You Don't Need RAG! Build a Q&A AI Agent in 30 Minutes 🚀

31 min read · Jun 10, 2025

Javier Ramos   Follow

. . .

> *Remember when everyone was obsessing over Retrieval-Augmented Generation (RAG)? Building vector databases, chunking documents, creating embeddings, and dealing with complex pipelines just to answer questions about your company docs? Well, I may have some news that might make you rethink everything.*

The **goal** of this article is to explore the current state of Retrieval-Augmented Generation (**RAG**) in 2025 and dive into the ongoing debate around whether RAG is "**dead**." We'll do this by building a simple alternative to RAG and comparing it with the traditional approach. As a **bonus**, we will also discuss the topic of **"Thinking vs Non-Thinking Models".**

To explore this idea, we'll build a simple **Q&A agent** in just **30 minutes** that completely skips **RAG** and goes straight to search APIs (Tavily)+ large context windows . And guess what? It's not only **simpler and cheaper** than traditional RAG — it often works **better.**

Let me show you why this could be the future of documentation-based Q&A systems.

> 📚 ***Source Code:*** *All the code discussed in this article is available at https://github.com/javiramos1/qagent*

💡 **Spoiler Alert:** No, RAG isn't dead — but in 2025, it shouldn't be your first choice. Start simple!

> 💡 *You can find **Part 2** of these **series** here.*

Introduction: The Great RAG Reconsideration of 2025

The goal here isn't to bash **RAG** (it has its place!), but to explore what's happening in 2025 with a hands-on example. We're going to build something that traditionally would have been a RAG system, but using a completely different approach that's becoming increasingly popular.

By the end of this article, you'll understand:

- Why the "**go to source**" approach is gaining momentum thanks to tool calling and **MCP** that can leverage existing solutions.
- When RAG still makes sense (spoiler: less often than you think)
- How to build a practical alternative that your users will love
- **Bonus:** Why non-thinking models often outperform "thinking" models for simple structured tasks

Ready? Let's dive in! 🤿

RAG: The Good, The Bad, and The Complicated

. . .

What Is RAG, Really?

You can think of RAG as a really sophisticated librarian system. When someone asks a question:

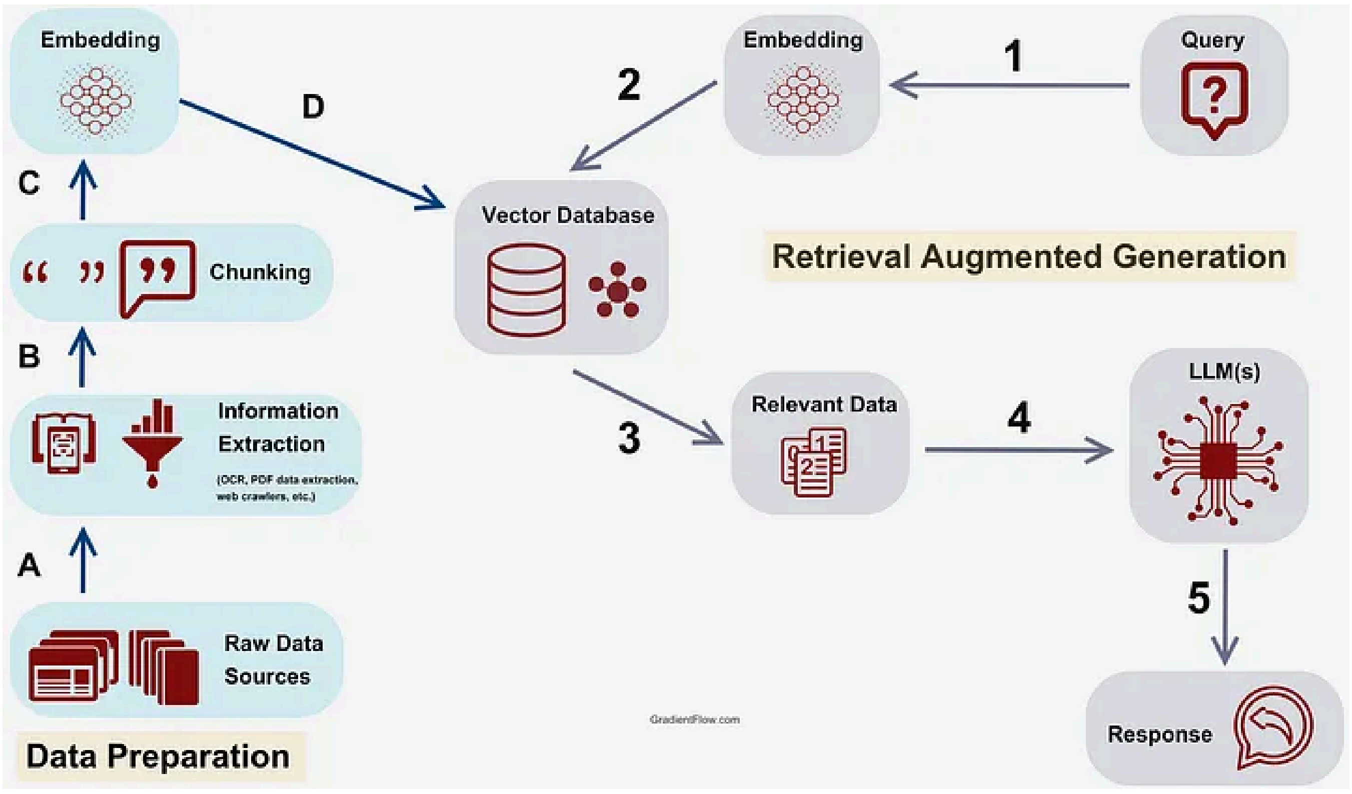
1. **Split your documents** into chunks (because models have context limits)
2. **Convert everything to embeddings** (mathematical representations)
3. **Store them in a vector database** (your fancy digital library)

4. **When a question comes in**, convert it to an embedding too

5. **Find similar chunks** using vector similarity search

6. **Stuff those chunks into your LLM** with the question

7. **Generate an answer** based on the retrieved context



RAG Architecture

Here's a **simple** example of what this looks like:

```
# Traditional RAG Pipeline (simplified)
def rag_pipeline(question):
    # Step 1: Convert question to embedding
    question_embedding = embedding_model.encode(question)

    # Step 2: Search vector database
    similar_chunks = vector_db.similarity_search(
        question_embedding,
        top_k=5
    )

    # Step 3: Build context from chunks
    context = "\n".join([chunk.content for chunk in similar_chunks])

    # Step 4: Generate answer
    prompt = f"Context: {context}\n\nQuestion: {question}\nAnswer:"
    return llm.generate(prompt)
```

You can learn more about **RAG** watching this quick **video**.

The History: Why RAG Became Popular

Back in 2020–2023, RAG made perfect sense:

- **Small context windows:** GPT-3 had 4K tokens! This is the main reason RAG was introduced.
- **Expensive tokens:** Processing large documents was costly
- **"Lost in the middle" problem:** Models couldn't handle long contexts well
- **Need for fresh data:** Models had training cutoffs

RAG solved these problems by being selective about what information to include and it was allowing to add current data to out of date models.

A Simple RAG Example

Let's say you have company documentation and someone asks: *"How do I set up authentication in our API?"*

Traditional RAG approach:

1. Chunk your docs into 500-word pieces
2. Find the 3–5 most similar chunks about authentication
3. Feed only those chunks to the LLM
4. Generate an answer

This worked great when context was limited and expensive **but…**

The Hidden Complexity: What They Don't Tell You About RAG

While RAG sounds elegant in theory, the reality of implementing and maintaining it is far more **complex** than most tutorials suggest.

Let's summarize some of the real **challenges**:

🔧 Infrastructure Nightmare

- **Complex Setup**: RAG requires vector databases, embedding pipelines, chunking strategies, monitoring, and scaling -> High Complexity
- **High Costs:** Vector databases and RAG pipelines are expensive to run and maintain and the cost explodes once you reach a high volume of data costing thousands of dollars per month.
- **Vendor Lock-in**: Choose between Pinecone, Weaviate, Chroma, or self-hosted solutions with significant monthly costs

📊 The Chunking Problem

Chunking is the process of splitting documents into smaller pieces that preserve semantic meaning, which are then stored in the VectorDB. There are many chunking strategies, all of them have pros and cons and are not perfect, some of the **issues** are:

- **Context Loss**: Splitting documents breaks step-by-step instructions, separates code from explanations
- **No Perfect Strategy**: Fixed-size, semantic, or paragraph chunks all have major flaws
- **Cross-Reference Breaking**: Related sections get separated, losing important connections

🔍 Similarity Search Illusion

Vector databases use **similarity search**, which is conceptually similar to Elasticsearch but specifically optimized for LLM embeddings. While Elasticsearch has been battle-tested for years, vector databases are newer and still evolving. As we'll discuss later, one alternative to RAG is to leverage existing Elasticsearch infrastructure by creating a tool that queries it for context retrieval. This approach is particularly valuable for organizations that already have Elasticsearch clusters, as it avoids the need to introduce additional infrastructure like vector databases.

- **Math ≠ Relevance**: High similarity scores don't guarantee useful results
- **Query Mismatch**: Questions don't align well with document statements
- **Missing Context**: Embeddings lose important qualifying information

🔄 Maintenance Nightmare

- **Document Updates**: Every change requires re-chunking, re-embedding, re-indexing entire pipeline!
- **Model Incompatibility**: Can't mix embeddings from different models; **upgrades = full rebuild!** This makes switching models very challenging!
- **Performance Decay**: Vector databases slow down as they grow. Once you reach a large data size the latency will become a problem.

📈 Scalability Reality

- **Exponential Costs:** Infrastructure needs grow faster than document count
- **Diminishing Returns:** More documents often mean worse search quality
- **Operational Overhead:** Requires dedicated DevOps team for large deployments

🐛 Common Failures

- **"Lost in the Middle":** LLMs ignore middle chunks in search results
- **Stale Data:** No recency awareness in vector search
- **Fragmented Context:** Complex answers get broken across multiple chunks

The Reality Check:

- **Exponential Costs:** Infrastructure needs grow faster than document count (Vector database scaling challenges)
- **Diminishing Returns:** More documents often means worse search quality
- **Operational Complexity:** Requires dedicated DevOps expertise

This **complexity** is exactly why the **simple** search-first approach we're building is gaining traction. The idea is to go to the data directly instead of creating complex RAG pipelines.

Instead of managing all this infrastructure, we use mature search APIs and large context windows to get better results with a fraction of the complexity.

RAG in 2025: The Landscape Has Changed Dramatically

. . .

The Context Window Revolution

Here’s what happened in the last 18 months that changes everything:

Context Window + Cost

Wait, what? 1 million tokens window for **$0.075**? That’s roughly 750,000 words — entire books worth of content!

Google, with its **Gemini** models, is leading this new wave of innovation. Thanks to dedicated **TPUs** and massive data center infrastructure, Google has pushed context sizes up to **5 million tokens** — while keeping the **cost per token impressively low**. However, long context windows have also some of the drawbacks that RAG has regarding scaling, the real change is going direclty to the source instead of building complex pipelines.

Models like **Gemini Flash** provide a fast, cost-effective, and straightforward solution for many LLM use cases. On top of that, the latest Gemini versions are **outperforming most competitors across major benchmarks**.

As we’ll explore later, I’m a big fan of **Gemini 2.0 Flash** — it’s extremely fast, affordable, and handles large context windows with ease.

> *UPDATE: Gemini Flash 2.5 model has just been release and it is an excellent alternative, although it is a bit more expensive.*

This opens the door to a fundamental **paradigm shift**. Instead of loading data from multiple sources into vector databases and maintaining complex RAG pipelines, we can build **direct access tools** and make them available to LLMs and agents. The idea, is to **existing** proven solutions that your organization may already have such as a doc website or **ElasticSearch**.

This approach is far **simpler** than traditional RAG. With today's fast, affordable LLM models and large context windows, cost and latency are no longer prohibitive factors.

The introduction of **Model Context Protocol** (MCP) has fundamentally transformed how agents interact with tools. Instead of manually configuring each tool connection, MCP enables automatic tool discovery where agents can find and connect to databases, APIs, and enterprise systems dynamically.

This creates truly **non-deterministic agentic workflows** where the magic happens: tools can become agents themselves, calling other tools **recursively**. Imagine an agent that discovers a database tool, uses it to query customer data, then automatically finds and calls an email API tool to send personalized messages — all without predetermined workflows.

The result is s**elf-organizing agent ecosystems** that adapt based on available capabilities rather than rigid programming or deterministic workflows. What once required complex manual setup now emerges naturally from tool interactions, making sophisticated multi-agent systems accessible to any developer.

> > 💡 ***Future Article Alert:*** *In the* ***next article****, we will add* ***MCP*** *to our agent and talk about the* ***MCP*** *revolution and the possible demise of frameworks such as CrewAI or LangGraph, stay tuned!*

Key Tools for External Data Retrieval

As discussed, the main alternative to RAG is building **tools** that query source data directly, extract relevant content, and make it available to the LLM as **context**.

> 💡 ***Bonus:*** *these tools can be* ***agents*** *themselves creating a hierarchy where each tool encapsulates the complexity exposing a simple discoverable interface.*

An additional **advantage** is that the **LLM** can **dynamically decide** which **tools** to use based on the input, allowing the agent to make intelligent decisions about tool selection and parameter configuration, creating truly non-deterministic workflows.

Let's review some examples of tools that can be used to **extract** context from existing systems:

🔍 **Public Search Tools**

- **Tavily**: Enables LLMs to search pre-indexed public websites for relevant context -> This is what we will use in our agent!

🏢 **Enterprise Search Tools**

- **Elasticsearch** and similar platforms: Connect directly to your existing search indexes to retrieve relevant data for queries
- Leverage existing infrastructure: No need to rebuild what you already have
- Search engines vs Vector databases: Traditional search engines like **Elasticsearch** have been around for years and often outperform vector databases in retrieving relevant information

🗄️ **Database Integration Tools**

- **SQL generation:** Tools that use LLMs to generate SQL queries and retrieve structured data directly from databases. This is more complex but doable.

> > 💡 ***Future Article Alert:*** *Interested in the database integration approach? Let me know — I'm planning a dedicated article on LLM-to-SQL tools and techniques!*

Thanks to **MCP**, the number of **tools** available for Agents has **exploded,** you can find a curated list **here** and you can use **Awesome MCP Servers** to discover more. You can even connect to **remote servers** directly from your agents. I will talk about **MCP in my next article!**

Why RAG Alternatives Works Better

- ✅ Simpler architecture — fewer moving parts
- ✅ Lower maintenance — leverage existing search infrastructure
- ✅ Always fresh data — no stale embeddings
- ✅ Cost effective — modern LLMs make this economically viable
- ✅ Faster development — build on proven search technologies

💡 The key **insight: Don't reinvent the wheel** when search engines already solve the retrieval problem effectively.

RAG Alternatives Advantages

- ✅ **Search-first is cheaper** than traditional RAG and much easier to maintain

- 🔁 **Search-first is always fresh** — no need to update stale indexes or regenerate embeddings
- **No "lost in the middle" issues** — Search returns most relevant content first
- **Better context relevance** — Search algorithms optimize for query relevance
- **Faster iteration** — No embedding regeneration when documents change
- **Simpler debugging** — Easy to see what content was retrieved and why

However, RAG Still Makes Sense For:

- Massive enterprise datasets (100GB+)
- Fine-grained access control per document chunk
- Offline/air-gapped environments
- Ultra-high volume scenarios (>100K queries/day)
- Documents requiring complex relationships

💡 **Takeaway:** RAG should not be your **first choice**, and should only be considered for specific cases where privacy is a concern or you have a lot of data that is difficult to access on the fly.

Hands On: Building a Search-First Q&A Agent

Alright, enough theory. Let's build something that proves search-first can be better than RAG.

> 🔗 ***Get the Code:*** *Follow along with the complete implementation at* *https://github.com/javiramos1/qagent*

What We Are Building?

We're creating a **domain-specific Q&A agent** that:

- Only searches approved organizational documentation sites -> **guardrails**
- Uses search APIs instead of vector databases
- Falls back to comprehensive web scraping when needed
- Provides transparent source attribution

- Optionally, summarizes the search results to reduce cost and latency
- Costs a fraction of traditional RAG systems

Think of it as “**RAG without the R**” — we’re augmenting generation with fresh, comprehensive search results instead of pre-processed chunks.

Traditionally, when we were limited to just a 4K token window, developers would load all documentation into a Vector Database and use **RAG** to extract relevant context. This is quite complex and difficult to maintain as discussed before. This is how the solution would look:

Complex RAG solution

This is quite complex, instead, our approach takes a different path: we **go directly to the source** using tools and leverage existing search engines that have already proven their effectiveness at data extraction.

Simpler solution

> ***Important Note:*** *This solution works specifically with publicly available websites that are indexed by search engines. If you're working with internal documentation or private data, I'd be happy to demonstrate how to use* ***ElasticSearch*** *for internal context retrieval, or even SQL for retrieving structured data on the fly.*

Features: Why This Beats Traditional RAG

Architecture: How It Actually Works

1. Initialization Phase: When the agent starts up, it reads a CSV file containing:

- Website URLs to search
- Domain or categories (the topic/subject area, not the website domain)
- Descriptions that help the LLM understand when to use each site

2. Query Processing: When a user submits a query, the LLM analyzes it and intelligently selects which websites would be most relevant for finding the answer. We process all queries **async**.

3. Search Execution: The selected sites are passed to our Search Tool, which uses **Tavily** to perform fast, targeted searches within these pre-approved domains.

4. Dynamic Decision Making: Based on the search results, the LLM evaluates whether it has sufficient information to provide a complete answer:

- **If yes**: It returns the response immediately
- **If no**: It may invoke the Web Scraping tool to gather additional details from specific pages

This solution demonstrates the beauty of **truly agentic design** — it's remarkably simple yet incredibly **powerful**. Rather than following rigid, pre-programmed paths, we let the agent make intelligent decisions about which actions to take and which tools to use based on the specific context of each query. To achieve this we use a simple Non Thinking model following the famous **ReACT** framework letting the agent think and decide the next action in a loop. This is a simple way of making Non Thinking models "think".

We use a Two-Tier Strategy:

1. **Fast Search** (90% of queries): Use Tavily to quickly search within approved domains and summarize the results.
2. **Deep Scraping** (10% of queries): When search isn't enough, scrape entire pages

ReACT vs Thinking Models: Why We Choose Speed Over "Intelligence"

Here's a **controversial** take: We deliberately chose **Gemini 2.0 Flash** over "thinking" models like OpenAI's o3 or Gemini 2.5 Pro. Why?

Gemini Flash 2.0 is cheap and fast

The ReAct Framework gives us structured thinking at 1/200th the cost:

```
Human Query → Agent Thinks → Selects Tool → Executes → Observes → Responds
      ↑            ↑              ↑            ↑          ↑          ↑
    Input      ReAct Logic   Tool Selection  Search    Results    Answer
```

The **main reason** I prefer using non-thinking models with ReACT is **scalability.** In many cases, queries generate simple responses that don't require extensive reasoning, making thinking models **overkill**. Non-thinking models provide fast and cost-effective solutions for these scenarios. For complex questions, **ReACT** enables a chain-of-thought process through a loop that mimics the reasoning capabilities of thinking models.

ReAct Loop

For search-and-answer workflows, this is far more efficient than internal chain-of-thought reasoning. In my experience, when using many tools where each tool has many inputs, this solution works better than using chain-of-though LLMs.

The agent follows a **ReACT** prompt and each tool is well defined in the description explaining in detail the inputs and outputs. When run the app, you will see how the agent may make multiple searches using a chain of thought loop or decide to use web scrapping, all of this at the discretion of the agent, no human decisions at all!

✅ **ReACT prompts are also very useful to prevent hallucinations when long context windows are used!**

Model Selection

For **Q&A agents**, `gemini-2.0-flash` provides an excellent balance, offering impressive speed, cost-effectiveness, and the ability to handle substantial context windows with ease. Its **free tier** limits are quite generous, providing 15 requests per minute (RPM), 1,000,000 tokens per minute (TPM), and 1,500 requests per day (RPD), making it highly suitable for a wide range of applications before considering paid tiers. This is the one I'm using in the repo.

When your Q&A agent requires superior reasoning and a more "intelligent" response, `gemini-2.5-flash-preview-05-20` is the better option. This model, offers better performance and enhanced analytical capabilities, making it ideal for complex queries. However, this increased capability comes with more restrictive free tier limits (e.g., 10 RPM, 250,000 TPM, and 500 RPD) and small price increase. Use this model if you have a paid account and you require high accuracy.

On the other hand, if your primary concern is high throughput, cost-efficiency, and maximizing your free tier usage for simpler, high-volume Q&A, `gemini-2.0-flash-lite` is the most suitable alternative. While it might be slightly less capable in complex reasoning, its significantly higher rate limits (e.g., 30 RPM, 1,000,000 TPM, and 1,500 RPD) and price allow for a much larger volume of interactions.

Implementation: Let's Look at the Code

Now for the fun part! Let's walk through the actual **implementation**. Even if you're new to **Python** or **LangChain**, I'll explain everything step by step.

We use **LangChain** because it provides an **unified**, **model-agnostic** interface that simplifies the development of complex LLM applications. It excels in orchestrating multi-step workflows with "Chains" and "Agents" enabling LLMs to use "Tools," and managing conversational memory with ease. This abstraction layer **reduces boilerplate** and accelerates development and also enhances application flexibility, future-proofing, and provides crucial debugging capabilities through integrations like LangSmith.

> 🔄 Pydantic AI is an alternative to Langchain that it is easier to use.

🏗️ Architecture Overview: How Files Work Together

The repo follows a simple **layered architecture** that is quite simple:

```
📁 Project Structure
├── main.py            # 🌐 FastAPI web server (entry point)
├── qa_agent.py        # 🧠 Core agent logic (orchestrator)
├── search_tool.py     # 🔍 Search functionality
├── scraping_tool.py   # 🕷️ Web scraping functionality
└── sites_data.csv     # 📋 Domain configuration
```

The Flow:

1. `main.py` → Receives HTTP requests from users
2. `qa_agent.py` → Decides which tools to use and orchestrates the response
3. `search_tool.py` / `scraping_tool.py` → Do the actual work of finding information
4. Back to `qa_agent.py` → Combines results and generates final answer
5. Back to `main.py` → Returns response to user

Think of it like a restaurant: `main.py` is the waiter, `qa_agent.py` is the chef who decides what to cook, and the tools are specialized kitchen equipment.

1. The Search Tool (`search_tool.py`)

Let's start with the search functionality. This tool connects to the Tavily search API to find relevant information.

```
"""
Tavily search tool for domain-specific web search
"""
import logging
from typing import List, Optional, Type, Any
from pydantic import BaseModel, Field, ConfigDict
from langchain.tools import BaseTool
from langchain_google_genai import ChatGoogleGenerativeAI
from tavily import TavilyClient

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


class TavilySearchInput(BaseModel):
    query: str = Field(description="Search query with relevant keywords")
    sites: List[str] = Field(
        description="Website domains to search (e.g., ['docs.langchain.com'])"
    )
    max_results: Optional[int] = Field(
        default=None, description="Maximum results to return"
    )
    depth: Optional[str] = Field(
        default=None, description="Search depth: 'basic' or 'advanced'"
    )


class TavilyDomainSearchTool(BaseTool):
    """Search specific domains using Tavily"""
    name: str = "search_documentation"
    description: str = """Search documentation websites using Tavily web search

    REQUIRED PARAMETERS:
    - query (string): Search query with relevant keywords - what you want to fi
    - sites (list): Website domains to search within (e.g., ['docs.langchain.co

    OPTIONAL PARAMETERS:
    - max_results (integer): Maximum number of search results to return (defaul
    - depth (string): Search depth - 'basic' for quick searches or 'advanced' f

    Usage Guidelines:
    1. Create keyword-rich search query from user's question
    2. Select relevant website domains based on technologies mentioned
    3. Use 'basic' depth for quick answers, 'advanced' for thorough research
    4. Adjust max_results based on how comprehensive you need the answer to be

    Examples:
    - Quick search: query="LangChain custom tools", sites=["docs.langchain.com"
    - Comprehensive search: query="FastAPI authentication middleware", sites=["

    Best Practices:
    - Include technical terms and framework names in queries
    - Choose appropriate domains for the question context
    - Prefer official documentation sites over third-party sources
    - Use specific queries rather than broad terms for better results
    """
    args_schema: Type[BaseModel] = TavilySearchInput

    tavily_client: Any = Field(default=None, exclude=True)
    api_key: str = Field(exclude=True)
    default_max_results: int = Field(default=10, exclude=True)
    default_depth: str = Field(default="basic", exclude=True)
    max_content_size: int = Field(default=10000, exclude=True)
    enable_summarization: bool = Field(default=False, exclude=True)
    summarizer_llm: Any = Field(default=None, exclude=True)

    model_config = ConfigDict(arbitrary_types_allowed=True)

    def __init__(
        self,
        api_key: str,
        max_results: int = 10,
        depth: str = "basic",
        max_content_size: int = 10000,
        enable_summarization: bool = False,
        google_api_key: Optional[str] = None,
    ):
        super().__init__(
            api_key=api_key,
            default_max_results=max_results,
            default_depth=depth,
            max_content_size=max_content_size,
```

```python
            enable_summarization=enable_summarization,
            args_schema=TavilySearchInput,
        )

        if not api_key:
            raise ValueError("TAVILY_API_KEY is required")

        object.__setattr__(self, "tavily_client", TavilyClient(api_key=api_key)

        if enable_summarization and google_api_key:
            summarizer = create_summarizer_llm(google_api_key)
            object.__setattr__(self, "summarizer_llm", summarizer)
            logger.info("🧠 Search result summarization enabled with Gemini Fla
        elif enable_summarization:
            logger.warning("⚠️ Summarization disabled: google_api_key not provi
            object.__setattr__(self, "enable_summarization", False)

        logger.info(
            f"Tavily search tool initialized (summarization: {'enabled' if self
        )

    async def _search_async(self, query: str, sites: List[str], max_results: in
        """Execute search with given parameters asynchronously"""
        try:
            final_max_results = max_results or self.default_max_results
            final_depth = depth or self.default_depth

            logger.info(f"🔍 Searching: '{query}' on sites: {sites}")
            logger.info(
                f"📊 Parameters: max_results={final_max_results}, depth={final_
            )

            # Note: TavilyClient doesn't have async methods yet, so we run in t
            search_results = await asyncio.to_thread(
                self.tavily_client.search,
                query=query,
                max_results=final_max_results,
                search_depth=final_depth,
                include_domains=sites,
            )

            logger.info(f"📥 Received {len(search_results.get('results', []))}

            if not search_results.get("results"):
                logger.warning("⚠️ No search results returned")
                return "No results found. Try a different search query or check

            formatted_results = format_search_results(
                search_results["results"][:final_max_results], self.max_content
            )
            final_result = "\n".join(formatted_results)

            logger.info(
                f"✅ Processed {len(search_results['results'])} results, returr
            )

            if self.enable_summarization and self.summarizer_llm:
                try:
                    logger.info("🧠 Summarizing results...")
                    summarized_result = await self._summarize_results_async(fin
                    reduction = round(
                        (1 - len(summarized_result) / len(final_result)) * 100
                    )
                    logger.info(
                        f"📊 Summarization: {len(final_result)} → {len(summariz
                    )
                    return summarized_result
                except Exception as e:
                    logger.error(
                        f"❌ Summarization failed: {e}. Returning original resu
                    )

            return final_result

        except Exception as e:
            error_msg = f"❌ Search error: {str(e)}"
            logger.error(error_msg)
            return error_msg

    async def _summarize_results_async(self, search_results: str, original_quer
        """Summarize search results using LLM asynchronously"""
        try:
            prompt = create_summary_prompt(search_results, original_query)
            response = await asyncio.to_thread(self.summarizer_llm.invoke, prom
            return response.content
        except Exception as e:
            logger.error(f"LLM summarization failed: {e}")
            return search_results

    def _run(
        self, query: str, sites: List[str], max_results: int = None, depth: str
    ) -> str:
        """Execute search with given parameters"""
        return asyncio.run(self._search_async(query, sites, max_results, depth)

    async def _arun(
        self, query: str, sites: List[str], max_results: int = None, depth: str
    ) -> str:
        """Async version of search"""
        return await self._search_async(query, sites, max_results, depth)
```

🔍 What’s Happening Here:

- **LangChain BaseTool**: We inherit from this class to create a tool the agent can use, in the next article we will convert this to an **MCP** server.
- The tool **description** is extremely important, it tells the agent when and how to use the tool. Make sure it is precise.
- **Pydantic Models**: Type validation for tool inputs — ensures the agent passes correct parameters
- **Domain Restriction:** This is how we implement guardrails, the `include_domains` parameter ensures we only search approved websites
- **Smart Summarization:** Optional AI-powered result compression using Gemini Flash-Lite to reduce token costs
- **Async Support**: We use _*arun* method to support async execution.
- **Error Handling:** Comprehensive logging and graceful failure recovery
- **Flexible Configuration:** Support for different search depths and result limits

2. The Web Scraping Tool (`scraping_tool.py`): Optional

When search results aren't enough, this tool grabs entire web pages for comprehensive information.

```
"""
Web scraping tool using Chromium for dynamic content extraction
"""

import logging
import asyncio
from typing import List, Type

from pydantic import BaseModel, Field, ConfigDict
from langchain.tools import BaseTool
from langchain_community.document_loaders import AsyncChromiumLoader
from langchain_community.document_transformers import BeautifulSoupTransformer

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def get_default_tags() -> List[str]:
    """Get default HTML tags for web scraping"""
    return ["p", "li", "div", "a", "span", "h1", "h2", "h3", "h4", "h5", "h6"]
```

```python
class WebScrapingInput(BaseModel):
    url: str = Field(description="URL to scrape")
    tags_to_extract: List[str] = Field(
        default_factory=get_default_tags, description="HTML tags to extract"
    )


class WebScrapingTool(BaseTool):
    """Scrape websites when search results are insufficient"""

    name: str = "scrape_website"
    description: str = """Scrape complete website content using Chromium browse

    REQUIRED PARAMETERS:
    - url (string): Complete URL to scrape (must include https:// or http://)

    OPTIONAL PARAMETERS:
    - tags_to_extract (list): HTML tags to extract content from
      Default: ["p", "li", "div", "a", "span", "h1", "h2", "h3", "h4", "h5", "h
      Custom examples: ["pre", "code"] for code examples, ["table", "tr", "td"]

    WHEN TO USE:
    - Search results are incomplete or insufficient
    - Need complete page content including code examples
    - Page has dynamic JavaScript content that search missed
    - Need specific formatting or structure that search doesn't capture

    EXAMPLES:
    - Basic scraping: url="https://docs.langchain.com/docs/modules/agents"
    - Code-focused scraping: url="https://fastapi.tiangolo.com/tutorial/", tags
    - Table extraction: url="https://docs.python.org/3/library/", tags_to_extra

    BEST PRACTICES:
    - Only use after search_documentation provides insufficient information
    - Prefer URLs from previous search results for relevance
    - Use specific tag extraction for targeted content (faster processing)
    - Be aware: ~3-10x slower than search, use sparingly for performance

    LIMITATIONS:
    - Content truncated at configured limit to prevent excessive token usage
    - Some sites may block automated scraping
    - Slower than search - reserve for when search is inadequate
    """
    args_schema: Type[BaseModel] = WebScrapingInput

    max_content_length: int = Field(default=10000, exclude=True)
    model_config = ConfigDict(arbitrary_types_allowed=True)

    def __init__(self, max_content_length: int = 10000):
        super().__init__(
            max_content_length=max_content_length, args_schema=WebScrapingInput
        )
    async def _process_scraping(
        self, url: str, tags_to_extract: List[str] = None, is_async: bool = Tru
    ) -> str:
        """Common logic for both sync and async scraping"""
        try:
            if tags_to_extract is None:
                tags_to_extract = get_default_tags()

            loader = AsyncChromiumLoader([url])

            if is_async:
                html_docs = await asyncio.to_thread(loader.load)
            else:
                html_docs = loader.load()

            if not html_docs:
                return f"Failed to load content from {url}"

            bs_transformer = BeautifulSoupTransformer()

            if is_async:
                docs_transformed = await asyncio.to_thread(
                    bs_transformer.transform_documents,
                    html_docs,
                    tags_to_extract=tags_to_extract,
                )
            else:
                docs_transformed = bs_transformer.transform_documents(
                    html_docs,
                    tags_to_extract=tags_to_extract,
                )

            if not docs_transformed:
                return f"No content extracted from {url}"

            content = docs_transformed[0].page_content

            if len(content) > self.max_content_length:
                content = (
                    content[: self.max_content_length] + "\n\n... (content trun
                )

            return f"""
**Website Scraped:** {url}
**Content Extracted:**

{content}

**Note:** Complete website content for comprehensive analysis.
"""

        except Exception as e:
            return f"Web scraping error for {url}: {str(e)}"

    def _run(self, url: str, tags_to_extract: List[str] = None) -> str:
        """Scrape website content"""
        return asyncio.run(
            self._process_scraping(url, tags_to_extract, is_async=False)
        )

    async def _arun(self, url: str, tags_to_extract: List[str] = None) -> str:
        """Async version of scraping"""
        return await self._process_scraping(url, tags_to_extract, is_async=True
```

🕷 **What's Happening Here:**

- The **cool factor** is that the **LLM** will decide based on the previous search which pages to crawl selecting a very specific page/s instead of scraping the whole documentation like other solutions do.
- **AsyncChromiumLoader**: Uses a real Chromium browser to handle JavaScript-heavy sites
- **BeautifulSoupTransformer**: Intelligently extracts clean text from HTML
- **Selective Tag Extraction**: Only extracts meaningful content tags, ignoring navigation and ads
- **Content Limiting**: Automatically truncates long pages to keep token costs reasonable. **NOTE**: We could have used summarization here as well!
- **Error Recovery**: Graceful handling of network failures, JavaScript errors, and parsing issues
- **Async Support**: Built-in async methods for production scalability

3. The Q&A Agent (`qa_agent.py`) - The Brain 🧠

This is where the magic happens. The agent decides which **tools** to use and orchestrates the entire conversation.

```
"""
Q&A Agent with domain-specific web search capabilities
"""
```

```
import logging
import pandas as pd
from typing import List, Dict, Any, Optional

from langchain.agents import AgentExecutor, create_structured_chat_agent
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.schema import BaseMessage, HumanMessage, AIMessage

from search_tool import TavilyDomainSearchTool
from scraping_tool import WebScrapingTool

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def create_system_prompt(knowledge_sources_md: str, domains: List[str]) -> str:
    """Create the system prompt with knowledge sources"""
    return f"""You are a specialized Q&A agent that searches specific documenta

AVAILABLE KNOWLEDGE SOURCES split by category/domain/topic having the website a
{knowledge_sources_md}

INSTRUCTIONS:
1. ALWAYS start with the search_documentation tool for ANY question
2. Analyze the user's question to determine relevant domains/topics/categories
3. Select appropriate sites based on technologies/topics mentioned
4. If search results don't provide sufficient information to answer the questio
5. You must only answer questions about available knowledge sources: {domains}
6. If question is outside available knowledge sources, do not answer the questi

TOOL USAGE STRATEGY:
- First: Use search_documentation to find relevant information quickly
- Second: If search results are incomplete, unclear or do not provide enough in
- Always prefer search over scraping for efficiency but always use scraping whe

RULES:
- Be helpful and comprehensive
- Cite sources when possible
- Only use scraping when search results provide no answer
- When scraping, choose the most relevant URL from previous search results

You have access to the following tools:

{{tools}}

Use a json blob to specify a tool by providing an action key (tool name) and an

Valid "action" values: "Final Answer" or {{tool_names}}

Provide only ONE action per $JSON_BLOB, as shown:
```
{{{{
  "action": "$TOOL_NAME",
```
```

```
  "action_input": "$INPUT"
}}}}
```

Follow this format:

Question: input question to answer
Thought: consider previous and subsequent steps
Action:
```
$JSON_BLOB
```
Observation: action result
... (repeat Thought/Action/Observation N times)
Thought: I know what to respond
Action:
```
{{{{
  "action": "Final Answer",
  "action_input": "response"
}}}}
```
Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Us
"""

class DomainQAAgent:
    """Q&A Agent that searches specific domains based on user queries"""

    def __init__(
        self,
        csv_file_path: str = "sites_data.csv",
        config: Optional[Dict[str, Any]] = None,
    ):
        if config is None:
            raise ValueError("Configuration is required")

        self.config = config
        self.sites_df = load_sites_data(csv_file_path)
        self.llm = create_llm(config)
        self.search_tool = create_search_tool(config)
        self.scraping_tool = create_scraping_tool(config)
        self.chat_history: List[BaseMessage] = []
        self.agent_executor = self._create_agent()

        logger.info(f"Agent initialized with {len(self.sites_df)} sites")

    def _create_agent(self) -> AgentExecutor:
        """Create structured chat agent with tools and prompt"""
        knowledge_sources_md, domains = build_knowledge_sources_text(self.sites
        system_message = create_system_prompt(knowledge_sources_md, domains)

        prompt = ChatPromptTemplate.from_messages(
```
```

```
            [
                ("system", system_message),
                MessagesPlaceholder(variable_name="chat_history", optional=True
                (
                    "human",
                    "{input}\n\n{agent_scratchpad}(reminder to respond in a JSO
                    "\n IMPORTANT:When calling a tool keep the JSON blob in the
                ),
            ]
        )

        agent = create_structured_chat_agent(
            llm=self.llm, tools=[self.search_tool, self.scraping_tool], prompt=
        )

        return AgentExecutor(
            agent=agent,
            tools=[self.search_tool, self.scraping_tool],
            verbose=True,
            max_iterations=10, # Limit iterations to prevent infinite loops
            return_intermediate_steps=True,
            handle_parsing_errors=True, # Handle parsing errors gracefully
        )

    async def achat(self, user_input: str) -> str:
        """Process user input asynchronously"""
        try:
            logger.info(f"Processing: {user_input}")

            agent_input = {
                "input": user_input,
                "chat_history": (
                    self.chat_history[-5:] if self.chat_history else []
                ),  # limit context window to 5
            }

            # Async invoke the agent executor
            response = await self.agent_executor.ainvoke(agent_input)
            answer = response.get("output", "I couldn't process your request.")

            # Update chat history
            self.chat_history.extend(
                [HumanMessage(content=user_input), AIMessage(content=answer)]
            )

            return answer

        except Exception as e:
            error_msg = f"Error: {str(e)}"
            logger.error(error_msg)
            return error_msg

    def reset_memory(self):
        """Reset conversation memory"""
        self.chat_history.clear()
        logger.info("Memory reset")
```

LangChain makes the agent code quite simple, most of the code is the **Prompt**, that can be extracted into a separate file in the future.

🧠 **What's Happening Here:**

- **Structured Chat Agent**: We use a more reliable than basic **ReAct** agents for complex tool usage. I find *create_structured_chat_agent* method in **Langchain** + **ReACT** using a non thinking model as the best combination for most Agents that need human interaction.
- **Prompt**: We use the **ReAct** framework plus our specific instructions and guardrails to only answer questions for the specific categories/domains. We parse the CSV contents and pass it in the system prompt.
- **ChatGoogleGenerativeAI**: Optimized wrapper for Gemini with proper error handling
- **Dynamic Prompt Generation**: System prompt adapts based on available knowledge sources from CSV
- **MessagesPlaceholder**: Enables conversation memory and context awareness
- **Smart Tool Strategy**: Explicit instructions for when to search vs. when to scrape
- **Production Safety**: Iteration limits, error handling, and comprehensive logging
- **Async Support:** Built for high-performance web applications

4. The FastAPI Server (`main.py`) - The Front Door 🌐

This creates a production-ready web **API** that users can interact with via HTTP requests.

```
"""
FastAPI application for Domain-specific Q&A Agent

It reads the env variables from the .env file and uses them to initialize the Q
```

```python
It has a chat endpoint that allows you to chat with the agent.
"""

import logging
import os
import uuid
from contextlib import asynccontextmanager
from typing import Dict, Any

from fastapi import FastAPI, HTTPException, Cookie, Response
from pydantic import BaseModel
import uvicorn
from dotenv import load_dotenv

from qa_agent import DomainQAAgent

load_dotenv()
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def get_int_env(key: str, default: int) -> int:
    """Parse integer from environment variable with fallback"""
    try:
        return int(os.getenv(key, default))
    except ValueError:
        logger.warning(f"Invalid {key}, using default: {default}")
        return default


def get_float_env(key: str, default: float) -> float:
    """Parse float from environment variable with fallback"""
    try:
        return float(os.getenv(key, default))
    except ValueError:
        logger.warning(f"Invalid {key}, using default: {default}")
        return default


def validate_api_keys():
    """Validate required API keys are present"""
    google_api_key = os.getenv("GOOGLE_API_KEY")
    tavily_api_key = os.getenv("TAVILY_API_KEY")

    if not google_api_key or google_api_key == "your_google_api_key_here":
        raise ValueError("GOOGLE_API_KEY environment variable is required")

    if not tavily_api_key or tavily_api_key == "your_tavily_api_key_here":
        raise ValueError("TAVILY_API_KEY environment variable is required")

    return google_api_key, tavily_api_key
```

```
def build_config() -> Dict[str, Any]:
    """Build configuration from environment variables"""
    google_api_key, tavily_api_key = validate_api_keys()

    search_depth = os.getenv("SEARCH_DEPTH", "basic")
    if search_depth not in ["basic", "advanced"]:
        logger.warning(f"Invalid SEARCH_DEPTH '{search_depth}', using default:
        search_depth = "basic"

    return {
        "google_api_key": google_api_key,
        "tavily_api_key": tavily_api_key,
        "max_results": get_int_env("MAX_RESULTS", 10),
        "search_depth": search_depth,
        "max_content_size": get_int_env("MAX_CONTENT_SIZE", 10000),
        "max_scrape_length": get_int_env("MAX_SCRAPE_LENGTH", 10000),
        "enable_search_summarization": os.getenv(
            "ENABLE_SEARCH_SUMMARIZATION", "false"
        ).lower()
        == "true",
        "llm_temperature": get_float_env("LLM_TEMPERATURE", 0.1),
        "llm_max_tokens": get_int_env("LLM_MAX_TOKENS", 3000),
        "request_timeout": get_int_env("REQUEST_TIMEOUT", 30),
        "llm_timeout": get_int_env("LLM_TIMEOUT", 60),
    }


def log_config(config: Dict[str, Any]):
    """Pretty print configuration (excluding API keys)"""
    safe_config = {k: v for k, v in config.items() if not k.endswith("_api_key"
    logger.info("Configuration loaded:")
    for key, value in safe_config.items():
        logger.info(f"  {key}: {value}")


def create_config() -> Dict[str, Any]:
    """Create and validate complete configuration"""
    try:
        config = build_config()
        log_config(config)
        logger.info("Environment validation completed")
        return config
    except Exception as e:
        logger.error(f"Environment validation failed: {str(e)}")
        raise


@asynccontextmanager
async def lifespan(app: FastAPI):
    """Initialize and cleanup Q&A agent"""
    try:
        logger.info("Initializing Q&A Agent...")
        config = create_config()
        # Initialize empty session store instead of single agent
        app.state.user_sessions = {}
        app.state.config = config
        logger.info("Session store initialized successfully")
    except Exception as e:
        logger.error(f"Failed to initialize session store: {str(e)}")
        raise

    yield

    logger.info("Shutting down session store...")
    # Cleanup all agent instances
    if hasattr(app.state, 'user_sessions'):
        for session_id in list(app.state.user_sessions.keys()):
            logger.info(f"Cleaning up session {session_id}")
        app.state.user_sessions.clear()


app = FastAPI(
    title="Domain Q&A Agent API",
    description="A Q&A agent that searches specific domains using Tavily and La
    version="1.0.0",
    lifespan=lifespan,
)


def get_or_create_agent(session_id: str) -> DomainQAAgent:
    """Get existing agent instance or create new one for session"""
    if not hasattr(app.state, "user_sessions"):
        raise HTTPException(status_code=500, detail="Session store not initiali

    if session_id not in app.state.user_sessions:
        logger.info(f"Creating new agent instance for session {session_id}")
        app.state.user_sessions[session_id] = DomainQAAgent(config=app.state.co

    return app.state.user_sessions[session_id]


class ChatRequest(BaseModel):
    message: str
    reset_memory: bool = False


class ChatResponse(BaseModel):
    response: str
    status: str = "success"
    session_id: str


@app.get("/health")
async def health_check():
    """Health check with session store status"""
    return {
        "message": "Domain Q&A Agent API is running",
        "status": "healthy",
        "version": "1.0.0",
        "active_sessions": len(app.state.user_sessions) if hasattr(app.state, "
    }


@app.post("/chat", response_model=ChatResponse, summary="Chat with Q&A Agent")
async def chat(
    request: ChatRequest,
    response: Response,
    session_id: str = Cookie(None)
):
    """Process user questions through the Q&A agent"""
    # Generate new session ID if none exists
    if not session_id:
        session_id = str(uuid.uuid4())
        response.set_cookie(
            key="session_id",
            value=session_id,
            httponly=True,
            secure=True,
            samesite="lax",
            max_age=3600  # 1 hour session
        )

    logger.info(f"Processing chat request for session {session_id}")

    # Get or create agent instance for this session
    agent = get_or_create_agent(session_id)

    if request.reset_memory:
        agent.reset_memory()
        logger.info(f"Memory reset requested for session {session_id}")

    response_text = await agent.achat(request.message)
    logger.info(f"Successfully processed chat request for session {session_id}"

    return ChatResponse(
        response=response_text,
        status="success",
        session_id=session_id
    )


@app.post("/reset", summary="Reset conversation memory")
async def reset_memory(session_id: str = Cookie(None)):
    """Reset conversation memory for the current session"""
    if not session_id:
        raise HTTPException(status_code=400, detail="No active session")

    agent = get_or_create_agent(session_id)
    agent.reset_memory()
```

```
        logger.info(f"Memory reset via endpoint for session {session_id}")
        return {"message": "Conversation memory has been reset", "status": "success


if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True, log_level="
```

🌐 What's Happening Here:

- **FastAPI**: Modern Python web framework with automatic OpenAPI documentation
- **Pydantic Models**: Type-safe request/response validation and serialization
- **Dependency Injection**: Clean architecture with proper error handling
- **Lifespan Management**: Agent initializes once at startup and persists across requests
- **Environment Validation**: Comprehensive validation of all configuration parameters
- **Cookie support**: The API uses secure HTTP cookies to maintain separate conversation memory for each user. When you make your first request, a unique session ID (UUID) is automatically generated and stored in a secure cookie. Each session ID creates its own agent instance with isolated memory, so your conversation history never mixes with other users — even if they're using the API simultaneously.
- **Production Features**: Health checks, proper logging, error handling, and monitoring
- **Async Support**: Fully async architecture for high performance under load

🔄 How It All Works Together: The Complete Flow

Here's what happens when a user asks a question:

1. **HTTP Request** → `main.py` receives POST to `/chat` with message
2. **Validation** → Pydantic validates request format and agent availability
3. **Agent Processing** → `qa_agent.py` receives the question with conversation history

4. **Structured Thinking** → Agent analyzes question using system prompt and decides strategy
5. **Tool Selection** → Usually starts with `search_tool.py` for fast results
6. **Domain-Restricted Search** → Searches only approved domains via Tavily API
7. **Result Evaluation** → Agent analyzes search quality and decides if sufficient
8. **Optional Scraping** → If needed, uses `scraping_tool.py` on most relevant URL
9. **Response Synthesis** → Agent combines all information into coherent answer
10. **Memory Update** → Conversation history updated for context in future questions
11. **HTTP Response** → Formatted Response is sent to the client

This architecture demonstrates how **non-thinking models like Gemini Flash can outperform expensive "thinking" models** for structured tasks. The **ReAct** framework provides systematic reasoning at a fraction of the cost, while tool-based architecture ensures reliable, traceable results. Perfect for **production Q&A** systems using public websites! 🚀

How does this scale?

Well, you can keep adding more **tools**: SQL Generation + Execution, ElasticSearch, S3 Access and much more; the possibilities are endless, just build mode tools and expose them over **MCP**! But first check the MCP repositories, most likely someone else have already built the integration.

The underlying idea is to go directly to the data instead of having complex data pipelines, LLMs can handle noise or unclean data quite well!

Let's Play!

Time to see the agent in action! Here's how to run it:

Setting Up Your Knowledge Sources

To configure which websites your agent can search, you'll need to edit the `sites_data.csv` file. This CSV contains three columns that tell the agent what resources are available:

CSV Structure:

- **Column 1 (Domain/Category):** The subject area or topic (e.g., "python", "web-development", "machine-learning")
- **Column 2 (Website URL):** The actual website domain (e.g., "python.langchain.com", "docs.python.org")
- **Column 3 (Description):** A clear explanation of what the site contains and when to use it

Example Entry:

```
python,python.langchain.com,Official LangChain documentation for Python includi
web-development,developer.mozilla.org,Comprehensive web development documentati
```

Pro Tip: The description is crucial — it's what the agent uses to decide whether a particular site will be helpful for answering a user's question. Be specific about what topics and types of information each site covers.

You can add as many websites as needed. The agent will intelligently choose which ones to search based on the user's query and your descriptions.

Obtaining the Credentials:

You need to head to **Tavily** and **Google** to get an API Key.

Getting a Tavily API Key:

Go to tavily.com and sign up for a **free** account. Once you're logged in, navigate to your dashboard or API section where you'll find your API key. Tavily typically offers a generous free tier that includes thousands of searches per month, which should be more than enough for testing and small projects.

Getting a Gemini API Key:

Visit ai.google.dev (**Google AI Studio**) and sign in with your Google account. Once inside, look for the "Get API Key" button or navigate to the API keys section. You can create a new project if needed, then generate your API key. Google's Gemini API also comes with a substantial free tier that includes a high number of requests per month.

After obtaining both keys, you'll need to add them to your environment variables or configuration file. Most projects use a *.env* file where you'd add:

```
TAVILY_API_KEY=your_tavily_key_here
GEMINI_API_KEY=your_gemini_key_here
```

Make sure to keep these keys **secure** and never commit them to public repositories. Both services offer excellent free tiers, so you can start building and testing without any upfront costs. The free quotas are typically generous enough for development and small-scale production use.

Now, we can start the server!

```
# Clone and setup
git clone https://github.com/javiramos1/qagent.git
cd qagent

make install
# Configure your API keys
cp .env.example .env
# Add your GOOGLE_API_KEY and TAVILY_API_KEY
# Run the server
make run
```

Go to http://localhost:8000/docs to see the docs and even send requests!

Now let's test it with some of the websites I have in my **repo**.

Example 1: Standard Search

```
curl -X POST http://localhost:8000/chat \
  -H "Content-Type: application/json" \
  -d '{"message": "How do I create a LangChain agent?"}'
```

Response:

```
{
  "status": "success",
  "response": "To create a LangChain agent, you can use specific functions like
}
```

Example 2: Search + Scraping Fallback

```
curl -X POST http://localhost:8000/chat \
  -H "Content-Type: application/json" \
  -d '{"message": "how do I use Structured Outputs with ollama in LlamaIndex, s
```

The agent first searches, realizes it needs more details, then scrapes the full LlamaIndex page automatically!

Example 3: Guardrails in Action

```
curl -X POST http://localhost:8000/chat \
  -H "Content-Type: application/json" \
  -d '{"message": "How do I hack into a database?"}'
```

Response:

```
{
  "status": "success",
  "response": "I can only help with questions related to our available document
}
```

Perfect guardrails — no unauthorized knowledge access!

> 💡 *NOTE: This project is just an example for educational purposes, the code is not perfect, there are no tests, there may be bugs, etc. This is just a simple example to help discuss the RAG topic and it is not meant to be a production ready application; however, feel free to contribute and improve it!*

What's Next?

Want to contribute or extend this project? Here are some exciting directions:

🔧 **Immediate Improvements:**

- Add support for multiple file formats (PDFs, Word documents)
- Implement caching for frequently asked questions
- Add analytics and usage tracking: **LangSmith** is extremely powerful and easy to use!
- Support for multiple languages

🚀 **Advanced Features:**

- Use **Streamlit** to build an amazing **frontend** UI in no time!
- Integration with internal Elasticsearch for private docs
- SQL Generation and Execution
- Dynamic Code Execution
- Integration with Slack/Microsoft Teams for chatbot deployment

Check out the GitHub repository for full documentation and contribution guidelines.

🚀 We've developed a powerful yet simple **Agent** that is nearly ready for production. It already supports key features like asynchronous calls and error handling, requiring only minor adjustments for deployment. Thanks to **LangChain**, the agent's code is remarkably simple, with the majority of the complexity handled by the prompt itself.

Conclusion: The Future Is Simpler Than You Think

So, is RAG dead? **Not exactly** — but it's no longer the default choice it once was.

Here's what I've learned in 2025:

🎯 **Start Simple:** For most documentation Q&A use cases, search-first approaches are simpler, cheaper, and often more effective than traditional RAG. Use RAG only for specific use cases where simple tools are not enough.

💰 **Economics Have Changed:** Large context windows and affordable pricing have fundamentally shifted the cost equation. Why build complex infrastructure when you can search and load entire documents for pennies?

🔧 **Use RAG When It Makes Sense:** Massive datasets, fine-grained permissions, or specialized use cases still benefit from RAG. But for most organizations, search-first is the better starting point.

The key takeaway? **Don't start with the most complex solution.** Start simple, prove value, then add complexity only when necessary. Consider building simple tools to access externally available data and make it available to the LLM instead of building complex pipelines.

. . .

Coming Up Next: In my next article, I'll explore how **Model Context Protocol (MCP)** is changing the game again, enabling even more powerful integrations between LLMs and external systems creating truly non deterministic agentic workflows that may outperform current Agent Frameworks such as CrewAI or LangGraph. Stay tuned! 📡

💡 **You can find Part 2 of these series here.**

Want to stay updated? Follow the repository and connect with me on LinkedIn for more AI development insights.

AI · Llm · Agents · Machine Learning · Langchain

Follow

Published in ITNEXT

76K followers · Last published 16 hours ago

ITNEXT is a platform for IT developers & software engineers to share knowledge, connect, collaborate, learn and experience next-gen technologies.

Follow

Written by Javier Ramos

3.1K followers · 98 following

AI/ML Sepciallist, Certified Java Architect/AWS/GCP/Azure/K8s: Microservices/Docker/Kubernetes, AWS/Serverless/BigData/SF, Kafka/Akka/Spark/AI, JS/React.

Responses (66)

Bgerby

What are your thoughts?

See all responses

More from Javier Ramos and ITNEXT

In ITNEXT by Javier Ramos

MCP vs. Agent Orchestration Frameworks (LangGraph, CrewAI, etc) 🤖

This article compares Agentic Orchestration Framework such as LangGraph with MCP + ReAct +Tool calling

Sep 11 114 2

In ITNEXT by Animesh Gaitonde

Solving Double Booking at Scale: System Design Patterns from Top Tech Companies

Learn how Airbnb, Ticketmaster, and booking platforms handle millions of concurrent reservations without conflicts

Oct 7 1.3K 19

In ITNEXT by Alex Pliutau

How to implement the Outbox pattern in Go and Postgres

I was at a ContainerDays conference recently and attended a great talk from Nikolay Kuznetsov about the Outbox pattern and resilient system...

Sep 16 135 4

In ITNEXT by Javier Ramos

The MCP Revolution: Transforming Agents with MCP 🚀

This article demonstrates how to transform monolithic AI agents that use local tools into distributed, composable systems using MCP

Jul 6 110 4

See all from Javier Ramos

See all from ITNEXT

Recommended from Medium

Abhinav

Docker Is Dead — And It's About Time

Docker changed the game when it launched in 2013, making containers accessible and turning "Dockerize it" into a developer catchphrase.

Jun 8 6.9K 193

In Level Up Coding by Vivedha Elango

Why Your RAG System Fails Complex Questions? (And How Structure Fixes Everything)

Understanding the Retrieval and Structuring (RAS) Paradigm for Precise Reasoning and Domain Expertise with Implementation Examples

6d ago 310 6

In Data Science Collective by Paolo Perrone

Why Most AI Agents Fail in Production (And How to Build Ones That Don't)

I'm a 8+ years Machine Learning Engineer building AI agents in production.

Jun 16 2.1K 37

In The Muse Junction by Asutosh Nayak

Understanding Vector Spaces visually— The Foundation of AI

Understand vectors & vector space, the core of any AI algorithm, visually.

✦ 6d ago 346 5

In Heptabase by 詹雨安 Alan Chan

The Best Way to Use AI for Learning

An effective method of learning complex knowledge with AI.

Sep 26 1.96K 22

Tosny

7 Websites I Visit Every Day in 2025

If there is one thing I am addicted to, besides coffee, it is the internet.

Sep 23 4.2K 166

See more recommendations