Local RAG: Search Your Documents with a Private AI

January 30, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** RAG lets your local LLM answer questions about your own files — PDFs, notes, code, whatever — without fine-tuning or sending data to the cloud. The fastest way to start: install Ollama, pull a model and an embedding model, then use Open WebUI's built-in document upload. For more control, AnythingLLM gives you workspaces and better chunking. For full customization, Python with LangChain and ChromaDB takes about 30 lines of code. You need at least 8GB VRAM (or 16GB RAM for CPU) to run a 7B model plus embeddings comfortably.

📚 **More on this topic:** Open WebUI Setup Guide · Text Generation WebUI Guide · Best Models for Chat · VRAM Requirements

You've got a local LLM running. It answers general questions fine. But ask it about your company docs, your research notes, or a PDF you downloaded — and it hallucinates confidently. The model doesn't know your data because it was never trained on it.

RAG fixes this. Instead of retraining the model (expensive, slow, overkill), RAG searches your documents for relevant chunks and feeds them to the LLM as context. The model reads your actual text and answers based on it, not from memory.

The best part: everything stays on your machine. No API calls, no cloud storage, no wondering who's reading your files. This guide walks through three ways to set it up, from zero-config to fully custom.

What RAG Actually Is

RAG stands for Retrieval-Augmented Generation. Here's what happens when you ask a question:

1. **Your question gets converted to an embedding** — a numerical representation of its meaning
2. **The system searches your documents** for chunks that are semantically similar to your question
3. **The most relevant chunks get injected into the prompt** as context
4. **The LLM reads those chunks and generates an answer** based on your actual documents

It's not magic. It's a search engine feeding results to a language model. The LLM isn't "learning" your documents — it's reading them on the fly, every time you ask a question.

What RAG Is Not

- **Not fine-tuning.** Your model weights don't change. You can swap models anytime.
- **Not a database replacement.** RAG is for natural language questions, not structured queries. Don't use it to ask "how many invoices were over $10K" — use SQL for that.
- **Not perfect.** If your document chunking is bad, the retrieval is bad, and the answers are bad. Garbage in, garbage out.

RAG vs Fine-Tuning

	RAG	Fine-Tuning
Setup time	Minutes	Hours to days
Hardware needed	Same as inference	Much more (training)
Update documents	Just re-embed	Retrain the model
Best for	Answering questions about specific docs	Changing model behavior/style
Accuracy on your data	High if chunks are good	Variable, can overfit
Cost	Low	High

For most people reading this: you want RAG. Fine-tuning is for changing how a model behaves, not for teaching it facts.

Hardware Requirements

RAG adds two things on top of normal LLM inference: an embedding model and a vector database. The embedding model is the only one that matters for hardware.

Setup	VRAM/RAM Needed	What You Can Run
8GB VRAM	7-8B LLM (Q4) + embeddings	Good for personal use
12GB VRAM	13-14B LLM (Q4) + embeddings	Better quality answers
16GB VRAM	14B LLM (Q6) + embeddings	Sweet spot
24GB VRAM	32B LLM (Q4) + embeddings	Excellent quality

Setup	VRAM/RAM Needed	What You Can Run
CPU only (16GB RAM)	7B LLM (Q4) + embeddings	Works, slower
CPU only (32GB RAM)	13B LLM (Q4) + embeddings	Usable for smaller doc sets

The embedding model typically uses 200MB-1.2GB of VRAM depending on which one you pick. It runs alongside your LLM, so account for both. The vector database (ChromaDB, FAISS) uses minimal RAM — a few hundred MB even for thousands of documents.

Bottom line: If you can run a local LLM today, you can run RAG. The embedding model is tiny by comparison.

→ Use our Planning Tool to check exact VRAM for your setup.

The Easy Way: Open WebUI + Ollama

If you already have Ollama installed, this is the fastest path. Open WebUI has RAG built in — you upload documents and ask questions. No code, no config files.

Setup

1. Pull an embedding model:

```
ollama pull nomic-embed-text
```

2. Install Open WebUI:

```
# With pip (simplest)
pip install open-webui
open-webui serve

# Or with Docker
docker run -d -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -v open-webui:/app/backend/data \
  --name open-webui \
  ghcr.io/open-webui/open-webui:main
```

3. Pull a model good at RAG:

```
ollama pull llama3.1:8b
```

4. Upload documents: In the Open WebUI chat interface, click the `+` button and upload PDFs, text files, or markdown. Ask questions about them.

That's it. For a lot of people, this is all you need.

Critical Setting: Fix the Context Window

By default, Ollama sets `num_ctx` to 2048 tokens. That's way too small for RAG — your retrieved chunks plus the question will easily exceed that, causing the model to silently drop context.

Fix it in Open WebUI: go to **Settings → Models → your model → Parameters** and set `num_ctx` to at least `8192`. For longer documents, use `16384` or `32768` if your VRAM allows.

Or create a Modelfile:

```
FROM llama3.1:8b
PARAMETER num_ctx 16384
```

```
ollama create llama3.1-rag -f Modelfile
```

Open WebUI RAG Settings

Under **Settings → Documents**, you can tune:

- **Chunk size:** Default is 1500 characters. Good starting point. Decrease to 500-800 for precise Q&A, increase to 2000-3000 for longer-form answers.
- **Chunk overlap:** Default is 100 characters. Set to 15-20% of chunk size (so ~225 for 1500-char chunks). This prevents answers from being cut off mid-sentence.
- **Embedding model:** Select the model you pulled (`nomic-embed-text`).
- **Top K:** How many chunks to retrieve per question. Default 4 is fine. Increase to 6-8 for complex questions spanning multiple sections.

When Open WebUI Isn't Enough

Open WebUI's RAG works but has limitations:

- No workspace separation — all documents are in one pool
- Limited chunking strategies (character-based only)
- Can't mix different embedding models per collection
- No hybrid search (keyword + semantic)

If you hit these limits, move to AnythingLLM or the Python approach.

The Power User Way: AnythingLLM

AnythingLLM is a desktop app that gives you more control over RAG without writing code. The key feature: **workspaces**. You can have separate document collections for different projects, each with their own settings.

Setup

1. Download AnythingLLM from anythingllm.com. Available for Windows, Mac, and Linux.

2. Connect to Ollama: In AnythingLLM settings, set your LLM provider to Ollama and point it at `http://localhost:11434`.

3. Set your embedding model: Under Settings → Embedding, choose Ollama and select `nomic-embed-text`.

4. Create a workspace: Give it a name (e.g., "Research Papers" or "Work Docs").

5. Upload documents: Drag files into the workspace. AnythingLLM chunks and embeds them automatically.

6. Chat: Ask questions and the system retrieves relevant chunks from that workspace only.

Why Choose AnythingLLM Over Open WebUI

Feature	Open WebUI	AnythingLLM
Workspaces	No (all docs in one pool)	Yes (separate collections)
Supported file types	PDF, TXT, MD	PDF, TXT, MD, DOCX, CSV, and more

Feature	Open WebUI	AnythingLLM
Chunking control	Basic	More options
Vector DB options	Built-in	LanceDB, ChromaDB, Pinecone, others
Citation/sources	Basic	Shows which chunks were used
Setup difficulty	Easy	Easy
Also does chat	Yes (primary purpose)	Yes

AnythingLLM is the right choice if you want separate document collections for different projects, need to process many file types, or want to see exactly which chunks the model used to answer.

The Developer Way: Python + LangChain

If you want full control — custom chunking, preprocessing, filtering, or integration into your own tools — use Python. This is about 30 lines of working code.

Prerequisites

```
pip install langchain langchain-community langchain-ollama chromadb
```

Make sure Ollama is running with a model and embedding model pulled:

```
ollama pull llama3.1:8b
ollama pull nomic-embed-text
```

Minimal Working Example

```
from langchain_community.document_loaders import DirectoryLoader, TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_ollama import OllamaEmbeddings, ChatOllama
from langchain.chains import RetrievalQA
```

```
# 1. Load documents from a folder
loader = DirectoryLoader("./my_docs", glob="**/*.txt", loader_cls=TextLoader)
docs = loader.load()

# 2. Split into chunks
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
chunks = splitter.split_documents(docs)

# 3. Create embeddings and store in ChromaDB
embeddings = OllamaEmbeddings(model="nomic-embed-text")
vectorstore = Chroma.from_documents(chunks, embeddings, persist_directory="./chroma_db")

# 4. Create a retrieval chain
llm = ChatOllama(model="llama3.1:8b", num_ctx=8192)
qa = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(search_kwargs={"k": 4}),
    return_source_documents=True,
)

# 5. Ask questions
result = qa.invoke({"query": "What are the main findings?"})
print(result["result"])
for doc in result["source_documents"]:
    print(f"  Source: {doc.metadata['source']}")
```

Loading Different File Types

```
from langchain_community.document_loaders import PyPDFLoader, CSVLoader

# PDFs
loader = PyPDFLoader("report.pdf")

# CSVs
loader = CSVLoader("data.csv")

# Multiple file types from a directory
from langchain_community.document_loaders import DirectoryLoader
loader = DirectoryLoader("./docs", glob="**/*.pdf", loader_cls=PyPDFLoader)
```

Using FAISS Instead of ChromaDB

FAISS is faster for large collections (100K+ chunks):

```
pip install faiss-cpu  # or faiss-gpu if you have CUDA
```

```
from langchain_community.vectorstores import FAISS

vectorstore = FAISS.from_documents(chunks, embeddings)
vectorstore.save_local("./faiss_index")

# Load later
vectorstore = FAISS.load_local("./faiss_index", embeddings,
                                allow_dangerous_deserialization=True)
```

Why Go Custom

- **Preprocessing:** Strip headers, clean OCR output, normalize formatting before chunking
- **Hybrid search:** Combine semantic search with keyword (BM25) for better retrieval
- **Metadata filtering:** Only search documents from a specific date, author, or category
- **Pipeline integration:** Feed RAG into a larger application, API, or automation
- **Evaluation:** Measure retrieval quality and answer accuracy programmatically

---

Best LLMs for RAG

Not all models are equally good at RAG. The model needs to actually read and reason about the provided context, not just generate plausible-sounding text. Models that are good at instruction following and have decent context handling work best.

VRAM Budget	Model	Why It Works
8GB	Llama 3.1 8B (Q4)	Strong instruction following, good at citing context
8GB	Qwen 2.5 7B (Q4)	Excellent at structured answers from context
12GB	Qwen 2.5 14B (Q4)	Noticeable jump in answer quality
16GB	Qwen 2.5 14B (Q6)	Higher quant = better comprehension
24GB	Qwen 2.5 32B (Q4)	Best local RAG quality for most people
24GB	Mistral Small 24B (Q4)	Strong alternative, good at synthesis

VRAM Budget	Model	Why It Works
48GB+	Llama 3.1 70B (Q4)	Cloud-quality answers, needs dual GPUs or Mac

Avoid for RAG: Models that tend to ignore context and rely on training data. Older models (Llama 2, Mistral 7B v0.1) are noticeably worse at staying grounded in retrieved chunks.

Context Length Matters — But Not How You Think

Models advertise 128K context windows, but research consistently shows RAG performance peaks around 16K-32K tokens of actual context. Stuffing more chunks in doesn't help — it hurts. The model loses focus and starts hallucinating.

Practical rule: Retrieve 4-6 chunks of 1000-1500 characters each. That's roughly 2000-4000 tokens of context, well within any model's strong range. If you need more coverage, use better chunking and retrieval rather than cramming more text.

Best Embedding Models

The embedding model determines how well your system finds relevant chunks. A bad embedding model means the right information never reaches the LLM. This matters more than most people realize.

Model	Dimensions	Size	MTEB Score	Best For
`nomic-embed-text`	768	~270MB	62.4	General purpose, good default
`mxbai-embed-large`	1024	~670MB	64.7	Better quality, still reasonable size
`bge-m3`	1024	~1.2GB	68.2	Multilingual, highest quality
`all-minilm` (sentence-transformers)	384	~80MB	56.3	Minimal resources, CPU-friendly

Start with `nomic-embed-text`. It's small, fast, and good enough for most use cases. Move to `mxbai-embed-large` if you notice retrieval quality issues. Use `bge-m3` if you work with multiple languages.

Pull them with Ollama:

```
ollama pull nomic-embed-text
# or
ollama pull mxbai-embed-large
```

Important: Once you embed your documents with a specific model, you can't switch without re-embedding everything. The dimensions and vector space are model-specific. Pick one and stick with it, or plan to re-embed if you upgrade.

Tips for Better RAG Results

These are the most common mistakes and how to fix them.

1. Your Chunks Are Too Big or Too Small

Too big (3000+ chars): Multiple topics per chunk. The LLM gets irrelevant information mixed with relevant information and produces muddled answers.

Too small (200-300 chars): Individual sentences with no context. The LLM gets fragments that don't make sense on their own.

Sweet spot: 800-1500 characters with 15% overlap. Start at 1000 and adjust based on your documents. Technical docs with clear sections can go larger. Conversational text should go smaller.

2. Your Context Window Is Too Small

The single most common RAG failure. If `num_ctx` is 2048 (Ollama's default) and you're injecting 4 chunks of 1000 characters, you've already used most of your context before the model even starts generating. It silently drops the oldest context.

Set `num_ctx` to at least 8192. Check this first if your answers seem to ignore the documents.

3. The Model Is Ignoring Context

Some models are better at grounding answers in provided context than others. If the model keeps giving generic answers instead of citing your documents:

- Try a different model (Qwen 2.5 and Llama 3.1 are both strong here)

- Add explicit instructions in the system prompt: "Answer ONLY based on the provided context. If the context doesn't contain the answer, say so."
- Reduce the number of retrieved chunks — sometimes less is more

4. PDFs Are Messy

PDF text extraction is unreliable. Tables come out garbled. Headers and footers repeat on every page. Columns get merged.

Fixes:

- Use `PyMuPDF` (`fitz`) instead of basic PDF extractors — it handles layouts better
- Preprocess extracted text to remove headers/footers and fix formatting
- For scanned PDFs, run OCR first (Tesseract) and clean the output
- Consider converting PDFs to markdown before indexing

5. You're Not Seeing Sources

Always configure your RAG pipeline to return source documents alongside the answer. This lets you verify the answer and catch hallucinations. Open WebUI shows this by default. In LangChain, use `return_source_documents=True`.

6. Re-Embedding Is Slow

Embedding thousands of documents takes time on first run. After that, only embed new or changed documents. Both ChromaDB and FAISS support incremental updates. AnythingLLM handles this automatically.

What Hardware to Buy for RAG

If you're building or upgrading specifically for local RAG, here's what to prioritize:

Budget	GPU	What You Get
~$750	RTX 3090 (used)	24GB VRAM, runs 32B models + embeddings. Best value for RAG.
~$400	RTX 4060 Ti 16GB	16GB VRAM, runs 14B + embeddings. Adequate.
~$550	RTX 5060 Ti 16GB	16GB VRAM, newer architecture. See our 16GB guide.

Budget	GPU	What You Get
~$2000	RTX 4090 / 5090	24-32GB VRAM, fastest inference.
~$2500+	Mac Studio M4 Max (128GB)	128GB unified memory, runs 70B+ models. Mac vs PC comparison.

For most RAG use cases, an RTX 3090 for ~$750 is hard to beat. 24GB VRAM comfortably runs a 32B model plus embeddings — that's genuinely excellent RAG quality, and it's the cheapest way to get 24GB.

If you're CPU-only, RAG still works — it's just slower. A 7B model on CPU with 16-32GB RAM handles personal document collections fine. Speed is the tradeoff, not quality.

Getting Started Today

Here's the fastest path from zero to working RAG:

1. Install Ollama if you haven't already
2. Pull a model: `ollama pull llama3.1:8b`
3. Pull an embedding model: `ollama pull nomic-embed-text`
4. Install Open WebUI: `pip install open-webui && open-webui serve`
5. Fix context window: set `num_ctx` to 8192+ in model settings
6. Upload a document and ask it a question

You'll have private, local document search running within minutes. No API keys, no subscriptions, no data leaving your machine.

If you outgrow Open WebUI, AnythingLLM gives you workspaces and better organization. If you outgrow that, Python and LangChain give you unlimited control. The beauty of local RAG: you own the whole stack, so you can swap any piece at any time.

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/local-rag-search-documents-private-ai/

Free guides for running AI locally