

# Building an **Agentic RAG** in the Rails Way with RubyLLM

# Giovanni Panasiti

Rome, Italy

<https://panasiti.me>

[giova.panasiti@gmail.com](mailto:giova.panasiti@gmail.com)

@giovapanasiti



**“We need AI,  
because... it’s cool”**

*Every manager since 2024*

# Outsourcing Proposal

4

languages

2

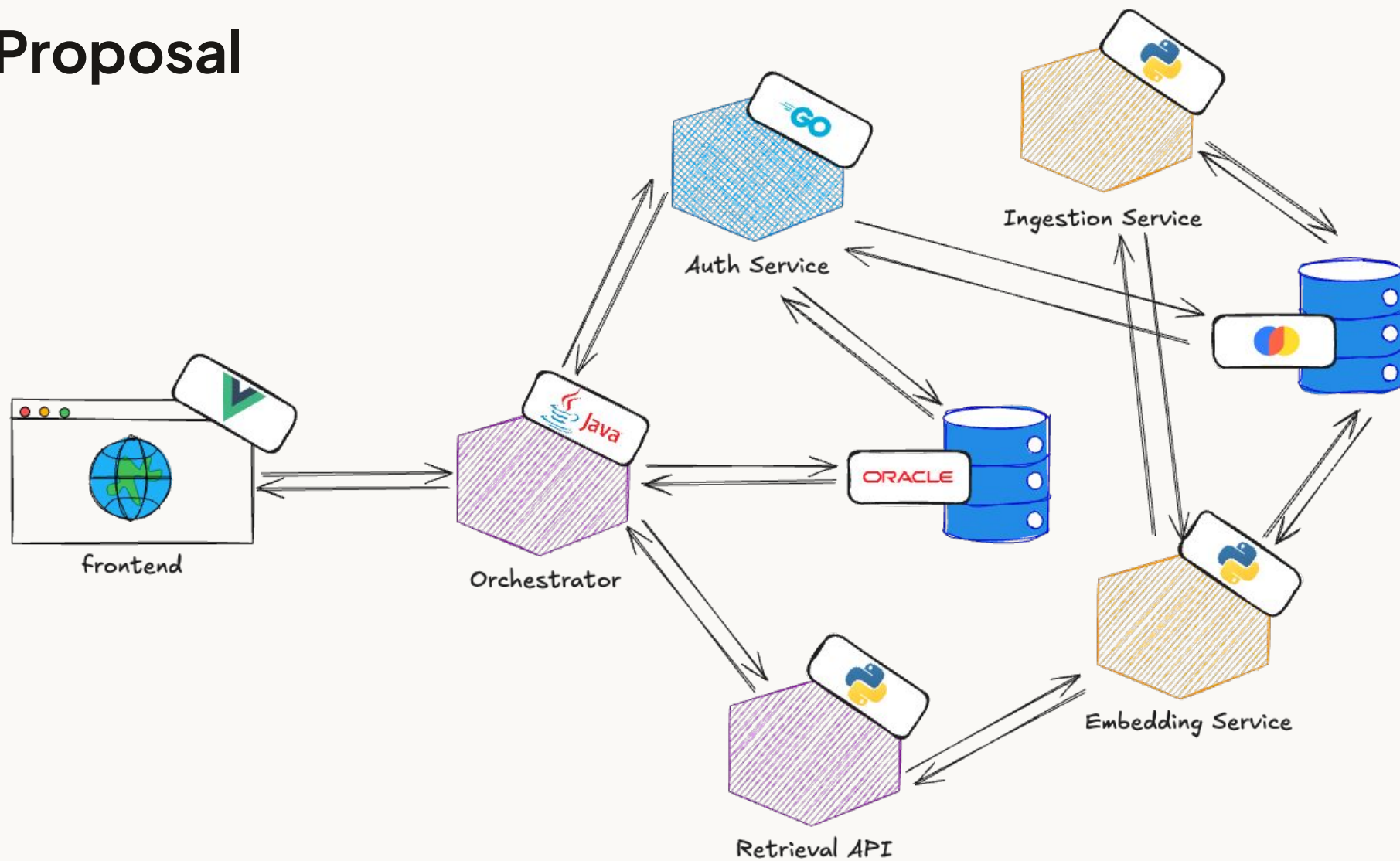
databases

6

services

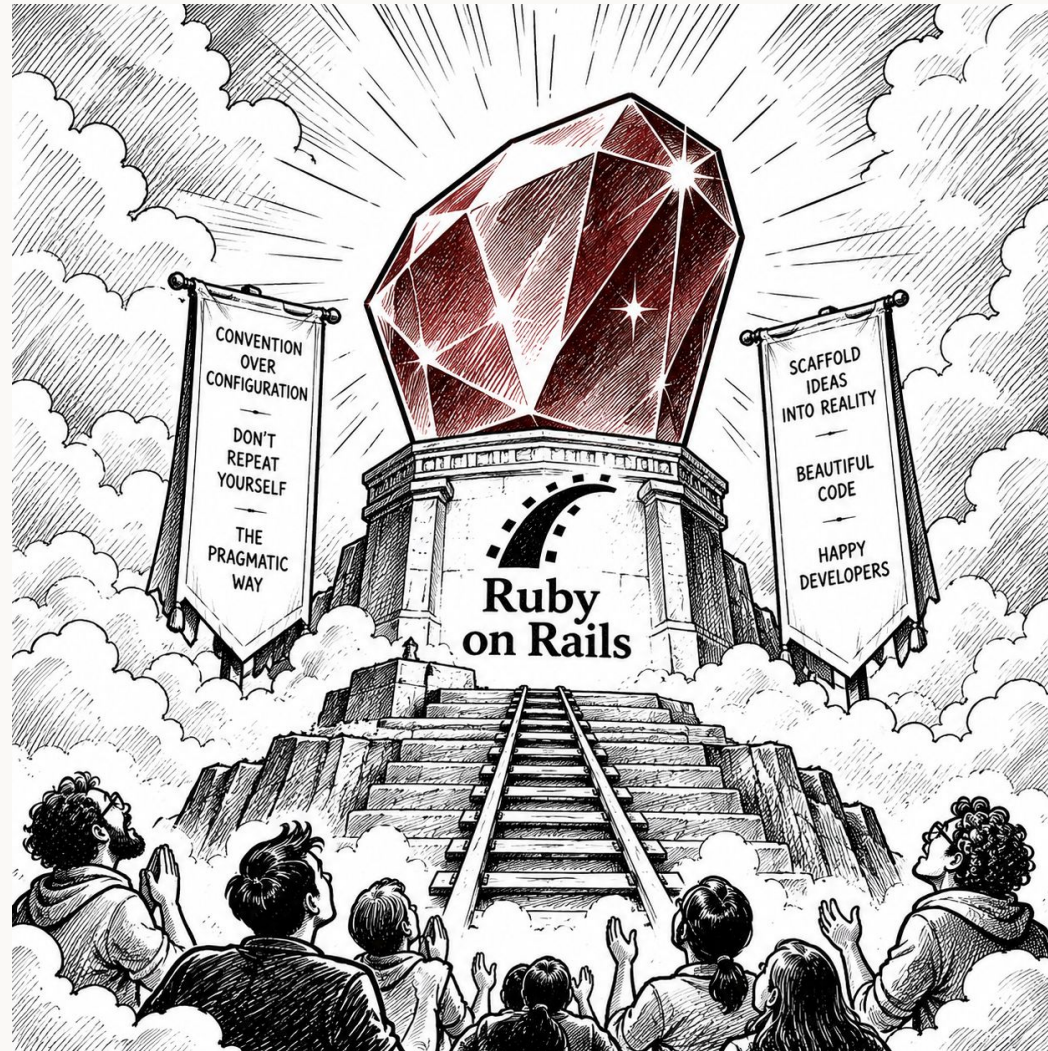
1

answer



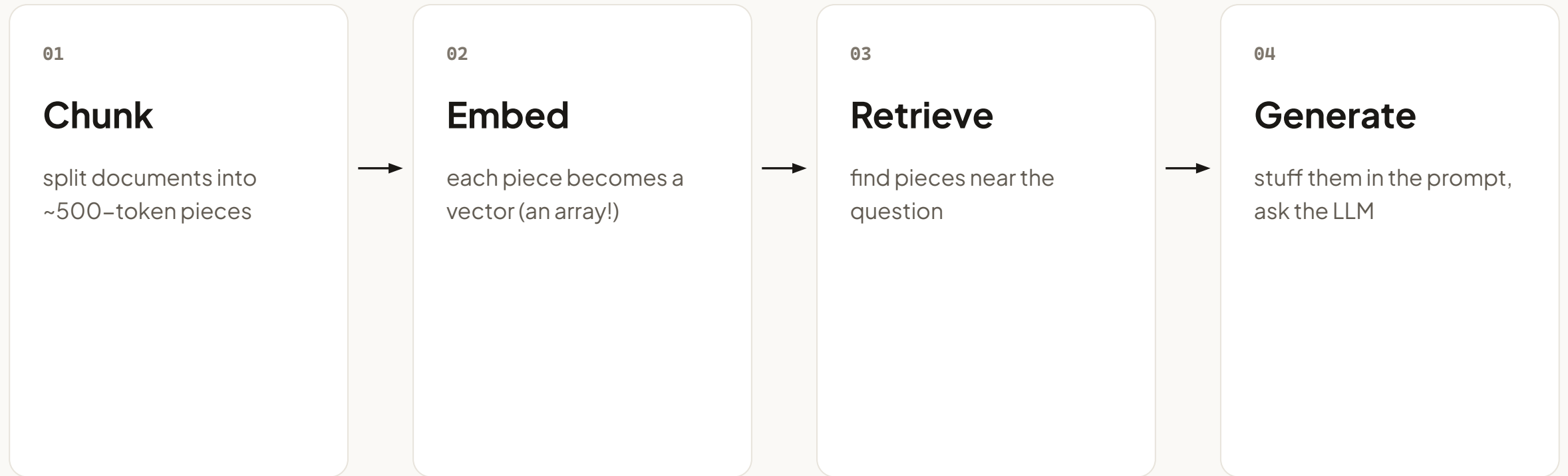
**All those moving pieces, to answer a  
question about our own documents...  
something looked off.**

# The bet: same features, one monolith



**Let's build a RAG!**  
**what is a RAG?** 🤔

# RAG is four verbs, nothing else



**Retrieval-Augmented Generation = search, then prompt.**

# The whole AI stack fits in a Gemfile

```
1 # Gemfile – the entire "AI infrastructure"
2 gem "ruby_llm"      # chat · embeddings · tools · agents
3 # structured output: Schematist ships inside ruby_llm
4 gem "neighbor"      # pgvector, ActiveRecord-style
5 gem "pg_search"     # Postgres full-text search
6 gem "pdf-reader"    # ingestion
7 gem "tiktoken_ruby" # token budgets
8 gem "numo-narray"   # fast math for reranking
9
10 # ...and the boring rest you already run
11 gem "solid_queue"   # async pipelines (in Postgres)
12 gem "solid_cable"   # websockets      (in Postgres)
13 gem "turbo-rails"   # streaming UI
14
```

0

new languages

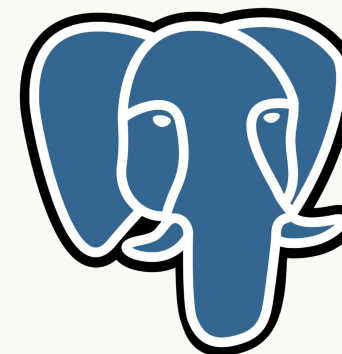
0

new services

0

new databases

# Two databases became one migration



## HNSW

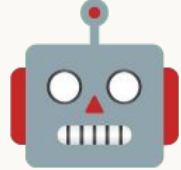
fast approximate nearest-neighbor search over vectors

---

## GIN

inverted index for full-text tsvector queries

```
1 # db/schema.rb – one Postgres, wearing both hats
2 enable_extension "vector"
3
4 create_table "chunks" do |t|
5   t.references :document, :parent_chunk
6   t.text       "content"
7   t.integer    "position", "page_number", "tokens"
8
9   t.vector     "embedding", limit: 1536           # the "vector DB"
10  t.virtual    "searchable", type: :tsvector,      # the search engine
11    as: "to_tsvector('italian', content)", stored: true
12
13  t.index ["embedding"], using: :hnsu, opclass: :vector_cosine_ops
14  t.index ["searchable"], using: :gin
15 end
16
```

**and we are ready to**  
**RAG** 

# 1. Processing and embedding documents became one job

```
1 class DocumentProcessingJob < ApplicationJob # condensed from the real 729-line job
2   queue_as :ingestion                        # Solid Queue – supervised, in Postgres
3
4   retry_on RubyLLM::RateLimitError, RubyLLM::OverloadedError,
5           wait: :polynomially_longer, attempts: 5
6
7   def perform(document_id)
8     document = Document.find(document_id)
9     text     = extract_text(document)        # pdftotext -layout, pdf-reader fallback
10    chunks   = create_chunks(document, text) # structure-aware splitting, ~500 tokens
11    embed_in_batches(chunks)                 # RubyLLM.embed, 100 contents per call
12    document.completed!
13  end
14 end
```

The part that most improves answer quality is not AI: it's chunking that respects headers, article numbers and tables. **String handling.**  
Do you think you need Python or some fancy language to handle strings?

## 2. The retrieval API became a class method

```
1 class Chunk < ApplicationRecord # condensed from app/models/chunk.rb
2   has_neighbors :embedding # neighbor gem
3   pg_search_scope :text_search, against: :content, using: { tsearch: { dictionary:
  "italian" } }
4
5   def self.hybrid_search(query, user:, limit: 10)
6     embedding = cached_query_embedding(query) # RubyLLM.embed + Rails.cache
7     # HNSW
8     vector = nearest_neighbors(:embedding, embedding, distance: :cosine).limit(limit * 3)
9     fulltext = text_search(expand_synonyms(query)).limit(limit * 3)
10    merged = rrf_merge(vector, fulltext) # ↓ the whole "fusion algorithm"
11    mmr_rerank(merged, embedding, limit:) # diversity, via Numo::NArray
12  end
13
14  def self.rrf_merge(vec, txt) # reciprocal rank fusion, complete
15    scores = Hash.new(0.0)
16    [vec, txt].each do |list|
17      list.each_with_index { |c, rank| scores[c.id] += 1.0 / (60 + rank + 1) }
18    end
19    (vec + txt).uniq(&:id).sort_by { |c| -scores[c.id] }
20  end
21 end
22
```

### 3. The answer: retrieve, do stuff

```
1 # ChatResponseJob – classic single-shot mode (condensed)
2 query = rewrite_query(content, history) # "and in 2024?" → standalone question
3 chunks = Chunk.hybrid_search(query, user: user, limit: 60)
4 context = build_context_with_budget(chunks) # best 15, ≤ 10k tokens
5
6 chat.with_instructions(PromptBuilder.build(context))
7 chat.complete { |chunk| message.broadcast_streaming_content(chunk) }
```

# Streaming is a Turbo broadcast, like everything else

```
1  class Message < ApplicationRecord
2    acts_as_message                # ruby_llm Rails integration
3
4    def broadcast_streaming_content(full_content)
5      broadcast_update_to "chat_#{chat_id}",
6        target: "message_#{id}_content",
7        partial: "messages/content",
8        locals: { content: full_content }
9    end
10 end
```



# 5 weeks in we had the RAG

## The original proposal

6 services · 4 languages · 2 databases

8 months from kickoff to production

## The Rails build

1 app · 1 language · 1 Postgres

5 weeks

1 developer, alongside the day job

every change: a commit

# The questions that kept failing share a shape

## **“Compare the 2019 and the 2024 revision”**

needs several targeted searches but the pipeline gets exactly one

## **“As per Art. 12 of the annex...”**

the answer lives one cross-reference away; the pipeline can't follow it

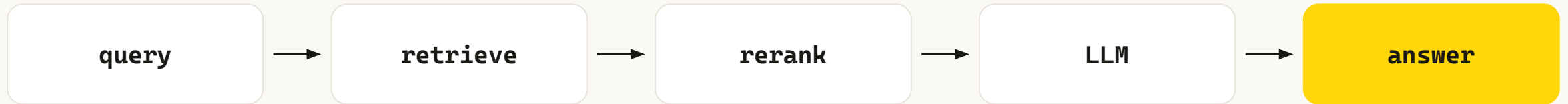
## **“What documents do you even have?”**

a catalog question not answerable by similarity search at all

## **A retrieval miss**

weak context in, confident improvisation out and it can't notice, retry, or refuse

# What I had shipped was a pipeline



```
# the pipeline, honestly:
chunks = search(query)      # one query, one shot
answer = llm(chunks)        # no feedback, no retry

# the ONE decision that matters: what to read
# is taken blind, before the model can react.
```

## An open loop.

Control theory has a name for systems that act once and never observe the result.

**I had built the system.  
Now I wanted to make it  
better...**

**Let's build it  
with agents.**

# What is an agent? (the de-hyped version)

An agent is an LLM that runs in a loop, calling your code, **until it decides it's done.**

## An LLM model

any chat LLM that supports tool calls

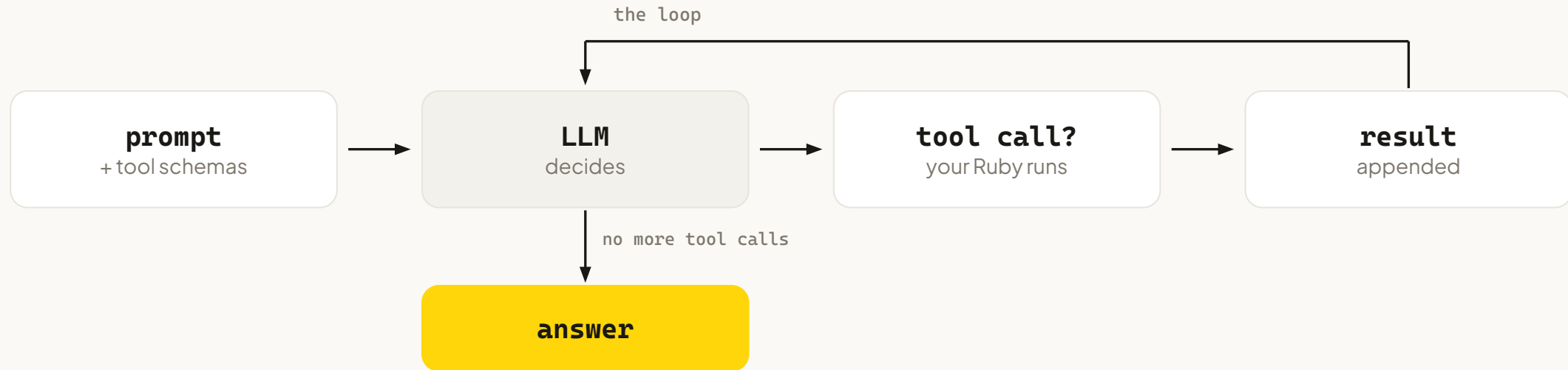
## Tools

plain Ruby methods, described to the model

## A loop

call → observe result → decide → repeat

# The loop and who runs it



```
chat.with_tools(SearchKnowledgeBase.new(user:, collector:),  
                FetchDocumentSection.new(user:, collector:))  
  .ask("What changed between the 2019 and 2024 revisions?")
```

# A tool is just a Ruby class

```
1 class SearchKnowledgeBase < RubyLLM::Tool      # app/services/rag/tools/, trimmed
2   MAX_TOOL_CALLS = 6
3
4   description <<~DESC
5     Search the company knowledge base (regulations, policies, manuals).
6     Use this BEFORE answering. For compound questions, call it several times.
7   DESC
8   parameter :query, description: "A specific, targeted search query", required: true
9
10  def execute(query:)
11    return limit_message if @collector.limit_reached?(MAX_TOOL_CALLS)
12    @collector.increment_tool_call
13    chunks = Chunk.hybrid_search(query, user: @user, limit: 15) # same method!
14    @collector.record(chunks)                                     # citations + step trail
15    return "No results for: #{query}" if chunks.empty?
16    format_sources(chunks)                                       # "[S1] Title (p.3) [doc:42] ..."
17  rescue StandardError => e
18    "Search failed for '#{query}'. Try again with different terms."
19  end
20 end
21
```

# My first agent

```
1 class Researcher < RubyLLM::Agent
2   model      "gpt-5.5"
3   temperature 0.2          # loops should be boring
4
5   instructions <<~PROMPT
6     Search before answering – never answer from memory.
7     Decompose compound questions into separate,
8     targeted searches. Follow cross-references.
9     Cite every claim. If the documents are not
10    enough, say so – do not invent.
11  PROMPT
12
13  tools SearchKnowledgeBase, FetchDocumentSection,
14        ListDocuments, Note
15 end
16
17 Researcher.new.ask "How do the 2019 and 2024 revisions differ?"
```

## First run, live

- 🔍 2019 revision – approval limits
- 🔍 2024 revision – approval limits
- 💬 2024 text points to Annex B – reading it
- 📄 Reading: 2024 revision, Annex B
- 🔍 transition rules 2019 → 2024
- ✅ **side-by-side answer, 5 sources cited**

*The compare-question, the one the pipeline structurally could not answer, worked on day one.*

# Then I got greedy: one agent, every tool

The do-everything agent



The specialist team



# So I hired specialists (they're objects, they work for free)

*Because we have infinite tokens!!*

## Researcher

deep document search:  
decomposes questions,  
follows references, cites  
everything



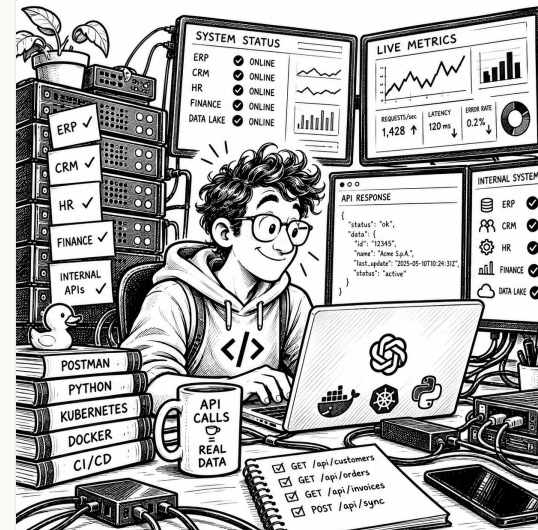
## Law expert

interprets regulations:  
quotes the exact article,  
follows every  
cross-reference



## Operations

the one with API access: live  
status from internal  
systems, not documents



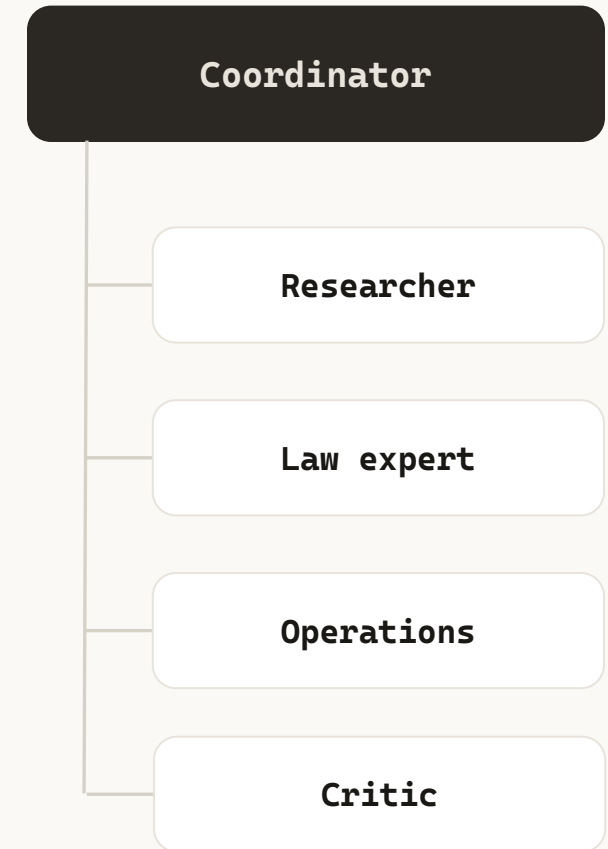
## Critic

reviews a draft answer  
against its sources before it  
ships



# The Coordinator: an agent whose tools are agents

```
1 class AskLawExpert < RubyLLM::Tool
2   description "Hand off regulatory questions to the Law expert. " \
3     "Send one complete, self-contained request."
4   parameter :request, description: "The question for the specialist"
5
6   def execute(request:)
7     LawExpert.new.ask(request).content # an agent is just Ruby -
8   end                                 # so an agent can be a tool
9 end
10
11 class Coordinator < RubyLLM::Agent
12   instructions <<~PROMPT
13     Route each question to the right specialist. Prefer ONE handoff;
14     combine answers only when the question truly spans domains.
15   PROMPT
16   tools AskResearcher, AskLawExpert, AskOperations
17 end
```



# Hotwire makes the team visible (and trustworthy)

```
1 # in the runner:
2 chat.before_tool_call { |tc| message.broadcast_agent_step(step_for(tc)) }
3
4 # in the model:
5 def broadcast_agent_step(step)
6   broadcast_append_to "chat_#{chat_id}",
7     target: "message_#{id}_steps",
8     partial: "messages/agent_steps",
9     locals: { step: step }
10 end
11
```

## what users watch, live

-  Routing to: Law expert
-  Searching: 2024 annex, art. 12
-  Art. 12 refers to Annex B – reading it
-  Reading: Annex B, section 3
-  **Answer with 4 cited sources**

# That's how **we built** our agentic RAG



# Just another tool in the box

|                | Full context                                 | Classic pipeline          | Agent team                                    |
|----------------|----------------------------------------------|---------------------------|-----------------------------------------------|
| WHEN           | the whole document fits ( $\leq 15k$ tokens) | a simple factual question | <b>multi-hop, cross-references, live data</b> |
| LLM CALLS      | 1                                            | 1 (+ rerank)              | several, variable                             |
| COST / LATENCY | lowest                                       | low, predictable          | several times, visible steps                  |
| ON A MISS      | nothing to miss                              | improvises                | <b>searches again or says no</b>              |

**Rails developers are  
not second-class citizens  
in the AI conversation.**

# Grazie.

`https://panasiti.me`

`giova.panasiti@gmail.com`

`@giovapanasiti`

I ❤️ Engines

`active_canvas`

`active_storage_dashboard`

`sqlite_dashboard`

`active_analytics`