EN

Practical Guide

OpenClaw + Databases
PostgreSQL, RAG and Semantic Search

Database integration for production AI agents

March 2026 — OpenClaw Series B10/10

Table of Contents

Flat files have their place. But when an agent network needs to read, write, and reason over structured data at scale — databases change the game. This guide closes the OpenClaw technical series: PostgreSQL, RAG, and semantic search.

1. Beyond Files: Why Connect to a Real Database

The Git workspace covers 80% of automation scenarios. But some situations demand more:

• Large volumes — 50,000 transactions aren't reasonably readable as Markdown

• Dynamic queries — filtering by date, status, revenue: SQL > grep

• Concurrency — multiple agents writing the same file = race conditions

• Semantic search — by meaning, not exact keyword

■ *Gartner 2025: 61% of production AI agent deployments need a structured persistence layer within the first 6 months.*

2. PostgreSQL + OpenClaw: Direct Connection

The most direct integration: a Python script in the workspace. The agent runs it via the exec skill, the script connects via psycopg2, returns results in a JSON file.

```
conn = psycopg2.connect(
host=os.environ['PG_HOST'],
dbname=os.environ['PG_DB'],
user=os.environ['PG_USER'],
password=os.environ['PG_PASS']
)
cur.execute("SELECT client, SUM(hours) FROM timesheets
WHERE month=%s GROUP BY client", ('2026-03',))
```

Absolute rule: credentials never transit through the Git workspace. Environment variables injected by vault only.

3. Concrete Use Cases

Automated Timesheets

The LEDGER agent runs a script every Friday that aggregates timesheets from PostgreSQL and generates a Markdown report. Zero human intervention.

CRM Updated by an Agent

After each client call logged in messaging, the NEXUS agent extracts key info and updates the corresponding row in the contacts table. The CRM stays current in real time.

Structured Operation Logs

Every significant agent action is logged in agent_logs (timestamp, source agent, type, status). Full audit — impossible to do reliably with text files.

4. RAG: Why It Changes Everything

An LLM has a limited context window. You can't inject 10,000 pages. RAG solves this: instead of injecting everything, retrieve only the most relevant passages for the question asked.

Simple architecture:

```
1. Indexing (once)
Documents -> Embeddings (1536-dim vectors) -> pgvector

2. Query (each question)
Question -> Embedding -> Similarity -> Top-K passages -> LLM
```

“Relevance is calculated by semantic similarity — closeness of meaning in vector space, not exact keyword matching.”

5. Implementing a Basic RAG

Generate Embeddings

```
def get_embedding(text: str) -> list[float]:
response = client.embeddings.create(
model='text-embedding-3-small',
input=text
)
return response.data[0].embedding
```

pgvector (Production)

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE documents (
id SERIAL PRIMARY KEY,
content TEXT,
embedding vector(1536)
);
CREATE INDEX ON documents USING hnsw (embedding vector_cosine_ops);
```

Similarity Query

```
SELECT content, 1-(embedding <=> %s::vector) AS score
FROM documents ORDER BY embedding <=> %s::vector LIMIT 5;
```

■ *For volumes < 10,000 docs: JSON + cosine in Python is sufficient. For production: pgvector.*

6. Semantic Search on Your Data

BOTUM use cases:

- Retrieve past decisions: 'what was the decision on backup architecture?' -> exact passage in meeting notes
- Find similar posts before publication
- Search a CRM: 'which clients mentioned security concerns?'

• Internal knowledge base: agents find the applicable standard procedure

7. Limits and Ground Truth

Embedding Quality

A corpus of cryptic notes produces useless embeddings. The quality of the indexing corpus determines result quality.

Cost

text-embedding-3-small: $0.02/million tokens. 10,000 docs x 500 tokens = ~$0.10 for initial indexing.

Latency

200-500ms per query (embedding API call + vector search). Negligible for an interactive agent.

When RAG Is Unnecessary

Data < 50,000 tokens: inject directly into context. Claude 3.5 handles 200K context tokens.

8. Recommended Architecture

Situation	Solution	Why
Data < 50K tokens	Direct injection	Zero latency
Structured data	PostgreSQL psycopg2	SQL > grep
< 10K docs, proto	JSON + cosine Python	Simple, 0 deps
10K-1M docs, Postgres	pgvector	Natural, performant
> 1M docs	Qdrant/Weaviate	Dedicated, scalable
Operation logs	PostgreSQL agent_logs	Audit, dashboards

9. DB Integration Checklist

■ Credentials via vault only — never in the Git workspace, never hardcoded

■ Dedicated PostgreSQL user per agent — minimal rights (SELECT/INSERT on required tables)

■ Connection pooling — pgBouncer or application pool in production

■ Query timeouts — always set statement_timeout

■ Parameterized queries only — never f-strings in SELECT statements

■ Results written to data/ — agent reads a file, not the DB connection directly

■ Index on frequent columns — avoid full scans on 100K-row tables

■ RAG similarity threshold — filter results with score > 0.65-0.7

■ Scheduled re-indexing, not real-time — batch at night/weekend

■ Regression testing — verify reference queries after each corpus modification

10. Series Conclusion

This tenth post marks the end of the OpenClaw technical series. Across ten episodes, BOTUM documented a real deployment: from initial installation through database integration and semantic search.

B1-B2: concept, installation. B3-B4: security, vault. B5: agent network. B6-B7: LLM selection. B8-B9: automation, memory. B10: structured persistence.

The next series will explore business dimensions: ROI metrics, SMB vs Enterprise, edge deployments. Articles at botum.ca/blog.

Full article: blog.botum.ca/openclaw-databases-postgresql-rag-semantic-search

Website: www.botum.ca • contact@botum.ca