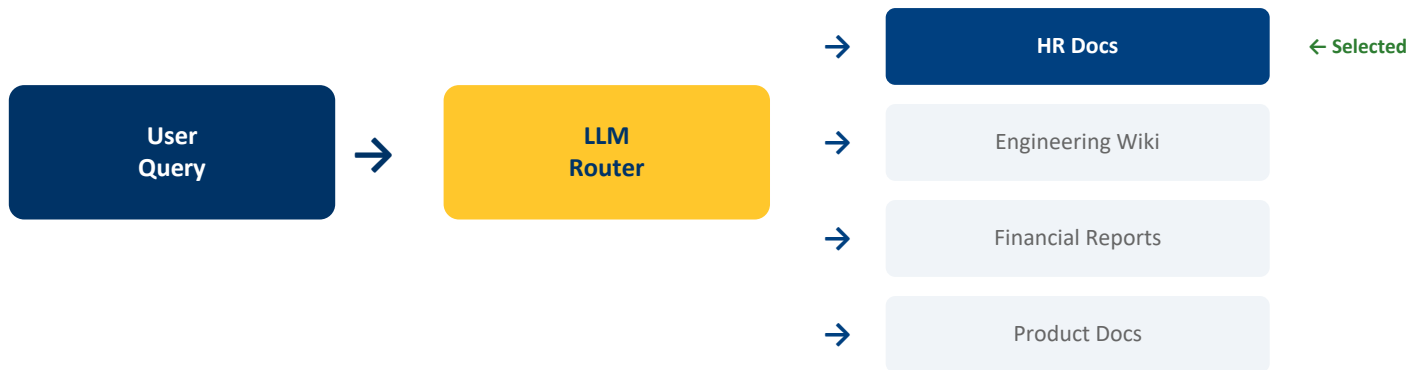
Pre-filter vs Post-filter: Pre-filtering (before vector search) is more efficient — it reduces the search space first.

Query Routing

Deciding where to search before you search

KAUST Academy

In production, you often have **multiple knowledge bases**. An LLM or classifier decides which one(s) to search.



Examples

How many vacation days do I have?

→ HR Docs

Why is the API returning 500 errors?

→ Engineering Wiki

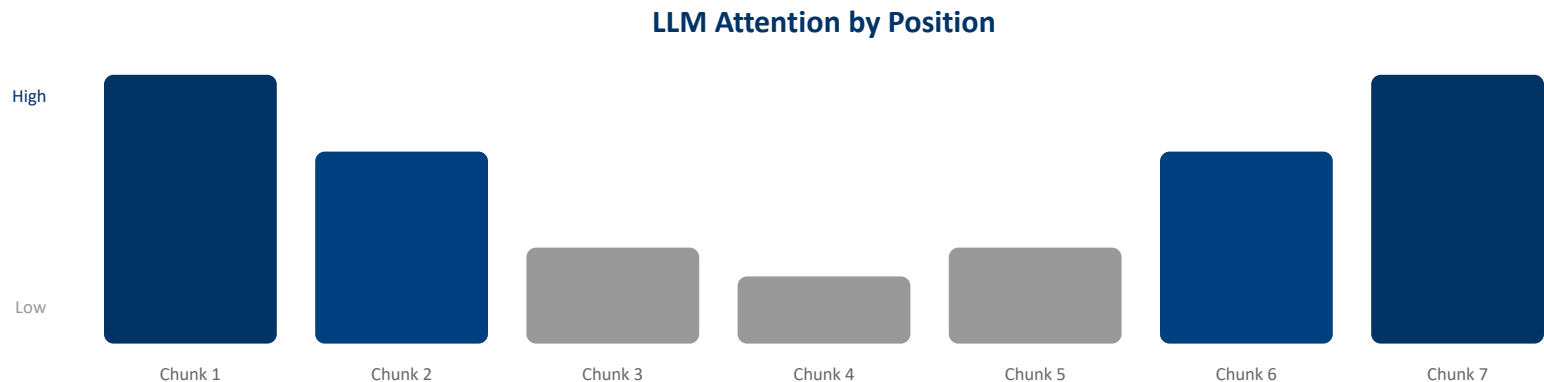
Why: Searching the wrong index wastes time and returns irrelevant results.

The "Lost in the Middle" Problem

How chunk ordering affects LLM attention

KAUST Academy

Research shows LLMs pay more attention to the beginning and end of context, ignoring the middle.



What To Do

Place the most relevant chunks first and last in the prompt. Put less critical chunks in the middle. This simple reordering can measurably improve answer quality.

CHAPTER 5

Generation

Prompt construction with retrieved documents

RAG prompts

Grounding

Citations

Edge cases

Conversational RAG

The Generation Stage

The final step: turning retrieved context into an answer

KAUST Academy

After retrieval gives you the top chunks, you construct a prompt that includes the user's question and the retrieved context, then send it to the LLM.

Three Key Decisions

1

How to structure the prompt

Where to place context, instructions, and the question

2

How to ground the answer

Strict (only context) vs permissive (context + knowledge)

3

How to handle edge cases

Junk retrieval, contradictions, out-of-scope queries

Remember: The LLM only sees what you put in the prompt. If the right chunk wasn't retrieved, or if it's buried in noise, the LLM can't produce a good answer. **Generation quality is capped by retrieval quality.**

RAG Prompt Anatomy

The specific structure of a RAG prompt

KAUST Academy

A RAG prompt has 4 distinct parts:

System Instructions

Role, behavior, grounding rules

You are a helpful assistant. Answer ONLY based on the provided context.

Retrieved Context

Numbered chunks with source metadata

[1] Returns accepted within 30 days... (Source: handbook.pdf, p.12)
[2] Shipping costs are non-refundable... (Source: faq.md, §3)

User Question

The original query

What is our refund policy?

Response Instructions

Format, citation rules, fallback

Answer concisely. Cite sources using [1], [2]. If unsure, say "I don't have that information."

Order matters: System instructions first, then context, then question. This structure helps the LLM distinguish between instructions, evidence, and the task.

Strict vs Permissive Grounding

How much should the LLM rely on context?

KAUST Academy

The grounding instruction tells the LLM whether to answer only from context or also use its own knowledge.

Strict Grounding

"Answer ONLY based on the provided context. If the context doesn't contain the answer, say so."

Use when: Legal, medical, financial domains. When accuracy and traceability matter more than completeness.

Risk: Says "I don't know" even when the answer is obvious general knowledge.

Permissive Grounding

"Use the provided context as your primary source. You may supplement with general knowledge, but clearly distinguish between the two."

Use when: Internal assistants, educational tools. When helpfulness matters more than strict traceability.

Risk: LLM may hallucinate and mix its knowledge with retrieved facts.

Recommendation: Start strict. Loosen only if users consistently hit dead ends on questions the LLM could easily answer.

Citation & Source Attribution

Making answers verifiable

KAUST Academy

Users need to verify answers. *"According to our Q3 report, revenue grew 15% [Source: Q3_Report.pdf, p.12]"* is far more trustworthy than just *"Revenue grew 15%."*

How to Implement

1

Number each chunk in the prompt

[1] chunk text (Source: file.pdf, p.12)
[2] chunk text (Source: faq.md, §3)

2

Instruct the LLM to cite

"After each claim, cite the source using [1], [2], etc."

3

Post-process citations

Convert [1], [2] references into clickable links in your UI

Example LLM Output

Our refund policy allows returns within 30 days of purchase [1]. Shipping costs are non-refundable [2]. You'll need the original receipt to process the return [1].

When Retrieval Fails

Handling junk results, contradictions, and out-of-scope queries

KAUST Academy

Junk Retrieval

None of the retrieved chunks actually answer the question. If you still pass them to the LLM, it may hallucinate.

Solution: Set a similarity threshold (e.g. > 0.5). Instruct: "If context is irrelevant, say I don't have that information."

Contradicting Chunks

Two retrieved chunks say different things. Which one is right?

Solution: Use metadata (prefer newer docs, higher-authority sources). Instruct the LLM to flag contradictions explicitly.

Out-of-Scope Query

The user asks something your knowledge base simply doesn't cover.

Solution: Route to a human, suggest related questions, or offer to search a different knowledge base.

Best practice: Always have a graceful fallback. A confident wrong answer is worse than admitting uncertainty.

Temperature & Parameters for RAG

Tuning generation for grounded answers

KAUST Academy

RAG generation has different parameter needs than creative writing or open-ended chat.

Parameter	RAG Recommendation	Why
Temperature	0.0 – 0.3	Lower = more deterministic, less hallucination. The model sticks closer to the retrieved context.
Top-P	0.9 – 0.95	Slightly restrict sampling to avoid unlikely tokens
Max Tokens	Set based on expected answer length	Prevent runaway generation. 200–500 for Q&A, more for summaries.
Frequency Penalty	0.0 – 0.3	Slight penalty reduces repetition without harming faithfulness

Conversational RAG

Multi-turn conversations with retrieval

KAUST Academy

Problem: Follow-up questions are often ambiguous. "What about the deadline?" only makes sense with conversation history. How do we handle this?

User: "What is our refund policy?"

Assistant: "Returns within 30 days with receipt..."

User: "What about the deadline?" ← meaningless as a standalone search query!

Three Strategies

1

Push Full History

Include the entire conversation in the prompt alongside retrieved chunks.

✓ Simple, no extra LLM calls

✗ Eats context window fast

Best for: Short conversations (< 5 turns)

2

Summarize History

LLM summarizes conversation so far into a condensed paragraph after each turn.

✓ Fixed-size context usage

✗ Loses nuance from earlier turns

Best for: Long conversations, tight context

3

Query Reformulation

LLM rewrites each follow-up into a standalone query using chat history.

✓ Best retrieval quality

✗ Extra LLM call per turn

Best for: Most production systems (recommended)

Query Reformulation in Detail

The recommended approach for conversational RAG

KAUST Academy

The LLM reads the conversation history and rewrites the ambiguous follow-up into a clear, standalone search query.

Example

Conversation History

User: What is our refund policy?

AI: Our policy allows returns within 30 days with original receipt. Refunds are processed in 5-7 business days.

User: What about the deadline?



LLM Reformulation

Original query:

"What about the deadline?"

↓ Reformulated to:

"What is the deadline for submitting a refund request under the return policy?"

Conversational RAG Pipeline



Output Formatting & Guardrails

Structuring responses and protecting sensitive data

KAUST Academy

Output Formatting

Inline citations

Answer text with [1], [2] references woven in. Best for human reading.

Answer + Sources block

Answer first, then a separate sources section. Clean separation.

Structured JSON

For programmatic use — APIs, chatbots, downstream processing.

Production tip: Test your guardrails adversarially. Try to get the system to leak PII, go off-topic, or hallucinate. If you can break it, your users will too.

Guardrails & Safety

PII Protection

Instruct: "Never reveal personal information like SSN, email, or phone numbers from the context."

Scope Limiting

Instruct: "Only answer questions related to [your domain]. Decline unrelated requests politely."

Confidence Signals

Instruct: "If you're uncertain, say so. Prefix with 'Based on available documents...'"

No Speculation

Instruct: "Do not speculate beyond what the documents state. Avoid phrases like 'probably' or 'I think'."

Be minimalist

Build your RAG to give only information needed all the time. For example, if the personal info is not critically needed to answer, then don't include it in the context

CHAPTER 7

Advanced RAG Techniques

When basic retrieve-and-generate isn't enough

Query Transform

HyDE

Parent-Child

Self-Query

Beyond

Umm.. I mean it is alright?

Reach for these only when the basics fall short

KAUST Academy

Golden rule: Always start with basic RAG. Only adopt these techniques **when you've identified a specific failure**. Complexity has real costs — latency, tokens, debugging.

Common failure modes of basic RAG:



Vague queries

"Tell me about that thing with HR" — no content words to match on



Vocabulary mismatch

User says "He is cooked"; doc says "terminated employment"



Small chunk = no context

Precise match, but the LLM can't understand it alone since most of the time it won't be informative



Structured constraints

"HR policies related to travelling outside of Saudi Arabia from 2024" — needs filters, not just search

Each failure has a matching technique. We'll cover them in order of how often they're needed.

Query Transformation

Rewrite the question before searching

KAUST Academy

Ask an LLM to reshape the user's query into something that retrieves better. Three common patterns:

Rewriting

Rephrase into better search terms

BEFORE

"what did the ceo say about layoffs pls not me"

AFTER

"CEO statement on workforce reductions"

Decomposition

Split complex questions into parts

BEFORE

"Compare Q2 and Q3 revenue"

AFTER

Q1: Q2 revenue figures
Q2: Q3 revenue figures
Q3: Factors causing change

Multi-Query

Generate N paraphrases; search all

BEFORE

"green tea benefits"

AFTER

"medical benefits of green tea"
"green tea wellness effects"
"green tea health advantages"

Cost: Extra LLM call before retrieval. Worth it when queries are messy or vocabulary mismatches are common.

HyDE: Hypothetical Document Embeddings

Search with a fake answer, not the question

KAUST Academy

The asymmetry problem: Queries are short questions. Documents are long declarative passages. They live in different parts of embedding space.

The HyDE trick

1

User asks

"What's our refund policy?"

2

LLM generates fake answer

"Our refund policy allows returns within 30 days of purchase..."

3

Embed the fake answer

Not the question — the hypothetical answer

4

Search vector DB

Find real chunks similar to the fake one

Why it works

The hypothetical answer is a document-like passage. Its embedding lands in the right neighborhood — close to real document chunks.

Caveat: The fake answer might be wrong. That's fine — we only use its *embedding*, not its content.

We never use the FAKE answer, we only get the closest vector to it in the embedding space, then we throw it away

Best for: Short, vague queries where the user's phrasing doesn't match how the information is written in your documents.

Parent-Child Chunking

Search with small chunks, answer with large ones

KAUST Academy

The core idea: A small chunk wins the *search* (it's focused and most likely have has 1 certain information, embedding is precise). A large chunk wins the *answer* (it has full context).
So: store each small chunk with a pointer to its parent large chunk. Search using the small one but hand the LLM the big one.

Example

PARENT · ~1000 tokens · sent to LLM

"Employee Handbook § 4.3 — Parental Leave"

Child 1 · eligibility rules · ~200 tokens · embedded

Child 2 · ★ duration & pay · ~200 tokens · embedded

Child 3 · how to apply · ~200 tokens · embedded

What happens at query time

1

Query: "how much paid leave?"

Embed and search the children

2

Best match: Child 2 (★)

It's specifically about duration & pay

3

Look up Child 2's parent

Retrieve the full § 4.3 section

4

Send parent to LLM

Full context → confident, grounded answer

Result: Precision of small chunks + richness of large chunks. One of the most effective fixes in production RAG.

Self-Querying Retrieval

Extract filters from natural language automatically

KAUST Academy

Users mix **semantic intent** with **structured constraints**. An LLM splits them apart — typically by emitting structured JSON that we parse.

USER QUERY

"Show me HR policies updated after January 2024 about remote work"

Step 1 — Prompt the LLM

```
Schema: { query: str,
          filters: {
            department?: str,
            date_after?: date } }
```

User: "Show me HR policies
updated after Jan 2024
about remote work"

Return JSON only.

Step 2 — Parse the JSON

```
{
  "query": "remote work policies",
  "filters": {
    "department": "HR",
    "date_after": "2024-01-01"
  }
}
```

Step 3 — Execute: Pass **filters** to the vector DB as a pre-filter, then do semantic search on **query**. Tools: LangChain [SelfQueryRetriever](#), OpenAI function calling, or any structured-output mode.

When to Use What

A diagnostic guide — pick the fix that matches the symptom

KAUST Academy

Debug your RAG like you'd debug any system: reproduce the failure, identify the symptom, then apply the targeted fix.

Symptom you see	Root cause	Apply	Cost · Priority
Retrieval fine but query words don't appear in any doc	<i>Vocabulary mismatch — user slang vs doc formality</i>	Query Transformation	1 extra LLM call · ~200ms Try first
Short queries return top-K with very low similarity scores	<i>Query is too short/vague for doc-style embeddings</i>	HyDE	1 LLM call before search If vocab fix fails
Right chunk retrieved but LLM answers incompletely	<i>Chunk too small to contain full context</i>	Parent-Child Chunking	Re-index with 2 sizes High impact
User asks with time/category constraints, filter ignored	<i>No automatic metadata extraction from query</i>	Self-Querying	1 LLM call + schema design Required for filters

Stacking: Techniques compose. Parent-Child + Self-Querying is a common combo. But add one at a time and measure each change before stacking more.

CHAPTER 8

Evaluating Your RAG Pipeline

You can't improve what you don't measure

Retrieval metrics

Generation metrics

LLM-as-a-Judge

Diagnosis

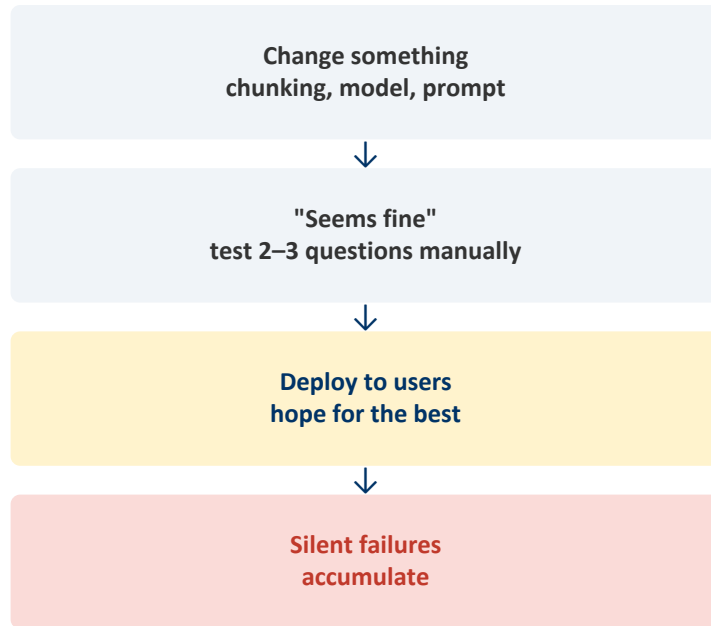
Why Evaluation Matters

The "it seems to work" trap

Most teams deploy RAG systems and never systematically evaluate them.

They rely on "it seems to work" until a user reports a bad answer. By then, you don't know how many other bad answers slipped through.

Without evaluation, every change is a **guess**, not an improvement.



The Two Failure Modes

Where things go wrong in RAG



Retrieval Failure

The right document exists in your knowledge base, but the retriever didn't find it.

The LLM never saw the correct information.



Generation Failure

The retriever found the right document, but the LLM misinterpreted it, ignored it, used wrong chunk, or hallucinated on top of it.

Key insight: These require completely different fixes. Evaluate each component separately.

Retrieval Metric: Context Precision

Of what you retrieved, how much was actually useful?

Example — "What is our parental leave policy?"

Retrieved Chunk	Relevant?
"Employees get 12 weeks..."	✓
"Company holiday schedule..."	✗
"Parental leave can be..."	✓
"Health insurance covers..."	✗
"To apply for leave..."	✓

$$\text{Precision} = 3/5 = 60\%$$

Low precision means...

Your retriever returns too much noise. The LLM wastes context window space on irrelevant content.

How to improve:

Better chunking, reranking with a cross-encoder, metadata filtering.

Retrieval Metric: Context Recall

Did you find everything that was relevant?

Example — 4 relevant chunks exist in knowledge base:

Relevant Chunk	Retrieved?
Policy overview	✓
Eligibility requirements	✗
How to apply	✓
Leave extension options	✗

$$\text{Recall} = 2/4 = 50\%$$

Low recall means...

Your retriever is missing important information. The LLM's answer will be incomplete.

How to improve:

Increase K, use hybrid search, multi-query retrieval.

The trade-off: Retrieving more chunks increases recall but can decrease precision. Balance is key.

Retrieval Metrics: MRR & Hit Rate

Standard information retrieval metrics

Hit Rate (Recall@K)

For each query: is the correct answer anywhere in the top-K retrieved chunks?

Query 1: correct at rank 3 → **Hit**

Query 2: not in top-5 → **Miss**

Query 3: correct at rank 1 → **Hit**

Hit Rate = $2/3 = 67\%$

Simple binary check. Good for quick health checks.

Mean Reciprocal Rank (MRR)

Where does the first correct result appear? Higher rank = better.

Query 1: first correct at rank 3 → $1/3$

Query 2: not found → 0

Query 3: first correct at rank 1 → $1/1$

MRR = $(0.33 + 0 + 1) / 3 = 0.44$

Rewards putting the right answer first.

Generation Metric: Faithfulness

Is the answer grounded in the retrieved context?

How it works:

1. Extract individual claims from the answer
2. Check each claim against retrieved context
3. Score: supported claims / total claims

Example

Context: "Employees get 12 weeks paid leave after 1 year."

Claim	
12 weeks paid leave	✓
Need 1 year tenure	✓
Extend by 6 months	Hallucinated!

$$\text{Faithfulness} = 2/3 = 67\%$$

How to improve: Stronger grounding instructions, more instruction-following LLM, lower temperature.

Generation Metric: Answer Relevance

Does the answer address what was actually asked?

How it's measured:

1. Take the generated answer
2. Generate hypothetical questions it would answer
3. Compare to original question (embedding similarity)
4. High similarity = relevant answer

Example

Question: "What is the deadline for leave applications?"

Answer: "Our parental leave policy provides 12 weeks of paid leave for eligible employees."

The answer is about parental leave in general — not the deadline. It's faithful but **not relevant**.

Relevance $\approx$ 0.35 (low)

LLM-as-a-Judge

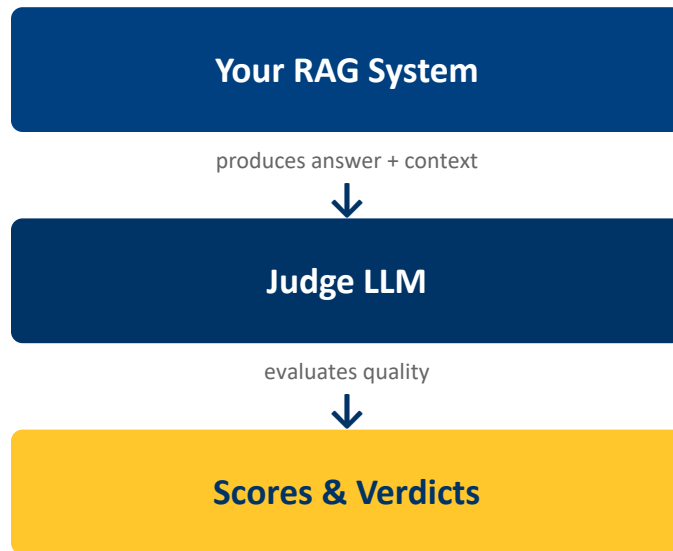
Using one LLM to evaluate another

The Problem

Metrics like faithfulness and relevance require understanding meaning — not just string matching. You can't compute them with simple code.

The Solution

Use a powerful LLM (the "judge") to evaluate your RAG pipeline's output. Prompt it with the question, context, and answer, then ask it to score.



Why this works: LLMs achieve 80–90% agreement with human evaluators — much cheaper and faster.

The Four Key RAG Metrics

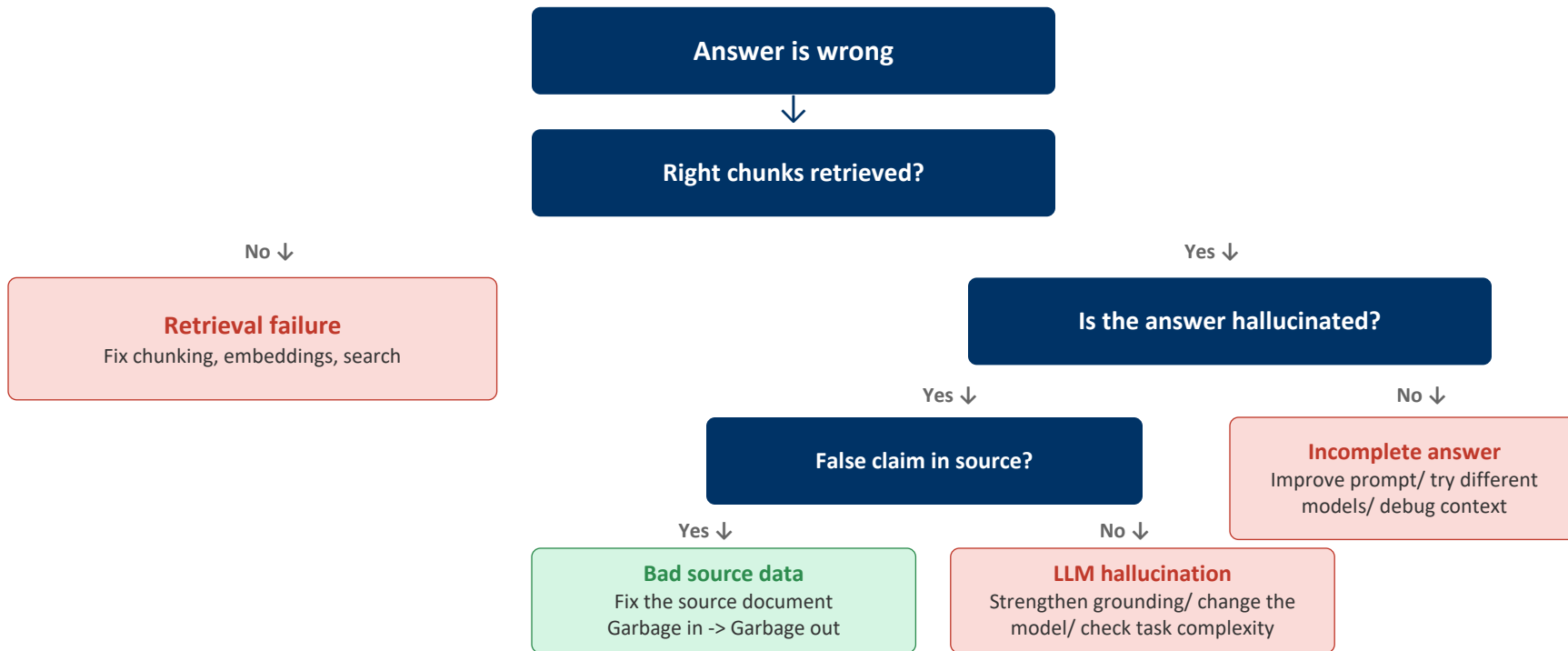
Complete picture of pipeline health

Metric	Evaluates	Question It Answers	If Low, Fix...
Context Precision	Retriever	Are retrieved chunks relevant?	Chunking, reranking, filters
Context Recall	Retriever	Did we find all relevant chunks?	Increase K, hybrid search
Hit Rate (Recall@K)	Retriever	Is the correct in the retrieved chunks?	Increase K, reranking, chunking
Mean Reciprocal Rank (MRR)	Retriever	Where does the first correct result appear?	Reranking, filtering, hybrid search
Faithfulness	Generator	Is the answer grounded in context?	Grounding prompt, lower temp
Answer Relevance	Generator	Does answer address the question?	Better retrieval, query rewrite

Metrics 1 & 2 & 3 & 4 evaluate your retriever. Metrics 5 & 6 evaluate your generator. Together they give you a complete picture.

Diagnosing RAG Failures

A unified decision tree



Why You Need a Test Dataset

The foundation of reliable evaluation

Without a Test Dataset

- ✗ You test 2–3 questions manually and call it done
- ✗ You can't tell if a change helped or hurt
- ✗ Bugs slip through silently to users
- ✗ Every deployment is a gamble
- ✗ You discover failures from angry users, not metrics

With a Test Dataset

- ✓ Every change is measured against the same benchmark
- ✓ Problems are caught before deployment
- ✓ You know exactly where the pipeline fails
- ✓ Confidence to ship updates quickly
- ✓ Data-driven decisions, not gut feelings

Test Dataset Structure

What each entry looks like — with a real example

Field	Description	Example
Question	The user's natural language query	<i>"What is our parental leave policy for new fathers?"</i>
Ground Truth	The correct, complete answer	<i>"12 weeks paid leave after 1 year tenure, applies equally to all parents..."</i>
Source Docs	Which documents contain the answer	<i>hr_policies_2024.pdf, page 14–15</i>
Difficulty	Easy / Medium / Hard	<i>Medium (answer spans two paragraphs)</i>
Category	Topic area for analysis	<i>HR / Benefits</i>

Another example entry:

Field	Value
Question	"What error code does the API return when the rate limit is exceeded?"
Ground Truth	"HTTP 429 Too Many Requests, with a Retry-After header indicating wait time in seconds."
Source / Difficulty	api_reference_v3.md, section 4.2 Easy (single sentence answer)

How to Build Your Test Dataset

Three approaches from gold standard to scalable

1

Manual Creation — Subject Matter Experts

Domain experts read your actual documents and write question-answer pairs. This is the gold standard for quality — every QA pair is accurate and meaningful. Best for your initial 50–100 examples.

2

LLM-Generated — Scale with Human Review

Feed your documents to an LLM and ask it to generate questions a user might ask. Then have a human verify each QA pair for accuracy. Faster than manual, but requires careful review — LLMs can generate plausible but wrong answers.

3

Log-Based — From Real User Questions

Collect actual user questions from your production logs. Have humans annotate the correct answers. Most realistic — these are the questions users actually ask, including edge cases you'd never think of.

Recommendation: Start with Method 1 for quality, then scale with Method 2, then continuously improve with Method 3.

Test Dataset Best Practices

Building a dataset that actually catches problems

How Many Examples?

50–100 well-curated examples is a strong starting point. Quality matters far more than quantity. A dataset of 50 carefully verified QA pairs beats 500 noisy ones every time.

Include Edge Cases

- Questions with no answer in your docs
- Ambiguous questions with multiple valid answers
- Questions requiring info from multiple docs
- Adversarial phrasing that might confuse retrieval

Difficulty Distribution

Mix difficulty levels deliberately:

- ~30% **Easy** (single chunk, direct answer)
- ~40% **Medium** (multi-chunk, needs synthesis)
- ~30% **Hard** (ambiguous, multi-hop reasoning)

Keep It Alive

- Add new examples from production failures
- Update ground truths when source docs change
- Cover new document types as they're added
- Review quarterly to remove stale entries

DAY 3 —

Thank You

From frozen knowledge to grounded, cited answers



CONTRIBUTED BY

Yazan Alshuaibi