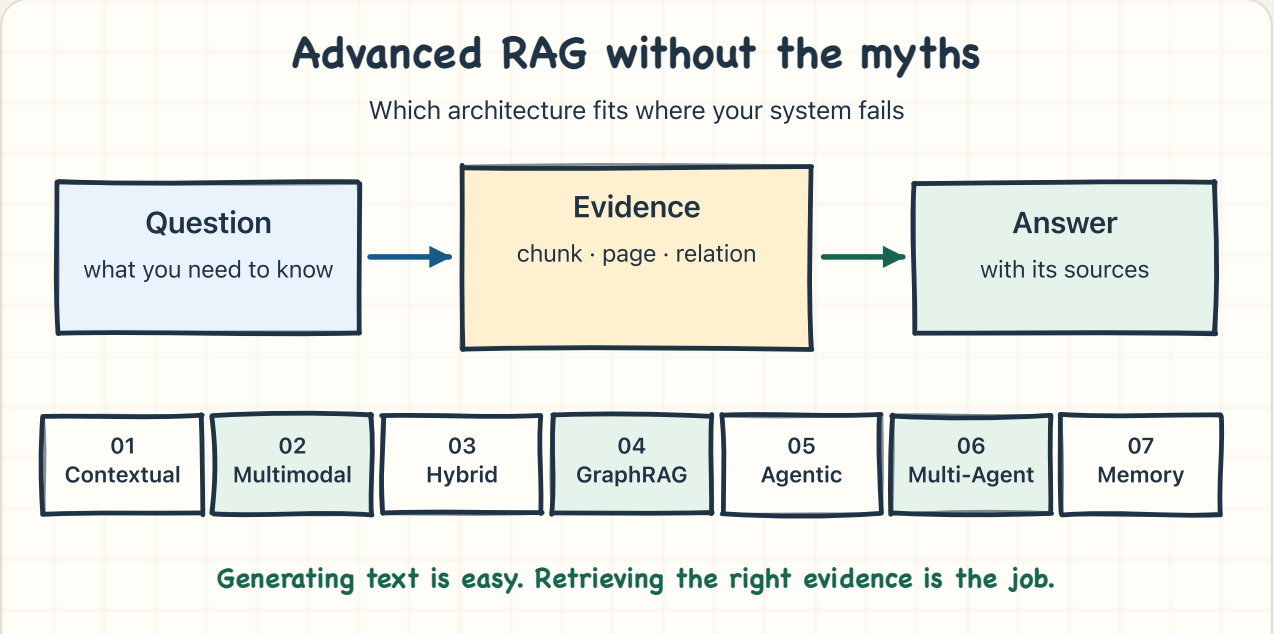
Advanced RAG Without the Myths: Which Architecture Fits Where Your System Fails

Created by Manuel Herrera Mesén · November 10, 2026

rag artificial-intelligence llm architecture search

It is two in the morning and a payments alert fires. The on-call engineer asks the team's assistant: "how do I fix the connection timeout in PaymentService?". The answer arrives in seconds, well written and confident. And it comes from another service's runbook.

The uncomfortable part is that there may be no programming error at all. The system did exactly what it was designed to do: it searched for the chunks most similar to the question and wrote an answer with them. The problem happened before the writing, in what it found.



Generating text is easy. Retrieving the right evidence is the job.

> **Takeaway:** most of the time, the underlying problem is not generating good answers but retrieving evidence that is correct, sufficient, and verifiable.

That idea runs through this whole guide. A RAG system does not improve by adding tools; it improves when it fixes a weakness you can demonstrate. The seven architectures we will look at are neither mandatory stages nor mutually exclusive alternatives: they are patterns you combine depending on the failure in front of you.

Where you are starting from	Where I suggest you read
You have never built a RAG system	Start with the case and the foundations, then read in order.
You have one and it fails	Go to the failure map, find your symptom, and jump to its architecture.
You design production systems	Read the enterprise implementation, the evaluation, and open the "Going deeper" notes.

▼ IN THIS ARTICLE

01. The Case We Will Follow
02. First: What RAG Is and How It Works
03. Where Naive RAG Starts to Fail
04. 01 · Contextual RAG: Giving the Chunk Its Meaning Back
05. 02 · Multimodal RAG: What Is Not Just Text
06. 03 · Hybrid Search: Meaning and Exact Match
07. 04 · GraphRAG: Relations Between Entities
08. 05 · Agentic RAG: Searching Again When the Evidence Falls Short
09. 06 · Multi-Agent RAG: Specialized Investigations in Parallel
10. 07 · Memory RAG: Retrieving What Was Decided Before
11. How to Choose an Architecture
12. What It Would Look Like in a Company
13. How to Really Evaluate a RAG System
14. Common Myths
15. What the On-Call Assistant Would Look Like
16. Sources

The Case We Will Follow

To keep this from staying theoretical, I will use a single case from start to finish. It is not a real system: I put it together for this guide. But every failure that shows up is one you will run into in a real one.

An online store runs on four Java microservices that talk to each other through events in Kafka: orders (OrderService), payments (PaymentService), inventory (InventoryService), and notifications (NotificationService). The team wants an **on-call assistant**: whoever gets an alert asks it, in plain language, what is going on and what to check.

The assistant queries four sources:

- each service's **runbooks**, the procedures for operating and recovering it;
- the **ADRs** (architecture decision records), the documents that record each architecture decision and why it was made;
- past incident **tickets**;
- and the services' **logs**.

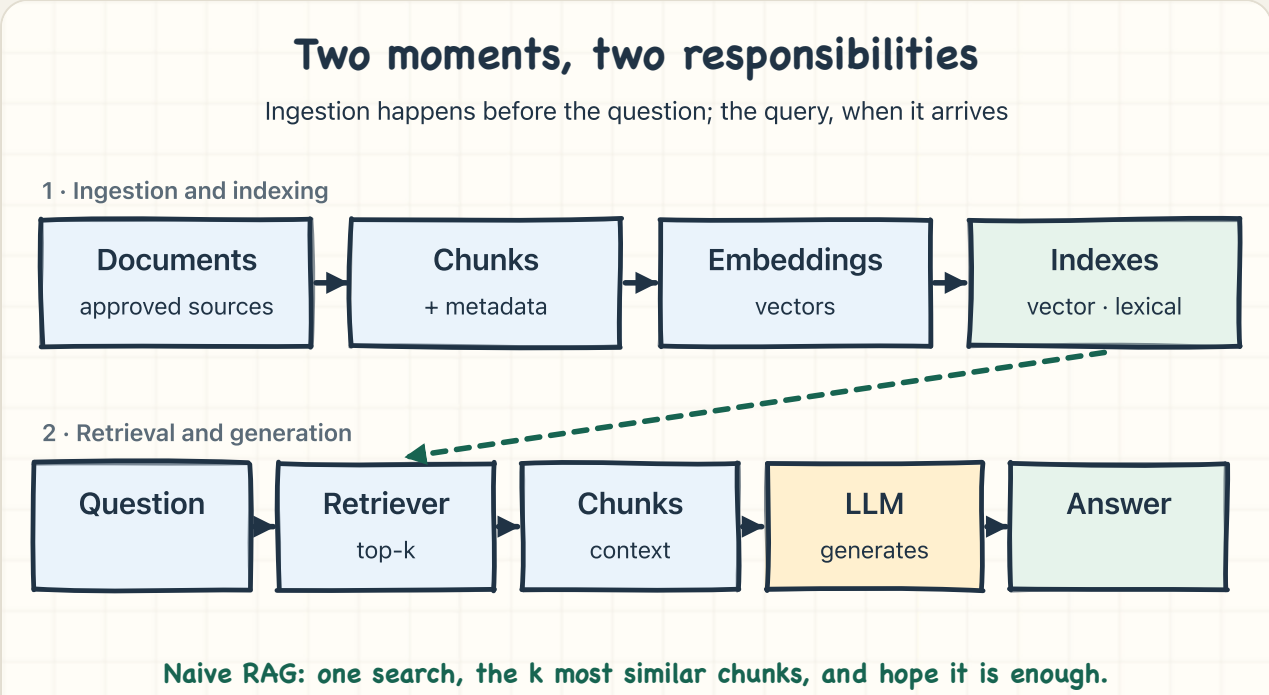
Each chapter opens with an on-call question that the simplest version answers badly, and at the end we put together the version I would build.

First: What RAG Is and How It Works

Retrieval-Augmented Generation (RAG) is a way of building applications in which you look up external information and hand it to a language model before it answers. The model does not learn that information permanently: it receives it as context for that call and does not keep it once the call ends.

Every RAG system has two moments, and it pays to keep them apart:

1. **Ingestion and indexing**, before any question exists: you collect the documents, keep their metadata, extract the text, split it into chunks, and build the indexes.
2. **Retrieval and generation**, when the question arrives: you interpret it, query the authorized sources, retrieve the evidence, and the model builds the answer with it.



Ingestion prepares what the query will be able to find. If something is lost at indexing time, no query will retrieve it.

The simplest implementation retrieves the **top-k**, the k chunks most similar to the question, and passes them to the model. It is often called **Naive RAG**. That is not an insult: for small collections and simple questions it is a good starting point, and the baseline you will measure everything else against. Its limit is assuming that a single search brings back everything needed.

The Concepts You Will See Throughout the Guide

Term	What it is
LLM	Large language model: the model that reads the context and writes the answer.
Token	The unit a model splits text into; a word takes one or several. Cost and context size are measured in tokens.
Prompt	Everything the model receives in one call: instructions, question, and evidence.
Chunk	A fragment of a document: the unit that gets indexed and retrieved.
Embedding	A vector of numbers that represents the meaning of a text. Two similar texts have nearby vectors; closeness is usually measured with cosine similarity, which ranges from −1 to 1.
Semantic search	Searches by meaning, by comparing embeddings.
Lexical search	Searches by matching words. **BM25** is its most widely used algorithm: it scores by the shared words and by how rare they are in the collection.

Term	What it is
Vector database	The system that stores embeddings and finds the ones closest to a given one.
Retriever	The component that, given a question, returns relevant candidates.
Reranker	A second pass that reorders the best candidates with a more expensive and more precise model.
Tenant	Each customer or organization that shares the same system and whose data must stay isolated from the rest.

How You Split the Document Matters More Than It Seems

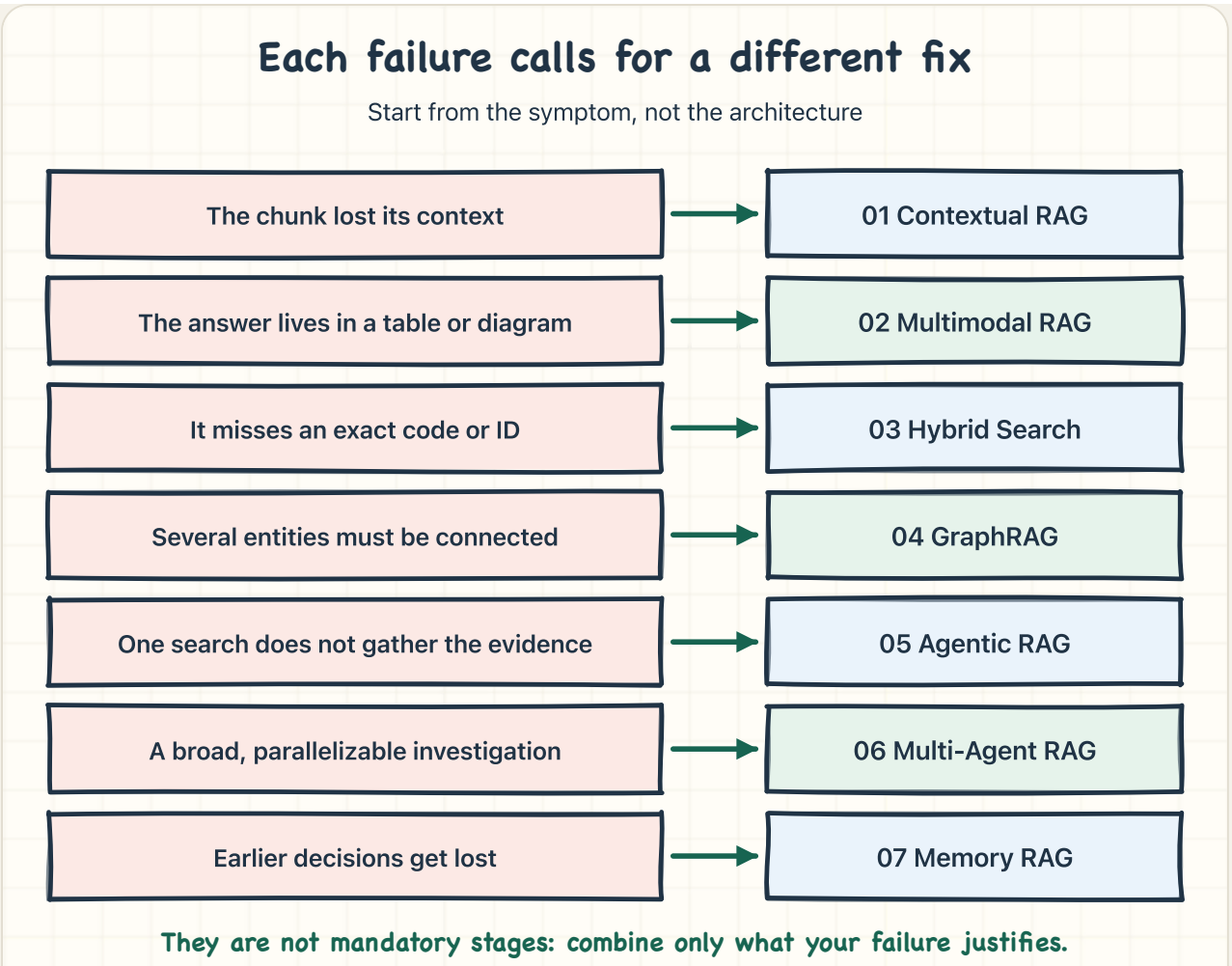
Before any advanced architecture, there is one decision behind many failures: **chunking**. A chunk that is too large mixes topics and dilutes its embedding; one that is too small loses the context that gives it meaning. The usual approach is to split along the document's structure (headings, sections, paragraphs), with some overlap between chunks.

And each chunk records where it came from: document, section, version, and who can see it. In the on-call assistant, also the service and the environment. There is no chunk size that works for everything: you choose it with data, like everything else in this guide.

> **– Going deeper**
>
> if you change the embedding model, you have to reindex everything. Vectors from two different models do not live in the same space and cannot be compared with each other. Store the model's name and version alongside the index so that change is an explicit decision and not a silent error.

Where Naive RAG Starts to Fail

Each of these failures can justify a different improvement. This map is the practical index of the guide: find your symptom and jump to its architecture.



Start from the symptom, not the architecture.

Symptom	The on-call question that reveals it	Architecture
The chunk lost its context	"How do I fix the connection timeout in PaymentService?"	01 · Contextual RAG
The answer lives in a diagram or a table	"Which services consume OrderCreated?"	02 · Multimodal RAG
It misses an exact code or identifier	"HikariPool-1 connection timeout"	03 · Hybrid Search
Several entities must be connected	"If OrderService stops publishing, what breaks?"	04 · GraphRAG
One search does not gather the evidence	"Why did order ORD-10482 fail?"	05 · Agentic RAG
The investigation is broad and parallelizable	"What caused Friday's payments outage?"	06 · Multi-Agent RAG
Earlier decisions get lost	"Is what we decided about the pool still in effect?"	07 · Memory RAG

01 · Contextual RAG: Giving the Chunk Its Meaning Back

The problem: when a chunk is separated from its document, it can lose exactly the information that made it useful.

This is the failure from the opening of this guide. All four runbooks have a nearly identical section on database timeouts, and the chunk that holds it says this:

BEFORE

Original chunk

```
If the error persists, raise
maximumPoolSize to 20 and restart the
service.
```

AFTER

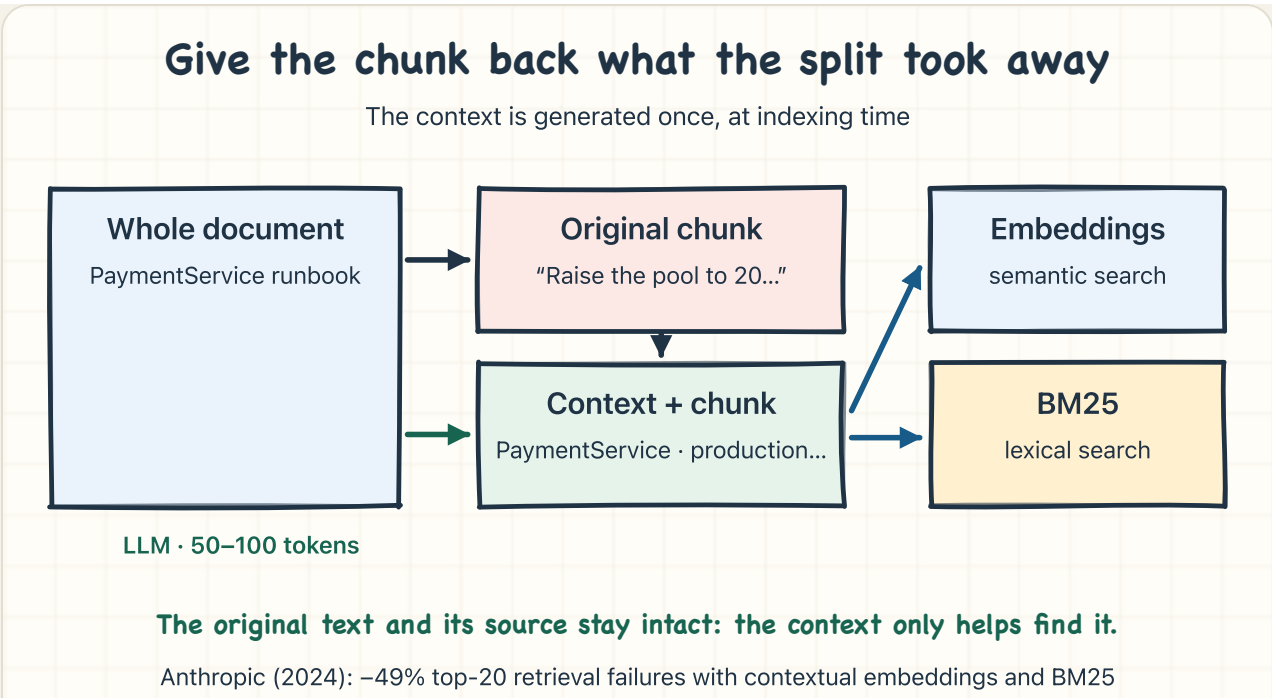
Contextualized chunk

```
PaymentService runbook, production.
Section "Database connection timeouts".
If the error persists, raise
maximumPoolSize to 20 and restart the
service.
```

The original chunk does not say which service or which environment it refers to. For the search, all four chunks are equally good, and the assistant can hand you InventoryService's instruction for a payments problem. The contextualized chunk names the service, the environment, and the section: now the question has something to hold on to.

How It Works

During ingestion, a language model reads the whole document and writes a short description for each chunk, which is prepended to the text before indexing it. Anthropic presented the technique under the name *Contextual Retrieval*: that context usually takes between 50 and 100 tokens, and it is used both for the embeddings and for the BM25 index, which it calls **Contextual Embeddings** and **Contextual BM25**.



The context is generated once, at indexing time. The original text is not modified.

In their experiment, the retrieval failure rate (the questions whose evidence did not appear in the top 20 results) was 5.7%. It dropped to 3.7% with contextual embeddings (−35%), to 2.9% when adding contextual BM25 (−49%), and to 1.9% when adding reranking (−67%) (Anthropic, *Contextual Retrieval*, 2024). These are results on their data, not a promise about yours.

When to use it	Its limits
Long manuals, contracts, policies, corporate documentation, and code, where an isolated chunk is ambiguous.	It makes indexing more expensive: one model call per chunk. If the generated context is wrong, it puts misleading information into the index.

Before paying that cost, check whether a metadata field is enough. If each chunk stores its service, filtering on `service = PaymentService` solves this specific case, and it is cheaper. Contextual RAG helps when the ambiguity does not fit in a field.

> – Going deeper
>
> the cost of one call per chunk skyrockets if each call sends the whole document. With **prompt caching**, which caches the repeated part of the prompt so you do not pay for all of it on every call, the document is sent once and each chunk pays only its share.
>
> Anthropic estimated USD 1.02 per million document tokens assuming 800-token chunks, 8,000-token documents, 50 tokens of instructions, 100 tokens of context per chunk, and 2024 Claude 3

> Haiku pricing: recalculate it with your sizes and today's prices. And when the document changes, you have to regenerate the context of its chunks, not just their embeddings.

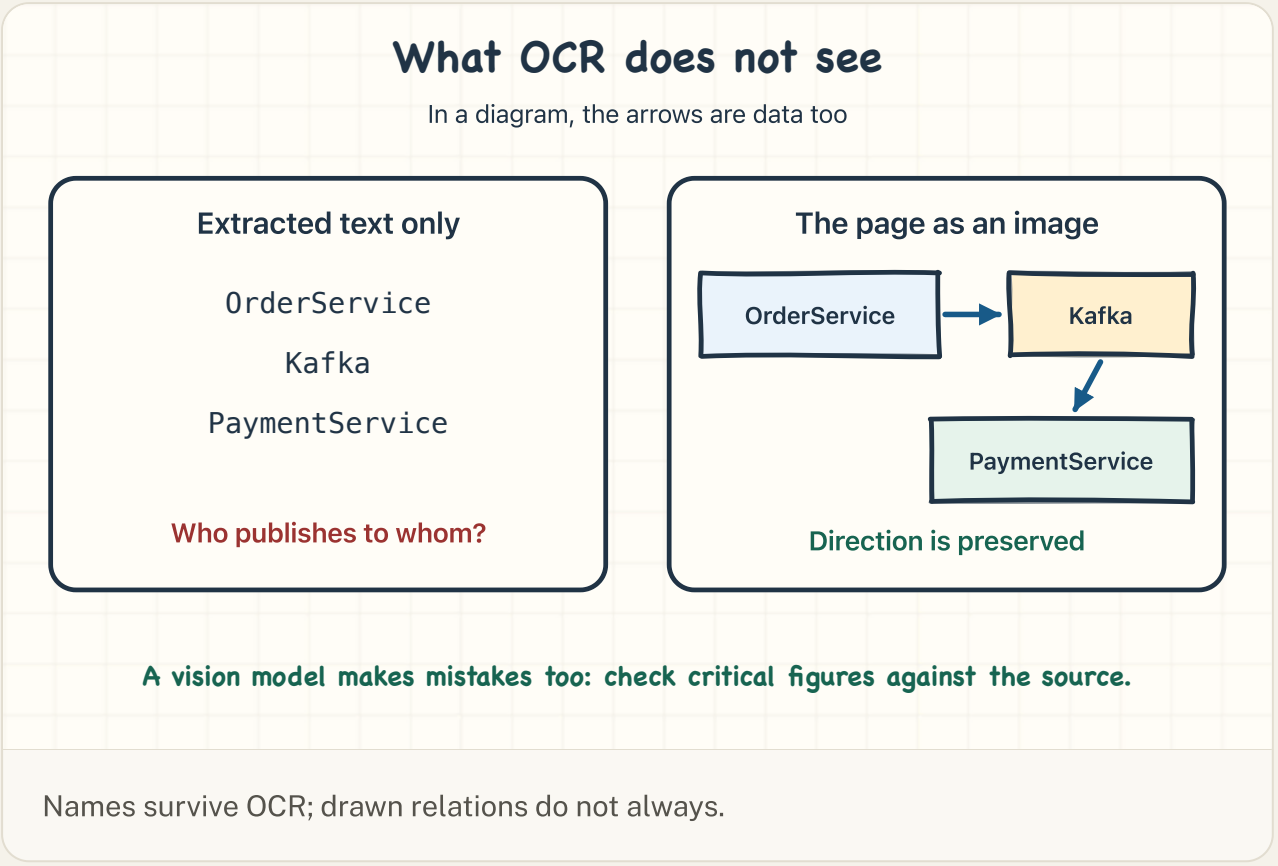
> **Takeaway:** always keep the original text and its provenance. The generated context helps find the chunk; it is not a new source of truth.

---

02 · Multimodal RAG: What Is Not Just Text

The problem: extracting the text from a PDF can destroy information that lived in tables, diagrams, charts, and images.

In the on-call assistant, the store's architecture diagram only exists as an image inside an ADR. Someone asks: "which services consume OrderCreated?". **OCR** (optical character recognition, the step that turns an image into text) recovers the names in the boxes, but the answer lies in the direction of the arrows, and arrows are not text.



Names survive OCR; drawn relations do not always.

How It Works

There are two paths, and they can be combined:

- **Retrieve the page as an image.** The system computes visual representations of each page, retrieves the most relevant ones for the question, and hands them to a model that can read images. ColPali is the academic reference for this approach: it indexes pages directly as images, without going through OCR (Faysse et al., ICLR 2025).
- **Extract better.** Combine OCR, structured table extraction, image descriptions, and multimodal embeddings so the structure reaches the index as meaningful text.

When to use it	Its limits
Scanned PDFs, illustrated manuals, financial tables, architecture diagrams, and any document where the visual layout matters.	More compute and more storage. A vision model also makes mistakes reading figures, relations, or structures.

> – **Going deeper**
>
> ColPali does not produce one vector per page but many: one for each patch of the image. Similarity is computed with *late interaction* (the same mechanism as ColBERT): each token of the question looks for its most similar patch and those matches are added up. It is more precise than comparing a single vector, but it takes considerably more space in the index.

> **Takeaway:** looking at the image does not eliminate errors. With critical data, check the values the model read against the original source.

---

03 · Hybrid Search: Meaning and Exact Match

The problem: semantic search finds related concepts, but it can miss exact codes and terms.

In the on-call assistant, the engineer pastes the error just as it appears in the log: **`HikariPool-1 connection timeout`**. HikariCP is the connection pool Spring Boot uses by default, and **`HikariPool-1`** is the name it gives the first one. Semantic search brings back general documentation about connection pools; lexical search finds the ticket where that exact message already showed up. Each one sees part of the picture.

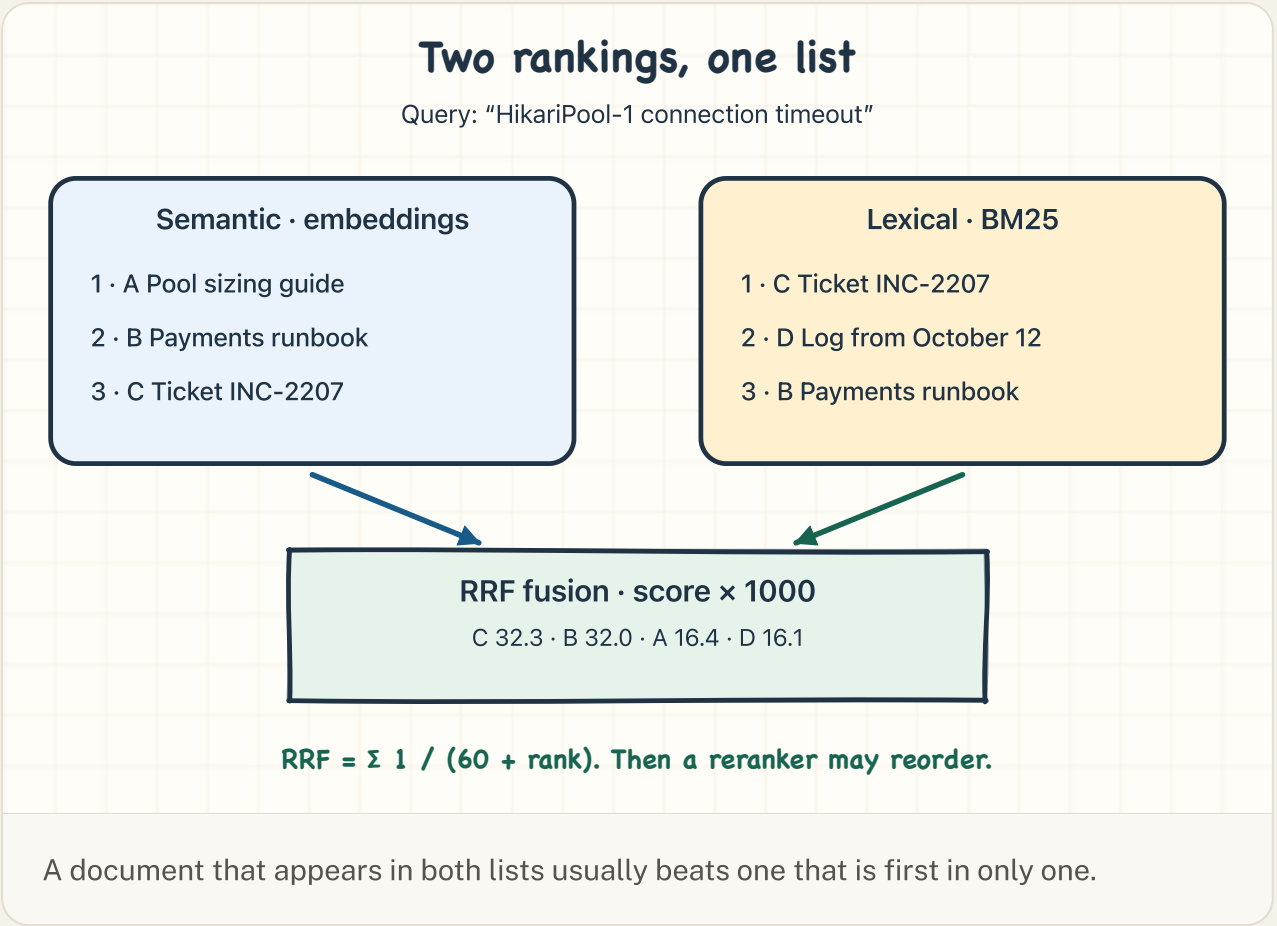
How It Works

Hybrid search runs both searches over the permitted sources and fuses their results. The problem is that their scores cannot be compared: a cosine similarity of 0.82 and a BM25 score of 14.3 measure different things.

The usual technique is **Reciprocal Rank Fusion (RRF)**, which ignores the scores and uses only the positions. Each document adds up, for every list it appears in:

```
RRF
score(d) = Σ  1 / (60 + position of d in that list)
```

That 60 is a smoothing constant: the original paper calls it k, which has nothing to do with the k in top-k. Its authors set it in a pilot test and found that its exact value was not critical (Cormack, Clarke, and Büttcher, SIGIR 2009). Here is how it works with four documents:



A document that appears in both lists usually beats one that is first in only one.

RRF fuses two rankings step by step

Candidates	RRF score (× 1000)	Final list
		✓ C · Ticket INC-2207
		✓ B · Payments runbook
		✓ A · Pool sizing guide
		✓ D · Log from October 12

The candidates arrive. Semantic search ranked A, B, C; lexical search (BM25) ranked C, D, B. The original scores are discarded: only positions count. To make them easier to read, RRF scores are shown

multiplied by 1000.

A is first in semantic search and does not appear in BM25: 1000 / (60 + 1) ≈ 16.4.

D is second in BM25 and does not appear in semantic search: 1000 / (60 + 2) ≈ 16.1.

B appears in both: second in semantic search and third in BM25. 1000 / 62 + 1000 / 63 ≈ 32.0.

C also appears in both: third in semantic search and first in BM25. 1000 / 63 + 1000 / 61 ≈ 32.3.

Final order: C, B, A, D. The ticket with the exact message, which semantic search left third, ends up first because both searches found it.

After fusion, a **reranker** can reorder the top candidates. The retriever scores each document with representations prepared in advance, without reading it alongside the question. The reranker, usually a *cross-encoder*, reads the question and each document together, and that is why it judges relevance better. It is more expensive, so it is applied only to the top results.

When to use it	Its limits
Logs, API documentation, source code, tickets, endpoints, error codes, and identifiers.	You have to configure two indexes, their tokenization, and the fusion, and evaluate relevance. BM25 does not guarantee an exact match either.

– Going deeper

whether BM25 finds `HikariPool-1` depends on the analyzer, the component that splits text into terms. A standard one usually splits it into `hikaripool` and `1`. That `1` appears in half the corpus, so it adds almost nothing: a log for `HikariPool-2`, which is a different pool, scores practically the same. For identifiers, use a separate field that keeps them whole.

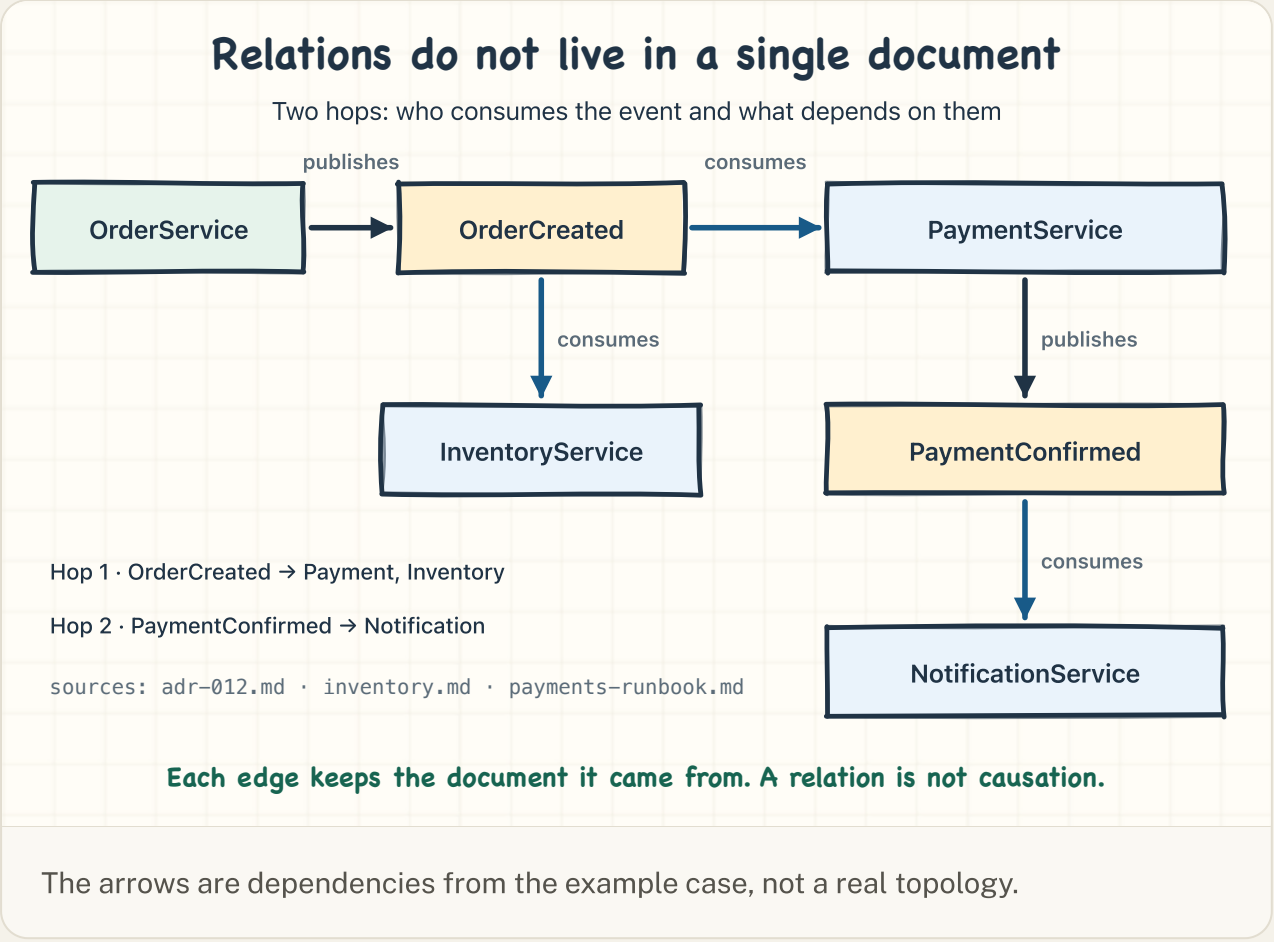
And RRF is not the only fusion: normalizing the scores and adding them with weights can perform better if you tune those weights with labeled data; RRF needs no tuning, which is why it is the starting point.

Takeaway: measure Recall@k and nDCG@k before and after adding hybrid search. The first tells you how much of the correct evidence appears in the top k results; the second, whether it appears near the top. Both are explained in the evaluation. If they do not improve on your real queries, you gained nothing but complexity.

04 · GraphRAG: Relations Between Entities

The problem: sometimes the answer depends on relations spread across several documents and components, and no single chunk contains it.

In the on-call assistant: "if OrderService stops publishing OrderCreated, what breaks?". You need to know who consumes that event and, in a second hop, what depends on those consumers. That information is spread across an ADR, each service's runbook, and the service inventory. A similarity search brings back loose chunks; it does not walk the chain.



The arrows are dependencies from the example case, not a real topology.

How It Works

The system extracts entities and relations from the documents, builds a knowledge graph, and stores in each relation the document that supports it. A question about a specific entity retrieves its neighborhood in the graph, and with it the chunks that back it up.

Microsoft GraphRAG, the best-known implementation, goes one step further. It groups entities into communities with the Leiden algorithm and writes a summary of each one. That way it can answer general questions about the whole collection, such as "what are the main themes?".

It offers four search modes: **local** (an entity and its neighbors), **global** (over the community summaries), **DRIFT** (local, but enriched with the community's context), and **basic** (the usual vector top-k). Not every GraphRAG implementation uses communities.

When to use it	Its limits
Dependency analysis, incident investigation, organizational knowledge, and questions that connect many entities and sources.	Building and maintaining the graph is expensive: extracting entities and summarizing communities requires model calls during indexing. Extracted relations can be wrong or go stale.

Two traps before you build it. The first is **entity resolution**: if the same service appears as **`PaymentService`**, **`payments-svc`**, and "the payments service", without merging them you get three nodes and the chain breaks. The second: if the relations already exist as data (a service catalog, the Kafka consumer configuration), build the graph from there. It is more accurate than having a model extract them from text.

> **Takeaway:** every relation must keep its provenance. And two things appearing related in a text does not mean one causes the other.

---

05 · Agentic RAG: Searching Again When the Evidence Falls Short

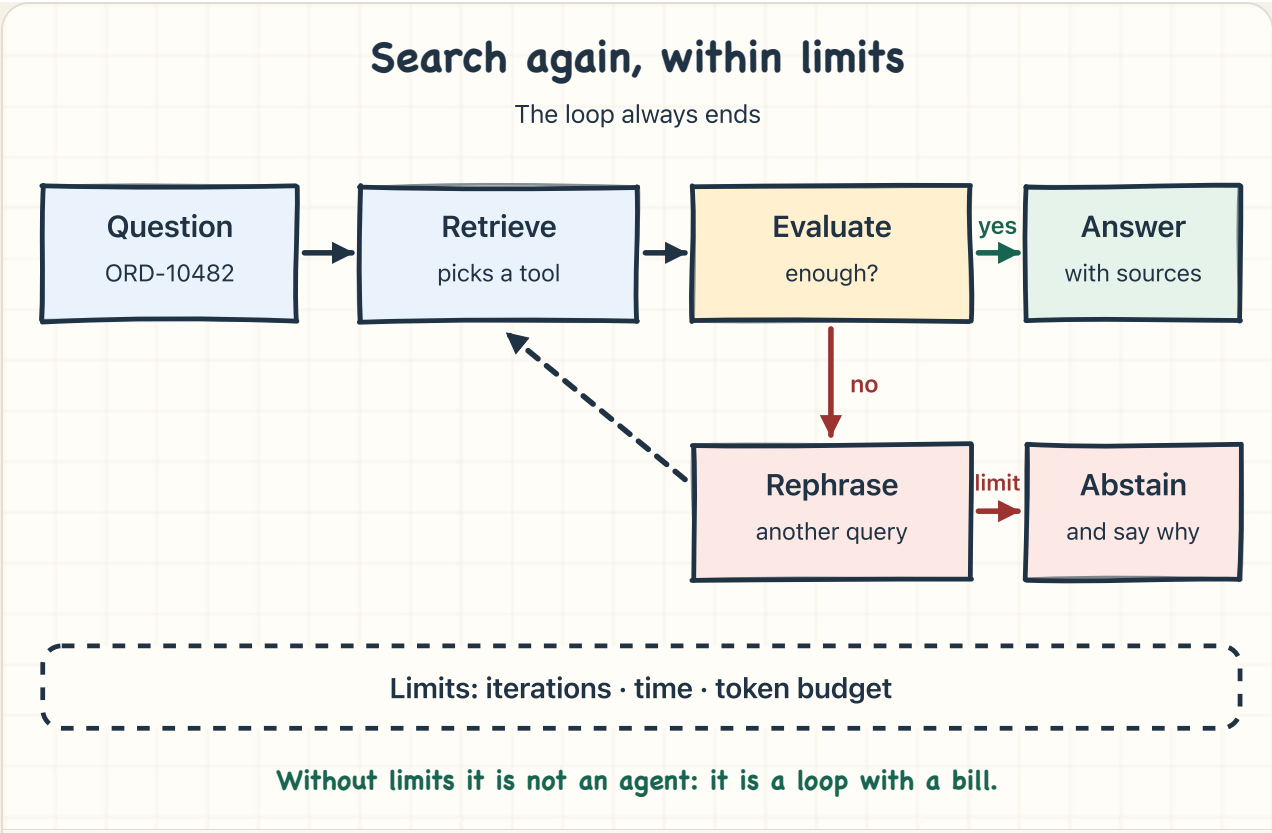
The problem: a single retrieval is not always enough to answer a complex question.

In the on-call assistant: "why did order ORD-10482 fail, and which service should I check first?". First you have to find the order in the logs; only what shows up there tells you which runbook to consult. That is several searches, and the second depends on what the first one found.

How It Works

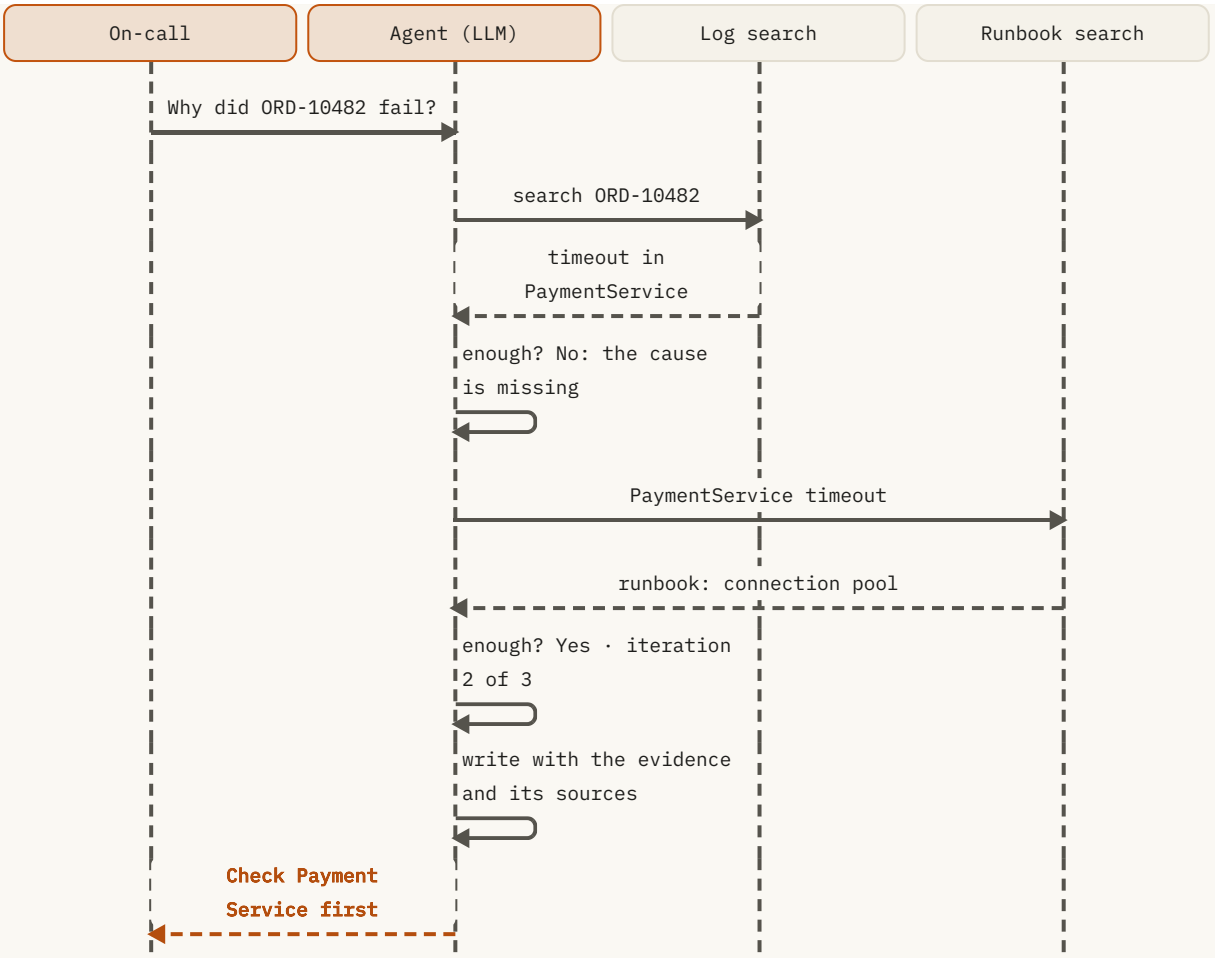
An agent, which is the language model itself with access to search tools, decides what to look for, inspects what it found, and judges whether it is enough. If evidence is missing, it rephrases the question or queries another source. The process ends when it has gathered enough information, when it hits a limit, or when it concludes that the available evidence cannot answer the question.

Those limits are not requested from the model: they are enforced by the code that orchestrates it, which counts iterations, measures time, and stops when the budget runs out. A model can ignore an instruction; a counter in your code cannot.



The loop always ends: with an answer, or with an explained abstention.

Two searches for a single question



The question needs a fact (what happened) and an explanation (why, and where to look).

It is an exact identifier: the agent picks the log search.

It now knows what happened, but not why.

It evaluates the evidence before answering. Iteration 1 of at most 3: the code orchestrating the agent keeps count, not the model.

It rephrases with what it learned: now it searches the service's runbook.

It finds the procedure that explains that timeout.

It has the fact and its explanation, within the limit.

The same model writes the answer with the retrieved evidence, not with what it thinks it remembers.

Every claim points to the log or runbook it came from, so the on-call engineer can verify it.

Before building an agent, ask yourself whether you need one. Anthropic distinguishes *workflows*, where code defines the steps in advance and the model executes them, from agents, where the model decides the steps (Anthropic, *Building effective agents*, 2024). If you can write the flow down ("search the logs for the order; if a service shows up, search its runbook"), a workflow is cheaper, more predictable, and easier to test.

The idea of retrieving more than once was not born with today's agents. **Self-RAG** trained a model to decide when to retrieve and to critique its own answer (Asai et al., 2023). **Corrective RAG** added an

evaluator that scores the retrieved documents and, if they are not useful, discards them and searches the web (Yan et al., 2024). Agentic RAG generalizes those ideas with tools.

When to use it	Its limits
Technical investigations, diagnostics, multi-hop questions, and problems that require more than one retrieval.	More latency, more cost, and the risk of unnecessary searches. The agent can also make the wrong decision.

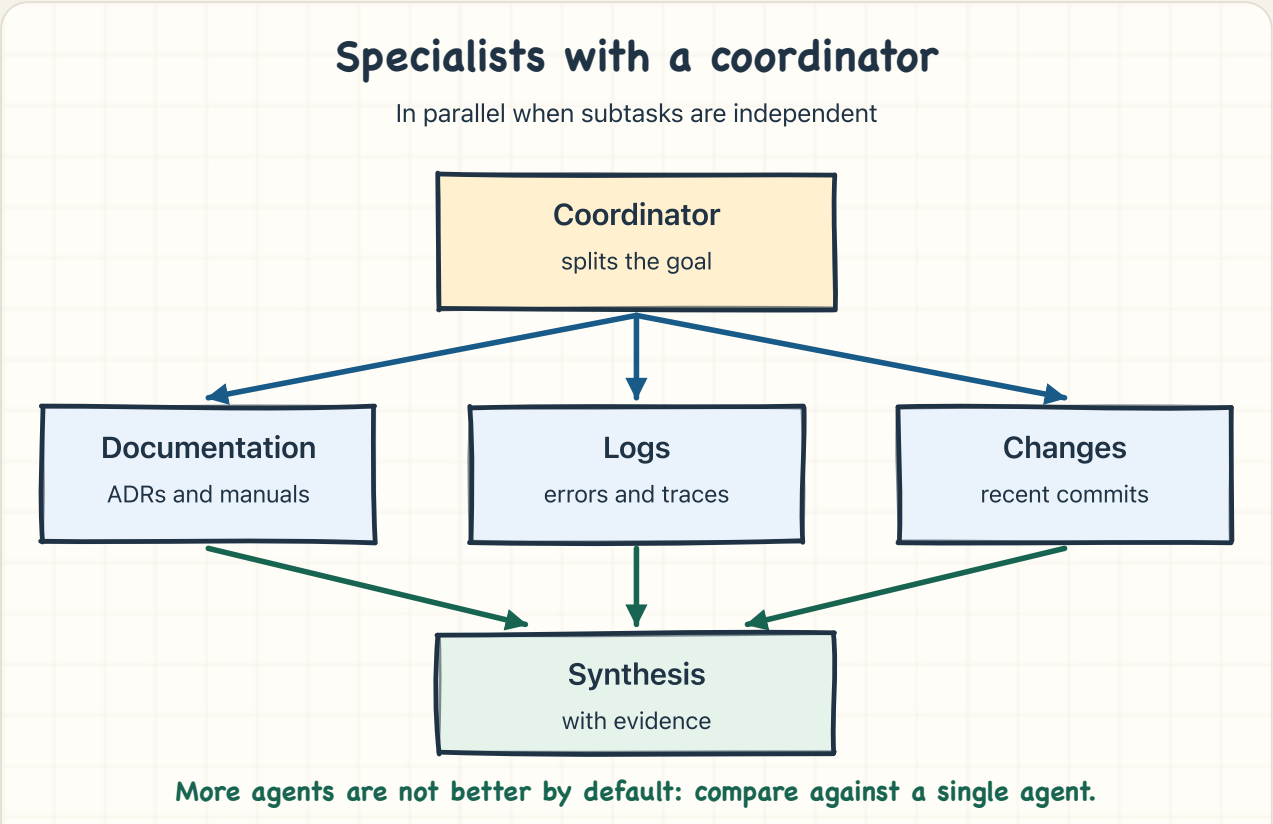
> **Takeaway:** set limits on iterations, time, and budget, and enforce them in code. A loop without limits is not an agent: it is a cycle with a bill.

---

06 · Multi-Agent RAG: Specialized Investigations in Parallel

The problem: some investigations need different tools, sources, and subtasks that are independent of each other.

In the on-call assistant, no longer at two in the morning but in the post-incident review: "what caused Friday's payments outage?". You need to review the architecture documentation, the logs, and the recent code changes. Those are three searches that do not depend on one another and that call for different tools.



Independent subtasks can run in parallel.

How It Works

A coordinator agent breaks the goal into subtasks. Each specialist queries its sources and returns findings with references. The coordinator combines the evidence, detects contradictions, and prepares the answer. When the subtasks are independent, they run in parallel.

Working does not make it free. In Anthropic's research system, the multi-agent version (Claude Opus 4 as coordinator with Claude Sonnet 4 subagents) outperformed Claude Opus 4 working alone by 90.2% on their internal evaluation. The price: a single agent uses about 4 times the tokens of a chat conversation, and the multi-agent system about 15 times.

In their own analysis, on the BrowseComp evaluation, the number of tokens spent explained 80% of the variance in performance: much of the improvement comes from spending more (Anthropic, *How we built our multi-agent research system*, 2025). The same article warns that it fits poorly when all agents need to share the same context or when there are many dependencies between them.

When to use it	Its limits
Broad investigations, heterogeneous tools, independent sources, and tasks with clear parallelism.	More complexity in coordination, observability, and cost control. Several agents do not guarantee a better result.

> **Note:** Agentic RAG and Multi-Agent RAG are not the same thing. The first is about dynamic decisions on retrieval; the second, about coordinating several agents. A system can have one without the other.

> **Takeaway:** before justifying a multi-agent architecture, compare it against a single well-designed agent, with the same questions and measuring the cost.

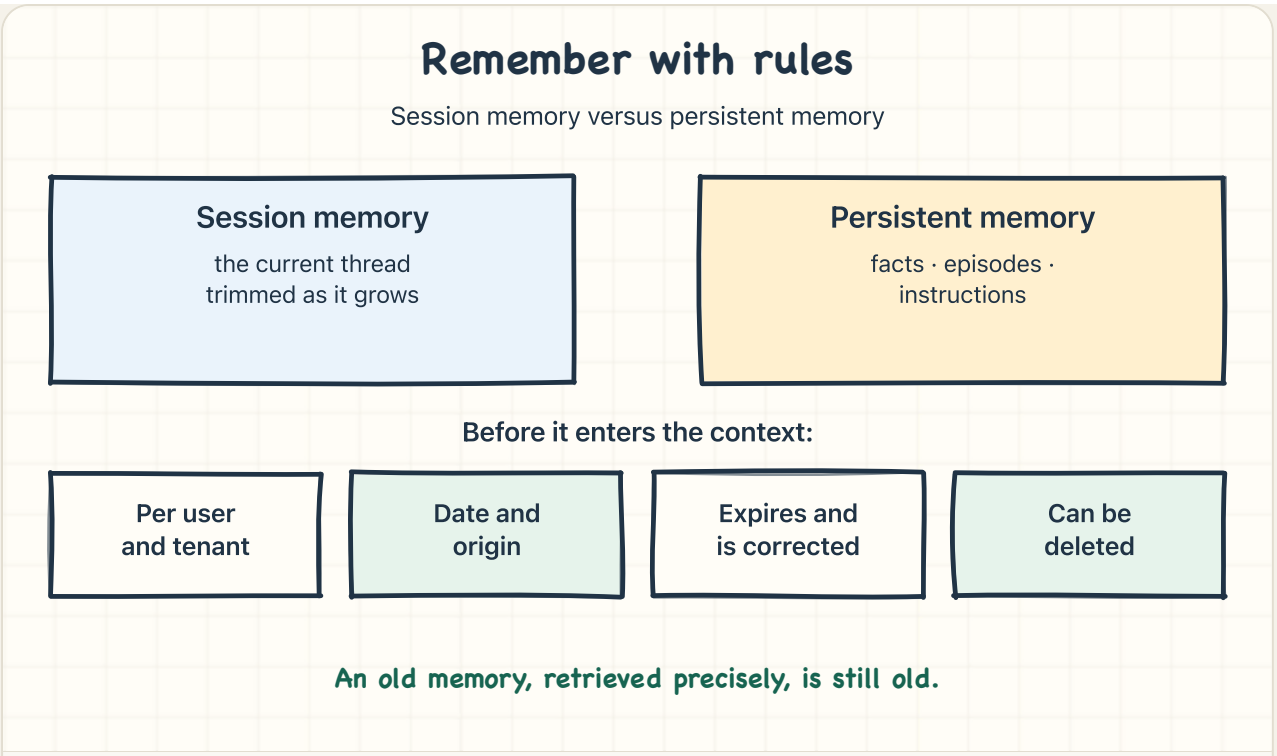
---

07 · Memory RAG: Retrieving What Was Decided Before

The problem: a new conversation can depend on decisions or data shared in another one.

In the on-call assistant: "last week we decided to keep the payments pool at 20 connections until we migrate the database. Is that still in effect?". Without access to that decision, the model cannot answer reliably: at best it says it does not know; at worst, it makes something up.

"Memory RAG" is not a standard term: I use it to name the pattern of retrieving memory with the same techniques you use to retrieve documents.



An old memory, retrieved precisely, is still old.

How It Works

The model remembers nothing between calls: what remembers is the application, which stores and retrieves. It distinguishes two memories. **Session memory** is the thread of the current conversation and gets trimmed when it grows too large. **Persistent memory** survives across conversations: it stores facts, decisions, or summaries with an identifier, a date, their origin, and retention rules.

LangGraph, for example, splits persistent memory into semantic (facts), episodic (past experiences), and procedural (the instructions that guide the agent). When a new query arrives, the system retrieves only the relevant memories that this user is authorized to see.

A memory can also be poisoned. If the assistant stores anything someone claims in a conversation, a false statement becomes a "fact" that comes back in every conversation after it. In the on-call assistant, I would only store confirmed decisions, with who made them and when they expire.

When to use it	Its limits
Personal assistants, long projects, recurring technical support, and systems that need continuity.	Stale or poisoned memories, incomplete summaries, privacy risks, and information leaking between users.

> **Takeaway:** define explicit rules for updating, expiring, correcting, deleting, and isolating memories per user or tenant before you store the first one.

How to Choose an Architecture

The right question is not which architecture is more advanced. It is this one: **what retrieval problem does my system have, and how will I check that the solution fixes it?**

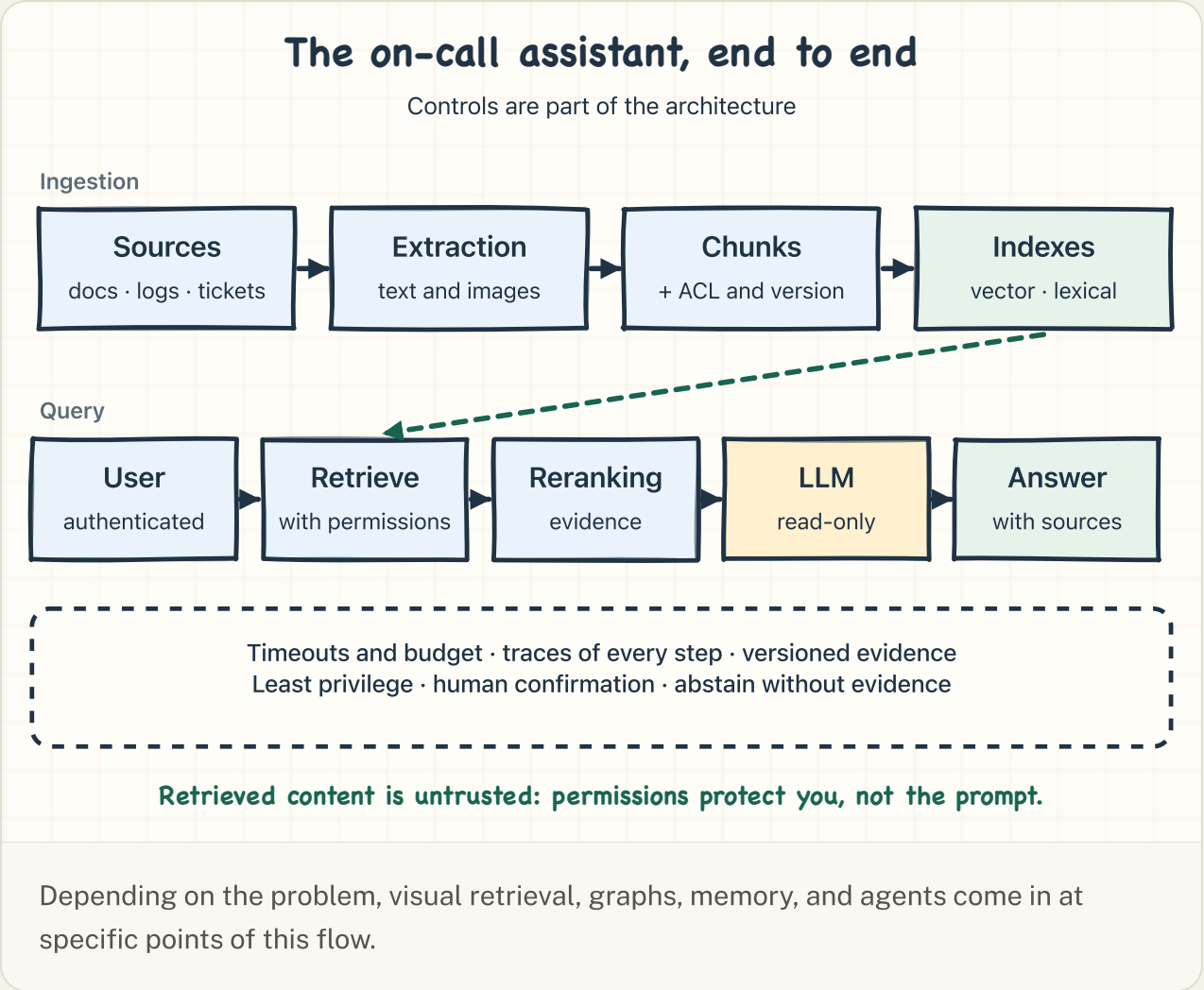
Problem	Candidate architecture	How you validate it
Ambiguous chunks	Contextual RAG	Recall@k on ambiguous questions
Tables and diagrams	Multimodal RAG	Accuracy on visual and document questions
Codes and identifiers not found	Hybrid Search	Recall@k and nDCG@k
Relations between entities	GraphRAG	Accuracy of relations and of multi-hop questions
One query is not enough	Agentic RAG	Resolution rate, cost, and latency
Parallelizable investigations	Multi-Agent RAG	Quality versus a single agent
Loss of continuity	Memory RAG	Correct retrieval and privacy

In a real application they get combined. Contextual RAG can enrich the chunks, hybrid search can retrieve the candidates, and an agent can decide whether another query is needed. Nothing obliges you to use all seven.

> **– Going deeper**
>
> with context windows of hundreds of thousands of tokens, the objection is obvious: why retrieve if I can put everything in? Sometimes you can. Anthropic says so in the same *Contextual Retrieval* article: if your knowledge base is smaller than 200,000 tokens (about 500 pages), putting all of it in the prompt, with prompt caching, can be simpler than building RAG. If the on-call assistant only had the four runbooks, I would start there.
>
> With years of tickets and logs, no. First, because they do not fit in any window. Second, because cost and latency grow with every token of every query. Third, because permissions apply per document, and you cannot give the model what the user must not see. And fourth, because models have made worse use of evidence in the middle of a long context than of evidence at the beginning or the end (Liu et al., *Lost in the Middle*, TACL 2024). That study used the models of its time: measure it with yours before taking it for granted.

What It Would Look Like in a Company

This is the on-call assistant end to end, with ingestion, the query, and the controls that cut across them:

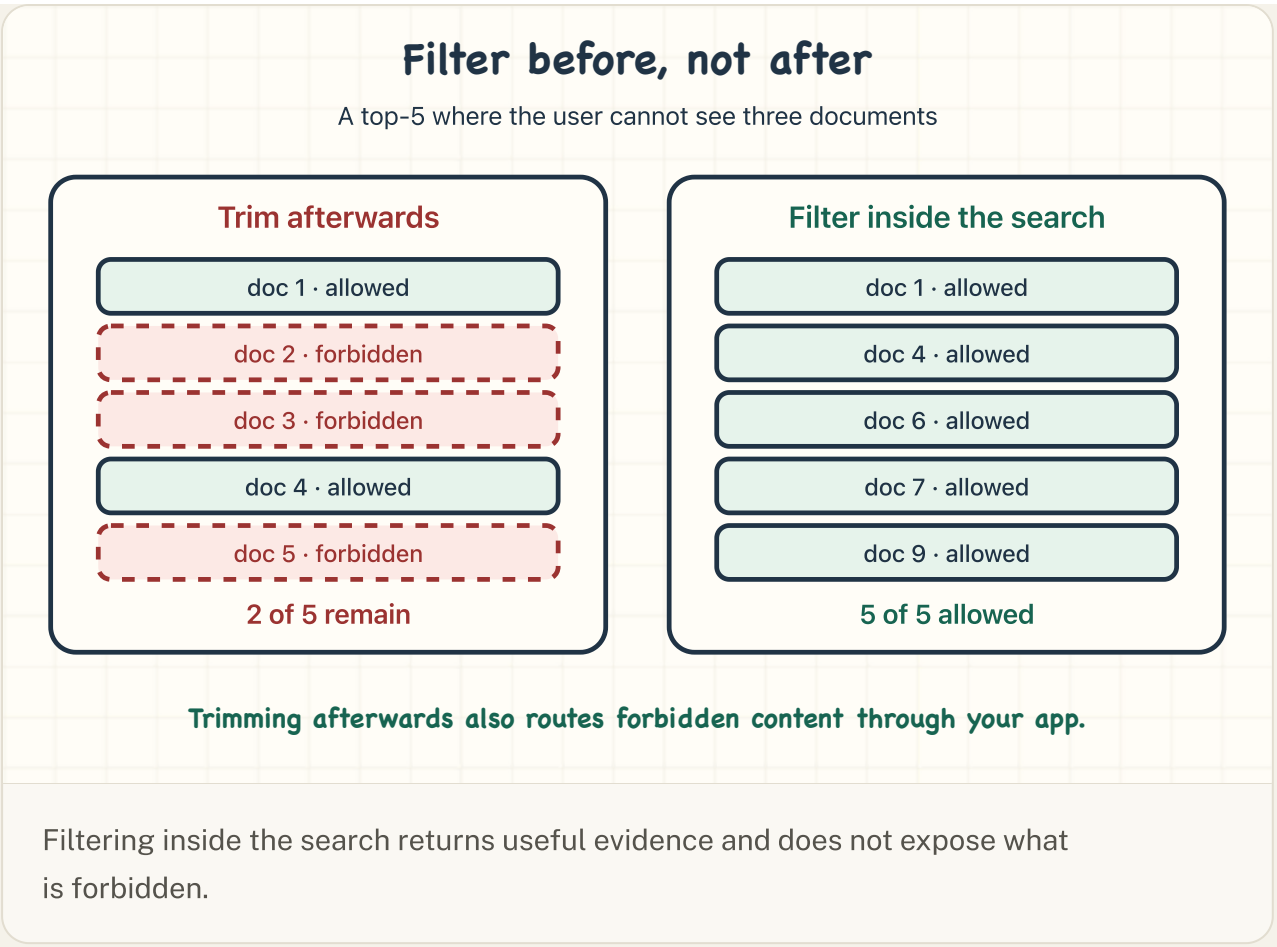


Depending on the problem, visual retrieval, graphs, memory, and agents come in at specific points of this flow.

Permissions Apply During the Search, Not After

In the store's case, security incident tickets can only be seen by the security team. The temptation is to retrieve the top-k and then remove what the user cannot see. It is the easiest control to get wrong, because the wrong version works in the demo.

It has two problems: you end up with less evidence than you asked for, and the forbidden documents have already passed through your application, your logs, and perhaps a cache.



Filtering inside the search returns useful evidence and does not expose what is forbidden.

The right approach is to store each chunk's permissions in the index and filter inside the search itself. Keep in mind that those permissions are only as current as the last sync: if someone loses access in the source system, the index only learns about it when it is updated (Microsoft Learn, *Document-level access control in Azure AI Search*).

An implementation detail: in approximate vector indexes, such as HNSW, a very restrictive filter can return fewer results than requested or lose recall, depending on how your database applies it. Measure retrieval with the users who can see the least, too.

The Non-Negotiable Controls

Control	Why
Authentication and authorization before retrieval	The retriever should not see what the user cannot see.
Filtering by user, tenant, and document	Permissions live in the index, next to each chunk.
The user's identity reaches every tool	Each search, tool, or subagent queries with the permissions of whoever is asking, not with a service account that sees everything.
Retrieved content treated as untrusted	A document, ticket, or log can carry hidden instructions: this is indirect prompt injection (OWASP LLM01).

Control	Why
Evidence versioning and traceability	To know which document, in which version, supported each answer.
Deletion that reaches every copy	A deleted document has to disappear from the chunks, the embeddings, the graph, and the caches. And an answer cache includes the permissions in its key.
Timeouts, limits, and budget	So a query does not cost ten times what was planned.
Metrics and traces for every step	Without them you will not know whether retrieval or generation failed.
Explicit abstention	If evidence is missing, the system says so instead of filling the gap.

> **Watch out:** no instruction in the prompt guarantees that the model will ignore malicious text inside a retrieved document. And in the on-call assistant, anyone who opens a ticket or triggers a log is writing into your context. The defense cannot depend on the model: it depends on the architecture.
>
> - **Least privilege:** the assistant reads; it does not restart services or change configuration.
> - **Human confirmation** before any action with side effects, such as closing a ticket.
> - **Separate** retrieved content from the system instructions and mark it as untrusted.
> - **Sanitize the output:** do not render links or images pointing to unapproved domains, which can be used to exfiltrate data.
>
> OWASP also gives a dedicated entry to the risks of vectors and embeddings (LLM08): leaks between tenants, poisoned data, and permissions misapplied in the vector database.

How to Really Evaluate a RAG System

A convincing answer does not prove quality. Evaluation has to look at three layers separately, because each one fails for different reasons.

1. Retrieval. Did it bring back the right evidence?

Metric	What it measures
Recall@k	What share of the relevant evidence appears in the top k results.
Precision@k	What share of the k results is relevant.
nDCG@k	Whether the relevant items appear near the top: it rewards relevance and penalizes poor position.

An example with the on-call assistant. For the question "why does HikariPool-1 time out in payments?" you marked four pieces of evidence: the PaymentService runbook section, ticket INC-2207, the ADR that set

the pool size, and the October 12 log. If the top ten results contain three of them, Recall@10 = 3 / 4 = 0.75.

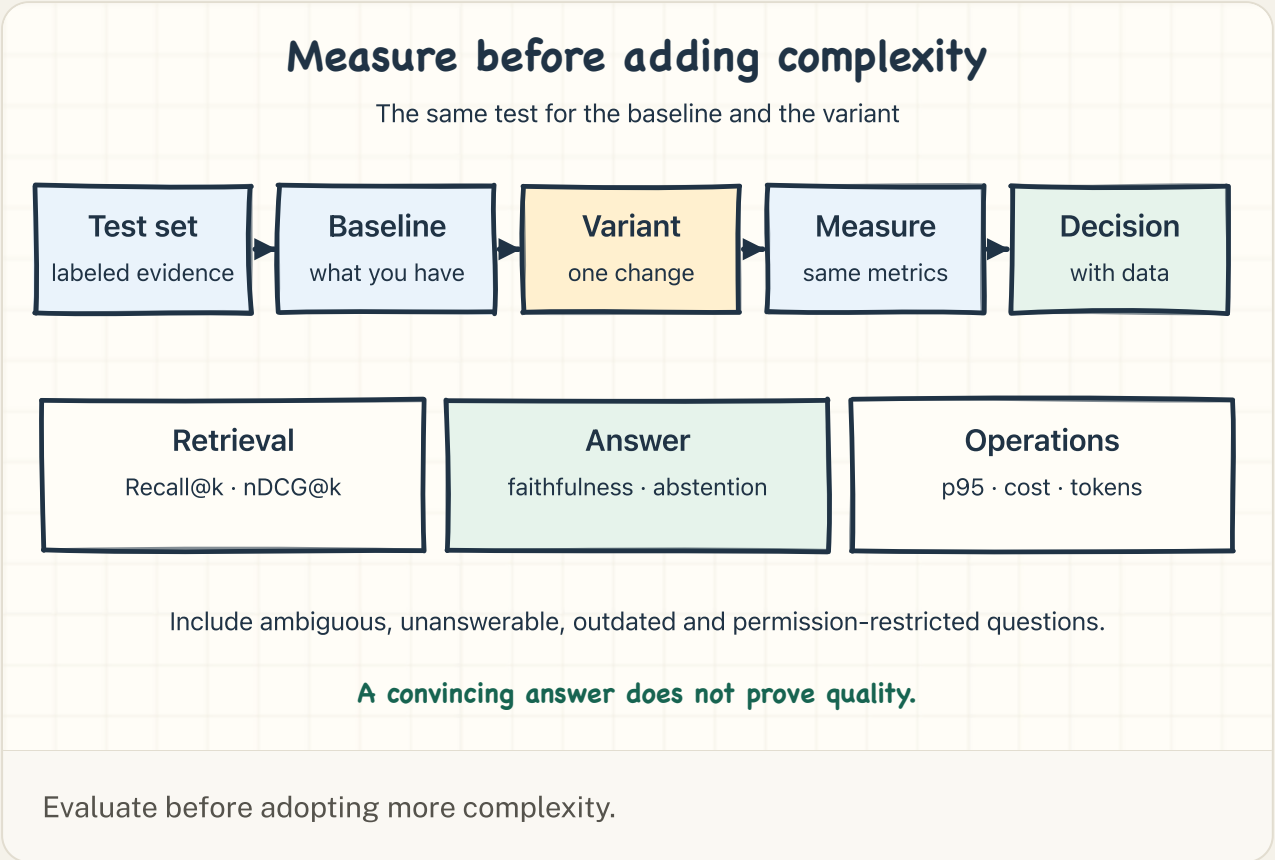
2. The answer. Did it use that evidence well?

Metric	What it measures
Faithfulness	How many claims in the answer are supported by the evidence provided.
Correct citations	Whether each citation really points to the chunk that supports that claim.
Answer relevance	Whether the answer actually addresses the question.
Correctness	Whether it matches a verifiable reference answer or fact.
Abstention	Whether it recognizes when the available information does not allow an answer.

Tools like Ragas implement several of these metrics, and recent versions call answer relevance *Response Relevancy*.

3. Operations. Can it be sustained in production? Measure **p95** and **p99** latency, the time under which 95% and 99% of queries complete. Add the cost per query, the tokens consumed, the number of searches and tool calls, the error and timeout rate, the freshness and coverage of the indexes, and authorization violations.

Automated evaluations that use a language model as a judge are useful, but they are not a guarantee. Check them against reference tests and human review when the risk justifies it.



Evaluate before adopting more complexity.

The Test Set

Everything above needs a set of questions with their evidence marked. In the on-call assistant, the questions come from the incident history. I would start with about fifty and grow from there. They need to include simple, ambiguous, unanswerable, outdated, and permission-restricted questions.

Mark the evidence per document or per section, not per chunk: if you change the chunking, chunk-level labels stop being valid. And there is no universal configuration for chunk size, top-k, number of agents, or iterations: you tune them with your data, your service objectives, and your constraints.

Common Myths

These ideas get repeated a lot when people talk about RAG. They all have some truth in them, and that is why they mislead.

What people usually say	What is true
"Naive RAG is the weakest option and you have to move past it."	For small collections and simple questions, it can be enough. It is the baseline everything else is measured against.
"A multimodal model ends extraction problems."	It changes the kind of error, it does not eliminate it: a vision model also misreads figures and relations.
"GraphRAG always uses communities."	It depends on the implementation. Communities are Microsoft's approach for global questions.
"An agent searches as many times as needed."	It searches as many times as you let it. Without code-enforced limits on time, cost, and iterations, it is an operational risk.
"Several specialized agents outperform one."	Sometimes, and at a cost several times higher. You have to compare it with a single well-designed agent.
"Memory makes the model remember conversations."	The model does not remember: the application stores and retrieves. And without expiry, correction, deletion, and isolation, that memory is a privacy risk.
"With long context you no longer need retrieval."	With a small knowledge base, that may be true. With a large one, size, cost, permissions, and lost attention in the middle of the context are still there.

What the On-Call Assistant Would Look Like

If I had to build this assistant, I would not start with the flashiest architecture. I would go in this order, measuring each step against the previous one:

1. **The baseline and the test set.** Naive RAG and fifty real questions with their evidence marked. Without this, no later step can be justified.

2. **Section-based chunking and metadata:** service, environment, version, and permissions on every chunk.
3. **Hybrid Search with a reranker**, because on-call questions often carry an identifier or an error message.
4. **Contextual RAG**, only if the runbooks still get mixed up after filtering by service.
5. **Agentic RAG, or better a workflow**, for questions like "why did this order fail?", with limits in code.

And what I would leave out until a measurement calls for it. Multimodal, only for the diagrams that exist only as images. GraphRAG, built from the service catalog and not from the text. Multi-Agent RAG, only for post-incident reviews, never at two in the morning. And memory, only for confirmed decisions with an expiration date.

> **Takeaway:** a reliable RAG system is not the one that produces the most convincing answer, but the one that can explain what evidence it built the answer from, what information it is missing, and when it should abstain.

Now think about your own RAG system, or the one you are about to build: what real question does it answer badly today, and which of these seven failures does it fall into? Tell me in the comments; I want to know which failure shows up most.

Sources

1. Anthropic, Contextual Retrieval (2024).
2. Faysse et al., ColPali: Efficient Document Retrieval with Vision Language Models, ICLR 2025.
3. Cormack, Clarke, and Büttcher, Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods, SIGIR 2009.
4. Elastic, Hybrid search.
5. Microsoft, GraphRAG.
6. Anthropic, Building effective agents (2024).
7. Asai et al., Self-RAG (2023), and Yan et al., Corrective Retrieval Augmented Generation (2024).
8. LangGraph, Agentic RAG and Memory overview.
9. Anthropic, How we built our multi-agent research system (2025).
10. Liu et al., Lost in the Middle: How Language Models Use Long Contexts, TACL 2024.
11. Ragas, Available metrics.
12. OWASP, Top 10 for LLM Applications 2025.
13. Microsoft Learn, Document-level access control in Azure AI Search.