Vector Databases for MCP & Agents — Deep Dive

Generated: 2025-09-11 07:06 UTC

Vector Databases for MCP & Agents — Deep Dive

Generated: 2025-09-11 06:58 UTC

Introduction

Why this matters: Vector databases are central to retrieval-augmented generation (RAG). They enable fast semantic search over embeddings produced by models, which is critical for providing LLMs with relevant context beyond small prompt windows. For MCP, vector DBs are typically exposed as *resources* or *search tools* so agents can query knowledge stores efficiently.

Suggested Interview Answer (short): Vector DBs store embeddings and support approximate nearest neighbor (ANN) search to retrieve semantically similar items; they are used in RAG pipelines to provide LLMs with context.

Core Concepts

Embeddings

Explanation: Transform text (or other modalities) into dense numeric vectors. Similar texts map to nearby vectors in high-dimensional space.

Code snippet (pseudo):
```
from embedding_model import embed
vec = embed("How do I reset my password?")
```

Indexes & ANN (Approximate Nearest Neighbors)

Explanation: Exact nearest neighbor search is expensive; ANN indexes (HNSW, IVF, PQ) trade tiny accuracy for large speed improvements.
Common engines: FAISS (local), Annoy, HNSWlib, Milvus, Pinecone, Chroma.

ASCII Diagram:
```
[Document Corpus] -> [Embedding Model] -> [Index (HNSW/IVF)] -> [Query Embedding] -> [ANN Search] -> [To
```

Retrieval Strategies

- *Nearest neighbor (kNN)*: return top-k vectors by similarity.
- *Hybrid*: combine keyword (BM25) and vector search, then rerank.
- *Filtered retrieval*: apply metadata filters (tenant_id, doc_type).

How to Hook It Up (Integration with MCP)

As a Resource (simple read)

Expose a read-only resource that runs a vector search and returns top-k documents.
```
@mcp.resource
def search_knowledge(query: str, k: int = 5) -> list:
    qvec = embed(query)
    ids, scores = vector_index.search(qvec, k=k)
    return [{"id": i, "score": s, "text": docs[i]} for i,s in zip(ids, scores)]
```

As a Tool (actionable search + follow-up)

Expose a tool that performs search + optional summarization via LLM.

```
@mcp.tool
def answer_from_docs(query: str, k: int = 5) -> dict:
    results = search_knowledge(query, k)
    # summarize with LLM (pseudocode)
    summary = llm.summarize([r["text"] for r in results])
    return {"summary": summary, "sources": [r["id"] for r in results]}
```

Reusing repo scaffolding: You can add `server/vector_store.py` and `server/tools.py` to call the vector index and embedding model. The existing `client` can call these resources/tools as usual.

Gaps & Challenges

- **Embedding Drift:** Models and embeddings evolve; old embeddings may not align with new models — plan for reindexing strategies.
- **Scale & Cost:** Managed vector DBs (Pinecone, Milvus Cloud) cost; local FAISS needs memory management & sharding.
- **Security & Multi-tenancy:** Ensure tenant isolation, metadata filters, and encrypted storage for sensitive documents.
- **Latency:** ANN search is fast, but embedding generation adds latency — consider caching embeddings for frequent queries.

Enterprise Considerations

- **Indexing pipelines:** Batch ingestion, incremental updates, background reindexing, and monitoring of index health.
- **Observability:** Track query latency, vector distribution, recall metrics, and reindex events.
- **Compliance:** PII redaction before embedding, retention policies, and audit logs for data access.

Code Examples (FAISS + Pinecone pseudocode)

FAISS (local)

```
import faiss, numpy as np
xb = np.array([...], dtype='float32')  # data vectors
index = faiss.IndexFlatL2(d)  # simple index
index.add(xb)
D, I = index.search(np.array([qvec]), k)
```

Pinecone (managed)

```
import pinecone
pinecone.init(api_key="...")
idx = pinecone.Index("kb-index")
res = idx.query(vector=qvec, top_k=5, include_metadata=True)
```

Interview Angles & Suggested Answers

Q: How would you handle reindexing when model embeddings change?
A: Plan for rolling reindex: re-embed in batches, maintain dual indexes (old/new) during transition, validate with recall tests, and schedule reindexing off-peak. Keep metrics to compare recall differences.

Q: When would you use FAISS vs Pinecone?

A: FAISS for local, cost-sensitive setups with control over infra; Pinecone for managed scaling, low operational burden, and advanced features like hybrid search and multi-region replication.

Appendix & Further Reading