RAG Pipeline for Local AI: A Practical Guide to Retrieval-Augmented Generation

March 6, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** RAG lets your local LLM answer questions using YOUR documents instead of its training data. The fastest path: install Ollama + Open WebUI, drag in your PDFs, and start asking questions (15 minutes, zero code). The developer path: Python + LangChain + ChromaDB + Ollama, ~50 lines of code. Use nomic-embed-text for embeddings (274MB, runs on CPU), Qwen 3.5 9B for generation (needs 6GB+ VRAM). Chunk your docs at 512 tokens with 50-token overlap. The number one reason RAG fails: your chunks are too big or split in the wrong places.

Related: Best LLMs for RAG | Embedding Models for RAG | Ollama Troubleshooting | VRAM Requirements

Your local LLM knows a lot about the world in general and nothing about your documents. Ask it about your company handbook, your research notes, or a contract you downloaded, and it'll either admit ignorance or confidently make something up.

RAG fixes this without retraining anything. You build a pipeline that searches your documents, grabs the relevant pieces, and hands them to the LLM as context. The model reads your actual text and answers from it. Everything stays on your machine — no API calls, no cloud storage, no one reading your files.

Most RAG guides assume you're using OpenAI's API. This one assumes you're running Ollama on your own hardware. Every component is local, every tool is free, and the whole thing works offline.

RAG in plain English

Here's what happens when you type a question into a RAG system:

1. Your question gets converted to a vector (a list of numbers that represents its meaning)
2. That vector gets compared against vectors of your document chunks
3. The most similar chunks get pulled out — usually 3 to 5 of them
4. Those chunks get stuffed into the prompt alongside your question
5. The LLM reads the chunks and writes an answer based on them

That's it. RAG is a search engine bolted onto a language model. The "retrieval" is the search, the "augmented generation" is the LLM answering with the search results as context. There's no learning, no fine-tuning, no weight updates. Swap the model tomorrow and your documents still work.

The local RAG stack, piece by piece

A RAG pipeline has four components. You need all four, and each one has a few options worth knowing about.

1. Document loading

You need to get your files into text. PDFs, markdown, plain text, code files, HTML — whatever you have. This step is boring but matters. A bad PDF parser drops tables, misreads columns, or loses formatting. That garbage propagates through your entire pipeline.

For most setups, LangChain's document loaders or LlamaIndex's readers handle this. They support dozens of formats out of the box. The common ones:

- **PDF**: `PyPDFLoader` (LangChain) or `SimpleDirectoryReader` (LlamaIndex). For complex PDFs with tables, consider `unstructured` or `pymupdf` instead — they preserve layout better.
- **Markdown/text/code**: Straightforward. Any loader works.
- **HTML**: Strip the boilerplate. `BeautifulSoup` or `unstructured` both handle this.
- **Word/DOCX**: `python-docx` via LangChain's `Docx2txtLoader`.

If you're using Open WebUI, document loading is built in. Drag a file into a chat, and it handles parsing automatically.

2. Chunking

This is where most RAG pipelines quietly fail. You have to split your documents into smaller pieces because embedding models have limited context and because smaller, focused chunks retrieve better than huge ones.

Three strategies, in order of what I'd actually use:

Recursive character splitting is the default for a reason. It tries to split on paragraph breaks first, then sentences, then words, then characters. You set a chunk size (in tokens or characters) and

an overlap. Start with 512 tokens and 50 tokens of overlap. This works well for 80% of documents.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=50,
    separators=["\n\n", "\n", ". ", " ", ""]
)
chunks = splitter.split_documents(documents)
```

Semantic chunking groups text by meaning instead of fixed size. It embeds each sentence, then splits where the embedding similarity drops — a topic boundary. Better results, but slower and more complex. LlamaIndex has a `SemanticSplitterNodeParser` for this. Use it when recursive splitting gives you chunks that mix unrelated topics.

Fixed-size splitting just cuts every N characters. Fast, dumb, and it will split mid-sentence. Only use this if you're processing millions of documents and need speed over quality.

Here's what actually matters: **chunk size and overlap**. Too big (2000+ tokens) and your chunks contain multiple topics, so retrieval returns semi-relevant noise. Too small (under 100 tokens) and you lose context — a sentence about "it" with no antecedent is useless. 512 tokens with 50-token overlap is a good starting point. Tune from there based on your documents.

3. Embedding models

The embedding model converts your chunks (and your queries) into vectors. Two texts with similar meaning produce similar vectors, which is how retrieval finds the right chunks.

You run this locally through Ollama. The embedding model is separate from your chat model and much smaller — it runs on CPU without slowing anything down.

Model	Size	Dimensions	Context	Best for
nomic-embed-text	274MB	768	8K tokens	Default pick. Good quality, tiny footprint.
all-minilm	46MB	384	512 tokens	Minimal hardware. Fine for small collections.
mxbai-embed-large	670MB	1024	512 tokens	Higher quality embeddings, short context.

Model	Size	Dimensions	Context	Best for
qwen3-embedding: 0.6b	1.2GB	2048	32K tokens	Best benchmark scores (MTEB 70.7). Newest option.
bge-m3	1.2GB	1024	8K tokens	Multilingual documents (100+ languages).

My recommendation: Start with nomic-embed-text. It's 274MB, runs on CPU, supports 8K token context (enough for any reasonable chunk size), and produces solid retrieval quality. Pull it with:

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

Switch to qwen3-embedding if you need better accuracy and don't mind the 1.2GB download. Switch to bge-m3 if your documents aren't in English.

One thing people get wrong: the embedding model you use for indexing must be the same one you use for queries. If you re-embed with a different model, you have to re-index everything. Pick one and stick with it.

4. Vector store

Vectors need to go somewhere searchable. Three options dominate the local scene:

Store	Stars	Language	Best for	Setup
ChromaDB 1.5	26.5K	Python/ Rust	Getting started. Simplest API.	`pip install chromadb`
FAISS 1.14	39.3K	C++/ Python	Speed at scale. Millions of vectors.	`pip install faiss-cpu`
Qdrant 1.17	29.3K	Rust	Production features. Filtering, payloads.	Docker or `pip install qdrant-client`

ChromaDB if you want to build something this weekend. It's an in-process database — no server to run, no Docker needed. Create a collection, add documents, query it. Five lines of code. It persists to disk automatically.

FAISS if you have millions of chunks and need sub-millisecond search. Facebook built it for billion-scale similarity search. No metadata filtering though — it's a vector index, not a database. You manage document metadata yourself.

Qdrant if you're building something that needs to run in production. It has payload filtering (search only documents from a specific folder, date range, or author), snapshots, and a REST API. Runs as a Docker container or embedded in Python.

For this guide, I'm using ChromaDB. If you outgrow it, migrating to Qdrant takes an afternoon.

The 15-minute setup: Open WebUI

If you want RAG working today with zero code, this is the path. Open WebUI (126K GitHub stars) has RAG built in. You upload documents, it chunks them, embeds them, and lets you chat against them.

Prerequisites

- Ollama installed and running
- Docker installed (for Open WebUI)
- 8GB+ VRAM or 16GB+ RAM for CPU inference

Steps

```
# Pull your models
ollama pull qwen3.5:9b
ollama pull nomic-embed-text

# Run Open WebUI
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
```

Open `http://localhost:3000`, create an account (local only), go to Settings > Documents, and verify the embedding model is set to nomic-embed-text.

Now start a chat, click the + button, upload a PDF or text file, and ask questions about it. Open WebUI handles chunking, embedding, retrieval, and generation. The defaults are reasonable — 512-token chunks, top-5 retrieval.

That's it. If this is enough for your use case, stop here. The developer setup below gives you more control but takes more work.

The developer setup: Python + LangChain + ChromaDB

This is for when you need custom chunking, multiple document sources, or want to integrate RAG into your own application. About 50 lines of Python.

Prerequisites

- Python 3.10+
- Ollama running with `qwen3.5:9b` and `nomic-embed-text` pulled
- 8GB+ VRAM

Install dependencies

```
pip install langchain langchain-ollama langchain-chroma chromadb pypdf
```

The pipeline

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

# 1. Load documents
loader = DirectoryLoader("./docs", glob="**/*.pdf", loader_cls=PyPDFLoader)
documents = loader.load()
print(f"Loaded {len(documents)} pages")

# 2. Chunk
splitter = RecursiveCharacterTextSplitter(chunk_size=512, chunk_overlap=50)
chunks = splitter.split_documents(documents)
print(f"Split into {len(chunks)} chunks")

# 3. Embed and store
embeddings = OllamaEmbeddings(model="nomic-embed-text")
```

```
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"
)

# 4. Query
llm = ChatOllama(model="qwen3.5:9b", temperature=0)
qa = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(search_kwargs={"k": 5}),
    return_source_documents=True,
)

result = qa.invoke({"query": "What does the refund policy say?"})
print(result["result"])

# Show which chunks were used
for doc in result["source_documents"]:
    print(f"  Source: {doc.metadata['source']}, page {doc.metadata.get('page', '?')}")
```

Put your PDFs in a `./docs` folder and run it. The first run embeds everything (takes a minute for a few hundred pages — embedding is fast). Subsequent runs load from the ChromaDB directory on disk.

LlamaIndex alternative

If you prefer LlamaIndex (0.14.15, 47.4K stars), the equivalent is shorter:

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader, Settings
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.ollama import OllamaEmbedding

Settings.llm = Ollama(model="qwen3.5:9b", request_timeout=120)
Settings.embed_model = OllamaEmbedding(model_name="nomic-embed-text")

documents = SimpleDirectoryReader("./docs").load_data()
index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine(similarity_top_k=5)

response = query_engine.query("What does the refund policy say?")
print(response)
```

LlamaIndex is more opinionated — it picks defaults for chunking, retrieval, and prompting. LangChain gives you more knobs. Neither is wrong. Pick whichever reads better to you and move on.

What actually makes RAG fail

RAG looks simple in demos and breaks in production. Here are the failure modes I've hit, in order of how often they happen.

Bad chunking

The most common problem. Your chunks are 2000 tokens, spanning multiple topics. The retriever returns a chunk that's 30% relevant and 70% noise. The model either picks up the noise or gets confused.

Fix: smaller chunks (512 tokens), overlap (50 tokens), and consider semantic chunking for documents where topics shift frequently.

Wrong embedding model for the domain

General embedding models work for general text. But if your documents are full of legal jargon, medical terminology, or code, a general model may not capture the domain-specific similarity well. "Breach of fiduciary duty" and "violation of trust obligations" should be close in vector space. A general model might not see them that way.

Fix: test retrieval quality before blaming the chat model. Run a query, look at the retrieved chunks, and ask yourself: are these the right chunks? If not, try a larger embedding model (qwen3-embedding) or a domain-specific one.

Context window stuffing

You retrieve 20 chunks and shove them all into the prompt. The model can technically handle it — Qwen 3.5 9B has 262K context — but models lose accuracy in the middle of long contexts (the "lost in the middle" problem, documented in a 2023 Stanford/Berkeley paper that keeps getting re-confirmed). The first and last chunks get attention. The middle gets skimmed.

Fix: retrieve 3-5 chunks, not 20. If you need more context, use a reranker to sort chunks by relevance and take only the top results. Quality over quantity.

The model ignores your documents

You retrieve the right chunks, inject them into the prompt, and the model answers from its training data anyway. This happens most often with smaller models (under 7B) that are weak at instruction following.

Fix: use a model that respects system prompts. Qwen 3.5 9B is good at this. Command R 35B is specifically trained for grounded generation. Also, set temperature to 0 — you want deterministic answers from documents, not creative riffs.

Stale index

You update your documents but forget to re-embed the changed files. The index still has the old content. This is obvious in hindsight but easy to miss when you're adding files to a directory over time.

Fix: track file modification times and re-embed changed files. Or use Open WebUI, which handles this automatically when you re-upload.

Hardware reality

RAG has two compute phases, and they have different hardware needs:

Embedding is cheap. nomic-embed-text runs on CPU at hundreds of chunks per second. Even a 1.2GB embedding model runs fine on CPU. You don't need a GPU for this part. 16GB of system RAM is enough.

Generation needs a GPU (or Apple Silicon with unified memory). The chat model is where your VRAM goes. Qwen 3.5 9B needs about 5-6GB at Q4. On CPU-only setups, expect 2-5 tok/s with a 7B model and 16GB RAM — usable but slow.

Setup	Embedding	Chat model	Experience
8GB VRAM (RTX 3060)	CPU, fast	Qwen 3.5 9B (Q4)	Good. 30-40 tok/s generation.
16GB VRAM (RTX 4060 Ti)	CPU, fast	Qwen 3.5 9B (Q6-Q8)	Better quality, more context headroom.
24GB VRAM (RTX 3090/4090)	CPU, fast	Qwen 3.5 27B (Q4)	72.4% SWE-bench. As good as local RAG gets on one GPU.
	CPU, fast		

Setup	Embedding	Chat model	Experience
CPU-only (16GB+ RAM)		Qwen 3.5 9B (Q4, CPU)	Slow generation (2-5 tok/s). Works for patience.
Mac M-series (24GB+)	CPU, fast	Qwen 3.5 9B via Ollama on Mac	25-35 tok/s. Solid experience.

When RAG is the wrong answer

RAG solves a specific problem: querying large or changing document collections with natural language. It's not always the right tool.

Small document sets — if your entire knowledge base fits in one prompt (under 50K tokens), just paste it into the context window. No embedding, no retrieval, no chunking, no failure modes. Qwen 3.5 9B has 262K context. A single 100-page document is roughly 30K tokens. Just paste it.

Structured data — if you need to ask "how many invoices were over $10K last quarter," that's a SQL query, not RAG. RAG is for natural language over unstructured text.

Real-time data — RAG searches a static index. If you need current stock prices, live documentation, or today's news, use a web search tool or an API.

Math and calculations — "what's the total from column B" will fail. The model can't reliably do arithmetic on retrieved text. Extract the data and compute it properly.

The bottom line

The fastest path to local RAG: Ollama + Open WebUI + a PDF. Fifteen minutes, zero code, and you're chatting with your documents privately.

The developer path: Python + LangChain (or LlamaIndex) + ChromaDB + Ollama. More control, custom chunking, and you can embed it in your own application. About an hour to get working, including fumbling with dependencies.

Either way, the model pair that works: nomic-embed-text for embeddings, Qwen 3.5 9B for generation. Pull both and you're set:

```
ollama pull nomic-embed-text
ollama pull qwen3.5:9b
```

If your answers are bad, check your chunks first. Retrieval quality is the bottleneck in almost every RAG pipeline, and chunking is the bottleneck in retrieval. Get that right and the rest falls into place.

Get notified when we publish new guides.

Subscribe — free, no spam
