Lesson 7 · Rebuild RAG with LangChain

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

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

> **Interactive version (slides):** https://nikolareljin.github.io/local-ai-lab/lesson-7-langchain-rag.html **Read it locally (no GitHub Pages):** `./run -l 7 lesson` **Course home:** https://nikolareljin.github.io/local-ai-lab/ **Source:** https://github.com/nikolareljin/local-ai-lab **Author:** Nik Reljin **Time:** ~45-60 min · **Prerequisites:** Lesson 1 (Lesson 3 helpful) · full objectives in SYLLABUS.md

> **Lessons:** 1 · RAG → 2 · MCP → 3 · Hybrid retrieval → 4 · RAG safety → 5 · RAG evaluation → 6 · Repo assistant → **7 · LangChain (you are here)** → 8 · LangGraph → ... → 15 · Docs from changes

> **Status: working demo.** Runnable in **Python and Node.js**. This is the one lesson that is **not** dependency-free, and that is the measurement - see **Why this lesson installs something** below.

First - what is LangChain?

If you have not used it: **LangChain is an open-source framework for building applications on top of language models.** It is not a model and not a database. It is two things:

1. **A catalogue of components** for the steps a system like ours already has - loading documents, splitting them into chunks, storing and searching embeddings, formatting prompts, calling a chat model, parsing what comes back. You import them instead of writing them.
2. **A way to compose those components**, called LCEL, where each stage is piped into the next with `|` and the resulting chain can stream, batch, and run async without you implementing any of it.

That is the whole idea. The catalogue is large - hundreds of loaders, dozens of vector stores, an adapter for nearly every model provider - and that breadth is the main reason people reach for it.

Three things worth knowing before you start:

- **It ships two official SDKs, Python and JavaScript.** Both are in this lesson. There is no official .NET version, which is why there is no C# port here - see below.
- **It moves fast.** Packages get split, renamed, and retired. One package this lesson was drafted against was sunset while it was being written, and you will see the consequence in Concept 2.

- **It is a dependency, not a library you vendor.** Two lines in `requirements.txt` pull in 18 packages. Whether that is a good trade is what the rest of the lesson measures.

You already have a working RAG pipeline from Lesson 1, built by hand. That makes you the ideal reader for this: you can look at each LangChain component and know exactly which of your own files it replaces, because you wrote the equivalent.

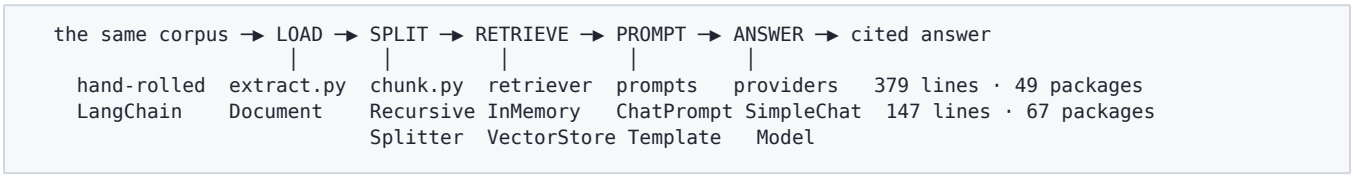
> **Why no C# in this lesson?** LangChain has no official .NET SDK - NuGet's `LangChain` 0.17.1 is an unofficial community port, still pre-1.0. Rebuilding on it would teach you a third party's reading of LangChain rather than LangChain itself. **.NET is not being skipped**: Microsoft's answer to this problem is **Semantic Kernel**, and it gets its own lesson - **Lesson 10** - rather than a footnote here.

What you'll learn

In Lesson 1 you built every RAG primitive by hand: a loader, a splitter, a retriever, a prompt, and a provider abstraction. LangChain ships all of those as components. This lesson rebuilds the **same** pipeline over the **same** corpus with the **same** system prompt, and then asks the only question worth asking about a framework:

> **What did it buy, and what did it cost?**

The goal is not to crown a winner. It is that, having written the primitives yourself, you can read LangChain and see exactly which of your files each component replaces - and price the swap.



By the end you'll understand:

- **Loaders and splitters** - `RecursiveCharacterTextSplitter` against your `chunk.py`, and why they do not produce the same chunks
- **Vector stores and retrievers** - `InMemoryVectorStore` and a `BaseRetriever` subclass against your `store.py` and `retriever.py`
- **The escape hatches** - `SimpleChatModel`, `Embeddings` and `BaseRetriever`, the three base classes you extend when the catalogue does not cover your system
- **LCEL** - `engine.answer_question()` rewritten as a chain composed with `|`
- **The bill** - lines of code you maintain against packages you did not read

> **The one idea:** a framework does not remove work, it moves it. It moves it out of your files and into your dependency tree. Both columns have a cost; only one of them shows up in a diff.

The demo

The corpus is seven small markdown documents under `data/corpus/` - a manual, an FAQ, an HTTP API reference, troubleshooting, networking, warranty, and installation notes for a fictional field sensor. The three questions live in `data/questions.json`, one aimed at each of three different documents.

Both pipelines load that corpus, split it, retrieve the top three chunks, and render a prompt. The demo compares what they **ground** on, not what a model says about it.

> **Why compare grounding, not the generated answer?** Generation is not reproducible; retrieval and the rendered prompt are. Comparing what each pipeline **feeds the model** is also the sharper test: if the evidence differs, the answer was always going to. The real model call is one command away (`ask`, below) and runs through the chat model adapter you wrote.

Run the comparison

From the repo root. The first run installs this lesson's dependencies into the course venv:

```
./run -l 7 demo                 # Python - the comparison, then the scorecard
./run -l 7 --lang node demo     # Node.js - same pipeline, different bill
./run -l 7 test                 # the offline test (works with or without LangChain)
./run -l 7 lesson               # read this lesson in a browser, served locally
./run -l 7 show                 # walk through this lesson's steps
./run -l 7                      # the playground (default)
```

> **A dependency can move your runtime floor.** `@langchain/core` and `@langchain/textsplitters` both declare `node >= 20`, so taking them would have ruled out Node 18 - which the course used to recommend - even though this lesson's Python half runs happily on 3.10. That cost is real and worth watching for. It happens not to bind here: Node 18 and 20 are both end-of-life now, so the whole course targets **Node 22** regardless, and every port declares that engine. Check this on your own projects, where the arithmetic often comes out the other way.

Output:

```
Rebuild RAG with LangChain  -  7 documents, same corpus, same system prompt
chunked at size=700 overlap=120, retrieving top 3

 hand-rolled   15 chunks   (chunk.py: collapse whitespace, break on sentences)
 langchain     15 chunks   (RecursiveCharacterTextSplitter: keep text, split on separators)

Q1  What is the factory reset procedure?
   hand-rolled   sources: manual.md:1 . warranty.md:1
   langchain     sources: manual.md:1 . warranty.md:1
   GROUNDING AGREES  -  same sources, same order

Q3  Which endpoint exports the logging buffer?
   hand-rolled   sources: api.md:1 . manual.md:1
   langchain     sources: api.md:1 . manual.md:1 . installation.md:1
   GROUNDING DIFFERS  -  langchain also cites installation.md:1

   2/3 questions grounded identically. Different chunk boundaries, mostly the same evidence.
```

Read the last line carefully, because it is the honest result. Two of three questions ground identically. One does not: LangChain's splitter carved the corpus differently, so retrieval pulled in an extra file. Neither pipeline is wrong. **"Drop-in replacement" was never true**, and a lesson that engineered all three into agreement would have taught you nothing.

Then the scorecard:

```
Component by component
 step      hand-rolled (Lesson 1)              LangChain (Lesson 7)
 ------------------------------------------------------------------------------
 load      localrag/extract.py             45L  Document(...)
 split     localrag/chunk.py               46L  RecursiveCharacterTextSplitter
 index     localrag/store.py               66L  InMemoryVectorStore
 retrieve  localrag/retriever.py          119L  BaseRetriever subclass (yours)
 prompt    localrag/prompts.py             19L  ChatPromptTemplate
 provider  localrag/providers/__init__.py  37L  SimpleChatModel subclass (yours)
 chain     localrag/engine.py              47L  LCEL  ( | )
 ------------------------------------------------------------------------------

What it cost
                          hand-rolled      LangChain
 code you maintain          379 lines      147 lines
 requirements lines                 8             10
 packages installed                49             67
 install size                 ~34 MB         ~43 MB

 Two requirements lines cost 18 packages and ~9 MB.
 Taking BM25 from the sunset langchain-community, rather than writing the
 twenty-line retriever, would have made it 80 packages and ~52 MB.
```

Those line counts are read off disk at run time, not typed into this README, so they cannot drift from the code. Re-derive the dependency numbers on your own machine with `--measure`.

Get a real answer

The comparison is offline. To run the whole chain through a model:

```
./run -l 7 ask "What is the factory reset procedure?"    # your adapter, default provider
./run -l 7 ask --native "What is the factory reset..."   # framework-native ChatOllama
./run -l 7 ask --arm embed "..."                         # real vectors, local, via Ollama
```

`--native` needs `pip install langchain-ollama`, which the lesson deliberately does **not** install for you - see **Concept 5**. Run it without that and you get the one-line command, not a traceback. `--arm embed` on its own does not: it embeds through `LocalRagEmbeddings`, the adapter you wrote, so it needs Ollama **running** but no extra package.

Experiment in the playground (needs Flask)

```
./run -l 7
```

Control	What moves
Chunk size	how each splitter carves the corpus - the control that makes the arms diverge
Chunk overlap	how much context straddles a boundary
Top k	how many chunks reach the prompt

Show the rendered prompt	the exact strings each pipeline would send a model

> The playground is a small Flask app that calls the **same functions** as the demo and the test, so nothing on the page is a special case built for the page.

Start with Q3 at the default chunk size and watch the arms disagree. Push chunk size to 1600 and watch them agree again, because at that size both splitters keep each document whole. The disagreement was never about retrieval quality; it was about where the text got cut.

Concept 1 · Loaders and splitters - what extract.py and chunk.py became

LangChain ships loaders for PDF, DOCX, HTML, and dozens more, plus a family of text splitters. For markdown, though, a `Document` is a string and a metadata dict, so this lesson builds them directly and keeps the dependency list short. That is itself a framework skill: **use the component when it earns its place, not because it exists.**

The splitter is where the two pipelines genuinely part company:

Behaviour	`localrag/chunk.py`	`RecursiveCharacterTextSplitter`
whitespace	collapses everything to single spaces first	preserves the text as written
break points	`". "`, `"! "`, `"? "`, newline, space, past the halfway mark	walks blank line, newline, space, empty in order
result here	15 chunks	15 chunks, cut in different places

Same count, different boundaries. That is the whole of the Q3 divergence.

Concept 2 · Vector stores and retrievers - what store.py and retriever.py became

`InMemoryVectorStore` replaces your `store.py` outright: it holds vectors, does the cosine search, and hands you `as_retriever()`. No FAISS, no Chroma, no binary wheel - it lives in `langchain-core`.

Retrieval is more interesting, because this lesson **writes its own retriever**:

```
class LocalRagBM25Retriever(BaseRetriever):
   documents: List[Document]
   k: int = 3
   _bm25: Any = PrivateAttr(default=None)

   def model_post_init(self, context, /) -> None:
       # Index once, at construction - the same contract as Lesson 1's
       # Bm25Retriever.__init__. Rebuilding per query would make this a slower
       # retriever than the one it is being compared against.
       self._bm25 = BM25Okapi([_tokenize(d.page_content) for d in self.documents])
```

```
    def _get_relevant_documents(self, query, *, run_manager) -> List[Document]:
        scores = self._bm25.get_scores(_tokenize(query))
        ...
```

LangChain does ship a `BM25Retriever` - in `langchain-community`, which is now **sunset upstream** and prints a deprecation warning on import. Twenty lines against `BaseRetriever` removes the dependency, the warning, and the migration you would otherwise be doing next year. It also keeps the install at 67 packages instead of 80.

> **This is the version-churn cost, arriving on schedule.** Not a hypothetical from a blog post: a package this lesson was drafted against went end-of-life during the writing of it.

Concept 3 · The escape hatches - SimpleChatModel and Embeddings

The course's default provider is the Claude Code CLI. LangChain has no adapter for it and never will. So you write one:

```
class LocalRagChatModel(SimpleChatModel):
    config: Any
    provider_name: str = "claude"

    def _call(self, messages, stop=None, run_manager=None, **kwargs) -> str:
        system = "\n".join(str(m.content) for m in messages if isinstance(m, SystemMessage))
        user = "\n".join(str(m.content) for m in messages if isinstance(m, HumanMessage))
        return get_provider(self.provider_name, self.config).chat(system, user)
```

One method. In exchange, every LCEL chain in the ecosystem can now drive Claude Code, Ollama, Gemini, or OpenAI - streaming, batching, and callbacks included, none of which you implemented.

`LocalRagEmbeddings` is the same move for `embed_documents` and `embed_query`. Note what it inherits from Lesson 1 unchanged: the Claude provider has no embedding endpoint, so asking it to embed still raises `EmbeddingError`. The wrapper did not paper over the limit, because the limit is real.

> **Read the scorecard again with this in mind.** LangChain replaced five of seven components outright. The two it could not replace are the two that had to know something about **your** system. That ratio is what a framework actually is.

Concept 4 · LCEL - what engine.answer_question() became

```
chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt_template()
    | llm
    | StrOutputParser()
)
```

Read it left to right: fan the question into a context lookup and a passthrough, render the prompt, call the model, take the text out. Your `engine.answer_question()` did the same five

things in imperative Python.

What you gain is real: every stage is a `Runnable`, so the whole chain streams, batches, and runs async without you writing any of it. What you lose is also real: when a chain misbehaves, the stack trace runs through the framework's dispatch machinery rather than your five lines. Both facts belong in the decision.

Concept 5 · Your adapter, or the ecosystem's?

`langchain-ollama` ships `ChatOllama` and `OllamaEmbeddings` off the shelf, so the same pipeline can run two ways. It is **not** in this lesson's `requirements.txt`, on purpose - the scorecard has to match exactly what `./run -l 7` installs, and this is opt-in:

```
pip install langchain-ollama            # 2 further packages, only for the second line

./run -l 7 ask "..."             # LocalRagChatModel - 61 lines you wrote and understand
./run -l 7 ask --native "..."    # ChatOllama        - one import, two more packages
```

Identical behaviour. The trade is narrow and worth naming: the adapter you wrote works with **every** provider the course supports and costs you sixty lines of maintenance. The off-the-shelf package works with one provider, costs nothing to write, and costs you a dependency plus whatever its maintainers decide next. Neither answer is universally right, which is exactly why you should be the one choosing.

What the framework buys, and what it costs

Dimension	LangChain wins	Hand-rolled wins
Swapping components	`InMemoryVectorStore` to FAISS is one line	you edit `store.py`
Ecosystem	loaders, stores, and integrations already written	you write what you need
Streaming, batching, async	free, once a stage is a `Runnable`	you implement it
Tracing and callbacks	built in	you add logging
Dependency surface	+18 packages for two requirements lines	49, and no framework among them
Cold start	slower to import	near instant
Debugging	traces run through framework dispatch	the stack is your code
Version churn	`langchain-community` sunset mid-writing	`rank_bm25` has not moved

The pattern underneath: LangChain is worth it when you will use the ecosystem - many loaders, many stores, many providers, streaming, tracing. It is a poor trade when you need

one loader, one store, and one provider, which describes a great many real systems.

Why this lesson installs something

Every other live lesson in this course runs on the standard library. This one cannot, and pretending otherwise would defeat the point: the dependency **is** the subject.

The lesson still degrades honestly. With LangChain absent, `./run -l 7 demo` runs the hand-rolled side, marks the LangChain column `not installed`, prints the install command, and exits 0. The offline test passes either way, because LangChain-dependent tests skip rather than fail.

Python and Node, and why there is no C# port

LangChain ships two official SDKs: Python and JavaScript. Both are here, and the retrieval half of their output is **byte-identical** - the Node port carries a line-for-line port of `rank_bm25`'s `BM25Okapi`, epsilon floor included, so the two runtimes rank chunks the same way.

The scorecard half deliberately is **not** identical:

Runtime	Packages	Install size	Minimum runtime
Python (`langchain-core` + `langchain-text-splitters`)	+18, to 67	+~9 MB, to ~43 MB	3.10, unchanged
Node (`@langchain/core` + `@langchain/textsplitters`)	+12, from none at all	+~48 MB	needs 20; course floor is 22

Same framework, same components, a different bill. Faking parity there would have hidden the one number this lesson exists to show you.

There is no C# port because there is no official LangChain for .NET, as the primer at the top said. NuGet has `LangChain` 0.17.1, a community port, still pre-1.0, and rebuilding on it would teach you a third party's reading of LangChain rather than LangChain. **.NET is not being skipped** - Microsoft's answer to this problem is **Semantic Kernel**, and it gets a lesson of its own, **Lesson 10**, where it can be taught on its own terms instead of as a LangChain impersonation.

Exercises

- **Make them agree:** find the chunk size at which all three questions ground identically. Is that size better for retrieval, or just better for the comparison?
- **Swap the arm:** run `ask --arm embed` to retrieve with real Ollama vectors instead of BM25. Which questions improve, and which get worse? Why would a keyword arm ever beat a semantic one?

- **Delete the adapter:** replace `LocalRagChatModel` with `ChatOllama`. How many lines actually change, and which of them are not the constructor?
- **Price your own project:** run `pip download --no-deps` on the LangChain packages a real service of yours would need. Is the number closer to 31 or to 131, and would you have guessed before measuring?

From demo to production

- **Pick one pipeline** - the comparison is a teaching artefact. Shipping both means maintaining two chunkers, and two chunkers means two sets of citations to explain.
- **Pin your versions** - `langchain-core>=1.0` is fine for a lesson and reckless for a service. Pin exactly, and read the changelog before every bump; this ecosystem moves.
- **Keep the citation contract at the boundary** - `source:page` survived the rewrite untouched. Make that contract the thing your tests assert and you can change frameworks without changing what you promise callers.
- **Measure before you adopt** - packages, install size, cold start, and stack depth on a failure. Four numbers, ten minutes, and they outlast any framework comparison you will read online.
- **Own your adapters** - the sixty lines of `LocalRagChatModel` are the reason this pipeline is not locked to one vendor. Adapters at the edge are cheap insurance.
- **Fold it into Lesson 5** - add both pipelines to the golden set and let the evaluation gate tell you whether the rewrite changed answer quality, rather than guessing from three questions.

Next lesson

Lesson 8 · LangGraph → - turn this linear chain into a stateful agent graph with retries, tool routing, and memory.

Course: nikolareljin.github.io/local-ai-lab · **Author:** Nik Reljin