READ MODE · ANSWER FIRST

RAG Query Lifecycle

An interactive, visual explanation of RAG Query Lifecycle, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

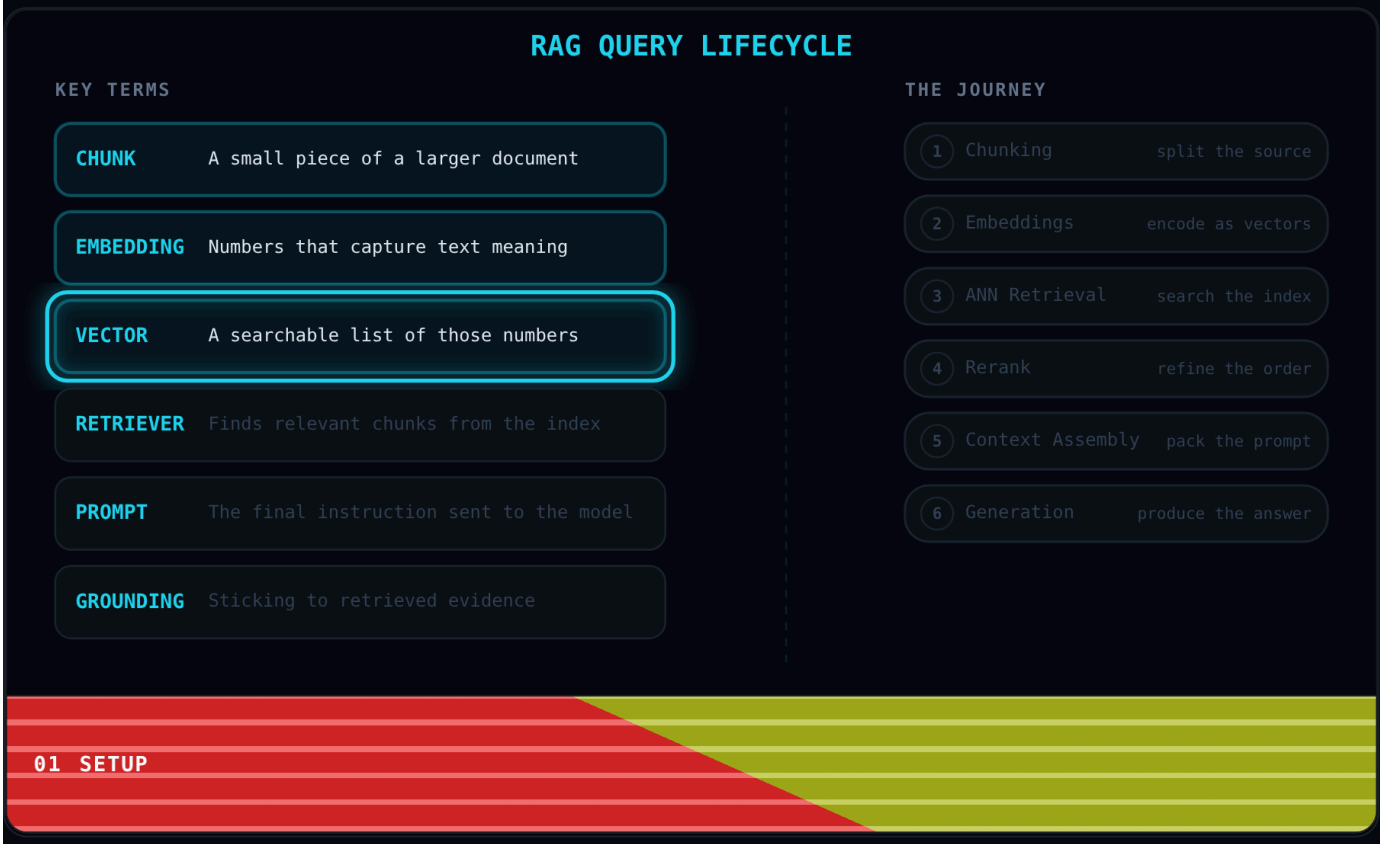
The whole story in 6 lines

RAG Query Lifecycle becomes easier to reason about when every stage is connected as one system.

1. Chunk boundaries decide what evidence survives retrieval. Too coarse loses precision; too fine loses context.
2. Embeddings turn language into geometry. The retriever will search near vectors, not near raw strings.
3. ANN trades exhaustive recall for latency. Probe depth and index shape control that tradeoff.
4. Reranking is the precision pass. It burns more model compute on fewer items to improve final context quality.
5. Context assembly is a packing problem. The best passage is useless if it never fits into the final prompt.
6. Generation is the visible result, but its quality is largely constrained by the upstream retrieval and assembly decisions.

STAGE 1 OF 8

Setup



Welcome. RAG stands for Retrieval-Augmented Generation. It is a way to make AI models answer questions using real documents instead of only their training data. Think of it like an open-book exam: the model gets to look things up before answering.

A chunk is a small piece of a larger document. We break documents into chunks because the model cannot read an entire library at once. Each chunk is small enough to search and compare.

An embedding is a list of numbers that captures the meaning of a piece of text. Two texts about the same topic will have similar embeddings, even if they use different words.

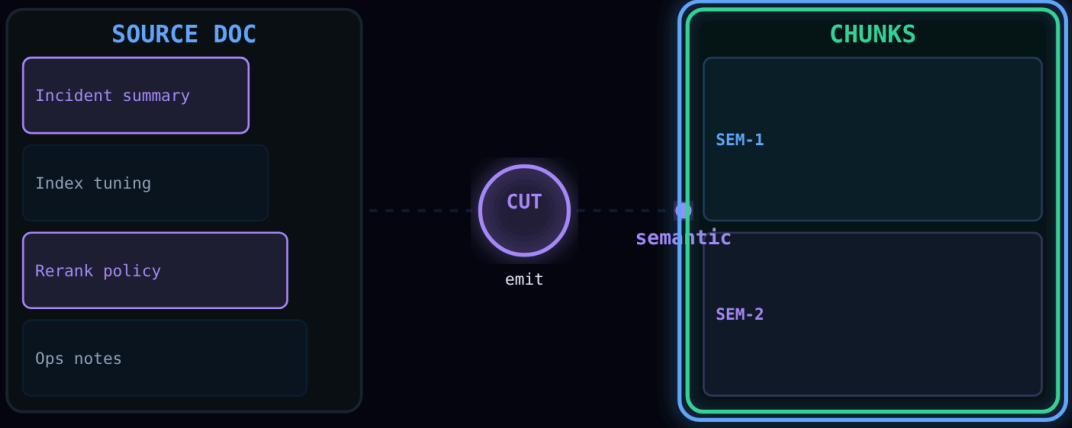
A vector is just another name for that list of numbers. A vector store is a database that holds these vectors and lets you search for the ones most similar to a query.

A retriever finds relevant chunks from the vector store, and a prompt is the final instruction sent to the model. Grounding means the model sticks to the retrieved evidence instead of making things up.

Over the next six stages we will follow a question from the moment it enters the system to the moment an answer comes out. Let us start with the first decision: how to break documents into chunks.

STAGE 2 OF 8

Chunking



02 CHUNKING

Here is a long source document entering the ingestion pipeline. It might be a runbook, an incident report, or a product manual. Right now it is one unbroken text stream, and nothing in this form is searchable yet.

The chunker scans the raw source and prepares to place boundaries. Think of it like a librarian deciding where to insert bookmarks so readers can jump to the right page later.

The chosen strategy decides where one passage ends and the next begins. Switch the Strategy control in the sidebar from Semantic to Fixed. Notice how the boundary markers shift to evenly spaced positions instead of topic breaks.

Chunks are emitted as separate retrieval records. Each one will get its own embedding in the next stage, so the size and content of each chunk directly affects what the retriever can find.

When overlap is enabled, neighboring chunks share a small token window. Toggle the Overlap control off and notice how the carry markers disappear. Overlap prevents edge facts from being lost between boundaries.

The chunk set is ready. These chunks are the raw material for everything downstream. Next, we will turn them into vectors so we can search by meaning instead of by keywords.

STAGE 3 OF 8

Embeddings

03 EMBEDDINGS

Now that we have chunks from the previous stage, we need a way to compare them to a question. Raw text comparison would miss synonyms and rephrasings. Embeddings solve that by encoding meaning as numbers.

The query text is encoded first so the retriever has a search vector. Remember how we said chunks get their own embedding? The query gets one too, using the exact same encoder model.

The encoder produces a dense query embedding. This is the list of numbers that captures what the question is really asking about, regardless of the exact words used.

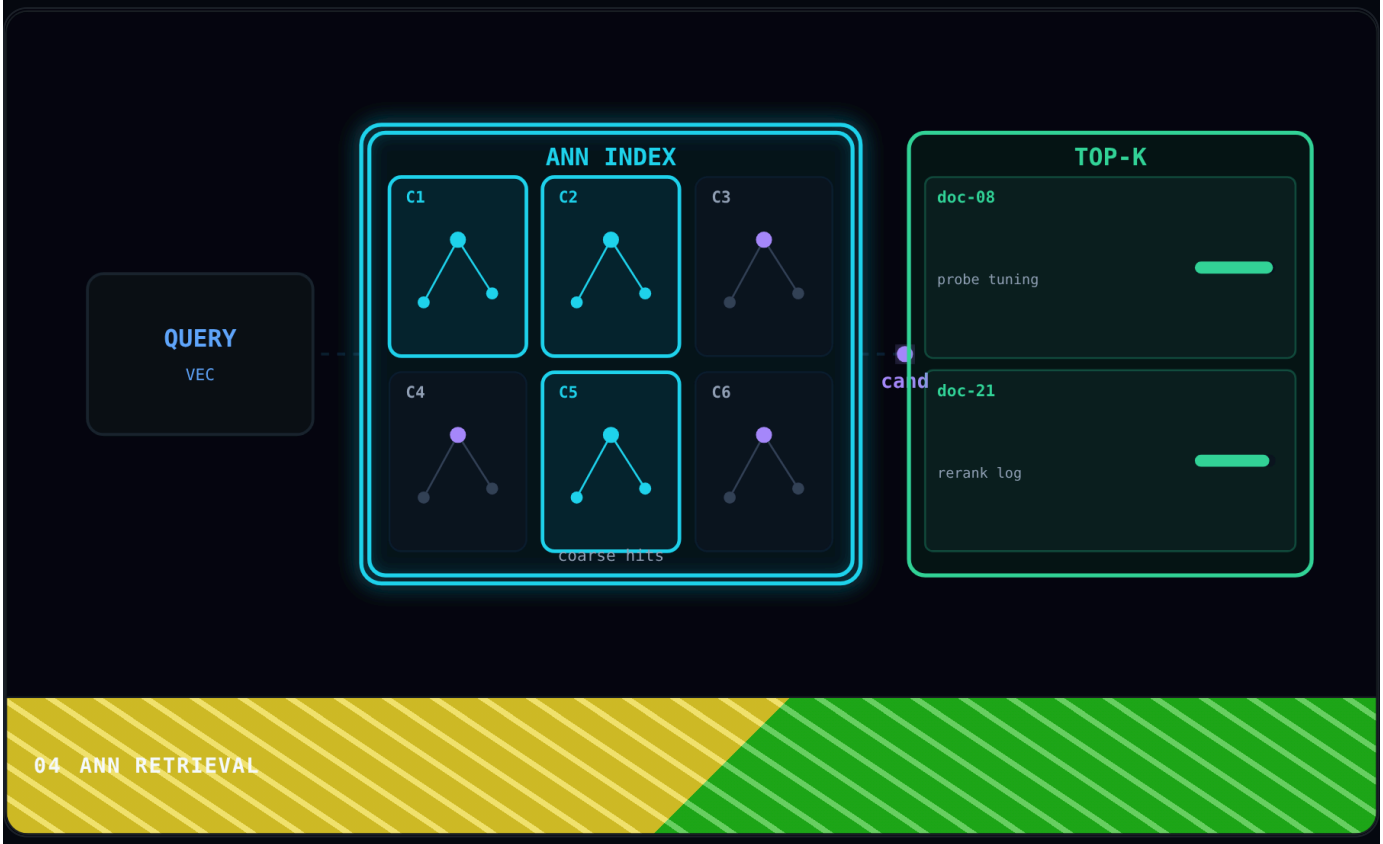
Chunk text is encoded into its own batch of vectors. Increase the Chunks control in the sidebar to see more vectors appear in the store. Each chunk becomes a single row of numbers.

The vector store now holds both the query vector and all the chunk vectors in the same space. Because they share a coordinate system, we can measure distance between any pair.

Toggle Normalize off and watch the bar heights change. Normalization scales all vectors to the same length so that cosine similarity works cleanly. With vectors ready, the next stage will search the index to find the closest matches.

STAGE 4 OF 8

ANN Retrieval



04 ANN RETRIEVAL

We now have a query vector from the embedding stage. Instead of comparing it against every stored chunk, the retriever uses an approximate nearest neighbor index to find the closest matches quickly.

The query embedding enters the index structure. Switch the Index control between HNSW and IVF to see different internal layouts. HNSW uses a navigable graph. IVF partitions vectors into clusters around centroids.

Only a few neighborhoods or centroids are probed. Increase the Probes control and watch more cells light up. More probes means higher recall but slower search.

The retriever accumulates approximate hits from the explored regions. These are the chunks whose embeddings are closest to the query vector in the shared space we built in the previous stage.

The strongest candidates are packed into a top-k shortlist. This list is ordered by raw vector distance, which is a useful but coarse signal.

That shortlist moves forward to reranking. ANN gave us speed. The next stage will spend more compute on fewer items to improve precision.

STAGE 5 OF 8

Rerank

The ANN retrieval stage gave us a coarse shortlist ordered by vector distance. That ordering is fast but shallow. The reranker now reads each candidate alongside the query to make a more informed judgment.

The query is paired with each candidate passage. Unlike ANN which only compared vectors, the reranker reads the actual text of both the question and the passage together.

The cross-encoder reads each query-passage pair. This is the expensive step. It uses a model that sees the full text, not just distance in vector space.

Switch the Signal control to Semantic or Hybrid to see how scoring changes. Hybrid blends vector similarity with keyword matching, which helps when the query uses exact technical terms.

The leaderboard reshuffles as new scores arrive. Notice how the rank order can change dramatically. A passage that was fifth by vector distance might jump to first after the reranker reads it in context.

Only the top reranked passages continue into prompt assembly. The reranker decided what evidence deserves scarce prompt space. Next, we will pack those winners into the final prompt.

STAGE 6 OF 8

Context Assembly

06 CONTEXT ASSEMBLY

We now have a reranked shortlist from the previous stage. But the model cannot read an unlimited amount of context. There is a token budget, and not every passage will fit. This stage decides what makes the cut.

The assembler receives the reranked passages with their token costs attached. Increase the Budget control in the sidebar and watch more passages fit into the prompt.

The packer starts filling the prompt with the highest-value passages first. Each passage has a token cost, and the packer stops when the budget runs out.

Switch the Order control between Score and Chrono. Score order puts the most relevant evidence first. Chronological order arranges passages by their original document order, which helps when time sequence matters.

The prompt now has a system instruction, the user question, and the packed evidence. Anything not packed here is invisible to the model. This is the last chance to shape what the generator sees.

The prompt is fully assembled. Every upstream decision, from chunking boundaries to rerank scores, has shaped what evidence made it here. Next, the model will read this prompt and generate a grounded answer.

STAGE 7 OF 8

Generation

07 GENERATION

This is the final stage. The assembled prompt from context assembly enters the generator model. Everything the model knows about the question comes from what we packed. It has no other access to the corpus.

The generator reads the full prompt context before emitting the first token. This prefill step is where the model absorbs all the retrieved evidence at once.

The decode loop begins and answer lines stream out. Switch the Grounding control to Strict and notice how the answer stays tightly connected to retrieved evidence. Switch to Creative and watch unsupported claims appear.

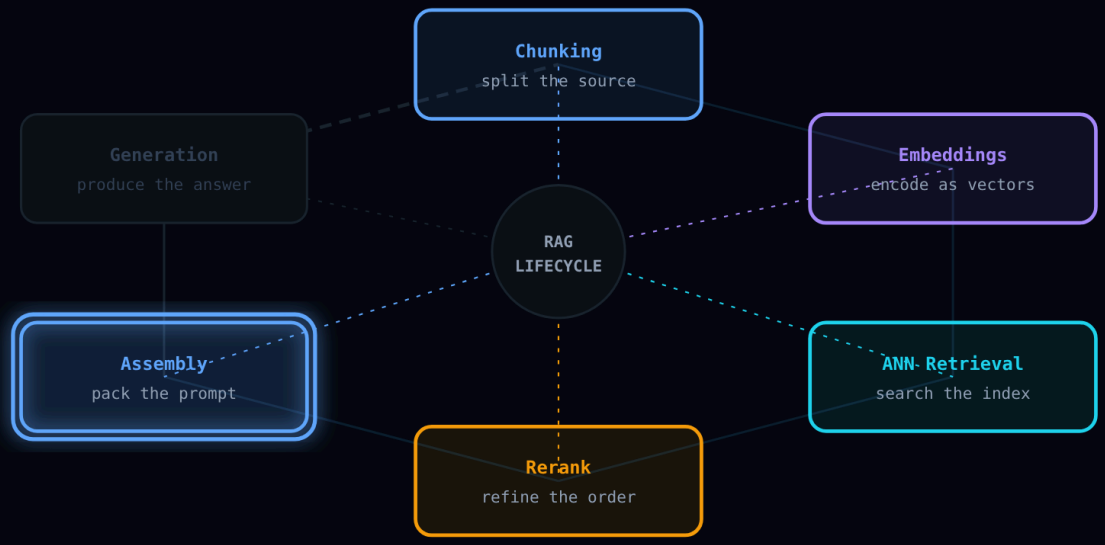
Each answer line is either supported by the packed evidence or not. The amber lines are claims the model generated beyond what the evidence supports. This is the grounding tradeoff in action.

Toggle Citations on to see source references attached to the answer. These markers let the reader trace each claim back to the original chunk. Without citations, the answer looks authoritative but is harder to verify.

The answer is complete. Every stage upstream shaped it: chunk boundaries decided what evidence existed, embeddings made it searchable, ANN found candidates, reranking refined them, and assembly packed the winners. Now let us step back and see the whole picture together.

STAGE 8 OF 8

Recap



08 RECAP

We started with Chunking, where raw documents were broken into searchable passages. The chunk boundaries set the grain of everything downstream. Too coarse and the retriever misses details. Too fine and context gets fragmented.

Then we turned those chunks into Embeddings, projecting text into a shared vector space. This is what makes meaning-based search possible. Without embeddings, we would be stuck matching keywords.

ANN Retrieval searched the index to find the chunks closest to our query vector. It traded exhaustive accuracy for speed by probing only a few neighborhoods. The top-k shortlist was fast but coarse.

Reranking spent more compute on that shortlist, reading each candidate alongside the query to produce a more informed ordering. This precision pass decided which evidence deserved scarce prompt space.

Context Assembly packed the top-ranked passages into a finite token budget. Anything that did not fit was invisible to the model. This was the last gate before generation.

Generation produced the final answer from the packed prompt. Its quality was bounded by every upstream decision. That is the key insight: RAG is not one step. It is a chain of decisions, and the weakest link sets the ceiling for the answer.

All six stages are now lit. Together they form the RAG query lifecycle. When an answer quality problem appears, trace it backward through this chain. The fix is almost never in generation alone. It is usually in chunking, retrieval, or assembly.