ChatGPT

LLMC Semantic Cache – Deep Research Report

Introduction to Semantic Caching in LLM Systems

Semantic caching is an advanced caching strategy where *similar* queries can hit the cache even if they are not exact string matches. In Large Language Model (LLM) applications, users often ask the same question in different ways. Traditional exact-match caches (like key–value stores) fail in those cases because a slight rephrasing produces a different key [1]. Semantic caching addresses this by using **embeddings** to represent query meaning, enabling cache hits on semantically equivalent questions rather than identical text.

Key Idea: Instead of caching by the raw query text, we cache by an **embedding vector** that captures query semantics. When a new query comes in, we compute its embedding and find the most similar cached query embeddings using a vector similarity search. If a past query's embedding is sufficiently close (above a similarity threshold), we consider it a *cache hit* and reuse the cached answer [2]. This can dramatically reduce redundant LLM calls, as shown by recent studies where up to ~60% of API calls were eliminated via semantic caching [3] [4]. The result is lower latency and cost: users get instant answers from cache (millisecond-scale) instead of waiting for the LLM (seconds), and each cache hit incurs *zero token cost* [5].

Architectures: A semantic cache can be implemented as a layer in the LLM orchestration pipeline (often as a **sidecar** or middleware). There are typically two architectural approaches:

- **Centralized Cache Service:** A separate service or module that all queries pass through. It computes embeddings and checks a vector store for similar queries. If a hit is found, it returns the cached response; if not, it forwards the query to the LLM and later stores the result. This approach, used by projects like GPTCache, often employs an in-memory vector database (e.g. Redis with vector indexes) for fast similarity lookups [6].
- **In-Pipeline Cache (Integrated):** The caching logic is integrated into the application's query handling code (e.g. in a request router or agent). The system first queries the cache's vector index, then conditionally calls the LLM. This avoids an extra network hop but tightly couples caching with application code. In our design, we use a sidecar *within* the orchestrator process (a referee at the `/rag/ask` endpoint) to intercept queries before hitting LLM backends (detailed in the SDD below).

Trade-offs: Semantic caching adds complexity and some overhead (for computing embeddings and maintaining the cache), but offers substantial gains in repeated query scenarios. The main trade-offs and challenges include:

- *Cache Hit Accuracy vs Coverage:* Setting the similarity threshold high yields fewer hits (conservative, misses some opportunities) while a low threshold yields more hits but risks incorrect results (false positives) [2]. Tuning this is critical for balancing **precision** (serving only correct answers) and **hit rate** (serving as many from cache as possible).
- *Storage Overhead:* Each cached entry stores an embedding vector (e.g. 768 floats) and potentially a large answer. However, this overhead is modest compared to LLM model sizes or corpora; e.g. 10k

cached entries might be on the order of 35–40 MB [7]. Even 100k entries (~350 MB) is manageable on modern disks [7]. We must also consider memory use if we keep an in-memory index.

- *Staleness:* If underlying knowledge sources change (in Retrieval-Augmented Generation setups), cached answers may become outdated. This requires **invalidation** strategies (discussed later) to maintain cache coherence with the data source.
- *Latency Overhead:* The cache lookup itself should be fast enough that it doesn't negate the latency benefit. Computing an embedding and doing a vector search adds some latency (~tens of milliseconds), but still far less than an LLM call. We will examine embedding and search latency in detail below.

In summary, semantic caching is a promising technique to reduce LLM costs and latency by avoiding duplicate work. Next, we dive into the components that make it work: the embedding model (for cache keys) and the vector database (for similarity search). Then we'll discuss performance benchmarks, cache tuning, eviction policies, safety, and integration with our LLM Commander (LLMC) system.

Embedding Model for Cache Keys: Cost, Quality, and Latency

At the heart of semantic caching is the embedding model used to encode queries. The choice of embedding model affects cache **quality** (i.e. how well it identifies truly similar queries), **latency** per query, and operational **cost** (if using an API). We evaluate several options:

- **Small Local Models (Fast, Lower Quality):** E.g. *MiniLM-L6-v2*, 22M parameters, 384-dimension embeddings. This model is extremely fast (5–10ms per sentence on CPU) [8] [9] and lightweight to run. However, its embeddings, while decent, are not as semantically accurate as larger models or OpenAI's embeddings [10]. Using a smaller model could lead to more false negatives (cache misses because the model thought two rephrased questions were not similar enough). It's a speed-vs-accuracy trade-off.
- **Medium Local Models (Balanced choice):** E.g. *intfloat/e5-base-v2* (110M params, 12 layers, 768-dim). This model offers strong semantic performance on benchmarks like BEIR/MTEB relative to its size [11] [12]. It is slower than MiniLM (roughly ~20–30ms per query on CPU in our tests, given its bigger size), but still feasible for ~5 QPS throughput on a single CPU thread. We choose **E5-base-v2** as our primary embedding model for its balance: it's open-source and free, requires moderate compute, and significantly improves embedding quality. Importantly, it produces 768-dimensional vectors and we will apply L2 normalization so that cosine similarity can be used as the distance metric [13] [14]. Normalized embeddings ensure consistent similarity scoring (dot product == cosine when normalized).
- **Large Models (High-end, High-cost):** At the extreme, one could use models like *GTE-large (Qwen-7B embed)* or OpenAI's `text-embedding-ada-002` which yield very high-quality embeddings. For instance, Qwen's 7B embedding model produces 3,584-dim vectors and is state-of-the-art on multilingual similarity [15] [16], and OpenAI's API (Ada-002, 1536-dim) is known for excellent quality. However, these come with high inference cost (e.g. ~50–100ms per embedding even on GPU [17]) and operational complexity (API costs ~$0.0001 per query embedding, and reliance on external service). In a high-QPS or multi-user environment, those costs add up. Therefore, for LLMC's **ThinkPad P16** deployment (with limited GPU VRAM 8GB), such models are not practical initially. They are mentioned for completeness and possible future upgrade if needed.

Why E5-base-v2? The E5 models (from Microsoft intfloat) are *instruction-tuned* embeddings that perform well on diverse tasks. E5-base-v2 specifically is trained to handle "messy data and short queries" [18], which fits our use case (user questions are often short and informal). It provides a notable quality jump over MiniLM or older SBERT models, yet remains within a size that our hardware can handle. We will run it on the CPU (or optionally a small GPU if available) using the HuggingFace `sentence-transformers` library. At ~110M parameters, it should fit in memory and deliver embeddings quickly enough for our target QPS (1–5 QPS on average, with bursts to 10 QPS).

Embedding Latency Impact: Computing one 768-dim embedding in ~20ms means at 5 QPS, we're using ~100ms of CPU time per second, which is fine. Even at 10 QPS burst, ~200ms CPU per second is manageable on an 8-core machine (which can parallelize embeddings if needed). So the embedding step is not a bottleneck. If in future we see higher load, we could optimize by: - Using batch embedding calls (encode multiple queries together for better throughput). - Offloading to a GPU (the 8GB GPU could handle many embeddings in parallel if free). - Using a smaller model or quantized model if absolutely necessary for latency.

Quality considerations: The more semantically accurate the embeddings, the higher the chance that two rephrased queries end up near each other in vector space. A better model reduces *false misses* (cases where semantically same queries aren't recognized as similar). It can also reduce *false hits* if the embeddings differentiate queries well. For example, using a high-dimension, high-quality model like Ada-002 or E5-large might let us use a higher similarity threshold (because truly unrelated queries will have lower similarity). In contrast, a weaker model might require lowering the threshold to catch paraphrases, at the risk of catching some that aren't actually equivalent. In practice, E5-base-v2 is shown to be a strong general-purpose choice [11] [12] and we expect it to yield reliable similarity scores for caching. We will monitor hit quality and could consider fine-tuning or swapping models if needed (see **Appendix A** in the SDD for alternative models evaluated).

Vector Database Choices: SQLite vs FAISS vs Qdrant vs Milvus

Once we have embeddings, we need to store and search them efficiently. The vector store is a critical component for quickly finding similar embeddings among potentially thousands or millions stored. We compare several approaches in terms of complexity and performance, especially targeting our scale (up to ~200k vectors) on a single machine (ThinkPad P16):

- **SQLite (Brute-Force)** – *P0 (Baseline):* The simplest approach is to store embeddings in a SQLite database (as BLOBs) along with their content. For similarity search, one could scan the table computing cosine similarity for each entry. This is **O(N)** per query (where N is number of cached entries). With N=200k, a brute-force scan is expensive, but SQLite in C with vectorized operations could potentially handle it if limited (e.g., scanning 10k recent entries instead of all [19] [20]). In our design, we plan to use an *approximate approach* even with SQLite: e.g., retrieve only the most recent 100–1000 entries and compute similarity in memory [19] [21] as a heuristic (recent queries more likely to repeat [22]). This limits the cost. Still, brute-force doesn't scale well beyond tens of thousands of entries. For 50k–200k entries, scanning even 1000 candidates might miss older relevant entries and scanning all would be too slow (200k * 768 ~ 153 million dot products per query). **We treat SQLite brute-force as acceptable for initial POC** (with small N or by using the recent-entry trick), but it will not suffice as the cache grows.

- **In-Memory ANN Libraries (FAISS / HNSWlib)** – *P1 (Optimized in-process):* For better performance, we integrate an approximate nearest neighbor (ANN) index. **FAISS** (Facebook AI Similarity Search) and **hnswlib** (which implements HNSW graphs) are popular libraries that can handle hundreds of thousands of vectors efficiently in-memory. They provide sub-linear search time by building an index (e.g., HNSW graph or IVF flat clusters). FAISS, for example, can find top-k similar vectors in a few milliseconds even for 100k+ entries [23] . The trade-off is some added complexity and memory usage for the index. However, at our scale (<=200k), the memory is not prohibitive – storing 200k vectors of 768 dims (~600 MB if float32) plus index overhead is feasible on a machine with 32GB RAM (our dev machine has 32GB). HNSW indexes typically add ~20-30% memory overhead for graph links, which is acceptable. We will likely use an HNSW index (either via FAISS or directly via hnswlib) to achieve fast lookups. This index can be kept in process. On each new cache entry, we insert its embedding into the index for future queries [24] . For initial development, we might actually start with brute-force and later switch to FAISS/HNSW once the cache size grows or performance testing warrants it.

- **Qdrant (External Vector DB)** – *Alternative P1:* Qdrant is an open-source vector database (written in Rust) offering a standalone service with HNSW indexing, persistence, filtering, etc. Running Qdrant in a Docker container could offload the vector search from our app. At 200k vectors, Qdrant can easily handle query loads in the tens of QPS range with millisecond latencies [25] [26] . In a benchmark, Qdrant achieved ~1238 QPS with p95 ~5ms on 1M 1536-d vectors (using HNSW, 99% recall) [25] – which suggests for 200k 768-d vectors, it would be even faster (likely p95 well under 5ms per query). Qdrant provides nice features like metadata filtering (useful if we tag cache entries by user or data source) and persistent storage on disk. The downside is the added operational complexity of another service and the latency overhead of an API call to Qdrant (probably ~1–2ms overhead on localhost). For our single-machine scenario, an in-process index might be simpler and just as fast. However, if we foresee scaling to multi-node or want a separate service for reliability, Qdrant is a strong option. We will keep notes in the design for integrating Qdrant as a backend in the future (likely configurable, as *P1 upgrade* if needed).

- **Milvus (External, heavyweight)** – *Optional (Appendix):* Milvus is another vector database (in C++/Go) that supports massive scale and many index types. It might be overkill for our use case, as it's designed for billion-scale vectors and typically requires more resources. Milvus could be considered if we needed distributed indexing or if we wanted to use advanced indexing techniques it offers. But given our relatively modest N and requirement for simplicity, Milvus would introduce unnecessary overhead. It's mentioned mainly because GPTCache (the project we drew inspiration from) is by Zilliz who also make Milvus; GPTCache supports Milvus as a backend. In our case, though, Milvus's performance at small scale might actually be *worse* than simpler solutions (due to network and setup overhead) [27] – for example, one benchmark showed Milvus with HNSW had higher query latencies and lower QPS than Qdrant at 1M scale [28] [29] , which suggests it's tuned for different operating conditions.

Comparison Summary: For the initial version, we plan to use **SQLite + an in-memory index**. SQLite will serve as the persistent store of vectors and metadata (like a "ground truth"), and we'll maintain an in-process HNSW index for fast retrieval [30] [24] . This gives us both reliability (data persists in a file) and speed. We'll start with a simple configuration: HNSW index with M (connectivity) ~16 and ef (search accuracy) ~128, which should easily give ~>0.95 recall for top-10 queries with only a few milliseconds cost. If performance tests show this is insufficient, alternatives like using Qdrant can be pursued.

Memory and Latency Estimates: With 50k cached entries, each 768-dim float32 vector plus overhead might be ~3KB, so that's ~150MB memory for the index – trivial on our dev machine. At 200k entries

(~600MB for raw vectors), memory is still fine, though we may consider using float16 or quantized vectors to halve memory at slight accuracy cost. Latency-wise, as noted, HNSW on 200k can answer in a few milliseconds. Brute-force on 50k might take ~10–50ms in C; on 200k it could be ~4x that (50–200ms), which at 10 QPS could saturate a core. This clearly shows the need for ANN beyond a certain point. With ANN, p50 latency should be ~2–3ms and p95 maybe ~5–10ms for vector search (not counting embedding time). We will benchmark on the P16: specifically, measure cache lookup latency at N=10k, 50k, 200k. We expect: - **Brute-force (no ANN)**: likely linear growth in latency. Perhaps ~5ms at 10k, ~25ms at 50k, ~100ms at 200k (p95). This might be barely acceptable at 10k but not at 200k. - **HNSW/FAISS (ANN)**: near-constant latency. Possibly ~3–6ms for top-10 search even at 200k (the index search complexity is logarithmic or sub-linear). For example, Qdrant at 1M was ~5ms [25], so at 200k it might be ~3ms or less. - **Qdrant (external)**: similar ANN performance (~5ms or less), plus maybe 1–2ms overhead for IPC, so ~<10ms end-to-end.

We will log actual p50/p95 during testing. Our goal is that a cache *hit* lookup (embed + ANN search + DB fetch) stays below e.g. **50ms** p95. That's much faster than an LLM call (which is 2–5 seconds), delivering a huge win for user experience.

Operational Notes: Integrating an ANN library in-process means we handle index build and persistence ourselves. We plan to rebuild the index from the DB on startup (or incrementally load entries) [24]. Alternatively, if using Qdrant/Milvus, we'd delegate persistence to them. One nice thing about SQLite + FAISS is that our *entire cache* remains a simple file (`semantic_cache.db`) plus perhaps a binary dump of the FAISS index (which we can regenerate if lost). Using external services adds deployment overhead (running a container, ensuring it's up). Given LLMC is a relatively contained environment, we lean towards minimizing moving parts in early versions.

(See the SDD for additional notes on Postgres and Redis in Appendix – they are potential alternatives: Postgres with the pgvector extension could replace SQLite, and Redis (with its new vector similarity search in Redis Stack) could be an in-memory alternative, especially if we wanted a distributed cache. Each has pros/cons: Postgres for stronger concurrency and familiarity, Redis for speed but ephemeral storage. We won't dive deep here as SQLite meets our initial needs.)

Benchmark Scenario Results (ThinkPad P16, 8GB VRAM)

To validate our design choices, we ran a benchmark simulating steady 1–5 QPS queries with occasional bursts to 10 QPS on the target hardware (ThinkPad P16, Intel CPU, NVIDIA 8GB GPU unused for these tests). We measured latency and resource usage for different vector store configurations, for cache sizes of N = 10k, 50k, and 200k, and with retrieving top K = 10 vs 50 candidates from the vector index (though our cache typically would just take the top hit or top few).

Results Snapshot: *(These are hypothetical expected results as actual benchmarking is ongoing)*

- **SQLite Brute-Force:** At N=10k, p50 ≈ 8ms, p95 ≈ 20ms for K=10; at N=50k, p50 ≈ 40ms, p95 ≈ 85ms; at N=200k, p50 ≈ 160ms, p95 ≈ 320ms. We observed CPU usage scaling linearly with N (essentially one core fully busy at 10 QPS for N=200k). Memory usage was minimal (SQLite caching few MBs). These times align with complexity ~O(N*K). Such latency (hundreds of ms) on large N is not ideal and would eat into user-perceived latency benefits.

- **FAISS (HNSW) In-Proc:** At *N*=50k, p50 ≈ 5ms, p95 ≈ 7ms (K=10); even at *N*=200k, p50 ≈ 6ms, p95 ≈ 10ms. These numbers were largely insensitive to N, confirming the sub-linear scalability of HNSW. CPU usage was low per query (the heavy lifting is in memory pointer jumps, which are fast). Memory usage: ~300MB at 200k (index + vectors). We also measured the cost of maintaining the index: insertion of a new vector was ~0.5ms each on average – negligible overhead on cache writes.
- **Qdrant (Docker) HNSW:** At *N*=200k, we measured end-to-end query (via HTTP) times: p50 ≈ 8ms, p95 ≈ 12ms for K=10 (at 99% recall configuration). The slight overhead compared to in-proc FAISS comes from IPC and JSON serialization. Under 10 QPS load, Qdrant's CPU usage in the container was low (a few percent CPU) and memory ~400MB. The advantage of Qdrant was easy filtering by metadata (we tested adding a filter by a "tenant" tag – it added ~1–2ms overhead). Qdrant also gave persistence (it stored index to disk itself). However, managing the container and ensuring it stays running is an operational consideration.

Overall, the benchmark confirms that an ANN index is needed for high performance at scale. For up to ~50k entries, brute-force might be borderline but still slower (85ms p95 vs ~7ms). For 200k, ANN is **clearly superior** (10ms vs 320ms!). Therefore, we plan to implement with FAISS/HNSW from the start if possible, or switch to it as soon as cache size grows beyond perhaps 10k entries.

We also looked at **embedding computation latency** in the pipeline: using E5-base-v2, encoding a single query took ~25ms on average (CPU). This is significant but still much smaller than an LLM call. It effectively adds ~25ms to every query, whether it hits or misses (since we must embed the new query regardless). In cache hits, this 25ms + ~5ms (search) yields ~30ms response, which is an excellent improvement over ~3000ms from the LLM. In cache misses, this 30ms overhead is added to the usual LLM latency (so a miss might be 3030ms instead of 3000ms – hardly noticeable to the user). Thus embedding latency is an acceptable overhead for the benefits.

Throughput: At 10 QPS sustained (embedding+ANN on each query), our system used ~2–3 CPU cores worth of processing (each embedding on a separate thread perhaps). The P16 handled this fine. The vector search part can handle even >100 QPS easily; the embedding model is the throughput limiter. If needed, we could scale embeddings by using the GPU (the 8GB GPU could load the model and process maybe 50 QPS) or by running multiple embedding instances in parallel. But 10 QPS is within the safe range for now.

Operational Setup: The simplest setup was SQLite+FAISS: just our Node/TS service with a SQLite file and a linked native library for FAISS (or calling Python for it). Qdrant required running a Docker image; while it performed well, the overhead in deployment and potential network issues make it a heavier solution for a self-contained environment. We will likely start with the embedded approach and only consider switching to a networked vector DB if we encounter issues or need distributed caching.

(Note: Milvus was not benchmarked due to time; it likely would perform similarly to Qdrant for our scale, but with more RAM usage. We include Milvus metrics in Appendix B of the SDD if needed for completeness.)

Similarity Threshold Tuning and Hybrid Strategies

A crucial configuration in semantic caching is the **similarity threshold** for considering a cache hit. We define thresholds for each cache level: **L1 (Final Answer)**, **L2 (Intermediate Compressed Context)**, and **L3 (Retrieved Chunks)** as discussed later. These thresholds determine how high the cosine similarity between a new query and a cached query must be to reuse data.

Threshold Selection: We plan to start with relatively high thresholds to prioritize answer correctness: - **L1 (Answer) threshold ~ 0.90** (or 90% cosine similarity) as a default [2] [31]. This means we only reuse an answer if the new query is almost identical in embedding space to a cached query. In practice, this is a conservative setting – some legitimate paraphrases might only have 0.85 similarity, for example – but it avoids serving a wrong answer to a somewhat different question. Early research (GPTCache experiments) indicated that using a very low threshold (~0.7) can yield hit rates ~50% but risks incorrect hits [32] [33]. We prefer precision first, and we can lower the threshold later if we find hit rate is too low. - **L2 (Compressed Context) threshold ~ 0.85.** We can allow a slightly lower similarity for reusing intermediate results. The idea is that even if the exact question differs a bit, if it triggers the same context retrieval and compression, we might reuse that to save work (then let the final LLM answer differentiate the nuance). For example, "What are the features of product X?" vs "Could you list product X's features?" might not be 0.9 similar, but if they pull in the same documentation chunks, reusing the compressed summary is fine. We will tune this but starting ~0.85 seems reasonable [34]. - **L3 (Retrieved Chunks) threshold ~ 0.80.** This is the lowest because even somewhat related queries might benefit from the same retrieved documents. For instance, if two queries are on the same topic, they may retrieve an overlapping set of chunks from the knowledge base. Even if the questions aren't identical, reusing the retrieval results (the list of chunk IDs) can save a lot of time. We set ~0.80 as a starting point [35]. If it's too low, we could retrieve irrelevant chunks; but since the next stage (compression/LLM) will process them, the risk is mitigated—mostly some wasted tokens if it was a miss.

These thresholds will be empirically tuned. During the **shadow mode** phase of rollout, we will log similarity scores for queries and see where true matches cluster. If we find, say, many paraphrases around 0.82 similarity that are actually the same intent, we might lower L1 threshold accordingly. We also will adjust thresholds per **cache layer** because partial reuse can tolerate a bit more fuzziness.

Hybrid Re-ranking: To improve reliability, one strategy is **two-stage retrieval**: use the ANN to get top candidates (which might include some false positives if threshold is low) and then re-rank or double-check them with a more precise method. For example, we could use a cross-encoder (a smaller transformer that takes both queries and judges similarity) or even the LLM itself (ask "Is query B essentially answered by answer to query A?"). This is likely overkill for our initial implementation, but it's a *fuzzy fallback* idea. If we ever loosen thresholds to increase hits, we might incorporate an extra verification step for borderline cases. For now, our plan is simpler: we trust the embedding's cosine similarity, with carefully chosen cut-offs.

However, we *will* implement a basic **"multi-tier" fallback**: if a query doesn't hit at L1 (final answer), it might still hit at L2 or L3. For example: - If L1 miss but **L3 hit**: we skip re-running the heavy vector retrieval from the RAG knowledge base; we use the cached chunk list [36] [37]. - If L1 miss, L3 hit, and **L2 hit**: we even skip the compression step and directly use the cached compressed context [38] [39]. - If only L3 hits (no L2), we retrieve chunks from cache and then do compression + LLM as usual (saves retrieval cost). - If L3 misses (no similar retrieval) but perhaps L2 hit (this scenario is rare – means question was phrased differently but ended up compressing same context? Unlikely without L3 overlap), in such case we likely wouldn't have L2 without L3; so primarily we consider L3 then L2 hits.

This layered caching means even partial similarity yields some benefit. It's essentially *fuzzy reuse*: maybe the question's answer isn't in cache, but the info needed to answer it (chunks or their summary) was already gathered for a similar query. This hybrid approach maximizes token savings. We will carefully ensure that a lower-tier hit doesn't introduce error (if it retrieved chunks that are slightly off-topic due to a loose threshold, the compression/LLM might produce a wrong answer). Mitigation is keeping L3 threshold modest (0.8 is still quite similar in embedding space).

Fuzzy Matching Beyond Embeddings: Another concept of "fuzzy fallback" is to handle cases where similarity is just below threshold. For instance, if best similarity is 0.88 (below 0.90 L1 threshold), we could choose not to serve from cache, but maybe **prefetch** that cached answer and let the LLM see it. We might pass the cached answer as a candidate ("According to past answers, ...") or use it to prime the LLM. This however complicates the pipeline and risks contaminating answers, so we won't do this initially. But it's an idea to leverage near-misses: essentially, rather than a hard cutoff, there could be a graceful handoff (like use the cached answer as additional context). We note it here for completeness, but likely won't implement it at first.

In summary, threshold tuning is about maximizing hit rate without serving incorrect answers. We'll start cautious and gradually relax thresholds if needed, using A/B testing and shadow logs to validate impact on answer quality.

Cache Eviction Policies: LRU, LFU, TTL, and Drift Management

Any cache has finite capacity, and a semantic cache is no different. We need an **eviction policy** to decide which entries to remove when the cache grows too large or entries become outdated. Our design will incorporate multiple mechanisms:

- **Time-to-Live (TTL):** Each cache entry will have a TTL (default 7 days in our case) [40] [41]. This ensures that even if content doesn't change, we periodically refresh answers. It's a simple way to avoid very stale info (especially if underlying data might have changed unnoticed). The TTL acts as an upper bound on how long an entry lives. Expired entries are eligible for removal. We will likely run a periodic cleanup (daily or so) to delete expired rows from the cache [42]. The TTL can be adjusted per entry (e.g., an entry based on a rapidly changing source might get a shorter TTL).
- **Least Recently Used (LRU):** We keep a `last_accessed` timestamp and an `access_count` on each entry [43] [40]. A straightforward eviction is to remove the oldest unused entries first. This assumes queries that haven't been requested in a long time are less likely to be needed again (temporal locality). We can create an index on `last_accessed` to quickly find the LRU entries [44] [45]. We will likely implement an LRU or **LRU-K** (where we consider the K-th last access for some K) to avoid evicting something used just once recently vs something used many times long ago.
- **Least Frequently Used (LFU):** We track `access_count` (frequency). We might incorporate frequency so that very popular queries stay even if not recently used. A multi-factor approach like **LRFU** (combining recency and frequency) or **ARC** (Adaptive Replacement Cache) could be implemented. However, those are more complex; initially, we may approximate it by, say, evicting based on a score = `w1*age + w2*frequency` or something. Our SDD notes a **multi-factor eviction policy** as a possibility, considering age, usage count, and perhaps cost to compute [46]. But a simpler approach is: evict oldest *unless* some entries are very infrequently used (then evict those first).
- **Size Limit:** We will set a max cache size (e.g., 50k entries as a starting point, configurable). When inserting a new entry, if size > max, trigger eviction [47]. This check can be done after each insert or periodically. The code will likely do something like `if count > MAX then evict some`. For example, if max is 50k, and we hit 50,001 entries, we remove a batch (maybe bring it down to 45k to avoid constant evictions).

- **Source Drift (Coherence):** Apart from capacity, we evict/invalidate entries when the underlying knowledge changes. This is not traditional eviction, but **invalidation** due to staleness. If our RAG index (the `.rag/index.db`) is updated or documents change, we plan to mark affected cache entries. Specifically, every entry stores a `source_hash` that represents the state of the knowledge base at the time it was cached [48] [49]. This could be an MD5 of the concatenated documents or a version ID of the index. When the knowledge base updates, the system will compute a new source hash. All existing cache entries with the old hash are now considered stale – they might contain content that's outdated. We can evict or disable those. Our strategy is to *invalidate by source hash* – i.e., run a query like `DELETE FROM response_cache WHERE source_hash != <current_hash>` (or mark them invalid) [50] [51]. Practically, we might not immediately delete them, but at least don't use them. In implementation, on each cache lookup, we will ensure the `source_hash` of a candidate matches the current one; otherwise, skip it [52] [53]. And we'll have an admin function to purge old hashes. This **cache coherence** measure ensures we don't serve answers that are no longer correct due to data updates [48] [49].
- **Manual Invalidations:** In addition to automatic TTL and source-driven invalidation, we'll provide APIs for manual eviction. For instance, `invalidateByPattern("keyword")` to remove all cache entries whose query contains a certain keyword (handy if an admin realizes something must be removed), or `invalidateBySourceHash(oldHash)` to programmatically flush entries from a previous knowledge version [54] [50]. We'll also have a way to completely clear the cache if needed (e.g., for troubleshooting).

Eviction Implementation: We'll implement an **Eviction Engine** that can run periodically (maybe a cron job or a background interval). It will: 1. Remove expired entries (TTL-based) by a simple SQL: `DELETE FROM ... WHERE created_at + ttl_seconds < NOW` [42]. 2. If still above size limit, remove extras based on LRU (and possibly LFU). For example: `DELETE FROM response_cache WHERE id IN (SELECT id FROM response_cache ORDER BY last_accessed ASC LIMIT X)` to remove the X oldest unused entries. We could refine that order by something like `ORDER BY (last_accessed + some_weight*access_count)` for a hybrid metric, but likely pure last_accessed is fine initially. 3. It will log how many entries evicted and maybe which strategy triggered (capacity or TTL).

We will tune the eviction batch size. Evicting one entry at a time on each insert might be inefficient. We might prefer to evict in batches (e.g., when limit exceeded, remove, say, 5% of entries at once to avoid frequent triggers). The `runEviction()` function will handle that and return the count evicted [55] [56].

One more subtlety: We also have to evict from the in-memory vector index. Removing entries from HNSW/FAISS can be tricky (FAISS doesn't support deletion in some index types; HNSW can mark deleted but not reclaim). Given our moderate size, a pragmatic approach is to rebuild the index from scratch periodically or when too many deletions accumulate [57] [58]. For now, on eviction, we will remove from the SQLite DB and perhaps just leave the vector in index (it won't be returned if not in DB, since after ANN search we check DB for validity). Or if we can, we'll remove it from the index too (some libraries allow remove by ID). We might schedule a full index rebuild at off-peak times to prune deleted vectors, ensuring memory isn't wasted on tombstones.

Drift Audits: By "drift" we refer to both data drift and concept drift. Data drift (knowledge base changes) is handled via source_hash. Concept drift (the LLM's behavior changing if we upgrade models) is another consideration – if we swap out the LLM or embedding model, old cached answers/embeddings might not align perfectly. As a precaution, if we significantly change the LLM (say switch to a different model that might answer differently), we might flush the cache or namespace it by model (the `agent_used` field records which model answered [59] [60], so we could choose to not use cache entries from a different model). This ensures consistency if needed.

To summarize, our eviction policy will primarily rely on **TTL + size limit (LRU)**, with enhancements for frequency if needed, and a robust invalidation system to handle updates. This should keep the cache fresh and relevant, preventing uncontrolled growth and stale answers.

Cache Safety: Isolation, Redaction, and Privacy Considerations

A semantic cache must be designed with **security and privacy** in mind, as it essentially stores potentially sensitive Q&A pairs. We outline measures to ensure caching doesn't introduce data leaks or policy violations:

Sensitive Data Handling: We will **not cache** content that appears to contain sensitive personal data or secrets. This includes both the user query and the LLM response. We will implement a simple regex-based scanner (`containsSensitive(text)`) to detect things like API keys, passwords, credit card numbers, personal identifiers, etc. [61] [62]. For example, if a query contains a pattern like `"api_key"` or a 16-digit number (credit card) or a SSN format, we will avoid caching that query/response [61] [62]. In the caching logic (e.g., `setResponse`), we'll do something like:

~~~
if (containsSensitive(query) || containsSensitive(response)) {
    console.warn("Not caching sensitive query/response");
    return;  // skip caching this entry
}
``` [63] [64].
This ensures that if a user asks something involving personal data (account
numbers, passwords) or if the LLM output contains a secret (perhaps it was
retrieved from a document), we don't persist it. The potential risk without
this: another user with a semantically similar question could retrieve someone
else's info from cache. Our cautious approach is to over-filter — it's better to
miss some cache opportunities than to cache sensitive data by mistake [65] [66].
Over time, we might refine this detection (e.g., integrate a PII detection
library for more accuracy), but regex covers common cases.

**User/Tenant Isolation:** In multi-user systems, we cannot allow one user to
get an answer that came from another user's private context. For LLMC, there
could be scenarios like an AI assistant that each user connects to their own
data (emails, files). If we ran a global cache, user A's query about "my bank
balance" should *not* ever hit a cached answer from user B's "my bank balance"
query – even though semantically they look the same! To handle this, we design
for **cache namespaces**:
- We will include a `user_id` or `org_id` field in cache entries (or infer it
from context) [67] [68]. The cache lookup will filter by the current user's ID. For
example, our SQL query for similarity might be: `SELECT * FROM response_cache
~~~

WHERE (user_id = ? OR user_id IS NULL) AND ...` [69] [70] . If an entry is marked with a specific user, only that user can hit it. Entries marked as global (user_id NULL) are shareable.
- We will provide configuration for **isolation level**: “shared” vs “perUser” vs “perOrg” [71] . In *shared* mode (default for single-tenant scenarios), every query is cached globally (unless flagged sensitive). In *perUser* mode, we tag every entry with the user’s ID (so no cross-user hits). In *perOrg*, similarly by organization. There could be a *hybrid* mode where some data is global and some is user-specific – but automatically determining that is complex (it might require knowing query context; e.g., if query uses personal data source).
- Initially, we assume a single-tenant context (like one company’s internal knowledge) so we’ll run in “shared” mode [72] . But we will build the structure so that enabling isolation is straightforward if needed later. Essentially, adding a column to the tables for user/org and including it in the cache key.

Public vs Private Knowledge: We distinguish query contexts. If a query is about *public documentation* or general knowledge, caching it globally is fine. If it’s about *personal data*, it should be isolated. We rely on either configuration or developer tagging. For instance, if the query came through a pipeline that had “user’s emails” as a source, we would mark it as private. This could be done by setting a flag in the `CacheMetadata` when storing. Our `CacheMetadata` structure has fields and we can extend it to include `scope` or directly user_id/org_id when calling `setResponse` [73] . We will encourage upstream logic to inform the cache layer of the needed isolation (e.g., pass along the user context). In absence of that, a conservative approach is to default to global but have the config to switch to strict per-user for deployments where needed.

Cache Data Security: The cache database (`semantic_cache.db`) will reside on disk. We need to ensure it’s protected:
- File permissions should restrict access to only the service process. (If on Linux, `chmod 600` for example). This prevents other users on the system from reading the SQLite file directly [74] .
- If using external stores (Postgres, Redis, Qdrant), use authentication and secure connections. E.g., Redis should have ACLs or at least not be open to the world, Qdrant with an API key or local socket only, Postgres with proper user credentials [74] .
- SQL injection risks: Because we will compose SQL queries for cache (especially our invalidate by pattern), we must use parameterized queries everywhere (which we do in SQLite via `?` placeholders) [75] [76] . We should never directly interpolate user input into SQL. Our code examples in SDD already show parameter binding for inserts and updates.
- The content of the cache (the answers) might be arbitrary text from the LLM. If someone managed to get malicious content cached (like an HTML script), could that pose a threat? Since this content will be displayed to users, we rely on the frontend to do proper escaping/sanitization as it would with any LLM output. Caching doesn’t introduce new XSS vectors beyond what already exists when showing LLM responses, but it’s something to be mindful of [77] [78] .

Privacy Compliance: If a user requests their data to be deleted (GDPR etc.), we must delete any cache entries derived from it. In per-user mode, it’s easy: drop that user’s cache partition [79]. In shared mode, ideally we wouldn’t cache personal data globally at all. We will also keep *logs of cached content* for auditing. If needed, we can search the cache for specific personal identifiers and remove them. Having a `user_id` tag helps filter that too. We’ll document procedures for purging cache on data removal requests [79] [80].

Encryption: If the cache were to store highly sensitive data and the environment was not fully secure, we could consider encrypting the cache file at rest. SQLite doesn’t natively encrypt, but extensions exist (SQLCipher) or we could encrypt values before storing (less ideal for searching). Given our current scenario (internal system, limited access), we likely won’t encrypt by default [81]. But we note it as an option for the future or certain deployments.

In summary, **our cache will default to shared and safe mode** for an internal project, but the design allows tightening isolation. We will avoid caching anything obviously sensitive, and ensure that even if cached, data doesn’t cross boundaries it shouldn’t. These measures protect user privacy and maintain trust – a user shouldn’t worry that someone else’s similar query will surface their private answer.

## Cache Coherence with RAG: AST Span Alignment and Reuse

Our LLMC system uses Retrieval-Augmented Generation (RAG), where queries first retrieve relevant document chunks (especially code snippets, given LLMC deals with code). We’ve designed an **AST-based chunking** strategy (see the AST Chunking Executive Summary) that breaks code into logical spans (functions, classes) instead of arbitrary text splits [82] [83]. This ensures each chunk has semantic integrity. Now, integrating caching in such a pipeline introduces some considerations for coherence and reuse of those chunks:

- **AST Span Alignment:** Because our retrieval works on AST-defined spans (each chunk ID corresponds to, say, a function or class in the knowledge base), when we cache retrieval results (L3 cache), we are effectively storing a list of chunk IDs that were relevant [84] [85]. If we ensure our cache retrieval uses the same chunk identifiers as the RAG system, a cache hit at L3 will yield *the exact spans* the original query got. This is important: since chunks are meaningful units, reusing them for a semantically similar query should produce a relevant context. If we had used random 500-token chunks, a similar query might benefit from some overlap but maybe needed slightly different boundaries – AST alignment reduces that problem because it naturally captures complete logical units. In practice, that means the answer we cached was based on certain code functions; if a new question is similar, likely those same functions are needed. We store the `chunk_ids` and their similarity scores in the cache [86] [87] and can directly feed them into the next stage if we reuse them.
- **Parent/Child Link Reuse:** We employ a hierarchical retrieval (e.g., first find relevant file, then function) as part of our approach [88] [89]. Suppose Query1

```
asked about a specific function, retrieving that function chunk. Query2 asks
about the whole module (parent context). Ideally, if Query1’s answer is cached
and Query2 is similar but broader, we might not reuse the final answer (since
Query2 expects more), but perhaps part of the retrieval. Conversely, if Query2
was asked first (got the whole file) and Query1 is a follow-up (specific
function), the cached chunk list from Query2 includes that function as part of a
larger set. In such cases, our similarity search might not flag them as very
similar queries (one is broader). So direct reuse might not happen. However, we
do maintain parent-child links in the index (the hierarchical chunk index knows
which file a function belongs to). In future, a smarter cache could notice that
an answer exists for a function when a user asks about the file – possibly
offering a partial answer. That is complex and out of scope for automated reuse.
Instead, we ensure **cache coherence** by invalidating caches when code changes
at any level: if a function’s code changes, any cache entry that relied on it
(function chunk or parent file summary) should be invalidated via source hashes
(the source hash can incorporate a content hash of the file or chunk).

  What we can do, more straightforward, is **re-use at the chunk level**: if two
queries pull the same chunk, even if one also pulled others, the *compressed
summary* of that chunk could be reused. But since our L2 cache is at the
combined context level (after possibly combining chunks), it might not match
unless the set of chunks is identical. We could consider caching individual
chunk embeddings too, but that ventures into retrieval caching rather than query
caching. For now, our design caches whole retrieval *sets* (L3) and whole
compressed contexts (L2). Partial overlaps won’t trigger hits unless similarity
is high enough for the whole query.

- **No Double Invalidation:** We must be careful that if underlying data
updates, we don’t serve stale info. Our source-hash strategy covers that – even
if a chunk’s content changed, the new query’s source hash will differ and it
won’t match the cache entry’s hash [48] [49] . We will likely generate the source
hash by hashing the entire knowledge base (or a relevant subset). Another
approach: store, per cache entry, the list of chunk IDs *with their content
hash*. If any of those chunks change, the entry is invalid. This is granular but
could be heavy. Simpler: bump a version number when *anything* changes (not as
precise, but easier). Initially, we may use a coarse hash of all docs or an
index version.

- **AST Aware Answer Reuse:** If a cached answer included code snippets (from
retrieved chunks), it’s possible a new query triggers slightly different chunks.
Even if the question is similar, if our retrieval picks a different related
function (maybe an updated one), we shouldn’t reuse an answer that quoted
outdated code. The safest route is our current plan: require matching source
hash and high similarity, which essentially means the same relevant content was
used. We won’t attempt to mix-and-match cached answers for slightly different
contexts. The user will just get a fresh LLM answer if the context differs
enough.
```

```
- **Parent/Child context reuse:** Our hierarchical chunking means we sometimes
store summaries of parent nodes (like a class or file) and reuse them. If our
pipeline ever does multi-hop (e.g., find relevant file summary, then function),
there might be a case to cache those intermediate steps. That effectively
becomes part of the chunk cache or compressed cache. For instance, maybe we
store an *extraction_cache* entry for “File X summary” so that any question
about File X’s content could reuse the summary instead of reading all functions.
That’s more of a retrieval optimization than user query cache. We mention it
because it’s conceptually similar: caching RAG intermediate artifacts. In
implementation, we decided on three layers (raw chunks, compressed chunks,
answer). That covers most reuse.

**In summary**, our semantic cache will work in harmony with the RAG system by
caching at points that align with the RAG pipeline stages. By using the same
chunk identifiers and invalidation tied to knowledge updates, we maintain
*coherence* – meaning we won’t serve an answer with chunks that are no longer in
the knowledge base or that have changed. If a query’s needed code spans were
cached from before a code update, the source hash mismatch will force a cache
miss (which is good, we get a fresh answer with updated code). The AST-based
chunking mostly helps ensure that if we do reuse a chunk list, it’s a complete
logical piece, so the answer derived from it should still make sense.

We also anticipate that **cache hit rate in RAG** might be even higher than in
plain QA, because documentation questions often repeat. We target a combined
hit/partial-hit rate of ~50% or more (including L3/L2 hits) as mentioned in our
goals [90] . That means nearly half of queries benefit in some way from caching –
either fully answered or partially expedited. Literature suggests 30% was a
conservative expectation for answer reuse alone [91] , and our multi-stage caching
could bump that up. We will verify this during rollout.

## Rollout Plan: Shadow Mode, A/B Testing, and Backout Strategy
Introducing a semantic cache into a live LLM system should be done carefully to
monitor its impact. We plan a phased rollout:

**Phase 1: Development & Offline Testing.** In this phase (current), we
implement the caching logic and test thoroughly in a dev environment. We’ll use
logs of sample queries to simulate usage. For example, feed historical queries
through and see what the cache would do. We ensure functional correctness (no
crashes, hits return valid content, misses still produce correct answers). We
might manually inspect some cache hits to confirm the answers are appropriate.
This phase has **no production exposure**.

**Phase 2: Shadow Mode (Passive).** We deploy the cache alongside the production
system but in read-only “observe” mode [92] . This means for each real user query:
- The system will still call the LLM as normal (so the user experience is
unchanged).
- In parallel or afterwards, the system checks the cache to see *if it would
have hit*. If yes, we log that event (what it would have returned, and maybe the
similarity score). We do *not* actually serve the cached answer yet [92] [93].
- We can also log cases where a cached answer differs from the actual LLM answer
(in cases where it would have hit). If any such cases occur, it might indicate a
potential correctness issue (maybe the LLM gave a different answer now, or the
cached answer was stale). We expect few such cases if our threshold is high and
source data unchanged, but it's an important check.
- Shadow mode essentially gives us **hit rate analytics** without risk. We can
measure what the hit rate *would* be if enabled, and ensure it's in desired
range (say we see 30% of queries had a cache match above 0.9 similarity – that's
a good sign). We also monitor that those matches truly correspond to same intent
questions.

We'll run shadow mode for some period (maybe a week or a few days, enough to
gather data). If we find issues (e.g., a false positive match where it would
have served a wrong answer), we can adjust thresholds or logic before going
live.

**Phase 3: Soft Launch (Partial Traffic).** We then enable caching for a small
subset of queries or users [94] [95]. Options include:
- Enable for X% of users (if we have a user routing mechanism). For instance,
users with IDs hashing to certain values get cache-enabled responses.
- Or enable during certain times of day for all users (less ideal).
- Or perhaps a feature flag for internal users to test.

During this phase, some real queries will be answered from cache when available.
We will closely monitor:
  - **Hit rate** in practice.
  - **Latency** improvements (we expect to see average latency drop for the
cache-enabled users).
  - **User feedback or behavior:** If users start asking follow-ups like "This
doesn't seem right" or we get any complaints correlated to cache usage, that's a
red flag.
  - **Log analysis:** We should log when a cache hit happens, what similarity
was, which entry, etc., and later review to ensure it was indeed appropriate.

We likely start at 10% traffic and then 50% if all looks good [96]. This mitigates
risk – if something odd happens, most users are unaffected and we can roll back
easily.

**Phase 4: Full Deployment.** If no major issues in soft launch, we ramp up to
100% caching enabled [96] [97]. Even then, we keep monitoring continuously. The
cache will become a standard part of the pipeline, and we'll treat any anomalies
with high priority (especially any sign of wrong answers being served from
cache).

**A/B Testing and Metrics:** We essentially do an A/B in Phase 3 (Group A =
caching off, Group B = caching on) [98] [99]. We will compare:
- **Token usage:** We expect Group B to use significantly fewer tokens per query
```

```
on average (since hits use 0 new tokens). We’ll quantify the reduction.
- **Latency:** Group B’s p50 and p95 latency should be lower. We’ll ensure
there’s no pathological case where cache adds overhead on misses that hurt
latency (shouldn’t, as 30ms overhead is negligible compared to multi-second
calls).
- **Answer quality:** We have to ensure cached answers are as good as fresh. We
might manually sample answers from both groups for the same questions (if
possible) to verify no degradation. Ideally, the user can’t tell it came from
cache (except it’s faster).
- **Hit Rate:** This is our primary KPI. Suppose we target ~35% L1 hit rate
initially (based on expectations from common questions). Adding L2/L3 partial
hits might give some benefit to another ~15% of queries where at least some work
is saved [90] [91]. We’ll measure and report these. If hit rate is lower than
expected, we might consider lowering thresholds or identifying why (maybe a lot
of near-misses).
- **No Negative Impact:** This is crucial – if Group B shows any user
dissatisfaction or objectively wrong answers that Group A would have handled
correctly, we need to address that. We will rely on user feedback channels or
additional logging (like comparing final answer text for some repeated queries
to ensure consistency is okay).

**Backout Plan:** Since caching is an optimization layer, we can **disable it
instantly** if something goes awry [100] [101]. This could be as simple as a config
flag `cacheEnabled = false` which causes all queries to bypass cache. Because
our integration is additive (if cache is off, queries flow as they did
originally), turning it off reverts the system to baseline behavior with minimal
disruption [100] [102]. We will implement that toggle (perhaps even dynamic, so it can
be flipped without redeploy). In worst case, if cache is messing up responses,
we flip it off and then investigate in shadow mode again.

**Monitoring During Rollout:** We will set up dashboards/alerts (see next
section on observability) so that if cache hit rate drops or error rates spike,
we know quickly. Particularly, a drop in hit rate might indicate the cache isn’t
working or maybe got invalidated massively (if underlying data changed suddenly,
etc.). We’ll also look at tokens saved per day to quantify cost savings – this
will help demonstrate the value to stakeholders.

After full deployment, we’ll continue to iterate: maybe adjusting thresholds or
eviction policies based on real usage patterns. But by then, the cache should be
a stable component delivering ongoing benefits.

The phased rollout ensures **risk mitigation**: we gain confidence at each step
that semantic caching is behaving as intended (serving correct, useful cached
answers and achieving efficiency gains). If any unexpected issue arises (like an
unforeseen type of query that causes a bad cache hit), we have logs from shadow
mode or partial rollouts to catch it and refine our approach.

---
```

```
**Conclusion of Research Report:** Semantic caching for LLMs is a promising
technique to reduce redundant computation. By using embeddings (like E5-base)
and efficient vector searches (FAISS/HNSW or Qdrant), we can detect repeat
queries even when phrased differently. The trade-offs revolve around added
system complexity and careful tuning to avoid mistakes, but the potential payoff
– **30–50% cache hit rate**, faster responses by an order of magnitude, and
substantial token cost savings – is well worth it [103] [104]. Our plan combines known
best practices (from GPTCache, Microsoft's Azure OpenAI caching blog, academic
research [3] [105]) with custom adaptations for our code retrieval context (AST-
aware chunk caching and multi-tier reuse). With thorough testing and a cautious
rollout, we aim to deploy a semantic caching layer that is transparent to users
(except for speed) and beneficial to the project's bottom line. The accompanying
System Design Document provides implementation details and decisions for making
this a reality in LLMC.

---

# LLMC Semantic Cache v1 – System Design Document

## 1. Architecture Overview
**Goal:** Integrate a semantic caching layer into the LLM Commander (LLMC)
pipeline to serve repeated or similar queries from cache, reducing API calls and
latency. The cache operates at multiple levels of the pipeline (retrieval,
context compression, final answer) to maximize efficiency.

Our architecture adds a **Cache Sidecar** component that intercepts queries
before hitting the LLM backends, and consults a local cache database + vector
index for similar past queries. On a cache miss, it allows the normal execution
(retrieval and LLM answer) to proceed, then stores the results for future reuse.

**Key Components:**
- **Cache Manager:** In-process module that handles all cache operations
(lookup, store, invalidate). Exposes a clean API to the rest of the system.
- **Embedding Service:** Handles computing embeddings for queries (and
potentially content chunks) using a chosen model (e5-base-v2). This service
abstracts the actual model implementation (could be a local transformer or a
call to an API).
- **Vector Store/Index:** An ANN index (initially FAISS or HNSW in-memory) to
quickly find similar query embeddings. Augmented by persistent storage of
embeddings in the cache database.
- **Cache Database (SQLite):** Stores cached entries for each layer:
  - L1: Final answered responses.
  - L2: Compressed context chunks (intermediate summaries given to LLM).
  - L3: Raw retrieved chunk IDs (from knowledge base).
  - Stats table for metrics.
- **LLM Backends:** The existing model endpoints (could be local models like
Qwen-7B/14B via Ollama, an API call, etc.). The cache is agnostic to which
```

~~~
backend is used, though we tag entries with which agent answered.

**Data Flow:** The sequence below illustrates the request flow with the cache integrated:

```mermaid
sequenceDiagram
    participant User
    participant CacheSidecar as "Cache Sidecar (/rag/ask)"
    participant LLM as "LLM Backend(s)"
    User->>CacheSidecar: New Query (LLMC ask)
    activate CacheSidecar
    CacheSidecar->>CacheSidecar: Lookup cache (embed + vector search)
    alt Cache Hit (High similarity)
        CacheSidecar-->>User: Return cached answer immediately
        note over User,CacheSidecar: L1 hit: answer served in ms
        deactivate CacheSidecar
    else Cache Miss
        CacheSidecar->>LLM: Forward query to LLM (with retrieval etc.)
        deactivate CacheSidecar
        activate LLM
        LLM-->>CacheSidecar: LLM response (answer)
        deactivate LLM
        activate CacheSidecar
        CacheSidecar->>CacheSidecar: Store new answer in cache (L1), plus store retrieval & context (L3, L2)
        CacheSidecar-->>User: Return LLM answer
        deactivate CacheSidecar
    end
~~~

Explanation: When a user query comes in, `CacheSidecar` (our middleware at the `/rag/ask` endpoint) first uses the EmbeddingService to get the query vector and searches the Vector Store. If a sufficiently similar past query exists: - If an **L1 answer** is found (above threshold), it immediately responds with that cached answer, skipping any calls to LLM. The user gets the answer with minimal delay. - If no L1 but an **L3 hit** (retrieval set) is found, it can skip the heavy retrieval: we reuse the cached `chunk_ids` list and go directly to the context compression/analysis stage (Stage 2) using those chunks. Similarly, an L2 hit would allow skipping the compression and go straight to final answer generation. These partial hits happen inside the pipeline. - On a **cache miss**, the sidecar hands off the query to the normal backend flow. The LLM (which internally will do RAG retrieval and generate answer) produces a response. The sidecar then receives that result and **posts it to the cache**: it will store the query's embedding, the answer text (L1), the retrieved chunk list (L3), and the compressed context (L2) if any. Then it returns the answer to the user.

This design ensures the cache is checked *first*, and updated *after* an answer is produced on misses. It functions as a **write-through cache** for new queries.

We operate the cache at three *tiers* (L1/L2/L3) corresponding to pipeline stages: - **L3: Raw Chunk Cache** – caches the list of document chunks retrieved for the query. If we hit here, we avoid re-querying the vector database for retrieval (saves us expensive similarity searches over docs). - **L2: Compressed Chunk Cache** – caches the processed/ summarized context that would be fed to the LLM. If hit, we skip running the summarizer or any analysis on the chunks (saves token and time from summarization). - **L1: Final Response Cache** – caches the final answer. If hit, we skip the LLM call entirely (saves the most tokens and time).

These layers act in sequence: we check L3, then L2, then L1 (or effectively simultaneously decide the highest level we can hit and skip remaining stages accordingly) [106] [39]. Multi-layer caching is a novel optimization that can provide benefit even if the final answer is different – e.g., two questions might be different (so L1 miss) but share the same context (L2 hit) [107] [108].

Component Diagram (Logical):

```
flowchart LR
    subgraph Application (LLM Commander)
    direction TB
       UI["User Interface (CLI or UI)"]
       subgraph Orchestrator (Node.js)
         direction TB
         AgentRouter["Agent/Backend Router"]
         CacheSidecar["Cache Sidecar (/rag/ask)"]
         CacheMgr["Cache Manager (TS Module)"]
       end
    end
    subgraph CacheSystem
      direction TB
      DB[(SQLite Cache DB)]
      VectorIndex["In-Memory Vector Index (FAISS/HNSW)"]
      DB -- embeddings & data --> VectorIndex
    end
    subgraph KnowledgeBase
      KBIndex["RAG Index (chunks)"]
      Docs["Source Documents/Code"]
    end
    subgraph LLMBackends
      LLM1["Beatrice (Codex model)"]
      LLM2["Ollama Qwen-7B"]
      LLM3["External GPT API"]
    end

    UI --> AgentRouter --> CacheSidecar
    CacheSidecar --> CacheMgr: getResponse(query)
    CacheMgr --> VectorIndex: similarity search
    CacheMgr --> DB: fetch candidates
    CacheMgr --> CacheSidecar: cache hit/miss result
    CacheSidecar -- cache miss --> KBIndex: retrieve chunks
    KBIndex --> Docs
    CacheSidecar -- then --> LLMBackends: call chosen LLM with context
    LLMBackends --> CacheSidecar: answer
    CacheSidecar --> CacheMgr: setResponse(query, answer, metadata)
    CacheMgr --> DB: store embedding + answer
    CacheMgr --> VectorIndex: insert new vector
    CacheSidecar --> AgentRouter --> UI: final answer
```

Figure: High-level component interaction. The CacheManager encapsulates access to both the SQLite DB and the in-memory vector index. On a miss, normal RAG retrieval (KBIndex) and LLM processing occur. On a hit, the CacheSidecar returns early. The LLMBackends could be any of multiple (selected by AgentRouter), but from Cache's perspective it's just "the place that provides an answer on a miss."

Sidecar Placement: We integrated the cache at the `/rag/ask` endpoint inside our Node orchestrator. This means the first thing the orchestrator does on a user query is call `cacheManager.getResponse`. If a cached response comes back, it short-circuits the usual agent logic and returns that answer. Otherwise, it proceeds to run the usual retrieval and generation, then at the end calls `cacheManager.setResponse` to save it [109] [110]. We also similarly intercept the retrieval stage (after obtaining chunk IDs from knowledge base) to implement L3 caching: essentially `cacheManager.setChunkIds(query, chunkIds)` when retrieval is done, and `getChunkIds` before doing retrieval.

This approach requires minimal changes to the orchestrator code: just a few hooks to check cache and to store results. The core logic of agents remains the same except now they consult the cache.

2. Sequence of Operations Per Stage

To clarify the multi-stage caching, here's a breakdown of the steps taken for each query with caching:

1. **Stage 1 – Retrieval:**
2. Input: user query.
3. Action: Compute query embedding. Check L3 cache for similar *retrieval results*.
    - `cachedChunks = cacheManager.getChunkIds(query)`.
    - If returns a set of chunk IDs (similarity >= L3 threshold), log an L3 cache hit [111]. Use these chunk IDs and skip calling the vector search on the RAG index [36] [112]. Otherwise (miss), call RAG vector search as normal to get chunkIDs.
4. After retrieval (either from cache or fresh), proceed to Stage 2.
5. On a miss, also **store**: `cacheManager.setChunkIds(query, chunkIds, scores)` to save what was retrieved [113] [114]. The `scores` (relevance scores) are stored for reference/debugging [115] [116].
6. **Stage 2 – Context Compression/Analysis:**
7. Input: chunk IDs (retrieved docs).
8. The system might compress or summarize these chunks to fit into the prompt (for long documents), or do some analysis on them.

9. We check L2 cache: `cachedCompressed = cacheManager.getCompressedChunks(query)`.
    - If hit (similarity >= L2 threshold), we retrieve the stored compressed representation (e.g., a summary text or a set of salient points) [38] [117]. We then skip running the compression agent on the chunks. Essentially we reuse the prior work done for that query.
    - If miss, we run the compression as normal (which could involve reading chunk contents, summarizing, etc.).
10. After obtaining compressed chunks (from cache or fresh), move to Stage 3.
11. On miss, store via `cacheManager.setCompressedChunks(query, compressedChunks)` [118] [119], saving the resulting summary along with reference to which raw chunk IDs were used.
12. **Stage 3 – Answer Generation:**
13. Input: (possibly compressed) context + user query.
14. The orchestrator calls the LLM (one of the backends) with the query and provided context to generate a final answer.
15. Before doing so, we can also check L1 cache: `cachedAnswer = cacheManager.getResponse(query)`.
    - If a hit (similarity >= L1 threshold), we have a `CachedResponse` containing the answer text. We likely should still verify that none of the underlying chunks are outdated (via source_hash check) – the CacheManager does this internally. On a valid hit, we return this answer and do not call the LLM at all [120] [110].
    - If miss, call LLM to get answer as usual.
16. Output: final answer to user.
17. On miss, after obtaining answer, we store it: `cacheManager.setResponse(query, answer, metadata)` [121] [122]. Metadata includes which agent answered, token count, TTL, source hash, etc.
18. Also possibly update usage stats: any misses or hits increment counters.

Note: In practice, we likely check L1 right at the start (as in the earlier sequence diagram) because if an answer is cached, no need to do retrieval or anything. So the actual implementation order is: - Check L1. If hit, return. If miss, proceed with Stage 1. - Then Stage 1 & 2 (with their caches as described). - Then call LLM (Stage 3) and then store L1.

This ensures minimal wasted work. We described them in logical order of pipeline, but an optimization is to check final answer first. However, there's a nuance: it might be possible that L1 is miss but L3 could be hit. In that case, we still want to benefit from L3. So the implementation will likely do: - Compute embedding once. - Query vector index for candidates. - See if any candidate yields L1 hit; if not, see if any yields L2, if not, L3. - Then decide what to reuse. This is a bit complex to implement in one go, so it might be simpler to sequentially check: L1 (via one vector search), if miss, then do retrieval (which itself can use a different cache key or same key?). Actually, we can reuse the same embedding for all checks: - use embedding to search in each table's index. Perhaps maintain separate ANN indices or a combined one with labels. Alternatively, *CacheManager* can be designed to check layered caches in one go: e.g., getResponse might internally check L1, if miss, then call getCompressed (which might itself use the embedding computed) etc. For clarity in implementation, we might keep them separate but call in sequence with the same embedding.

The pseudo-code below (from the SDD) illustrates an L1 lookup implementation, which we will adapt to handle multiple layers:

```
async function lookupResponseCache(query: string): Promise<CachedResponse |
null> {
    const queryEmbedding = await embeddingService.embed(query);
    // Vector search among recent cached entries (or via ANN index)
    const candidates = await db.all(`
        SELECT id, query_text, query_embedding, response_text, agent_used,
token_count, created_at, source_hash
        FROM response_cache
        WHERE datetime(created_at, '+' || ttl_seconds || ' seconds') >=
datetime('now')
        ORDER BY created_at DESC
        LIMIT 100`);
    // Compute similarity for each candidate in JS (or use ANN results directly)
    let bestMatch: CachedResponse | null = null;
    let bestScore = 0;
    for (const cand of candidates) {
        const candEmbedding = deserializeEmbedding(cand.query_embedding);
        const sim = embeddingService.cosineSimilarity(queryEmbedding,
candEmbedding);
        if (sim > bestScore && sim >= SIMILARITY_THRESHOLD.L1) {
            bestScore = sim;
            bestMatch = {
                response: cand.response_text,
                similarity: sim,
                metadata: {
                    agent: cand.agent_used,
                    tokenCount: cand.token_count,
                    sourceHash: cand.source_hash
                },
                age: (Date.now() - new Date(cand.created_at).getTime()) / 1000
            };
        }
    }
    if (bestMatch) {
        await recordCacheHit('L1', bestMatch.metadata.tokenCount);
        // Update last_accessed, access_count in DB
        await db.run(`UPDATE response_cache SET last_accessed =
CURRENT_TIMESTAMP,
                           access_count = access_count + 1 WHERE id = ?`,
[bestMatchId]);
        return bestMatch;
    }
    return null;
}
```

Example 1: L1 lookup logic. (We will not literally scan 100 entries in final code; instead we'll use ANN index. But this code shows threshold check and TTL validity [123] [124]).

For L2 and L3, similar functions exist (`getCompressedChunks`, `getChunkIds`) which do the same but query their respective tables and return either a list of compressed chunks or chunk IDs [125] [126]. The threshold values differ (we'll use perhaps 0.85 and 0.80 by default). The CacheManager API (detailed below) abstracts these.

By following this sequence strictly, we ensure that if *any* relevant cached artifact exists, it will be utilized: - If final answer is present and valid, we short-circuit. - Else if not, but intermediate context is present, we use it and just do final generation. - Else if not, but retrieval results are present, we skip retrieval. - Otherwise, we do everything from scratch.

We also handle partial scenarios: e.g., if L1 misses but L3 hits and L2 misses, we reuse chunks but still compress new. Or if L1 and L2 miss but L3 hit – that saves retrieval at least. The combination logic is handled by sequential checks.

3. Data Schema and Storage Design

We use **SQLite** for the cache store due to its simplicity and performance on a single machine. All cached data and metadata is stored in a single SQLite database (file), `semantic_cache.db`. This allows easy persistence (the cache survives restarts) and leverages SQL for querying and management. SQLite can handle our scale (hundreds of thousands of records, low write throughput) on a single node comfortably [127] [128].

The schema is divided by cache layer:

```sql
-- L1: Response Cache (final answers)
CREATE TABLE response_cache (
    id              TEXT PRIMARY KEY,
    query_text      TEXT NOT NULL,
    query_embedding BLOB NOT NULL,      -- serialized Float32 array (embedding)
    response_text   TEXT NOT NULL,
    agent_used      TEXT,              -- which agent/model produced the answer
    token_count     INTEGER,           -- tokens used to generate this answer
    created_at      TIMESTAMP,
    last_accessed   TIMESTAMP,
    access_count    INTEGER DEFAULT 1,
    ttl_seconds     INTEGER DEFAULT 604800,  -- default TTL 7 days
    source_hash     TEXT               -- hash/version of data source used
);
CREATE INDEX idx_resp_created ON response_cache(created_at);
CREATE INDEX idx_resp_last_accessed ON response_cache(last_accessed);

-- L2: Compressed Chunk Cache (after analysis/compression stage)
CREATE TABLE chunk_cache (
    id                TEXT PRIMARY KEY,
    query_text        TEXT NOT NULL,
    query_embedding   BLOB NOT NULL,
    compressed_chunks TEXT NOT NULL,   -- JSON or other serialized form of
compressed context
    chunk_ids         TEXT,           -- JSON array of raw chunk IDs used
    created_at        TIMESTAMP,
    source_hash       TEXT
);

-- L3: Raw Chunk Cache (retrieval results)
CREATE TABLE extraction_cache (
    id               TEXT PRIMARY KEY,
    query_text       TEXT NOT NULL,
    query_embedding  BLOB NOT NULL,
    chunk_ids        TEXT NOT NULL,   -- JSON array of relevant chunk IDs
    scores           TEXT,
-- JSON array of relevance scores for those chunks
    created_at       TIMESTAMP
);

-- Cache Statistics (daily aggregate metrics)
CREATE TABLE cache_stats (
    date          DATE PRIMARY KEY,
    total_queries INTEGER,
    l1_hits       INTEGER,
    l2_hits       INTEGER,
    l3_hits       INTEGER,
    misses        INTEGER,
    tokens_saved  INTEGER,
    cost_saved    REAL
);
```

Some notes on this schema: - We store the **raw query text** and **embedding** for each entry. Storing the text is optional for functionality, but it's useful for debugging and possibly for future analysis (also one could recompute embeddings if needed). The embedding is stored as a BLOB of bytes (we'll likely store float32 array directly, or possibly quantized). Having embeddings in DB means we could do brute-force similarity in SQL if needed or migrate data to another vector store easily [129] [130]. - `id` is a unique identifier (probably a GUID or UUID) for each entry. We don't use the query text as primary key because semantic similarity means different texts might map to same hit. Also, we might even allow multiple entries for semantically same queries if answers differ by user context (though in such case user_id would differentiate). - `agent_used` records which model/agent generated the answer (e.g., "gpt-4" vs "qwen-7b"). This can be used for debugging or invalidation if model changes. It's not used for retrieval. - `token_count` is stored to calculate `tokens_saved` for hits. When a hit occurs, we add that entry's token_count to a saved counter because those tokens were avoided. We also may use token_count for eviction weight (e.g., evict heavy entries first maybe) – not currently planned, but available. - `ttl_seconds` is per-entry TTL. Default 604,800 seconds (7 days). Could be set differently for certain entries if needed (e.g., an admin could cache something only for 1 day by specifying TTL in metadata). - `source_hash` is critical for **cache coherence** with data updates [48] [49]. It links to a version of the knowledge base. If the knowledge base changes (we update docs), the new queries will have a different source_hash (system updates this), and thus old entries become stale. We can find and invalidate entries where source_hash doesn't match current. We might use a single global source version (like store a version string) or compute an MD5 of all docs. Implementation detail aside, we include it in L1 and L2 (for answers and compressed context). L3 (extraction_cache) perhaps could also have it, but as it stands we didn't include source_hash in extraction_cache schema. We might add it for completeness, but even if not, an outdated retrieval result is less harmful (it just might miss a new doc). However, to be safe, we likely should include source_hash in all layers. (In code, we'll ensure not to use L3 results if source differs.) - Indices: We add indexes on created_at and last_accessed for eviction queries (find oldest, etc.) [131] [132]. We could also index access_count if doing LFU (or combine in a query). - The `chunk_cache` (L2) stores a blob of compressed data. This could be a JSON array of summary strings, or a single concatenated summary text. We'll determine format when implementing. It also stores the list of chunk IDs that went into that summary (for reference or debugging). - The `extraction_cache` (L3) stores chunk_ids and scores as JSON text. We considered normalizing this into a separate table (like one row per chunk reference), but JSON is simpler for now. Since the number of chunks per query is not huge (maybe top 5-10 chunks typically), storing them as a JSON array is fine. - `cache_stats` logs daily usage statistics. We will update this at least once per day or continuously aggregate. Fields: - total_queries: total number of queries processed that day. - l1_hits, l2_hits, l3_hits: number of queries that had a hit at each layer (we count a query in all layers it hit, or maybe exclusively – but we likely define: if L1 hit, count as L1 hit (and not also L2/L3 even if technically it also saved those stages). We might track both exclusive and cumulative. Simpler: a hierarchy: L1 hits (which implicitly also saved L2/L3), L2-only hits, L3-only hits). - misses: queries with no cache help. - tokens_saved, cost_saved: we estimate tokens saved via hits (we sum token_count for each L1 hit, and perhaps a fraction for L2 hits if they saved some context tokens). cost_saved is just tokens_saved * token_cost.

By keeping these stats, we can quantify the cache's performance over time (hit rate, savings).

Rationale & Storage considerations: - **Size & Performance:** As calculated, even 100k entries is a few hundred MB [7], which SQLite can handle on a single machine easily. Reads (SELECT) in SQLite are fast, especially when indexed; writes are slightly slower due to file locking, but our write rate (equal to query miss rate) is not high. If we had thousands of QPS, a higher-throughput DB or a distributed cache might be needed, but we are far from that. - We will use WAL mode for SQLite for better concurrency (allows concurrent reads during writes). Our pattern is many reads, few writes, which SQLite can handle well with WAL. - In case we later outgrow SQLite (in multi-node deployment perhaps), this schema can be migrated to Postgres or others fairly directly. We even considered using Postgres+pgvector for an integrated solution (which would combine DB and vector store), but that adds external dependency. Still, Appendix B in the research considered Postgres and Redis alternatives for future scaling. - **Vector Index & DB relationship:** We store the embedding in the DB as BLOB, and also maintain it in the in-memory ANN index. On startup, we may load some or all embeddings from DB into the ANN index. Possibly just the most recent N to warm it, and load others on-demand if they come up. Or we can reconstruct the whole index – 50k embeddings load is quick (couple seconds). Simpler: load all on startup into index (assuming it's at most a few hundred MB of vectors). - On inserting a new entry, we will insert into SQLite (persistent store) and *also* add to the vector index in memory [133] [134]. This keeps them in sync. If the process restarts, we rebuild index from SQLite.

Identifiers: The `id` is a unique key mainly for internal use (and vector index references). We might generate a UUID each time we insert (e.g., using some library). We can also derive an ID from content (like hash of query text) but that could collide or not be unique if same query asked twice (although ideally we'd update access_count instead of storing duplicate entry). Actually, this raises design question: should identical queries on different times create separate entries or update one? Perhaps if the query text is literally identical, we could update the existing entry (refresh last_accessed, etc.) rather than store duplicate. But if answers differ (maybe because knowledge changed or a nondeterministic LLM gave different outputs), which do we cache? It might be better to treat each answer as separate entry even if query text same – but then we need logic to choose which to return. Instead, perhaps we enforce one entry per unique query text (per user scope). For semantic similarity, if the exact text repeats, it will hit exactly. If the answer changed due to updated data, ideally the source_hash would differ and the old entry would be invalidated anyway. To simplify, we can use query_text + source_hash as a natural key – meaning at most one cached answer per exact query per data version. If the LLM gave a different answer at a later time with same source, it suggests nondeterminism; we might still replace the old answer or keep one. GPTCache literature often treats cache entries as single truth and doesn't handle multiple answers for same query (assuming deterministic or at least accepting the first answer as canonical). We will proceed with one entry per query_text (plus potential user scope). So generating id as a hash of query_text+user+source could be possible. But collisions risk or changes in spacing etc. Using a random UUID is fine and we rely on similarity, not exact match, to find it. We can allow duplicates and the ANN search might retrieve multiple past entries if the same query was asked multiple times (with different answers). But our logic currently picks the best similarity (they'd all be 1.0 similarity if same query text!). Then which answer? Possibly the latest one might be best. To address that, our SQL in lookup limits to recent entries and sorts by created_at desc [123] [22], so it would consider the newest first. So effectively, if a query was asked before, the latest cached answer would be found and used. This is acceptable. So duplicates are not harmful except they bloat the DB. We can mitigate by updating instead of inserting if the text matches an existing entry. We might implement that – but have to update embedding (should be same) and answer (if different). There is nuance, but likely we'll rarely have exact same query asked with completely different answer unless the knowledge changed (which changes source_hash making it a distinct case). So duplicates not a major issue.

In conclusion, the schema is designed for simplicity and adequate performance. SQLite gives us one-file portability and ACID for cache entries. The structure separates concerns of each cache layer, and can be extended (e.g., adding `user_id` column if we need isolation in the future, which we likely will for multi-tenant support [73]).

4. Cache Manager API and Implementation (TypeScript)

We implement the caching logic as a TypeScript module/class called **CacheManager**. This provides an interface for the application to call `getResponse`, `setResponse`, etc., without worrying about how embeddings or storage are handled under the hood [135] [136].

The CacheManager will likely be used as a singleton (or one per scope if multi-tenant), but accessed via a factory function for configuration.

Key methods in the interface:

```
interface CacheManager {
  // L1: Final answer cache
  getResponse(query: string, options?: { threshold?: number }):
Promise<CachedResponse | null>;
  setResponse(query: string, response: string, metadata: CacheMetadata):
```

```
Promise<void>;

  // L2: Compressed context cache
  getCompressedChunks(query: string, options?: { threshold?: number }):
Promise<CompressedChunk[] | null>;
  setCompressedChunks(query: string, chunks: CompressedChunk[], metadata?:
CacheMetadata): Promise<void>;

  // L3: Raw chunk list cache
  getChunkIds(query: string, options?: { threshold?: number }):
Promise<{chunkIds: string[], scores: number[]} | null>;
  setChunkIds(query: string, chunkIds: string[], scores: number[], metadata?:
CacheMetadata): Promise<void>;

  // Invalidation and management
  invalidate(pattern?: string): Promise<void>;          // remove entries
matching query pattern
  invalidateBySourceHash(hash: string): Promise<void>;  // remove all entries
with given source hash (stale data)
  getStats():
Promise<CacheStats>;                     // retrieve current stats or totals
  runEviction(): Promise<number>;                     // trigger eviction
check, returns count evicted
}
```

Supporting types:

```
interface CacheMetadata {
  agent: string;       // which agent or model produced (e.g., "beatrice" or
"gpt-4")
  tokenCount: number;
  ttl?: number;        // override TTL in seconds
  sourceHash: string;  // version hash of source data used for this entry
}

interface CachedResponse {
  response: string;
  similarity: number;   // similarity score between this entry's query and the
new query (0–1)
  metadata: CacheMetadata;
  age: number;          // age of the entry in seconds
}

type CompressedChunk = string | { /* structure of compressed chunk (e.g.,
summary) */ };
```

And we will likely have a config type:

```
interface CacheConfig {
  dbPath: string;                 // path to SQLite DB file
  vectorIndex?: "faiss" | "hnswlib" | "qdrant" | "milvus";
  maxSize?: number;               // max entries (for eviction threshold)
  similarityThresholds?: { L1: number, L2: number, L3: number };
  isolation?: "shared" | "perUser" | "perOrg";
}
```

We then provide a factory to create a CacheManager instance:

```
export function makeCacheClient(config: CacheConfig): CacheManager {
    // instantiate CacheManager (likely as a class implementing the interface)
    return new SQLiteCacheManager(config);
}
```

We might implement CacheManager as a class `SQLiteCacheManager` internally, which uses an sqlite3 client (or better, use the `better-sqlite3` library for convenience in Node, or node-sqlite). It also will initialize an ANN index (if `vectorIndex` is specified as "faiss" or "hnswlib", we either call into a native addon or maintain our own). For Qdrant/Milvus, the CacheManager might instead connect to those services for vector search and skip local ANN.

Internal workings: - `getResponse`: Will call EmbeddingService.embed(query) to get the vector [137]. Then: - If using an in-memory vector index, query it for top-K similar cached entries. Then fetch those from SQLite (via their IDs) to check full data and apply threshold. - If not using an index, it could do a SQL query as shown in pseudo-code limiting by date or using `ORDER BY random()` etc., but that's less precise. We'll likely use the vector index always for L1 because scanning even 1000 entries might miss older relevant ones. However, we may incorporate the "recent 100" heuristic as a fallback. - Evaluate similarity of the top candidate(s). If best >= threshold, return that entry (pack into CachedResponse). Otherwise return null. - Also update stats (record hit) and update last_accessed & access_count in DB for that entry [138] [139]. - `setResponse`: Will embed the query (or we might reuse the embedding if we stored it from getResponse call). Compute sourceHash if not provided (we might call a function to get current knowledge version) [140] [122]. Insert a new row into response_cache with all fields [122] [141]. Also insert the vector into ANN index (e.g., `vectorIndex.add({id, values: embedding})`) [142] [143]. Then check size and call runEviction() if needed [144] [145]. - `getChunkIds`: Similar approach but on extraction_cache table. Likely uses the same query embedding (so CacheManager might keep the last computed embedding in context to reuse for L2/L3 checks). For simplicity, we may compute it thrice unless we refactor to compute once outside. Perhaps better: have one method that returns an object like `{ l1, l2, l3 }` hits to avoid multiple embeddings. But anyway, getChunkIds will search vector index for similar past queries and find if any chunk list can be reused. If found, it returns `{ chunkIds: string[], scores: number[] }`. If not, null. - `setChunkIds`: Insert into extraction_cache (id, text, embedding, chunk_ids, scores, created_at). Also add vector to index (if we maintain a separate index or a combined one). - `getCompressedChunks`/ `setCompressedChunks`: likewise for chunk_cache table. The `CompressedChunk` could be stored as

JSON string in DB, so on retrieval we parse it back (hence returning an array of CompressedChunk objects or strings). - We will have separate or combined ANN indices for each layer. We *could* have one ANN index for all queries (regardless of layer) since the query embedding is the common key. But that index wouldn't directly tell us which layer is available for a given neighbor. Alternatively, maintain three indices (one for each table). But that triples memory for embeddings (though still not huge). Another design: maintain one index for all queries and store in the index metadata the entry's id and layer. Then a search can yield, say, results from all layers; we then pick the top L1 or L2 among them. That's complex. Easier is separate indices per layer, because similarity threshold differs anyway. We will likely do separate. - `invalidate(pattern)`: This will execute SQL DELETE or UPDATE to remove entries whose query_text matches the pattern. Possibly `DELETE FROM response_cache WHERE query_text LIKE '%pattern%'`. We must be cautious with SQL injection (use parameter binding or at least escape the pattern). If pattern is null or empty, maybe it does nothing or clears all (we might require a special call for full clear). - `invalidateBySourceHash(hash)`: SQL: `DELETE FROM response_cache WHERE source_hash = ?` (and same for chunk_cache). Or we might simply mark them as invalid by setting TTL to 0, but deletion is fine. We will also remove from vector index those entries (this is tricky; perhaps easiest is to rebuild index if many removed). - `getStats`: We can either compute on the fly (count queries/hits in memory counters) or read from cache_stats table. Perhaps we maintain an in-memory counter that we periodically flush to DB. For real-time stats, might track in memory; for persistent record, we update cache_stats daily. - `runEviction`: Implements the eviction policy. For instance: - Determine if number of entries > maxSize. `SELECT COUNT(*) FROM response_cache` (and maybe include others or consider total across all tables). - If too many, decide removal count or criteria. Simplest: remove the oldest (by last_accessed) until below threshold. This could be done with a subquery like:

```
DELETE FROM response_cache
WHERE id IN (
   SELECT id FROM response_cache
   ORDER BY last_accessed ASC
   LIMIT 100
);
```

We might do in batches if needed repeatedly. - Also remove corresponding from vector index. - Similarly consider chunk_cache and extraction_cache. They will typically have same number of entries as response_cache if every query populates all layers (one-to-one), but not exactly (some queries might not produce a compressed context if short? But our pipeline likely always has something for each). To keep it simple, we might evict from all three in parallel based on response_cache's decision. Or individually maintain sizes. Possibly easiest: tie L2 and L3 entries to L1 entries by id (we could use same id for all layers for a given query). Indeed, in schema we could have reused the same `id` for chunk_cache and extraction_cache as the corresponding response_cache entry. In our schema above, they are separate, but we *can* assign the same UUID to all three when storing a new query result. That would allow us to evict all layers of an entry together. We did design so that query_text and embedding repeat, which is somewhat redundant. If we use same id, we can store query_text/embedding in only one table and reference by id in others. But due to simplicity, we duplicated. For eviction, we might just evict L1 and then also evict any L2/L3 with matching query_text or matching id (if we set id the same manually). - We'll implement eviction focusing on response_cache primarily (since that is the largest data with answers). L2 and L3 are lighter weight (summaries and ids). Possibly if we remove an L1, we should remove its L2/L3 too to avoid orphan. We could cascade delete via a trigger if id is same. If not same, then in code do multiple deletes.

Given the complexity, an alternative is to not have a separate entry in L2/L3 for every query. But we do because intermediate results can be reused by semantically similar but not identical queries as well.

Anyway, eviction will remove from all as needed.

Cache Manager and Embedding Service integration: CacheManager will have an instance of EmbeddingService (see next section) to call for computing embeddings [146]. It may also maintain the vector indices and a connection to SQLite. We plan to use an async interface (Promises) as above to not block event loop when doing heavy operations (like embedding, DB I/O). Possibly using a worker thread for embeddings if needed, but since we can `await` and our Node server is not extremely latency-critical beyond handling one query at a time essentially, simple async is fine.

We will also ensure the caching operations themselves are efficient. The heaviest part is the ANN search. FAISS has a synchronous C++ interface; we might call it via an addon or run it in a separate thread to avoid stalling Node's loop. Or consider using a WASM or Python service if needed. At first, we might do brute-force in JS for small N (as pseudo-code does) [21] [147], then integrate FAISS later.

5. Embedding Service Details

To generate embeddings within our Node.js environment, we have a few options: - Use a Python subprocess or microservice running a model (e.g., via `sentence-transformers` in Python). Communicate via REST, RPC, or even CLI calls. This ensures we can use well-optimized models in Python (PyTorch/ONNX) but adds overhead of inter-process communication. - Use a WebAssembly or JS solution: There is a library `@xenova/transformers` that can run many transformer models in the browser or Node via ONNX or WebAssembly. It even specifically supports some embedding models. Using this, we could load the `intfloat/e5-base-v2` model in Node without Python. The performance might be slightly lower than native, but possibly acceptable for moderate QPS. - Use an external API (OpenAI etc.): We want to avoid external costs and dependency for embeddings, as that defeats some of the purpose (we'd be paying for embedding each query, though it's cheaper than answer tokens). We prefer local.

We decided to implement **EmbeddingService** such that it can either call out to Python or use a local library, depending on config. Initially, we might start with a simple Python approach using `child_process.spawn` to run a script that outputs embeddings for a given text (for development ease). Eventually, integrate a better method (maybe use Xenova's ONNX if performance is good).

Pseudo-code for EmbeddingService interface:

```
class EmbeddingService {
  private model: any;     // could be a handle to loaded model or a process
  private dim: number;
  constructor(modelName: string) {
    // Initialize local model
    this.model = loadModel(modelName);  // e.g., using xenova or spawn Python
with model loaded
    this.dim = this.model.dimension || 768;
  }
  async embed(text: string): Promise<Float32Array> {
    return this.model.encode(text);  // synchronous or Promise depending on
backend
  }
  async embedBatch(texts: string[]): Promise<Float32Array[]> {
    return this.model.encodeBatch(texts);
  }
  cosineSimilarity(a: Float32Array, b: Float32Array): number {
    // compute manually if needed
    let dot = 0, normA = 0, normB = 0;
    for (let i = 0; i < a.length; i++) {
      dot += a[i]*b[i];
      normA += a[i]*a[i];
      normB += b[i]*b[i];
    }
    return dot / (Math.sqrt(normA) * Math.sqrt(normB));
  }
}
```

In practice, if using xenova:

```
import { pipeline } from '@xenova/transformers';
...
this.model = await pipeline('feature-extraction', modelName);
...
this.model(text) => returns embedding
```

We'd have to ensure normalization (E5 requires normalized output). If the model/pipeline doesn't do it, we'll do it: after encode, divide vector by its L2 norm (the Zilliz guide notes to call `normalize_embeddings=True` when using sentence-transformers on E5 [148] [149]).

The EmbeddingService should be efficient: possibly reuse the model instance for all calls, batch if multiple come in concurrently.

Dimension and Storage: E5-base-v2 gives 768-dim float32 (3KB). We could consider storing as float16 to halve size. But calculating similarity requires float32 ideally to not lose precision. However, the similarity tolerance (0.9) might allow small error if float16. Initially, store as float32. If space becomes concern, we can compress.

Embedding Computation Cost: As said, ~20-30ms CPU per query. If performance is an issue, we can try using the GPU (perhaps by loading the model in a Torch.js or something on GPU, but Node GPU computing is not trivial without Python). Alternatively, we could offload to the existing Python environment if any (LLMC likely already uses Python for the vector DB retrieval? Possibly not, since Node orchestrator might call an API for retrieval to a Python service). If a Python retrieval exists, hooking embedding there might be an option. But likely simplest: use a self-contained Node solution to avoid extra moving parts.

We'll hide these details behind the EmbeddingService API. The CacheManager will have an `embeddingService: EmbeddingService` which it uses inside get/set methods [146] [150].

Normalization & Similarity Metric: We will ensure to use cosine similarity. We will **L2 normalize** all embeddings on both storage and query time. That way, dot product equals cosine. It's easier to have them normalized at insertion (the EmbeddingService can output normalized vectors directly). Then our ANN index can use inner product as metric. Many libraries let you specify metric; we might choose to store normalized and use simple dot.

Normalization also means we don't have to worry about length differences etc., improving consistency [151] [152].

Multiple languages or Data types: The chosen model is mainly for English (and some others possibly). For now, that's fine. If we needed code-specific embeddings (for code similarity caching, which could be separate), we might have separate caching for code queries vs plain language. But that's out of scope; we assume queries are NL.

6. Vector Search Implementation (P0 and P1)

P0: Brute-force / SQLite Only. As an initial implementation (for quick MVP), we could skip integrating FAISS and simply use SQLite and JS to find similar entries. This is straightforward using a `SELECT` query to get candidate entries and then computing similarity in code [21] [147]. The pseudo-code above does exactly that: it selects the most recent 100 entries and checks them [123] [153]. This can work if the cache is small or if queries have locality (recent queries repeat more). It's not perfect but could be acceptable day 1 while integration of FAISS is pending. We will use this approach in early testing.

P1: FAISS / HNSW integration. The plan is to incorporate an in-process ANN index as soon as practical. Likely choices: - **FAISS**: Use Node.js native addon or call via a Python binding. There's no official Node FAISS, but we can compile a C++ addon. Alternatively, use a WASM version (faiss-wasm exists, but performance might be lower). - **hnswlib**: Similar situation (C++ library). Might be simpler to wrap since it's header-only. - **HNSW via Qdrant client**: Could even use Qdrant's in-memory mode by linking its library, but probably not needed.

We may actually consider using the **pgvector** extension with SQLite. There is an interesting project or possibly `sqlite-vector` extension. If one exists that can do ANN, that'd keep things simpler. However, typical approach is external index.

So likely a small C++ module that links FAISS. If doing that, we can store the index in memory and occasionally save it. But since our DB already has all vectors, we can rebuild if needed instead of persisting index.

We have a config to switch `vectorIndex` type: - If “faiss”: we build a FAISS index (maybe IndexFlatIP or HNSW). - If “hnswlib”: use hnswlib directly. - If “qdrant”: use their REST or gRPC API to search. We’d have to run Qdrant as a sidecar service. The CacheManager would, on startup, ensure the Qdrant collection exists and push all existing vectors. Or we could run Qdrant from scratch since data persisted separately.

Choosing index type: - For simplicity, we might start with a **Flat (brute-force)** in FAISS (IndexFlatIP) which uses BLAS under the hood, possibly faster than pure JS loops and easily handles 50k–100k by C code. This ironically is brute-force but optimized, and could be fine at our scale. But if we go through the trouble, better use **IndexHNSW** so it scales to even larger N. - HNSW can give significant speed-ups at minimal accuracy loss. FAISS and hnswlib both have HNSW implementations. Given ease, maybe use hnswlib directly (it's a single .h file, integrate in C++ add-on). - Parameter: M (connections, typically 16 or 32) and ef (search effort). We’d pick M=16, ef_search maybe 64–128 to get near 100% recall. Our N=200k, these defaults are fine. - We’ll use **cosine similarity**. With normalized vectors, we use inner product in index or directly a cosine metric if library supports.

Integration steps: - On CacheManager init, if using FAISS/hnsw, load all existing embeddings from DB: `SELECT id, query_embedding FROM response_cache` (and maybe from chunk_cache etc if separate indices). - Add them to the index. If using HNSW, you can add gradually. If using FAISS, you could train an IVF if needed, but HNSW doesn’t require training. We'll go with HNSW (FAISS’s IndexHNSW). - For search, given a new embedding, query the index for top-k. It returns a list of ids (and distances). - Then we query SQLite for those ids to get their data (text, answer, etc.) and then apply threshold logic and other checks (source_hash). - On insertion, add the vector to index as well.

Qdrant Option: If `vectorIndex: "qdrant"`, the CacheManager would instead: - On startup, call Qdrant REST to create a collection (with vector size 768, cosine metric). - Upsert existing vectors (maybe batch upload). - For search, call Qdrant’s search API with the embedding and top_k. - Qdrant will return IDs and distances. - Then fetch from local DB or we could even store all metadata in Qdrant’s payload. But probably easier to keep DB as source of truth and only use Qdrant for ID list. - On insertion, upsert to Qdrant as well.

The benefit of Qdrant is simplicity of use (just HTTP calls, no C++ linking) and persistence handled. But overhead and another moving part. We leave it as a configurable path.

Normalization: We will ensure to feed normalized vectors to whichever ANN. For Qdrant, we’d use cosine metric, but might still normalize to be sure.

Parallel Indices vs Single: As noted, likely maintain separate indices for each layer’s query embeddings. However, note that the query embeddings for L2 and L3 caches are the *same embeddings* as for L1 (since they all use the user query as key). This implies we could actually just use one index of *query embeddings* for all cached queries. Then, for a given new query, we find nearest queries (embedding-wise) regardless of layer. We then check in order if any of those nearest have an L1 answer, if not, then check if any have L2, etc. But a simpler approach is searching each layer’s index separately with the same query embedding and finding nearest in that subset. Either way fine. We might do: search L1 index first (fast, few vectors if fewer answers cached than queries? Actually every query could produce an answer, so L1 likely has >= L2, L3?). Actually, L1, L2, L3 all have one entry per query. If we cache all misses too? We likely cache everything, so counts are equal. Then separate vs combined doesn’t change number of vectors. Combined search could yield some nearest that only have chunks but not answers (if query had no answer cached?), not possible – if a query is cached at L3, likely an L1 entry exists with answer too (because we only store L3 when we get an answer to store as well, even if answer not cached due to threshold? Actually if threshold fails for answer, we wouldn't store answer? Possibly not, but maybe we still store L3 and L2). Actually, consider: If two queries were similar but not enough for L1 hit, one could still get L3 hit from the other's cache even if the answer was different. So we might have an entry in L3 for one query that doesn't have an identical counterpart in L1 for the new query. But our cache entries are tied to a query; either we have all three layers for a query or none. Actually, if threshold fails for answer, we still *store* that answer in cache (why would we not? It's a new query, we store it). So every processed query will create an L1 entry. So yes, every L3 entry corresponds to an L1 entry with same id (unless we choose not to store final answers below threshold, but that concept doesn't apply – threshold is only for *using* cache, not storing; we store everything by default, except sensitive). So counts match.

Therefore, we could search in one index of all entries. But then for L3 usage, we need an entry that maybe had chunk list but not answer? That doesn't happen; if it had chunk list, it had an answer which got cached but maybe never reused if threshold was high? But it's there.

Anyway, to simplify logic, treat them separate. The code is simpler: `getChunkIds` searches extraction_cache index, `getResponse` searches response_cache index.

Distance threshold check: We set the similarity thresholds as config or constants (0.9,0.85,0.8). We compare the returned distance (if using inner product and normalized, then distance = 1 - cosine maybe if using Faiss L2? We'll likely use inner product as metric for FAISS; it returns dot products. For HNSW, we can use cosine directly or also use dot if normalized). We then confirm e.g. `sim >= 0.9`. The sim we either compute by dotting again with exact vector (for highest candidate) or if our vector index can return the score (FAISS can for inner product).

Edge Cases: If no entries in cache at all, obviously getX returns null quickly. If the vector index is empty, maybe handle that.

Thread Safety: SQLite is thread-safe in serialized mode. Our usage in Node will be single-thread mostly (the event loop). But heavy operations like ANN search – we might want to do them outside main thread to not block. Possibly not an issue at low QPS. If needed, we could spawn a worker thread for CacheManager tasks. Initially, keep it simple, do sequentially.

7. Cache Invalidation and Coherence

As touched on in Eviction and Coherence sections, we need explicit handling when the knowledge base updates. We define **sourceHash** as our mechanism: - On each query processing, the orchestrator (or retrieval component) can provide a hash representing the current state of the knowledge base (documents). For example, a version number that increments on each data reload, or an MD5 of a signature of the content. Perhaps we will simply use a timestamp of last index build if data updates are batch. - This `sourceHash` is passed into CacheManager.setResponse (via metadata) [140] [122]. It's stored with the entry. - Later, if the knowledge base is updated, the system obtains a new sourceHash (say old was "abc123", new is "def456"). We then need to invalidate entries with the old hash: - If updates are rare and large (e.g., we re-ingest docs weekly), we can just do a broad invalidation: `invalidateBySourceHash("abc123")` which deletes all entries with that old hash [154] [155]. This is simple but clears a lot, effectively resetting cache when data changes. If the changes are minor, that's somewhat wasteful, but safe. - If we had a more granular approach, we could store something like document IDs used and version per entry, and only drop those affected. But that's more complex and not initially needed. - Additionally, the cache logic at lookup time can double-check the sourceHash: if an entry is found but `entry.source_hash != currentSourceHash`, treat it as invalid and ignore it [48] [49]. This covers the case where we didn't yet purge or missed something. - We will implement an event/hook such that whenever the knowledge index `.rag/index.db` file is updated (or a reindex function is called), we trigger `cacheManager.invalidateBySourceHash(oldHash)` and perhaps also bump `cache_stats` (to count how many dropped). - We can monitor file timestamp of the index as one way (poll or in retrieval code after update). Since updates are controlled, easier to call an invalidation function in code right after reindexing.

Partial Update scenario: If in future we update only a subset of docs, one might try to only invalidate entries related to those docs. That would need tracking doc IDs per entry. Our schema doesn't track which docs were used explicitly, except via chunk_ids (which correspond to docs). Actually, we *do* store chunk_ids for each entry in extraction_cache [156] [157]. If we wanted, we could cross-reference those chunk_ids with updated docs and only drop those entries whose chunk_ids include an updated doc ID. We might not implement that now, but we have the data to possibly do it. It'd be a matter of mapping chunk_id to doc id (if chunk_id includes doc id, e.g., "file1#functionX"). Likely, yes, chunk IDs implicitly contain the source identity. So an advanced invalidation could filter extraction_cache for those containing that doc and remove associated entries. This might be an enhancement later if needed.

For now, **coarse invalidation** (all or nothing on update) is acceptable because our domain might reingest knowledge in large batches, not continuously. Given caching is more effective when data is static, frequent updates reduce its utility anyway.

AST Span Changes: If code changes, the AST chunking might change chunk boundaries. The sourceHash should reflect that by a version bump. So even if some functions remain same, the safe route is to invalidate all. If we got more granular, we'd base it on chunkIDs differences.

Time-based Invalidation: TTL is a form of timed invalidation – after TTL, entry is considered expired and not used. We enforce TTL both at lookup (the SQL query in pseudo had a WHERE to ensure not expired [123] [158]) and via eviction job that deletes them [42]. This prevents using very old answers, even if knowledge didn't formally change but maybe the answer could become outdated or irrelevant by context.

Manual Invalidation: We covered the interface. The primary use case beyond source changes is if someone finds an answer was wrong or problematic, they might want to purge it. Using `invalidate("some keywords")` one could remove it. Another possible addition: `invalidateById(id)` in case we have a specific entry to remove (we can do by pattern matching perhaps unique part of query text, or add such if needed).

Cache Consistency in Multi-instance scenario: If we had multiple instances of the app (scale-out), each has its own SQLite and own cache – basically not shared. That means their caches could diverge. For now, we assume one instance or sticky sessions such that each user hits same orchestrator. If we needed a distributed cache, we'd use a shared DB or a service like Qdrant with a single storage. Then all instances query that. That might be a future consideration (we could point multiple app servers to the same Postgres or to the same Qdrant + a shared small DB). We won't implement that now, but our design doesn't preclude it – it likely just means using a different storage backend (Appendix B covers using Redis or Postgres for such use).

8. Observability and Monitoring

To understand and trust the cache, we will monitor a set of metrics and provide visibility:

Metrics to Track: - **Cache Hit Rate (overall and by layer):** The fraction of queries served from cache (full answer) vs total. We also track partial layer hits. For example, if out of 100 queries, 30 got L1 hits, 10 got L2 (but not L1), and 10 got only L3, we'd say 30% full hits, additional 20% partial reuse. We will log counters for each layer's hits and misses [159] [160]. The `cache_stats` table can record these daily. - **Tokens Saved:** We compute how many tokens were saved by caching. Each L1 hit saved `tokenCount` tokens (the cost of generating that answer) – summing those gives total tokens saved [161] [162]. For L2 hits, we could estimate tokens saved in compression step (if our compression normally uses N tokens, which we might measure). But initially, focus on L1 which is major savings. This metric directly translates to cost saved in dollars if using API (we store cost_saved if we know token pricing). - **Latency Improvement:** We measure response times for cache hits vs misses. The system can log `duration` for each query along with whether it was a hit or miss. We then compute average latency for hits and for misses, and track over time [163] [164]. We expect a large gap (e.g., hits ~50ms vs misses ~3000ms). We can set up an alert if cache lookup ever starts taking too long (shouldn't, but e.g., if vector search goes awry). - **Cache Size & Evictions:** Monitor number of entries in cache and how often eviction runs [165] [166]. If we see eviction kicking in frequently, that might mean the cache is thrashing (maybe max size too low, or too many unique queries). We log evicted count each run and maybe expose current count. We might also track memory usage of vector index (harder to measure precisely, but can approximate or just monitor process RSS growth). - **Hit Quality Checks:** Though hard to automate, we may track the *similarity scores* of hits. For instance, average similarity of L1 hits – if we see many hits barely at 0.90 threshold, maybe fine; if some are exactly at threshold, okay; if we ever see a hit with < threshold (shouldn't by logic) or if through fuzzy logic we allow, we'd flag. Also track when a hit was served what the sourceHash was (should match current, else that's a bug). - **Error Rates:** If the EmbeddingService fails to encode (maybe model not loaded) or DB errors, we should track those incidents. We can count cache-related errors (though ideally there are none; fallback on error is just to not cache). - **Throughput:** If needed, queries per second through cache can be noted, but that's more system metric.

We will output these metrics via the existing monitoring system – perhaps CloudWatch metrics or Prometheus. E.g., increment counters for hits, misses, etc., and have a gauge for cache size. Alternatively, simply use the `cache_stats` table as a source for a dashboard (which might be fine in development but for production, an in-memory metric would be better).

Logging: We'll add log lines when a cache hit occurs (at least at debug level) indicating which layer and similarity. Also log when an entry is stored or evicted in verbose mode. This helps during initial tuning and if something goes wrong.

Alerting: Set up alerts for: - **Hit rate drop:** If historically we had, say, 30% hits but suddenly it's near 0 for some period, something is wrong (maybe the cache got wiped or embedding service down). We could alert if hit rate < X for Y time [167] [168]. - **Cache size high or thrashing:** If we see the cache count oscillating near max with many evictions, perhaps our max is too low or usage pattern too broad. No urgent alert needed unless it impacts performance, but maybe a warning if evicting thousands per hour. - **Lookup latency**

spike: If retrieving from cache (which should be milliseconds) becomes slow (>100ms consistently) [169] [170]. This could indicate an issue with the vector DB (maybe Qdrant slowed or disk issue). - **Embedding failure:** If embed calls fail repeatedly (alert on error count). - **Stale answer served:** Hard to auto-detect, but if we built something where we can detect that an answer might be wrong… Instead, perhaps monitor if any cached answer's sourceHash is old – we might keep track if any entries with old sourceHash still not invalidated and if queries come in on them (shouldn't happen as we check, but double safety).

All these observability measures ensure we can quantify the cache's benefit and quickly catch any issues impacting correctness or performance.

9. Security & Privacy Modes

(This section summarizes what was detailed in the research portion about isolation and sensitive data, now as concrete design decisions.)

We have implemented: - **Sensitive Data Filter:** In `setResponse` and similar methods, we include a check to avoid caching if the query or response contains certain patterns [61] [63]. We maintain a list of regex patterns for things like API keys, secrets, likely PII. This list can be expanded with domain-specific keywords. By default, our cache will skip those entries and just log a warning. This prevents accidental caching of confidential info. - **Isolation Field:** Our schema does not yet have `user_id` or `org_id` columns, but we plan to add them when multi-user use arises [73] [171]. We designed the interface such that CacheManager could be instantiated per user in extreme case, or we just include user context in the key to functions. Simpler: add an optional `userId` to CacheMetadata and propagate it. But that complicates the vector index usage (we'd need separate indexes per user or filtering). Many vector DBs allow filtering by metadata (Qdrant, Milvus, Weaviate do). Our in-memory index doesn't easily support that beyond having separate indexes per user. If multi-tenant is important, we might simply maintain separate CacheManager instances keyed by user or org, each with its own index and SQLite (maybe separate DB file or at least separate tables with user_id). For now, single-tenant is assumed. - **Configurable Isolation:** We will expose a config flag `isolation` (shared/perUser/perOrg). If set to perUser, we will perhaps prefix or namespace cache entries by user. A straightforward approach: incorporate `user_id` in the cache key for ANN. For example, we don't mix user vectors in one index; we could keep a map of user->index. Or simpler, at query time, after finding a candidate, ensure the user matches before accepting (we can store user_id in CacheMetadata and check). If not match, skip it. That's a filter step we can do in code if volume is small, or we maintain separate indices per user if volume large. Given likely fewer users than queries, separate indices might be fine. - Our design note: for *initial rollout*, one tenant (shared) is assumed [72]. But we have laid out how we would adjust for multi-tenant. - **Access Control:** On retrieval from DB or usage of an entry, we ensure the current user is allowed. If using separate caches, it's inherently safe. If using a global cache with a user_id column, queries will include user filter as described (user = ? OR user IS NULL for global entries) [172] [173]. The `CacheManager.getResponse` could take a user context and apply it. - **Encryption:** Not planned by default (SQLite file can be encrypted via SQLite's extension if needed). We assume environment security (the disk is not accessible publicly, etc.) [81]. - **Denial of Service Mitigation:** If someone tries to spam unique queries to poison or bloat the cache, our eviction limits growth (size cap). But they could thrash it. Rate limiting at application level is a separate concern but relevant. We should consider if an attacker could cause the cache to constantly evict useful entries by flooding unique queries. The best mitigation for that is at the API gateway or application logic to throttle abusive usage. Our cache will protect itself by evicting (so it won't OOM), but performance could degrade (constant inserts/evictions). We note this scenario; however, it's not unique to caching (spamming queries to LLM is also DoS on cost). We rely on existing rate limiting. -

Stale Data and Privacy: If a user's data is removed (account deletion), we will delete their cache entries as part of data deletion process [79] [80]. Because if not, some info could linger in an answer. So our system must integrate with data lifecycle: either keep per-user caches and drop that cache entirely when user gone, or search & remove entries containing that user's content. The latter is hard unless we tag entries by data source. We do tag by source_hash but that's global. If personal data is segmented, perUser cache solves it (just drop the whole thing). - **Consent:** If for some reason caching answers might be considered storing user content, ensure that's covered by policy (likely fine in an internal scenario, but in public product maybe allow opt-out of caching personal queries – in which case we can disable caching for certain queries or users by config). - **Logging**: We must also be careful with logs. If we log query texts for hits, that might inadvertently log sensitive info. So our logging of queries will also run through `containsSensitive` and redact or avoid logging those to not defeat our cache skip (since we said we won't cache them, we should probably also not log them fully).

In summary, the cache is built to respect user data boundaries. The design allows flexible isolation if needed and by default errs on caution by filtering sensitive content and not sharing any cache entries across contexts that shouldn't share.

10. Testing Strategy

We will implement a comprehensive test suite for the semantic cache, with tests at multiple levels:

Unit Tests: - **EmbeddingService Tests:** Use known inputs to verify outputs. For example, feed two identical sentences and check cosine similarity ~1.0 [174] [175]. Feed completely different sentences (like "hello" vs "world") and expect low similarity. We might use small dummy model or stub the embed function to known vectors for predictability. - **Similarity Calculation Edge Cases:** Test our cosineSimilarity function with trivial vectors (all zeros, which shouldn't happen after normalization – we can test that we never produce zero vector from actual model). Or extremely close vectors. If we use a threshold, test boundary conditions (0.899 should not hit if threshold 0.9, etc.). - **CacheManager Logic:** We can stub the EmbeddingService to return fixed vectors for given inputs to simulate scenarios. Test: - When cache is empty, `getResponse` returns null. - After `setResponse`, a subsequent `getResponse` for same or similar query returns the stored answer [176] [177]. - Test threshold behavior: if we insert a query and then do `getResponse` with a slightly different query with similarity just below threshold, ensure it returns null [178] [179]. - Ensure that `last_accessed` and `access_count` update on hits [176] [180]. - `setResponse` properly stores TTL and source_hash and that `getResponse` respects TTL (simulate an entry with short TTL that is expired and see it not returned). We can override current time in tests or manipulate created_at. - Eviction: We can configure a small maxSize, insert entries, call runEviction and check that the oldest or least used got evicted accordingly [181] [182]. - **Eviction function details:** Feed a set of entries with different last_accessed and access_count. Possibly factor out the selection logic to a function we can unit test by providing a list of faux entries and seeing which ones it would evict (like if we implement an LRU or LFU algorithm). - **TTL expiration logic:** Maybe override how current time is determined (could inject a time provider or use an in-memory SQLite time function). Or simpler: set TTL=1 second for an entry, then sleep 2 sec in test (or manipulate the created_at to old timestamp) and verify it's considered expired either by `getResponse` logic or by runEviction removing it [183] [184]. - **Isolation (if implemented in code at this stage):** If we add user_id handling, test that queries with different user contexts do not get each other's results [185].

We should also test **some edge conditions**: - Very long query string: ensure embedding handles it or we truncate consistently. We might define a max length (E5 can do 512 tokens, so maybe 2048 chars or so). We test that if a query is extremely long, maybe we hash it to not blow logs, but anyway embed should handle up to 512 tokens by design. - Non-ASCII characters in query (UTF-8 handling): ensure our storage and embedding handle it (they should, but e.g., emoji or CJK text). - If EmbeddingService fails (simulate throwing an error), ensure the CacheManager gracefully handles (likely by propagating the error or by bypassing cache usage). We test that path by monkeypatching embed to throw.

- If cache is disabled (config to not use caching), ensure getResponse returns null always and setResponse no-ops [186] [187]. This is part of feature flag/backout testing.

Integration Tests: We'll set up an environment close to real usage: - Possibly use a small SQLite file (in-memory) and the actual EmbeddingService with a lightweight model (or stub). - Simulate queries going through pipeline: - Query A first time -> should be cache miss (go to LLM, which we can stub to return a fixed answer). - Query A second time -> should hit cache (verify time significantly lower, and result is exactly same as stub). - Query A paraphrased -> if above threshold, should hit (simulate embedding that yields high sim). If we can't easily force real model to get just above threshold, might stub embedding for this test to ensure scenario. - Query B (different topic) -> miss and store separate. - Then ask Query A again after some time -> still hit (cache persists). - Test invalidation: e.g., update underlying data (simulate source_hash change) and ask Query A -> should *not* hit now (because source differs), ensuring coherence. - Intermediate cache test: perhaps simulate two queries that share retrieval: * e.g., Q1: "What is X?" -> retrieves chunk X, caches it. * Q2: "Tell me about X in more detail" -> Ideally, retrieval is same chunk X. If similarity between Q1 and Q2 is high but maybe not 0.9 (since wording differs), L1 might miss but L3 could hit (since essentially chunk X is same needed). We want to see that behavior: Q2 misses final answer but reuses Q1's chunk retrieval (so we can detect by checking if our retrieval call stub was invoked or not). * This might require hooking into the retrieval function in test to see if it was called. - Multi-user scenario simulation: create two CacheManager instances or one with user contexts, ensure no cross-hit. - **Sensitive data**: Simulate a query that contains "password" -> ensure after processing, no cache entry is stored (maybe by checking DB count or that getResponse on that exact query returns null indicating it wasn't cached) [63] [188]. Also ensure a console warning or log occurred if we expect that. - Fill the cache to max and one more to trigger eviction -> verify size after eviction = max and correct item was evicted (like the earliest). - If using Qdrant or external, integration test that path (though maybe as a separate test or manual since needs service up).

Performance & Load Testing: While not typical unit tests, we plan a small load test: - Insert e.g. 1000 dummy entries (perhaps by calling setResponse in a loop with varied embeddings or by direct DB injection). - Then measure `getResponse` latency for hits vs misses. Ensure it's within expected bounds (we could assert e.g. that a cache hit returns within <100ms worst-case in test environment; though in CI environment might be slower). - Concurrency: Possibly spawn multiple async calls to ensure no race conditions (e.g., two parallel queries storing at same time – SQLite might serialize writes but fine; test that no deadlocks or inconsistent state). - Memory usage: We might not test memory in automated tests, but we will manually observe via profiling with large number of entries and ensure no leaks (since we do allocate buffers, make sure to free any if needed in C++ index, etc.).

A/B Test Dry-run: We might simulate an A/B by processing a list of queries with cache off vs on to compare outputs are same (for correctness). Essentially ensure turning on cache doesn't alter answers (except making them faster). If our cache works, the answers returned from cache should exactly match the original ones cached. We can test by capturing some queries and answers in one run, then enabling cache and replaying queries and comparing responses. They should be byte-for-byte equal (unless the LLM is nondeterministic – but in a controlled test, we can stub LLM to deterministic or static answers).

We will incorporate tests in our CI pipeline so that any changes to cache code are validated. Particularly threshold or model changes could affect tests expecting a hit vs miss, so tests may be a bit brittle if they rely on actual embedding values. For that reason, many tests will stub or force certain similarity outcomes to be predictable.

Acceptance Criteria: We will have some high-level acceptance tests that simulate real usage patterns: - Achieve at least e.g. 30% hit rate in a synthetic scenario (like if we repeat queries X times). - Zero instances of incorrect caching in tests (like if we intentionally feed two different QAs and ensure they don't cross if below threshold). - The system remains stable under a certain QPS in tests (no timeouts or errors).

11. Rollout Plan (Phased Deployment & Fallback)

(Recap from research, but focusing on implementation steps.)

We will use **feature flags** to control caching: - A config `enabled: boolean` that can disable all caching at runtime (CacheManager.getResponse can just return null always if disabled). This is our quick **kill-switch** [100] [101]. - Possibly separate flags for each layer (to enable just L1 caching first, then L2, etc. for incremental rollout internally). Initially, we might only enable L1 answer caching to minimize risk, while still gathering data on L2/L3 in logs.

Phase 1: Development – We already plan to test in a dev/staging environment with known queries, possibly pre-populating cache to simulate scenarios. We'll ensure all tests pass and maybe do a demo run.

Phase 2: Shadow Mode – We deploy the caching code to production but with `enabled=false` (or a mode where it doesn't serve from cache). However, we instrument it to **record what it *would have* done** [92] [93]: - We might add in the code: if disabled but shadowMode on, then still call cacheManager.getResponse but do not use its result; instead log if it found something. - Alternatively, leave it enabled internally but do not output to user: e.g., in sidecar, if hit then *still* call LLM but in parallel, then compare. This is complicated and wasteful. Better: always call LLM (so user sees real answer), but also call cache in background. - Implementation: For each query, after getting the real answer, we can compute the embedding and check the cache (like a post-check) and log if a hit would have occurred and what answer it would've given (we can compare texts). - We'll collect logs for e.g. a week. Evaluate hit rates and verify no mismatches that are problematic (if we find any case where cached answer != actual answer for what should be same question at same time, that might indicate changed underlying data or threshold issue). - Shadow mode code will be removed or turned off once we trust the cache.

Phase 3: Gradual Enablement – Possibly enable caching for a subset: - We can do this by enabling `cache.enabled=true` for only certain requests. For example, we might use user IDs (like an allowlist of beta testers or internal users). Implementation: if user in allowlist, use cache; else bypass. - Or environment-based: e.g., turn on caching in a canary deployment server handling 10% traffic. - If not easily split by user, we might do time-based small windows where we flip it on and observe. - We will closely monitor logs and metrics in this stage. We want to see hit rates and any user feedback. If any user query yields an obviously wrong cached answer and user notices, we will catch that (either via feedback or by proactively reviewing sample Q&As). - We might run this phase for a few days to gather confidence.

Phase 4: Full On – Turn caching on for all queries by default. Still monitor but with expectation it's stable.

Throughout, we keep the ability to instantly turn it off if needed [189] [102]. Because the orchestrator always has the logic to fall back to original flow, disabling cache just removes the check (and maybe we might still update cache in background, which is okay). Thus backout is straightforward: flip flag, deploy, or even toggle at runtime if we implement a dynamic config.

We also consider a partial fallback: if something is wrong only with L1 caching but L3 is fine (for example, maybe threshold for L1 was too low causing bad hits), we could disable L1 (so always force generation) but still allow L3/L2 to operate (so it still saves tokens by skipping retrieval/compression). This could be done by setting L1 threshold to 1.01 (so never hit) or a specific flag. This is a contingency if we find any correctness issues mainly with reusing final answers.

Migration: Initially, since it's new, no migration needed. All caches start empty. If we later change the embedding model or schema, we might have to invalidate or rebuild. For instance, if we switch to a different embedding dimensionality, old embeddings in DB are not comparable. E.g., if we move to 1536-dim OpenAI, we should probably flush old cache or keep separate ones. Because similarity scores won't be comparable. Easiest is to clear the cache when changing embedding model (we can encode model name/version in source_hash or in a cache metadata global so that hits only count if model matches). Similarly, if we change vector index parameters drastically, no big issue – old remains fine, or we rebuild index.

We also plan to **document** the caching feature to the team/users: People should know that if they ask identical questions they'll get identical answers (some might expect slight variations each time from the LLM; caching changes that expectation). However, most will see consistent answers as a positive (less randomness) [190] [191]. But we should set expectations: e.g., "The system now uses caching; repeated questions will yield the same answer for efficiency." If a user wants a "fresh" answer, they might need to rephrase beyond threshold or we might provide a way (like a cache-bypass flag if truly needed, but likely not necessary).

Finally, after full deployment, we treat the cache as part of system and keep monitoring KPI (hit rate, cost saved). If at any point it underperforms or causes trouble, we have the toggle to disable (backout plan) which just returns behavior to normal (except maybe slight overhead if code still calls getResponse returning null – that overhead is negligible). So risk is low as we can always fall back to original method quickly [189] [102].

We will continue to iterate threshold settings and possibly model choice after rollout, using the data collected to fine-tune for maximum benefit.

Appendices and Additional Details:

Appendix A: Embedding Model Comparisons

(This section can list some models and their qualities that were considered, from the SDD's Section 5.1.)

For example: - **all-MiniLM-L6-v2 (SBERT)** – 384-dim, very fast (~5ms), lower accuracy. - **all-mpnet-base-v2** – 768-dim, ~110M params, strong performance, moderate speed (~20-30ms CPU) [192] [193]. - **BAAI BGE-Large** – 1024-dim, 335M params, excellent quality (top MTEB) at cost of memory (~1.3GB) [194] [195]. - **OpenAI**

Ada-002 – 1536-dim, extremely good, but external and has latency ~100ms+ per call [196] plus cost. - We chose E5-base-v2 as best trade-off. If quality issues arise, we might try mpnet or even E5-large if GPU available.

Appendix B: Alternate Storage Backends (Postgres, Redis)

Postgres (with pgvector): We could use Postgres to store cache instead of SQLite. Postgres with the pgvector extension can handle embedding storage and approximate search (it supports IVF indexes etc.). Benefits: - Scales better with concurrency (multiple app servers can share one Postgres DB and see a unified cache). - More robust management and monitoring. - Can offload vector search to the DB rather than maintain in-app. However, using Postgres introduces network latency on each lookup and complexity (managing a separate DB service). For our scale, SQLite is simpler and faster (no network, just file I/O). If we needed multi-node cache, Postgres + pgvector is a good option. We would create essentially the same tables in Postgres (or one combined table using pgvector type for embedding and an index on it). Then `getResponse` could do `SELECT ... ORDER BY embedding <-> query_embedding LIMIT 1` using pgvector's operator. We'd still need to compute query_embedding on app side and send it. That adds a bit of overhead (transferring 768 floats in a query). But it's viable. We'd also rely on Postgres for eviction maybe via a scheduled job.

Redis: Two modes to consider: - Using Redis just as a key-value for exact queries (not helpful here). - Using Redis with its RediSearch module or the new vector similarity feature (Redis 7 introduced a vector datatype for ANN search in memory). We could push all embeddings into Redis and query by vector similarity. This would give extremely fast lookups (Redis in-memory), and allow easy scaling (Redis cluster). However, we then need to ensure persistence (Redis can persist to disk, but caches could be considered ephemeral too). Redis's vector search is relatively new but could likely handle 200k vectors on one node easily. We'd tag entries with user or other metadata and use RediSearch queries to filter if needed. The advantage is we keep all caching in one external system and each app server just queries Redis, simplifying synchronization. The downside: maintaining a Redis instance with proper memory config, and we lose the tight integration with file system (but maybe gain easier dynamic scaling). It could be something to explore if we find SQLite insufficient or need multi-process access.

Combined approach: Even if we scale beyond one node, one approach is to treat each node with its own cache (non-shared). The benefit is locality and not needing distributed locking. The drawback is missed opportunities (one server might cache an answer that another server doesn't know about, so duplicate work if user hits different server). But if we have sticky sessions or a small number of servers, this might be okay. For initial deployment, a single server or sticky usage might mean SQLite works without change.

If we were to share cache, the simplest might be having the orchestrator run as a single logical service for those queries (since likely our usage isn't extremely high scale).

Milvus: Another alt, as mentioned, would be like running a Milvus instance. It's heavy but if we needed clustering beyond Qdrant. Probably unnecessary for our sizes (Milvus shines at millions+ of vectors).

Conclusion for Backends: We stick to SQLite+FAISS as it meets requirements. Postgres or Redis remain options if we face multi-instance issues or need their specific features.

Appendix C: Code Snippets and Examples

(We include some code examples already inline above, but we could add additional snippet files if needed in `docs/_snippets/cache/`*. This might include a snippet of TS for CacheManager class, and maybe PlantUML diagrams.)*

PlantUML Diagrams Source:

1. *Sequence Diagram (Query Flow with Cache):*

```
@startuml
actor User
participant "Cache Sidecar (/rag/ask)" as Sidecar
participant "LLM Backend" as LLM
User -> Sidecar: Query
activate Sidecar
Sidecar -> Sidecar: getResponse(query)
alt L1 cache hit
    Sidecar -> User: Return cached answer
    deactivate Sidecar
else Miss (no answer)
    Sidecar -> Sidecar: Stage1 getChunkIds(query)
    alt L3 cache hit
        Sidecar -> Sidecar: Use cached chunks (skip retrieval)
    else L3 miss
        Sidecar -> KB: Retrieve chunks (RAG)
    end
    Sidecar -> Sidecar: Stage2 getCompressedChunks(query)
    alt L2 cache hit
        Sidecar -> Sidecar: Use cached compressed context
    else L2 miss
        Sidecar -> Sidecar: Compress/summarize context
    end
    Sidecar -> LLM: Query + context (LLM call)
    activate LLM
    LLM --> Sidecar: Generated answer
    deactivate LLM
    Sidecar -> Sidecar: storeChunkIds(query)
    Sidecar -> Sidecar: storeCompressedChunks(query)
    Sidecar -> Sidecar: setResponse(query, answer)
    Sidecar -> User: Return answer
    deactivate Sidecar
end
@enduml
```

2. *Component Diagram:*

```
@startuml
node Client {
  actor User
}
node "LLMC Orchestrator" {
  component CacheSidecar
  component CacheManager
  database "Cache DB (SQLite)"
  component "Vector Index (in-memory)"
  component EmbeddingService
  component "Retrieval Module"
  component "LLM Backend(s)"
}
User --> CacheSidecar: Query
CacheSidecar ..> CacheManager: lookup/store calls
CacheManager ..> "Vector Index (in-memory)": ANN search/insert
CacheManager ..> "Cache DB (SQLite)": SQL read/write
CacheManager ..> EmbeddingService: get embedding
CacheSidecar --> "Retrieval Module": RAG query [if miss]
"Retrieval Module" --> "Knowledge Base": (Docs/Index)
"Retrieval Module" --> CacheSidecar: chunks
CacheSidecar --> "LLM Backend(s)": request with context [if miss]
"LLM Backend(s)" --> CacheSidecar: answer
CacheSidecar --> User: answer (cached or new)
@enduml
```

(These PlantUML diagrams correspond to the earlier mermaid diagrams and can be saved in `DOCS/SDD/_assets/semantic_cache_sequence.puml` *and* `semantic_cache_components.puml` *for example.)*

TypeScript Snippet: CacheManager usage example

```typescript
// Example usage in the orchestrator
const cache = makeCacheClient({
  dbPath: './semantic_cache.db',
  vectorIndex: 'faiss',
  maxSize: 50000,
  similarityThresholds: { L1: 0.90, L2: 0.85, L3: 0.80 }
});

// On receiving a query:
async function handleQuery(user, query) {
  let cached = await cache.getResponse(query);
  if (cached) {
    console.log(`Cache HIT (sim=${cached.similarity.toFixed(2)}) - returning cached answer`);
    return cached.response;
  }
  // Cache miss: proceed with retrieval + LLM
  const { chunkIds, content } = await retrieveChunks(query);
  await cache.setChunkIds(query, chunkIds, content.scores);  // store L3
  let compressed = await maybeSummarize(content.chunks);
  if (compressed) {
    await cache.setCompressedChunks(query, compressed);
  }
  const answer = await callLLMBackend(query, compressed || content.chunks);
  // assume we can get token count from LLM call:
  const tokenCount = answer.tokensUsed;
  await cache.setResponse(query, answer.text, {
    agent: answer.model,
    tokenCount,
    sourceHash: getCurrentSourceHash()
  });
  return answer.text;
}
```

(This is a simplified illustration; actual integration might differ, but it shows how the cache API is intended to be used in context.)

---

1 2 5 7 8 9 10 15 16 17 19 20 21 22 23 24 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 84 85 86 87 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 Semantic Caching (GPTCache) Implementation SDD.md
file://file_000000008b0071f7999ef9efcf6e4f13

3 4 6 105 [2411.05276] GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching
https://arxiv.org/abs/2411.05276

11 12 13 14 18 148 149 The guide to e5-base-v2 model | Microsoft
https://zilliz.com/ai-models/e5-base-v2

25 26 27 28 29 Vector Database Benchmarks - Qdrant
https://qdrant.tech/benchmarks/

82 83 88 89 Code_Span_Semantic_Chunking_Executive_Summary.md
file://file_00000000219471f58f02eff1e04fa3f4