Module 6: Local LLMs & RAG Architectures

Capstone: Your Private Knowledge Assistant — Local LLM + Production RAG, End to End

Overview

This capstone proves the four claims that anchor Module 6 — and it does it on a single, real codebase that the room watches grow from "empty directory" to "production-shaped local knowledge assistant" in twenty minutes:

- An LLM is just an HTTP endpoint with a runtime, a model file, and a sampling policy — and you can run all three on your laptop with no cloud bill and no API key.
- Customizing a model without retraining is a Modelfile away — version-controlled, reviewable, and deployable.
- RAG turns an LLM with no knowledge of your domain into an LLM that answers from your documents with citations — and that pipeline fits in ~300 lines of Python.
- "Production RAG" means hybrid retrieval, reranking, citation-required generation, and an eval harness that runs every commit. Skip any of the four and the system stops being honest about what it knows.

You will start with an empty directory, ingest the Module 6 reference corpus, build a custom persona model, ask the assistant a real question with citations, demonstrate refusal on an off-topic question, and run an automated eval — all inside one ~20-minute live session. The students leave the room with a complete, working, private knowledge assistant on their own laptop.

Before You Start

- Have everything from Hands-On 6.1–6.4 already installed: Ollama running, llama3.2 and nomic-embed-text pulled, Python 3.11+, and these packages: pip install ollama chromadb langchain-text-splitters rank_bm25
- Pre-copy two files into a fresh demo folder: capstone_knowledge_assistant.py and Modelfile.capstone (renamed to Modelfile for the demo). Both are in the course repo.
- Pre-copy the sample_corpus/ folder used in Hands-On 6.4 — five Markdown files covering LLM basics, Ollama, customization, embeddings, and production RAG.
- Have your editor open with the Python script visible — students will want to see the named functions on screen as you trigger each one.
- Have a terminal open in the demo folder. Live demo time is precious — the prep is not.
- Verify the service is alive before class: curl http://localhost:11434/api/tags should return JSON, not a connection error.

During Hands-On — Four Segments at a Glance

- SEGMENT 1 — Build the persona — Modelfile to a custom model with a tight system prompt — ~4 min
- SEGMENT 2 — Ingest the corpus — chunk + dense index + sparse index in one command — ~4 min

- SEGMENT 3 — Answer a real question end-to-end — hybrid retrieval, rerank, citations, refusal — ~7 min
- SEGMENT 4 — Eval harness — print the pass/fail number that lets you change anything safely — ~5 min

Remember Module 6 is not about which model is biggest. It is about owning the entire stack — model, runtime, embedding, retrieval, reranker, prompt, eval — on hardware you can touch. Every step in this capstone is a piece of architecture you should be able to defend in a design review. By the end of these twenty minutes you will have demonstrated the entire AI-native engineering workflow for a private knowledge assistant — the same workflow that powers every enterprise "chat with your docs" product, running on your laptop with zero outside dependencies.

Segment 1 — Build the Persona

What to Learn An LLM with no persona is a chatbot. An LLM with a tight persona is a product. A Modelfile is the smallest possible deployable unit of persona: 10 lines of plain text, version-controlled, and reviewable in a pull request. Before we touch a single document, we lock in WHO the model is and HOW it is allowed to behave. That order matters — persona before knowledge — because persona constrains every answer the model will give in segments 2 through 4.

Step 1. Open the Modelfile.

Open Modelfile in your editor. Read it aloud, line by line. Do not paraphrase — read the actual text. The students need to see how short the file is.

```
FROM llama3.2

SYSTEM """
You are kb-assistant, an internal knowledge assistant for an engineering team.
You answer questions strictly from CONTEXT provided in each prompt.

Style:
- Concise, technical, no preamble.
- Use bullet lists when the answer has multiple distinct points.
- Cite the source for every factual claim using [SOURCE: filename, chunk_id]
  exactly as it appears in the CONTEXT block.

Refusal:
- If the CONTEXT does not answer the question, reply exactly with:
  "I cannot answer from the provided context."
- Do not invent facts. Do not use outside knowledge.
- Do not apologize.
"""

PARAMETER temperature 0.2
```

```
PARAMETER num_ctx 8192
PARAMETER top_p 0.9
```

What you should get Every engineer in the room should be able to read this and predict three things the model will refuse to do: invent facts, apologize, and answer without citations. That is the value of a Modelfile — behavior is specified, not hoped for. The temperature of 0.2 is the other key choice: we want consistent, low-creativity answers, not poetic ones.

Step 2. Build and verify the model.

```
ollama create kb-assistant -f ./Modelfile
ollama list
```

Point out the new entry in ollama list. Note its size in the list — almost zero, because the model is a thin overlay over llama3.2 and shares the underlying weights.

What you should get A new model named kb-assistant in the list. Run one quick test: ollama run kb-assistant "Hello!" — the model should respond curtly, no preamble, with the persona's style. That curtness is the persona working. We have not given it any context yet, so any factual question right now should produce a refusal — that is also expected and good.

Segment 2 — Ingest the Corpus

What to Learn Ingestion is where most RAG systems silently fail. Big chunks hide relevant facts. No overlap loses facts that span paragraph boundaries. One retrieval index makes the system fragile to query phrasing. The capstone script handles all three: it splits documents at ~400 characters with 50 characters of overlap using a recursive splitter that respects paragraph and sentence boundaries, it builds a Chroma dense index using nomic-embed-text, AND it builds a BM25 sparse index — all in one command. The room sees the chunk counts per file print to the terminal in under a minute.

Step 1. Show the corpus.

```
ls sample_corpus/
```

Five Markdown files: 01_llm_basics.md, 02_ollama.md, 03_customization.md, 04_embeddings.md, 05_production_rag.md. About 2 KB each. Mention that they are the same docs students worked with in Hands-On 6.4 — but the capstone script handles them differently because the script is built to handle ANY corpus, not just this one.

Step 2. Run ingest.

```
python capstone_knowledge_assistant.py ingest ./sample_corpus
```

The script prints one line per file with the chunk count, then a summary footer showing both index sizes and where they were persisted.

What you should get Output that looks like: 01_llm_basics.md 6 chunks 02_ollama.md 7 chunks 03_customization.md 8 chunks 04_embeddings.md 6 chunks 05_production_rag.md 10 chunks Indexed 37 chunks across 5 files. Chroma -> .kb_chroma/ BM25 -> .kb_bm25.json Point at the two index lines. Two retrievers, one ingest. That is the disciplined version of what most teams do with only one.

Segment 3 — Answer a Real Question, End-to-End

What to Learn This is the moment where every concept from the module lands. ONE call to answer() runs: embed the question, search dense, search sparse, fuse with Reciprocal Rank Fusion, rerank the top-8 with the LLM, format the top-4 with [SOURCE] tags, hand them to the kb-assistant model, and stream back an answer with inline citations. Five steps, one function call. The room sees it happen in real time. Then we ask a hostile question and the model refuses cleanly — that refusal is the proof that the persona and the prompt are doing the work they were designed to do.

Step 1. The on-topic question.

```
/ver
```

Read the answer aloud as it streams. Three things to point out: (1) the bullet structure — that is the persona's style block enforcing structure, (2) the [SOURCE: 05_production_rag.md, chunk_N] tags throughout — those are auditable, (3) the conciseness — temperature 0.2 plus the "no preamble" rule are visibly working.

What you should get An answer that explains hybrid search as the fusion of dense and sparse retrieval, mentions BM25 by name, mentions Reciprocal Rank Fusion, and ends with one citation pointing to 05_production_rag.md. Each cited chunk is the actual passage that contained that fact — show the students how they could click through to verify. That auditability is the property that makes this system shippable to an enterprise.

Step 2. The verbose pass — see retrieval, rerank, prompt assembly.

```
python capstone_knowledge_assistant.py chat
>  /verbose
>  What is hybrid search and why does it beat dense retrieval alone?
```

Verbose mode prints the full CONTEXT block the model received, so the room can see exactly which 4 chunks were chosen and how they were formatted. Point at one chunk and trace its [SOURCE] tag through to the final answer. That trace is the working hallucination guard — every citation is a real, retrievable chunk.

What you should get Four formatted chunks in the CONTEXT block, each prefixed with [SOURCE: filename, chunk_N]. The reranker has placed the most directly relevant chunk first — that ordering matters because LLMs pay disproportionate attention to early context. This is exactly why we rerank: getting the right chunk on top, not just into the top-K.

Step 3. The refusal — proof the guardrail works.

```
>  What is a good recipe for lasagna?
```

What you should get Exactly one line: "I cannot answer from the provided context." No apology, no creative attempt, no "however, here is what I know about Italian cooking." Point at it. THIS is the test of a real RAG system: can it correctly decline. Every commercial "chat with your docs" product either does this or has a credibility problem. We just built it from scratch in twenty lines of prompt.

Step 4. The fusion question — corpus reasoning across docs.

```
>  How do tokens, parameters, and embeddings each relate to memory usage?
```

What you should get An answer that cites at least 01_llm_basics.md AND 04_embeddings.md — two distinct sources fused into one coherent answer. Hybrid retrieval plus reranker pulled chunks from both. The model synthesized them under the persona's structure rules. This is the property RAG buys you that no single-document chatbot can match: cross-document reasoning, with citations preserved.

Segment 4 — Eval Harness — The Number That Makes Everything Else Safe to Change

What to Learn Without an automated eval, every change to the pipeline is theater. A chunk-size change "feels better." A prompt tweak "reads cleaner." Neither claim survives contact with a regression test. The eval harness in this capstone runs four fixed questions, scores each answer for relevance and grounding via LLM-as-judge, checks for the expected citation, and prints one pass/fail number. That single number is the artifact that lets you change anything tomorrow — chunk size, embedding model, rerank strategy, prompt — and immediately know whether you helped or hurt.

Step 1. Read the eval set on screen.

Open capstone_knowledge_assistant.py and scroll to EVAL_QUESTIONS. Four entries, each with a question, the source it should cite, and three key facts the answer must touch. Point out the modest size — four questions is enough to catch major regressions. Five to twenty curated questions is the sweet spot for a small-team RAG eval.

Step 2. Run the eval.

```
python capstone_knowledge_assistant.py eval
```

The script runs each question through the full pipeline, scores the answer with the LLM-as-judge, checks the citation, and prints per-question results plus a final summary block.

What you should get Output that ends with: avg relevance : 8.3 avg grounding : 9.0 passed : 4/4 RESULT : PASS Point at the RESULT line. That is the single artifact that makes RAG engineering. Without it you have prompt-engineering-by-vibe. With it you have a regression test you can run on every commit. The eval harness is what turns a demo into a system you can change without breaking.

Step 3. Break it, watch it fail.

Open capstone_knowledge_assistant.py and intentionally weaken the system prompt — change "I cannot answer from the provided context" to "I will do my best to help" and remove the "Do not invent facts" line. Save. Re-run the eval.

```
python capstone_knowledge_assistant.py eval
```

What you should get Grounding scores drop. At least one question fails. RESULT: FAIL. Restore the original prompt. Re-run. RESULT: PASS. That round-trip — change, eval, revert, eval — is the workflow that lets a team iterate on RAG safely. The eval makes "that helped" or "that hurt" answerable in 30 seconds instead of a week of guesswork in production.

Wrap-Up — The Whole Stack, Running on Your Laptop

Remember In twenty minutes you built — from scratch — a complete private knowledge assistant: a custom local LLM persona, a hybrid retrieval index over a real document corpus, an LLM reranker, a citation-required prompt with working refusal, and an automated eval harness with pass/fail output. That is not a toy. That is the actual workflow your students will use on real codebases and real internal documentation starting Monday morning. Every component is replaceable: swap the model, swap the embedding, swap the vector DB, swap the reranker — the eval harness will tell you within minutes whether the swap was an improvement.

Key takeaways:

- Persona before knowledge — lock the behavior in a Modelfile, then add documents.
- Always run two retrievers — dense for meaning, sparse for exact tokens, fused with RRF.
- Rerank the top-N before generation — it is the cheapest accuracy lift in RAG.
- Citation-required prompts plus a clean refusal line are the working hallucination guard.
- An eval harness is not optional — it is the regression test that makes everything else safe to change.
- Every component in this stack runs on your laptop. No API keys, no cloud bills, no data leaving your machine. That is the entire promise of Module 6.

Student Lab — Module 6 Capstone Submission Take the capstone code and your own document collection — your team's runbooks, your project README files, your saved technical blog posts, your meeting notes — and run the full Module 6 pipeline on it: ingest the corpus, build a custom persona Modelfile tuned for your domain, and run the eval harness against at least eight test questions YOU write that genuinely probe the corpus (including at least one question that should refuse).

Submit five things: the GitHub repo link, the Modelfile, the eval question set with expected citations, a screenshot of the eval RESULT: PASS line, and a 3-minute video walkthrough showing one on-topic answer with citations and one correct refusal. This is the gate to Module 7 — Applied Computer Vision.

Appendix — capstone_knowledge_assistant.py (the full source used in this demo)

This is the complete, runnable Python file that drives the entire 20-minute demo. Save it as capstone_knowledge_assistant.py at the root of the demo folder before class begins. The full source is also in the course repo at code/capstone_knowledge_assistant.py and code/Modelfile.capstone.

The script has four CLI subcommands — ingest, ask, chat, eval — and roughly 300 lines of Python total. Every concept from Module 6 is implemented in named functions that map directly to the four lectures: ingestion (Lecture 6.4 chunking), embedding (Lecture 6.3), hybrid retrieval and RRF (Lecture 6.4), LLM reranking (Lecture 6.4), citation-aware generation (Lecture 6.4), and the LLM-as-judge eval harness (Lecture 6.4). The kb-assistant model itself is built from the Modelfile shown in Segment 1 (Lecture 6.3). The whole module — every concept, every technique — is reachable from one file.

Walk through the script briefly at the end of class if time allows. Otherwise direct students to the repo and the comments at the top of the file.