Build a second brain that answers from your notes and PDFs

Ask "what does my lease say about breaking it early?" and get the answer with the file and page it came from. A real retrieval pipeline (hybrid search, a relevance floor, an eval you run yourself), free, on your laptop.

Building · 60 min build

About 60 min · $0 · beginner-friendly, real engineering. It runs on your laptop and the Claude plan you already pay for, so there's no API key.

What you'll have at the end

Claude Code with a new tool, second-brain. In any project you can ask:

- "how did I fix that Supabase logout bug last time?"
- "what was error PGRST301 about?"
- "what does my lease say about breaking it early?"

It pulls the answer out of your own notes and PDFs and tells you which file, which page and which passage it came from, with a score. When nothing in your files is close to the question, the tool returns "NO RELEVANT PASSAGES" instead of five weak ones, so Claude has nothing to make an answer out of.

You also get a number for how good the search is on your own files: hit@1, hit@3 and MRR, measured on questions you write, for plain vector search and for the hybrid search you'll build. On my test files hybrid put the right passage in the top 3 for 22 of 22 questions; vector search alone managed 20 (the full numbers are in "Tested on").

What you need

- Claude Code (Pro plan or higher)
- Python 3.10 or newer. Check with python --version (Windows) or python3 --version (Mac).
- A folder of files: notes as .md or .txt (an Obsidian vault already is one; from Notion, export as Markdown) and any PDFs you want to ask about: a lease, an insurance policy, a car manual, a job contract, a syllabus. No notes yet? Write 5 quick ones about real stuff (a bug you fixed, a decision you made, your gym split) and drop in one PDF.

- About 92 MB of disk for the embedding model (downloaded once, in step 2) and an internet connection for that one download. After it, nothing leaves your laptop.

You don't have to write the code. Each step is a prompt you paste into Claude Code, and a "Check it" line that tells you what you should see. If Claude Code's version misbehaves, the code I tested is in the appendix: paste a file from there and say "use this instead".

How it works

Your files get cut into passages. Each passage is stored two ways: as a vector (a list of 768 numbers that captures what it means) and as plain words. A question is searched both ways, by meaning and by exact words, and the two result lists are merged. Passages that aren't close enough are dropped. A tiny server hands that search to Claude Code as a tool, and Claude Code does the answering, so there's no API key and nothing extra to pay.

That's retrieval-augmented generation (RAG): retrieve the right passages first, then let the model write from them. "Under the hood" below walks through each stage.

Step 1: set up the project

Open a terminal in an empty folder, start Claude Code, paste:

Paste into Claude Code

```
Set up a Python project in this folder called second-brain:
- create a virtual environment in .venv
- install chromadb, pypdf, mcp, rank-bm25, onnxruntime, tokenizers and huggingface_hub into it,
  and write requirements.txt
- create a notes/ folder
Then tell me the exact command to activate the venv on my operating system.
```

Copy your notes and PDFs into notes/. Copy, don't move: the originals stay where they are. Subfolders are fine.

Check it: with the venv active, this shows all four, and mcp's version starts with 2.

Copy and paste

```
pip show chromadb pypdf mcp rank-bm25
```

Step 2: load your notes and PDFs

Paste into Claude Code

```
Write two files.

rag.py, the shared retrieval code (every other script imports it):
- build every path from Path(__file__).parent: notes/ and brain_db/
- an embedder for the model MongoDB/mdbr-leaf-ir, run with onnxruntime (no torch, no
  sentence-transformers): get onnx/model.onnx, onnx/model.onnx_data and tokenizer.json with
  huggingface_hub (try local_files_only=True first, so it works offline after the first run),
  tokenize with the tokenizers package (truncate at 512 tokens, pad the batch), feed input_ids,
  attention_mask and token_type_ids (zeros), and read the "sentence_embedding" output: 768 numbers,
  already unit length. For questions only, put this in front of the text:
  "Represent this sentence for searching relevant passages: "
- a Chroma PersistentClient in brain_db/, collection "notes", embedding_function=None because we
  pass our own vectors, configuration={"hnsw": {"space": "cosine"}}
- chunk_file(path): a .md file is split at its headings and every chunk starts with its heading
  path ("Title > Section"); a .txt file is one section; a PDF is read page by page with pypdf and
  chunked per page, so the page number is exact. Inside a section, pack whole paragraphs into
  chunks of up to 800 characters with no overlap; cut a longer paragraph at a sentence end.
- a chunk's id is the first 16 hex characters of sha256("file|page|text"), with -2, -3 added when
  the same text repeats in a file. Metadata: source (path under notes/), page (0 for notes),
  heading.

ingest.py:
- chunk every .md, .txt and .pdf under notes/ (subfolders too)
- compare the new chunk ids with the ids already in the collection: embed and add only the new
  ones, delete the ones that no longer exist, leave the rest alone
- a PDF with no text at all is skipped and listed at the end under "No text found (scanned?)"
- write the current time to brain_db/last_ingest.txt
- print "Stored N chunks from M files (added A, unchanged U, removed R)"
Then run it twice.
```

Check it: the first run prints "Stored N chunks from M files (added N, unchanged 0, removed 0)" and M matches how many files you have. The second run prints "added 0, unchanged N, removed 0": nothing was embedded twice. The first run downloads the model (about 92 MB) once; on my connection the whole first run took 14 seconds.

If it lists a file under "No text found (scanned?)": that PDF is a photo of pages, so there's no text to read. Claude Code can read it for you. Paste:

Paste into Claude Code

```
Read notes/[the scanned file].pdf and type out everything it says into notes/[same name].txt,
keeping the headings. Then run ingest.py again.
```

It will print "added 1" (or more) and leave everything else alone. Answers from that file will cite the .txt.

Step 3: test the search, with numbers

First build the search:

Paste into Claude Code

```
Add hybrid search to rag.py and write search.py.

In rag.py, a class Index that:
- loads every chunk (text, metadata, vector) from the collection into memory and builds a BM25
  keyword index over the texts with rank_bm25.BM25Okapi. Tokens: lowercase runs of letters, digits
  and underscores; a hyphenated term is kept whole and in parts (INV-2291 -> inv, 2291,
  inv-2291); drop a short list of stopwords
- reloads itself when brain_db/last_ingest.txt changes
- search(question, mode="hybrid", top_k=5, floor=SIM_FLOOR) does this:
  1. vector: embed the question, take the dot product with every chunk vector (exact cosine
     similarity, not Chroma's approximate query), keep the 20 best
  2. keyword: the 20 best BM25 chunks that share at least one word with the question
     (skip this when mode="vector")
  3. fuse the two lists with reciprocal rank fusion: a chunk's score is the sum of 1/(60 + rank)
     over the lists it appears in; break ties by similarity
  4. relevance floor: drop a chunk whose similarity is under SIM_FLOOR (start at 0.35) unless it
     contains a question term that looks like an identifier (has a digit or an underscore, 3+
     characters), like PGRST301. floor=None turns the floor off.
  5. return the top 5 (id, source, page, heading, text, similarity, vector rank, keyword rank)
     and the best similarity seen
- format_hits(): one block per passage, starting with a line like
  "[1] lease.pdf, page 3 | chunk 2a04633acca1f167 | similarity 0.51 | vector #1, keyword #2"
  then the text. With no passages it returns "NO RELEVANT PASSAGES", the best similarity and the
  floor, and tells the model to say the notes don't cover it and not to answer from general
  knowledge.

search.py: python search.py "a question" prints format_hits; add --vector for vector-only.
Then run it on a question I'll give you.
```

Then measure it. This is the step that tells you whether the thing works, instead of hoping:

Paste into Claude Code

```
Now measure the search. Ask me for 12 questions about my notes, in my own words (not copied from
the notes):
- 8 that my notes answer: at least 2 about a PDF, and at least 2 that contain an exact code,
  number or name (an error code, an invoice number, a section number)
- 4 that my notes do NOT answer
For each answerable one, find the file and page that answers it by reading notes/ yourself (not
with search.py) and show me the list to confirm. Save it as evals.jsonl, one per line:
{"q": "...", "file": "path under notes/", "page": 3}   (page 0 for notes; "file": null for the
no-answer ones)

Then write evals.py that:
- for vector-only and for hybrid, with the floor off, finds the rank of the right file and page
  in the top 10 for each answerable question, and prints hit@1, hit@3 and MRR for both modes
- lists the questions where the two modes disagree
- for floors 0.15, 0.20, 0.25, 0.30, 0.35, 0.40 prints how many answerable questions still have
  the right passage in the top 3, and how many no-answer questions return nothing
Run it.
```

Check it: you get a table like this one (mine, on the 109-file stress test):

What you should see

```
mode       hit@1   hit@3     MRR
vector      0.86    0.91   0.890
hybrid      0.95    1.00   0.977
```

hit@1 is how often the right passage came first, hit@3 how often it was in the top three, MRR the average of 1/rank (1.0 means always first). Hybrid's hit@3 should be at least as high as vector's; the questions with exact codes are where it pulls ahead. Then read the floor table and set SIM_FLOOR in rag.py to the row you like: a higher floor refuses more off-topic questions and loses more real ones. If hit@3 is under 0.8, fix it here (see "When it breaks"), not later.

Step 4: turn the search into a tool

Paste into Claude Code

```
Write brain_server.py, an MCP server using the mcp package version 2.
Use: from mcp.server.mcpserver import MCPServer   (NOT FastMCP, that's the old v1 name)
- one tool, search_notes(question: str), that creates rag.Index() on the first call, runs
  index.search(question) and returns rag.format_hits(...)
- the tool's docstring tells the model: it searches the user's own notes and PDFs; cite the file
  and page; if it returns NO RELEVANT PASSAGES, say the notes don't cover it and don't guess;
  include exact identifiers (error codes, invoice numbers, names) in the question
- never print anything in this file (stdout is the MCP channel)
- run it with server.run()
Then write test_server.py: start brain_server.py over stdio with the mcp client, with the working
directory set to my home folder, list its tools, and call search_notes with the questions given on
the command line. Run it with one question my notes answer and one they don't.
```

Check it: the test lists one tool, search_notes. The first question comes back with a passage whose first line names the file (and page), a chunk id and a similarity. The second comes back as "NO RELEVANT PASSAGES".

Step 5: plug it in with one command

Ask Claude Code for the full paths to your venv's Python and to brain_server.py, then run this in a normal terminal (not inside Claude Code):

Copy and paste

```
claude mcp add --scope user second-brain -- [FULL PATH TO THE VENV'S PYTHON] [FULL PATH TO
brain_server.py]
```

On Mac the Python path ends in .venv/bin/python; on Windows it ends in .venv\Scripts\python.exe. --scope user means it works in every project, not just this folder.

Check it: this shows second-brain with a check mark and Connected.

Copy and paste

```
claude mcp list
```

Step 6: ask it anything

Open a new Claude Code session anywhere and paste:

Paste into Claude Code

```
Use the second-brain tool to answer: [a question about your notes or PDFs].
Quote the passage you used and name the file (and page) it came from.
If the tool says NO RELEVANT PASSAGES, or the passages don't answer it, say so instead of guessing.
```

Check it: the answer names the right file and page. Then ask something that's NOT in your files. It should say it can't find it. If it makes something up, it isn't using the tool: tell it to.

Keep it fresh: after you add, edit or delete files, run ingest.py again. It embeds only what changed and removes what's gone; a running server picks the change up on the next question.

Copy and paste

```
python ingest.py
```

Under the hood

Six stages. The names are the ones you'd use in an interview.

1. Chunk. A model can't search a whole file at once, so each file is cut into passages of up to 800 characters (about 200 tokens of plain English). Notes are cut at their headings, and each passage carries its heading path ("Gym split > Rules") so it still makes sense alone. PDFs are cut page by page, which is why a citation can name the page. Each chunk's id is a hash of its file, page and text: same text, same id. That is what makes re-ingest incremental: unchanged chunks are skipped, changed ones are added, and ids that no longer exist are deleted.
2. Embed. Each chunk goes through an embedding model, MongoDB/mdbr-leaf-ir (23 million parameters, Apache 2.0, published September 2025), which turns it into 768 numbers. Passages that mean similar things get similar numbers. It runs on your CPU through onnxruntime (no GPU, no PyTorch); on my desktop it embeds about 31 chunks a second. The model reads at most 512 tokens of a chunk, so chunks are kept well under that. The vectors, text and metadata are stored in Chroma, a free database that lives in the brain_db/ folder.
3. Hybrid retrieve. The question is searched two ways. Vector search embeds the question and compares it with every chunk (cosine similarity), which finds passages that mean the same thing in different words ("signed out when they reloaded" finds "logged out on every refresh"). Keyword search (BM25) scores chunks by the exact words they share with the question, weighting rare words more, which is what finds

PGRST301 or invoice INV-2291. Embeddings are weak on strings like those; BM25 is blind to paraphrase. Each covers the other's hole.

4. Fuse. The two lists are merged with reciprocal rank fusion: a passage scores 1/(60 + its rank) in each list and the scores are added. It only uses ranks, so it doesn't matter that a similarity (0 to 1) and a BM25 score (any size) aren't on the same scale. A passage near the top of both lists wins.
5. Floor. A passage whose similarity to the question is under 0.35 is dropped, unless it contains an exact identifier from the question. If nothing is left, the tool returns "NO RELEVANT PASSAGES" and tells the model to say so. Without this step a search always returns its 5 "closest" passages, even to "what is the capital of Mongolia?", and the model is tempted to build an answer on them.
6. Cite. Every passage comes back with its file, page, chunk id, similarity and its rank in each search, like [1] lease.pdf, page 3 | chunk 2a04633acca1f167 | similarity 0.51 | vector #1, keyword #2. Claude Code writes the answer from those and names the file and page.

Two decisions came out of measuring, not out of a tutorial:

- Exact search, not the database's approximate index. Vector databases answer fast by searching an approximate index (HNSW). On my 109-file test, Chroma's index at its default settings returned the true best passage for only 19 of 27 questions; the small cluster of personal notes was hard to reach among 2,000 other chunks. Comparing the question with every vector in memory is exact and took half a millisecond at 2,200 chunks (8 ms at 50,000 random vectors), so that is what rag.py does. Chroma is the store; the search is exact. Approximate indexes earn their place at millions of vectors, not at notes-folder size.
- A 2025 embedding model, not the default. Chroma's built-in model is all-MiniLM-L6-v2, which predates 2023. Same size class, similar speed, but on my stress test it put the right passage first for 73% of questions against 82% for mdbr-leaf-ir (77% against 95% with hybrid). The table is in "Tested on".

Why it's built this way (sources)

Every choice below comes from a source I opened or from the eval in "Tested on", and each line says which. Where the research doesn't settle a number, it's my tuned default, not a finding. "Origin" marks the older paper a technique comes from.

Retrieve, then generate. Current overview: Wampler, Nielson, Seddighi, "Engineering the RAG Stack" (arXiv, 2026), a review of RAG systems 2018 to 2025. Origin: Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (2020)

- **Engineering the RAG Stack (2026)** Wampler, Nielson, Seddighi · arXiv
  `https://arxiv.org/abs/2601.05264`
- **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks (2020)** Lewis et al. · arXiv
  `https://arxiv.org/abs/2005.11401`

Hybrid search (keywords + vectors). Akarsu, Karaman, Mierbach, "From BM25 to Corrective RAG" (arXiv, Apr 2026), 23,088 questions over 7,318 documents: BM25 alone beat a state-of-the-art dense model (Recall@5 0.644 against 0.587) and hybrid beat both (0.695). My eval agrees (hit@3 0.91 to 1.00). Origin of BM25: Robertson and Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond" (2009), which gives the usual parameter ranges (k1 between 1.2 and 2, b between 0.5 and 0.8); the rank_bm25 defaults (1.5 and 0.75) sit inside them

- **From BM25 to Corrective RAG (2026)** Akarsu, Karaman, Mierbach · arXiv
  https://arxiv.org/abs/2604.01733
- **The Probabilistic Relevance Framework: BM25 and Beyond (2009)** Robertson and Zaragoza · PDF
  https://www.staff.city.ac.uk/~sbrp622/papers/foundations_bm25_review.pdf

Reciprocal rank fusion, k = 60. The 2025 review above describes RRF with k = 60 as the standard way to combine result lists. Origin: Cormack, Clarke and Büttcher (SIGIR 2009), where k = 60 was fixed in a pilot study and later found to be close to the best value and not a sensitive choice. The 2026 benchmark above found k = 10 slightly better than 60 on its data; on mine 10 and 60 scored the same, so I kept the standard.

- **Reciprocal Rank Fusion (SIGIR 2009)** Cormack, Clarke and Büttcher · PDF
  https://cormack.uwaterloo.ca/cormacksigir09-rrf.pdf

The embedding model. Vujanic and Rueckstiess, "LEAF" (arXiv, Sep 2025, revised Apr 2026), and the model card, which reports BEIR nDCG@10 of 53.55 against 41.95 for all-MiniLM-L6-v2 at the same 23M parameters. I picked it on my own eval, not on the card (see "Tested on"). Origin of this kind of model: Reimers and Gurevych, "Sentence-BERT" (2019)

- **LEAF (2025)** Vujanic and Rueckstiess · arXiv
  https://arxiv.org/abs/2509.12539
- **MongoDB/mdbr-leaf-ir model card** Hugging Face
  https://huggingface.co/MongoDB/mdbr-leaf-ir
- **Sentence-BERT (2019)** Reimers and Gurevych · arXiv
  https://arxiv.org/abs/1908.10084

Chunk size, about 200 tokens, no overlap. Bennani and Moslonka, "A Systematic Analysis of Chunking Strategies for Reliable Question Answering" (arXiv, Jan 2026): overlap gave no measurable gain and only raised indexing cost, sentence-level chunks were the best value, and 150 to 300 tokens is its recommended default. Bhat et al., "Rethinking Chunk Size for Long-Document Retrieval" (arXiv, May 2025): small chunks suit short factual answers, and the best size depends on the data and the model. My eval: 400, 800 and 1,200 characters scored the same with hybrid search, and so did 150 characters of overlap against none, so I took the papers' defaults.

- **A Systematic Analysis of Chunking Strategies for Reliable Question Answering (2026)** Bennani and Moslonka · arXiv
  https://arxiv.org/abs/2601.14123
- **Rethinking Chunk Size for Long-Document Retrieval (2025)** Bhat et al. · arXiv
  https://arxiv.org/abs/2505.21700

Five passages back. The same 2026 chunking paper found answer quality stops improving, then drops, as more context is added, with about 2,500 tokens a strong default. Five chunks of up to 200 tokens is about 1,000. In my eval the right passage was in the top 3 for every question, so 5 leaves room. Tuned default: raise TOP_K if your hit@3 is low.

The relevance floor. Joren et al., "Sufficient Context" (arXiv, Nov 2024, revised Apr 2025): strong models often give a wrong answer, where they should decline, when the retrieved context doesn't contain the answer. That is the reason to return nothing instead of weak passages. The number 0.35 and the exact-identifier exception are mine, tuned on the eval; they are not research findings, and yours may differ, which is why evals.py prints the floor table.

- **Sufficient Context (2024)** Joren et al. · arXiv
  `https://arxiv.org/abs/2411.06037`

Measuring with hit@k and MRR. Gan et al., "Retrieval Augmented Generation Evaluation in the Era of Large Language Models: A Comprehensive Survey" (arXiv, Apr 2025) defines both and reports that these traditional metrics still dominate RAG evaluation

- **Retrieval Augmented Generation Evaluation in the Era of Large Language Models (2025)** Gan et al. · arXiv
  `https://arxiv.org/abs/2504.14891`

What I left out: a reranker. The 2026 benchmark above got its best result by reranking the fused list with a second model (Recall@5 0.695 to 0.816). It used a paid API. A local reranker needs PyTorch; my test environment with it came to 1.4 GB, for a search that already put the right passage in the top 3 on every test question. Add one when your eval says you need it.

Docs. Chroma's docs for its default model and distance settings, the MCP Python SDK, and Claude Code's MCP docs (claude mcp add, scopes). All read on Nov 15, 2026.

- **Chroma: embedding functions** Its default model
  `https://docs.trychroma.com/docs/embeddings/embedding-functions`
- **Chroma: configure a collection** Its distance settings
  `https://docs.trychroma.com/docs/collections/configure`
- **MCP Python SDK** GitHub
  `https://github.com/modelcontextprotocol/python-sdk`
- **Claude Code: MCP** claude mcp add, scopes
  `https://code.claude.com/docs/en/mcp`

What it's bad at (so you don't think it's broken)

It finds the 5 passages closest to your question. So it's great at "what did I write about X?" and bad at questions about all your files at once: "summarize everything", "how many times did I…", "what changed this week". For those, point Claude Code at the folder itself.

Three more limits, measured:

- The floor catches off-topic questions, not near misses. In my test it refused 4 of 5 questions the notes don't answer. The one it let through, "what did I decide about switching to Firebase?", scored 0.36 because the notes do talk about hosting decisions, just not that one. There the passages come back and Claude has to read them and say "that's not in here". The step 6 prompt asks for exactly that.
- It can refuse a real question. One of my 22 answerable questions ("who pays the water bill?", similarity 0.35) fell just under the floor. If it says NO RELEVANT PASSAGES about something you know is there, ask again in the words the note uses, or lower SIM_FLOOR.
- A passage that runs across two PDF pages is two chunks, and the model is English-only.

When it breaks

- No module named 'mcp.server.fastmcp': mcp 2.x renamed FastMCP to MCPServer. Paste the error into Claude Code and tell it: "use from mcp.server.mcpserver import MCPServer". (Or pin mcp<2 and keep the old code.)
- claude mcp list says Failed to connect: use full paths in step 5, not python or ./brain_server.py, and make sure rag.py builds its paths from its own folder. Claude Code starts the server from a different folder. Also check nothing in brain_server.py prints: stdout is the MCP channel.
- A PDF shows up under "No text found (scanned?)": it's a scan. Use the read-it-for-me prompt in step 2.
- The first run looks frozen: it's downloading the 92 MB model, once. Wait. If the download is blocked (work network), run it once on another connection; after that it works offline.
- hit@3 is low in step 3: look at the questions evals.py lists. Wrong file entirely: check that file was ingested and isn't a scan. Right file, wrong passage: the note is one long block, so add headings to it. Exact codes missed: check the keyword tokens keep PGRST301 and INV-2291 whole.
- It says NO RELEVANT PASSAGES too often: lower SIM_FLOOR in rag.py to the row of the floor table you prefer.
- You changed the embedding model: delete brain_db/ and run ingest.py again. Vectors from two models don't mix.
- Claude Code's code doesn't behave like the checks say: paste the matching file from the appendix and say "use this instead".

Tested on

Windows 11, Python 3.13.15, an Intel i7-11700F (CPU only), on Nov 14 and 15, 2026. chromadb 1.5.9, pypdf 7.1.0, mcp 2.3.0, rank-bm25 0.2.2, onnxruntime 1.32.0, tokenizers 0.23.3, huggingface_hub 2.4.0. The numbers come from the code in the appendix, run in a fresh folder outside any project.

The sample test: 10 notes (bug write-ups, decisions, a gym split, a client call), a 4-page text PDF (a lease) and a scanned, image-only PDF (an insurance summary). 39 chunks.

- Ingest: "Stored 38 chunks from 11 files", the scan flagged. After Claude Code typed the scan out: "added 1, unchanged 38". Third run: "added 0, unchanged 39, removed 0" in 2.4 seconds. The first run, with an empty model cache, took 14 seconds including the 92 MB download.
- Eval, 22 answerable questions (8 with an exact code or number) and 5 the notes don't answer: vector-only and hybrid both hit@1 1.00, hit@3 1.00, MRR 1.000. Too small a pile to separate them, which is why there's a stress test.
- Floor at 0.35: 21 of 22 answerable questions kept the right passage in the top 3; 4 of 5 no-answer questions were refused. At 0.30 it's 22 of 22 and 2 of 5; at 0.40, 21 of 22 and 5 of 5.
- MCP: the server, started from my home folder by a client over stdio, listed one tool and answered "how much does it cost to break my lease early?" with lease.pdf, page 3, answered "what is my collision deductible?" from the typed-out scan, and returned NO RELEVANT PASSAGES for "what is the capital of Mongolia?". With the server still running I added a note, edited it and deleted it, re-running ingest each time; the next tool call saw each change.

The stress test: the same files and the same 22 questions, plus 97 open-source package READMEs as noise (109 files, 2,191 chunks; 2,350 with overlap, which the model comparison used).

- Ingest 70 seconds (about 31 chunks a second), re-ingest with nothing changed 2.5 seconds, search 5 ms a question.
- Vector-only: hit@1 0.86, hit@3 0.91, MRR 0.890. Hybrid: hit@1 0.95, hit@3 1.00, MRR 0.977. Hybrid fixed "PGRST301" (vector rank 4, hybrid rank 1) and "what happened with evt_3QK2?" (not in vector's top 10, hybrid rank 2).
- Embedding models, same chunks, same questions, exact search (hit@1 / hit@3 / MRR, vector-only then hybrid):

Model comparison

model	download	vector-only	hybrid
all-MiniLM-L6-v2 (Chroma's)	79 MB	0.73 / 0.77 / 0.742	0.77 / 0.95 / 0.875
bge-small-en-v1.5	129 MB	0.68 / 0.73 / 0.717	0.82 / 0.95 / 0.886
snowflake-arctic-embed-s	128 MB	0.91 / 0.95 / 0.932	1.00 / 1.00 / 1.000
mdbr-leaf-ir (used here)	92 MB	0.82 / 0.95 / 0.879	0.95 / 1.00 / 0.977

arctic-embed-s scored one or two questions higher. I still chose mdbr-leaf-ir: it's the 2025 model, it embedded about twice as fast in a like-for-like run (47 chunks a second against 22), it's the smaller download, and its similarity scores separate on-topic from off-topic questions, where arctic's don't (its off-topic questions scored 0.60 to 0.65, above some real ones at 0.52), so the relevance floor wouldn't work with it. With 22 questions, a gap of one question is not proof either way.

- Chroma's approximate index at default settings found the true nearest passage for 19 of 27 questions; with ef_search raised to 500, 26 of 27; exact search, 27 of 27 in 0.5 ms.
- Chunk size 400 / 800 / 1,200 characters, overlap 0 or 150, and fusion constant 10 or 60: hybrid results identical.

Not tested by me in this run: claude mcp add and claude mcp list (they change Claude Code's settings, so I checked the syntax against the current docs instead; they were tested on Linux on Nov 13 with Claude Code 2.1, showing Connected); a Mac; Python versions other than 3.13; and the prompts themselves, which describe the appendix code but weren't run through Claude Code end to end. Your Claude Code may write the files a little differently, which is what the "Check it" lines and step 3 are for.

Appendix: the code I tested

Six files, about 410 lines. These are the exact files behind the numbers above.

rag.py

```
"""Second brain: the retrieval pipeline. chunk -> embed -> hybrid retrieve -> fuse -> floor -> cite.

Everything runs locally. ingest.py, search.py, evals.py and brain_server.py all import from here,
so the eval measures exactly the code the MCP tool runs.
"""
import hashlib
import re
from pathlib import Path

import chromadb
import numpy as np
import onnxruntime
from huggingface_hub import hf_hub_download
from rank_bm25 import BM25Okapi
from tokenizers import Tokenizer

ROOT = Path(__file__).parent          # paths come from this file, not the folder it's launched from
NOTES = ROOT / "notes"
DB = ROOT / "brain_db"
STAMP = DB / "last_ingest.txt"        # ingest touches this; a running server reloads when it changes

CHUNK_CHARS = 800      # largest chunk; about 200 tokens of plain English
OVERLAP_CHARS = 0      # characters repeated between neighbouring chunks (0: overlap bought nothing in
our eval)
CANDIDATES = 20        # how many each retriever contributes before fusion
TOP_K = 5              # how many passages the tool returns
RRF_K = 60             # reciprocal rank fusion constant
SIM_FLOOR = 0.35       # cosine similarity below this is "not about the question" (tuned with evals.py)

STOPWORDS = set("""a about after all also am an and any are as at be been before but by can did do does
for from had
has have how i if in into is it its me my no not of on or our so than that the their them then there
these they this
to up was we were what when where which who why will with would you your""".split())

MODEL = "MongoDB/mdbr-leaf-ir"        # 23M parameters, 768 dimensions, Apache 2.0; its ONNX export runs
on the CPU
QUERY_PREFIX = "Represent this sentence for searching relevant passages: "   # the model's own query
prompt
MAX_TOKENS = 512                      # the model's limit; longer text is cut off for the vector (BM25
sees all of it)
```

```
def _model_file(name):
    try:                                            # after the first run this never touches the
network
        return hf_hub_download(MODEL, name, local_files_only=True)
    except Exception:
        return hf_hub_download(MODEL, name)          # first run: about 92 MB, once, no account or key


class Embedder:
    """Text -> unit-length vectors, with onnxruntime and tokenizers (both already installed with
chromadb)."""

    def __init__(self):
        self.session = self.tokenizer = None

    def _load(self):
        _model_file("onnx/model.onnx_data")          # the weights; model.onnx points at this file
        self.session = onnxruntime.InferenceSession(_model_file("onnx/model.onnx"), providers=
["CPUExecutionProvider"])
        self.tokenizer = Tokenizer.from_file(_model_file("tokenizer.json"))
        self.tokenizer.enable_truncation(MAX_TOKENS)
        self.tokenizer.enable_padding()

    def __call__(self, texts, query=False):
        if self.session is None:
            self._load()
        out = []
        for i in range(0, len(texts), 16):
            batch = [(QUERY_PREFIX + t) if query else t for t in texts[i:i + 16]]
            enc = self.tokenizer.encode_batch(batch)
            ids = np.array([e.ids for e in enc], dtype=np.int64)
            feed = {"input_ids": ids, "attention_mask": np.array([e.attention_mask for e in enc],
dtype=np.int64),
                    "token_type_ids": np.zeros_like(ids)}
            out.append(self.session.run(["sentence_embedding"], feed)[0])   # mean-pooled, projected,
normalized
        return np.vstack(out).astype(np.float32)


EMBED = Embedder()


def collection():
    client = chromadb.PersistentClient(path=str(DB))
    return client.get_or_create_collection(         # Chroma is the store: ids, text, metadata and our
own vectors.
        "notes", embedding_function=None, configuration={"hnsw": {"space": "cosine"}})   # Its built-in
model is unused.


# ---------- chunking ----------

def _pack(text, size=CHUNK_CHARS, overlap=OVERLAP_CHARS):
    """Pack whole paragraphs into chunks of up to `size` characters. With overlap > 0, each chunk after
```

```
    the first starts with the last `overlap` characters of the one before it (cut at a word boundary)."""
    paras = [p.strip() for p in re.split(r"\n\s*\n", text) if p.strip()]
    pieces = []
    for p in paras:                                  # a single paragraph longer than `size` is cut on
sentences
        while len(p) > size:
            cut = p.rfind(". ", 0, size)
            cut = cut + 1 if cut > size // 2 else size
            pieces.append(p[:cut].strip())
            p = p[cut:].strip()
        if p:
            pieces.append(p)
    chunks, cur = [], ""
    for p in pieces:
        if cur and len(cur) + len(p) + 2 > size:
            chunks.append(cur)
            tail = cur[-overlap:] if overlap else ""
            tail = tail[tail.find(" ") + 1:] if " " in tail else tail
            cur = (tail + "\n\n" + p) if tail else p
        else:
            cur = (cur + "\n\n" + p) if cur else p
    if cur:
        chunks.append(cur)
    return chunks


def _markdown_sections(text):
    """Yield (heading path, body) for each heading section, e.g. ('Gym split > Rules', '...')."""
    path, body, out = [], [], []
    for line in text.splitlines():
        m = re.match(r"^(#{1,6})\s+(.*\S)\s*$", line)
        if m:
            if "".join(body).strip():
                out.append((" > ".join(h for _, h in path), "\n".join(body)))
            level = len(m.group(1))
            path = [(l, h) for l, h in path if l < level] + [(level, m.group(2))]
            body = []
        else:
            body.append(line)
    if "".join(body).strip():
        out.append((" > ".join(h for _, h in path), "\n".join(body)))
    return out


def _pdf_pages(path):
    from pypdf import PdfReader
    return [(i + 1, (page.extract_text() or "").strip()) for i, page in
enumerate(PdfReader(str(path)).pages)]


def chunk_file(path):
    """Return [(chunk_id, text, metadata)] for one file. An empty list for a PDF means no text layer (a
scan)."""
    source = path.relative_to(NOTES).as_posix()
    parts = []                                       # (page, heading, text)
    if path.suffix.lower() == ".pdf":
```

```
        for page, text in _pdf_pages(path):          # chunked per page, so a citation's page is exact
            parts += [(page, "", c) for c in _pack(text)]
    else:
        text = path.read_text(encoding="utf-8", errors="replace")
        sections = _markdown_sections(text) if path.suffix.lower() == ".md" else [("", text)]
        for heading, body in sections:
            parts += [(0, heading, c) for c in _pack(body)]
    out, seen = [], {}
    for n, (page, heading, text) in enumerate(parts):
        doc = f"{heading}\n{text}" if heading else text      # the heading path travels with every chunk
        digest = hashlib.sha256(f"{source}|{page}|{doc}".encode("utf-8")).hexdigest()[:16]
        seen[digest] = seen.get(digest, 0) + 1
        cid = digest if seen[digest] == 1 else f"{digest}-{seen[digest]}"   # same text twice in one file
        out.append((cid, doc, {"source": source, "page": page, "heading": heading, "order": n}))
    return out


# ---------- retrieval ----------

def tokens(text):
    """Lowercase words for BM25. A hyphenated term is kept whole and in parts: INV-2291 -> inv, 2291,
inv-2291."""
    out = []
    for word in re.findall(r"[a-z0-9_]+(?:-[a-z0-9_]+)*", text.lower()):
        parts = word.split("-")
        out += parts + ([word] if len(parts) > 1 else [])
    return [t for t in out if t not in STOPWORDS]


class Index:
    """Every chunk's vector in memory (exact search) plus a BM25 keyword index over the same chunks."""

    def __init__(self):
        self.stamp = None
        self.refresh()

    def refresh(self):
        stamp = STAMP.stat().st_mtime_ns if STAMP.exists() else 0
        if stamp == self.stamp:
            return
        chromadb.api.client.SharedSystemClient.clear_system_cache()   # reopen: another process re-
ingested
        self.col = collection()
        got = self.col.get(include=["documents", "metadatas", "embeddings"])
        self.ids, self.docs, self.metas = got["ids"], got["documents"], got["metadatas"]
        self.vecs = np.asarray(got["embeddings"], dtype=np.float32).reshape(len(self.ids), -1)   # unit
length
        self.row = {cid: i for i, cid in enumerate(self.ids)}
        self.toks = [set(tokens(d)) for d in self.docs]
        self.bm25 = BM25Okapi([tokens(d) or ["_"] for d in self.docs]) if self.docs else None
        self.stamp = stamp

    def search(self, question, mode="hybrid", top_k=TOP_K, floor=SIM_FLOOR):
        """Return (hits, best_similarity). mode is 'vector' or 'hybrid'. floor=None turns the floor
off."""
        self.refresh()
```

```
        if not self.docs:
            return [], 0.0
        qvec = EMBED([question], query=True)[0]
        n = min(CANDIDATES, len(self.ids))
        sims = self.vecs @ qvec                     # exact cosine similarity against every chunk (see
the guide:
        order = np.argsort(-sims)[:n]               # Chroma's approximate index missed passages this
finds)
        vec_rank = {self.ids[i]: r for r, i in enumerate(order, 1)}

        kw_rank, kw_score = {}, {}
        qtok = tokens(question)
        if mode == "hybrid" and qtok:
            scores = self.bm25.get_scores(qtok)
            order = [i for i in np.argsort(scores)[::-1][:n] if self.toks[i] & set(qtok)]
            kw_rank = {self.ids[i]: r for r, i in enumerate(order, 1)}
            kw_score = {self.ids[i]: float(scores[i]) for i in order}

        fused = {}
        for ranks in (vec_rank, kw_rank):                       # reciprocal rank fusion
            for cid, r in ranks.items():
                fused[cid] = fused.get(cid, 0.0) + 1.0 / (RRF_K + r)

        sim = {cid: float(sims[self.row[cid]]) for cid in fused}
        best = float(sims.max())
        hits = []
        # with two rankers RRF ties are common (#1 in one list only = #1 in the other only): break them
on similarity
        for cid in sorted(fused, key=lambda c: (-round(fused[c], 9), -sim[c], c)):
            exact = self._exact_terms(qtok, cid) if mode == "hybrid" else []
            if floor is not None and sim[cid] < floor and not exact:
                continue                                        # the relevance floor
            m = self.metas[self.row[cid]]
            hits.append({"id": cid, "source": m["source"], "page": m["page"], "heading": m["heading"],
                         "text": self.docs[self.row[cid]], "similarity": round(float(sim[cid]), 3),
                         "rrf": round(fused[cid], 5), "vector_rank": vec_rank.get(cid),
                         "keyword_rank": kw_rank.get(cid), "bm25": round(kw_score.get(cid, 0.0), 2),
                         "exact_terms": exact})
            if len(hits) == top_k:
                break
        return hits, round(float(best), 3)

    def _exact_terms(self, qtok, cid):
        """Query terms that look like identifiers (they contain a digit or an underscore: PGRST301,
        refresh_token_not_found, C-38) and appear in this chunk. An exact identifier match passes the
        floor even when the embedding similarity is low, because embeddings are weak on such strings."""
        have = self.toks[self.row[cid]]
        return [t for t in dict.fromkeys(qtok) if t in have and re.search(r"[0-9_]", t) and len(t) >= 3]


# ---------- output ----------

def cite(hit):
    where = hit["source"] + (f", page {hit['page']}" if hit["page"] else "")
    if hit["heading"]:
        where += f" ({hit['heading']})"
```

```
    return where


def format_hits(question, hits, best, floor=SIM_FLOOR):
    if not hits:
        return (f"NO RELEVANT PASSAGES. Nothing in the notes is close enough to this question "
                f"(best similarity {best:.2f}, floor {floor:.2f}, no exact identifier match). "
                f"Tell the user their notes don't cover it; do not answer from general knowledge. "
                f"If they expect it to be there, search once more using the words the note would use.")
    out = []
    for n, h in enumerate(hits, 1):
        ranks = f"vector #{h['vector_rank'] or '-'}, keyword #{h['keyword_rank'] or '-'}"
        out.append(f"[{n}] {cite(h)} | chunk {h['id']} | similarity {h['similarity']:.2f} |
{ranks}\n{h['text']}")
    return "\n\n".join(out)
```

ingest.py

```
"""Load notes/ into the store. Incremental: unchanged chunks are skipped, stale chunks are deleted."""
import time

from rag import DB, EMBED, NOTES, STAMP, chunk_file, collection


def main():
    col = collection()
    files = sorted(p for p in NOTES.rglob("*") if p.is_file() and p.suffix.lower() in (".md", ".txt",
".pdf"))
    new, scanned, loaded = {}, [], 0
    for path in files:
        chunks = chunk_file(path)
        if not chunks and path.suffix.lower() == ".pdf":
            scanned.append(path.relative_to(NOTES).as_posix())
            continue
        loaded += 1
        for cid, doc, meta in chunks:
            new[cid] = (doc, meta)

    existing = set(col.get(include=[])["ids"])
    add = [cid for cid in new if cid not in existing]
    stale = sorted(existing - set(new))
    for i in range(0, len(add), 100):                       # embed and store only what's new
        batch = add[i:i + 100]
        texts = [new[c][0] for c in batch]
        col.add(ids=batch, embeddings=EMBED(texts), documents=texts, metadatas=[new[c][1] for c in
batch])
    if stale:
        col.delete(ids=stale)                               # chunks whose text no longer exists in notes/
    DB.mkdir(exist_ok=True)
    STAMP.write_text(str(time.time()))

    print(f"Stored {col.count()} chunks from {loaded} files "
          f"(added {len(add)}, unchanged {len(new) - len(add)}, removed {len(stale)})")
    if scanned:
        print("No text found (scanned?):")
        for s in scanned:
            print(f"  {s}")


if __name__ == "__main__":
    main()
```

search.py

```
"""Search from the command line:  python search.py "your question"  [--vector]"""
import sys

from rag import Index, format_hits

args = [a for a in sys.argv[1:] if a != "--vector"]
mode = "vector" if "--vector" in sys.argv else "hybrid"
question = " ".join(args)
hits, best = Index().search(question, mode=mode)
print(format_hits(question, hits, best))
```

evals.py

```
"""Measure retrieval. Reads evals.jsonl (question -> the file and page that should come back) and reports
hit@1, hit@3 and MRR for vector-only and hybrid search, then what the relevance floor costs and
catches."""
import json
from pathlib import Path

from rag import SIM_FLOOR, Index

lines = (Path(__file__).parent / "evals.jsonl").read_text(encoding="utf-8").splitlines()
rows = [json.loads(l) for l in lines if l.strip()]
answerable = [r for r in rows if r["file"]]
unanswerable = [r for r in rows if not r["file"]]
index = Index()


def rank_of(row, hits):
    for n, h in enumerate(hits, 1):
        if h["source"] == row["file"] and h["page"] == row.get("page", 0):
            return n
    return None


print(f"{len(answerable)} answerable questions, {len(unanswerable)} with no answer in the notes, "
      f"{len(index.ids)} chunks\n")
print(f"{'mode':<8}{'hit@1':>8}{'hit@3':>8}{'MRR':>8}")
ranks = {}
for mode in ("vector", "hybrid"):
    rr = ranks[mode] = [rank_of(r, index.search(r["q"], mode=mode, top_k=10, floor=None)[0]) for r in
answerable]
    hit1 = sum(1 for x in rr if x == 1) / len(rr)
    hit3 = sum(1 for x in rr if x and x <= 3) / len(rr)
    mrr = sum(1 / x for x in rr if x) / len(rr)
    print(f"{mode:<8}{hit1:>8.2f}{hit3:>8.2f}{mrr:>8.3f}")

print("\nWhere they differ (rank of the right passage; - means not in the top 10):")
for r, v, h in zip(answerable, ranks["vector"], ranks["hybrid"]):
    if v != h:
        print(f"  vector {v or '-':>2}  hybrid {h or '-':>2}  {r['q']}")

print("\nRelevance floor (hybrid; a passage needs this similarity, or an exact identifier match):")
print(f"{'floor':>7}{'right passage in top 3':>25}{'no-answer questions refused':>30}")
```

```
for floor in (0.15, 0.20, 0.25, 0.30, 0.35, 0.40):
    ok = sum(1 for r in answerable if (rank_of(r, index.search(r["q"], floor=floor)[0]) or 9) <= 3)
    refused = sum(1 for r in unanswerable if not index.search(r["q"], floor=floor)[0])
    mark = "   <- SIM_FLOOR in rag.py" if abs(floor - SIM_FLOOR) < 1e-9 else ""
    print(f"{floor:>7.2f}{f'{ok}/{len(answerable)}':>25}{f'{refused}/{len(unanswerable)}':>30}{mark}")

print(f"\nAt SIM_FLOOR = {SIM_FLOOR:.2f}:")
for r in answerable:
    hits, best = index.search(r["q"])
    if (rank_of(r, hits) or 9) > 3:
        print(f"  LOST      best similarity {best:.2f}  {r['q']}")
for r in unanswerable:
    hits, best = index.search(r["q"])
    print(f"  {'refused ' if not hits else 'ANSWERED'}  best similarity {best:.2f}  {r['q']}")
```

brain_server.py

```
"""MCP server: gives Claude Code one tool, search_notes, backed by the hybrid search in rag.py."""
from mcp.server.mcpserver import MCPServer

from rag import Index, format_hits

server = MCPServer("second-brain")
index = None


@server.tool()
def search_notes(question: str) -> str:
    """Search the user's own notes and PDFs (hybrid: meaning + exact keywords). Returns up to 5 passages,
    each with its file, page (for PDFs), chunk id and similarity score. Cite the file and page in your
answer.
    If it returns NO RELEVANT PASSAGES, tell the user their notes don't cover the question; don't guess.
    Include exact identifiers (error codes, invoice numbers, names) in the question when you have
them."""
    global index
    if index is None:
        index = Index()
    hits, best = index.search(question)
    return format_hits(question, hits, best)


if __name__ == "__main__":
    server.run()
```

test_server.py

```
"""Start brain_server.py the way Claude Code does (from another folder, over stdio) and call the tool."""
import asyncio
import sys
from pathlib import Path

from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

QUESTIONS = sys.argv[1:] or ["what is the capital of Mongolia?"]


async def main():
    here = Path(__file__).parent
    params = StdioServerParameters(command=sys.executable, args=[str(here / "brain_server.py")],
                                   cwd=str(Path.home()))
    async with stdio_client(params) as (read, write):
        async with ClientSession(read, write) as session:
            await session.initialize()
            tools = await session.list_tools()
            print("tools:", [t.name for t in tools.tools])
            for q in QUESTIONS:
                result = await session.call_tool("search_notes", {"question": q})
                print(f"\nQ: {q}\n{result.content[0].text[:420]}")


asyncio.run(main())
```

The line format of evals.jsonl:

evals.jsonl

```
{"q": "how much does it cost to break my lease early?", "file": "lease.pdf", "page": 3}
{"q": "PGRST301", "file": "bugs/2026-10-supabase-rls.md", "page": 0}
{"q": "what is the capital of Mongolia?", "file": null}
```

The page version, with a copy button on every box: murphmaxxing.com/vault/second-brain-notes-pdfs