AI-CHITECT · WORKING PAPER · V6

Retrieval is a memory problem

Why RAG keeps breaking, and a design built around the way a language model actually reads.

By Rob Vugts · AI-chitect August 2026 ~20 min read

THE SHORT VERSION

The usual way to give a language model your own documents is called RAG: chop the documents into pieces, find the pieces that look most like the question, paste them in, and hope for the best. This paper argues that the problem isn't the pieces — it's the goal. A model can only read a small amount at once, so the real job isn't "find text that looks similar to the question." It's "decide what deserves a place in the small space the model actually reads." Once you see it that way, the design almost writes itself: do the heavy thinking up front when you load the documents, keep a well-organised store, let the model look things up as it reasons, clean up what it finds before it reads it, make it cite its sources, and let the whole thing learn from what worked. We'll walk through each part — what it does, what it costs, and where it still falls short.

WHO THIS IS FOR — AND WHEN IT'S WORTH IT

This is aimed at knowledge that **changes over time and has to be right**: prices, tax and legal rules, contracts, policies. Places where a confident wrong answer is expensive and you have to be able to show where the answer came from. It is *overkill* for a simple FAQ bot, where a good search plus a "cite your source" rule already does the job. Build only as much of this as your problem actually needs.

First, how RAG works today

If you already know what "chunks" and "embeddings" are, this just sets the words straight. Skip to §2.

A language model only knows what it learned during training, and it can only read a limited amount of text at once — that limited space is called its *context window*. RAG (retrieval-augmented generation) is the standard trick for giving it *your* documents without retraining it.

It has four steps. **Chop** every document into small pieces (chunks). **Index** each piece by turning it into a string of numbers that captures roughly what it's about (an embedding), and store those. When a question comes in, **find** the handful of pieces whose numbers look most like the question. Then **answer**: paste those pieces in front of the question and let the model reply.

It works, and for a while it was the whole game. But look at the shape: every important decision is made *before the model is even involved*, and the lookup happens *once*, based only on the user's opening words. That shape is behind nearly every problem in the next section.

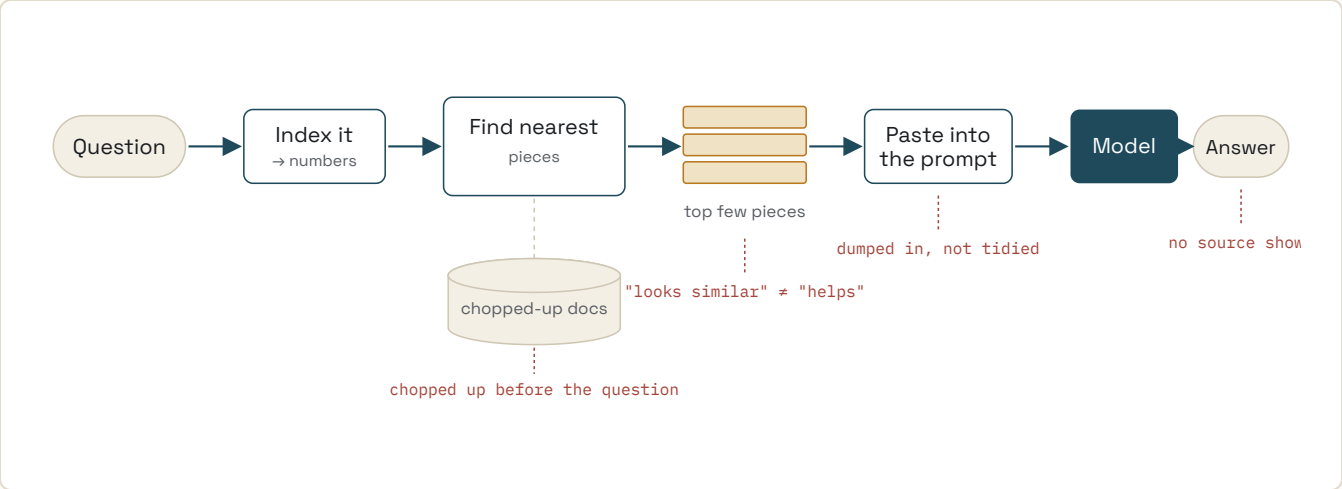

Figure 1 — RAG today. A one-way line. Every choice is fixed before the model runs, the lookup happens once, and the red notes show where that hurts.

§2

The one idea this paper is built on

Nearly every RAG problem comes from the same root. Today's systems chase one thing: *find the documents that look most like the question, once.* That made sense when the model could only read a tiny amount and looking things up was the only way in. It's the wrong goal now.

Flip it around. Don't ask "what looks like the question?" Ask: **what does this model need in front of it to give an answer it can stand behind — including the answer that the documents simply don't cover it?**

THE CORE IDEA

A model can only read a small amount at once, and that space is precious. So the job isn't finding text that **looks similar** to the question. It's deciding what **earns a place** in the small space the model reads. Get that goal right and everything else — the up-front prep, the organised store, the clean-up step, the source-checking — follows from it.

This isn't a proof, it's a way of seeing. But it earns its place, because it lines up the whole history as one story. Old search engines returned *documents*. RAG returned *pieces of documents*. Both just *fetch*. The step this paper names is from fetching to *managing*: deciding what belongs in the model's small reading space at each moment, and keeping everything else in cheaper storage until it's needed.

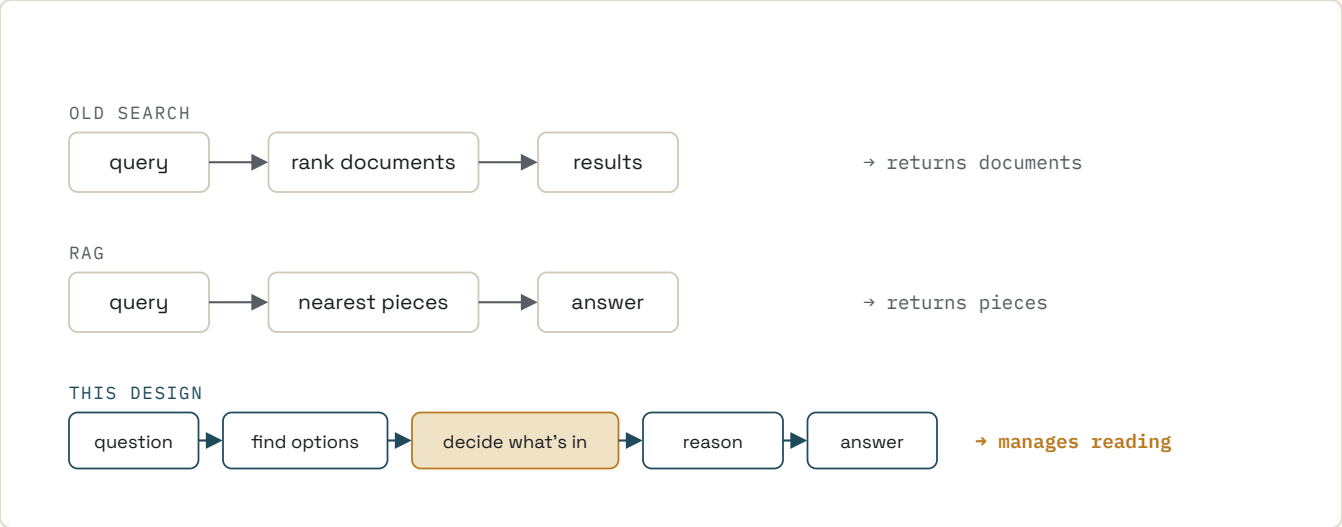

Figure 2 — The shift. Search returned documents; RAG returns pieces of them; the next step decides what actually goes in front of the model. "Decide what's in" (amber) is the new move neither had.

If the model's reading space is the scarce thing, there's a familiar picture that fits: the way a computer's memory is arranged. A processor has tiny, lightning-fast registers, then cache, then main memory, then a big slow disk. The whole craft is moving the right bits up to the fast level at the right moment and leaving the rest below. Line that up with retrieval and the design falls out.

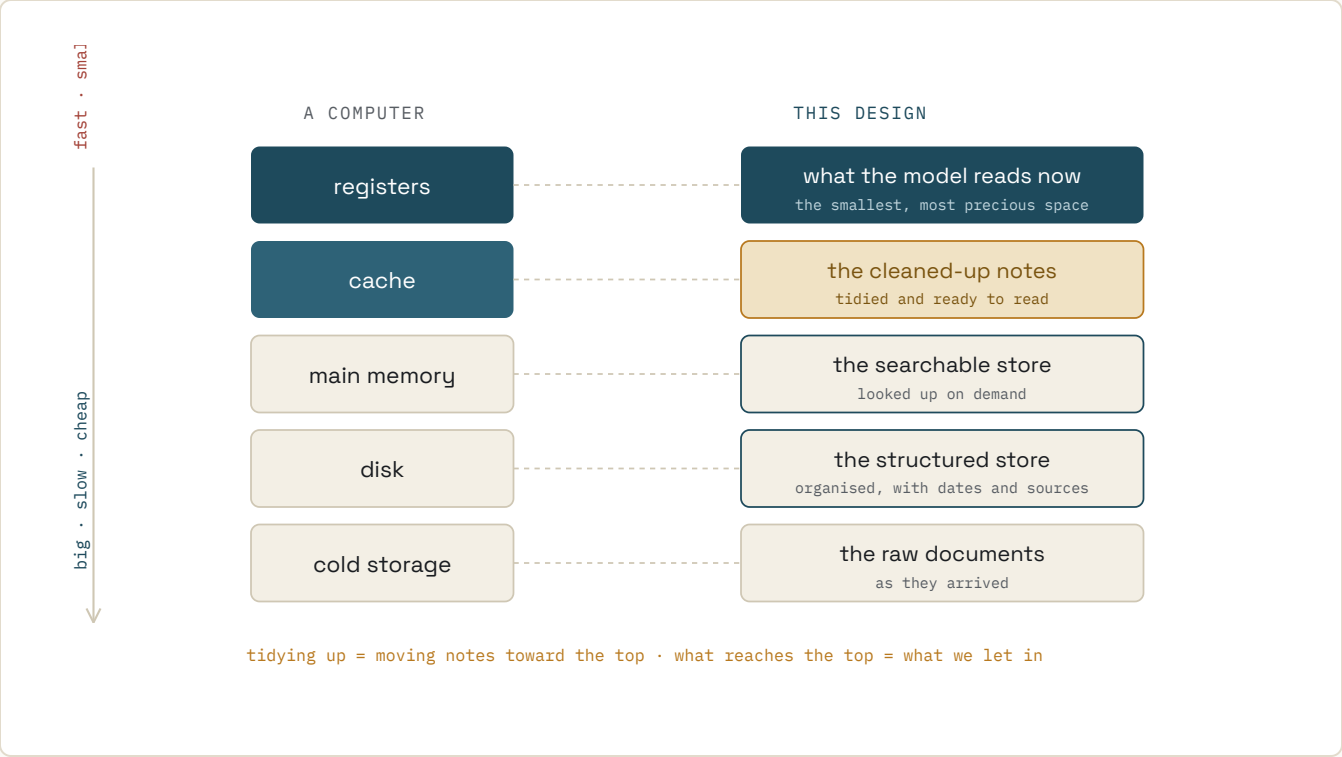

Figure 3 — Retrieval as a memory ladder. You don't load the whole disk into the registers; you don't pour every document into the model. Tidying up is moving notes toward the top, and letting in is deciding what reaches it. *It's an analogy, not a perfect match* — but as a way to think, it explains why each part below exists.

The rest of the paper is the two halves of that idea. §3 is what goes wrong when you chase "looks similar." §4 onward is what falls out when you chase "earns a place."

§3

Six ways today's RAG breaks

There are dozens of ways RAG goes wrong. Almost all of them come back to six root causes — each one a symptom of chasing "looks similar" instead of "earns a place." Grouping them like this matters: every one is answered by a part of the design in §7.

1 · IT DECIDES EVERYTHING TOO EARLY

The important choices are locked in before the question arrives

Cutting documents into fixed-size pieces splits tables and sections and separates facts that belong together. One way of indexing is used for every question. Relationships between facts get flattened into a single "similarity" score. And the lookup fires on the user's opening words — not on what the model turns out to be missing halfway through thinking.

2 · "LOOKS SIMILAR" ISN'T "HELPS"

Matching the wording isn’t the same as being useful

A piece can match the words of the question closely and still not help. Near-duplicates and re-wordings pile up and drown out the one fact that was actually missing. Questions that need two or three facts joined together lose the steps in between. Exact things — reference numbers, error codes, article numbers — that a plain keyword search would nail get blurred. Tables and forms fall apart into meaningless fragments.

3 · NO SENSE OF TIME OR TRUTH

The pieces don’t know when they were true, where they came from, or whether the answer is even in there

"Freshness" is "we re-index now and then," and there's no difference between *when a rule applied* and *when it was written down* — so a 2024 rule gets used for a 2026 question. Two pieces that contradict each other get pasted in side by side and quietly averaged into confident mush. The model's own guesses can quietly overwrite what the documents say. And worst of all, the system can't tell "this isn't in the documents" from "I didn't find anything," so it makes something up rather than admitting the gap.

4 · IT DUMPS INSTEAD OF TIDYING

Nothing prepares the pieces before the model reads them

Too many pieces crowd the model's attention and bury the useful one in the middle, where models read least reliably. The order is accidental. Answers rarely say which source each claim came from, so a guess looks just as trustworthy as a fact. And because the line between "instructions" and "information" is just a habit, not a real boundary, text hidden inside a document can act as an instruction to the model — an attack smuggled in through the documents themselves.

5 · IT CAN'T REMEMBER OR IMPROVE

The lookup is a fixed recipe that never learns

Every question starts from scratch; nothing is kept from one step to the next. It's one shot — no chance to rephrase and dig deeper when the evidence is thin. And nothing checks whether a piece it pulled in actually made the answer better, so the system never learns from its own history. Worse, a setup tuned for one model quietly breaks the next time you switch models — which you will, every few months.

6 · IT'S HARD TO RUN AND TO TRUST

Safety and accountability were never part of the plan

Accuracy drops as the pile of documents grows. Permissions ("who is allowed to see what") get bolted on after the lookup, where they're easy to get wrong. Cost and speed are murky. Poisoned documents are a real risk. And when it fails, it fails as a smooth, confident, wrong answer with no trail back to where it came from.

§4

Retrieval as a bouncer at the door

Accept the core idea and the lookup's job changes completely. It stops being a thing that hunts for answers and becomes a **bouncer at the door** of the model's reading space: it decides what gets in, arranges it neatly, and — importantly — is allowed to say "nothing here qualifies" instead of padding the room with filler. "What matches

the question?" becomes "what does this model actually need to give an answer it can defend — including saying the documents don't cover it?"

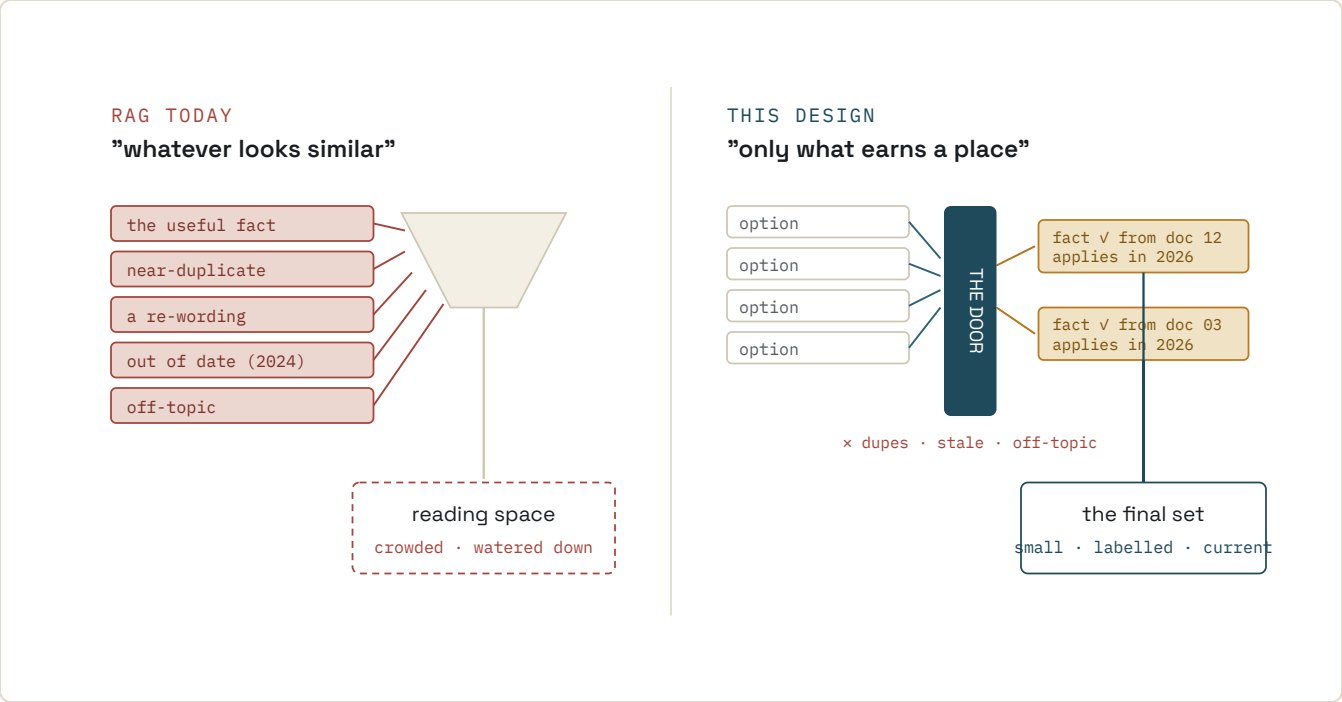

Figure 4 — A door, not a hunt. The door lets in only pieces that earn a place, each labelled with where it came from and when it applies, and turns the rest away — filling the "cache" from Figure 3 on purpose.

A door is only worth having if it keeps its word. So the design makes a few plain promises that must hold every time, or it's built wrong:

THE PROMISES THE SYSTEM KEEPS

- Every piece that gets in says where it came from.
- Every fact about a rule or price says when it applies.
- Nothing goes into the answer that can't be backed up — cite it, flag it as the model's own guess, or say "not in the documents."
- Text from the documents is treated as information, never as instructions.
- The same question over the same store gives the same "what got in" decision.
- Permissions are checked before anything is let in, not filtered out afterward — though what the user is *told* when the answer itself is restricted is a separate call (see §11).

§5

The design, part by part

Do the heavy thinking up front, keep an organised store, let the model look things up as it reasons, tidy what it finds before it reads it, make it cite sources, and let it learn. A fast lane keeps simple questions cheap. Read it from the bottom up.

BEFORE ANYTHING ELSE · YOU NEED A WAY TO MEASURE

None of this works — or is honest — without a way to check it. Before you build a single part, set up tests that measure: does the answer actually match its sources? Does the cited source really back up the claim? Does it get the dates right? Can it handle a question that needs several facts joined up? Does it resist hidden instructions? Does it say "I don't know" when it should? Without these tests, the learning part has nothing to learn from, and you can't tell a real improvement from a lucky demo. And be honest about what this costs: for a regulated corpus it is the *hardest* part, not a warm-up. Building the graded test set means domain experts hand-labelling what the right answer *was on a given date* — bitemporal ground truth over tax or legal rules — which is months of expert work, and it's the real reason most efforts never get past the demo. Budget for it as the entry cost it is.

A fast lane and a careful lane

In front of everything sits a cheap sorter that reads the question and picks a lane. Simple, low-stakes questions take the *fast lane*: a quick search, cite the source, done. Hard, high-stakes, or multi-step questions take the *careful lane*: the full design below. Without this, you build something lovely that's far too slow and expensive on the everyday questions that don't need it. And the sorter's mistakes aren't equal: sending an easy question down the careful lane just costs a little time, but sending a hard rules question down the fast lane gives a confident wrong answer with no date check and no source. So the sorter is built to *err toward caution* — when it's unsure, it sends the question the careful way.

0

The prep work — done ahead of time

The part most teams skip, and the one that makes everything else cheap. When documents come in, do the slow, expensive work once, in the background: read them keeping their structure intact, pull out the individual facts, note which things relate to which, and — crucially — record *when each fact applies* and flag anything that looks like it contradicts something already on file. Every bit of effort spent here is effort you never spend while a user is waiting. This is why the live system can stay quick.

1

The structured store — with two clocks

Your documents kept in an organised form — with their structure, their sources, and the links between facts — rather than a flat pile of numbered pieces. The key trick is **two clocks on every fact**: *when it was true in the world* and *when the system recorded (or removed) it*. A 2024 rule isn't junk — it's right for a 2024 question and wrong for a 2026 one, and the store knows the difference. Those two clocks also give you clean deletion: to honour a "please forget this" request, you mark the fact as removed rather than erasing it, so it stops showing up in answers while the record that it was removed still exists.

2

The search tools

Not one search box but several, which the model can use as it reasons: keyword search for exact terms, meaning-based search for fuzzy ideas, and link-following for questions that need several facts joined up. A quick second-pass re-sort sharpens the results. The model looks something up, thinks, and looks again — guided by what it's actually missing partway through, not just by the opening words. A store you can *explore* beats one you can only *search*.

3

The tidy-up step — the heart of it

The part almost nobody builds. It takes what the search turned up — already filtered to what this user is allowed to see — and gets it ready to read: removes duplicates, puts things in a sensible order, keeps the version that applies to the question's date, and when two sources disagree it *shows the disagreement* ("source A says this, from 2024; source B says that, and it's current") rather than quietly splitting the difference. It also stamps everything as **information, not instructions**, so a document can't sneak a command past the model. This runs *once*, right before the model answers — not on every little step — so the expensive tidying happens a single time per question.

4

The ”show your sources” rule

A rule on the answer: cite the source for each claim, or clearly flag it as the model's own knowledge (a lower bar of trust), or say "I don't have that." A guess can no longer pass itself off as a fact, and when nothing qualifies the model says so instead of inventing something. There's a fourth response the rule has to allow for — *restricted*: the user asked something only restricted content could answer. Whether that's ever surfaced ("there is material you can't access") or hidden (made to look like "not in the documents") is a policy choice, and — crucially — one made *outside* the model, for reasons §11 gets into. And it's not enough that a citation *exists* — a check makes sure the cited source **actually supports the claim**, rather than being a real source stapled to an unrelated answer. Every piece that got in can answer three questions: why are you here, who vouches for you, and how old are you.

5

The part that learns — a later, optional layer

What lets the system get leaner over time, once the rest is solid. Every question leaves a trail — what was searched, what got let in, and how good the answer was. The honest version of this is narrower than it sounds: usefulness isn't really a property of a piece on its own, it's a property of a piece *for a given kind of question*, so there's rarely enough repetition to judge any single piece with confidence — especially across a large organisation asking many different things. What *does* show up often enough to learn from is the frequent, structural junk: near-duplicate sheets, superseded versions, boilerplate that gets pulled in constantly and never once earns its place. For that, take it out and see if the answer gets worse; if it never helps, weight it down. Learn *types*, not individual pieces, and the lesson carries across questions and departments instead of staying stuck to one narrow case. Because it's fitted to one particular model, keep it as a thin, swappable layer, so switching models means re-tuning that layer rather than rebuilding everything. Worth being clear about what this buys: a leaner, slightly cheaper working set on the junk it manages to catch — not trust. The door already refuses to state the unsourced (part 4); this part doesn't make it more honest, only more efficient.

TWO KINDS OF MEMORY, KEPT APART

The **structured store** is the long-lived, shared knowledge — your documents. A separate **scratchpad** holds just this one question's working: what was searched, what was rejected, what's been figured out so far, so the model doesn't repeat itself mid-question. The learning part (5) can later fold useful lessons from the scratchpad into the door's rules. Long-term memory of a particular user across many sessions is a different job, and out of scope here.

A NOTE ON CONFIDENCE — GO BY THE EVIDENCE, NOT THE MODEL'S GUT

Having evidence isn't the same as being sure. The store gives you honest signals to build a confidence marker from: how solid the source is, whether there's an unresolved disagreement, whether the fact is out of date, whether there was thin evidence to begin with. Show *that* to the user ("two current sources agree" versus "one out-of-date source, and there's a conflict"). Don't show the model's own sense of how sure it feels — models are bad at that, which is the same reason they can't reliably tell you what they don't know. Confidence built from evidence is honest; confidence taken from the model's gut is theatre.

WHEN PARTS FAIL, IT SHOULD BEND, NOT SNAP

Every part has a fallback, so the worst case is "plain RAG with citations," never silence. If the prep work fails on a document, you can still find it by ordinary search — just without the extras. If the links are incomplete, search still works. If a fact's dates are missing, the tidy-up step treats it as uncertain and trusts it less, rather than trusting it blindly.

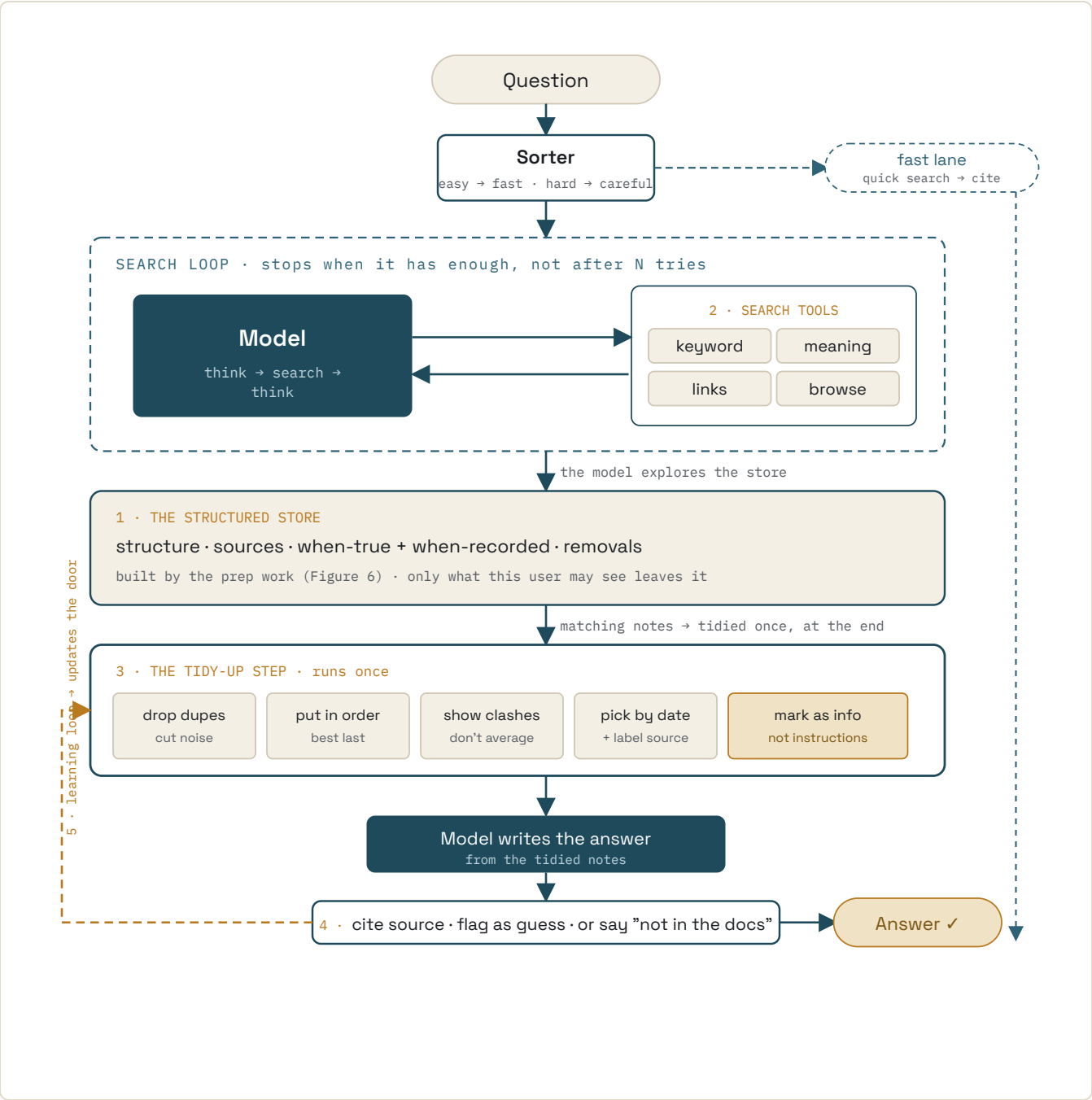

Figure 5 — The live system. The sorter splits traffic: easy questions take the **fast lane** (right) straight to the answer; hard ones enter the loop. The model explores the **store (1)** — built by the **prep work (0)** — using the **search tools (2)**; the **tidy-up step (3)** gets a small, labelled set ready once; the **show-your-sources rule (4)** checks the answer; and the amber **learning loop (5)** feeds back what actually helped.

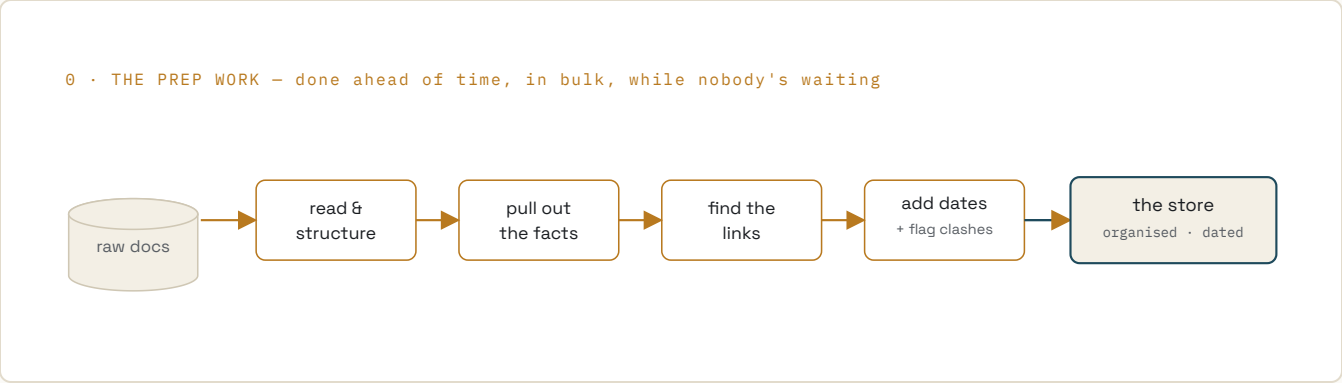

Figure 6 — The prep work. The effort the live system doesn't have to spend. It's a background job you control — which is exactly what lets the live path stay quick, and it's done once per document, then reused for every future question.

The same design, wired up as a system

The figures so far explain the idea. Engineers will want the shape they'd actually deploy — so here it is as a reference architecture, with roles rather than product names, so it doesn't date. Read it for one thing above all: **the seam the whole design turns on.** The expensive work lives on two offline planes — loading documents (0) and, later, learning (5). The shared stores sit in the middle. The live query plane only ever *reads* them. Nothing a user waits on writes to the store — which is exactly what lets the live path stay quick and the records stay trustworthy.

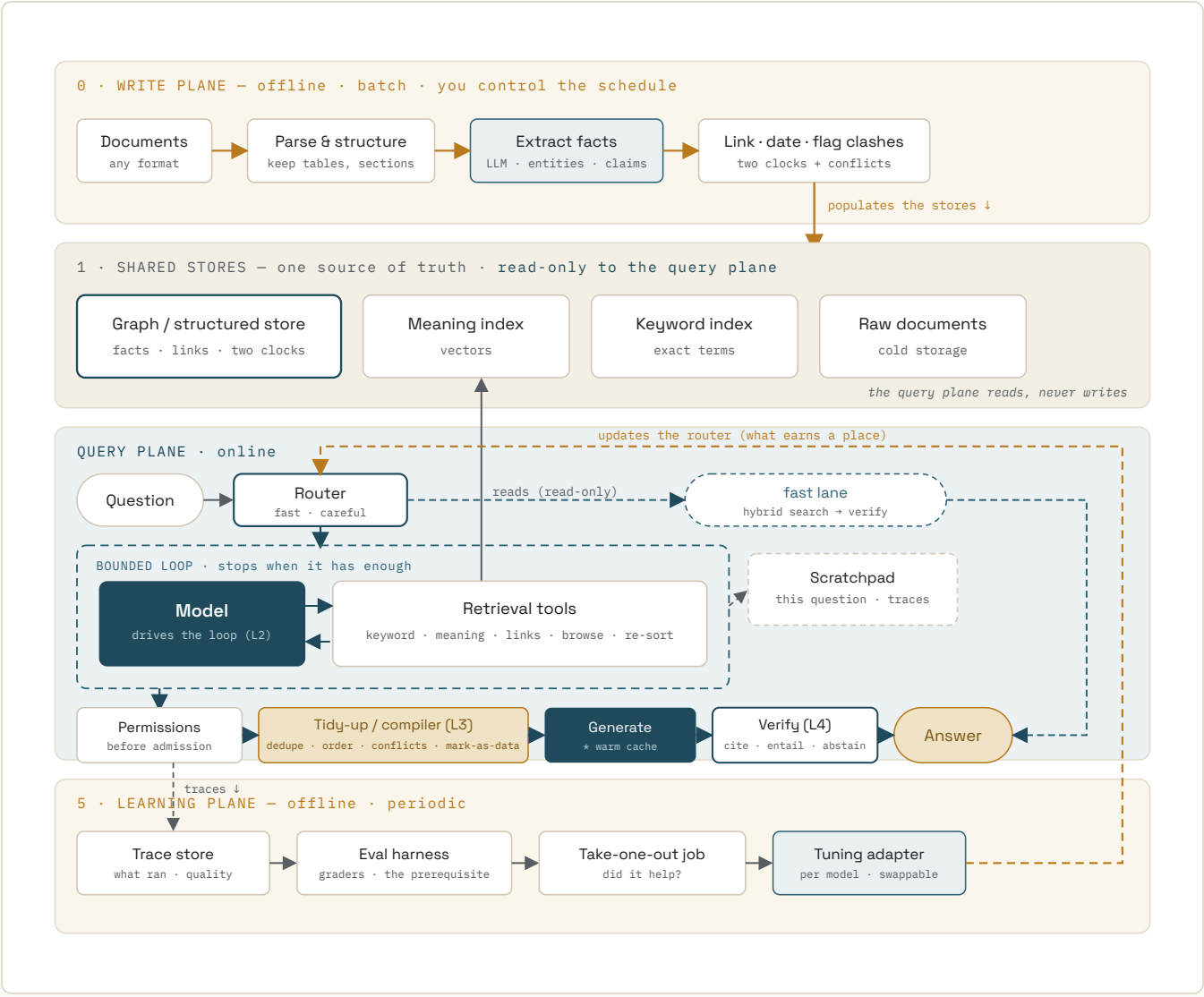

Figure 7 — Reference architecture. The same design as parts you'd deploy. **Solid arrows** are the live data flow; **dashed arrows** are offline or feedback. The three planes never blur: the **write plane (0)** and **learning plane (5)** do the slow work off to the side, the **stores** hold the one source of truth, and the **query plane** only reads them — so nothing a user waits on can slow down or corrupt the store.

REFERENCE IMPLEMENTATIONS — EXAMPLES, NOT REQUIREMENTS

Every box is a *role* you can fill with whatever you already run. The graph / structured store can be a graph database or a plain relational store with date columns (Zep's Graphiti is one ready-made option that already keeps the two clocks); the meaning index, any vector store; the keyword index, a standard search engine; the loop, any agent framework; the eval harness, your own test set. The value is in the arrangement and the offline/online seam — not the brand names, which is why they're kept out of the diagram.

A concrete stack you could build on — August 2026

The section above stays deliberately vendor-neutral, because the design outlives any tool. But a real team wants a starting point, so here's one credible stack as of **August 2026** — with an eye on what a regulated shop can *self-host*. Treat it as a snapshot: this is the one part of the paper guaranteed to date fastest, so check every choice against its own current docs and your own tests before committing.

ROLE IN THE DESIGN	OPTIONS AS OF AUG 2026	IF YOU MUST SELF-HOST (REGULATED)
Structured store · two clocks (1)	Zep's **Graphiti** on Neo4j, FalkorDB or Kùzu — a bi-temporal graph built for exactly this; Cognee or Mem0 as alternatives; or plain Postgres with valid-time / record-time columns.	Graphiti + Neo4j on your own tin, or Postgres for a simpler two-clock store with row-level security.
Meaning index · vectors (1)	**pgvector** in Postgres up to ~10M vectors; **Qdrant** (fast, filtered) or Milvus / Weaviate at larger scale; Pinecone or Turbopuffer if you'd rather it be managed. Embeddings: **BGE-m3**, Voyage, Cohere, or OpenAI.	pgvector or Qdrant (both open-source); BGE-m3 embeddings run on your own GPUs.
Keyword index · exact terms (1–2)	OpenSearch / Elasticsearch (BM25) or Postgres full-text; many vector stores now do hybrid natively. Merge keyword and vector hits with **Reciprocal Rank Fusion**.	OpenSearch or Postgres full-text — both self-hostable.
Re-sort · reranker (2)	**Cohere Rerank 4** or **Voyage rerank-2.5** (managed); **Jina Reranker v3** or **BGE-reranker-v2-m3**	BGE-reranker-v2-m3 (Apache-2.0) on your hardware.

ROLE IN THE DESIGN	OPTIONS AS OF AUG 2026	IF YOU MUST SELF-HOST (REGULATED)
	(open weights) to self-host.	
Search loop · orchestration (2)	**LangGraph** (explicit state + checkpoints give you a natural audit trail), LlamaIndex (retrieval-first), or PydanticAI (typed, schema-safe); vendor SDKs from Anthropic, OpenAI, or Google if you live in one ecosystem.	LangGraph self-hosted; or a governed platform — IBM watsonx Orchestrate, AWS Bedrock AgentCore, Google Vertex Agent Builder.
Reasoning + writing model (2, 4)	Closed frontier-class: **Claude Opus 5**, GPT-5.6, Gemini 3.1 Pro. A smaller, cheaper model (Claude Haiku, Gemini Flash) is fine for ingest extraction and drafting.	Open-weight self-host: **Qwen 3.5** or **Llama 4** for data-residency rules — weights run on your own hardware, nothing leaves. **DeepSeek** models are open-weight too, and self-hosting them sidesteps the 2026 restrictions several governments placed on DeepSeek's *hosted* service — mostly government-device and consumer-app bans, driven by data reaching servers under Chinese jurisdiction — but a regulated institution still needs a separate provenance and policy sign-off for a Chinese-origin model, air-gapped or not, and that's a call for your risk function, not this paper.
Tidy-up / compiler (3)	Mostly your own code. For the model-	Runs inside your app; keep the model-assisted steps logged.

ROLE IN THE DESIGN	OPTIONS AS OF AUG 2026	IF YOU MUST SELF-HOST (REGULATED)
	assisted parts — judging a real conflict, checking support — an NLI model or a tightly-constrained LLM-as-judge.	
Verify · show sources (4)	Groundedness / faithfulness scoring via **Ragas** metrics, an NLI model, or LLM-as-judge; live guardrails via Galileo.	Ragas or a self-hosted NLI model.
Eval + traces (the prerequisite; 5)	**Ragas** (reference-free RAG metrics), DeepEval (CI-style), or TruLens (span-level) for scoring; **Langfuse**, Arize Phoenix, or Braintrust for traces and production monitoring.	Langfuse (MIT) + Ragas, both self-hostable; Braintrust or Galileo if you need a managed audit trail.

Two cross-cutting pieces don't need their own row. **Warm caching** is native to the major model APIs now — switch on the provider's prompt/context cache and keep the changeable content last. And **permissions** belong in the store, not a bolt-on filter: Postgres row-level security (via Supabase or your own roles) or per-tenant indexes are the usual way to keep "check access before admission" true by construction. Once more, though: this list is an August-2026 snapshot — the design is the durable part; the names will churn.

Roughly what it costs to run

A design should show it has thought about cost. These are rough shapes, not exact figures — but they show where the effort goes.

PART	ROUGH COST	NOTE
0 · Prep work	`once per document`	Spread across every future question — *as long as documents change slowly*, which is this design's home turf.
1 · The store	`fast lookups`	Well-indexed, so it slows only gently as the pile grows.
2 · Search	`capped per question`	The loop stops when it has enough — that cap is what keeps it bounded.
3 · Tidy-up	`grows with what's found`	The reason to keep the found set small. Runs once per question.
Answering	`set by the reading budget`	How much you put in front of the model is the lever — the whole point of the door.

One honest correction: the prep work pays off *because* the documents change slowly. If they change constantly — news, filings, chat — the prep work isn't a one-off but a never-ending job, and keeping it up to date can cost as much as answering questions. That's not a flaw; it's the edge of where this design fits.

§6

A worked example

Designs get nodded at and not built. So here's one real question, start to finish. The details are made up for illustration.

The setup. An assistant inside a bank is asked: "Does the monthly fee apply to this savings account?" Two things make it tricky. First, the fee rule *changed at the start of 2026*, so the answer depends on which year the question is about. Second, the document store still holds *last year's rate sheet next to this year's* — and they disagree.

`Prep` **Done earlier, in the background**

Both versions of the fee rule are stored with the dates they apply (old rule: through 2025; new rule: from 2026). Both rate sheets are linked to the same product, and the two conflicting figures are flagged as a possible clash. None of this happens while the user waits.

`Sorter` **Which lane?**

Money involved, a rule involved — this goes down the careful lane, not the fast one.

`Search` **Look it up**

A keyword search finds the exact fee clause; a meaning-based search finds the surrounding context; following the link from the product reaches the current rate sheet.

`Tidy-up` **Get it ready, once**

The question is about 2026, so the tidy-up step keeps the 2026 rule and drops the 2025 one. It removes the duplicate rate-sheet rows, and instead of quietly averaging the two conflicting figures, it *shows the clash*: "sheet A says X, from 2024; sheet B says Y, and it's current." Everything is labelled with where it came from and when it applies.

`Answer` **With sources**

The answer cites the 2026 rule and the current rate sheet. If the current figure had been missing, it would say "I don't have the current fee on file — please confirm" instead of guessing.

`Learn` **In the background, later**

Over time the system notices the old rate sheet never once improved an answer, and quietly stops surfacing it.

That's the whole idea in one question: the right rule for the right year, a conflict shown instead of hidden, and "I don't know" instead of a confident guess.

Which part fixes which problem

The six problems from §3, matched to the parts that answer them. Notice the caveats: in a few cases the design *moves* a hard problem somewhere you can work on it, rather than making it vanish. Saying so is what separates a design from a sales pitch.

PROBLEM	ANSWERED MAINLY BY	HONEST CAVEAT
1 · Decides too early	Prep work + the store (kept organised) + search tools (strategy chosen as you go)	*The organising happens up front, and for ordinary prose that isn't free — it's the prep-work bill.*
2 · "Similar" ≠ "helps"	Search tools (keyword + meaning + links + re-sort), on the structure the prep work built	*Link-following only helps if the prep work drew the right links. The effort moves up front; it doesn't disappear.*
3 · No time or truth	The store (two clocks, sources) + tidy-up (shows clashes) + the sources rule (cite or say "don't know") + the learning part	*Spotting that two things actually conflict is itself unsolved. The tidy-up gives you a place to show a clash — not a way to always catch one.*
4 · Dumps, doesn't tidy	The tidy-up step (the door, de-duping, ordering, info-not-instructions)	*The tidy-up is a real live step with its own speed cost — often the hardest part to get right.*
5 · Can't learn	The learning part (take-one-out testing on *types* of spans + a swappable tuning layer) + the scratchpad	*Only the frequent, structural junk gets enough repetition to learn from — a single piece rarely gets asked about the same way often enough, especially across a large, varied organisation. Buys efficiency, not trust; long-term user memory is a separate job.*
6 · Hard to run and trust	The sources rule (a trail) + tidy-up (permissions checked first) + the store (updatable, clean deletion)	*The design points at where safety goes. It isn't a full permissions product.*

What it costs to build

Benefits without a bill are marketing. Here's how hard each part is, where the effort goes, and how you'd know it's actually working. Read the difficulty column as the order to build things in.

PART	DIFFICULTY	WHERE THE EFFORT GOES	HOW YOU'D KNOW IT WORKS
0 · Prep work	`Hard`	Building a real pipeline that reads documents and pulls out facts, links, and dates.	The facts and links come out right on a sample; harder multi-step questions start working.
1 · The store	`Med–hard`	Designing the store and keeping structure, sources, and two sets of dates straight.	Date questions come out right; every fact shows its source; deleted facts really stop appearing.
2 · Search	`Medium`	Several search methods kept in sync, plus a re-sort step.	Each method finds what it should; the model picks the right one; multi-step questions beat plain top-few.
3 · Tidy-up	`Hard`	A live step with a speed budget: de-dupe, order, show clashes, check permissions, mark as info. Often the hardest part.	Duplicates drop; order changes the scores; hidden-instruction attacks get blocked.
4 · Sources rule	`Easy–med`	Checking answers cite real, supporting sources; a little discipline in the prompt.	Citations are present *and* actually back the claim; it says "I don't know" on an empty set.
5 · Learning	`Hard`	A test setup, a judge, background compute, and re-tuning on every model switch. Partly research.	The tuned door beats a plain "most similar" baseline on held-back questions.

The **sorter** is cheap — build it early. Otherwise: **the sources rule first** (cheap, and "shows its sources" is a feature you can ship day one), then **search** (mostly off-the-shelf), then **the prep work and the store** together (one investment), then **the tidy-up step**. **Learning** goes last on purpose, not by default: the system's trustworthiness doesn't depend on it — that's already carried by the sources rule and the permissions gate — and its real yield is narrower than it first sounds (see part 5), so it's the one part worth building only once everything with a surer payoff is already standing. Most of parts 1–4 are known techniques the average system just skips. The learning part, and truly knowing what the model doesn't know, are the parts still genuinely open. One caveat outranks the whole order, though: the difficulty table above rates the six parts, but the single hardest and least-automatable piece for a regulated buyer isn't in it — it's building the graded, date-aware test set the whole thing is measured against (see §5). That's expert-labour, not engineering, and it's the real first line of any honest budget.

§9

Making it fast enough

A weakness the grand idea glosses over: the careful lane is slow. Owning that is part of the design.

On a hard question, the careful lane does a lot — sort, search several times, tidy up, write, check sources — and that can take several seconds, sometimes more than ten. For slow-moving, high-stakes work, a few seconds for a careful, up-to-date, sourced answer is usually the right trade — the expensive mistake in tax or law is a *fast wrong number*, not a slow right one. But "acceptable" isn't "ignore it." Four levers, biggest first.

1 · The sorter does the most. Most questions are easy and never enter the loop. Make the fast lane quick and you've handled the bulk of traffic; the full design runs only for the minority that earns it.

2 · Keep the warm part warm. The fixed parts — the standing instructions and the steady core of the store for an active session — barely change. Modern tools let you keep that already-read part "warm" so a follow-up question only pays for the bit that changed. It cuts both cost and the wait before the answer starts. A rule of thumb falls out of it: put the things most likely to change *last* in that warm part.

3 · Drop stale bits from the warm part cleanly. Keeping things warm risks serving something out of date. The two clocks fix it: when a fact is removed or replaced, drop the warm bits that leaned on it. One honest limit — the warm part is a running start, so dropping something early forces re-reading everything after it. That's exactly why you put the changeable things last.

4 · Draft-and-check — with a warning that matters here. There's a real trick where a small fast model drafts while a bigger one checks, roughly halving the wait. Note the order in the real version: it checks *before* it shows anything. The tempting shortcut — show the draft instantly and fix it if the check fails — breaks this design's core promise in exactly the places it's meant for. A wrong tax or legal figure shown for two seconds and then corrected isn't a speed win; it's a wrong answer someone may already have acted on. So here, checking comes before showing. Show-then-fix is fine for low-stakes chat — but that's the everyday case the fast lane already covers.

There's also a genuine middle ground the show-then-fix idea misses: **show the work, not the claims.** While the careful lane runs, the user can watch what's happening — "found the 2024 and 2026 versions; working out which applies; checking the figure against the current sheet" — which kills the dead wait without ever showing an unchecked fact. Showing what the system is *doing* is honest; showing what it hasn't *checked* is not.

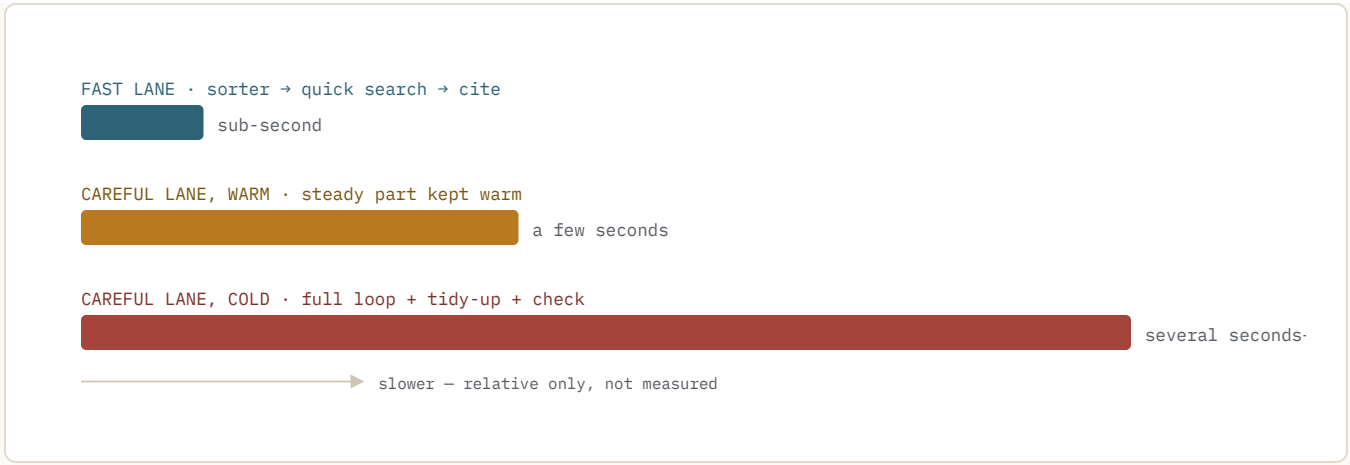

Figure 8 — Relative speed (schematic, not measured). No numbers here on purpose: the paper ships no benchmarks, so this shows only the *shape* — the sorter keeps most traffic on the fast bar, keeping the steady part warm pulls the careful lane down, and the tidy-up and source check are the price of the promise, paid only where it's needed. Bar lengths are illustrative order-of-magnitude, not timings.

THE OTHER APPROACH — JUST LOAD EVERYTHING

A different school skips the search loop almost entirely: load the whole set of documents into one big reading space and keep it warm. That's what NotebookLM popularised for research over a *fixed, bounded* set of sources. It's fast and simple, and for a small, stable set it's often the right answer — the fastest search is no search. It isn't this design's target for three reasons already given: it gets crowded and less reliable as the pile grows, it has no built-in sense of dates or permissions for a big *changing* collection, and "keep it all loaded" doesn't answer "which version applied on the date in question?" The two ideas meet in the middle, and this design already sits there: keep the steady part warm (their insight), and be picky about the changeable rest (ours).

§10

How this compares, and the obvious objections

Why ”just use a knowledge graph” isn’t enough

A knowledge graph — facts stored as things and the links between them — is the comparison people reach for. It's a great *search tool* (part 2): it's how you follow links for a multi-step question. But that's one part. On its own a graph doesn't tell you

which version applies on a given date, doesn't tidy anything up before the model reads it, doesn't decide what earns a place (it just returns neighbours), doesn't make the answer cite its sources, and doesn't learn what actually helped. The door and the tidy-up are separate from link-following — you put them *on top of* a graph. The graph is a tool inside part 2, not the whole design.

Where this actually sits — and what's genuinely new

Honesty means naming what already exists, because most of the pieces do. **Zep's Graphiti** already stores facts with two clocks (when-true and when-recorded), pulls facts out of documents as they arrive, and marks facts as removed rather than deleting them — that's the prep work, the store, and clean deletion, in production today. **Microsoft's GraphRAG** already pulls out facts and relationships when it loads documents. Keyword-plus-meaning-plus-links search with a re-sort is a standard toolkit. So parts 0–2 are largely known techniques the average system just hasn't adopted.

And the *framing* isn't new either — this is the honest part to own. **MemGPT** (Packer et al., 2023, now Letta) already cast the context window as RAM and external memory as disk, with the model paging in what it needs — that is the exact metaphor behind Figure 3. "**Context engineering**" (the term Anthropic and others made standard through 2025) is by now the accepted name for "curate what earns a place in the window" — my §2 objective, near-verbatim. And putting the model in the retrieval loop is **agentic RAG**, with its own surveys. So neither the components nor the memory frame are the contribution, and a reviewer who says "this is context engineering plus MemGPT's metaphor plus agentic RAG" is not wrong about those parts.

So what *is* the contribution? Something narrower, and I think more durable for being honest about its size. Not the memory frame and not the objective — but four things the frame's usual literature doesn't do. The **tidy-up step as a distinct, guarded, mostly-deterministic stage**, with the "information, not instructions" boundary as a real job rather than a hope (plenty of systems compile context; few

treat the compile as a governed stage with an injection boundary of its own). The **show-your-sources rule with a real support check**, not merely a citation. The **present-but-restricted handling** of §11 — the third ending, and keeping that decision out of the model. And, most of all, carrying the whole frame into **changing, regulated document retrieval**, where the two clocks, the source trail, and the governance actually bite. Almost all context-engineering work is about agents; very little is about governed retrieval over a corpus that changes and has to answer to a regulator. That gap is the real contribution — the coherence and the application, not the inventory.

There's a second family worth naming, because it answers the same complaint from the opposite direction: the **compiled-wiki pattern** (Karpathy's "LLM Wiki" and the many tools built on it). It shares this design's prep work — read a source, pull out the facts, note where they clash — but then makes a different bet. Where this design assembles the working set *fresh* from raw, citable facts on every question, the compiled wiki treats the *synthesis itself* as the durable thing: a good answer gets written back as a new page, so the next related question reads an already-worked-out page instead of re-deriving it. That compounds beautifully — insight accumulates instead of being rebuilt each time — and for personal research or a slow-moving knowledge base it may be the better shape.

But it carries a risk this design was built to avoid, and it's worth being plain about because it's the crux of the trade. A compiled wiki answers more confidently than raw retrieval — that's the point of it — but confidence drifts loose from whether the answer is actually grounded, and in a *compounding* store that failure feeds itself: a fabricated synthesis gets filed back as a page, and from then on it *is* a source. This design refuses that by construction — the model's own synthesis never becomes a citable fact; every admitted span traces to a real document, with its source and its dates. So the two aren't rivals so much as two answers to "stop re-deriving from scratch," tuned for different worlds: accumulate synthesis for speed and reach, or re-derive from sources every time to keep provenance clean. A personal wiki can live

with a wrong page it corrects next week; a bank answering "does the fee apply" cannot let a fabrication quietly become the truth. That's the whole reason this design lands where it does.

Google's **Open Knowledge Format** (OKF, mid-2026) is worth a word here, because it's this same wiki pattern given a portable file convention — a directory of markdown files, each a "concept," with a little YAML on top (type, title, tags, a timestamp) and plain links between them. It's explicitly a *format, not a service*: no store, no retrieval, no runtime. Which places it cleanly in this design — it's a tidy, version-controlled, vendor-neutral way to *serialise* the document-shaped parts of the structured store (part 1) and to ship them between systems, and their reference "enrichment agent" that drafts a concept per table is a nice instance of the prep work (part 0). Two of this design's promises even get easier: knowledge that lives as plain files in git is knowledge whose every change is an auditable diff, and files move across tools without lock-in. But note what OKF carries per concept — a single *timestamp*, and no notion of source-versus-synthesis. It has one clock where this design needs two, and it's provenance-agnostic where this design insists a fact trace to a real document. So it slots in *underneath* — a good serialization for part 1's contents — but the two clocks, the source trail, and the "synthesis is never a citable fact" rule are exactly what you'd add around it. It standardises the wiki pattern; it doesn't make it safe for a regulated answer. That remains the work.

One of those existing systems also carries a warning. Graphiti deliberately keeps the model *out* of the search loop — plain search only, very fast — for speed and reliability. This design puts the model back in the loop so it can adapt mid-question. That's a real trade: you buy flexibility and pay in speed and in the reliability risk named first in §11. A production system chose the other side of that trade for good reasons; the sorter and the "stop when you have enough" rule are how this design tries to afford its choice.

How you'd check the claim

Plainly: this is a design paper, and it ships *no benchmarks yet*. What it would be judged on is the test set from §5 — does the answer match its sources, does it get dates right, does it handle multi-step questions, is it efficient, is it fast enough, does it say "I don't know" when it should, does it resist hidden instructions. The core claim is small and can be proven wrong: *a small, tidied, "earns-a-place" set beats an equally-sized pile of "most similar" pieces on accuracy and on getting dates right, at the same or better speed.* If it doesn't, the tidy-up step is just for show and the paper is wrong.

Objections worth answering up front

"This is just a knowledge graph with extra steps."

The extra steps — deciding what gets in, tidying up, citing sources, tracking dates, learning — are the point, and none of them is link-following. A graph is a good search tool inside this, not a replacement for it.

"The prep work is enormous."

It's done once per document and spread across every future question. On anything asked more than a handful of times, doing the thinking up front beats redoing it on every question — and it's a background job you control, not a delay the user feels. (This holds only while documents change slowly; see §5.)

"How much of the tidy-up is a fixed recipe versus the model improvising?"

Mostly a fixed recipe, and it says which parts aren't. Permission checks, de-duping, picking by date, labelling sources — all fixed and repeatable. Judging whether two things truly clash, and some of the ordering, lean on the model — named as such, and where you spend your testing effort.

"Does it fall over gracefully?"

Yes — see §5. If the prep work, the links, or the dates are missing, each drops back a level; the floor is "plain RAG with citations," never silence.

What this design still can’t do

The most useful part of any design write-up — where no part fully closes the gap, and the interesting work is left.

The model is also the driver

The model sits at the centre of the search loop, deciding what to look up next — but a model is a poor judge of what it doesn't know, and can loop, chase dead ends, or ask for a tool that isn't there. "Stop when you have enough" caps the looping, not the wrong turns, and the learning part tunes what gets in, not what the model chooses to look for. Tellingly, Graphiti keeps the model out of search for exactly this reason. Putting it in the loop is a bet that mid-question flexibility is worth the risk — a bet, not a free lunch.

Two ways in that ”information, not instructions” misses

Marking documents as information stops them acting as commands — but not everything harmful looks like a command. A poisoned document could quietly redefine a key term to bend the model's reasoning, which no such label catches. Worse, poison slipped in *before* the prep work gets baked into clean-looking facts and links — trusted, and harder to spot than a bad raw piece. This design hardens the obvious attack; the subtle ones stay open.

Promises that are easier to state than to keep

Two of the promises take real work. "Check permissions first" is right, but doing it on a big shared store with fine-grained rules is genuinely expensive — you engineer it with pre-filtered views, not naive checks at every step; it isn't free. And the tidy-up's model-assisted parts make it slightly unpredictable exactly where a regulated setting wants a clean audit trail; the honest discipline is to keep those parts on a short leash, log them, and hand an unresolved clash to a human rather than let the model quietly decide it.

”Not in the documents” versus ”I didn’t find it”

The sharpest gap. "We don't have this" and "the search gave up too soon" look identical — an empty set — but call for opposite responses. The best partial fix is to never say "I don't know" just because the search hit its limit without first doing one cheap, wide sweep to check whether the answer is plausibly in there at all. Better than a bare time-out; not perfect.

"There's an answer — but you're not allowed to see it"

A subtle one, easy to miss, and the reason "check access before admission" isn't the whole story. Keeping restricted content out of the model is necessary — but it says nothing about what the *user* is told, and there both honest answers can leak. "There's an answer, you're just not cleared for it" confirms the content *exists* — and for some material (an open investigation, a legal hold, an unannounced deal) the existence *is* the secret. "Not in the documents" protects that, but now misleads: the user may act on a false "nothing here." So there aren't two endings but three — found, genuinely absent, and present-but-restricted — and the third sometimes has to be made to look exactly like the second. One firm rule falls out: the model must never be the thing that decides, because a model that "knows but withholds" leaks under pressure. Keep it on the permission-filtered set as always, and put the reveal-or-disguise choice in a separate, privileged layer *above* it that can see what the model can't — the classic multilevel-security move. Even then two edges stay sharp: the disguise only holds if timing and wording are truly identical (a slower "restricted" path is a tell), and a determined user can still reconstruct a restricted fact from many permitted ones. Which of the two behaviours you pick is a per-class policy, not a default — and it belongs to the deployer, not the design.

Cited but beside the point

The sources rule checks that a citation exists and that it supports the claim — but that support check is itself imperfect. A plausible answer stapled to a real-but-slightly-wrong source is the failure that slips through, and it's the common one.

The bigger frontier

Taken all the way, "build around how the model reads" points past this kind of pipeline — toward baking the documents into the model itself. That's elegant for a fixed set of documents and a real problem for a changing one: you lose easy updates and the clean source trail. Which reframes the whole thing — keeping the model and the documents *separate*, so the documents stay updatable and auditable, isn't a compromise waiting to be dissolved. For anything that changes or has to answer to a regulator, it's the point.

What it deliberately isn't

To keep the scope honest, this design is:

- × not a replacement for ordinary reporting over neat, structured databases;

- × not a model trained end-to-end to do its own searching;
- × not a way to bake a changing set of documents into a model's weights;
- × not a full permissions-and-access product.

RAG today asks for whatever looks like the question. This design asks **what this model actually needs to give an answer it can stand behind** — including admitting when the documents don't cover it. The heavy thinking moves up front, dates and sources live in the store, the model explores through tools, a tidy-up step gets things ready, a rule makes it cite, and a quiet loop lets it improve.

Search gave you documents. RAG gave you pieces. What comes next manages what the model reads.

A note on how this was made: the framing is mine — design retrieval around the way an LLM actually works — and I carried it through the whole process. I drafted the approach, then put it through several AI models in turn, instructing each to attack and extend the last version, doing my own research alongside and judging which criticisms to keep. The models were sharp critics; the direction and the decisions were mine.

BUILDING ON THE WORK OF OTHERS

- Studies showing models read long, crowded contexts less reliably.
- Keyword-plus-meaning search with a re-sort, the common production setup.
- Two-clock knowledge stores (for example Zep / Graphiti) — the closest prior work to parts 0–1.
- The compiled-wiki pattern (Karpathy's LLM Wiki and its offshoots) — the same prep work, but accumulating synthesis rather than re-deriving from sources.
- Open Knowledge Format (Google, 2026) — a portable, git-friendly file convention for the wiki pattern; a serialization for part 1, single-clock and provenance-agnostic.
- Link-based retrieval (the GraphRAG family) as a search tool.
- MemGPT (Packer et al., 2023, now Letta) — the operating-system metaphor for the context window that Figure 3 builds on.
- "Context engineering" (Anthropic and others, 2025) — the now-standard name for curating what earns a place in the window.
- Agentic RAG — the model-in-the-loop retrieval pattern behind the search loop.
- Draft-and-check answering (Speculative RAG, Wang et al., 2024).
- Keeping steady context warm to cut cost and waiting.

A sketch of the lineage, not a citation list — add your own checked sources before publishing.

Glossary

Plain-word glossary

RAG	Retrieval-augmented generation: giving a model your own documents to answer from, by finding relevant bits and pasting them in.
Chunk / piece	A slice of a document, small enough to look up on its own.
Embedding	A string of numbers that captures roughly what a piece of text is about, so similar meanings end up near each other.
Context window	The limited amount of text a model can read at once — its reading space. Everything competes for room in it.

The reading space / working set	The small, final set of notes actually put in front of the model for one answer.
The memory ladder	The idea (borrowed from how computers store data) that memory runs from small/fast/costly at the top to big/slow/cheap at the bottom, and the skill is moving the right things up at the right time.
”Earns a place” (admission)	The idea at the centre of this paper: the job is deciding what deserves room in the model's reading space, not just finding lookalikes.
The structured store	Your documents kept organised — with structure, sources, dates, and links — instead of a flat pile of numbered pieces.
Two clocks (when-true / when-recorded)	Tracking both when a fact was true in the world and when the system recorded or removed it — so a 2024 rule is right for 2024 and wrong for 2026.
Keyword vs meaning search	Keyword search matches exact words (great for codes and reference numbers); meaning search matches ideas (great for fuzzy questions). You want both.
Re-sort (re-rank)	A quick second pass that re-orders search results by how relevant they really are.
The tidy-up step (compiler)	The stage that turns raw search results into a small, ordered, labelled, de-duplicated set the model can read cleanly.
Source (provenance)	Where a fact came from — which document, which version — so an answer can be traced and checked.
Hidden-instruction attack	Text planted in a document that tries to act as a command to the model. Blocked by treating all document text as information, never instructions.
Does the source back the claim (support check)	Checking that a cited source actually supports what was said — not just that a citation is present.

The model's own knowledge (parametric)	What the model knows from training, as opposed to what it was handed from your documents. A lower bar of trust, so it's flagged.
Take-one-out testing	Removing one note and checking whether the answer gets worse — a way to measure whether that note actually helped.
The sorter (router)	A cheap first check that sends easy questions down a fast lane and hard ones down the careful lane.
Keeping context warm (caching)	Reusing the model's reading of an unchanged part so a follow-up question only pays for what changed — cheaper and faster.
Two-clock knowledge store (Graphiti / Zep)	An existing, widely used store that already tracks when-true and when-recorded on every fact — the closest prior work to parts 0–1 here.
Data poisoning	Corrupting source documents so a system learns or retrieves attacker-chosen falsehoods — especially dangerous if it slips in before the prep work and gets baked into clean-looking facts.
Permissions (access control)	The rules for who is allowed to see what. Safest checked *before* anything is let in, not filtered out afterward.

Working Paper v6 · August 7, 2026 · Shareable draft — comments welcome.