Author - Vikas Pandey

Published at - 09-Feb-2026

website - https://vikaspandey.dev

Agentic RAG — Architecture Design

> **Audience:** Basic to intermediate engineers who understand REST APIs, databases, and have written some Python or JavaScript. **Goal:** By the end of this document you will understand not just *what* Agentic RAG is, but *why* every design decision exists and what goes wrong when you skip it.

---

1. What Problem Are We Actually Solving?

Before any architecture, understand the problem deeply.

A user uploads 200 PDFs — contracts, policies, runbooks. They ask:

> *"What is the penalty clause if a vendor misses the SLA defined in the Mumbai office contract?"*

A naive approach sends all 200 PDFs to GPT-4. That fails for three reasons:

1. **Context window limits** — 200 PDFs = ~40 million tokens. No model accepts that.
2. **Cost** — Even if it fit, you'd pay $400 per question.
3. **Quality** — Models hallucinate when drowning in irrelevant text. Less context = sharper answers.

RAG (Retrieval-Augmented Generation) solves this by retrieving only the 2–3 most relevant passages before calling the LLM. **Agentic RAG** goes further — instead of a fixed retrieve-then-answer pipeline, an agent *reasons* about which tools to use, in what order, and whether the answer it found is good enough.

---

2. The Spectrum: Naive → RAG → Agentic RAG

Understanding the evolution prevents you from over-engineering or under-engineering.

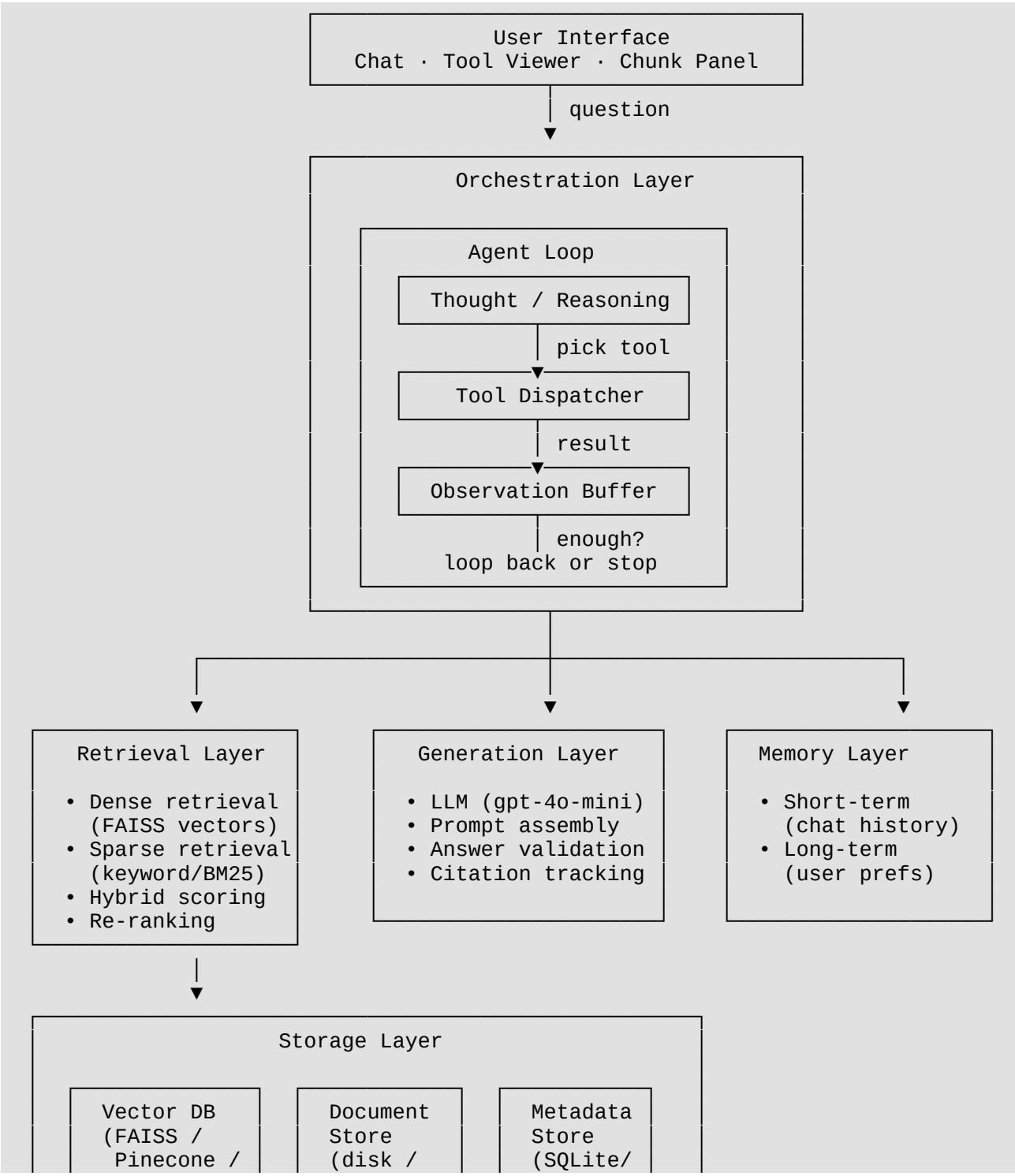
```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│  Naive LLM          Basic RAG           Agentic RAG          │
│  ─────────          ─────────           ─────────────        │
│  Dump all docs  →   Retrieve then   →   Agent decides:       │
│  into prompt        answer once           • Which tool to use│
│                                           • Whether to re-search
│  Problems:          Problems:             • Whether answer is good
│  • Token limits     • One retrieval       • Whether to ask clarify
│  • High cost        • No verification     • Whether to combine docs
```

```
  • Hallucination    • No reasoning
                                          Result: higher accuracy,
                                          handles complex queries
```

Rule of thumb: Start with Basic RAG. Move to Agentic RAG when you observe the agent needing to combine information across multiple sources, verify its own answers, or decompose multi-part questions.

---

3. High-Level Agentic RAG Architecture

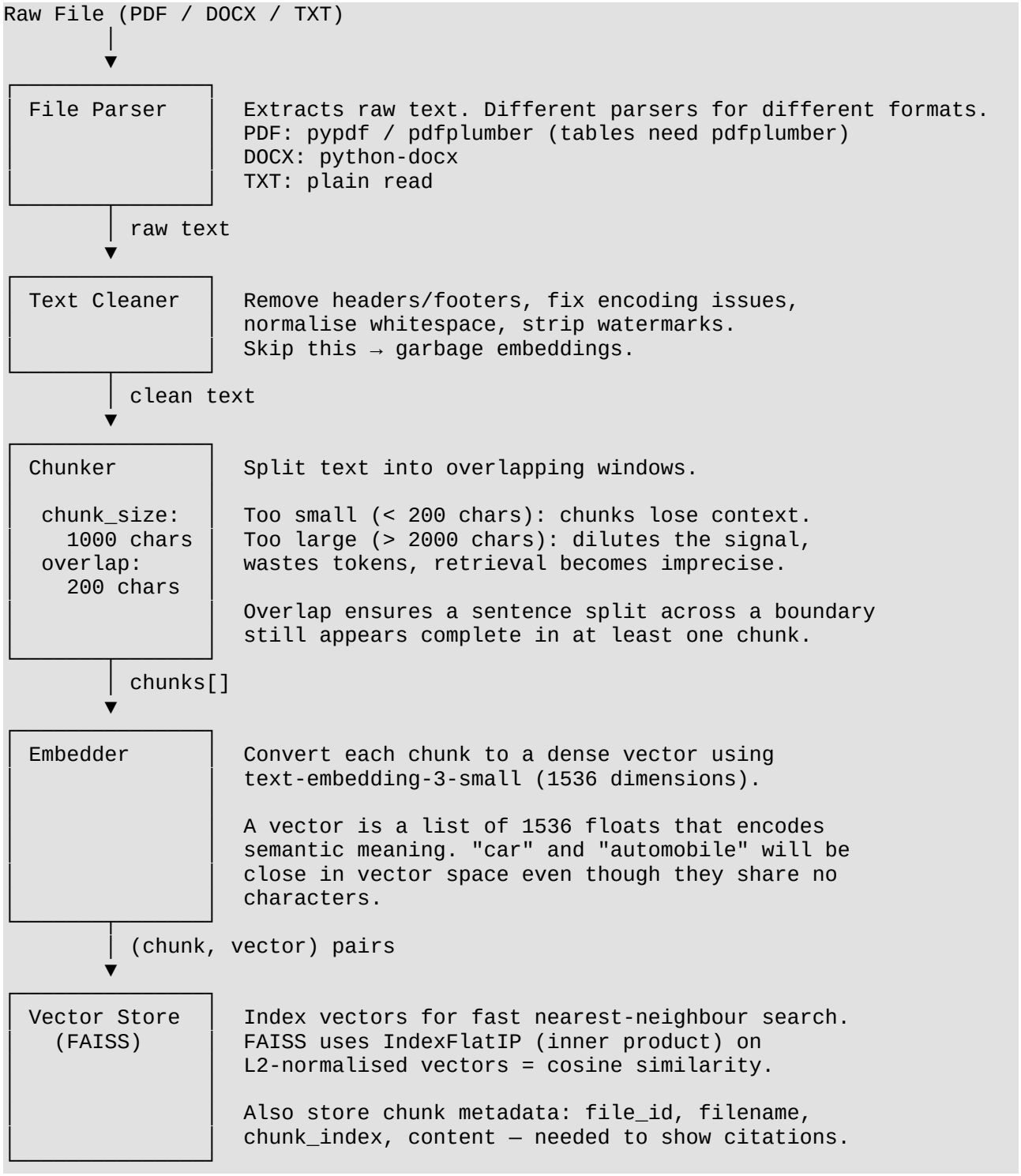


---

4. Component Deep-Dive

4.1 Document Ingestion Pipeline

This runs *before* any user question. It is the foundation. Get this wrong and the whole system fails regardless of how clever your agent is.



Why overlap matters (diagram):

```
Original text:
"...The penalty clause states that vendors must respond within 4 hours.
Failure to respond results in a $500 fine per incident..."

Chunk A (chars 0-1000):    "...vendors must respond within 4 hours."
Chunk B (chars 800-1800):  "within 4 hours. Failure to respond results in a $500
fine..."

Without overlap, "respond within 4 hours" and "$500 fine" live in
separate chunks with no connecting context. The agent retrieves only
one and gives an incomplete answer.
```

4.2 The Agent Loop (Core of Agentic RAG)

This is what separates Agentic RAG from Basic RAG. Instead of one fixed retrieval → answer pass, the agent runs a loop:

```
          ┌──────────────────────────────────┐
          │                                  │
          │  while not satisfied:            │
          │                                  │
          │    thought = llm.think(          │
          │      question,                   │
          │      tools_available,            │
          │      observations_so_far         │
          │    )                             │
          │                                  │
          │    if thought.action == "done":  │
          │      break                       │
          │                                  │
          │    result = tools[               │
          │      thought.tool                │
          │    ].run(thought.input)          │
          │                                  │
          │    observations.append(result)   │
          │                                  │
          └──────────────────────────────────┘
          │
          ▼
    final_answer = llm.generate(
        question,
        all_observations
    )
```

The ReAct pattern (Reason + Act) is the standard implementation:

```
Thought:  I need to find the penalty clause for SLA breach in the
          Mumbai office contract.

Action:   search_chunks
Input:    "penalty clause SLA breach Mumbai"

Observation: [Chunk from contract_mumbai.pdf, score 0.91]
             "Section 8.2: Vendor SLA penalties — Response within 4
              hours required. Breach results in $500 per incident..."

Thought:  I found the penalty amount but need to verify if there is
          a cap on total penalties.
```

```
Action:   search_chunks
Input:    "penalty cap maximum liability Mumbai contract"

Observation: [Chunk from contract_mumbai.pdf, score 0.87]
             "Section 8.4: Total liability shall not exceed 10% of
              annual contract value..."

Thought:  I now have both the per-incident penalty and the cap.
          I can answer.

Action:   done
```

Why this is powerful: A basic RAG would have retrieved chunks for the first search and answered without knowing about the liability cap. The agent self-corrects.

4.3 Tool Registry

Tools are functions the agent can call. Each tool has a name, description, input schema, and implementation. The LLM sees the descriptions and decides which tool to use.

```
Tool Registry
│
├── search_chunks(query: str, k: int = 5) → list[Chunk]
│     Semantic + keyword hybrid search across all embedded chunks.
│     Use when: user asks a factual question about documents.
│
├── extract_keywords(text: str) → list[str]
│     LLM-powered extraction of search terms from a query.
│     Use when: query is conversational and needs reformulation.
│
├── summarise_file(file_id: str) → str
│     Summarise an entire document in < 500 words.
│     Use when: user asks "what is this document about?"
│
├── compare_chunks(chunk_a: str, chunk_b: str) → str
│     Ask LLM to compare two retrieved passages.
│     Use when: user asks to compare two documents or clauses.
│
└── generate_answer(context: str, question: str) → str
      Final answer generation with citations.
      Use when: enough context has been gathered.
```

Design principle: Keep tools narrow and single-purpose. A tool that does too much is hard to test, hard to debug, and the agent will misuse it. "search_and_summarise_and_answer" is a red flag.

4.4 Retrieval Layer

This is the most technically nuanced component. The quality of retrieval determines 80% of answer quality. No amount of clever prompting saves a bad retrieval.

Dense Retrieval (Vector Search)

```
Query: "What is the penalty for SLA breach?"
         │
         ▼
```

```
   Embed query → [0.02, -0.14, 0.87, ...] (1536 floats)
         │
         ▼
   FAISS: find top-K vectors with highest cosine similarity
         │
         ▼
   Returns chunks whose *meaning* is closest to the query
   even if they share no exact words.
```

Strength: Handles paraphrasing, synonyms, semantic variations. **Weakness:** Poor at exact matches — "Section 8.2", "clause 4(b)", specific numbers.

Sparse Retrieval (Keyword / BM25)

```
Query keywords: ["penalty", "SLA", "breach"]
         │
         ▼
   Score each chunk by keyword frequency + rarity weight (BM25)
         │
         ▼
   Returns chunks containing the exact terms.
```

Strength: Exact term matching — great for codes, IDs, proper nouns. **Weakness:** No semantic understanding. "car" ≠ "automobile".

Hybrid Scoring (What This App Uses)

```
final_score = (0.5 × vector_similarity) + (0.5 × keyword_match_ratio)
```

This is the simplest hybrid approach. Production systems use learned weights or Reciprocal Rank Fusion (RRF) to combine ranked lists without needing to tune weights.

Re-ranking (Next Level)

After retrieval, a second model re-scores the top-20 candidates more carefully:

```
FAISS retrieves top-20 (fast, approximate)
         │
         ▼
Cross-encoder re-ranker scores all 20 against the query
(e.g. cross-encoder/ms-marco-MiniLM-L-6-v2)
         │
         ▼
Take top-3 by re-ranker score (slow but accurate)
```

Cross-encoders read query + passage *together* — they understand relevance in context, not just similarity in isolation. They are 10–20× slower than FAISS but 15–30% more accurate in precision@3.

4.5 Generation Layer

The final LLM call. Most engineers treat this as the *whole* system — it is actually the last 20%.

```
System Prompt
─────────────

You are a document assistant. Answer using only the supplied context.
If the answer is not present, say so explicitly. Never invent facts.
```

```
Always cite the source document and section when possible.

Context (assembled from top-3 chunks)
──────────────────────────────────────

[contract_mumbai.pdf · chunk 12]
Section 8.2: Vendor SLA penalties — Response within 4 hours...

[contract_mumbai.pdf · chunk 14]
Section 8.4: Total liability shall not exceed 10% of annual...

Question
────────

What is the penalty if a vendor misses the SLA?

Answer
──────

According to Section 8.2 of contract_mumbai.pdf, vendors must respond
within 4 hours. A breach results in a $500 fine per incident.
Section 8.4 caps total liability at 10% of the annual contract value.
```

Critical prompt engineering decisions:

Decision	Wrong	Right
Instruction to not hallucinate	"Try to answer accurately"	"If answer is not in context, say: 'I could not find that information'"
Context format	Dump raw text	Label each chunk with `[filename · chunk N]` for citations
Temperature	0.7 (creative)	0.0–0.2 (factual, deterministic)
Model choice	GPT-4 for everything	GPT-4o-mini for extraction/ranking, GPT-4o only for final answer

---

4.6 Memory Layer

Basic RAG has no memory — every question is answered in isolation. This breaks for follow-up questions:

```
User:  "What is the penalty for SLA breach?"
Agent: "$500 per incident, capped at 10% annual contract value."

User:  "Is that the same for the Delhi contract?"
Agent: "I could not find that information."   ← WRONG: forgot context
```

Short-term Memory (Conversation History)

Pass the last N turns as context:

```
messages = [
    {"role": "system", "content": SYSTEM_PROMPT},
    {"role": "user",   "content": "What is the penalty for SLA breach?"},
    {"role": "assistant", "content": "$500 per incident..."},
    {"role": "user",   "content": "Is that the same for the Delhi contract?"},
```

```
]
```

Gotcha: Don't pass entire conversation history — at turn 50 you'll hit context limits. Use a sliding window of last 6–10 turns, or summarise old turns.

Long-term Memory (User/Session State)

Store preferences, documents the user has focused on, previous questions. Useful for personalisation but adds complexity. Skip until basic memory works well.

4.7 Storage Layer

Three distinct storage concerns that engineers routinely conflate:

```
┌──────────────────────────────────────────────────────────────┐
│                                                              │
│  Vector Store          Document Store       Metadata Store   │
│  ────────────          ──────────────       ──────────────   │
│                                                              │
│  Stores:               Stores:              Stores:         │
│  Chunk embeddings      Raw files            File records    │
│  (float arrays)        (binary)             Chunk counts    │
│                                             Upload timestamps│
│  Queried by:           Queried by:          Chunk status    │
│  Similarity search     File ID / name                       │
│                                             Queried by:     │
│  Options:              Options:             File ID / user ID│
│  FAISS (in-memory)     Local disk                           │
│  Pinecone (managed)    S3 / GCS             Options:        │
│  Weaviate (self-host)  MinIO (self-host)    SQLite (simple) │
│  Qdrant (self-host)                         PostgreSQL (prod)│
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Current implementation uses:

- FAISS in-memory (lost on restart — acceptable for development)
- Local disk under `/context` (persisted via Docker volume)
- In-memory Python dicts for metadata (lost on restart)

Production should use:

- FAISS serialised to disk, or Pinecone/Qdrant for managed vector search
- S3 or mounted volume for documents
- SQLite minimum, PostgreSQL for multi-user

4.8 Observability Layer

You cannot improve what you cannot measure. This is the component most junior engineers skip and most senior engineers insist on from day one.

```
Every agent step emits a span:
```

```
┌──────────────────────────────────────────────────────┐
│  Span: extract_keywords                              │
│  ──────────────────────                              │
│  trace_id:    abc-123                                │
│  start_time:  2024-01-15T10:30:00.000Z               │
│  end_time:    2024-01-15T10:30:00.342Z               │
│  duration_ms: 342                                    │
│  input:       "What is the penalty for SLA breach?"  │
│  output:      ["penalty", "SLA", "breach"]           │
│  model:       gpt-4o-mini                            │
│  tokens_in:   45                                     │
│  tokens_out:  12                                     │
│  cost_usd:    0.000057                               │
└──────────────────────────────────────────────────────┘
```

What to track per request:

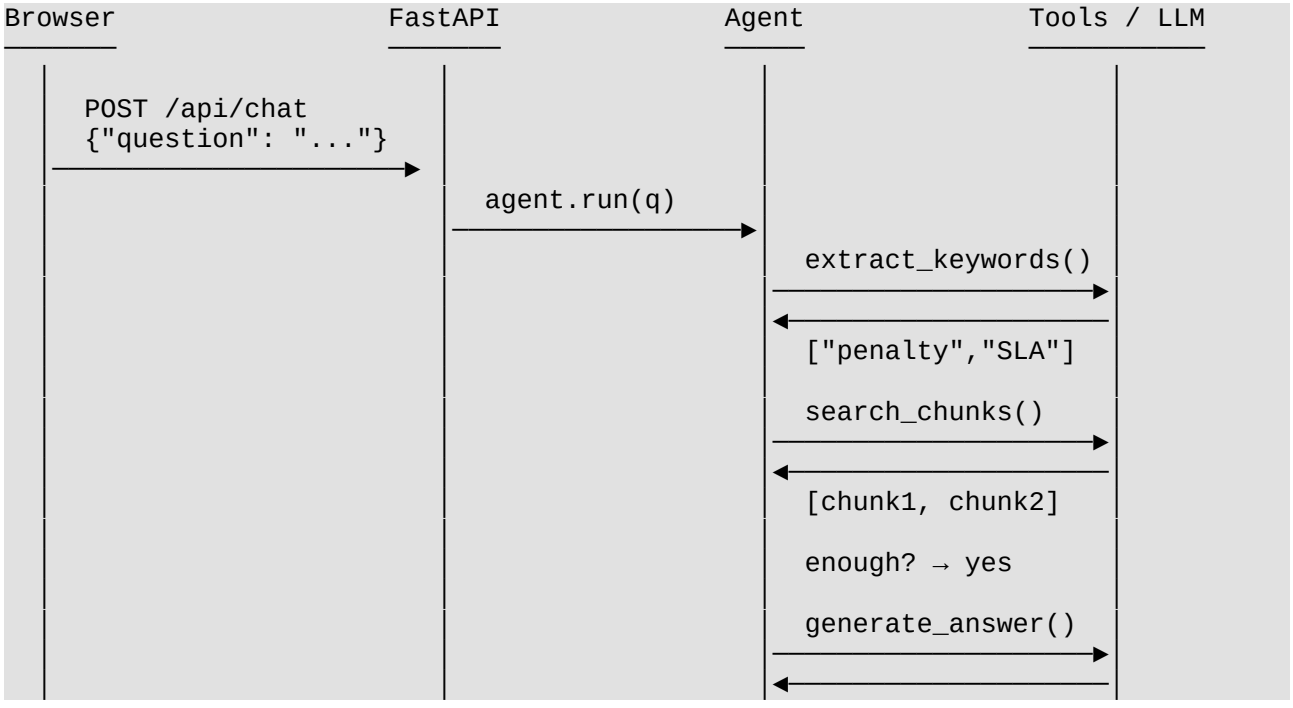
- Total latency (user experience)
- Per-step latency (where is time spent?)
- Token usage per call (cost tracking)
- Retrieval scores (is retrieval quality degrading?)
- Whether agent looped (how many iterations?)
- Final answer — did user follow up with a correction? (implicit feedback)

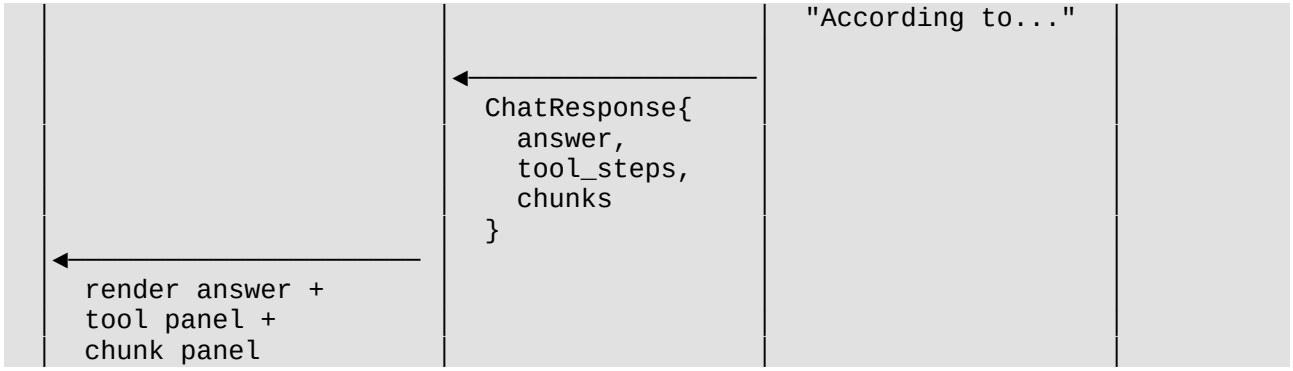
Tools: OpenTelemetry → Jaeger/Tempo for traces. Prometheus for metrics. LangSmith/Langfuse specifically for LLM pipelines.

---

5. The Full Agentic RAG Request Lifecycle

A single user question, end-to-end:





---

6. Where This Implementation Sits on the Maturity Scale

```
Level 1: Basic RAG          ← Most tutorials stop here
  Fixed: retrieve → answer
  No loops, no tool choice, no memory

Level 2: Pipeline RAG       ← This implementation
  Fixed steps but transparent:
  keywords → search → rank → context → answer
  Hybrid scoring, tool step visibility

Level 3: Agentic RAG        ← Next evolution
  Agent decides steps dynamically
  Can loop, verify, decompose, combine sources
  Has short-term conversation memory

Level 4: Multi-Agent RAG    ← Production at scale
  Specialised agents (retrieval agent, summariser,
  fact-checker, citation verifier) coordinated
  by an orchestrator agent
  Async, parallel tool calls
  Full observability
```

This codebase is at **Level 2**, architected to evolve to Level 3 without a rewrite — the service layer, tool step recording, and response schema are already agent-compatible.

---

7. Common Failure Modes and How to Prevent Them

Failure	**Symptom**	**Root Cause**	**Fix**
Wrong chunks retrieved	Irrelevant answer	Chunk too large, poor embedding model	Reduce chunk size, add re-ranker
Hallucination	Answer not in documents	System prompt too weak	Explicit "say I don't know" instruction, lower temperature
Agent infinite loop	Request never returns	No loop limit, LLM keeps searching	Hard cap on iterations (max 5)
Stale index	Answers from old file version	Re-chunking doesn't clear old vectors	`remove_file()` before re-adding

Failure	Symptom	Root Cause	Fix
Slow response	User sees loading for 10s+	Embedding + multiple LLM calls	Streaming, async parallel tool calls, cache embeddings
Context overflow	API errors on large docs	Too many chunks in prompt	Strict top-3 limit, summarise chunks before combining
Lost data on restart	Empty file list after deploy	In-memory metadata store	SQLite + serialised FAISS index

---

8. Key Design Principles to Remember

1. Retrieval quality > generation quality. A powerful LLM cannot compensate for irrelevant retrieved chunks. Invest in hybrid retrieval, clean chunking, and re-ranking before investing in a better LLM.

2. Narrow tools, broad agent. Each tool should do exactly one thing. The agent's intelligence comes from combining simple tools, not from complex tools.

3. Always show your work. Tool step transparency is not a nice-to-have. Users who can see *why* an answer was produced trust it more and catch errors faster.

4. Fail loudly, not silently. An agent that says "I could not find that information" is more valuable than one that makes up an answer. Hallucination erodes trust permanently.

5. Design for restart. In-memory state is a prototype convenience. Every production system must survive a server restart: serialise the vector index, persist metadata to a database.

6. Measure cost from day one. LLM calls cost money. Log token counts on every call. Set budget alerts. A naive agent in a loop can burn $50 in a single misfired request.

---

9. Recommended Reading

Topic	Resource
RAG fundamentals	*Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* — Lewis et al. 2020
ReAct agent pattern	*ReAct: Synergizing Reasoning and Acting in Language Models* — Yao et al. 2022
Hybrid retrieval	*SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking*
Re-ranking	*MS MARCO passage ranking* dataset + cross-encoder benchmarks
Production RAG	*Building RAG-based LLM Applications for*

Topic	Resource
	Production — Anyscale blog
Evaluation	*RAGAS: Automated Evaluation of Retrieval Augmented Generation*