A CRASH COURSE

Give your AI Searchable Context

Retrieval-augmented generation on Postgres with `pgvector` — built by directing your coding agent, not by hand-writing SQL.

15 concepts | 80% of real use | built by your agent

THE PROMISE

Ask your computer to understand meaning — and it does

Point it at a folder of documents. Tell it to build a search where asking for **"Empire State of Mind"** returns quotes about New York — **even the ones that never say the words "New York."**

Not a slideshow about vector databases. A running system you can query — and you direct an agent to build it.

search —▸

"Empire State of Mind"

0.91 "the city that never sleeps"

0.88 "concrete jungle where dreams are made"

0.84 "if I can make it there…"

↳ matched by meaning, not by words

THE IDEA THAT MAKES IT CLICK

RAG is context management, one layer down

YOUR CODING AGENT

A finite context window

+ Load the rules file at session start
+ Pull in only the files the task needs
+ Compact to drop stale content

so the model finds the signal

YOUR APP'S AI

A finite context window, too

+ pgvector retrieves the few chunks that match
+ The other million rows stay out
+ One source of truth, queried by meaning

so the LLM finds the signal

Same principle — **give it the right information, keep the irrelevant out** — now serving your users.

YOUR JOB IN IT

Direct the agent. Don’t write the SQL.

The agent already knows pgvector's operators and the embedding workflow cold. So **your value moves up the stack** —

from typing	~~CREATE INDEX~~
to deciding	**which index**
from writing	~~the query~~
to judging	**whether it's any good**

THE MINDSET SHIFT

Stop asking "what's the SQL for semantic search?"

Build me semantic search over this table. Here are my constraints. Show me the plan first.

You never open `psql`, write SQL, or run a migration yourself. Every SQL block ahead is shown so you can read and judge — not type.

WHAT THIS COURSE COVERS

Eight parts, one discipline

01 — 02

Foundations

The mindset, vectors from zero, the extensions, connecting your agent.

— Part 2 —

Your first RAG

Schema, embedding worker, chunking, semantic search, RAG.

— Part 3 —

Make search fast

When to index, which index, tuning recall.

— Part 4 —

Make search good

Evals, filters, hybrid search, multi-tenancy.

— Part 5 —

The full build

One task, empty database to working RAG.

— Part 6 —

Ship it as a tool

Wrap your RAG in an MCP server any agent can call.

— Part 7 —

Where it runs

Branching, the worker, production notes.

— Part 8 —

Hand it to an agent

Plug your RAG into an agent — the bridge onward.

PART ONE

01

Foundations

1. The AI-engineer mindset
2. Vectors & embeddings, from zero
3. The extensions that make it work
4. Connecting your agent

01 THE ROLE YOU'RE STEPPING INTO

You’re an AI engineer — not an ML researcher

The common misconception is that building AI applications needs a machine-learning team. It doesn't.

- → The **models** are off-the-shelf.
- → The **infrastructure** is a database you may already run.
- → What's left is **using** AI to build products.

THREE THINGS YOU MUST BE ABLE TO DO

01 Give the agent a **precise instruction.**

02 Read back what it produced and **spot when it's wrong.**

03 Decide the **architecture choices** it shouldn't make for you.

PLAN-THEN-EXECUTE, ENFORCED BY THE PLATFORM

One loop runs the whole course

1

Plan

Strong model, in plan mode. "Show me the plan first."

→

2

Review

Read it. Branch? pgvector on? Keys in env?

→

3

Execute

Cheaper model runs the routine build.

→

4

Evaluate

Re-run the eval set. Better or worse?

↻

5

Iterate

Fix the stage that's actually broken.

Master the loop and **the specific SQL stops mattering** — you can always have the agent produce it, and always tell whether it's right.

WHY RETRIEVAL QUALITY IS EVERYTHING

A system of record for the agent era

Agents don't remove the need for authoritative ground truth — they *depend* on one. Without it, an agent hallucinates. With it, it executes.

READS FROM
retrieval — the right chunks, on demand

WRITES TO
the worker keeps it current

VERIFIES AGAINST
your eval set holds the line

RAG on Postgres is how you hand an agent that ground truth — which is why this whole course treats retrieval quality as the thing that decides whether the agent can be trusted.

Vectors and embeddings, in one minute

`vector`

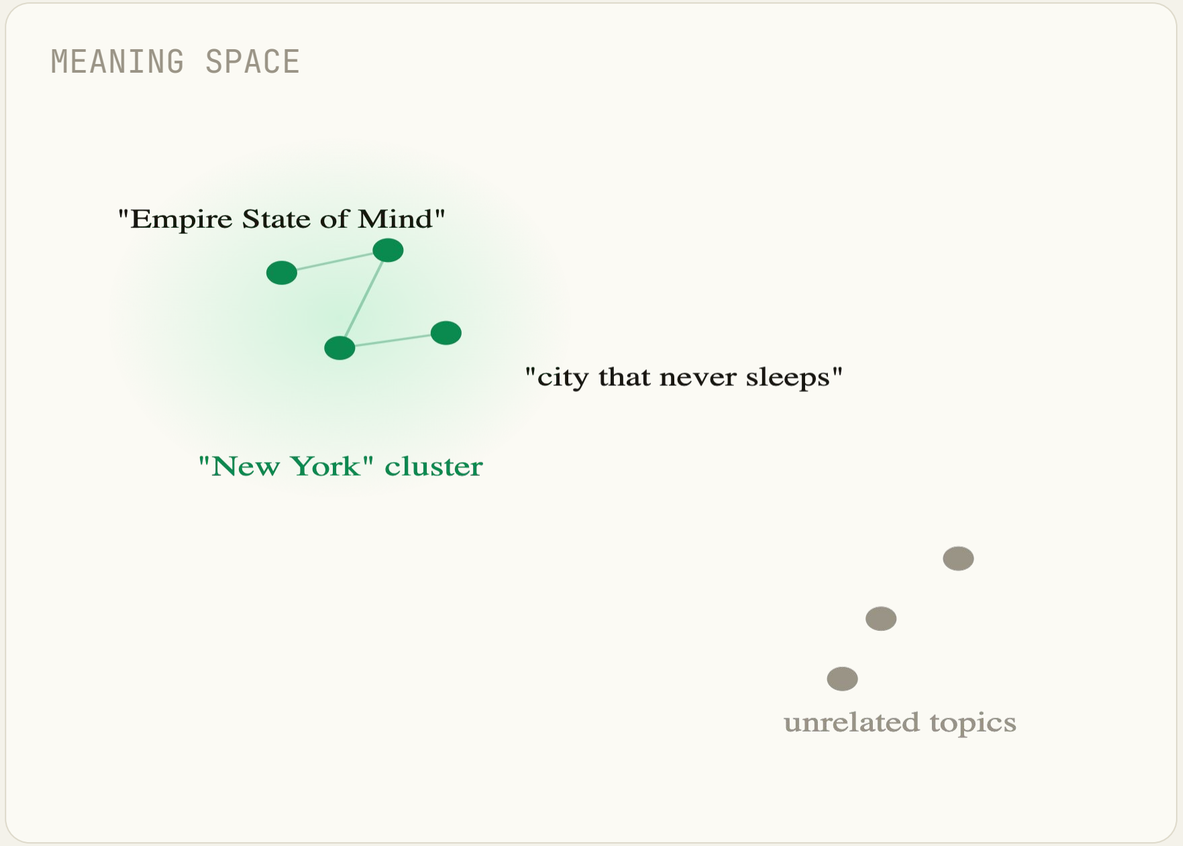
A list of numbers. `[0.021, -0.88, 0.14, …]`

`embedding model`

Turns a sentence, paragraph, or image into one of those lists — one that captures its **meaning.**

`the trick`

Two things that *mean* similar things get lists that sit **close together.**



A VECTOR DATABASE, IN ONE SENTENCE

Store the lists. Find the closest ones. Fast.

A USER ASKS A QUESTION

"what's it like in NYC?"

→

EMBED THEIR QUESTION

`[0.04, -0.7, …]`

→

ASK THE DATABASE

"which stored vectors are nearest?"

→

THE NEAREST ONES

are the most relevant context

That's how your app fetches exactly the right context to hand an LLM. You don't need to know **how** the model works — only that closeness equals similarity.

Two extensions, and a worker beside them

SIDE WORKER

Your embedding worker

Ordinary app code your agent writes — calls the embedding model, writes vectors back.

pgvectorscale `scale tier`

StreamingDiskANN index, vector compression, high-accuracy filtered search at large scale. The host you'd graduate to.

pgvector `always`

The `vector` type, distance operators, and the HNSW + IVFFlat indexes. Install this first.

PostgreSQL

Your tables and relational data — the single source of truth you already know.

pgvector is your whole stack. It ships built in and is all you need for a complete RAG engine.

pgvectorscale is the tier you graduate to only if you ever outgrow HNSW — not a gap, just the next rung.

THE ONE-DATABASE PAYOFF

The embeddings come from the worker — but vectors live **next to** the rows they describe.

THE WHOLE REASON FILTERED SEARCH IS EASY LATER

Vectors live next to your data

```
SELECT name, price, in_stock
FROM products
WHERE price < 2000 AND in_stock
ORDER BY embedding ⟺ $1 -- by meaning
LIMIT 5;

↳ similarity + filter, one query, one source of truth
```

A similarity search and a `WHERE price < 2000` filter happen in the **same query**, on the **same source of truth.**

- × No second database
- × No sync pipeline
- × No data drift

THE ONE THING PGVECTOR DOESN'T HAND YOU

The embedding worker is yours to build

Everything *after* embedding — search, indexing, evals, filters, hybrid, the MCP server — is pure pgvector. The one thing it doesn't do is **create** the embeddings.

The clean, portable way to do that is a small **worker** — a short program your agent writes that runs *off* the database.

Keeping it outside the database is the point, not a workaround: a system of record shouldn't depend on a volatile external API. Learn the pattern — it outlives any one package.

04 THE ONLY STEP YOU DO BY HAND

Connect your agent to the database

You → Coding agent → MCP server →

Postgres + pgvector

The agent operates the database through an **MCP server** — the same connector mechanism from the coding course. It creates the project, opens a branch, runs the SQL, and previews each migration **before** committing it.

You type prompts, review, and approve. You never open a console or a `psql` shell. Every action is the agent's — shown to you for review.

TRY THIS — IN PLAN MODE

Create a project and enable the `pgvector` extension. Then create a `dev` branch for us to build on, and read me its connection string (never print my API key). Show me the plan before you run anything.

What you check in the plan: it works on a branch, not production · it enables pgvector · the worker reads the API key from the environment.

PART TWO ·
BUILT BY YOUR
AGENT

02

Your first RAG

1. The schema — one source of truth
2. The embedding worker & the model choice
3. Chunking — the lever that sets your ceiling
4. Semantic search &
5. RAG

The schema: one source of truth

```
CREATE TABLE quotes (
id bigint GENERATED … PK
person text NOT NULL,
city text NOT NULL,
quote text NOT NULL
);

↳ embeddings join in a companion table next
```

Quotes by historical figures about US cities — soon searchable by **meaning.**

The mental model to hold: the quote, who said it, the city, and (soon) its meaning-vector all live in the **same database.**

TRY THIS

Create a `quotes` table with `person`, `city`, `quote`. Insert a handful of real quotes about New York, San Francisco, and Chicago so we have something to search.

06 THE ONE DECISION THAT'S ANNOYING TO REVERSE

Choosing an embedding model

Quality vs cost

Start small & cheap

A 1536-dim small model is the sensible default — plenty for most RAG. Reach for a larger one only if your evals show the lift.

Dimensions

More isn’t free

More storage, memory, and slightly slower search. A real gotcha: pgvector's index caps at `2,000` dims — `halfvec` extends it to 4,000.

Language & domain

Match your content

Non-English, or specialized (legal, medical, code)? A multilingual or domain-tuned model can beat a bigger general one.

In the worker, the model is a **single setting** — so you don't guess. Shortlist a couple of candidates, then let *your* evals pick the winner on *your* data.

TRY THIS

We're embedding English support docs, cost matters, stay under the index limit. Recommend a model and dimension count, explain the tradeoff, and set it as the worker's model.

RUN IT ONCE TO BACKFILL, THEN ON A SCHEDULE

The embedding worker pipeline

quotes
source text — the rows you already have

→

find rows
with no current embedding

→

chunk
split the text into pieces

→

embed
call the model · key from env

→

quotes_embedding
chunk + vector, FK back to source

runs OFF the database ⏱ same code backfills and stays in sync

Decoupled
The database never calls the embedding API itself. Your writes never block on a slow endpoint.

Stays in sync
Poll on a schedule, or a trigger *marks* a row dirty — the embedding still happens out-of-band.

Security boundary
The API key lives in the worker's environment — never in SQL, never committed.

07 QUIETLY THE MOST IMPORTANT STEP IN THE WORKER

Chunking: the lever that sets your ceiling

One long doc → one vector = a blurry average of everything it says. You retrieve near-misses.

Split into chunks, each with its own vector = focused meaning, precise retrieval.

RECALL CEILING

If the right answer never lands cleanly in a chunk, no model and no prompt can recover it. So you tune it first — against your evals.

Size

Too large spans topics; too small loses context. A few hundred tokens is a common start.

Overlap

10–20% keeps a straddling sentence from being orphaned. Some almost always helps.

Strategy

By character count, by structure (headings, paragraphs), or semantically. For structured docs, headings beat blind counts.

Semantic search: order by distance

```
SELECT person, city, quote
FROM quotes_embedding
ORDER BY embedding ⟺ $1 -- embedded in app code
LIMIT 5;
```

The phrase comes in as $1 — embedding it happens in app code, where you control the model, retries, and caching. The nearest-neighbor search stays in Postgres.

THE DISTANCE OPERATORS THAT MATTER

⟺	**Cosine**	Text-embedding similarity — **the default**
⟷	**L2 (Euclidean)**	When magnitude matters
<#>	**Inner product**	Certain models that recommend it

Search **"Empire State of Mind"** → the top results are quotes about New York, including ones that never contain the words.

RAG: retrieve, then generate

Question
embedded into a vector

→

IN POSTGRES
Retrieve
vector search returns the top-k chunks

→

IN YOUR APP
Build the prompt
system + chunks + question

→

IN YOUR APP
Generate
call the LLM

→

Grounded answer
from *your* data

Retrieval stays in Postgres (fast, filterable). **Generation lives in your app** (you control the prompt, model, retries, streaming). The split is the whole point.

WHEN THE ANSWER IS BAD

It's almost always retrieval — wrong chunks, too many, irrelevant ones. Which is exactly why Part 4 exists.

PART THREE · THE TUNING LAYER

10

Making search fast

1. You can ship a working system on Parts 1–2 alone.
   Come back here when search needs to be faster.

10

2. Why you need an index — and when you don't

11

3. Three indexes, tuning, and the failure modes

Why you need an index — and when you don’t

Exact search

Compares the query to **every** row. Perfectly accurate, perfectly fine while you're small. Gets slow as the table grows.

Approximate search

Trade a sliver of accuracy for a big speed-up by not checking every vector. A vector index makes that approximation good.

100k

Below ~100,000 vectors, **exact search is often fast enough — and always correct.**

The real threshold shifts with dimensions, latency target, filters, and churn — so have the agent benchmark before adding an index. Add it when searches actually get slow, not before.

Three indexes, one decision

IVFFlat

legacy

Low memory, faster than a full scan

Rebuilds when data changes · lower accuracy

Mostly superseded — reach for the next two

HNSW

your default

Strong speed/accuracy balance · updates without a rebuild · wide support

RAM-bound — cost scales with memory

~100k–10M vectors. **Start here.**

StreamingDiskANN

scale tier

Best accuracy with filters · scales past a billion · cost scales with disk

Longer first build

10M+ or filter-heavy workloads

< 100k	100k – 10M	10M+ / heavy filtering	don't guess — have your agent benchmark both and read you the latency
no index — exact is fine	HNSW	StreamingDiskANN	

THE ONE KNOB WORTH KNOWING

Tune `ef_search`, then confirm the index is doing its job

```
SET LOCAL hnsw.ef_search = 100;
-- higher → more recall, slower
-- lower → faster, less accurate
```

Tell the agent the recall you actually need and have it tune `ef_search` to hit it — **not blindly max it.**

✓ **Index Scan — the index is working**

```
Index Scan using …hnsw… Order By: (embedding ⟺ $1)

Execution Time: 0.8 ms
```

✕ **Seq Scan — every row scanned**

```
Seq Scan on quotes_embedding (rows=2000000)
Execution Time: 52.4 ms
```

Have the agent run `EXPLAIN ANALYZE`. A `Seq Scan` usually means the query's operator doesn't match the index.

YOUR JOB IS JUDGING THE AGENT'S WORK — SO WATCH FOR THESE

Five failure modes to catch in review

01	**Dimension mismatch**	the column's dimensions don't match the embedding model's output
02	**No index past the point you needed one**	a silent slowdown, not an error
03	**Operator / index mismatch**	query uses ⟷ but the index is cosine — so it's ignored
04	**Whole documents instead of chunks**	retrieval can never find anything precise
05	**Skipping `EXPLAIN ANALYZE`**	so nobody catches any of the above until users do

PART FOUR · WHERE
MOST PROJECTS
STALL

04

Making search good

1. A working RAG is not a *good* RAG. Knowing these moves separates you from someone who can only produce a demo.
2. Eval-driven development
3. Filtered search
4. Hybrid search
5. Multi-tenancy & text-to-SQL

Eval-driven development: start with the questions

Before writing anything, write down 10 questions your users will actually ask. **That file is your eval set** — your yardstick for whether a change made things better or worse.

Change the model, the chunking, a filter → re-run the set and *see* the effect, instead of guessing. As the app grows, the set grows with it.

TRY THIS

Run each eval question through our pipeline. Show the chunks retrieved, the final answer, and whether it matches. Summarize where it's failing — retrieval or generation.

WHEN AN ANSWER IS BAD, TRACE THE STAGES

Retrieval	did search even return the right chunks? (often an inventory gap)
Context	were the right chunks passed — or too many / too few?
Generation	given good context, did the model still answer poorly?

Nine times out of ten the failure is retrieval, not the LLM. Fix the stage that's actually broken.

Filtered search: the `WHERE` clause is your friend

PATTERN	USE CASE	THE ADDED CLAUSE
Metadata	Docs across products	`WHERE product = 'CRM'`
Composite	E-commerce recs	`… price BETWEEN 500 AND 2000`
Time	Recent articles only	`… published_at > now() - '7d'`
Permissions	See only what you're cleared for	`… clearance_level ⩽ $lvl`
Geospatial	"within 5 km" (PostGIS)	`ST_DWithin(location, $pt, 5000)`

Each is the same shape: `ORDER BY embedding` ⟺ `$1` with a `WHERE` in front. No second system, no juggling.

THE PERMISSIONS ONE MATTERS MOST

It's how you keep tenant A from ever retrieving tenant B's documents — enforced in the database, not hoped for in app code.

Hybrid search: meaning and keywords

Keyword
full-text search · nails the exact rare term — a product code, a name

Vector
understands paraphrase · but underweights exact terms

↓ over-fetch top 20 from each ↓

FUSE WITH RRF

Reciprocal Rank Fusion merges the two lists by **position, not score** — sidestepping the headache that keyword scores and cosine distances live on different scales.

A ROW THAT LANDS IN BOTH LISTS WINS

```
-- each row tagged with its rank in each list
SUM(1.0 / (60 + rank)) AS score
-- summed across keyword + vector
ORDER BY score DESC LIMIT 10;
```

Wins most on queries that mix a concept with a specific term. Whether it helps *your* corpus is a question only your eval set answers — so measure vector-only vs hybrid first.

15 TWO MORE YOU SHOULD RECOGNIZE

Multi-tenancy and text-to-SQL

MULTI-TENANCY · A LADDER OF ISOLATION

Shared table + `tenant_id`
weakest · cheapest · internal tools

Schema per tenant
good isolation · the sweet spot for most SaaS

Database per tenant
strongest · highest ops · regulated clients

Enforce the boundary in the database with Row-Level Security — write a policy once, and Postgres applies it to *every* query. A single forgotten `WHERE` can't leak one tenant's vectors.

TEXT-TO-SQL · ASK IN PLAIN ENGLISH

An agent translates **"what were Q3 sales by region?"** into correct SQL against your real tables.

What makes it accurate: a well-described schema — clear names, `COMMENT`s, and a few example question→query pairs.

Always review generated SQL before it runs. A wrong `SELECT` wastes a second; a wrong `UPDATE` ruins your afternoon.

05

A complete worked example

An empty database to a working, grounded Q&A over a folder of documents. The prompts are the whole job — identical in either tool.

THE WHOLE RHYTHM: PLAN → REVIEW → EXECUTE → EVALUATE → ITERATE

Four prompts, end to end

1 Plan first

`strong model · plan mode`

"I have a folder of markdown. Build a RAG system: create a project and a `dev` branch, enable pgvector, load the docs, build an embedding worker, and give me an `answer_question()`. **Show me the full plan and schema before running anything.** "

2 Read the plan

`you, not the agent`

Working on a **branch** ? pgvector enabled? Does the worker read the API key from the **environment** , kept out of the repo? Is generation in app code? **If all yes — approve.**

3 Execute

`cheaper model`

"Looks right. Proceed. Run the migration on the branch, start the worker, confirm the embeddings backfilled, then run these 5 questions and show me the retrieved chunks plus the answer for each."

4 Evaluate, then iterate

`the loop`

"Q3 and Q5 returned irrelevant chunks. Diagnose: retrieval or generation? If retrieval, try a different chunking strategy on a fresh branch and re-run the same 5 questions."

PART SIX · WRITE
THE
CAPABILITYONCE

06

Ship it as a tool

Wrap your RAG in an MCP server and the same retrieval becomes a tool *any* agent can discover and call.

YOU'VE NOW MET BOTH — THEY DO OPPOSITE JOBS

Two MCP servers — don’t confuse them

DEV-TIME ONLY

The admin MCP server

Lets your agent operate the database *while you build* — create branches, run SQL, preview migrations.

never for production or end users

YOUR PRODUCT SURFACE

The RAG MCP server

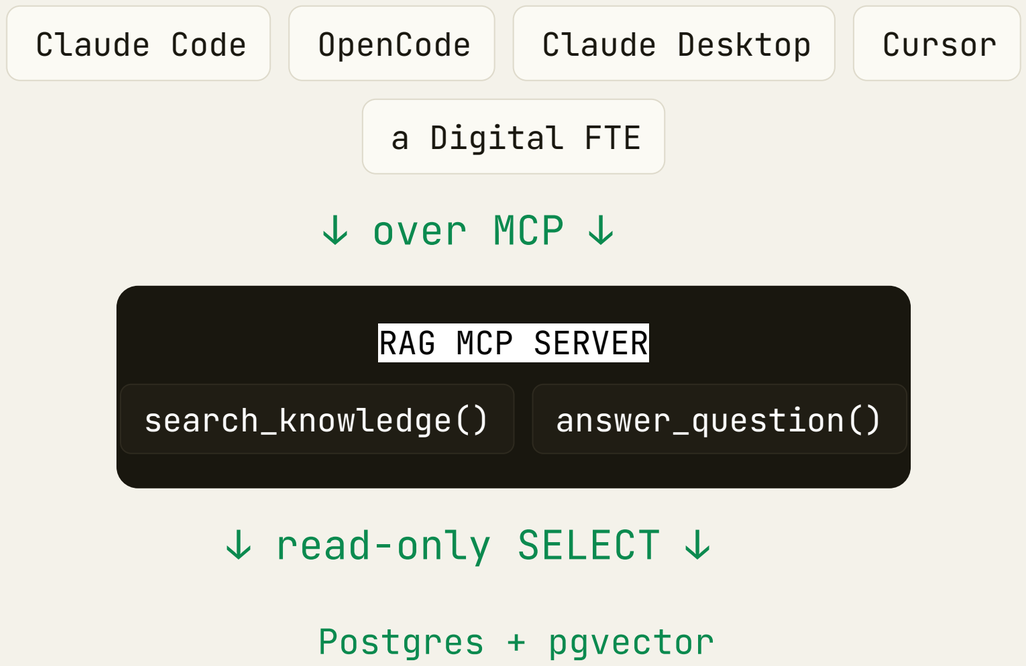
Exposes *read-only* retrieval — "search my knowledge," "answer from my data" — to whatever agent you point at it.

offered to agents at runtime

The first one **builds** the system. The second one **is** the system, offered to agents.

WRITE THE CAPABILITY ONCE; EVERY AGENT SPEAKS TO IT THE SAME WAY

One server, every agent



With FastMCP, each tool is just a typed function with a docstring — you write no JSON-RPC plumbing.

THE DOCSTRING IS THE INTERFACE

It's the text the calling agent reads to decide when to use the tool — so it must say plainly what it does and when to reach for it.

RETRIEVAL STAYS READ-ONLY

Run the parameterized search under a read-only role, so a tool argument can never mutate or leak your data.

For most applications, Postgres is the vector database

A dedicated store (Pinecone, Weaviate, Qdrant, Milvus) earns its place when you've *measured* a need it meets and Postgres can't. The trap is reaching for one by default. Decide it the way you decide indexes — let your evals and a real benchmark make the call.

Memory and knowledge: hand it to an agent

MEMORY · SESSIONS

What it recalls

the running conversation the agent reads and writes every turn

Agent

draws on both

KNOWLEDGE · POSTGRES + PGVECTOR

What it looks up

a corpus far too big to fit any window — `search_knowledge()` returns the top-k

An agent wired to both can **remember what was just said** and **look up what it never knew** — and the "fetch only the relevant chunks" discipline is what keeps that retrieved context from flooding the window.

Where this course ends — a callable `search_knowledge` — is where building agents begins. **Your RAG is one of its tools.**

THE THROUGHLINE
THAT NEVER CHANGES

Right information, right moment, irrelevant out.

You learned it for your agent. Now you can build it for everyone else's.

YOU NOW HAVE THE 80%

Go build it.

OPEN A SESSION · PLAN MODE · PASTE

Build a RAG system over my `docs/` folder: create a project and a `dev` branch, enable pgvector, load the docs, build an embedding worker, and give me an `answer_question()`. Show me the full plan and schema before running anything.

WHERE TO GO NEXT

Make it reliable
Eval-Driven Development

Make it an agent
Build AI Agents — the loop, guardrails, sessions

Make it a product
turn the assistant into a deployable Digital FTE

You can turn Postgres into a vector database, build a grounded RAG system, choose an index on evidence, and improve retrieval — **all by directing an agent and judging its work.**