RAG: Giving an LLM Your Own Knowledge

This guide takes you from the problem to a working retrieval pipeline. Read it in order. Every concept is explained the first time it appears, and each section ends with a plain-English summary you can reuse. You do not need the class video.

> **Running example.** A company has a 40-page leave policy. An employee asks: *"How does our leave policy work?"* We follow that one question from the document, through the database, to the answer.

1. Three problems with LLMs

A large language model learned from a big public snapshot of text, and then stopped learning. That causes three problems:

Problem	What it looks like	Example
1. It cannot see your private data	It has never read your files.	"How many leave days do I get?" It guesses a typical answer.
2. Knowledge cutoff	Its knowledge stops at a training date.	"What changed in last week's policy?" It cannot know.
3. Hallucination	It answers confidently, and the answer is wrong.	It invents a clause that does not exist.

Three ways to fix these are covered below: fine-tuning, in-context learning and RAG. They are not interchangeable, and choosing the wrong one is the most common mistake.

2. Fine-tuning: retraining the model

Fine-tuning means continuing the model's training on your own examples, so its internal weights change. Think of it as sending the student back to school.

Three types

- **Supervised fine-tuning (SFT):** you give input and ideal output pairs, and the model learns to copy that style and behaviour.

- **Continued pretraining:** the model keeps reading large amounts of new raw text in your domain, for example medical papers.
- **RLHF (reinforcement learning from human feedback):** people rank answers, and the model learns to prefer the ones people rank higher. This is how many chat models are tuned for helpfulness.

How supervised fine-tuning works, step by step

1. **Data collection:** gather hundreds to thousands of clean examples. Quality matters more than quantity.
2. **Method selection:** choose SFT, a parameter-efficient method, or RLHF, based on your goal.
3. **Training:** run the job on a training platform and watch the loss curve.
4. **Evaluation:** test on examples the model never saw, and compare with the base model.

What fine-tuning fixes

Problem	Solved?	Why
Private data	Solved, partly	The model learns patterns from your data, but it can still mix up facts and cannot cite a source.
Recent data	Partly	Every new fact needs another training run.
Hallucination	Partly	Tuning on good answers can reduce invention, but it does not guarantee correct facts.

3. In-context learning: teaching in the prompt

In-context learning means the model learns a pattern from examples inside the prompt, with **no change to its weights**. Nothing is retrained. The learning lasts only for that request.

Few-shot example: sentiment

```
Below are examples of text labelled with their sentiment.
Use the examples to determine the sentiment of the final text.

Text: "I love this phone. It's so smooth." → Sentiment: Positive
Text: "This app crashes a lot."            → Sentiment: Negative
Text: "The camera is amazing."             → Sentiment: Positive
Text: "I hate the battery life."           → Sentiment: ???
```

The model answers **Negative**. It learned the label pattern from three examples, with no training. This is called **few-shot prompting**. Zero examples is zero-shot, and one example is one-shot.

> **Limit:** every example costs input tokens on every call (see the tokens guide). In-context learning is cheap to try and expensive to scale when the examples are long.

4. RAG: retrieval-augmented generation

RAG means: before the model answers, a search step finds the most relevant passages from your documents, and those passages are placed into the prompt. The model then answers using both its general knowledge and the passages you supplied.

Use the open-book exam picture. An LLM with fine-tuning is a student who studied the textbook months ago. An LLM with RAG is a student who is allowed to open the textbook to the right page during the exam.

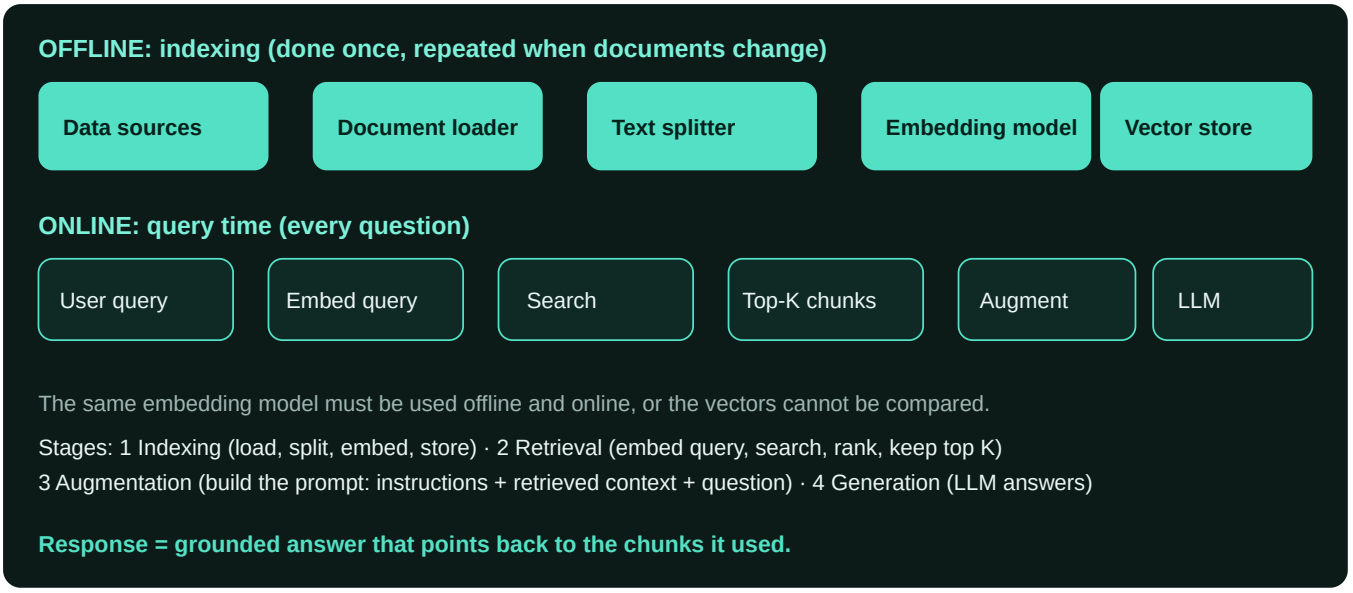


Figure 1. The complete RAG pipeline: an offline indexing half and an online query half.

5. Stage 1: indexing (the offline half)

Step 1: data sources and document loaders

A loader reads each source and turns it into plain text with metadata (file name, page, date, URL). Common sources: PDFs, web pages, YouTube transcripts, cloud storage folders, databases, CSV files. Keep the metadata: you will need it later to filter results and to cite sources.

Step 2: text chunking

A model cannot search a 40-page file as one unit. The text is cut into **chunks**. Each chunk needs to be small enough to be specific, and large enough to hold a complete thought.

Worked example: recursive character splitting

A recursive splitter tries the largest separator first, and only falls back to a smaller one when a piece is still too long:

1. Paragraph break (`\n\n`)
2. Sentence end (`. `)
3. Comma (`, `)
4. Word (space), and as a last resort single characters

Two settings control the result. **Chunk size** is the maximum length of each chunk. **Chunk overlap** is how many characters the next chunk repeats from the end of the previous one, so a sentence cut in half still appears whole in one of the chunks.

The class demo used a 519-character paragraph about machine learning, with chunk size 120 and overlap 15. The first chunk came out as *"Machine learning is a subset of artificial intelligence that focuses on building systems that learn from data"* (about 109 characters). The next chunk started with *"learn from data. Unlike traditional programming..."*, so the words "learn from data" appear in both chunks. Seven chunks came out in total.

Setting	Too small	Too large
Chunk size	Each chunk loses context; answers are fragments.	Search becomes vague; too much irrelevant text goes to the model.
Overlap	Ideas cut at a boundary get lost.	Many duplicate chunks; more storage and cost.

> **Starting point:** chunk size 300 to 800 tokens with 10 to 15 percent overlap for prose policies. Test on 20 real questions and adjust.

Step 3: embeddings: turning meaning into numbers

An **embedding** is a list of numbers (a **vector**, often hundreds or thousands of values) that represents the meaning of a piece of text. Texts with similar meaning get vectors that point in similar directions. An **embedding model** produces these vectors. Common choices include OpenAI's embedding models and open sentence-transformer models, and hosted options such as Google's and Hugging Face's models.

Rule: the same embedding model must be used for the chunks and for the questions. Vectors from different models are not comparable.

Step 4: the vector store

A **vector database** stores each chunk with three things: the chunk text, its embedding vector, and its metadata. It can run locally (for a laptop demo or a small internal tool) or in the cloud (for

shared, large or production systems). Examples include local libraries and managed cloud services; choose one by data size, privacy needs and budget.

6. Stage 2: retrieval (the online half)

Retrieval finds the right context for one question.

1. **Embed the query** with the same embedding model.
2. **Semantic search:** compare the query vector with every chunk vector, usually with **cosine similarity** (how closely two vectors point the same way; 1.0 is identical direction).
3. **Rank** the chunks by similarity.
4. **Keep the top K** chunks. K is the number of chunks you pass on, often 3 to 8.

Retrieval in action

Question: *"How does our leave policy work?"* The vector store holds chunks such as "Office dress code policy", "Annual leave and holiday rules", "Remote work equipment", "Leave policy: sick and casual", and "Employee onboarding steps". Their similarity scores to the question might be 0.91 for annual leave, 0.88 for sick and casual leave, and much lower for the rest. With K = 2, the two leave chunks are passed on. The search matched **meaning**, not the exact words "leave policy", which is why semantic search beats keyword search for questions phrased in different ways.

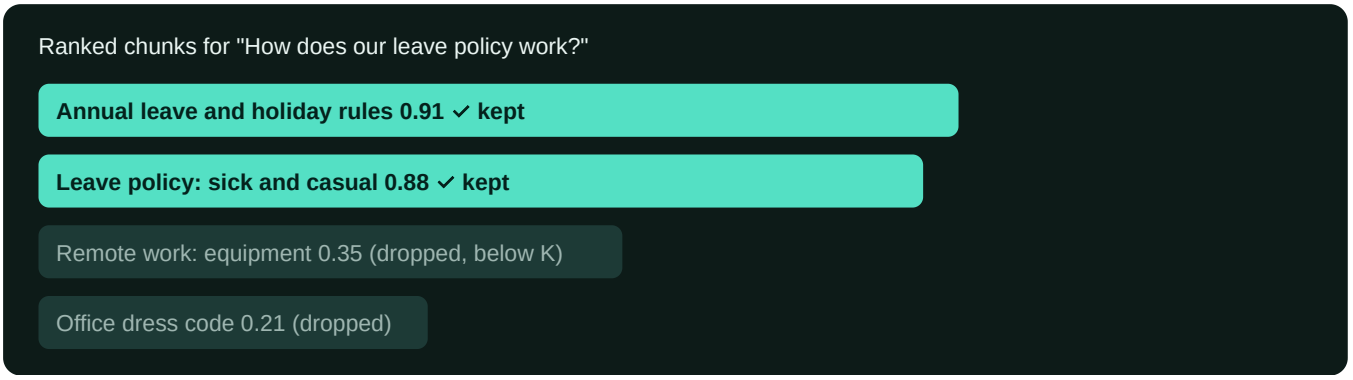


Figure 2. Retrieval keeps the top K. Scores are illustrative.

7. Stage 3: augmentation (building the prompt)

Augmentation is where you combine the instructions, the retrieved text and the question. This is the step that makes RAG work:

```
[System instruction]
You are a helpful assistant. Answer ONLY from the provided context.
If the context is insufficient, say "I don't know."
```

```
[Context]
{retrieved_chunks_text}

[Question]
{user_query}
```

Three rules make this safe. Put the rule about "answer only from context" in the system instruction. Label the context clearly so the model knows what is evidence. Ask the model to name the chunk it used. Without the instruction, the model will fall back to its general knowledge and you will not know which answers came from your documents.

8. Stage 4: generation

The LLM writes the answer from the augmented prompt. Good answers follow three checks: they are supported by the context, they say "I don't know" when the context is missing, and they cite the source. Log the question, the retrieved chunks and the answer, so you can review failures later.

9. How RAG solves the three problems

Problem	How RAG helps	What it does not fix
Private data	Your documents are searched at query time; the model sees them.	Access control: you must still decide who may see which document.
Knowledge cutoff	Re-index a document and the answer updates without retraining.	Re-indexing is still a pipeline you must run and monitor.
Hallucination	The answer is grounded in retrieved text, and you can check it.	Bad retrieval still produces bad answers, so you must measure retrieval.

10. RAG or fine-tuning? Questions and answers

Q11. A formal Hindi chatbot: RAG or fine-tuning?
Start with prompting for tone and language, which is cheap. Use RAG for facts such as policies and prices. Fine-tune only if the style still fails after good prompting and you have enough high-quality examples. The usual answer is RAG plus a clear system prompt, not fine-tuning.

Q12. Can you use RAG and fine-tuning together?
Yes. They do different jobs. Fine-tuning can teach a consistent style or format. RAG supplies

the facts for each question. Many production systems do exactly this, but add fine-tuning last, after RAG is measured.

Q13. RAG for real-time news from the last hour: suitable?
RAG can work only if the index is refreshed quickly enough, with a pipeline that loads and embeds new articles within minutes. Fine-tuning is the wrong tool, because it cannot keep up. Measure the delay between publication and availability in the index.

Q14. RAG over 500 PDFs gives partly correct answers. What is wrong?
Usually retrieval, not the model. Check: are the right chunks in the top K? If not, the chunks may be too large or too small, K may be too low, the PDFs may be scanned images with bad text extraction, or the metadata filter may be too narrow. Fix retrieval first; the model cannot answer from chunks it never received.

Q15. RAG works in English but fails in Hindi. Why?
The embedding model is probably not strong in Hindi, so Hindi questions and Hindi chunks land far apart in vector space. Use a multilingual embedding model, index the Hindi text in the same space, and test with Hindi questions. Check tokenization too: Hindi text often uses more tokens for the same meaning, which affects chunk size.

11. A test plan you can follow

1. Write 20 real questions and the answer you expect, with the source page.
2. For each question, log the top K chunks. Is the right source in the top K? (This is **retrieval accuracy**.)
3. Check whether the final answer is supported by the chunks. (This is **faithfulness**.)
4. Ask 3 questions the documents cannot answer. The correct response is "I don't know".
5. Change one setting at a time (chunk size, overlap, K, embedding model) and re-run the 20 questions.

12. Quick check

1. Which of the four RAG stages happens offline?
2. Why must the same embedding model be used for chunks and queries?
3. Name two things you should store with each chunk.
4. An answer is wrong, but the right chunk was in the top K. Is retrieval the problem?

Answers: 1. Indexing (load, split, embed, store). 2. Vectors from different models cannot be compared. 3. Its text and its metadata, and its vector. 4. No; the chunk was retrieved, so the problem is the prompt or the model's use of it.

13. Glossary and further reading

Term	Plain meaning
RAG	Search your documents first, then answer with the results in the prompt.
Chunk	A piece of a document small enough to search and send.
Embedding	A list of numbers that represents meaning.
Vector store	A database that stores chunks, their vectors and metadata.
Cosine similarity	How closely two vectors point the same way.
Top K	How many best-matching chunks are passed to the model.
Few-shot	Giving examples in the prompt so the model copies the pattern.
Fine-tuning	Retraining the model's weights on your examples.
Hallucination	A confident answer that is not supported by facts.

- Anthropic documentation on prompting and long-context use: docs.anthropic.com.
- Embedding model and vector database documentation from your chosen provider; check current options before you pick one.

13. Extension: evaluating a RAG system

Measure each stage separately. Most RAG failures come from retrieval, and you cannot fix what you do not measure.

Measure	Question it answers	How to measure simply
Retrieval recall	Is the right chunk in the top K?	For 20 questions, mark whether the source is in the top K (yes or no). Report the share of yes.
Grounding (faithfulness)	Is every claim supported by the retrieved text?	Read each answer; mark any sentence not found in the chunks.
Abstention	Does it say "I don't know" when the context is missing?	Ask 3 unanswerable questions; count correct abstentions.
Latency	How long does one answer take?	Time from question to answer, median of 10 runs.
Cost	What does one answer cost?	Input tokens (question plus K chunks) and output tokens, priced with the tokens-guide formula.

Failure modes and fixes

Symptom	Likely cause	First fix to try
Right document, wrong passage	Chunks too large or too small	Change chunk size; keep 10–15 percent overlap
Right passage not retrieved at all	K too small or wrong embedding model	Raise K; test another embedding model
Answer adds facts not in the context	Instruction too weak	Tighten the augmentation prompt; ask for citations
Hindi or regional-language questions fail	Embedding model not multilingual	Use a multilingual embedding model; test with native-language questions
Recent documents missing	Index not refreshed	Add an ingestion schedule; log the last index time
Slow answers	Too many chunks or a large model	Lower K; use the smallest model that passes

14. Diagram: the augmented prompt, labelled

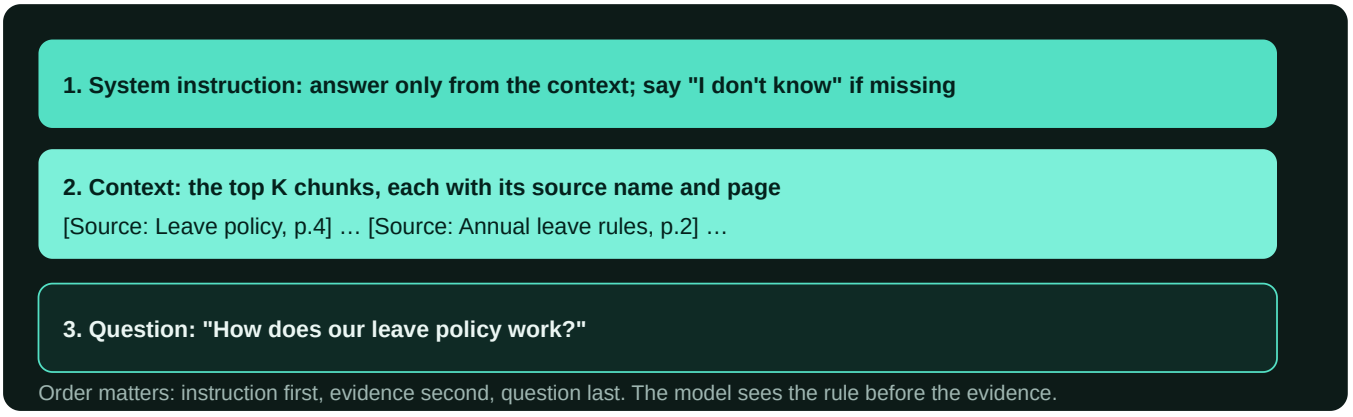


Figure 5. The three blocks of an augmented prompt.

15. Advanced: a retrieval test set you can reuse

```
question_id, question, expected_source, expected_answer_points, notes
Q01, "How many annual leave days do I get?", Annual leave rules p.2, "number of days; carry-over rule", normal
Q02, "Can I take sick leave without a note?", Sick and casual p.1, "notice rule; note rule", normal
Q03, "What is the office dress code?", Office dress code p.1, "should be retrieved, not leave", distractor check
Q04, "What is the CEO's salary?", none, "must say I don't know", abstention check
```

```
Q05, "Leave policy in Hindi", Leave policy p.1, "same answer as English",
multilingual check
```

Run the set after every change (chunk size, overlap, K, embedding model, prompt). Keep a log so you can see which change helped and which hurt.

16. Extended answer key and checklist

Item	Done
Test set of at least 20 questions with expected sources	[ ]
Retrieval recall measured before tuning the prompt	[ ]
Abstention tested with unanswerable questions	[ ]
Multilingual questions tested if your users need them	[ ]
Index refresh schedule set and logged	[ ]