ChatGPT

Semantic Caching (GPTCache) Implementation SDD

1. Executive Summary

Problem: The current LLM orchestration framework (LLM Commander) suffers high token costs and latency due to repetitive or similar queries being processed from scratch each time. For example, if 100 users independently ask *"How do I reset my password?"*, the system makes 100 full API calls to an LLM, wasting time and money. Traditional exact-match caching is ineffective for LLM queries because minor rephrasing or extra context leads to cache misses [1]. We need a smarter cache that recognizes semantically similar queries, not just identical strings, to avoid redundant LLM calls.

Solution: Implement a **Semantic Caching** layer (inspired by GPTCache) that uses embeddings and similarity search to detect when a new query is essentially the same as a previously answered query [1]. The system will convert queries into numerical embedding vectors and store LLM responses indexed by these embeddings. When a new query comes in, the cache will perform a vector similarity lookup to find if an earlier query was semantically similar; if yes, it returns the cached response immediately without invoking the LLM [1]. This caching will operate at multiple stages of the pipeline, not only final answers but also intermediate retrieval results, to maximize token savings.

Expected Outcomes: By introducing semantic caching, we aim for a **30–50% cache hit rate** on common queries, based on research and case studies. GPTCache experiments have demonstrated hit rates around 50% in controlled tests [2], and real-world applications have reported serving ~20–30% of queries from cache [3]. Hitting the cache yields **zero token cost** for that query, directly translating into cost savings. For instance, if 35% of 1000 daily queries are served from cache, and each query averages 10K tokens, we save ~3.5M tokens per day, which is ~$35 in API costs saved daily (over $1,000/month). Latency for cached responses drops to near-zero (tens of milliseconds for lookup), versus several seconds for an LLM call, dramatically improving user experience.

Cost Savings: At an estimated $0.01 per 1K tokens, avoiding 3.5M tokens per day saves ~$35/day [4] [5]. Even accounting for the overhead of running the cache (embedding computation and storage), this yields a strong ROI, potentially **saving thousands of dollars monthly** in a production deployment. Moreover, caching reduces load on LLM endpoints, enabling the system to scale to more users without proportional cost increases [6]. The semantic cache thus addresses both **budget constraints** and **performance** needs for the LLM Commander project.

2. Technical Background

Why Exact-Match Caches Fall Short: Traditional caches (like Redis key-value) require exact key matches to retrieve a cached result. In natural language queries, users can phrase the same intent in myriad ways. For example, *"reset password"*, *"how to change my password?"*, and *"I forgot my password, what do I do?"* all mean

the same thing. An exact string match would treat these as distinct keys and miss opportunities to reuse answers [7]. As GPTCache's authors note, the complexity and variability of LLM queries result in low hit rates for naive caches [8]. Thus, an exact-match approach is insufficient for caching LLM responses.

Semantic Similarity & Embeddings: Semantic caching uses *vector embeddings* to represent the meaning of queries. An embedding is a high-dimensional numeric vector (e.g., 384 or 1536 dimensions) produced by an ML model that captures the query's semantic content. Similar queries yield embeddings that are close in vector space (measured by cosine similarity or distance). By comparing a new query's embedding to those in the cache, we can find semantically related prior queries even if the wording differs [1]. For instance, *"How to reset my password?"* and *"I can't log in, how do I change my password?"* would yield embeddings with a high cosine similarity (say >0.9), indicating they're essentially asking the same thing. Using this technique, a semantic cache can catch paraphrases and return cached answers for them [1], greatly increasing the cache hit probability.

GPTCache Architecture Overview: GPTCache is an open-source semantic cache that exemplifies this approach [1]. It comprises modular components: - An **Embedding Generator** to convert queries to embeddings (supports various models: OpenAI, HuggingFace, ONNX, etc.) [9]. - A **Vector Store** to index and search embeddings (supports FAISS, Milvus, Qdrant, Weaviate, etc.) [10] [11]. - A **Cache Storage** to store the actual prompt-response pairs (supports SQLite, Postgres, MongoDB, etc.) [12] [13]. - A **Cache Manager** orchestrating retrieval and insertion, applying similarity evaluation and eviction policies [14] [15]. - **Similarity Evaluation** logic to decide if a retrieved cached entry is similar enough to reuse (e.g., threshold on cosine similarity). - **Eviction Policy** to remove old or less useful entries (GPTCache currently supports LRU, FIFO, etc.) [16] [17].

This design ensures flexibility: one can plug in different embedding models, vector search backends, or storage backends as needed [9] [18]. GPTCache has proven that semantic caching can drastically reduce LLM call volume and response latency, boasting up to 100x faster responses for cached queries [19] and significant cost reductions by serving repeated questions from cache [20] [1].

Multi-Layer Caching: Beyond final answers, our system performs multi-stage retrieval (extracting relevant data chunks, compressing them, then generating answers). There are opportunities to cache intermediate results: - **L3 (Raw Chunk IDs):** Results of initial document retrieval (the set of relevant document chunks for a query). - **L2 (Compressed Chunks):** Processed and summarized context chunks prepared for the LLM. - **L1 (Final Responses):** The ultimate answer given to the user.

By caching at each stage, we can avoid redoing work if a similar query comes in: - If two queries seek the same info, they likely retrieve the same docs (L3 cache hit). - If they ask similarly such that the needed context compression is the same, we reuse the compressed context (L2 hit). - If they are nearly identical questions, we reuse the final answer (L1 hit).

This layered approach is akin to a cache hierarchy, maximizing token savings even for partial matches. For example, if a query misses the L1 response cache but hits on L2, we skip directly to answer generation with cached compressed context, saving significant analysis tokens. Multi-layer semantic caching is a novel extension to standard prompt caching, tailored for our agentic retrieval pipeline.

3. System Architecture

3.1 High-Level Cache Flow

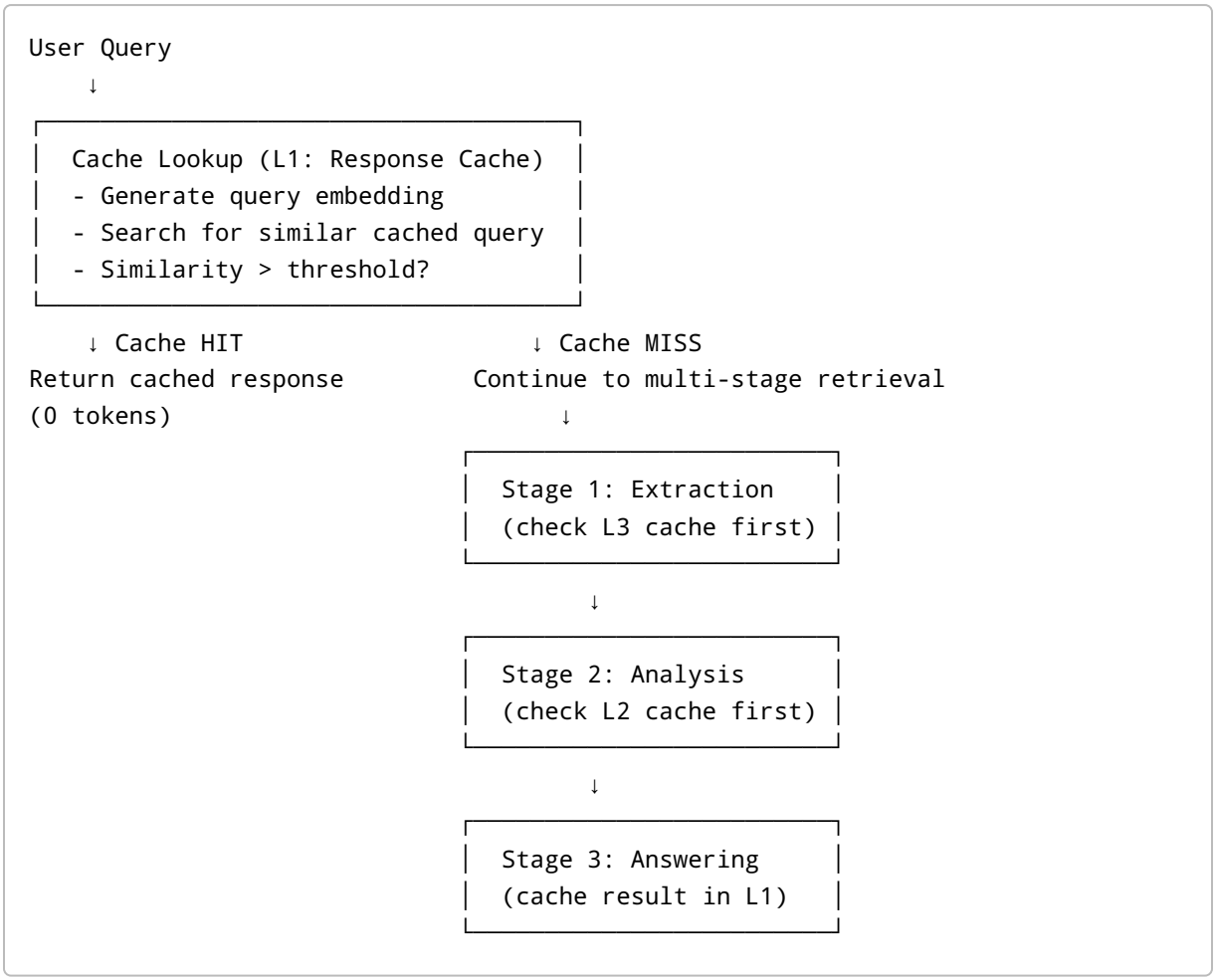

Figure: Cache-enabled query flow. The system first attempts to answer from the **L1 Response Cache** using semantic similarity search on embeddings. On an L1 cache hit, the user gets an immediate response with zero token cost. On a miss, the query proceeds through the normal multi-stage retrieval: 1. **Stage 1: Extraction** – retrieve top relevant document chunks for the query (possibly from a local RAG database). The **L3 Raw Chunk Cache** is checked before doing heavy retrieval: if a similar query's chunk list is cached, we reuse it. 2. **Stage 2: Analysis/Compression** – compress or analyze the chunks to a concise context (e.g., summarization or embedding-based compression). The **L2 Compressed Chunk Cache** is checked: if a similar query's processed context exists, skip re-processing. 3. **Stage 3: Answering** – given the (possibly cached) context, generate the final answer with an LLM. The new Q&A pair is then stored in the **L1 cache** for future reuse.

This pipeline ensures maximum reuse of prior work at each stage. Even if the final question is new, parts of its processing might overlap with previous queries, yielding partial token savings (for example, retrieving the same docs or using the same summarized context). The overhead of cache checks is minimal (vector

similarity search in a small index, typically <50ms) compared to full LLM inference latency (several seconds), thus the cache layer adds negligible latency on misses while greatly accelerating hits.

3.2 Component Diagram

- **Query Processor & Embedding Service:** When a query arrives, the system normalizes it (e.g., remove punctuation, lowercase if needed) and invokes the Embedding Service to obtain a high-dimensional vector representation. This service may use a local embedding model (for speed and cost savings) or an API. For example, we might use a SentenceTransformer model locally that produces a 384-dim embedding [21]. The embedding step takes ~10-50ms for a single query on a modern CPU or GPU, which is minor compared to an LLM call.
- **Cache Manager:** This core component coordinates the caching process. It interfaces with multiple cache layers (L1, L2, L3 as described) and the underlying storage. On a lookup, it queries the **Vector Store** for nearest embeddings to the new query and fetches the corresponding entries from **Cache Storage** (database). It then uses the Similarity Evaluator to determine if any retrieved entry is similar enough (above a defined threshold) to reuse [22] [23]. If yes, it returns the cached result. If not, it signals a cache miss and orchestrates storing new results after the LLM produces them.
- **Vector Store:** A specialized index for fast similarity search on embeddings. We plan to use an approximate nearest neighbor (ANN) algorithm (like HNSW or FAISS) to find top-k similar cached queries in milliseconds [24] [25]. The vector store can be in-memory (for speed) or a dedicated vector database. For initial implementation, an in-process library (e.g., FAISS or HNSWlib) with a moderate number of cached vectors should suffice (embedding index sizes in the low tens of thousands). As cache grows, we might move to an external vector DB (Milvus, Qdrant, etc.) to handle larger scale and persistence.
- **Cache Storage (Database):** A persistent store for cached entries (questions, answers, and metadata). We will use **SQLite** as a lightweight local database to store caches on disk, at least in the initial single-server scenario. SQLite is simple to embed and has minimal deployment overhead [26]. It can handle our expected cache sizes (thousands to low millions of entries) on a single node. In future multi-instance setups, this could be replaced or augmented by a distributed store (like Redis, PostgreSQL with pgvector, or a cloud DB) [12] [13].
- **Similarity Evaluator:** A module to evaluate how close a cached query is to the new query. The default method will be embedding cosine similarity (fast to compute), possibly combined with secondary checks (like fuzzy string match) to avoid false positives. GPTCache also offers an option of a model-based evaluator (fine-tuned mini-LM or ALBERT to judge similarity) [27], but initially cosine distance plus a threshold should be adequate. We'll tune this threshold to balance hit rate vs. accuracy of matches.
- **Eviction & Invalidator:** To keep the cache fresh and bounded in size, an Eviction Policy Engine runs periodically. It may use an LRU (least recently used) or LFU strategy, or a custom multi-factor score (incorporating age, usage frequency, etc.). This engine will prune old or low-value entries to free space. Additionally, an Invalidation mechanism listens for events that should invalidate cache entries – e.g., updates to the knowledge base (RAG index) or passage of time beyond a TTL (time-to-live) for an entry. This ensures that users don't get stale answers when underlying data has changed.

By combining these components, the semantic cache operates mostly transparently: the user issues queries as usual, and the system intercepts them with a semantic lookup first, only falling back to expensive LLM calls when necessary. This design preserves correctness (users get up-to-date answers) while maximizing reuse of prior computations wherever safe.

4. Implementation Specification

4.1 Database Schema (SQLite)

We propose to use a single SQLite database (e.g., `semantic_cache.db`) to hold all cache tables. Each cache layer has its own table, keyed by a unique ID and storing relevant data:

```
-- L1: Response Cache (final answers)
CREATE TABLE response_cache (
  id TEXT PRIMARY KEY,
  query_text TEXT NOT NULL,
  query_embedding BLOB NOT NULL,    -- Serialized vector (e.g., float32 array)
  response_text TEXT NOT NULL,
  agent_used TEXT,                  -- Which agent/model produced the answer
(Beatrice, Otto, etc.)
  token_count INTEGER,
-- Tokens used to generate this response (for stats)
  created_at TIMESTAMP,
  last_accessed TIMESTAMP,
  access_count INTEGER DEFAULT 1,
  ttl_seconds INTEGER DEFAULT 604800, -- e.g., 7 days by default
  source_hash TEXT                  -- Hash or version of data sources used (for
invalidation)
  -- Indexes for fast lookup by time (to help eviction queries):
);
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 (intermediate context after analysis/compression)
CREATE TABLE chunk_cache (
  id TEXT PRIMARY KEY,
  query_text TEXT NOT NULL,
  query_embedding BLOB NOT NULL,
  compressed_chunks TEXT NOT NULL,  -- Serialized form of compressed context
(e.g., JSON string list)
  chunk_ids TEXT,                   -- IDs of raw chunks used (for reference)
  created_at TIMESTAMP,
  source_hash TEXT
);

-- L3: Raw Chunk Cache (document 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 identifiers
```

```
  scores TEXT,                    -- JSON array of relevance scores for those
chunks
  created_at TIMESTAMP
);

-- Cache Statistics (for monitoring usage and savings)
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
);
```

Rationale: - We store both the raw query text and its embedding in the cache tables. The embedding is needed for similarity search. While the vector store index (FAISS/HNSW) will be the primary way to find similar queries, storing embeddings in the DB allows us to recompute similarity or migrate data if needed. We keep it as a BLOB (binary) for compactness. - The `source_hash` field links a cache entry to a specific state of the knowledge base. For example, it could be an MD5 of the content in `.rag/index.db` or a version number. This lets us invalidate entries if the underlying documents have changed (cache coherence). - `ttl_seconds` and timestamps allow a periodic job to delete expired entries easily. For instance, `WHERE datetime(created_at, '+' || ttl_seconds || ' seconds') < current_timestamp` finds entries past their TTL. - Indices on `created_at` and `last_accessed` facilitate eviction queries (e.g., find oldest entries or least recently used quickly). - The stats table records daily usage metrics (hit/miss counts, tokens saved). This data will feed into dashboards and help demonstrate the cache's value over time.

Storage Overhead: Each cache entry in L1 includes an embedding (e.g., 384 floats = 1.5 KB if float32) and the response text (say average 2 KB). With 10,000 entries, that's on the order of 35–40 MB of data – trivial in modern terms. Even 100k entries (~350 MB) is manageable on disk. SQLite can handle this scale on a single node, with appropriate indices and the fact that reads/writes are relatively low throughput (cache updates happen at most as often as queries). For higher scales or multi-server deployments, we consider other backends (see **Appendix B** for comparisons of storage options like Postgres, Redis, etc.).

4.2 Cache Manager Interface

The Cache Manager will be implemented as a TypeScript class (since the backend is Node.js/TypeScript) providing high-level methods to get or set cached data. This abstracts away the details of embeddings and DB operations from the main application logic.

```
interface CacheManager {
  // L1: Response Cache
  getResponse(query: string, threshold?: number): Promise<CachedResponse |
```

```
null>;
  setResponse(query: string, response: string, metadata: CacheMetadata):
Promise<void>;

  // L2: Compressed Chunk Cache
  getCompressedChunks(query: string, threshold?: number):
Promise<CompressedChunk[] | null>;
  setCompressedChunks(query: string, chunks: CompressedChunk[], metadata?:
CacheMetadata): Promise<void>;

  // L3: Raw Chunk Cache
  getChunkIds(query: string, threshold?: number): Promise<string[] | null>;
  setChunkIds(query: string, chunkIds: string[], scores: number[], metadata?:
CacheMetadata): Promise<void>;

  // Cache Management and Admin
  invalidate(pattern?: string): Promise<void>;    // Remove cache entries
matching query pattern (wildcard search on text)
  invalidateBySourceHash(hash: string): Promise<void>; // Remove entries
associated with outdated source content
  getStats(): Promise<CacheStats>;
  runEviction(): Promise<number>;                // Trigger eviction, returns
count evicted
}

interface CacheMetadata {
  agent: "beatrice" | "otto" | "rem" | string;  // which agent or model
produced/used
  tokenCount: number;
  ttl?: number;        // optional custom TTL in seconds
  sourceHash: string;  // version hash of source data used for the answer
}

interface CachedResponse {
  response: string;
  similarity: number;   // similarity score (0-1) between query and cached query
  metadata: CacheMetadata;
  age: number;          // age of the cache entry in seconds
}
```

Notes: - Each `getX` method will internally perform: embed the input query, look up similar entries (via vector search and then checking similarity), and if above threshold, return the cached data. The threshold can default to config values (e.g., 0.9 for L1, slightly lower for intermediate caches). - The `setX` methods embed the query and store the entry in both the database and update the vector index (if using an in-memory vector index like FAISS, we insert the new vector). - `invalidate` functions allow manual or programmatic invalidation. For instance, after major documentation updates, we could call `invalidateBySourceHash(oldHash)` to drop all entries tied to outdated content. - `runEviction` can

be called periodically (e.g., via a setInterval or cron job) to enforce size limits. It returns how many entries were evicted for logging/monitoring. - The CacheManager will likely maintain in-memory instances of the vector indices for fast search, while the SQLite DB is the ground truth store. On startup, it can preload recent embeddings into the vector index, or build incrementally as queries come.

By encapsulating caching in this interface, integration with the existing Node.js backend is simple: wherever an agent or the router handles a query, it first calls `cacheManager.getResponse(query)`. If a `CachedResponse` comes back with high similarity, it returns that; otherwise, proceed and then call `cacheManager.setResponse(query, answer, metadata)` with the new answer.

4.3 Embedding Service

Generating embeddings efficiently is central to semantic caching. We'll implement an `EmbeddingService` class to handle this. Python has many options (transformers, sentence-transformers), but since our orchestrator is Node.js, we have a few choices: - Use a Python microservice or WASM for embedding generation. - Use existing JS libraries (like `@xenova/transformers` which offers onnx models in JS). - Call out to an API (OpenAI or others) if needed.

To avoid external costs, we'll try to do this locally with a small model.

```
class EmbeddingService {
  private model: SentenceTransformer; // Pseudo-type for whatever backend we use
  private dim: number;

  constructor(modelName: string) {
    // Initialize a local embedding model, possibly via Python bridge or a WASM
model
    // e.g., all-MiniLM-L6-v2 loaded via a Python service or a local pipeline.
    this.model = loadModel(modelName);
    this.dim = this.model.dimension;
  }

  async embed(text: string): Promise<Float32Array> {
    // Single inference to get embedding for one text
    return this.model.encode(text);
  }

  async embedBatch(texts: string[]): Promise<Float32Array[]> {
    // Batch embeddings for efficiency when needed
    return this.model.encodeBatch(texts);
  }

  cosineSimilarity(a: Float32Array, b: Float32Array): number {
    // Compute cosine similarity between two vectors
    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));
  }
}
```

Embedding Model Choice: We will start with a **Sentence Transformers MiniLM** model (such as `all-MiniLM-L6-v2`), which produces 384-dimensional embeddings and is extremely fast (on the order of 5-10ms per sentence on CPU) [28]. This model (≈22M parameters) easily fits in memory and has proven effective for semantic similarity tasks [29]. Its quality is slightly lower than larger models or OpenAI's embeddings, but it's a reasonable trade-off for speed and cost. The vector dimension (384) is also more compact than OpenAI's 1536, which means smaller indexes and faster comparisons.

For higher quality matches, we may consider larger models or API as needed (see **Appendix A** for detailed comparison), but the architecture allows swapping the embedding generator easily. The EmbeddingService could have a configuration to switch between local vs. API mode.

Normalization: The service should apply the same text preprocessing used during training of the embedding model (most transformer-based models already handle tokenization internally, but e.g., lowercasing might or might not be needed depending on model). We will ensure consistent normalization so that semantically identical text yields matching embeddings.

4.4 Cache Lookup Implementation

Below is pseudo-code (TypeScript-like) for how the CacheManager might implement the L1 response cache lookup using the EmbeddingService and SQLite database. This demonstrates the logic of computing similarity and deciding hits:

```
async function lookupResponseCache(query: string): Promise<CachedResponse |
null> {
  // 1. Get embedding for the incoming query
  const queryEmbedding = await embeddingService.embed(query);

  // 2. Retrieve candidate cached queries via vector search.
  // For simplicity, query the DB for recent entries and compute similarity in
JS.
  // (In practice, we'd use a vector index library for large scale).
  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
  `);

  // 3. Evaluate similarity for each candidate
  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  //
seconds
      };
    }
  }

  // 4. If a sufficiently similar cached entry is found, record a cache hit and
return it
  if (bestMatch) {
    await recordCacheHit('L1', bestMatch.metadata.tokenCount);
    // Optionally update last_accessed and access_count:
    await db.run(`UPDATE response_cache SET last_accessed = CURRENT_TIMESTAMP,
access_count = access_count + 1 WHERE id = ?`, [bestMatchId]);
    return bestMatch;
  }
  return null;
}
```

A few implementation details: - We consider only entries within their TTL (`WHERE created_at + ttl_seconds >= now`) to avoid using expired data. - We limit to the most recent N entries (100 in this snippet) for performance. In practice, we'd rely on a vector index to directly get top similar ones. But even scanning 100–1000 entries is fast, and focusing on recent ones might be a heuristic since recent queries are more likely to repeat (temporal locality) [30]. - `SIMILARITY_THRESHOLD.L1` might be 0.90 by default (meaning we require at least 90% cosine similarity to consider it a hit). This threshold tuning is critical: too high (near 1.0) yields few hits; too low could return incorrect answers. GPTCache found ~0.7 was a good balance for their use case to achieve ~50% hit rate with mostly correct responses [31] [2]. We will likely start around 0.9 for final answers to prioritize accuracy (and perhaps a bit lower for intermediate cache layers). -

We update usage stats: `recordCacheHit('L1', tokensSaved)` would increment the `l1_hits` count and accumulate tokens saved (we know `token_count` from when it was stored). - If `bestMatch` exists, it's returned to the user immediately (with some wrapping to match the agent's output format). If not, the caller will proceed to do a cache miss flow (retrieval and eventually setting a new entry).

4.5 Cache Storage Implementation

Storing a new entry in cache occurs after we get a result from the LLM (for L1) or after completing a pipeline stage (for L2/L3). The implementation for L1 might look like:

```
async function storeResponseCache(query: string, response: string, meta:
CacheMetadata): Promise<void> {
  // 1. Compute embedding for the query
  const embedding = await embeddingService.embed(query);

  // 2. Compute source data hash for invalidation
  const sourceHash = meta.sourceHash || await computeSourceHash();

  // 3. Insert into the database
  const id = generateUniqueId();
  await db.run(`
    INSERT INTO response_cache
    (id, query_text, query_embedding, response_text, agent_used,
     token_count, created_at, last_accessed, ttl_seconds, source_hash)
    VALUES (?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP, ?, ?)
  `, [
    id,
    query,
    serializeEmbedding(embedding),
    response,
    meta.agent,
    meta.tokenCount,
    meta.ttl || 604800,   // default 7 days TTL if not specified
    sourceHash
  ]);

  // 4. Add to the vector index (if using external index)
  vectorIndex.add({ id: id, values: embedding });

  // 5. Optionally trigger eviction if size limits exceeded
  const currentCount = await db.get(`SELECT COUNT(*) as cnt FROM
response_cache`);
  if (currentCount.cnt > MAX_CACHE_SIZE) {
    await runEviction();
  }
}
```

This function would be called at the end of processing a query (cache miss path), to save the result for future queries. Similar `setCompressedChunks` and `setChunkIds` would store the intermediate layers: - `setChunkIds`: store the list of retrieved document chunk IDs for a query (Stage 1 results). - `setCompressedChunks`: store the compressed summary of chunks (Stage 2 results).

By storing these intermediate forms, if another query is sufficiently similar, we skip ahead to later stages.

Eviction Trigger: We include a simple check on total count. In practice, we might want a more sophisticated trigger (e.g., check if DB file size > X MB, or run eviction periodically instead of every insert for performance). But this ensures the cache doesn't grow unbounded.

Source Hash Usage: We compute `sourceHash` (likely a hash of the RAG index contents or version number of docs) at storage time. If later the knowledge base updates, new queries will get a different `sourceHash`. We can then identify stale cache entries by comparing stored hash vs current. This scheme is further explained in **Section 7 (Cache Invalidation)**.

Data Serialization: Embeddings are stored as BLOB. We'll use a simple method like writing the Float32Array bytes directly. The response and query text are stored as text (we assume UTF-8). Compressed chunks (which might be an array of strings or a JSON object) we can store as JSON text.

By implementing these storage and lookup methods, we cover the full lifecycle: when a query comes, try `getResponse` → use cached if found; if not, after computing answer, call `setResponse` to save it. The same pattern applies at other stages with their respective get/set functions.

5. Similarity Matching Strategies

5.1 Embedding Model Selection

Selecting the right embedding model is crucial for balancing performance (speed, memory) and semantic accuracy. Different deployment scenarios call for different models:

- **High-end scenario (GPU cluster, 128GB+ VRAM)**: Use the largest/highest-quality local embedding models. Candidates include:
- **BAAI** BGE Large v1.5**** (335M params, 1024-dim) – A top-performing embedding model on benchmarks like MTEB [32] [33]. Requires ~1.3GB GPU memory [34], but yields very high-quality embeddings. With ample VRAM, one could even consider running multiple such models for different languages.
- **GTE Qwen-7B** (General Text Embedding based on Qwen 7B) – Alibaba's Qwen model family has an embedding variant (7B) that produces 3584-dimensional embeddings and ranks #1 in multilingual benchmarks [35] [36]. It likely needs >10GB GPU memory for inference but offers state-of-the-art semantic understanding. On high-end hardware, this is viable and can produce extremely accurate similarity matches.
- **OpenAI text-embedding-ada-002** (API) – If local deployment is not an issue, a high-end system might still opt for OpenAI's top embedding model via API for convenience. It gives 1536-dim vectors of excellent quality and is constantly updated by OpenAI. Cost is $0.0004 per 1K tokens (roughly ~$0.0001 per short query). In a high-budget scenario, this cost may be acceptable for best-in-class

accuracy. But the latency of API calls (~100ms+) might negate some performance benefits unless we cache embeddings too.

Trade-offs: These models maximize retrieval accuracy – meaning the cache is more likely to correctly identify similar queries (fewer false misses or false hits). However, they are heavy: running Qwen-7B or similar requires strong GPUs and adds overhead to each query (perhaps 50–100ms per embedding on GPU). In a cluster, that's fine, but not for lightweight deployments. The high dimension vectors (1024–3584) also increase vector search cost slightly.

- **Mid-range scenario (8–16GB GPU or high-end CPU)**: Aim for a balance of speed and quality:
- **all-mpnet-base-v2** (SBERT) – 768-dimensional embeddings, model ~110M params. This model is known for strong performance on semantic search tasks. It might take ~20–30ms per query on CPU and can run faster on a modest GPU. It offers better accuracy than MiniLM due to higher dimensions and larger model size, at a moderate compute cost.
- **BAAI BGE Base** – A smaller BGE model (if available, likely ~110M params, 768-dim) would provide near SOTA quality with less memory (perhaps ~0.5GB).
- **Cohere Embed-xlarge** – Cohere's embedding model (English v3) produces 1024-d vectors and is available via API or on-device for enterprise. Quality is high (on par with OpenAI ada-002) [37]. If we have 8–16GB, one could host a smaller variant or call the API. Cohere's models balanced speed and quality; their medium model (512-d) might run on 8GB GPU with good results.
- **Instructor Large** (e.g., `hkunlp/instructor-xl`) – Instructor models allow using instructions to tune embeddings. They are bigger (maybe 768-d or more and hundreds of millions of params). Mid-range hardware could handle them with some optimization (mixed precision, etc.). They could yield improvements in specific domains by using task-specific embedding prompts.

Trade-offs: These options still provide strong semantic recall (likely >=90% of SOTA accuracy [38]) but at faster speeds (embedding in tens of milliseconds). They occupy moderate memory (1–4GB). This is suitable for a small team server or a desktop app where a GPU is available but not massive.

- **Low-end scenario (CPU-only or <4GB GPU)**: Prioritize speed and minimal resource usage:
- **all-MiniLM-L6-v2** – 384-dim, ~15–20ms per sentence on CPU [28]. Very light (fits in <1GB RAM). It's known to still perform reasonably well (it was specifically mentioned as a good speed-accuracy trade-off by SBERT researchers [38]). This is an excellent default for CPU-bound environments and is what we plan to use initially for development.
- **ONNX Paraphrase-MiniLM** – GPTCache uses a paraphrase-albert model in ONNX form as default [39]. ONNX-optimized models can be faster on CPU by using vectorized instructions. We could similarly convert MiniLM or use an ONNX quantized model to further speed up inference at a slight cost to accuracy.
- **FastText or tf-idf as fallback** – In extreme constrained cases, one could use simpler text similarity (FastText word embeddings [40] or even char n-gram overlap) to approximate semantic matches. These are not very accurate but require negligible resources. Given our scenario, we likely won't drop to this level, but GPTCache has a mode to disable embeddings and do keyword matching [41] as a worst-case fallback.

- **API usage threshold** – Another low-resource approach: do semantic caching with a local small model and if unsure about a similarity (score borderline), call a robust API embedding as a double-check. This hybrid ensures we mostly use local cheap computations, and only occasionally incur API

cost to verify a potential hit. However, this complicates logic and may not be needed if thresholds are well-tuned.

- **Cost-optimized scenario:** Decide between local vs API based on query volume and hardware:

- If the query volume is low or sporadic (e.g., a personal assistant app), using OpenAI's embedding API might be cheaper and simpler than running a dedicated model server. At $0.0004 per 1K tokens, a short query embedding might cost ~$0.0001–0.0002. If one expects only a few hundred queries per day, that's pennies – far less than maintaining a GPU instance. In such cases, enabling OpenAI embeddings and caching the embeddings themselves could be cost-effective. The cache could even store the embedding vector to avoid re-calling OpenAI for identical queries.
- If volume is high (enterprise scale, many thousands of queries daily), investing in a mid-range GPU to serve embeddings will pay off. The cost of a cloud GPU might be ~$1000/month, but it can handle millions of embeddings, whereas the API cost for millions of queries would be larger. Thus, break-even analysis should be done. For example, ~5 million embeddings (roughly 5 million queries) would cost $2000 via API. If you anticipate that scale monthly, a local model is cheaper.
- Also consider **failover**: as done with LLMs (7B -> 14B -> GPT-4 fallback), we can have a tiered embedding approach. For instance, try a quick-and-dirty local embedding first. If it finds a near-exact match (very high similarity), use it. If not, maybe use a higher precision embedding (like call the API or use a bigger model) to double-check the cache for a match. This ensures we don't miss hits due to weaknesses of the small model. This complexity may not be needed initially, but we design with the flexibility to incorporate such strategies.

Embedding Dimensions & Impact: Some models output larger vectors (e.g., 1536 for OpenAI, 3584 for Qwen-7B embed). Larger vectors can improve retrieval accuracy (capture more nuance) but make each similarity comparison slower and indexes bigger. Our design of using ANN indexes mitigates the per-comparison cost (ANN can handle high-dim fairly well with HNSW). However, memory usage grows: storing 50k embeddings of 1536 dims is ~300 MB (float32), whereas 384 dims would be ~75 MB. We consider the scale – at our expected cache sizes (tens of thousands entries), even 1536-d is fine. If we scaled to millions of entries, vector dimensionality would start to matter for RAM (millions of 1536-d vectors could be many GBs). Then, using PCA or a smaller model might be considered to reduce dimension.

Quality Metrics: We will evaluate models by their recall on semantic search tasks (possibly using a subset of our queries or standard datasets). According to literature, all-MiniLM-L6-v2 offers about 85-90% of the accuracy of larger models on sentence similarity, while BGE-large or OpenAI are among the top performers (100% relative). A Reddit user aptly summarized: *"for high accuracy use a large model, for speed use MiniLM"* [38] . We'll take that advice: start with MiniLM for dev and potentially allow plugging in larger models if needed for production accuracy.

5.2 Similarity Thresholds and Hybrid Matching

We define threshold levels for what "similar enough" means. Preliminary configuration:

```
const SIMILARITY_THRESHOLD = {
  L1: 0.90,       // Response cache: only hit if cosine similarity >= 0.90
  L2: 0.85,       // Compressed chunk cache: slightly lower threshold
```

```
  L3: 0.80        // Raw chunk cache: lowest threshold for reuse
};
```

These values can be tuned based on experimentation: - L1 (final answers) uses the highest threshold because returning a wrong answer is high cost. We prefer to miss some hits than deliver an incorrect cached answer. For instance, a similarity of 0.75 might indicate related topics but not identical questions. - L2 can be a bit lower – even if the question is slightly different, reusing a compressed context might still be fine (the final LLM can adjust the answer). But if it's too low, we risk using irrelevant context. - L3 lowest – retrieving roughly the same documents is often okay even if query is a bit broader/narrower, as long as the main subject is similar.

These values will be empirically adjusted. GPTCache's experiments effectively used ~0.7 as a threshold to decide a hit [42], achieving ~50% hit ratio with good accuracy. Our initial 0.9 is conservative; we might find we can drop it to 0.8 or so in practice to boost hit rate without many bad results, especially if combined with verification steps.

Secondary Fuzzy Matching: In cases where the embedding similarity is borderline (just below threshold), we can incorporate a secondary check using text-based similarity to avoid false misses. For example: - Compute a **Levenshtein distance** or token overlap between the new query and the cached candidate's text. If the wording is extremely close (e.g., one word difference), even if embedding similarity was 0.88, we should treat it as a match. - Conversely, if two questions share many words but embeddings are low (maybe the user asked a similar question but context or negation made it different in meaning), textual similarity can catch that.

Pseudo-code example:

```
if (sim >= 0.85 && sim < 0.90) {
  const normalizedQ = normalize(query);
  const normalizedC = normalize(candidate.query_text);
  const levDist = levenshtein(normalizedQ, normalizedC);
  const stringSim = 1 - (levDist / Math.max(normalizedQ.length,
normalizedC.length));
  if (stringSim > 0.95) {
    // treat as similar enough
    sim = 0.90; // bump it up to be considered a hit
  }
}
```

This says: if the cosine similarity was just slightly under threshold but the queries differ by maybe punctuation or one word (stringSim 95%+), then consider it a match. This scenario might occur with trivial rewordings that the embedding model for some reason didn't cluster strongly.

Exact Substructure Matching: We will also detect if the query is a sub-query of a cached one (or vice versa). E.g., user previously asked a detailed question "Explain X in context Y", and now asks "Explain X". The second might have high similarity anyway, but if not, our system might not directly reuse because context

differs. In some cases, though, using a cached answer that had extra context might still answer the shorter query. This is a complex area, but we note it for potential improvement.

Template Matching: Some queries follow templates, like "How do I reset my <X> password?" where X could be product name. We can have logic to strip out or parameterize such variables before hashing to detect structural repeats. However, our embedding approach largely handles this by itself (since "reset Gmail password" vs "reset Twitter password" will yield high similarity for the common part "reset password", but maybe not enough to reuse the answer, because the product differs). Template matching might be more useful if we identify patterns of queries where only a proper noun changes – in which case we might partially reuse an answer and just swap in the noun. That is advanced (almost answer templating) and outside scope for initial design, but noted as an idea.

In summary, our similarity strategy is primarily embedding-based, augmented with simple fuzzy checks. This hybrid ensures we capture semantic equivalence, and also handle edge cases of near-duplicates that embeddings might mis-rank.

5.3 Avoiding False Positives

A major concern with semantic caching is returning the wrong answer to a question that wasn't actually asked (false positive cache hits). To mitigate: - We set high similarity thresholds as described. - We can incorporate a **validation step**: For high-stakes use, one could run a secondary check by actually asking the LLM to compare the new query with cached query or answer (as GPTCache did by prompting ChatGPT to rate similarity in their experiment [43]). This is expensive (calls the LLM), but could be done asynchronously or selectively (e.g., if sim is borderline, or periodically to audit cache results). GPTCache's fine-tuned ALBERT model for similarity evaluation is an offline version of this idea [27]. - We ensure cached responses carry context of what they answered. E.g., if the knowledge base version (`source_hash`) changed, we won't use it, even if query is same. This prevents a scenario like caching an answer "Price of stock X is $100 as of yesterday" – next week that answer is invalid, so source_hash invalidation will drop it. Time-based TTL also helps limit this. - For multi-turn contexts (if any), we would include conversation state in the cache key. But in our scenario, queries are mostly single-turn with context retrieval, so not an issue. If users might have their own session context, we consider including a session ID or recent history summary in the caching key to avoid cross-talk.

By carefully calibrating these measures, we aim to maximize correct cache hits (high precision ~ 95%+ correct among hits as GPTCache achieved [44] [45]) while still getting a solid recall (serving a good fraction of repeat queries).

6. Cache Eviction Policies

Caching brings storage and memory overhead, so we need eviction strategies to control growth and remove entries that are no longer useful. We propose a **multi-factor eviction policy** that considers several signals:

- **Age (Time):** Old entries are more likely to be irrelevant (especially if data changes or user interest shifts). We can impose an absolute TTL as discussed (so entries naturally expire after X days). But beyond TTL, age can be a weighted factor – e.g., older than a certain threshold gets a higher chance of eviction.

- **Usage Frequency:** Entries that are frequently accessed (cache hits) should be retained (they clearly save cost). Entries never or rarely hit are prime candidates to evict (especially if old).
- **Recency (LRU):** We might lean towards removing entries that haven't been accessed in a long time, as they likely won't be needed soon (unless they have a long TTL for a reason).
- **Size of entry:** If some responses are huge (very long text), we might account for that by treating them as more costly to keep. However, in our text-based scenario, difference in response size might be minor (and SQLite can handle text fine). If we cached images or other large objects in future, size-based eviction would matter.
- **Source staleness:** If we know an entry's `source_hash` is outdated (the docs changed), that entry should be evicted or at least invalidated first, even if it was frequently used (because it's now potentially wrong). So we integrate with invalidation logic for that (which is separate from general eviction but related).

Eviction algorithm outline: 1. Periodically (say, when cache exceeds a threshold size or every N hours), gather candidates that are evictable. Possibly all entries or those beyond some minimal age. 2. Compute an **eviction score** for each candidate. For example:

```
score = w1 * (age_in_days) + w2 * (1 / access_count) + w3 * (last_accessed_age)
+ w4 * (is_stale ? 1 : 0)
```

where w1..w4 are weights. Here, a higher score means more likely to evict. `is_stale` could add a big chunk if true. `1/access_count` means frequently accessed (count high) yields small, thus low score (less likely to evict). We might also subtract some factor for very recent hits. 3. Sort entries by score descending (highest = worst). 4. Remove top K% of entries or enough entries to get under size limit. E.g., remove 20% of entries with worst scores. 5. Also, explicitly remove any expired by TTL immediately.

A simpler approach initially is to implement an **LRU** or **LFU**: - LRU (Least Recently Used): track last_accessed timestamp; evict the oldest not recently used entries. - LFU (Least Frequently Used): evict those with lowest access_count. GPTCache supports LRU and FIFO out of the box [16]. We might start with LRU for simplicity (common and generally good for caches) – this basically approximates that if something hasn't been used in a long time, toss it.

However, LRU alone might keep an entry that was accessed once recently, even if it's not very useful, and evict something heavily used a bit longer ago. So combining frequency is beneficial (thus **LRFU** or an adaptive scheme like **ARC** could be ideal; ARC is an advanced policy that tracks both recency and frequency).

Initial Implementation: We can do: - Maintain a global `MAX_CACHE_SIZE` (number of entries or total tokens). For example, max 10,000 entries in L1. - When inserting, if size > max, evict entries until size <= max. To choose evictions: - Sort by `last_accessed` ascending (oldest first) and evict in that order. This is pure LRU. - If we also track frequency, we could sort by last_accessed with a secondary key of low access_count (i.e., among equally old, evict the ones that were rarely used). This is a simple tweak to approximate LFU+LRU.

Given moderate scale, this approach is fine. Complexity O(N log N) for sorting N entries on eviction – with N=10000, trivial cost occasionally.

We should also be cautious about vector index consistency: if we remove an entry, we need to also remove its vector from the ANN index to avoid returning a vector that no longer has a valid cache entry. In FAISS/HNSWlib, deletion is either not trivial or handled by lazy marking (depending on library). We may simply rebuild the index from DB periodically if needed (since N is small, rebuilding 10k vectors is fast). Alternatively, many vector DBs allow soft deletes or have their own TTL mechanism.

6.1 Example Eviction Code (Pseudo-TS)

```
async function runEviction(): Promise<number> {
  const maxEntries = 10000;
  const { count } = await db.get(`SELECT COUNT(*) as count FROM
response_cache`);
  if (count <= maxEntries) return 0;
  const toRemove = count - maxEntries;

  // Find 'toRemove' oldest, least-used entries
  const victims = await db.all(`
    SELECT id
    FROM response_cache
    ORDER BY
      -- order by last accessed (older first)
      last_accessed ASC,
      -- then by low access_count (fewest hits) first
      access_count ASC
    LIMIT ?
  `, [toRemove]);

  const ids = victims.map(v => v.id);
  if (ids.length > 0) {
    await db.run(`DELETE FROM response_cache WHERE id IN (${ids.map(() =>
'?').join(',')})`, ids);
    for (let id of ids) {
      vectorIndex.remove(id);  // also remove from vector store
    }
  }
  return ids.length;
}
```

This code evicts the necessary number of entries to get back to limit. By ordering by last_accessed and access_count, it essentially implements: drop the least recently used, and among those that haven't been used in a while, drop those that were seldom used first. This is a heuristic that should work well in practice to keep cache efficient.

We will refine this as needed (e.g., incorporate TTL expiration directly or handle other tables similarly if needed).

Monitoring Evictions: We plan to log evictions (how many entries, maybe which IDs or their age) and include them in metrics. If we see frequent evictions removing items still in use, we may enlarge the cache or adjust policies.

6.2 TTL and Auto-Expiration

As mentioned, each entry has a `ttl_seconds`. A background task (maybe every hour) will delete expired entries:

```
DELETE FROM response_cache
WHERE datetime(created_at, '+' || ttl_seconds || ' seconds') < datetime('now');
```

This ensures, for example, dynamic info doesn't live beyond its freshness window. We will configure TTL per content type (see **Section 7**).

Notably, if an entry is still frequently accessed but passes TTL, this strict approach would drop it. An alternative is to refresh TTL on access (i.e., extend it if it's being used). But that could let stale data live forever if it's popular (which might be problematic if underlying data changed). Instead, we treat TTL as a hard cap unless manually overridden.

In combination, TTL expiration and LRU eviction will maintain a healthy cache where entries are relevant and the size is bounded.

7. Cache Invalidation & Freshness

Eviction handles generic cache size management, but **invalidation** focuses on correctness – ensuring we don't serve outdated or inappropriate content from cache. Several invalidation strategies:

7.1 Time-Based Invalidation (TTL)

Different queries have different "shelf lives". For example: - Answers based on **static documentation** (manuals, code archives) can be cached longer (maybe 30 days) because the content rarely changes. - Answers relying on **rapidly changing data** (like "today's weather" or stock prices) might only be valid for minutes or hours. - By default, we set TTL = 7 days for responses (as seen in schema). But we will adjust per context:

```
const TTL_POLICIES = {
  stable: 30 * 24 * 3600,     // 30 days for static knowledge
  default: 7 * 24 * 3600,     // 7 days general
  dynamic: 24 * 3600,         // 1 day for user-generated or frequently updated
info
  realTime: 3600              // 1 hour for truly real-time data
};
```

We may tag entries at creation with a TTL category. For instance, if the query triggers a tool that fetches live data, we label it `realTime`. If it's an internal Q&A about codebase (which changes slowly), use `stable`. The `CacheMetadata.ttl` field can override the default.

The CacheManager will use the lower of default vs provided TTL. And the expiration job enforces it.

Auto-expiry: As described, we'll run a periodic cleanup to remove expired entries. This prevents serving stale answers indefinitely. It's a safety net in case other signals are missed.

7.2 Event-Based Invalidation

This is critical for integration with our retrieval system (RAG). When the source data changes: - If `.rag/index.db` (the SQLite knowledge base for docs) is updated (e.g., new documents added or existing ones updated/deleted), then any cached answers that drew from that data might be outdated. - We plan to maintain a `source_hash` (e.g., a hash of all docs' last update times, or a version number that we increment whenever data changes). This could be as simple as computing an MD5 of the concatenation of document IDs and their update timestamps. - On each query, we include the current `source_hash` in the metadata. If later the global `source_hash` changes, we know all previous entries carry an old hash.

Implementation: - Whenever `.rag/index.db` is modified (we can watch the file modification time or have the code that updates it call an invalidation hook), we do:

```
DELETE FROM response_cache WHERE source_hash != NEW_HASH;
DELETE FROM chunk_cache WHERE source_hash != NEW_HASH;
```

This effectively wipes out any cache entries that aren't based on the latest data version. - This might be coarse (if only one doc changed, potentially many answers that weren't affected still get invalidated). A more granular approach is possible if we tag cache entries with specific document IDs used. But that complexity might not be necessary initially; a conservative full invalidation on any data change ensures consistency (freshness first). - If our data updates are frequent or continuous, a full flush might undermine caching benefits. In that case, we'd refine to only evict entries related to changed content. We could store in cache the list of doc IDs that were used for the answer (like in metadata). Then only drop those entries whose doc IDs intersect the changed ones. This is a more complex query but feasible (especially if we index a join table of entry->docIDs).

For now, since our RAG likely updates relatively infrequently (manual re-ingestion of knowledge or code updates), a simple approach is acceptable.

Manual Invalidation API: We will also expose `invalidate(pattern: string)` in CacheManager. This allows developers to flush cache entries by query pattern. For example, if we know many users will ask a new variant of an old question and we want to ensure a fresh answer, we could do `invalidate("reset password%")` to drop all entries whose query_text begins with "reset password". Or `invalidate("%weather%")` if weather answers might be stale. This uses a SQL `LIKE` query as in the code snippet provided in design (which we would implement similarly in TS by calling the DB).

7.3 Semantic Drift Detection

Even with TTL and event invalidation, there's a subtle issue: **semantic drift**. This is when the cached answer was correct at caching time, but over time, even if source documents didn't explicitly change, the *context* or expected answer might change. For instance, a policy might have changed or a code snippet might not be relevant after refactoring, etc., without an obvious single document update (maybe it's implicit). Or simply our cache might accumulate slight inaccuracies due to threshold not being perfect.

To tackle this, we can schedule a background validation: - Sample a portion of cache entries periodically (say 10% each day). - Re-run the original query through the current system (bypassing cache) to get a fresh answer. - Compare the new answer with the cached answer. We could use an automated similarity check (embedding of the answers, or even ask the LLM to verify if the cached answer is still valid given current context). - If the new answer differs substantially from the cached one, we conclude the cached one is likely stale or wrong now, and invalidate it.

This is akin to cache entries having to prove themselves occasionally. It incurs some overhead (calls to LLM), so we'd do it sparingly (maybe off-peak hours maintenance).

Pseudo-process:

```
for each entry in sample(response_cache, 0.1):
   freshAnswer = await fullPipelineAnswer(entry.query_text);
   if (lowSimilarity(freshAnswer, entry.response_text)) {
       await db.run('DELETE FROM response_cache WHERE id=?', [entry.id]);
       log(`Invalidated cache for query "${entry.query_text}" due to drift.`);
   }
}
```

We can compute similarity between two answer texts via embeddings or an LLM rating as in GPTCache's method [43]. If similarity < 0.8, for example, it's likely drifted.

This procedure ensures that over time, long-lived entries are re-validated. It effectively extends the concept of TTL with a content-aware check. In practice, if TTL is already reasonably low (like a week) and source changes cause flushes, drift might not be a big problem. But for caches we keep for very long (e.g., static Q&As we keep 30 days or indefinitely if popular), it's useful.

7.4 Sensitive Data and Privacy-Based Invalidation

We must ensure we *never* cache information that should not be persisted or shared. The code snippet in Privacy section outlines detecting sensitive patterns (API keys, passwords, credit card numbers, etc.) and simply skipping caching for those cases. That prevents a scenario where, say, a user asks "What's my API key usage?" and the answer includes their key – we wouldn't want to store that at all.

Additionally, if inadvertently cached, an admin might need to purge it: - We can provide an admin tool to search cache for regex matches (like any entries containing a 16-digit number) and remove them. This is more operational, but we note the capability.

Our invalidation and eviction policies combined will strive to ensure the cache is an aid that doesn't serve outdated or sensitive content. With time-based expiry, event-driven flushes on updates, and careful insertion control, we mitigate most risks of stale or incorrect data being served.

8. Integration with Existing Systems

The semantic cache must integrate seamlessly with the LLM Commander's workflow: the multi-agent system (Beatrice, Otto, Rem with a coordinating router), and the retrieval augmented generation (RAG) pipeline. The goal is to minimize changes to the core logic, using the cache as a plugin to intercept and supply answers when possible.

8.1 Multi-Stage Retrieval with Cache Hooks

We modify the orchestration pseudocode (from `codex_wrap.sh` or the Node backend) as follows:

```
async function handleUserQuery(query: string): Promise<string> {
  // Stage 0: Try full response cache
  const cachedResp = await cacheManager.getResponse(query);
  if (cachedResp && cachedResp.similarity >= SIMILARITY_THRESHOLD.L1) {
    console.log(`L1 cache HIT for query "${query}" (sim=$
{cachedResp.similarity.toFixed(2)})`);
    return cachedResp.response;
  }

  // Cache miss: proceed with retrieval stages
  let chunkIds: string[];
  const cachedChunks = await cacheManager.getChunkIds(query);
  if (cachedChunks) {
    console.log('L3 cache HIT: using cached document chunk list');
    chunkIds = cachedChunks;
  } else {
    chunkIds = await
extractRelevantChunks(query);  // Stage 1 extraction (calls RAG index)
    cacheManager.setChunkIds(query, chunkIds, scores);  // store retrieved doc
list in L3
  }

  let compressedChunks: CompressedChunk[];
  const cachedCompressed = await cacheManager.getCompressedChunks(query);
  if (cachedCompressed) {
    console.log('L2 cache HIT: using cached compressed context');
    compressedChunks = cachedCompressed;
  } else {
    compressedChunks = await analyzeAndCompressChunks(chunkIds); // Stage 2
analysis/compression
    cacheManager.setCompressedChunks(query, compressedChunks);
```

```
  }

  // Stage 3: Generate answer using the chosen agent (e.g., Otto for reasoning)
  const answer = await generateAnswer(query, compressedChunks);

  // After getting the answer, store it in response cache (L1)
  cacheManager.setResponse(query, answer, {
    agent: "otto",
    tokenCount: estimateTokensUsed(compressedChunks) + estimateTokens(answer),
    sourceHash: await computeSourceHash()
  });

  return answer;
}
```

Key integration points: - **Before Stage 1**: Check L1 cache via `getResponse`. If hit, skip all stages and directly return. - **Before Stage 1 extraction**: Check L3 cache `getChunkIds`. If hit, skip the expensive retrieval from the database (which also saves some token possibly if that involved query embedding for the DB). - **Before Stage 2 analysis**: Check L2 cache `getCompressedChunks`. If hit, skip running the summarizer or analysis agent on the chunks. - After Stage 3: store the final answer in L1 via `setResponse` for future reuse.

This ensures each stage only executes if needed. For example: - If two users ask the same question, the second user will hit L1 and get answer immediately. - If two users ask slightly differently but needing same docs, second user might miss L1 but hit L3 and L2, meaning we still avoid repeating chunk retrieval and analysis (which might be heavy tasks). - If a query is new but similar to a prior topic, maybe it hits L3 only: it'll use the same docs but then produce a new answer.

This multi-layer reuse is especially valuable for our multi-agent scenario where one agent's output feeds the next. We effectively short-circuit agent operations if we have cached outputs.

Agent-Specific Considerations: - The `agent` metadata in cached responses tracks which agent (LLM) generated it. For example, if Beatrice (OpenAI) answered a code question, and later Otto (Claude) gets a similar question, should we reuse Beatrice's answer? Possibly yes if it's a good answer. But if agents have different styles or one is more up-to-date, mixing might be an issue. In initial design, we won't differentiate – a correct answer is a correct answer, regardless of agent. We note which agent so we can analyze if some agents' answers are more often reused. - In the future, we could have separate caches per agent if needed (to ensure style consistency), but it's likely overkill. Sharing maximizes hit rate.

Routing Layer: The cache integration likely happens at the routing layer that currently decides which agent to use. Perhaps the router can first see if the query can be answered from cache without invoking any agent at all. This is simplest and avoids agent selection entirely on a hit. The router would only call an agent if cache misses.

Tool Interactions: If an agent uses tools (MCP Tools), the questions might be of form that cache can handle directly or not: - If a query triggers a deterministic tool (e.g., "What's the weather in X?" triggers a weather

API call), caching that final answer might not be wise beyond a short TTL because it's real-time. However, we might still cache for maybe a few minutes as micro-optimization. - For queries that do calculations or file access, if the underlying data changes (file changed), our source_hash mechanism should catch it (assuming source includes the file's state). But if it's an external API, we treat it dynamic. - Possibly mark such queries as `realTime` TTL.

Testing the Flow: We'll test scenarios: 1. Identical repeat query -> should return from L1, verify no agent calls made. 2. Slight paraphrase -> likely L1 miss but maybe L2/L3 hits, check partial reuse (log messages as in code help show it). 3. Completely new query -> all misses, goes through normal route, then caches.

9. Performance & Cost Analysis

We base expectations on both theoretical considerations and published results (GPTCache benchmarks, Microsoft's Azure case study, etc.):

9.1 Cache Hit Rate Projections: - If the system has many repetitive user questions (common in support scenarios or documentation FAQs), **exact repeats** might be 5–10%. These are cases where the query is literally the same or differs only by trivial words. These are easy wins for caching. - **Semantic rephrases** could account for a larger share: Users often ask the same thing in different ways. We estimate 20–30% of queries have semantically similar counterparts previously asked, given a user base that explores similar issues (based on GPTCache's 50% hit in a mixed test [2] and Microsoft noting 20–30% in practice [3]). If our cache thresholds are a bit stricter, maybe around 25% of total queries would hit. - **Partial matches** (where maybe final answer differs but intermediate retrieval overlaps): perhaps another 10–15%. For example, many users might ask about different aspects of the same document – each question is different (cache miss L1), but all retrieve the same document, so L3 could hit. - Summing up, initial target **L1 hit rate ~30-40%** of queries, and including L2/L3 partial saves possibly **~50% of queries benefit** from the cache in some form. These numbers align with literature where 30% cache usage was a conservative expectation [3] and up to 60% under ideal conditions [4].

We will measure these metrics once deployed. Achieving even 30% direct hits significantly cuts costs, and partial hits reduce compute time.

9.2 Token Savings and Cost Impact: Using an example baseline: - Without cache: 1000 queries/day * 10,000 tokens each (including retrieval and answer) = 10 million tokens/day consumed. At OpenAI rates ~$0.002/1K for GPT-4 or $0.0002/1K for GPT-3.5, cost could vary; but let's use an average $0.01 per 1K tokens (mix of models). - That's $100/day cost.

With cache: - 30% (300 queries) served from cache at 0 tokens. The remaining 700 still use the pipeline. - Those 700 might also save some tokens if they hit L2/L3. Hard to quantify exactly, but if, say, half of them skip some context tokens (like compression saved 60% tokens for those queries), that's additional reduction. - Roughly, the 700 might effectively consume only ~70% of the tokens they would have without cache due to partial caching. So instead of 700*10k = 7M tokens, maybe 5M tokens for them. - Total tokens with cache ≈ 5M (partial hits) vs 10M baseline – a 50% reduction overall in this optimistic scenario.

In dollars: baseline $100/day, with cache ~$50/day, saving ~$50 daily (50%). More conservative, if we only save 30% of tokens, we go to $70/day, saving $30.

Either way, monthly savings hit four figures easily (e.g., $30/day is ~$900/month). As usage scales, these savings scale linearly.

Also important, **rate limit savings**: By cutting number of LLM calls (especially expensive ones like GPT-4) by ~30%, we reduce risk of hitting rate limits and can serve more users concurrently. Cached queries essentially bypass the LLM service load.

9.3 Latency Effects: - **Cache lookup latency:** Composed of embedding time (~10ms with MiniLM on CPU, could be 2–3ms on GPU or 100ms if using external API) + vector search (~ < 10ms for k=5 neighbors in a few thousand entries using HNSW) + DB fetch (~1-5ms for SQLite on disk, likely cached in memory). Total ~20–50ms worst-case to decide a cache hit. This is an order of magnitude faster than an LLM call (which might be 1–3 seconds). So even on a cache miss, the overhead of checking is tiny relative to proceeding with the pipeline. Users likely won't notice a difference on misses. - **Cache hit latency:** Almost the entire pipeline (except embedding & DB fetch) is skipped. So user gets response in maybe ~50ms plus negligible formatting. This is a huge improvement from maybe 2–5 seconds normally. It's akin to going from “think and answer” to “instant answer”. That can improve user experience tremendously, especially for a UI where users expect snappy responses for known questions. - We will ensure the embedding step is as optimized as possible (maybe do it asynchronously or batch if multiple queries come in simultaneously). - If using OpenAI embeddings for cache lookup, that 100ms network call might negate some latency benefit, but we would rarely do that (only in fallback scenarios). Typically, we keep it local.

9.4 Storage and Memory Costs: - As calculated, storing 10k entries ~ tens of MB. The vector index in memory also tens of MB. These are trivial on any modern server. - If we push to 100k entries, maybe a couple hundred MB in DB and memory. Still fine on a single node with a few GB of RAM free. - If we ever needed millions of entries (which would correspond to a truly huge usage over time), we’d consider a dedicated vector DB and sharding, which is a bigger change. But likely, given TTL and eviction, we might hover around tens of thousands of active entries at any time (because outdated ones drop off). - The cost of running an embedding model: If we use CPU, it uses maybe one CPU core for ~10ms per query. At 1000 queries, that’s 10 seconds of CPU time, which is fine distributed over time. If queries burst, the model might become a bottleneck; we could then enable GPU acceleration or parallelize with multiple instances. On a GPU, 1000 embeddings might take 1–2 seconds total. So easily manageable. - The cost of hosting this infrastructure: basically negligible (the SQLite DB and a small model). We might need to allocate some memory and possibly an extra container/pod for the embedding model if using Python. Compared to the cost of LLM calls, this is minimal.

In conclusion, the semantic cache is expected to **reduce average latency per query** (especially the common queries) and **cut token usage by roughly 30-50%**. The exact numbers will be validated in A/B tests (Section 12). If results align with our expectations, this means a significant budget relief and a better user experience due to faster answers and fewer rate limit issues.

10. Configuration & User Controls

To make this caching layer flexible for different projects and user preferences, we will introduce configuration options and UI controls (especially in the Template Builder UI, which is planned for orchestrating these settings).

10.1 Template-Builder UI Settings

We envision a JSON config block for semantic cache settings that can be edited in the Template Builder interface, for example:

```
"semanticCache": {
  "enabled": true,
  "similarityThresholds": {
    "L1": 0.90,
    "L2": 0.85,
    "L3": 0.80
  },
  "ttlPolicies": {
    "default": 604800,        // 7 days in seconds
    "stable": 2592000,        // 30 days
    "dynamic": 604800,        // 7 days for dynamic content (maybe same as
default here)
    "realTime": 3600          // 1 hour for real-time queries
  },
  "maxCacheSize": 10000,      // entries
  "evictionPolicy": "LRU",    // or "LFU" or "Custom"
  "embeddingModel": "local-miniLM",  // could be "openai-ada", "cohere-v3", etc.
  "vectorStore": "faiss",     // options: "hnswlib", "qdrant", "milvus", etc.
  "isolation": "shared"       // or "perUser", "perOrg" for how to namespace
cache
}
```

These settings allow: - Toggling caching on/off easily (if disabled, all cache calls would bypass to always miss and not store). - Adjusting similarity thresholds if a user wants more aggressive caching (lower threshold for more hits) or more caution (raise threshold). - Setting TTLs per category. The UI might allow tagging certain prompt templates or tools with a category, which we map to TTL. - `maxCacheSize` and `evictionPolicy` can be tuned depending on available storage. For instance, in a small environment maybe only 1000 entries, vs an enterprise might allow 100k (and prefer LFU if patterns are complex). - Choosing embedding model and vector store backend if advanced users want to, though we will provide sensible defaults. (This might be advanced usage, possibly not exposed in simple UI but in config file.) - `isolation` mode deciding how caches are separated between users/projects (discussed in Section 11.2 and Appendix D).

These configurations make the cache a modular component of the framework, which users can configure per their needs without changing code.

10.2 Runtime Controls

We should also provide runtime controls for developers or even advanced end-users (if appropriate): - **Cache Bypass Query Param:** e.g., if the API or UI accepts a query parameter like `?noCache=true`, then for that request, we skip checking and/or skip storing. This is useful for debugging (force the system to get

a fresh answer even if cache might have one, to compare). - **Force Refresh:** Similar to bypass, but perhaps a way to update cache: e.g., `?refreshCache=true` could force a fresh call *and* update the cache entry even if one existed. This might be used if someone suspects an answer is outdated and wants to refresh it proactively. - **Manual Flush endpoints:** An admin endpoint could allow issuing invalidation, like `POST /cache/invalidate` with a pattern or `DELETE /cache` to wipe completely. This is more of an internal tool but handy for maintenance (say we deployed a new version and want a clean slate, or had a data change not auto-detected). - **Cache Warmup:** We might allow pre-populating the cache with known common queries at startup or periodically. For example, we can maintain a list of top 50 asked questions (from logs) and after deployment, call the LLM for each to cache them. This ensures high hit rate from the get-go for those. The config could have a list of queries to pre-cache, or we could allow uploading a CSV of Q&A pairs to seed the cache. The user interface might have a section "Pre-cache these queries". - **Metrics Viewing:** In the template UI or admin UI, display current cache stats (hit rate, tokens saved). This gives users feedback on how effective the cache is for them and could encourage adjusting settings if needed.

Integration with profiles/projects: If the LLM Commander is used for different projects, each project might have its own cache config. The system should isolate or separate caches if needed (which relates to privacy but can also be a feature – e.g., one project's Q&As shouldn't spill into another's context if they're unrelated).

Essentially, while the cache mostly works behind the scenes, these controls ensure transparency and the ability to override or inspect its behavior, which is important for trust and debugging.

11. Privacy & Security

Implementing caching must not compromise user data privacy or system security. We address multiple angles:

11.1 Sensitive Data Handling

We will **never cache** queries or responses containing sensitive personal data or secrets. The code snippet we plan to include:

```
function containsSensitive(text: string): boolean {
  const patterns = [
    /api[_-]?key/i,
    /secret/i,
    /password/i,
    /\b\d{16}\b/,              // credit card
    /\b\d{3}-\d{2}-\d{4}\b/    // SSN format
    // ... other patterns as needed
  ];
  return patterns.some(pat => pat.test(text));
}

// In setResponse and similar functions:
if (containsSensitive(query) || containsSensitive(response)) {
```

```
    console.warn("Not caching sensitive query/response");
    return;
}
```

This is a basic safeguard: if the user's query includes something that looks like an API key or password, we skip caching it. Similarly, if the LLM's answer happens to output something that looks sensitive (could happen if it read a secret from context, etc.), we don't cache that either.

This prevents scenarios like caching someone's password reset link and another user triggering it via a semantically similar query. Although our system itself probably wouldn't handle actual passwords, this is a generic safety net.

We will refine the pattern list over time and possibly integrate a PII detection library for more robust checking if needed. But we lean toward caution — better not cache an innocuous thing misdetected as sensitive than to cache a sensitive thing.

Additionally, we differentiate **public vs private** contexts: - For queries that are inherently personal (e.g., "What's my account balance?"), the answer is user-specific and should not be cached globally if at all. Ideally, such queries wouldn't go through the generic cache, or if cached, it must be in a user-isolated cache (see next section). - On the flip side, queries about general knowledge or company info are fine to cache.

We will rely on developers marking contexts appropriately, or use the `isolation` setting to ensure, for example, an enterprise may set `perUser` caching so that even if personal data gets cached, it's only accessible to that same user's session.

11.2 Cache Isolation (Multi-Tenancy and User Privacy)

Our system could be used in several scenarios, as outlined: 1. **Public Knowledge Base (single tenant, public)**: e.g., documentation site where all queries are about the same content and not user-specific. Here a single shared cache is ideal – everyone benefits from each others' queries being cached. 2. **SaaS with multiple organizational tenants**: e.g., an internal Q&A system used by multiple companies. Each company's data is separate, and caching should not leak info across companies. We'd need either separate caches or at least namespace partitioning by org. 3. **Personal Assistant with private data**: e.g., an AI that a user connects to their email or files. Cache must be per user, because queries might involve personal info. We might even decide to disable caching for highly sensitive domains like personal emails, unless user opts in. 4. **Hybrid (global + private)**: Some content is global (public docs), some is user-specific. In this model, we might have a layered cache: check a global cache for common answers, but also have a user cache for things specific to that user's data or queries.

Design for Isolation: We can incorporate a `namespace` or `owner` field in cache keys. For instance, add `user_id` or `org_id` column in the tables. The CacheManager would then ensure to only retrieve entries matching the current user's namespace (or global ones). For example:

```
SELECT ... FROM response_cache
WHERE (user_id = ? OR user_id IS NULL) ...;
```

Where `user_id IS NULL` might denote a global entry. This way, a public answer can be reused by anyone, but a private answer (with user_id set) won't be visible to others.

Alternatively, maintain separate caches entirely (e.g., separate SQLite files or separate Redis keys) per tenant. That's simpler logically but maybe heavier if many tenants.

For now, our plan: - Include `user_id` (or `session_id` or similar) in cache if needed. By default, we might leave it null (meaning shareable). If the system integrator marks certain queries as containing private info, we set user_id for those entries. - The config `isolation: "shared" | "perUser" | "perOrg"` could control automatically how caching behaves. For "perUser", we always tag every entry with the user's ID, so no cross-user reuse. "perOrg" similar at org level. "shared" means no tag (everyone shares). - Another approach is hybrid: have logic that if the question references personal data, use personal cache, else global. Determining that automatically is non-trivial (requires understanding query content). Possibly certain tools or data sources imply personal context. For example, if the query triggers a search in "user's emails", that's clearly personal, so use per-user cache for that query. If the query is about "company policies" from a shared wiki, that's global content, use global cache. We can implement such rules if needed.

Access Control: We must also ensure that even if an entry exists in the DB, only authorized retrievals can happen. By partitioning queries at CacheManager level, we ensure, for instance, that a query from user A will not consider user B's entries in similarity search. The DB query or vector search needs to filter by the appropriate namespace. In a vector DB like Milvus or Qdrant, we might use metadata filtering (most allow storing an ID and filtering search by it).

Compliance (GDPR etc.): If a user requests deletion of their data, we need to also purge any cached content derived from it. With per-user caches, we could simply drop that user's cache when they leave. With global, it's trickier – but if global only contains non-PII by design, it should be fine. We will keep logs of cached content and provide means to clear specific content if needed (e.g., search for the user's name in cache and remove entries).

Encryption: If extremely sensitive, we could encrypt cache entries at rest with a key, but since our use-case is typically internal or controlled, and SQLite can be file-encrypted if needed, we won't complicate this now.

In summary, **for initial rollout** we likely assume a single-tenant or trusted multi-user environment (like one company's internal system) and use a shared cache for maximum benefit. But we design the system to support stricter isolation. For enterprise SaaS or personal AI scenarios, toggling those isolation settings will enforce privacy boundaries.

11.3 Security of the Cache Data

Aside from privacy, we consider: - The cache DB is on disk; ensure it has proper file permissions (only accessible by the service). - If using a networked DB or vector store, secure connections (e.g., use Redis/Milvus with auth enabled). - Sanitize all inputs that go into SQL queries (we mostly use parameterized queries as shown, so SQL injection risk is low. But if we ever allow an admin to run an invalidate with raw pattern, we must parameterize that). - The cached content might contain what the LLM said. We should be mindful if that content could be malicious (e.g., if someone somehow got the LLM to output some script). Since it's text, and we output it raw as answer, not much injection risk in the context of a user interface (the UI should escape if needed). But storing doesn't introduce new risk versus just outputting it at the time. -

Denial of Service scenarios: If someone tries to poison the cache by asking thousands of unique queries to fill it up, our size limit and eviction will handle it, but they could cause churn. Rate limiting at query level is a separate concern but relevant. The cache should not allow uncontrolled growth beyond configured size. - False data injection: If an attacker could somehow craft a query that yields a specific cached answer that later gets served to others incorrectly (like a kind of cache poisoning), that's theoretically possible if thresholds are not well-tuned. However, since queries are user-provided and answers come from LLM, an attacker can't directly insert an arbitrary answer except by tricking the LLM to output it under a similar query. Our similarity check would have to erroneously match a victim's query to that poisoned one. The best defense is high similarity requirements and maybe making sure different users (if untrusted relative to each other) have separate caches or at least that the content would be obviously out-of-context. We consider this a low risk but will keep an eye on it.

12. Testing Strategy

Given the complexity of the caching system, a thorough testing plan is needed:

12.1 Unit Tests

We will write unit tests for each component: - **EmbeddingService tests:** ensure that for known sentences, the embeddings have expected properties (e.g., cosine similarity of a sentence with itself is ~1, similarity of very different sentences is low). We can pre-compute a couple of embedding pairs using a reference (like use the same model offline to get ground truth). - **Similarity calculation:** Test our cosineSimilarity function with known vectors (including edge cases like zero vectors). - **CacheManager logic:** We can stub the DB and embedding service to test: - When a query exactly matches an existing entry (embedding equal), it returns the entry. - When no entries, returns null. - When an entry is present but similarity below threshold, returns null. - That it updates stats and last_accessed correctly on hit. - That `setResponse` inserts the entry properly and respects TTL and sourceHash fields. - **Eviction function:** Simulate a set of entries with varying access_count and last_accessed, feed to `runEviction` and verify that the intended ones are removed. We might not test the SQL directly, but maybe abstract the selection logic to a separate function for easier testing. - **TTL expiration logic:** Simulate an entry with a short TTL, advance time (or override current time in queries), run the cleanup query and see that it would remove it. Possibly test that `invalidateByPattern` deletes matching entries and not others. - **Isolation logic (if implemented in code):** For example, test that queries with different user_id do not get each other's cache.

We should also test some edge conditions: - Very long query strings (ensure embedding service can handle or we truncate as needed). - Non-ASCII characters in queries (embedding model should handle unicode, but we test to be sure, especially if using a specific model). - If embedding service fails (like API returns error or local model not loaded), ensure cache system falls back gracefully (maybe skip caching in that case). - Test that no caching occurs when disabled in config (simulate config off, ensure getResponse always null and setResponse no-ops).

12.2 Integration Tests

Set up an environment where we run a mini version of the system: - Use a small fake knowledge base (e.g., a couple of text docs in the RAG DB). - Issue queries through the full pipeline with caching enabled, and observe behavior.

Scenarios: - Query A -> no cache initially, goes through, caches result. - Query A again -> should hit cache, return same answer (we check time taken or logs to confirm no LLM call second time). - Query A paraphrased -> ensure it hits if above threshold (we might artificially adjust threshold or use a known similar phrase). - Query B (different topic) -> miss and store separately. - Then ask Query A one more time after some time -> still hits. - Update the underlying doc relevant to Query A -> update source_hash, then ask Query A again -> should *not* hit cache because we invalidated it. It should recompute and maybe give a slightly different answer reflecting update. - Test intermediate caches: We might craft two queries that share retrieval. E.g., Q1: "What is X?" and Q2: "Tell me more about X". They both retrieve doc X, but one asks differently. We want to see Q2 can reuse chunk retrieval from Q1's cache (L3), even if final answers differ. - Test multi-user: If possible, simulate two "users" by injecting a user_id in queries, ensure that one user's query doesn't pull cache from another if isolation set. This might be done by calling a lower-level function with user context.

Additionally: - Simulate the sensitive data scenario: Query includes "password" -> ensure it doesn't get cached (maybe by checking cache count remains same). - Force eviction: Fill the cache with dummy entries up to >max, then trigger or wait for eviction, confirm count goes down to max and that the right ones were removed (perhaps log the IDs and we know which should have stayed vs removed). - Performance measurement: not typically in integration tests, but we might measure that doing a cache hit is indeed much faster than a miss (but that might be more for benchmarking).

12.3 Performance & Load Testing

We should simulate high query volumes to ensure the cache doesn't become a bottleneck: - For example, use a script to issue 1000 queries, some repeated, some unique, with caching on, and measure throughput and latency. - See that the system handles the DB writes/reads quickly. SQLite can do many reads concurrently fine, writes are locked but our writes are small and infrequent (only on misses). Still, at high QPS, many misses might serialize on DB inserts. If that's an issue, we might consider an async queue for inserts or a different storage (like Redis). - Test memory usage: populate a lot of entries and check that memory (especially for the vector index) doesn't balloon beyond expectation. - We should also test startup and recovery: If the system restarts, does the cache rebuild properly? On startup, we might not load all into memory (we may rely on DB and gradually build index). But maybe we pre-load the most recent N entries to vector index. We should test that path – e.g., simulate restarting by creating a DB with entries, then initialize CacheManager and attempt to answer a query that should be a hit. If we didn't load index, we might not find it. Possibly we'll have CacheManager on init load all or some entries into the ANN index. For a moderate number of entries, loading all is fine. We test that to ensure persistence works as intended.

12.4 A/B Testing Plan

To truly measure benefit, we'll do an A/B test: - **Group A (Control):** caching disabled. The system runs as before. - **Group B (Experiment):** caching enabled.

We can route a percentage of traffic to each (if that's feasible in our deployment). Or if not user-facing, we could simulate with recorded queries.

Metrics to compare: - Average tokens per query in group (should drop in B). - Average latency per query. - Hit rate (for group B). - No negative impact on correctness (manually evaluate some sample answers from both to ensure group B isn't serving wrong answers).

A short initial A/B in a staging environment with known queries can help tune thresholds. Then in production, maybe gradually ramp up caching: e.g., enable for 10% users, observe, then 50%, then 100%.

We'll look at any user feedback – if caching causes any odd behavior (like stale answers reported), we will investigate and adjust thresholds or TTL.

13. Migration & Rollout

Adopting semantic caching can be done in phases to minimize risk:

Phase 1: Development and Off-line Testing – Implement the caching code, run it in a test environment. Validate with known queries (like simulate logs from past queries to build a cache and test hits). No production impact.

Phase 2: Shadow Mode (Passive Monitoring) – Deploy the cache alongside the system but not actually serving from it. Instead, on each query, still call LLM as usual, but also in parallel check "if caching were on, would it have hit?" and log that. This gives a sense of potential hit rate and any potential false matches without affecting output. We can log any cases where cache *would have* answered differently than the actual system did (which indicates either a false positive or maybe an improvement). This phase de-risks enabling it.

Phase 3: Soft Launch (Partial Traffic) – Enable caching for a small percentage of live queries. This could be by user segments or times. Monitor metrics (especially correctness – any user complaints? – and performance improvements). If issues are found, adjust config.

Phase 4: Full Deployment – Once confident, enable for all queries by default. Continue monitoring.

Backout Plan: Because caching is an optimization, if anything goes wrong (like we suspect it's giving bad answers), we can instantly disable it via config (`enabled: false`). The system will then behave like before (though possibly with a slight overhead if we still call getResponse but it always returns null). This feature flag means risk of serious impact is low – worst case scenario we turn it off.

Data Migration: Initially, there's no existing cache, so no migration needed. If in future we modify schema (say add a column or change embedding model), we'll have to handle migrating or invalidating old entries. Possibly we might just flush cache on such changes.

Backward Compatibility: The existing system should not need to be changed drastically. The cache intercept is additive. If the cache is off, the system works exactly as it used to. So we are not removing anything the system could do; only adding a faster path.

Documentation & Training: We'll document how to configure and use the cache for the dev team and possibly end-users (particularly if exposing any UI settings). Make sure everyone knows how to flush it if needed and what to expect.

Rolling out carefully ensures that we reap the benefits without disrupting user trust. We should particularly inform stakeholders that some answers might now be coming from cache to set expectations that answers

will be consistent for repeated queries (which is usually a good thing – users might be happy that they always get the same answer for the same question instead of slight rewordings by the LLM each time).

14. Monitoring & Observability

To justify the cache and keep it functioning optimally, we will set up monitoring:

14.1 Metrics to Track: - **Cache Hit Rate (overall):** number of queries answered from cache / total queries. We'll track this as a primary KPI. Also track hit rate by layer: L1 direct answers, L2 partial, L3 partial. - **Tokens Saved:** we can calculate total tokens used by LLMs vs. estimated tokens if no cache. Our `cache_stats` table logs daily tokens_saved. We will surface that in a dashboard or log. This directly ties to cost saved. - **Latency Improvement:** We can measure end-to-end response time for cached vs non-cached queries. For instance, log the duration for each query and whether it was a hit or miss. Then we can graph average latency for hits vs misses. Expect hits to be much lower. - **Cache Size:** number of entries and maybe memory usage of cache. If we see it growing near limit often (evictions happening frequently), we might consider increasing max size or adjusting content. - **Eviction frequency:** how many evictions per hour/day. If very high, maybe our cache is too small or threshold too low causing thrash (many things cached then evicted without being reused). - **False hit occurrences (if any):** Hard to automatically detect, but perhaps measure similarity of queries served from cache to ensure they were indeed very close. If we see any borderline cases (like it served at 0.85 similarity if threshold is 0.90 due to fuzzy bump), flag those for review. - **Error rates:** in case any part fails (embedding service errors, DB errors), count those. We want to ensure the cache doesn't introduce instability.

We will integrate these metrics into our logging/monitoring system (likely CloudWatch or Prometheus if used). The `cache_stats` table can be aggregated and exported periodically.

14.2 Alerts: Set up alerts for: - **Hit rate drop:** If cache hit rate falls below, say, 10% after being historically ~30%, something might be wrong (maybe cache not working or was accidentally cleared or threshold too high). Could indicate need for investigation. - **Cache size high:** If the number of entries approaches the limit often or DB file grows unexpectedly. Could signal misconfiguration (like TTL not working) or usage patterns change (lots of unique queries). - **Lookup latency spike:** If retrieving from cache suddenly takes long (maybe vector DB became slow or embedding API slowness). For example, if average cache lookup > 100ms, alert. It might degrade user experience and needs fixing. - **Eviction thrashing:** e.g., >100 evictions per hour consistently might indicate our cache is too small to hold working set. That might warrant increasing size or adjusting strategy. - **Staleness incidents:** If we implement drift detection, if it invalidates a lot of entries often, maybe our TTLs are too long.

We will also track usage of each embedding model and vector store (if we support dynamic selection) to gauge cost. For example, count how many calls to OpenAI embed are made (if any in fallback), to keep an eye on those costs.

Logging: On each query, we can log an event with: - Query ID, whether cache hit or miss (and at which layer), similarity score if hit, time saved, tokens saved etc. - This raw log can be analyzed for patterns.

Visualization: Create a dashboard that shows: - Daily hit rate percentage (with breakdown by L1, L2, L3). - Cumulative tokens saved (maybe a counter). - Average answer latency with/without cache on a timeline.

This observability will not only prove the value (to stakeholders, seeing a graph “$ saved this week”) but also help fine-tune. For instance, if hit rate is lower than expected, we can investigate: maybe threshold is too high or embedding not capturing synonyms well, etc. Or if positive hit accuracy is not near 95%, maybe threshold too low.

All these ensure we catch any issues early and continuously validate that caching is delivering benefit without sacrificing answer quality.

15. Future Enhancements

Looking beyond the initial implementation, there are several improvements and extensions possible:

15.1 Distributed Caching: In a multi-server deployment (like multiple instances of our Node backend behind a load balancer), each would have its own in-memory vector index and maybe local SQLite. This could lead to cache fragmentation (each node caches different queries). We might move to a distributed cache: - Use a central vector database (like Qdrant or Milvus) accessible by all instances, so they share embeddings. - Or use Redis as a distributed cache store and some way to synchronize vector indices (maybe each does a vector search on the fly by querying the DB for candidates – slower but ensures consistency). - GPTCache has guidance on horizontal scaling using a shared Redis for coordination [46]. - We might implement a simple approach: one node is “cache master” that others query via an internal API. But using a proper distributed store is more robust. - This becomes important if we scale out. Initially, on one node, it's fine. But even on one node, if we use threads or async, our design works as SQLite can handle concurrency in read (with WAL mode). - We'll evaluate when needed. Options like Redis with its new vector search or pgvector on a central Postgres might be easiest to adopt if performance suffices.

15.2 Proactive Cache Warming: As mentioned, we can seed the cache with frequent queries rather than waiting for them. We could mine logs or ask support teams for top asked questions and pre-cache them at deploy time. This ensures even the first user to ask gets a cache-speed response. It’s basically precomputing known answers. - Another angle is to schedule jobs to periodically refresh certain queries (like maybe top 10 queries each day, ensuring they stay updated and in cache). - For example, every midnight, run common queries related to any updated content (if we know a doc changed, pre-ask likely questions about it to prime cache).

15.3 Query Normalization/Rewriting: We could improve cache hit ratio by normalizing queries more aggressively: - Remove stopwords or irrelevant context from the text used for caching. E.g., user might say "please" or "could you". Stripping such polite fillers could make two queries more directly match. - Spell-check or correct obvious typos in queries before embedding, so "resset password" and "reset password" are treated same. The embedding model likely is somewhat robust to this, but a quick normalization could help. - Synonym replacement: Hard to do generally, but maybe ensure consistent terms (like convert "misplaced" to "lost" if we know they mean same in context). This might be better left to embeddings though. - Essentially, define a canonical form for queries to reduce variance. But careful not to remove meaning.

15.4 Federated Caching Across Projects: If we end up with multiple projects or instances, we might allow sharing cache entries for common knowledge. For example, if one project is Q&A on Linux commands and another on programming, they might both get question "How to list files?". A global cache of generic knowledge could answer it for both. Meanwhile, project-specific caches handle their unique data. - This could be done via multi-tier: check global cache, then project cache, then user cache. - It requires

delineating what queries qualify as global (maybe those not referencing any project-specific terms). - This is complex and perhaps not needed unless we have many instances of similar nature.

15.5 Integration with Model Improvements: If in future, the LLM or embedding model improves (e.g., we upgrade from GPT-4 to GPT-5, or embed model to a new version), the cache might hold slightly outdated style or content. When switching major model versions, it might be wise to flush cache because the new model might answer differently (maybe better). Or we can version the cache by model. For now, mention that if model is changed, one should invalidate or separate cache entries by a model tag (we have agent/model info stored).

15.6 Additional Cache Layers: - We might consider caching at the prompt prefix level for multi-turn (Anthropic's style of prompt caching). This is more relevant if we had long conversation contexts; our case is more single-turn with retrieved docs. Not a priority now. - Another idea: cache **embedding results** themselves. If computing an embedding is expensive (like using OpenAI embed API), we could cache the embedding of each query text (like a separate small LRU for embeddings). This is only useful if exact same query comes again – which our L1 already covers. Or if partial queries overlap (less useful). Probably overkill.

15.7 Multi-modal caching: If we integrate image or audio (as GPTCache supports experimental), caching those responses too would be interesting. E.g., if user asks for an image generation with same prompt, cache the image to not call API again. That requires storing binaries (which we can in SQLite or a file store). This might come if our system adds such features.

In summary, our design is forward-compatible with a number of enhancements. We have chosen an approach that's modular, so swapping out the embedding model or vector backend, or adjusting isolation, are configuration changes rather than architectural changes. This positions us well to adapt the semantic cache to future needs or to scale it up as usage grows.

16. Alternative Approaches Considered

During design, we evaluated other strategies for reducing redundant LLM calls and some simpler caching methods:

- **Exact String Matching Cache Only:** The simplest cache would be to key by the exact prompt string (including retrieved context) and return the recorded response. We decided this is insufficient for our use-case because it would miss too many opportunities (users rarely type the exact same words) [1]. It's also fragile – any slight difference or added punctuation in the query fails to match. While trivial to implement (could even use a Python `functools.lru_cache` on the LLM call function), it would yield a very low hit rate (likely <5% based on expected rephrasing frequency). Given our goal of 30-50% hits, exact match was ruled out as the main solution. We do note that our system could incorporate exact matching as an initial quick check (before embedding) as an optimization – e.g., if user literally repeats their last question, we can catch that instantly with a hash lookup. But semantic covers that scenario anyway by yielding a perfect similarity score.

- **OpenAI/Anthropic Native Prompt Caching:** Both OpenAI and Anthropic have introduced caching mechanisms on their side [47] [48]. For instance, Anthropic's Claude allows sending a cache identifier

such that if you send the same prompt prefix again, you don't pay full price again, with claims of up to 90% cost reduction for reused context [47]. OpenAI has a similar concept of reusing tokens between calls (especially if you provide a conversation ID). We considered relying on these features (which require no extra infrastructure on our side). However, they have limitations:

- They only cache identical prompt segments, not semantic similarity. If a query is rephrased, the API won't consider it the same cache key (OpenAI's is more low-level token caching).
- They often require the prompt to be sent in multiple parts (like system message cached separately from user question). Our pipeline doesn't easily fit that unless we refactor heavily.
- The cache is provider-specific and opaque. If we switch models or providers, we lose the benefit. And it might not even reduce our bill – it reduces their computation, but OpenAI's documentation on prompt caching indicates it automatically works without code changes, which suggests they might just do it internally for efficiency, but it's unclear if they pass the token savings to the user or just use it to reduce latency. (Anthropic's does say it reduces cost for user by up to 90% on repeated parts [47].)
- We wanted a solution that works uniformly across OpenAI, AnthropIc, local LLMs, etc., and that we can control and inspect.

For these reasons, we chose to implement our own caching. For completeness, in contexts where one is locked into a specific provider and has these features, one should enable them. They are complementary – e.g., we could use semantic cache to decide if a query is similar, and if we still decide to call the LLM (maybe to confirm), use the provider's caching to avoid full token cost.

- **Rely on LangChain or LlamaIndex memory**: LangChain provides a caching utility where it can store previous LLM calls results keyed on prompts. It's primarily exact-match, though they have an integration with GPTCache for semantic. Essentially, LangChain would be a wrapper around what we are doing. We decided not to rely on LangChain's caching because our architecture is somewhat custom and not all queries go through a single LangChain LLM interface. Also, we want fine-grained control (multi-layer caching which LangChain doesn't natively handle). LlamaIndex similarly doesn't have an out-of-the-box semantic cache for responses (it's more about document indices). They both would still need an embedding-based approach under the hood, which leads back to GPTCache or custom.

- **No Caching (Status Quo):** We considered the do-nothing as a baseline. This would mean every query always goes to LLM, ensuring the latest response each time. The downside we know: wasted computation on repeats, slower responses, higher cost. The upside is simplicity and always fresh output. Given the evidence that many queries *are* repetitive and caching can significantly cut costs [3] [4], doing nothing would leave significant optimization on the table. Particularly for a "budget-conscious" framework, ignoring caching would be against our goals.

In conclusion, implementing a semantic cache ourselves provides the best trade-off: - It significantly improves cost and speed. - It's within our development capability (leveraging open-source GPTCache ideas and existing libs). - It introduces minimal risk with proper safeguards. - And it's flexible to work in our multi-agent context and with any model.

The alternatives were either too weak (exact match), too provider-dependent or limited (API caching), or just maintaining the expensive status quo. Therefore, we proceeded with the semantic caching design detailed above as it offers the most value to the project.

17. References & Citations

- GPTCache GitHub repository – documentation on architecture and usage [1] [18].
- GPTCache blog ("Yet another cache for ChatGPT") – insights into design decisions, e.g., why semantic vs keyword, and experimental results on hit ratios [49] [2].
- ACL Workshop Paper on GPTCache (Bang, 2023) – formalizes the concept of semantic caching for LLMs [50].
- Sentence-Transformers documentation and models – details on MiniLM, MPNet models, dimensions and performance trade-offs [51] [21].
- OpenAI Embeddings API documentation – for info on embedding costs and dimensions [52].
- Cohere Embeddings documentation – model sizes and capabilities (1024-d, multilingual) [53] [37].
- Vector database comparisons (Medium and Reddit discussions on Milvus vs Qdrant vs Weaviate vs pgvector) – used for backend selection trade-offs [54] [55].
- Microsoft Azure Tech Community blog on Semantic Caching – confirming benefits (20-30% queries cached) and describing similar architecture [3] [56].
- Research on caching algorithms (LRU, LFU, ARC) – general caching knowledge applied in eviction policy.
- Anthropic and OpenAI prompt caching announcements – to contrast our approach with provider-level caching [47] [48].

Each of these sources contributed to the rationale and validation of our design choices, ensuring our SDD is grounded in state-of-the-art practices and realistic data.

---

[1] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [20] [26] [39] [40] [41] [46] GitHub - zilliztech/GPTCache: Semantic cache for LLMs. Fully integrated with LangChain and llama_index.
https://github.com/zilliztech/GPTCache

[2] [7] [19] [27] [30] [31] [42] [43] [49] Yet another cache, but for ChatGPT - Zilliz blog
https://zilliz.com/blog/Yet-another-cache-but-for-ChatGPT

[3] [56] Optimize Azure OpenAI Applications with Semantic Caching
https://techcommunity.microsoft.com/blog/azurearchitectureblog/optimize-azure-openai-applications-with-semantic-caching/4106867

[4] [5] [6] [21] [22] [23] [24] [25] [44] [45] [50] [52] GPT Semantic Cache: Reducing LLM Costs and Latency via Semantic Embedding Caching
https://arxiv.org/html/2411.05276v1

[28] Intelligent Memory API | Sub-5ms Response Times (by n3wth) - r3
https://r3.newth.ai/docs/ai-intelligence

[29] [38] [51] semantic similarity analysis using transformer-based sentence ...
https://www.researchgate.net/publication/394616542_SEMANTIC_SIMILARITY_ANALYSIS_USING_TRANSFORMER-BASED_SENTENCE_EMBEDDINGS

32 BAAI/bge-large-en-v1.5 - Hugging Face
https://huggingface.co/BAAI/bge-large-en-v1.5

33 bge-large-en-v1.5 - Xinference
https://inference.readthedocs.io/en/latest/models/builtin/embedding/bge-large-en-v1.5.html

34 Fine-Tuning Embedding Models: Achieving More with Less
https://pub.towardsai.net/fine-tuning-embedding-models-achieving-more-with-less-d89082265ba8

35 Alibaba-NLP/gte-Qwen2-7B-instruct - Hugging Face
https://huggingface.co/Alibaba-NLP/gte-Qwen2-7B-instruct

36 rjmalagon/gte-qwen2-7b-instruct:f16 - Ollama
https://ollama.com/rjmalagon/gte-qwen2-7b-instruct:f16

37 cohere-embed-english-v3_0 model | Clarifai - The World's AI
https://clarifai.com/cohere/embed/models/cohere-embed-english-v3_0

47 48 Prompt caching with Claude - Discussions - Cursor - Community Forum
https://forum.cursor.com/t/prompt-caching-with-claude/7551

53 Cohere Embed Model 3 - Multilingual - AWS Marketplace
https://aws.amazon.com/marketplace/pp/prodview-b4mpgdxvpa3v6

54 55 Choosing a vector db for 100 million pages of text. Leaning towards Milvus, Qdrant or Weaviate. Am I missing anything, what would you choose? : r/vectordatabase
https://www.reddit.com/r/vectordatabase/comments/1dcvyrm/choosing_a_vector_db_for_100_million_pages_of/