Orchestrating Local RAG with Ollama

Modern Software Development with GenAI

Announcements

FINAL PROJECT UPDATE

Vertical prototype is due on Thursday 10/29. · **due Thu, Oct 29**

Prototype

P3 IS DUE ON 11/03

I gave you a working project. You need to extend it to get full credit

Start Your Downloads Now

If you did not complete pre-class checklist

```
# 1. Get today's files
git fetch upstream && git merge upstream/main
cd "ex12/exercise-12"

# 2. Start the four services, then build the app images
cp .env.example .env
docker compose up -d mysql chromadb ollama infinity
docker compose build
```

- **Docker Desktop must be running** first — and keep your laptop **plugged in**
- Need **~15 GB free disk**; an old lecture's stack still running? `docker compose down` in that folder frees its ports
- Let it download — **start the quiz while it runs**

Check 1 — Services Up

RUN

```
docker compose ps

# first run only — ~1 GB
docker compose exec ollama \
  ollama pull qwen2.5:1.5b
```

YOU SHOULD SEE

```
mysql     Up (healthy)
chromadb  Up
ollama    Up (healthy)
infinity  Up (healthy)

pulling manifest ...
success
```

- `infinity` says **(health: starting)**? It's still downloading its model — give it a minute
- **Healthy ≠ ready:** Ollama is healthy before any model exists — the **pull** is what makes it useful

Check 2 — Call Each Service by Hand

OLLAMA — GENERATE TEXT

```
curl localhost:11434/api/generate -d '{"model": "qwen2.5:1.5b",
  "prompt": "Say hi in five words", "stream": false}'
```

INFINITY — EMBED TEXT

```
curl localhost:7997/embeddings -H "Content-Type: application/json" \
  -d '{"input": ["hello"], "model": "sentence-transformers/all-MiniLM-L6-v2"}'
```

- ✓ Ollama returns JSON with a `"response"` field; Infinity returns `"data"` with an `"embedding"` list
- First Ollama call is **slow** — it's loading the model into memory
- **Windows:** run these in **Git Bash**, not PowerShell (quoting differs)

Check 3 — The App Runs

RUN

```
docker compose up --build -d \
  backend frontend

docker compose exec backend \
  pytest tests/ -q
```

YOU SHOULD SEE

- **localhost:5173** loads, with an **Ask AI** tab
- Asking a question **errors** — expected, two functions are stubs
- **7 passed, 7 failed** — the 7 are today's task

Stuck? The Usual Suspects

Symptom	Fix
`Cannot connect to the Docker daemon`	Start **Docker Desktop**, wait for it to say running
`port is already allocated`	An old stack holds the port — `docker compose down` in that lecture's folder
`no space left on device`	`docker system prune` — removes stopped containers and unused images
`infinity` stuck on **health: starting**	Still downloading its model — `docker compose logs -f infinity` to watch
Downloads crawling	Keep going — **pair with a neighbor** for the next steps

Every Mock Becomes a Real Service

LECTURES 11–12

Embed: **mock** vectors

Search: cosine **by hand** over a Python list

Generate: prompt built, **never sent**

→

TODAY

Embed: **Infinity** · `all-MiniLM-L6-v2`

Search: **ChromaDB** · cosine index

Generate: **Ollama** · `qwen2.5:1.5b`

- Chunker and prompt builder **carry over unchanged**
- Only the three stand-ins get replaced — **nothing you built is thrown away**

The Stack Grows from 3 Services to 6

ALREADY RUNNING

mysql	backend	frontend

▼ NEW TODAY — ONE JOB EACH

ollama	**infinity**	**chromadb**
generates text	turns text into vectors	stores + searches vectors
⬇ **pulls model weights**	⬇ **pulls model weights**	no model — just an index

- What you just saw: **named volumes** keep the weights across restarts; **healthchecks** make the backend wait
- **Ollama** — runs LLMs locally behind a REST API · **Infinity** — a REST server for embedding models · **ChromaDB** — a vector database

You Just Called Ollama

REQUEST

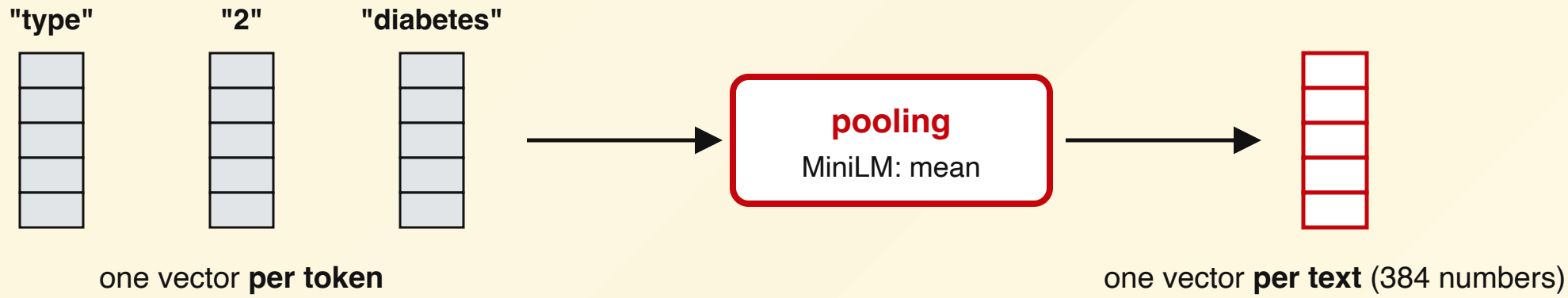
```
POST /api/generate
{"model": "qwen2.5:1.5b",
 "prompt": "...",
 "stream": false}
```

RESPONSE — ONE JSON OBJECT

```
{"model": "qwen2.5:1.5b",
 "response": "the answer...",
 "done": true, ...}
```

- Leave out `"stream": false` and you get **one JSON line per chunk** instead
- Same request shape as a cloud API — but **no key, no per-token bill, data stays local**

Pooling: Many Vectors → One



- **Pooling** — how a model collapses its per-token vectors into **one vector for the whole text**
- Mean (MiniLM), first `[CLS]` token (BERT), last token (gte-Qwen2) — **pooling comes with the model**

Infinity Lets You Pick the Model

OLLAMA

Built for **generation**

Embeds only with models **packaged for Ollama**

INFINITY

Built **only for embeddings**

Loads **any sentence-transformers model** by its Hugging Face ID

- Course standard: `sentence-transformers/all-MiniLM-L6-v2` — small, ungated, **mean pooling**
  — **swapping a embedding models is a config change**, not new code

Sort Embeddings by Index

✗ TRUST THE RESPONSE ORDER

```
# you send
texts = ["A", "B", "C"]
# you get back
data = [{"index": 2, C's vector},
        {"index": 0, A's vector},
        {"index": 1, B's vector}]
# stored by position
"A" ↔ C's vector
"B" ↔ A's vector
"C" ↔ B's vector
```

No error — every chunk is just **stored under the wrong text**

✓ SORT BY INDEX FIRST

```
data.sort(key=lambda d: d["index"])
# now data is
[{"index": 0, A's vector},
 {"index": 1, B's vector},
 {"index": 2, C's vector}]
# stored by position
"A" ↔ A's vector
"B" ↔ B's vector
"C" ↔ C's vector
```

Vectors line up **positionally with your input**

Each item in Infinity's `data` carries an `index` because the API **doesn't promise** input order — trusting it is a **silent** bug.

Ingest and Query: Same Model

SAME MODEL

Stored and query vectors live in **one space**

→ **correct neighbors**

DIFFERENT DIMENSIONS

384-d stored, 1536-d query

→ ChromaDB **rejects the query** (loud)

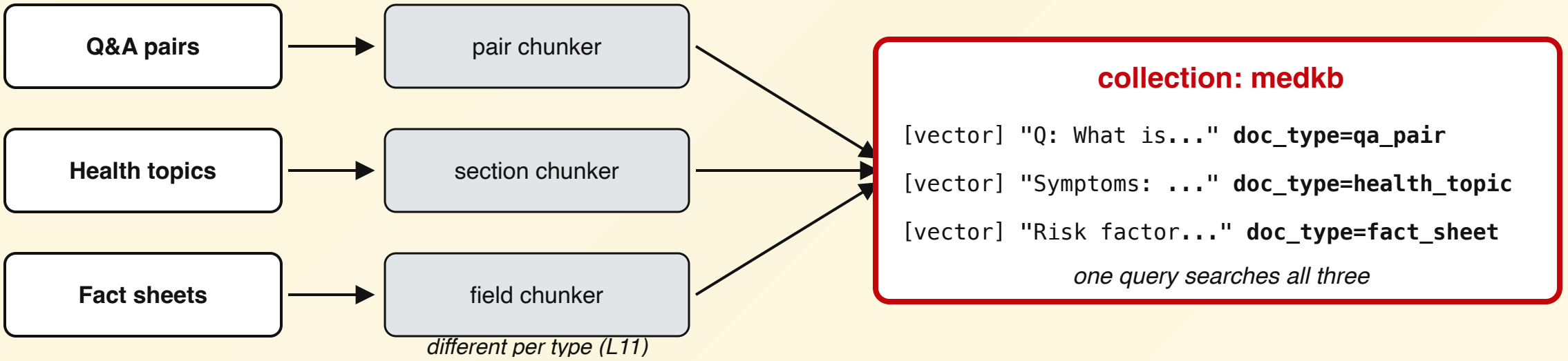
SAME SIZE, DIFFERENT MODEL

384-d and 384-d, different training

→ **wrong neighbors, no error** (silent)

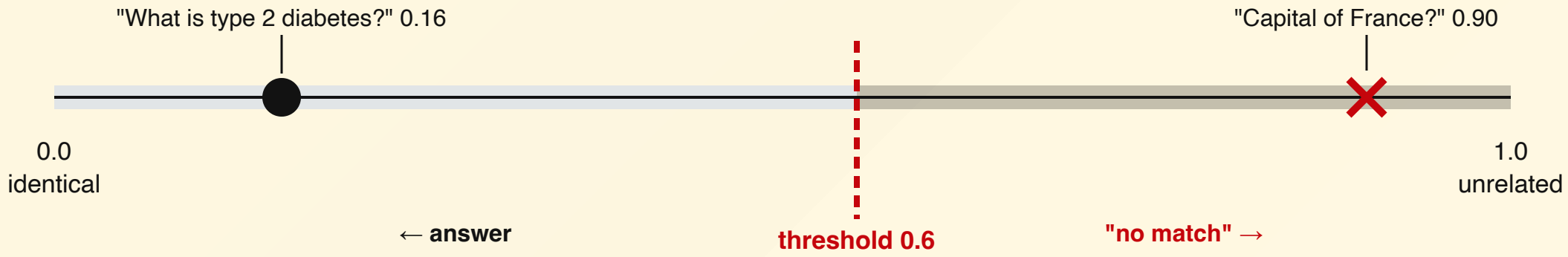
Each model builds its **own space** — vectors from two models aren't comparable, even at the same size. **Change the embedding model → re-ingest into a fresh collection.**

One Collection, Three Data Types



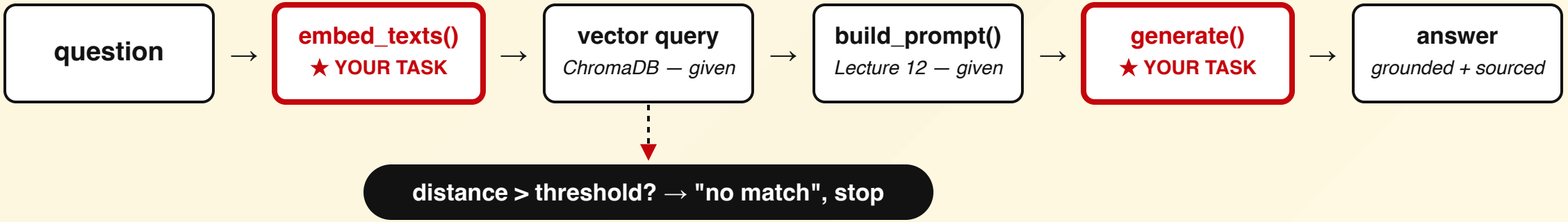
- **Chunk differently, retrieve uniformly** — a `doc_type` **metadata** field remembers each chunk's source
- We pass **Infinity's vectors** to `collection.add(embeddings=...)` — leave it out and Chroma embeds with **its own model**

Cosine Distance: Small Means Similar



- **Cosine distance** = 1 − cosine similarity — what ChromaDB returns for each hit
- Range is **0–2 in theory** (similarity −1…1); real text embeddings stay in **~0–1**
- Best hit **past the threshold** → nothing relevant was found → **answer "no match"**
- Distances measured with MiniLM on today's corpus; the threshold is a **tunable parameter** in Project 3

The Full Pipeline, Wired Together



This is **`pipeline.answer_question()`** — Lectures 10–13 each contribute a step.

The threshold check sits **before** generation — an out-of-scope question **never reaches the LLM**, so it can't be forced to answer from irrelevant context.

Example 1, Step 1 — Embed the Question

> User asks: **"What are common symptoms of type 2 diabetes?"**

CODE

```
question = "What are common symptoms of type 2 diabetes?"
query_vector = embed_texts([question])[0]
```

RESULT — 384 NUMBERS

```
[0.12, -0.08, 0.31, 0.44, ...]
```

- `embed_texts()` sends the question to **Infinity** and gets back **one vector**
- Values are illustrative — together they encode the question's **meaning** in the model's embedding space

Example 1, Step 2 — ChromaDB Finds the Closest Documents

```
results = collection.query(
    query_embeddings=[query_vector],
    n_results=3)
```

Stored document in `medkb`	Cosine distance
Symptoms of type 2 diabetes include increased thirst and frequent urination.	**0.16**
Diabetes risk factors include family history and inactivity.	0.34
The capital of France is Paris.	0.90

The closest result has distance **0.16** — small means similar.

Example 1, Step 3 — Check the Threshold

CODE — THRESHOLD = 0.60

```
best_distance = min(results["distances"][0])
if best_distance > 0.60:
    return "No match"
```

THIS QUESTION

0.16 < 0.60

→ a relevant document exists

→ **continue** to the prompt

Example 1, Step 4 — `build_prompt()` Prepares the Context

PROMPT SENT TO THE LLM

```
Instructions
Answer the question using the provided context. Do not introduce
unsupported medical claims. Cite the source of the information.

Context
Symptoms of type 2 diabetes include increased thirst and frequent urination.

Question
What are common symptoms of type 2 diabetes?
```

The question is combined with the **retrieved documents** — the LLM answers from this context, not from memory.

Example 1, Step 5 — `generate()` Produces an Answer

CODE

```
answer = generate(prompt)
```

THE MODEL MIGHT RESPOND

> Common symptoms of type 2 diabetes include increased thirst and frequent urination.
>
> **[Source: Diabetes symptoms document]**

The pipeline returns that **grounded, sourced** answer to the user.

Example 2 — A Question It Cannot Answer

> User asks: **"What is the capital of France?"** — the pipeline follows the same steps

STEP 1 · EMBED

Question → 384-number vector

STEP 2 · BEST CHROMADB RESULT

"Symptoms of type 2 diabetes include increased thirst..."

COSINE DISTANCE

0.90

The knowledge base is **medical** — even the "closest" document is **far away**.

Example 2 — The Threshold Rejects It

STEP 3 · THRESHOLD CHECK

0.90 > 0.60

→ Reject — "No match"

Pipeline **stops here**

STEP 4 · BUILD_PROMPT()

never called

STEP 5 · GENERATE()

never called

The key safeguard: the LLM never gets the chance to **invent an answer** from unrelated medical documents.

Putting It All Together

```
def answer_question(question):
    # 1. Embed the user's question
    query_vector = embed_texts([question])[0]
    # 2. Search the knowledge base
    results = collection.query(query_embeddings=[query_vector], n_results=3)
    # 3. Reject questions without a close match
    distances = results["distances"][0]
    if not distances or min(distances) > threshold:
        return "No match"
    # 4. Construct a prompt from retrieved documents
    documents = results["documents"][0]
    prompt = build_prompt(question, documents)
    # 5. Generate an answer
    return generate(prompt)
```

- Order: **embed → retrieve → check threshold → build prompt → generate** · illustrative — your signatures may differ
- The key design decision: the threshold check happens **before** generation

Activity — Two Functions

```
backend/app/rag/
  chunking.py          L11 — done
  prompt_builder.py    L12 — done
  infinity_client.py   embed_texts()
  ollama_client.py     generate()
  vector_store.py      given
  ingest.py            given
  pipeline.py          given
```

embed_texts() — POST to `/embeddings`, **sort by index**, return vectors in input order

generate() — POST to `/api/generate` with **stream: false**, return `response`

Pass a **generous** `timeout=` to `httpx.post` — a laptop CPU takes longer than httpx's **5-second default**.

Answer — `embed_texts()`

BACKEND/APP/RAG/INFINITY_CLIENT.PY

```python
def embed_texts(texts: list[str], base_url: str) -> list[list[float]]:
    """...docstring..."""
    response = httpx.post(
        f"{base_url}/embeddings",
        json={"input": texts, "model": EMBEDDING_MODEL},
        timeout=60,
    )
    response.raise_for_status()
    data = sorted(response.json()["data"], key=lambda d: d["index"])
    return [d["embedding"] for d in data]
```

- **POST** all the texts to `/embeddings` in **one request**
- **Sort by** `index` first (slide 18) so the vectors line up with `texts`
- `timeout=60`: a CPU batch is slow · `raise_for_status()` surfaces HTTP errors

Answer — `generate()`

BACKEND/APP/RAG/OLLAMA_CLIENT.PY

```python
def generate(prompt: str, base_url: str, model: str = GENERATION_MODEL) -> str:
    """...docstring..."""
    response = httpx.post(
        f"{base_url}/api/generate",
        json={"model": model, "prompt": prompt, "stream": False},
        timeout=120,
    )
    response.raise_for_status()
    return response.json()["response"]
```

- Same request as the Check 2 `curl` — now from Python
- `"stream": False` → **one JSON object**; return its `"response"` (slide 15)
- `timeout=120`: the **first** call loads the model — httpx's default is 5 s

Check 4 — Answer or "No Match"

RUN

```
docker compose exec backend \
  pytest tests/ -q
```

14 passed

THEN ASK, IN THE ASK AI TAB

"What is type 2 diabetes?" → **grounded answer + sources**

"What's the capital of France?" → **"no match"**, not a made-up answer

The first question is slow — it **embeds the whole corpus** once before answering. Commit and push both files when it works.

Questions?