Lesson 1 · Build a RAG System From Scratch

PDF: this lesson · **Install (Linux · macOS · Windows):** guide · PDF

> **Part of local-ai-lab** — a hands-on course for building local AI.

> **Interactive version (slides):** https://nikolareljin.github.io/local-ai-lab/lesson-1-rag.html
> **Course home:** https://nikolareljin.github.io/local-ai-lab/ **Source:** https://github.com/nikolareljin/local-ai-lab **Author:** Nik Reljin

> **Lessons: 1 · RAG (you are here)** → 2 · MCP → 3 · LangChain → 4 · LangGraph → 5 · Ollama tools → 6 · Semantic Kernel → 7 · Bedrock Agents → 8 · Google ADK

What you'll learn

Retrieval-Augmented Generation (RAG) makes a language model answer questions about **your** documents instead of guessing from its training data. In this lesson you build a complete, working RAG app — drag in a PDF, ask a question, get an answer grounded in the document **with citations** — in a few hundred lines of readable Python, no heavyweight frameworks.

By the end you will understand every stage of the pipeline:

```
documents/ ⟶ extract ⟶ chunk ⟶ index ⟶ retrieve ⟶ ground ⟶ answer + sources
```

- **What** RAG is and **why** it beats a bare LLM for private data
- **Extraction** — turning PDF/DOCX/TXT/MD into clean, citable text
- **Chunking** — splitting text so retrieval is precise and context survives
- **Indexing** — caching so you don't re-process unchanged files
- **Retrieval** — two techniques side by side: **lexical BM25** and **semantic embeddings**
- **Grounding** — the anti-hallucination prompt that cites sources and admits ignorance
- **Providers** — one interface, four back ends (Claude Code, Ollama, Gemini, OpenAI)
- A real **drag-and-drop web UI** over the same engine

> **Why from scratch?** Most tutorials teach you to glue frameworks together. Here you write the chunker, the retriever, the grounding prompt, and the provider abstraction yourself. Once you understand the primitives, every framework (LangChain, LlamaIndex, LangGraph) becomes obvious, because you already know what it automates.

Concept: why RAG?

A language model only knows what it saw during training. Ask it about your company's internal manual and it will either refuse or — worse — **hallucinate** a confident, wrong answer. RAG fixes this with a simple idea:

1. **Retrieve** the passages from your documents that are most relevant to the question.
2. **Augment** the prompt by pasting those passages in as context.
3. **Generate** an answer that is instructed to use **only** that context and to cite it.

The model is no longer guessing from memory; it is reading your text and summarizing it. That is the difference between a party trick and something you can trust.

Prerequisites

```
python -m venv venv && source venv/bin/activate
pip install -r requirements.txt          # pypdf python-docx rank-bm25 numpy requests python-dotenv flask
```

The simplest AI to answer questions is the **Claude Code CLI** — if you can run `claude` in your terminal, you're ready, no API key. Ollama / Gemini / OpenAI are wired in at Step 6.

Step 1 · Extract text from documents

RAG starts by turning files into plain text. Different formats need different libraries, so we dispatch on the file extension and normalize everything to a list of **pages** — each carrying its text, a page number, and the source filename. We need those last two for citations.

`localrag/extract.py`

```
from pathlib import Path
from typing import List, TypedDict

SUPPORTED_EXTS = {".pdf", ".docx", ".txt", ".md", ".markdown"}


class Page(TypedDict):
   source: str
   page_number: int
   text: str


def _extract_pdf(path: Path) -> List[Page]:
   from pypdf import PdfReader
   reader = PdfReader(str(path))
   pages = []
   for i, page in enumerate(reader.pages, start=1):
       text = (page.extract_text() or "").strip()
       if text:
           pages.append(Page(source=path.name, page_number=i, text=text))
   return pages


def _extract_docx(path: Path) -> List[Page]:
   from docx import Document
   doc = Document(str(path))
   text = "\n".join(p.text for p in doc.paragraphs if p.text.strip())
   return [Page(source=path.name, page_number=1, text=text)] if text.strip() else []


def extract_pages(path: Path) -> List[Page]:
   ext = path.suffix.lower()
   if ext == ".pdf":
       return _extract_pdf(path)
   if ext == ".docx":
       return _extract_docx(path)
```

```
    if ext in {".txt", ".md", ".markdown"}:
        text = path.read_text(encoding="utf-8", errors="replace").strip()
        return [Page(source=path.name, page_number=1, text=text)] if text else []
    return []
```

Why pages? PDFs have real pages, so a citation like `manual.pdf:4` points the reader to the exact spot. Formats without pages (DOCX/TXT/MD) collapse to a single page — still citable by name.

> **Teaching point.** Extraction is the unglamorous 80% of real RAG. Garbage text in → garbage answers out. Scanned PDFs need OCR; tables and multi-column layouts need smarter parsers. We keep it simple here, but this is where production systems spend most of their effort.

Step 2 · Split text into overlapping chunks

A whole document is too big to feed a model and too coarse to retrieve precisely. We split it into **chunks** of ~1000 characters. The key trick is **overlap**: each chunk repeats the last ~200 characters of the previous one, so a sentence split across a boundary still appears intact in at least one chunk. We break on sentence boundaries when we can, and carry the source + page along.

`localrag/chunk.py`

```
def _split_text(text: str, size: int, overlap: int) -> list[str]:
    text = " ".join(text.split())              # normalize whitespace
    if len(text) <= size:
        return [text] if text else []
    chunks, start, n = [], 0, len(text)
    while start < n:
        end = min(start + size, n)
        if end < n:                            # prefer a clean break near the limit
            window = text[start:end]
            for sep in (". ", "! ", "? ", "\n", " "):
                pos = window.rfind(sep)
                if pos > size // 2:
                    end = start + pos + len(sep)
                    break
        chunk = text[start:end].strip()
        if chunk:
            chunks.append(chunk)
        if end >= n:
            break
        start = max(end - overlap, start + 1)   # step back to create overlap
    return chunks


def chunk_pages(pages, size=1000, overlap=200):
    chunks, index = [], 0
    for page in pages:
        for piece in _split_text(page["text"], size, overlap):
            chunks.append({
                "source": page["source"],
                "page_number": page["page_number"],
                "chunk_index": index,
                "text": piece,
            })
            index += 1
    return chunks
```

> **Teaching point — chunk size is a dial.** Too large and retrieval is imprecise (you pull in irrelevant text); too small and you lose context (the answer is split across chunks). 800–1200 chars with 10–20% overlap is a sane default. Tuning this per corpus is half the art of RAG.

Step 3 · Index and cache

Now extract + chunk every file in `documents/` and cache the result so we don't redo the work on every question. We fingerprint each file by `(path, mtime, size)`; if nothing changed, we reuse the cache. **This is the "drop a file and ask again" loop** — new files are picked up automatically.

`localrag/store.py` (core)

```
def is_stale(config) -> bool:
    """True if the cache is missing or the documents folder changed."""
    index_path = config.cache_dir / "index.json"
    if not index_path.exists():
        return True
    data = json.loads(index_path.read_text())
    return data.get("fingerprint") != _fingerprint(discover_files(config.docs_dir))


def build_index(config):
    files = discover_files(config.docs_dir)
    chunks = []
    for path in files:
        chunks.extend(chunk_pages(extract_pages(path)))
    config.cache_dir.mkdir(parents=True, exist_ok=True)
    (config.cache_dir / "index.json").write_text(
        json.dumps({"fingerprint": _fingerprint(files), "chunks": chunks}))
    return chunks, len(files)
```

The index is just a JSON file. No database, no vector store to run — perfect for a local demo and trivial to inspect (`cat .localrag/index.json`).

Step 4 · Retrieve with BM25 (the zero-setup default)

Given a question, which chunks are relevant? The simplest robust answer is **BM25**, a classic keyword-ranking algorithm (a smarter TF-IDF). It needs no model and no embedding service, so it works with **any** provider — including Claude Code, which can't produce embeddings.

`localrag/retriever.py` (BM25)

```
import re
from rank_bm25 import BM25Okapi


def _tokenize(text: str) -> list[str]:
    return re.findall(r"[a-z0-9]+", text.lower())


class Bm25Retriever:
    name = "bm25"

    def __init__(self, chunks):
        self.chunks = chunks
```

```
        self.bm25 = BM25Okapi([_tokenize(c["text"]) for c in chunks] or [[""]])

    def search(self, query, k):
        if not self.chunks:
            return []
        scores = self.bm25.get_scores(_tokenize(query))
        ranked = sorted(range(len(self.chunks)), key=lambda i: scores[i], reverse=True)
        return [self.chunks[i] for i in ranked[:k]]
```

> **A real bug worth knowing.** BM25's IDF term goes **negative** when a word appears in every chunk — which happens constantly on a tiny corpus. An early version of this code filtered results with `score > 0` and returned **nothing**, because on a 2-chunk corpus every score was negative. The fix: don't apply an absolute score cutoff — return the top-k by rank and let the grounding prompt judge relevance. **Retrieval retrieves; the LLM decides.** This kind of small, non-obvious failure is exactly why building it yourself is worth it.

Step 5 · The grounding prompt (anti-hallucination)

This is the heart of RAG. We hand the model the retrieved chunks as **context** and instruct it to answer **from that context**, cite sources, and clearly mark anything it adds from general knowledge. This is what stops the model from confidently making things up.

`localrag/prompts.py`

```
SYSTEM_PROMPT = """You are a careful assistant answering questions over a set of \
the user's own documents. Follow these rules exactly:

1. Answer from the DOCUMENT CONTEXT below FIRST. For every claim that comes from \
the documents, cite the source like [filename:page].
2. If the answer is not contained in the document context, say so plainly: \
"This is not covered in your documents." You may then add general knowledge, but \
you MUST prefix it with "(general knowledge — not from your documents)".
3. Never invent document contents, quotes, or citations. Only cite sources that \
appear in the context.
4. Be concise. Prefer the documents' own wording.
"""


def build_context(chunks):
    return "\n\n---\n\n".join(
        f"[{c['source']}:{c['page_number']}]\n{c['text']}" for c in chunks)


def build_user_prompt(question, chunks):
    context = build_context(chunks) if chunks else "(no relevant documents found)"
    return f"DOCUMENT CONTEXT:\n{context}\n\nQUESTION:\n{question}"
```

> **Teaching point.** RAG quality is **retrieval** quality + **prompt** quality. A perfect retriever with a sloppy prompt still hallucinates; a strict prompt with bad retrieval answers "not in your documents" to everything. You need both. The rules above — cite, admit ignorance, label general knowledge — are the minimum viable anti-hallucination contract.

Step 6 · A provider abstraction (Claude / Ollama / Gemini / OpenAI)

We don't want the pipeline to care **which** AI answers. So we define one tiny interface and four implementations. Switching providers becomes a one-line env-var change.

`localrag/providers/__init__.py`

```
from typing import Protocol


class LLMProvider(Protocol):
   name: str
   def is_available(self) -> bool: ...
   def chat(self, system: str, user: str) -> str: ...


def get_provider(name, config):
   if name == "claude":
       from .claude_code import ClaudeCodeProvider; return ClaudeCodeProvider(config)
   if name == "ollama":
       from .ollama import OllamaProvider;        return OllamaProvider(config)
   if name == "gemini":
       from .gemini import GeminiProvider;        return GeminiProvider(config)
   if name == "openai":
       from .openai import OpenAIProvider;        return OpenAIProvider(config)
   raise ValueError(f"Unknown provider '{name}'")
```

The **default provider shells out to the Claude Code CLI** — no API key, it reuses your existing login:

`localrag/providers/claude_code.py`

```
import shutil, subprocess


class ClaudeCodeProvider:
   name = "claude"

   def __init__(self, config):
       self.bin = config.claude_bin

   def is_available(self):
       return shutil.which(self.bin) is not None

   def chat(self, system, user):
       result = subprocess.run(
           [self.bin, "-p", f"{system}\n\n{user}"],
           capture_output=True, text=True, timeout=180)
       return result.stdout.strip()
```

Ollama, Gemini, and OpenAI are equally small REST clients (`POST /api/chat`, `generateContent`, `/chat/completions`). Each `chat()` takes the same `(system, user)` and returns a string. That uniformity is the whole point.

> **Teaching point.** This is the pattern every "LLM framework" is built around — a provider interface plus adapters. Once you've written it by hand, LangChain's `ChatModel` and friends stop looking like magic. (You'll see exactly that in Lesson 3.)

Step 7 · Wire it together (the CLI)

Now the pipeline: ensure the index is fresh, retrieve top-k chunks, build the grounded prompt, call the provider, print the answer and its sources.

```
def answer(question, retriever, config):
   hits = retriever.search(question, config.top_k)
   provider = get_provider(config.provider, config)
   reply = provider.chat(SYSTEM_PROMPT, build_user_prompt(question, hits))
   print(reply)
   sources = []
   for h in hits:
       tag = f"{h['source']}:{h['page_number']}"
       if tag not in sources:
           sources.append(tag)
   print("Sources:", ", ".join(sources) or "(none)")
```

Run it:

```
python -m localrag index                      # build the index from documents/
python -m localrag ask "How do I reset the device?"
```

```
To reset the WidgetPro 3000, press and hold the power button for 10 seconds
until the status LED blinks blue three times. [sample_manual.md:1]

Sources: sample_manual.md:1
```

That's a **complete RAG system**. Everything below is upgrades.

Step 8 · Upgrade: semantic retrieval with embeddings

BM25 matches **words**. "How do I power-cycle it?" won't match a doc that says "restart" — no shared keywords. **Embeddings** fix this by matching **meaning**: we turn each chunk into a vector and rank by cosine similarity to the question's vector.

`localrag/retriever.py` (embeddings)

```
class EmbeddingRetriever:
   name = "embeddings"

   def __init__(self, chunks, config):
       import numpy as np
       from .providers import embed_texts
       self.chunks, self.config = chunks, config
       vectors = np.asarray(
           embed_texts(config.embed_provider, config, [c["text"] for c in chunks]),
           dtype="float32")
       self._vectors = self._normalize(vectors)

   def search(self, query, k):
       import numpy as np
       from .providers import embed_texts
       q = self._normalize(np.asarray(
           embed_texts(self.config.embed_provider, self.config, [query]),
           dtype="float32"))[0]
       sims = self._vectors @ q                  # cosine (vectors are normalized)
       ranked = np.argsort(sims)[::-1][:k]
       return [self.chunks[i] for i in ranked]
```

Claude Code can't embed, so embeddings use `RAG_EMBED_PROVIDER=ollama|gemini|openai`. And because a demo should never dead-end, the selector **falls back to BM25** with a clear message if no embed provider is reachable:

```
def build_retriever(chunks, config):
   if config.retriever == "embeddings":
       try:
           return EmbeddingRetriever(chunks, config)
       except Exception as exc:
           print(f"[localrag] Embeddings unavailable ({exc}). Falling back to BM25.")
   return Bm25Retriever(chunks)
```

```
RAG_RETRIEVER=embeddings RAG_EMBED_PROVIDER=ollama python -m localrag ask "power-cycle steps?"
```

> **Teaching point — BM25 vs embeddings.** BM25 is free, instant, and great for exact terms, names, and codes. Embeddings catch paraphrases and concepts but cost an embed call and a vector store. Production systems often run **both** (hybrid retrieval) and merge the rankings. You now have both — try the same question each way and watch the difference.

Step 9 · A drag-and-drop web UI

A terminal is fine for you; a web page is better for a demo. We add a tiny Flask app that reuses the **exact same** engine. Three endpoints: serve the page, accept dropped files (save + reindex), and answer questions.

`localrag/web.py` (essentials)

```
@app.post("/api/upload")
def upload():
   for f in request.files.getlist("files"):
       name = secure_filename(f.filename)
       if Path(name).suffix.lower() in SUPPORTED_EXTS:
           f.save(base_config.docs_dir / name)
   chunks, n = refresh_index(base_config)         # rebuild on every drop
   return jsonify({"files": _list_files(base_config), "chunks": len(chunks)})


@app.post("/api/ask")
def ask():
   data = request.get_json()
   return jsonify(answer_question(_request_config(), data["question"]))
```

The front end is one HTML file with vanilla JS: a dropzone using the browser's drag-and-drop events, a `fetch()` to `/api/upload`, then a question box that `POST`s to `/api/ask` and renders the answer plus clickable source chips. Launch it:

```
python -m localrag web      # http://127.0.0.1:5000
```

Drop a PDF, ask a question, watch it cite the file you just dropped. Same pipeline, nicer surface.

Step 10 · See the anti-hallucination work

Ask something **in** your documents and something **out of** them, and watch the model stay honest:

```
python -m localrag ask "How long is the warranty?"
# → 24 months from purchase date [warranty.txt:1]          ← grounded, cited

python -m localrag ask "What is the capital of France?"
# → This is not covered in your documents.
#   (general knowledge — not from your documents) The capital of France is Paris.
```

That clean separation — cited facts vs. clearly-labeled general knowledge — is the payoff of the grounding prompt. It's what makes RAG trustworthy.

A tiny **offline test** locks in the retrieval core (no network, no LLM):

```
def test_bm25_finds_reset_instructions():
   chunks = chunk_pages(extract_pages(SAMPLE), size=400, overlap=80)
   hits = Bm25Retriever(chunks).search("how do I reset the device", k=3)
   assert "power button" in hits[0]["text"].lower()
```

```
pytest -q      # 4 passed
```

Try it yourself — verify RAG on a brand-new document

The surest way to prove a RAG system actually **reads your files** (instead of reciting training data) is to feed it something no model has ever seen. Download this short **fictional** story and watch the app answer questions about it — with citations.

Download: The_Magic_Turtle_Astronaut.pdf — it's also in the repo at `docs/pdf/The_Magic_Turtle_Astronaut.pdf`.

It's a made-up legend, **"The Voyage of Caretta the Magnificent,"** so no language model could know its details unless it read the file.

1. Start the app: `./run -l 1` (or `python -m localrag web`) and open the page.
2. **Drag the PDF onto the dropzone** (or click to browse) — it indexes automatically.
3. Ask away — and always check the `Sources:` line.

Questions the story can answer (grounded — the answer should cite the file):

- What was the name of the magic turtle, and what species was she? → **Caretta; species Chelonia mythica** `[…:2]`
- Who discovered the turtle's secret, and how? → **Dr. Yuki Tanaka — her shell glowed under UV light and she registered no age** `[…:2]`
- What was the spaceship called, and how long did the journey to Alpha Centauri take? → **the Ocean's Memory; twelve years** `[…:3]`
- What planet did Caretta discover, and which star does it orbit? → **Nuevo Edén, orbiting Alpha Centauri B** `[…:3]`
- On what date was the habitable planet discovered? → **2 May 2126** `[…:3]`
- How did the turtle save the mission when the cooling line was damaged? → **she sensed the wrongness through her shell and woke the engineer, Commander Adaeze Okafor, in time to seal the breach** `[…:3]`

- Where did Caretta choose to live after returning to Earth? → **the tide pools of the Galápagos** `[…:4]`

Questions that are NOT in the story — these should trigger the honest **"This is not covered in your documents"** response. **This is the important test**: the anti-hallucination prompt staying honest instead of inventing an answer.

- How much did the spaceship cost to build?
- What did the turtle eat during the twelve-year voyage?
- Who was the President of Earth when the mission launched?

> **One nice touch for a talk:** ask **"What is the nearest star system to the Sun?"** The story states Alpha Centauri is ~4.25 light-years away, so the app answers **from the document, with a citation**, even though it's also general knowledge — a clean way to show retrieval preferring **your** text.

> **Why this works:** the story is fiction, so a bare LLM would either refuse or invent details. Right names, ship, planet, and dates — each with a `[file:page]` citation — prove the answer came from **your uploaded file**, not the model's memory. That's RAG doing its job.

Prefer the terminal? Drop the file into `documents/` and ask:

```
cp ~/Downloads/The_Magic_Turtle_Astronaut.pdf documents/
python -m localrag ask "What planet did Caretta discover, and which star does it orbit?"
```

Recap

You built, from scratch:

Stage	What it does
Extraction	PDF/DOCX/TXT/MD → pages
Chunking	overlapping windows with source+page metadata
Indexing	cached, auto-refreshing JSON index
Retrieval	BM25 **and** semantic embeddings behind one interface
Grounding	an anti-hallucination prompt that cites and admits ignorance
Providers	one interface, four back ends, switched by env var
Web UI	drag-and-drop, same engine underneath

No LangChain, no vector database, no cloud — and you understand every line.

Exercises

- **Hybrid retrieval:** merge BM25 and embedding rankings (e.g. reciprocal-rank fusion) and compare.
- **Better chunking:** split on Markdown headings or PDF layout instead of fixed size.
- **Citations with snippets:** show the exact sentence each citation came from.
- **Streaming:** stream the provider's tokens to the web UI instead of waiting for the full answer.

Next lesson

Lesson 2 · MCP servers → (interactive) — expose this document search as a Model Context Protocol tool so Claude Code can query your `documents/` folder natively, no copy-paste required.

Full source: github.com/nikolareljin/local-ai-lab · Course: nikolareljin.github.io/local-ai-lab · Author: Nik Reljin