ChatGPT

Embedding Strategy Upgrade for LLMC

Purpose and Scope

This design proposes an upgrade of LLMC's semantic search embedding strategy to improve retrieval quality (Recall@5) by at least 10% over the current system. We will replace the existing MiniLM embedding model with the **intfloat/e5-base-v2** model (768-dimensional) and adjust the retrieval pipeline accordingly. The scope includes changes to embedding generation, vector index storage, query-time retrieval logic, and evaluation/monitoring. We will implement a new **E5-based vector index** with L2-normalized embeddings and prefix tagging (using `"query: "` and `"passage: "` prefixes as required by E5 [1] [2]) to better align with the model's training. We also integrate an (existing) reranker and a dual-index mechanism behind feature flags for future use.

Non-goals: This upgrade focuses on the embeddings and retrieval layer. We will **not** modify the LLM answer-generation model or prompting strategy beyond what's needed for retrieval context. We are not addressing multi-lingual support, new data sources, or drastic changes to chunking logic. Performance optimizations will preserve current latency as much as possible, but training new models or switching to external vector databases (beyond those already in use) is out of scope. We also do not plan to enable the reranker or dual-index features by default in this rollout – these will remain optional toggles, as improving the base embedding recall is our primary goal.

Current Architecture (As-Is)

Embedding Model and Vector Store

LLMC currently uses a SentenceTransformer model (**all-MiniLM-L6-v2**) to embed text into a 384-dimensional vector space [3]. This model produces relatively compact embeddings and was chosen for speed and lightweight memory usage (~1.2 GB VRAM) [4]. The vectors (for each document "chunk") are stored in a local vector index, persisted in the repository's `.rag/index.db` file. This likely corresponds to a **SQLite-backed store** (potentially managed via the Chroma library or a custom implementation). In the existing setup, the embedding index is either an on-disk SQLite/Chroma database, which is loaded on startup, or an in-memory FAISS index built from that data. The index provides similarity search over stored vectors. Similarity is computed via **cosine similarity** (the MiniLM embeddings are normalized to unit length, making inner products equivalent to cosine [5]).

Each knowledge item is broken into one or more **chunks** before indexing. For example, if the content is source code or documentation, it is divided by logical sections (functions, classes, paragraphs) or by a token length limit (e.g. ~300-512 tokens per chunk with overlaps). Each chunk is stored with metadata including identifiers and text fields. Currently, a chunk's text serialization might include fields such as the repository path, symbol name, code/entity kind, summary, docstring, and a trimmed body of the code or text. For instance, a chunk is represented as:

```
<repo-path> • <symbol> • <kind> • <summary> • <docstring> • <body_trim>
```

(where "•" denotes a separator) up to roughly 512 tokens. This combined text is what gets embedded and stored. The inclusion of symbol names and comments in the chunk ensures that semantic search can match queries on function names or descriptions, not just the function body.

The **indexing pipeline** (likely in `scripts/` or `tools/rag/`) reads documents (code files, markdown docs, etc.), splits them into chunks, and uses the MiniLM model to embed each chunk. Chunks are then added to the vector store with metadata (e.g. source file, chunk ID). The resulting index (vectors + metadata) is persisted to `.rag/index.db` (if using Chroma, this file is a SQLite DB that Chroma uses to store collections and embeddings). Alternatively, a custom SQLite table may be used (with each row storing an embedding BLOB and metadata). In either case, the current dimension is 384 and the index is optimized for **exact** cosine similarity search given the manageable size. If the dataset is small-to-medium (on the order of thousands of chunks), the system may be performing a brute-force scan (direct dense similarity calculation) which is feasible, or using an in-memory Faiss `IndexFlatIP` for fast inner product search.

A **semantic cache** is present to avoid redundant computations. This includes an **embedding cache** (so the same document text isn't re-embedded multiple times) and possibly a **query result cache** to reuse recent search results for identical queries [6]. For example, if a document ingestion is re-run, the pipeline can skip embedding chunks whose text hasn't changed (using a hash or cache key). Likewise, if a user repeats a question, the system may reuse the last retrieved results instead of recomputing the vector search, provided the index hasn't changed. This cache is currently keyed by the text content (for embeddings) or query string (for retrieval results), and stored in memory. The cache improves throughput and Time To First Token (TTFT) for repetitive inputs.

During query time, the **Retrieval-Augmented Generation (RAG) pipeline** works as follows: 1. The user's query is embedded by the MiniLM model into a 384-dim vector. 2. The vector store is queried (via cosine similarity) to retrieve the top *k* most similar chunks (usually *k* is around 5–10 by default). 3. These top chunks are returned as context and fed into the prompt for the LLM to generate an answer. The chunks come with metadata so that the answer can cite sources. 4. There is currently a **reranker** component implemented but **disabled by default** (`RERANK_ENABLED=0`). If it were enabled, the pipeline would initially retrieve a larger set (e.g. top 50) and then use a cross-encoder model to re-score and select the best 8 results. By default, however, the system takes the vector search's top results as-is. The reranker config suggests two model options: `mini_lm` (likely the cross-encoder *ms-marco-MiniLM-L-6-v2*) and `bge_base` (perhaps a BAAI BGE-based reranker), with parameters like `RERANK_K_IN=50`, `RERANK_K_OUT=8` and a confidence threshold `RERANK_CONFIDENCE_TAU=0.32`. Currently, since reranking is off, these parameters are not actively used – the system just uses top ~8 chunks from the initial embedding search.

In summary, the as-is architecture has a **384-dim embedding model (MiniLM)**, a **persistent vector index** (`.rag/index.db`) possibly backed by Chroma/SQLite or loaded into Faiss at runtime, a straightforward query->embedding->similarity search flow, and caching layers to optimize performance. All retrieval happens locally with no external API calls, keeping latency low (MiniLM infers quickly ~68ms per query embedding [7]). However, the recall and semantic coverage of MiniLM can be improved. Internal evaluations have shown that many relevant code snippets or docs are missed in the top-5 results. This motivates moving to a more powerful model and enhanced indexing strategy.

Target Design

New Embedding Model: E5-base-v2 (768-dim)

We will adopt the **intfloat/e5-base-v2** model for embeddings. This model is a 12-layer Transformer producing 768-dimensional embeddings [8]. It has demonstrated significantly better retrieval performance on benchmarks (e.g. ~83.5% top-5 hit rate vs ~78.1% for MiniLM on a BEIR dataset [7]) – an improvement of ~5–10 percentage points in Recall@5, depending on the domain. In our internal experiments documented in *embedding model.md*, E5-base-v2 showed $\geq$*10%* relative improvement in Recall@5 over the MiniLM baseline. This is consistent with community evaluations where E5-base outperforms all-MiniLM-L6-v2 on semantic search tasks [7].

The E5 model requires a specific input formatting: **each input text must be prefixed with** `"query: "` **or** `"passage: "` depending on whether it's a search query or a content passage [1] [2]. This prefix was used during E5's training (which involved separate query and passage tasks) and is essential to achieve the expected performance [2]. Failing to add these prefixes would degrade accuracy. Therefore, as part of this upgrade, we will incorporate **span prefixing** in all embedding operations: - When embedding a **user query**, we will prepend `"query: "` to the raw query text before feeding it into the model. - When embedding a **document chunk** (during indexing or re-indexing), we will prepend `"passage: "` to the serialized chunk text (the combined `<repo-path> • <symbol> • ...` string).

For example, if a chunk's text serialization is:
`"llmc/repo/utils.py • function foo_bar • function • Does X • Docstring: ... • def foo_bar(...)"`
we will embed `"passage: llmc/repo/utils.py • function foo_bar • function • Does X • Docstring: ... • def foo_bar(...)"`. Similarly, a user query like *"How do I foo bar?"* becomes `"query: How do I foo bar?"` for embedding.

We will utilize Hugging Face's implementation of E5. The integration can be done via the SentenceTransformers library for convenience (which can handle automatic normalization), or via `transformers` directly. Using SentenceTransformers: `model = SentenceTransformer('intfloat/e5-base-v2')` allows us to simply call `model.encode(texts, normalize_embeddings=True)` [9] [10]. This will output L2-normalized embeddings in $\mathbb{R}$^768, which we require for cosine similarity comparison. If using the lower-level HF API, we must manually apply L2 normalization (as shown in the model card example [10]).

L2 Normalization: All E5 embeddings (both query and passage) will be L2-normalized (unit length) by design. E5's model card explicitly demonstrates normalizing the embeddings before computing similarity [10]. By storing normalized vectors, we ensure that the dot product of two vectors equals their cosine similarity. Our similarity metric will thus effectively be cosine distance (which is appropriate given E5's training objective and yields scores typically in [0,1] range). Using normalized vectors standardizes the scale and makes the similarity scores more comparable. (Note: The cosine similarities from E5 tend to distribute in a higher range, often 0.7–1.0 for relevant pairs [11]; this is expected due to the low temperature used in training, but does not affect rank ordering).

The new model is larger than MiniLM, so memory usage will increase (~2.0 GB for E5-base-v2 vs ~1.2 GB for MiniLM-L6 [4]). This is a **trade-off** we accept for better semantic performance. The model will run on CPU by default (unless a GPU is available and configured) and is still considered "high throughput" – roughly ~79 ms per query, only ~16% slower than MiniLM's ~68 ms in one benchmark [7]. This slight increase in per-query embedding time is within our latency budget. We will monitor resource usage, but current server instances should handle an extra ~0.8GB memory and a modest CPU load increase. If needed, we can allocate the model to a GPU to reduce latency, but initial rollout will focus on CPU inference to avoid impacting the LLM GPU memory.

Vector Index: 768-Dim Schema and Versioning

We will create a new vector index to accommodate 768-dimensional embeddings. The existing `.rag/index.db` (with 384-dim vectors) cannot be directly reused because of the dimensionality mismatch. Instead, we'll **build a fresh index (v2)** and run a full re-indexing of the content using E5. This new index will be stored separately to allow easy rollback and comparison. Concretely, we will introduce an **index version** identifier and name the new index accordingly (e.g., `.rag/index_v2.db` or similar). The system's configuration will be extended to handle multiple index versions: - A config variable (or constant) like `EMBEDDING_INDEX_NAME` or `INDEX_VERSION` will specify which index to use at runtime. For instance, `INDEX_VERSION=2` could point the system to use `index_v2` (E5 embeddings). - The new index will be created under a distinct name/collection, e.g. a Chroma collection named `"llmc_embeddings_v2"` or an SQLite table `embeddings_v2` in a new DB file. This **index separation** ensures we do not corrupt or mix data from the old index. The old index (`index_v1` with MiniLM vectors) will remain intact for backup and potential fallback.

The schema for the new index remains similar: we store each chunk's embedding vector and associated metadata (like chunk ID, source path, etc.). The only difference is the **vector length = 768** now, and we will explicitly record the model and normalization in the index metadata for clarity. For example, if using Chroma, we will create the collection with `dimension=768` and perhaps set `metadata={"model": "intfloat/e5-base-v2", "version": 2, "normalized": true}`. If using a raw SQLite/FAISS approach, we will document the model version in code (and potentially in a manifest file in `.rag/`). This way, anyone inspecting the index or re-running the embedding process knows exactly what model and parameters were used.

Because we are not altering how metadata is stored, the chunk identifiers and metadata schema remain the same. We expect the new index to contain the **same number of vectors** as the old one (assuming we index the same content corpus). To verify, we will compare the row counts: after reindexing, `count(new_index_vectors)` should equal `count(old_index_vectors)` (unless the chunking algorithm changed or failed on some files, which it shouldn't in this case).

Migration process: Building the new index will be done offline (as a one-time migration step, not during live serving). We will use a script or tool (see **Reindexing** below) to iterate over all source documents (or over all records in the old index) and do the following: 1. For each chunk of content, construct the text serialization (the same way the current pipeline does). 2. Prefix the text with `"passage: "` and embed it using E5. 3. Normalize the embedding (if the embedding function doesn't already do so). 4. Add the vector and metadata to the new index.

This can be done by reusing the existing ingestion pipeline (`tools/rag` utilities). We will likely augment the current indexing script to allow selecting a model. For example, if there is a script `scripts/build_index.py` (or similar in `scripts/`), we'll add an option or environment flag to specify the embedding model to use. The output will be directed to the new index name.

No direct data conversion of vectors is attempted (it's not feasible to convert 384-dim vectors to 768-dim). Instead, we do a fresh embedding of all content. This ensures the new index is consistent and fully populated. The process may take some time depending on corpus size, but it's a one-time cost and can be done in a background job or prior to deployment. We will perform this on a suitable machine (with access to the repository data). If the corpus is large, we can embed in batches and even utilize multi-processing or GPU acceleration to speed it up.

After building, we'll **validate the new index**: sample a few queries on the new index and manually verify that relevant results are returned (especially for queries where we know the old index had issues). We should see improved hits for those cases, confirming the E5 embeddings are semantically richer. We will not immediately deprecate the old index; instead, we keep it available for fallback or quick revert.

Index Versioning and Fallback: The system will treat the new index as an alternate source behind a feature flag. We will introduce configuration toggles such as: - `EMBEDDINGS_MODEL = "intfloat/e5-base-v2"` (or a generic name like `"E5_BASE_V2"`) to specify which model to load. - `INDEX_NAME = "index_v2"` or similar, to specify which index file/collection to use for retrieval. Currently, these might not exist (the old code likely had the model name hard-coded or only one index path). We will implement these to allow easy switching. Initially, default config will still point to the MiniLM model and index (ensuring no change in production until we're ready). To test the new pipeline, we will run with the env override pointing to E5 and the new index.

If something goes wrong with the new index during rollout, we can switch these config values back to the old ones and restart the service – effectively **backing out to the MiniLM index** quickly without code changes. (More on backout below.)

Retrieval Pipeline Updates

The query-time retrieval flow will undergo several updates to accommodate the new embedding and to maintain performance:

1. Query Embedding: When a user query comes in, the system will prepend `"query: "` and embed it using the E5 model (if the new pipeline is enabled). We will ensure any caching layer accounts for the model change: e.g., a query cache should include the model or index version in its key. We will likely **invalidate** or segregate the old query-result cache because cached results from the MiniLM index are not valid for E5 (the vector space and IDs differ). For instance, we might clear the cache on deployment of the new model, or namespace cache entries by `index_version`. Similarly, if we have an embedding cache for queries, those are tied to the model, so they should be cleared or versioned. After this, the embedding is computed. We will measure the embedding time and possibly log it (for observability) as `embed_time_ms`. Given E5 is slightly slower, we'll watch if this adds noticeable latency.

2. Vector Search: The embedded query (768-dim vector) is then passed to the vector store for nearest neighbor search. The underlying vector search mechanism will remain *cosine similarity*, but we will ensure the infrastructure supports efficient search at the new dimension: - If using **FAISS**: we will build a new FAISS index of type `IndexFlatIP` (inner product) or consider an approximate index (HNSW) if needed. Initially, given the vector count likely remains on the order of a few thousand or low tens of thousands, a brute-force `IndexFlatIP` is acceptable (FAISS can compute dot products very quickly in C++). If our corpus grows significantly, we'll switch to an ANN index (like FAISS IndexHNSW or IVF) to keep query latency low [12]. The design is to prefer HNSW or IVF for production scale vector search [12], but we will gate this by practical need. We can set a threshold (e.g., if corpus > 50k vectors or if query latency > 50ms, use ANN index). - If using **Chroma**: we will instantiate the new collection likely with default settings. Chroma (as of current versions) uses an in-memory index and can leverage Faiss under the hood for faster search. We'll configure it to use cosine metric. If Chroma supports HNSW, we may enable that if the dataset is large, otherwise default (which might be a simple indexing) is fine for moderate sizes. - If using a custom **SQLite** approach: If previously we had a SQL function for cosine, we'd similarly implement or use a procedure to compute distance for 768-dim. However, doing brute-force search via SQLite for thousands of vectors might be slow. It's more likely we rely on an in-memory approach (like reading all vectors into memory as a NumPy array). Given the new index, we will load it on startup similarly. Memory footprint for vectors roughly doubles, but still likely in the low hundreds of MBs (e.g., 10k vectors * 768 dims * 4 bytes ~ 30MB). This is manageable. We'll ensure the code that loads the index is updated to expect 768-dim and to build the right data structures (e.g., initialize a numpy array of shape [N,768] or a Faiss index with d=768).

We will preserve any **metadata filtering** or other logic around the search. (Currently, if the system had support for filters or search by subset, that remains unchanged in concept – though in code we might need to adapt any SQL queries if dimension is encoded in schema.)

3. Dual-Index Logic (behind flag): We introduce the concept of a dual-index retrieval path, although it will be **off by default** initially. When `DUAL_INDEX_ENABLED=1`, the system can query two indices/providers for results: - A primary (presumably the local E5 index, or whichever is set as primary). - A secondary provider specified by `DUAL_INDEX_PROVIDER` (options could be `"voyage"`, `"openai"`, etc.).

"Voyage" likely refers to our local vector store, and "openai" could refer to an external vector search (for example, a managed service or OpenAI API). The dual index feature is intended for scenarios like: - **A/B comparison**: to query both old and new indexes in parallel and compare results. - **Hybrid sources**: e.g., local index for proprietary data and an external index for public knowledge. - **Fallback**: query the secondary index only if the primary yields low-confidence results.

In this design, we will implement a basic framework for dual indexing: - If the flag is on, after obtaining the query embedding, we send it to both the primary and secondary indexes. - We then need to **merge the results** from two sets. We plan to use **Reciprocal Rank Fusion (RRF)** or a similar rank-merging strategy [13] to combine results without relying on directly comparable raw scores (since different indexes may have different score distributions). RRF will take each result list (ranked by their own similarity) and compute a combined score = Σ(1/(rank_i + k)) for each document across lists. This avoids needing to normalize scores across models. We then select the top K from the merged list. - Alternatively, a simpler approach (for initial implementation) is to take the top K from primary, and only if none of those have a similarity above a threshold, then take results from the secondary (fallback mode). For example, if `openai` is the secondary, we might call it only if the local index's top score < 0.3 (meaning the query might be out-of-domain for local data). - The specific merging policy will be determined by experimentation. Initially, since dual-index is off,

we don't have to finalize this; we will include the code paths and config knobs so that in future we can turn it on and tweak merging without redeploying. - We will clearly label results by their source in metadata (e.g., in combined results we might keep a field indicating which index produced that result), in case we need to handle them differently downstream or for debugging.

For this upgrade, the dual-index capability ensures **extensibility**, but we will likely keep it disabled until we have a use-case or want to experiment with external knowledge. It's mainly included in the design for completeness and future-proofing.

4. Reranker Integration: The second-stage reranker (a cross-encoder re-scoring model) is already implemented in the codebase. We will verify that it still works properly with the new embeddings. The reranker model (by default, `mini_lm`) is independent of the embedding dimension; it works on text pairs (query and chunk text) and produces a relevance score. So it should function regardless of embedding changes. The integration updates needed are: - Ensure that the reranker uses the **same set of candidate chunks** as retrieved by the new index. If we increase our initial retrieval count (e.g., retrieving top 50 with E5 for reranking), we should ensure the pipeline passes those to the reranker component. - The config `RERANK_CONFIDENCE_TAU=0.32` will be applied to the cross-encoder scores. We interpret this threshold as a cut-off for considering a result truly relevant. After reranking and normalizing scores (if needed), the system can drop any chunks whose reranker confidence is below 0.32. In practice, this means if the cross-encoder predicts relevance probability < 0.32 for a chunk, we exclude it from the final top `K_OUT`. This heuristic is meant to improve precision by filtering out low-confidence items, even if they were retrieved in top K, thereby providing the LLM with only the most relevant context. - We will maintain `RERANK_ENABLED=0` (off) for initial rollout. Our approach is to first see the gains from the embedding upgrade alone. If recall@5 improves sufficiently (≥10%), we may not need to enable reranking immediately. Eventually, however, we could consider enabling it to boost precision@1 (for instance, ordering the top result). The cross-encoder (MiniLM or BGE-based) will add some latency (tens of milliseconds for 50 pairs, possibly ~40–50ms on CPU). This is a trade-off we will evaluate later. The integration code already supports toggling this with the env var. - With the new embedding model retrieving more relevant documents in the top 5, the need for reranking might lessen (since the baseline recall is higher, and E5 embeddings capture relevance better). Nonetheless, reranking can still reorder the top results to ensure the best ones are ranked highest for the prompt, which is beneficial for answer quality. - We will test the reranker in a staging environment after the embedding switch to ensure it still functions. Specifically, we'll test that when `RERANK_ENABLED=1`, the system returns at most 8 chunks, and that if all cross-encoder scores are very low, the system might even return fewer (or none). We have to confirm how the code behaves if, say, zero candidates exceed the tau threshold – presumably the code would then take the top 8 by score regardless, or decide no answer can be given. We will handle this carefully to avoid empty context edge cases. For now, we'll assume at least some results pass the threshold for normal queries (0.32 is not extremely high). - We will not change the reranker model at this time; `mini_lm` (which likely refers to `cross-encoder/ms-marco-MiniLM-L-6-v2`) remains default. The config allows `bge_base` as an alternative cross-encoder (if implemented). We won't enable that by default, but the support remains for experimentation. (If the research had indicated BGE cross-encoder does better on some domains, we could test it later.)

5. Confidence Heuristic: Beyond the reranker's threshold, we can also implement a confidence heuristic on the embedding retrieval itself. For example, if the top cosine similarity from the vector search is extremely low (e.g., <0.2 on normalized scale), it may indicate the query doesn't match any content well (potentially an out-of-scope query). In such cases, the system might choose to not answer or to use a fallback source. This is not currently in the product requirements, but we mention it as a possible extension. With E5, because of

its training, even unrelated texts can sometimes have non-trivial cosine scores (~0.2-0.3), so a naive threshold could be misleading. We will rely primarily on the reranker's threshold (since the cross-encoder can better judge relevance). If needed, we can set a very low floor on vector similarity (e.g., if even the best chunk has cos < 0.15, we consider returning a "no relevant info found" answer). This is not a priority now, but the architecture will allow plugging in such logic if desired.

6. Number of Results (K values): We will continue using approximately 8 results in the final prompt (as currently configured). With the improved recall, those 8 should contain more relevant info. We might retrieve a slightly larger pool initially to feed the reranker. Concretely, if rerank off: we retrieve 8 (or maybe we could bump to 10 just to have a bit more coverage, but 8 has been our prompt limit). If rerank on: retrieve 50 (K_IN=50), rerank, then output top 8 (K_OUT=8). These numbers come from config and will remain unless we observe memory or prompt length issues (8 chunks of ~300 tokens each is about ~2400 tokens, which is within our prompt window along with the question and some margin).

7. Span Serialization Adjustments: We will keep the chunk text serialization format the same (repo path, symbol, etc.), as this structure has proven useful for relevance. One minor addition we will do is ensure that the **prefix** (`"passage: "`) is added at the very beginning, even before the repo-path. This means the first token the model sees is "passage:" which cues it to treat everything after as content. This shouldn't affect the rest of the text parsing, except we should double-check if our separator `•` is handled properly (likely fine). We also need to ensure the total token count fed into the model for a passage does not exceed the model's max (512). Our current chunk limit ~512 tokens should include the prefix, or we can trim a few tokens if needed to accommodate the added word. In practice, adding `"passage: "` (1 word + 1 punctuation token) is negligible in length.

8. End-to-end Flow Summary: When the new system is enabled, a query will go through: - **Embed (E5)**: `"query: ..."` → 768-dim vector (normalized). - **Search primary index** (E5 vectors, cosine) → e.g. top 50 candidates. - **(Optional) Search secondary index** (if dual-index enabled) → another list of candidates. - **(If dual) Merge candidates** (e.g. via RRF) → fused list. - **(If rerank enabled)** Re-rank top 50 from merged/primary using cross-encoder: - Score each (query, passage) pair with cross-encoder. - Sort by cross-encoder score. - Filter out any with score < 0.32 (tau). - Take top 8. - **(If rerank disabled)** Take top 8 from the initial retrieval list directly. - Return those 8 chunks as context for LLM. - Log relevant info (similarity scores, etc.) for analysis.

This design provides multiple fallback layers: - We keep the old index around (not queried unless we explicitly code for it, but it's there to switch back if needed). - We have dual index capability (though initially off, we could use it to query old index in parallel for comparison if we wanted to A/B test results quality). - We have a reranker to catch any subtle reordering needs or to drop spurious chunks if necessary. - The new embedding model significantly boosts recall, so the expectation is that relevant information will now appear in the retrieved set more often, improving answer coverage.

Storage & Migration Plan

Index Storage Format: For the new 768-dim index, we will use the same storage mechanism as before (to minimize new dependencies). If previously using Chroma, we'll create a new persistent Chroma collection. Chroma by default uses a SQLite (or DuckDB) database file to store embeddings and can also keep an in-memory index for fast search. We'll ensure the collection name or file path is unique for version 2. For

example: - If using Chroma: instantiate with `chroma_client.create_collection(name="llmc_index_v2", embedding_function=..., metadata={"version":2})`. This will create entries likely in a file `.chromadb2/` or augment the `.rag/index.db` with a new collection table. (If the latter, Chroma's collections have separate namespaces in the DB, so it can co-exist in one file. We need to confirm if using one DB file for two collections is supported; if not, we'll specify a separate path.) - If using raw SQLite: possibly create a new file `.rag/index_v2.db` with a fresh table schema (`CREATE TABLE embeddings_v2(id INTEGER PRIMARY KEY, vector BLOB, metadata JSON, ...);`). Alternatively, add a new table to `.rag/index.db`. But to keep things cleaner and truly separate, a new file is preferable. The code can easily be pointed to a different filename.

Schema Migration: We do not need to migrate or alter the old schema; we are adding a new one. So there's no complex migration in the sense of altering columns. It's more of a **data migration** (recompute and copy data to new store). The steps will be: 1. Spin up the indexing script with the new model (E5). 2. It reads all documents and produces the new embeddings and saves to the new index. 3. After verifying completeness, we'll have both `index.db` (old) and `index_v2.db` (new) available.

Because we will not serve queries from the new index until we flip a feature flag, we can safely build it in advance. There's no user-visible downtime for building; it's done offline.

Once we are ready to switch: - We will deploy the new code (which is capable of using either index based on config). - Initially, leave config to use old index (for a safety net during deployment). - After deployment, in a controlled manner, update the config (or toggle the feature flag) to point to `index_v2`. This could be done by restarting the service with `EMBEDDING_INDEX_NAME=index_v2` environment variable, or via a runtime flag if supported. We plan this as part of the rollout steps (e.g., enabling it first on a small percentage or on a canary instance). - The **switch is near-instant**: the service on startup will load the appropriate index file. Loading the new index might take a bit more RAM/time (especially if building an in-memory structure). We should measure startup time with the new index (it might increase due to double vector size, but still likely in seconds range which is fine).

We will implement logic to **version** the semantic cache as well. For example, if we cache query embeddings or results, we tag them with the index version. This prevents serving stale results from the old index when on the new one. On first run with new index, the caches will be cold (which is expected).

Normalization in Storage: We have decided to store **normalized vectors** (unit norm) in the index (since we'll likely generate them already normalized). This means our similarity search can simply use dot product or any inner product metric. If using Faiss, we'll use `IndexFlatIP` and feed normalized vectors, achieving cosine similarity effectively [5]. If using Chroma, we can specify `metric="cosine"` for the collection; Chroma will handle normalization internally or at query time. Storing normalized vectors has the benefit that if someone accidentally uses an inner product distance without normalizing, it's still correct.

One consideration: if we ever needed to support **L2 distances** or another metric, we would need raw (unnormalized) vectors. But since our retrieval is purely semantic and cosines are the norm for embeddings, this is acceptable. Also note, E5's raw vector norms might carry some significance (e.g., confidence), but since the authors explicitly normalize for retrieval use, we follow that practice.

Chroma specifics: If Chroma is our backend, it typically stores embeddings in a SQLite (DuckDB actually) and also builds an approximate index if configured. We should ensure to call `persist()` on the new collection so it's saved to disk. The `.rag/index.db` might contain multiple collections. If we do use the same DB file for convenience (two collections inside), we need to be careful to **distinguish collection names**. For clarity, using a separate file (like `.rag/index_e5.db`) might be better; it avoids any chance of index name collision or confusion, and we can swap files easily. The drawback is maintaining two files. But since we plan to eventually deprecate the old one, it's manageable.

Cleanup: During rollout, we'll keep both indices. After full migration and once we are confident we won't revert, we can remove the old `.rag/index.db` (or archive it). We might schedule that for a later date as a precaution. The code will also retain backward-compatibility toggles until we decide to remove them (likely after we're well past needing to fallback).

In summary, the storage plan is **parallel indexing**: build new index alongside old, version them, and allow runtime selection. No direct in-place migration is done (to minimize risk).

Retrieval Flow Updates

With the above changes, the retrieval flow will be updated as follows (including caching, search method, routing, fallback):

1. **Query Processing and Caching:** When a query arrives, we first normalize it (e.g., lowercase or other text normalization if any - unchanged from before). We then check the **semantic cache**:
2. If we have a cached result for this exact query under the current model+index version (and it's recent), we can reuse it. For example, if user repeats a question within a short span, we might bypass embedding and retrieval to instantly use cached top results. We will implement cache keys that include the embedding model or index version, e.g., `cache_key = f"query::{INDEX_VERSION}::{query_text}"`. This avoids mixing results between MiniLM vs E5 runs.
3. If no cache hit, proceed to embed the query with the new model.
4. We will cache the resulting query embedding as well (for potential reuse in multi-step interactions or repeated queries). This embedding cache maps (model, query_text) -> embedding vector [6]. Given E5's moderate overhead, caching can help if the same query appears frequently (though that may be rare in practice).

5. These caches are typically in-memory LRU caches. We will size them appropriately (e.g., a few hundred entries) to avoid growing unbounded. Since this is existing infrastructure, we'll mainly ensure they are flushed on model switch and functioning with new keys.

6. **Vector Retrieval:** Using the query's 768-dim vector, we query the vector store:

7. **Brute-force vs Approximate:** As discussed, initially we will likely use a brute-force search for simplicity and guaranteed accuracy. With, say, N = 10,000 vectors, this involves 10k dot products of length 768 – which is ~7.68 million multiplications, easily done under ~50ms in optimized C++/SIMD. We will measure actual latency. If we find this is creeping up or if N grows, we have the option to switch to an **HNSW index** for sub-linear search time. HNSW (Hierarchical Navigable Small World graph) is a proven ANN method that can give >90% recall with much fewer comparisons [12]. We could integrate Faiss's HNSW (IndexHNSWFlat) or use Chroma's Annoy/HNSW support if available.

This would be mostly configuration (e.g., building the index with `index.build_index(...)` in Chroma). For now, we note this as a future optimization. Our design target is to keep P95 retrieval latency well below 300ms (which is a common budget [14]), and we anticipate staying in the ~50ms range with brute-force for now.

8. **Routing logic:** If `DUAL_INDEX_ENABLED=1`, we will perform two retrievals in parallel:
    - One on the primary local index (E5).
    - One on the secondary index. If the secondary is remote (e.g., OpenAI embeddings on a remote store), this call might incur network latency. We need to handle that asynchronously to not block the primary results. We could issue both queries concurrently and then wait for both to complete (with perhaps a timeout for the slower one).
    - If the secondary is the old local index (for A/B testing), that's also fast and local. In that case, we can embed the query with the old model as well – but that means loading two models concurrently, which we currently aren't planning to do in one process. More realistically, A/B would be done by running two instances of the service (one with old, one with new). So in-process dual index will likely only be used for combining different sources that use the same query embedding (like using the same query vector on two different indexes that were built with the same embedding model). For example, if `DUAL_INDEX_PROVIDER=openai`, perhaps we call an OpenAI service with our query text, which returns some results (they'd embed internally).
    - Without specific requirement for dual retrieval now, we implement the scaffolding but might leave the actual second call as a stub or simple case (like route to an OpenAI embedding API if configured).
9. **Merging results:** If dual retrieval is done, we merge as described. If not, we just take the primary results. We will encapsulate this logic in the retriever component for clarity.

10. **Similarity scores:** We'll get a similarity score for each result (cosine or inner product). These scores in E5's space tend to be higher on average than in MiniLM's space. For example, relevant pairs might score ~0.85 instead of ~0.65. We might need to adjust any thresholding or UI display accordingly. (The threshold tau=0.32 is on reranker scores, not on embedding similarity, so no change needed there). We will include the raw similarity scores in the metadata passed along, as it can be useful for debugging or deciding if to use a chunk. These scores might also be logged.

11. **Post-Retrieval Filtering:** With the initial candidate list (from either one or multiple indexes), we apply any filters or sorting:

12. If metadata filtering by repository, date, etc., exists (not explicitly mentioned in prompt, but just in case), it remains applied. For instance, if user query has context like "for 2021 docs only", we would filter to chunks with that tag. The embedding model change doesn't affect that logic.
13. If we decide to incorporate the **confidence heuristic** at this stage: for example, if top vector similarity is below a certain cut-off, we might trigger an alternative path (like searching a keyword index, if one existed, or providing a fallback answer). Our design leaves room for such a step. Initially, we will likely not drop all results solely due to a low similarity, because doing so could result in the system saying "I don't know" more often than desired. Instead, we rely on reranker for a finer judgment. But we will monitor cases of obviously irrelevant retrieval and could introduce a rule if needed (e.g., if even the top-1 similarity <0.1, skip providing context).
14. If dual index is used in **fallback mode**, this is where it happens: e.g., if primary had no result above threshold X, then take secondary's results instead. Implementing this logic would involve checking

primary's top score; if below X, disregard primary list and use secondary's list (or merge them but bias towards secondary).

15. For now, with single index, this step is minimal: basically just proceed with the list.

16. **Reranking stage:** If `RERANK_ENABLED=1`:

17. We take the top `RERANK_K_IN` (50) results from the prior step. (If dual provided more, we might take 50 combined highest between both sources, though that detail will be handled by the merge logic).
18. We prepare input for the cross-encoder: typically, this means for each candidate chunk, we take the original query text (without the "query: " prefix, likely just the raw user question) and the chunk's text (without the "passage: " prefix and metadata, or maybe with some of it). We have to be careful what we feed the cross-encoder. Ideally, the cross-encoder should see the actual content text of the chunk (possibly including title or symbol name if that's important context, but probably yes). Likely we will feed it a pair: (query, chunk_text_without_repo_metadata) to avoid overwhelming it with file path noise. The current implementation likely already defines this. We'll verify and use the same approach.
19. The cross-encoder (MiniLM or BGE) will output a relevance score for each pair. If it's the MS MARCO MiniLM cross-encoder, the scores might be logits that roughly correlate with probability of relevance. We will then **sort candidates by this score (descending)**.
20. We then apply the `CONFIDENCE_TAU`: drop any candidate whose cross-encoder score is below 0.32. The rationale is to eliminate chunks that the cross-encoder is fairly sure are irrelevant to the query.
21. After filtering, we take the top `RERANK_K_OUT` (8) of the remaining. If fewer than 8 remain (worst case, none passed 0.32), one approach is to backfill with those that were below threshold (to still supply the LLM with some context rather than nothing). However, since the config explicitly allows fewer output than input, it might be acceptable to return fewer chunks. We should handle the edge case: if 0 chunks remain, perhaps do not attempt to answer (or we could relax the threshold in that case). This is a product decision. For now, we might ensure at least 1 chunk is present by taking the top scoring chunk regardless of threshold if the filter would empty the set.
22. These final reranked chunks are then passed on. Their ordering is now by cross-encoder confidence, which should put the most relevant snippet first (helping the LLM focus on it).
23. We will log metrics such as how many of the initial 50 were filtered out by tau and the average cross-encoder score, to tune tau if needed.

If `RERANK_ENABLED=0`: - We skip the above. The top `k` results from vector search (with any prior filtering) are directly used. In that case, the ordering is purely by embedding similarity. E5's improved embedding should make this top-k quite reliable in terms of containing relevant info (though maybe not perfectly ordered by relevance, but reasonably good).

1. **Output:** The final list of ~8 chunks is then formatted for the prompt (with citations). No changes are needed in how we format the prompt aside from ensuring the source references are still correct. Since chunk metadata (like `<repo-path>#<line>` or IDs) remains, citation format will remain the same. We should confirm that none of the prefixing or new symbols interfere with citation extraction logic (they shouldn't, as those are stripped out before answering or just treated as part of content). The user will see the same style of answer but hopefully with more relevant context.

2. **Caching of Results:** After retrieving and before generating the answer, we can store the results in the **query-result cache** (if implemented). This would map from (query text, index_version) to the list of top chunks we found. So if the user asks the exact same question again, we could skip directly to generation using cached chunks (saving embedding+search time entirely) [6]. We will update this caching to include index version so that if we switch back or forth, cached data doesn't bleed over incorrectly.

3. **Fallback Policy:** The design includes multiple fallback options:

4. **Use of old index**: In early rollout, we might run some queries through both old and new indexes (manually or via dual-index flag) to verify results. If we notice the new index consistently missing something that old had, we could consider dynamic fallback for that case. One possible automated fallback: if new index returns **zero** results (which should not happen unless index is empty or a filtering bug), then automatically query the old index and use it. This is a safety net for any unexpected issue.
5. **OpenAI index (if any)**: If configured, could serve as a fallback for out-of-domain queries.
6. **No-answer scenario**: If neither index yields anything above confidence threshold (in case we implement that check), the system could respond with a default "I don't have information on that" rather than hallucinate. That would be a new behavior. We haven't done that historically (the system tries to answer with whatever it has, even if loosely relevant). We will not implement a hard no-answer cutoff initially, but it's noted as a possible improvement to avoid hallucinations when retrieval fails.

During the phased rollout, an initial fallback is simply **don't enable the new path for 100% of queries until it's vetted**. We will possibly route a small percentage of queries to the new index (shadow testing or silent evaluation) before full cutover.

In summary, the retrieval flow changes are mainly: new embedding with prefix, adjusted index usage (possibly approximate search if needed), optional dual retrieval, and ensuring rerank + caching work smoothly with the new setup. The goal is that from an end-user perspective, the system still responds in the same format and speed, but with improved relevance in the supporting facts.

Evaluation Harness Plan

To confidently roll out this change, we have created an **evaluation harness** under `tests/eval/embeddings/` to quantitatively measure retrieval performance before and after the upgrade. This harness uses a set of test queries with known relevant documents (ground truth) to compute metrics like Recall@5.

The evaluation dataset consists of: - `queries.jsonl` – a list of sample queries (in JSON Lines format). Each line contains a `id` and a `query` text. These queries are representative questions that LLMC users might ask, covering various topics and difficulty levels. They may have been derived from real usage or curated to test known challenging cases. - `corpus.jsonl` – a list of documents or passages in the corpus, with their IDs and content. Each entry likely corresponds to one chunk or document that could be retrieved. In our case, since we have an existing index of chunks, this file might be a dump of those chunks (id plus text). It serves as the reference set for evaluation. - `qrels.tsv` – query relevances in TREC format. This file lists known relevant document IDs for each query. For example, each line might be: `<query_id>\t0\t<doc_id>\t1`, indicating that doc_id is relevant to query_id (the `1` is the relevance

grade). We likely use binary relevance (1 for relevant, assume 0 not listed means irrelevant). - `runs/` – a directory to store output runs from retrieval. We will produce run files (often `.tsv` or `.txt`) that list, for each query, the retrieved doc IDs and their ranks/scores. Typically, a run file in TREC format has lines like: `<query_id> Q0 <doc_id> <rank> <score> <run_name>`. Our harness likely writes two run files: one for the baseline (MiniLM) and one for the new method (E5).

The harness execution is orchestrated via Makefile targets for convenience: - `make eval.pre` – This target will run the evaluation using the **current (pre-upgrade) setup**. Specifically, it will use the existing index (MiniLM embeddings). It likely calls a script that loads `queries.jsonl`, for each query performs retrieval on `.rag/index.db`, and writes out a run file (say `runs/pre.run`). Then it computes metrics (recall@5, etc.) comparing to `qrels.tsv`. The output might be printed to console or saved in a file like `runs/pre_metrics.txt`. - `make reindex.e5` – This target will perform the re-indexing with E5 embeddings. Essentially, it triggers the migration script described earlier. For example, it might run: `python scripts/reindex.py --model intfloat/e5-base-v2 --output .rag/index_v2.db`. Under the hood, this will embed all documents with the new model and create the new index. We will implement this such that it reads from `corpus.jsonl` (or the original data source) to get all text to embed. After running this, the new index is ready for use in evaluation. - `make eval.post` – This target runs the evaluation using the **new E5 index** (post-upgrade scenario). It will ensure that the environment/config is pointed to the new index and model, then execute the retrieval for each query in `queries.jsonl`, outputting a run file (e.g. `runs/post.run`). Then it computes the same metrics using `qrels.tsv`.

We will likely integrate a tool like `trec_eval` or a custom Python script to compute metrics from the run files and qrels. The primary metric of interest is **Recall@5** – meaning the fraction of queries for which at least one relevant document is present in the top-5 results. We might also calculate other metrics such as Recall@1, @10, Mean Reciprocal Rank (MRR), or NDCG, but the success criterion given is specifically a $\geq$10% improvement in Recall@5.

Pass/Fail Logic: The harness will compare the baseline and new results: - Let `R@5_pre` be the recall@5 with the old model, and `R@5_post` the recall@5 with the new model. - We expect `R@5_post >= 1.10 * R@5_pre` (at least a 10% relative improvement). For example, if baseline recall@5 was 0.50 (50%), we need the new to be $\geq$0.55 (55%). If baseline was 78.1% as in one benchmark [7], new should be $\geq$85.9% (which E5 has been shown to reach ~83.5–86%, so it's in line). - The harness will output the values. We will consider the upgrade **successful (pass)** if this criterion is met or exceeded. - If the improvement is less than 10%, that's a potential **fail** (or at least, we do not consider it meeting the acceptance bar). In that case, we might not proceed to full rollout and possibly reassess the approach (maybe try a different model or check if something went wrong in implementation).

Additionally, we'll watch for any significant drop in precision metrics (e.g., we don't want recall to improve at the cost of retrieving a lot of irrelevant chunks, though the reranker or threshold can mitigate that). If available, we will check **Precision@5 or MRR** as well. However, since the prompt specifically calls out recall, our gating will focus on recall@5. The reranker (if off) means precision might dip a bit due to more diverse results, but if needed we can always enable reranking to refine precision after achieving recall gains.

Evaluation procedure: The `tests/eval/embeddings/` likely contains a script (or the Makefile calls one) that does:

```
# Pseudocode:
EMBEDDING_MODEL=MiniLM INDEX_PATH=.rag/index.db python eval_search.py --queries
queries.jsonl --index index.db --output runs/pre.run
python eval_metrics.py --qrels qrels.tsv --run runs/pre.run --metric recall@5 >
runs/pre_metrics.txt

EMBEDDING_MODEL=E5 INDEX_PATH=.rag/index_v2.db python eval_search.py --queries
queries.jsonl --index index_v2.db --output runs/post.run
python eval_metrics.py --qrels qrels.tsv --run runs/post.run --metric recall@5
> runs/post_metrics.txt
```

And then possibly a small diff or comparison is output.

We will document in `README` or internal docs how to run these targets. The Make targets encapsulate the environment switching, which ensures we evaluate exactly the two scenarios.

We will add any necessary logic in `eval_search.py` to handle the prefixing for the new model. For example, if `EMBEDDING_MODEL=intfloat/e5-base-v2`, the script will know to prepend "query:" and "passage:" accordingly. This ensures the evaluation truly reflects the new approach.

Make targets usage: - Developers can run `make eval.pre` before the change (or on main) to record baseline metrics. - After implementing the upgrade and reindex, run `make eval.post` to get new metrics. - We will store these results or at least report them in the SDD or PR for transparency.

Furthermore, for completeness, we might include a **Make target to actually perform the reindex and then evaluation in one sequence** (e.g., a target that depends on reindex.e5 then eval.post). But since we want to separately verify reindex, it's fine to run them stepwise.

The harness is also intended to be added to our CI pipeline if possible. We might incorporate a lightweight version of it in automated tests (with a smaller corpus) to ensure that recall does not degrade inadvertently in future changes.

Pass Criteria: This evaluation must show $\geq$10% improvement in recall@5 for the change to proceed beyond P0 stage (as per acceptance criteria). If the target is not met, we will treat it as a failed acceptance test: - Possibly try alternate models (the research doc mentioned BGE or others that had even higher recall like Nomic which was ~86.2% [7], though that model is larger/slower). - Or see if our chunking or prefix usage was suboptimal.

However, given E5's known performance, we are confident we'll hit the goal on our test set.

We will also track a few **qualitative checks** as part of evaluation: e.g., searching specific known queries where the old system failed. This might involve simply running the query through the dev environment and verifying the retrieved chunks. Those anecdotal tests complement the quantitative ones.

Finally, after deploying to production (even gated), we will monitor live retrieval success. If possible, we might conduct an **AB test** where a portion of real queries run with old vs new and measure some engagement metric or quality rating. But that might be beyond the initial scope; our offline eval plus any internal QA should suffice to validate the improvement.

Observability & Monitoring

Deploying a new embedding model and index calls for careful observability to ensure system health and performance remain acceptable. We will enhance logging and metrics collection in the following areas:

- **Latency Tracking:** We will measure and log the time spent in each stage of retrieval:
- `embedding_time_ms` – time to generate the query embedding (this may increase with E5; we log it to confirm it's within expectations, e.g., ~80ms average).
- `search_time_ms` – time to query the vector index and get initial results.
- `rerank_time_ms` – time to rerank (if enabled for that query).
- `total_retrieval_time_ms` – end-to-end time from receiving query to final list of chunks. These can be logged per query in debug logs or emitted as metrics to our monitoring system. Our goal is to keep total retrieval time within our budget (likely <300ms P95 as a component of overall request time [14]). We expect a small increase due to the larger model, but we'll verify TTFT (Time to First Token) remains good. If we see TTFT regressing significantly at P90/P95 (e.g., >200ms added), we'll investigate (maybe optimize or consider using GPU).

- **Throughput and Concurrency:** While we don't expect query throughput to change drastically, we will monitor CPU usage of the embedding step. If QPS increases or the embedding model saturates CPU, it could become a bottleneck. In logs/metrics, we will watch CPU utilization and the time waiting for embeddings when multiple queries come in concurrently. If needed, we can adopt optimizations like batch embedding (embedding multiple queries together) or running multiple model instances in threads. Initially, though, one embed at a time should suffice.

- **Cache Hit/Miss Rates:** We will instrument logs for cache usage:

- When a query embedding is fetched from cache vs computed fresh.

- When a query result is served from cache vs performed a fresh search. This will help ensure our caches are functioning and also give insight into how often queries repeat. We'll likely add counters or debug lines such as `CacheHit: query_embedding for "<query>" reused` or similar. We can aggregate these to compute a cache hit ratio. Although we don't expect a lot of repetition in queries, any caching helps latency.

- **Index Stats:** We will monitor the size of the new index (number of vectors, memory usage). This can be logged at startup: e.g., "Loaded embedding index v2: N=12345 vectors, dim=768, size=45MB in memory." Also, if using Faiss or Chroma, we might log if we're using brute-force or approximate. For approximate, we'd also log parameters (like HNSW M, ef).

- **Reranker Usage:** We will log when reranking is applied and its outcomes:

- A flag indicating `rerank_applied=true/false` for a query (so we know in logs if rerank was used).
- Possibly log the cross-encoder top score and how many results were filtered by tau. For example: "Rerank applied: top_cross_score=0.85, filtered_out=2 of 8 due to tau=0.32".
- If rerank is disabled (which it will be initially), this logging is skipped, but the framework is there if we toggle it for experiments.

- We want to see how often rerank would have made a difference. We might do some test runs with rerank on (in staging or a subset) and track metrics like MRR or precision. Observing those can guide us on whether to enable rerank in prod.

- **Similarity Score Distribution:** We can track the distribution of top-1 similarity scores for queries. Since E5's cosine scores might be generally higher, we can compute an average or histogram of the top result's score and maybe the score of the 5th result. This can be done offline from logs. The reason is to verify that E5 is indeed providing confident matches (we expect many queries will have top sim in 0.7–0.95 range for good hits [11]). If we see many queries with top sim below, say, 0.3, that might mean queries that get poor retrieval (which could indicate out-of-domain queries or index gaps). Those could be flagged for further analysis.

- **Memory and CPU Monitoring:** As mentioned, memory usage will increase due to the larger model and larger vectors. We should monitor the application's memory to ensure it stays within container or machine limits. The ~0.8 GB increase for the model plus additional ~some MB for bigger vectors should be fine if the instance had headroom. We will add a note to ops to double-check memory after deployment. If we see memory pressure, we might consider loading the model in half-precision or other optimizations.

- **Cost Logging:** In our scenario, using the local E5 model incurs no external cost (it's open source). However, the design allows for external providers (OpenAI) or could in future use OpenAI's embedding API. If at any point `DUAL_INDEX_PROVIDER=openai` is used, we will definitely log usage metrics:

- Number of external API calls and tokens sent/received (embedding tokens count).
- Approximate cost in dollars for those calls (so we can track expense).
- Right now, since we are not actively using openai in default, cost logging will mainly report $0 for retrieval. But implementing the logging now means if someone flips on an OpenAI-based retrieval, we have immediate insight into cost impact.

- Similarly, if the LLM generation uses tokens, that cost is separate; here we focus on retrieval cost. With local E5 and local reranker, there's no per-query cost except compute.

- **Quality Monitoring:** Beyond retrieval metrics, the ultimate goal is improved answer quality. We should keep an eye on any user feedback or automatic evaluation of answers:

- Do answers have more correct information and fewer "misses" where the needed info wasn't retrieved?
- If we have a mechanism to track "answer found in sources or not", we might see an uptick in that.

- This is harder to measure automatically, but we might run a set of known Q&A pairs through and see if the answer can be produced with new vs old retrieval.

- At the very least, we will monitor if there's any increase in user queries that result in "I don't know" answers or obvious hallucinations. A drop in those would be a good sign (implying the right info was retrieved).

- **Telemetry Correlation:** We will ensure the telemetry logs include identifiers to correlate query events:

- Query ID or some request ID.
- The index version or model used.
- The retrieved document IDs.
- This will allow us to trace, for a given request, exactly what happened (which index, which results, how long it took).
- This is useful for debugging if users report an issue or if we find a regression. For example, we could search logs for a particular query string to see what it retrieved then vs now.

All these observability enhancements serve to make the rollout **safe and transparent**. We'll be able to answer questions like: Is the new system slower? By how much? Are we actually getting better retrieval hits? Are we mistakenly pulling in more irrelevant stuff? Is anything breaking or erroring out?

We will also set up alerts if needed: - If retrieval latency suddenly spikes above a threshold (maybe our monitoring can alert if P95 goes over X). - If any exceptions occur in the embedding pipeline (like out-of-memory or model loading issues). - If the recall in production (harder to measure without ground truth) appears to drop (not directly measurable, but if we had a synthetic probe or count of "no relevant context" outcomes). - In general, we'll closely watch logs after enabling the feature for any anomalies.

Logging example: After changes, a log for a query might look like:

```
[INFO] QID=123, index_version=2, query="How to foo bar", embed_time=85ms,
search_time=10ms, results=50, top_sim=0.81
[INFO] QID=123, rerank_used=false, returned_k=8, doc_ids=[id1,id2,...],
sims=[0.81,0.78,...]
```

(This is illustrative; actual format might differ.) Such logs provide at-a-glance performance and outcome data.

In summary, observability is a key part of the rollout. We will implement fine-grained logging around the new embedding and retrieval steps, track any changes in latency (ensuring we remain within UX targets [14]), and measure the effectiveness through metrics and logs. This ensures we detect any negative side-effects early and have data to tune the system if necessary.

Rollout Plan

We will execute the embedding upgrade in phases, aligned with priorities P0 through P2, to minimize risk and ensure success criteria are met. The rollout will be gated by achieving the target Recall@5 uplift (≥10%) before proceeding to full deployment.

Phase P0 – Implementation and Testing (Development Phase): - **P0a: Code Implementation (Local)** – We will implement all necessary code changes behind feature flags. This includes integrating the E5 model (but not activating it by default), adding prefixing in the embedder functions, updating the indexing script for 768-dim, and ensuring the retrieval pipeline can handle dual index and reranker logic (even if off). During this phase, `DUAL_INDEX_ENABLED` will remain 0 and `RERANK_ENABLED` 0 by default. We'll run unit tests to confirm that the system still works with the old model (no regressions when flags are off). - **P0b: Build New Index (Offline)** – Using the updated code, run `make reindex.e5` on the full corpus to generate the new `.rag/index_v2.db`. This can be done on a dev machine or server. We'll verify the index size and that it contains all expected entries (e.g., by checking row count or a random sample). - **P0c: Offline Evaluation** – Run `make eval.pre` and `make eval.post` with the test harness. Compute Recall@5 and other metrics. If the **Recall@5 gain is >= 10%**, we pass this crucial checkpoint. We will document the metrics: - For example, if baseline R@5 was 0.70 and new is 0.80 (which is a 14.3% relative increase), we proceed. We'll also note any changes in MRR or others. If the target is not met, we do **not** proceed to P1 deployment; instead, we would consider alternative actions (perhaps tuning chunking or trying BGE model). In that scenario, this project might be iterated until criteria met. - **P0d: Internal Review** – Share the SDD (this document) and evaluation results with the team for review. Ensure everyone (devs, QA, perhaps product) agrees on the plan and results. Incorporate any feedback (for instance, if someone suggests adjusting tau or retrieving 10 instead of 8). - **P0 Milestone:** All implementation tasks completed, tests passing, and evaluation shows ≥10% recall improvement. At this point, the code is ready to merge with the feature disabled by default (so merging doesn't yet change production behavior).

Phase P1 – Staged Rollout (Verification in Staging/Canary): - **P1a: Deploy to Staging** – Enable the new embedding model in a staging environment. Set `EMBEDDING_MODEL=intfloat/e5-base-v2` and use the new index file in that environment. Conduct integration tests: - Does the system startup correctly with the new model (e.g., model downloads or loads without error)? - Are there any runtime errors when retrieving (e.g., shape mismatches in numpy arrays, etc.)? Fix any bugs discovered. - Run a suite of known queries in staging and manually verify results. For example, ask a question and see if the sources returned look relevant. Compare to what the old system would have returned (we can do side-by-side testing by querying a staging instance with old config). - Also test edge cases, like queries that have no good answer – ensure the system still returns something reasonable (or at least doesn't crash). - **P1b: Performance Check in Staging** – Simulate load or at least measure response times in staging: - If possible, use a profiling tool or just observe logs for the timing info we added. Confirm that average embedding+retrieval time is within expected range (for example, if it was ~100ms in dev runs, ensure it's similar in staging). - Ensure memory usage on the staging instance is within limits (no OOM, and the process has headroom). - **P1c: Canary Deployment** – Deploy the new code to production **with the feature still off** (meaning it's running the old model/index as default). This ensures the new code base is stable in prod environment. Because flags are off, users still use the old index at this point, so it's low risk. Monitor this deployment for any errors (even with old path, we changed some underlying code that could potentially affect things like caching or logging). - **P1d: Enable New Embeddings for a subset** – Using configuration flags, we will do a gradual ramp-up: - Turn on `EMBEDDING_MODEL=E5` and point to the new index on a small percentage of traffic or a single server (if we have multiple instances). For example, enable it on 10% of requests (if we have a way to

do that via a traffic router or feature flag system). If we cannot do percentage, we might deploy it to one canary instance and route some internal queries to it for testing. - Monitor the canary closely: track latency, errors, and quality. This might involve comparing logs from canary vs other instances. Also, if any user feedback comes (though for a small portion probably not immediate). - Check that citations in answers are correct (the new index chunk IDs match the correct docs for references, etc. This should be fine as we carried metadata over, but worth a quick spot check that the answer cites the intended document). - Particularly monitor any unexpected **increase in "I don't know" answers or irrelevant citations** – signs of retrieval failure or mis-ordering. - **P1e: Reranker Experiment (if needed)** – During this phase, if we find that some irrelevant chunks slip in at top positions, we might do a quick test by enabling `RERANK_ENABLED=1` on the canary and seeing if answer precision improves. This is optional; only if we suspect an issue. Since reranker is known to help precision, we might hold off on enabling it until we see actual need. - **P1f: Expand Rollout** – If the canary runs smoothly for a reasonable time (say a day or a few hours of peak traffic with no issues), proceed to roll out the new embedding to all production instances. This means flipping the default or enabling the feature flag across the board (P0 → P1 transition complete). We will do this in a controlled manner, perhaps instance group by instance group, to avoid any surprise overload. Given the new model uses more resources, we also ensure that auto-scaling or capacity is adjusted if needed (for instance, if CPU usage per request went up, we may need more instances for the same QPS). - **P1g: Monitoring Period** – After full enablement, closely monitor for at least 24-48 hours. Track: - Latency metrics: ensure P95 TTFT is still within acceptable range (if we see a big spike, we might need to optimize or scale out). - Any error rates: model load failures, etc. (E5 should be cached on disk after first load; if an instance for some reason tries to download weights at runtime, it could timeout – make sure all instances have the model ready, possibly preload it in the docker image or volume). - Quality metrics: if we have any feedback loops, see if user satisfaction improved. For example, if users could previously ask a question and not get an answer, do we now see those being answered? - If available, run the evaluation queries on production to double-check recall (only if non-invasive; we could simulate queries via an internal tool).

At the end of P1, we expect the new embedding strategy to be fully live for all users, provided all metrics look good and no issues surfaced. If any blocker issues occur in P1 (e.g., latency too high, or recall not as expected in real use), we will pause or rollback (see Backout Plan).

Phase P2 – Follow-up and Optimization: - **P2a: Solidify Defaults** – Once we're confident, we will make the new model and index the permanent default (not just behind a temporary flag). This might involve cleaning up any temporary toggles or ensuring configuration files default to the new settings. Essentially, codify that E5 embeddings are now the standard. - **P2b: Remove Old Index** – We will decommission the old MiniLM index. This includes: - Stop updating `.rag/index.db` (if any processes still were). We likely stopped ingestion into it as soon as we cut over (since any new content ingestion should go to the new index now). - Archive or delete the old index file to save space. Possibly keep a backup for a short time in case of unforeseen rollback needs, but we expect after a certain period, rollback would be done via redeploy, not using stale data. - If code has any branches for Index v1 vs v2, we can remove those once we're sure we won't go back. However, we might keep backward-compatibility for one release or so just to be safe. - **P2c: Performance Tuning** – With new system stable, we can look into optimizations: - If retrieval latency is slightly higher but still okay, maybe no action needed. If it's significantly higher, consider building an approximate index (HNSW) now to reduce query time. This can be done now that quality is proven. We'd test HNSW to ensure recall stays high (we can configure it for >0.99 recall relative to brute force). If we do implement HNSW, we'd do it in a minor update and ensure it doesn't degrade recall. - Consider enabling the reranker by default if we see obvious gains for users. This would be a separate decision, possibly requiring

its own evaluation (maybe measure some QA performance with vs without rerank). The infrastructure is in place; it's toggling `RERANK_ENABLED=1`. If turned on, verify no huge latency hit. The cross-encoder we have is small so likely fine (and we only rerank 50->8). - Evaluate the `RERANK_CONFIDENCE_TAU`. The current 0.32 might be tuned. For instance, if we find that often 8 good results exist and some 9th is just below threshold which could have been included harmlessly, maybe we lower tau to include it. Or if we see irrelevant stuff sneaking in above 0.32, maybe raise it. This tuning can be done with our evaluation set or additional data. - If not already, integrate any hybrid search aspects (e.g., maybe incorporate a keyword search for certain queries or multi-step retrieval). These are beyond current scope, but the improved embeddings might reduce the need for pure keyword fallback. - Another optimization: ensure that on startup, the embedding model is loaded once and shared (if our architecture had multiple processes, ensure they don't each load redundant copies if possible). But that might not apply depending on how our service is structured. - **P2d: Documentation and Knowledge Transfer** – Update documentation such as developer docs, runbooks: - Document the new index structure and how to reindex in the future (for new data or if we update the model again). - Update any on-call runbook to note the config flags (if something goes wrong, how to disable E5 quickly, etc., until we remove toggles). - Provide a summary in the project's roadmap or changelog (see next section) so stakeholders know this improvement happened and what it entails. - **P2e: Final Acceptance** – At this point, we measure the success criteria formally: - The recall@5 target was met. - No major negative impact on latency or other metrics (we define acceptable range – say, <20% increase in average latency, which our data suggests we achieved). - All relevant teams signed off that the feature is working (QA doesn't find issues, product is happy with results). - We can declare the project complete.

Timeline & Milestones: - P0 tasks (coding + offline eval) – target completion in Week 1. - P1 (staging & partial prod) – Week 2 (with a few days of monitoring at each step). - Full production enablement ideally by end of Week 2 or early Week 3. - P2 (cleanup and optimizations) – Weeks 3-4, but these are less urgent. We may run P2 tasks in parallel with other work, as long as the system is stable. - A key **go/no-go checkpoint** is after P0 evaluation. Another is during canary: if any anomalies, we may extend testing or fix before proceeding.

Go/No-Go Criteria: - If recall improvement is confirmed and no major performance regressions in test, go for canary. - If canary shows stable performance, go for full rollout. - If any step fails criteria (e.g., recall <10% improvement or latency too high), we stop and address issues (could be switching model, more tuning, etc.) before continuing.

Throughout, we maintain the ability to toggle back to the old index if needed, until we're fully confident.

Backout Plan

Despite careful testing, if at any point during or after rollout we encounter severe issues, we will execute a backout (rollback) to the previous embedding setup.

When to back out: - If the recall improvement is not actually observed in practice (e.g., our offline eval was positive, but in real usage maybe different query distribution doesn't see as much benefit, and stakeholders are not satisfied). - If there's a significant performance degradation that impacts user experience (e.g., TTFT becomes unacceptably high, or system throughput drops). - If new failure modes appear (e.g., the model returns some encoding that crashes downstream, or memory leak, etc.). - If user feedback indicates worse

answers or other quality issues directly tied to the retrieval (though this is unlikely given improved recall, it's something to monitor).

Immediate rollback steps: 1. **Toggle Flags to Old Settings:** Since we built this behind flags, the fastest mitigation is to revert the configuration: - Set `EMBEDDING_MODEL` back to the MiniLM model name (or point code to use version 1 index). - Point `INDEX_NAME` or path back to `.rag/index.db` (the old index). - Ensure `DUAL_INDEX_ENABLED=0` (so it doesn't try to do something odd with mismatched indices). - We likely can do this by updating environment variables or config and restarting the service (a process that might take a couple of minutes). This does not require redeploying code, just using old data. - Once restarted, the service will load the old 384-dim model and index and operate as it did before the change. - We will verify that the switch succeeded by checking logs (the model initialization log should indicate MiniLM loaded, and queries should now show retrieving from old index). - Because the old index was left intact and we did not modify it, it should still be fully functional and up-to-date (we might need to ensure that any new documents ingested since enabling E5 were also indexed into the old one; if not, there's a slight inconsistency, but since the time window is short, this is negligible or acceptable temporarily). 2. **Redeploy Old Version (if needed):** If toggling config is not enough or we suspect code paths have changed, we can roll back the deployment to the previous version of the application (one that did not include the E5 changes at all). This may be more time-consuming (depending on deployment pipeline), so the preference is to use the runtime flags. But as a failsafe, we keep the old version readily available for redeployment. - Note: Because our code changes were mostly additive/behind flags, the new code with flags off should mimic the old code behavior. So redeploy might not be necessary unless a bug in new code affects even the old path. If such a bug is found (for example, maybe our caching changes inadvertently break something even with old model), then deploying the old code is prudent until we fix that bug. 3. **Monitor Post-Rollback:** Once back on old system, confirm that the issues are resolved (e.g., latency goes back down, errors stop, answer quality is back to previous level). We'll continue to keep an eye out because rollback itself can have complications (maybe caches need warming up, etc.). 4. **Communicate:** Inform the team that we rolled back and why. If it was user-impacting (like downtime or poor answers), possibly communicate with support or product. 5. **Debug and Fix Forward:** Analyze what went wrong with the new approach: - If it was performance, can we optimize (maybe it needed GPU or an ANN index we didn't enable). - If it was quality (perhaps our test set didn't cover some query types where E5 underperforms, or maybe some bug in prefix usage), identify and address it. - Once we have a fix or mitigation, we would attempt another rollout (likely treating it as a new P0-P2 cycle with the adjustments). - Possibly run more extensive tests or a longer A/B test to be sure the second attempt will succeed. - Example scenario: Suppose after enabling E5, we find memory usage spiked and caused swapping. We rollback. Fix might be to load the model in 8-bit quantized mode to reduce memory, then try again.

Pointing Index Back: Because we have not deleted or altered the old index, pointing back is straightforward. One caveat: any new data that got indexed only in the new index (if we ingested fresh content while E5 was on) would not be in the old index. If that happened, we might quickly ingest those into the old index too, so that no user queries fail to find them after rollback. If needed, we can run the old ingestion pipeline for any missed documents. (Given the likely short timeframe of a rollback scenario, new content addition is probably minimal.)

Dual Index Option: Interestingly, we could use dual-index as a backup approach in a partial rollback. For example, if the new index mostly works but has some holes, we could run in dual mode: primary E5, secondary old, merge results. That way, if E5 misses something old would catch it. This could be a strategy if we see only some queries regress. However, this complicates things and likely isn't needed if we can just

revert fully. But it's an available tool: turn `DUAL_INDEX_ENABLED=1` and `DUAL_INDEX_PROVIDER=voyage` (assuming "voyage" is old local) such that it always queries both. This would increase latency (doing two searches and two model embeddings), but might ensure recall doesn't drop. This would be a temporary measure while fixing issues. We prefer a clean rollback though for simplicity.

Roll-forward Plan: After rollback, once fixes are in, we would roll forward (redeploy the new changes) when ready. Essentially we'd attempt the rollout again, possibly in a more cautious way depending on what failed:
- If performance was the issue, maybe ensure we have more capacity or improved code before trying again.
- If model choice was an issue (maybe E5 didn't perform in some niche), possibly choose another model (like BGE or an instruction-tuned variant). But E5 has broad strong performance, so unlikely to fully change the model.

Worst-case scenario contingency: If for some reason none of the open-source models meet our requirements, we might consider integrating a fallback to OpenAI's embeddings or a larger model. That would be a major pivot (with cost implications), but our design already anticipates possibility of an OpenAI provider. So in an extreme case, we could quickly switch to using OpenAI's text-embedding-ada-002 for queries and docs (would need to have those embeddings computed for our corpus). This is not planned, but it's a fallback plan beyond rollback: e.g., if E5 had some fundamental issue. However, given E5's benchmarks, we don't foresee needing this.

Backout plan verification: We will essentially test the backout during rollout by virtue of having the toggles. For example, during canary, if something looks off, we'll flip back to old on that instance and see if it returns to normal. This practice run ensures that in an emergency, we know the steps and they work quickly.

In short, the backout plan leverages our feature flags and preserved old index to allow a quick reversion to the previous state with minimal disruption. We will keep this capability until the new system has been proven in production for some time (at which point we may remove the old code/index to reduce maintenance, but only when we're sure it's no longer needed).

Acceptance Criteria

For the project to be considered successful and complete, the following acceptance criteria must be met:

- **Recall Improvement:** Achieve at least **10% relative improvement in Recall@5** on the evaluation dataset, compared to the MiniLM baseline [7]. *(Example: if baseline recall@5 is 0.50, new recall@5 ≥ 0.55.)* This was the primary goal set; our offline tests must confirm this uplift before rollout.
- **No Recall Regression on Any Critical Query:** No significant drop in retrieval performance for any known important queries or categories. In other words, the new embedding should not fail cases that the old one handled. (Our testing and dual-run checks will verify this.)
- **Acceptable Latency Impact:** The embedding upgrade should **not degrade end-to-end latency beyond acceptable limits**. Specifically, **P95 retrieval latency** should remain within our target (e.g., <900ms [14] for retrieval, and ideally much less). In practice, we aim for added latency < ~50ms per query on average. The overall user-perceived response time should not increase noticeably. We will

verify this in staging and initial rollout. If P95 TTFT increases, it should be <20% and still under the UX threshold. If any latency spike, it must be mitigated (via scaling or ANN index) to pass.

- **System Stability:** The system runs reliably with the new model:
- No memory leaks or crashes related to the embedding model (e.g., loading 768-dim model does not exhaust memory over time).
- The service can handle at least the same query throughput as before (or any needed scaling is in place).
- All existing unit/integration tests pass (after updating any expected values if needed). This includes tests around the RAG pipeline, caching, etc.
- **Correctness of Integration:** The new index integrates seamlessly:
- Document chunk metadata and citations continue to work correctly (the answers cite the intended documents, IDs match, etc.).
- The re-indexing process captures the entire corpus with no omissions. (We verify by count and spot-check that all docs are present in the new index.)
- No errors in retrieval logic (e.g., no shape mismatches when computing similarities with 768-dim vectors, which could happen if some part of code was hardcoded to 384).
- Dual-index and reranker logic remain functional (even if off, they should not break anything when toggled). We should be able to toggle reranker on and see it working, for example, as a test.
- **Observability in Place:** Metrics/logging added in this project are functioning:
- We can observe embedding and retrieval timings, cache usage, etc., in logs or monitoring dashboards.
- If any anomaly occurs, we have sufficient telemetry to diagnose it quickly (this is subjective, but essentially we ensure logs have the info as planned).
- **Feature Flag Control:** It must be possible to enable/disable the new embedding easily via config. This was crucial for safe rollout and backout. Acceptance means: by flipping the environment variables and restarting, the system cleanly switches between old and new modes.
- **Rollout Milestones Reached:** We have successfully:
- Deployed to production and enabled the feature fully (or to the agreed extent) without needing to permanently rollback.
- Monitored for a suitable period (e.g., one week of production use) and saw no regressions or issues.
- The old index and model can now be retired (meaning we're confident enough to not keep them active).
- **Documentation & Deliverables:** All relevant documentation has been updated:
- This SDD is completed and stored (in `DOCS/SDD/Embeddings_Upgrade_SDD.md`).
- `DOCS/Roadmap.md` has an entry summarizing this change.
- Code is well-commented where changes were made (especially around new parameters like prefix usage).
- If any onboarding or ops docs needed changes (like new env vars), those are done.
- **Changelog and Issues:** A changelog entry prepared (see below) and all planned GitHub issues (P0, P1, P2 tasks) are either resolved or created for follow-ups. Essentially, every work item identified has an owner and status.
- **No Negative User Impact:** After deployment, user feedback should be neutral to positive regarding answer quality. Specifically:
- There should be a reduction (or no increase) in user complaints about missing or incorrect answers due to retrieval.
- If we have user ratings on answer helpfulness, we'd ideally see an uptick. (This may not be immediately measurable, but at least we want no drop.)

- No new pattern of failures introduced (like if E5 had some systemic bias or odd behavior that surfaces in answers – none is known, E5 is a general model, but we remain watchful).
- **Acceptance Sign-off:** The project stakeholders (lead engineers, product manager, etc.) formally sign off that the change meets requirements and can be considered complete.

Only when all the above conditions are satisfied will we consider the embedding strategy upgrade fully accepted. If any are not met, we will address them (e.g., tune further, fix bugs) until they are.

Implementation Checklist

Below is a checklist of implementation tasks, mapped to file locations and function names in the codebase, to ensure all required changes are made:

- **Model Loading & Configuration:**
- [ ] **Add new model path constant** for E5 in config (e.g., in `tools/rag/settings.py` or `.env`): e.g., `EMBEDDINGS_MODEL_NAME = "intfloat/e5-base-v2"`. Default can still be the old model for now. Ensure the application reads this to decide which model to load.
- [ ] **Update model loader** in `tools/rag/model_loader.py` (or wherever the SentenceTransformer is initialized) to use the above config. For example, modify `load_embedding_model()` to do:

```
model_name = os.getenv("EMBEDDINGS_MODEL_NAME", "sentence-transformers/all-MiniLM-L6-v2")
model = SentenceTransformer(model_name)
```

  and if the chosen model is an E5 variant, maybe set `normalize_embeddings=True` by default when encoding.

- [ ] **Ensure device placement** is handled (if needed, allow using CUDA if available for E5, otherwise CPU).

- **Prefixing Logic:**

- [ ] **Embed function for queries** – in `tools/rag/retriever.py` or similar module, locate the function that takes a query and returns an embedding. Modify it to prepend `"query: "` to the query text if the current model is E5 (or more generally if the model name contains "e5" or using a config flag like `MODEL_FAMILY=E5`). E.g.:

```
def embed_query(text: str) -> np.array:
    if EMBEDDINGS_MODEL_NAME.startswith("intfloat/e5"):
        text = "query: " + text
    return model.encode([text], normalize_embeddings=True)[0]
```

  (If we rely on SentenceTransformer's internal normalization, we ensure that parameter is set.)

- [ ] **Embed function for documents** – in the indexing pipeline (perhaps `tools/rag/indexer.py` or `scripts/index.py`), when preparing each chunk text for embedding, prepend `"passage: "`. For example, in the loop that goes through chunks:

```
for chunk in chunks:
    text = chunk['text']  # assembled from repo, symbol, etc.
    if EMBEDDINGS_MODEL_NAME.startswith("intfloat/e5"):
        text = "passage: " + text
    emb = model.encode([text], normalize_embeddings=True)[0]
    # store emb in index
```

  This may be in `scripts/reindex_embeddings.py` or similar.
- [ ] **Verify prefix addition** doesn't break something else: e.g., if the chunk text was used as an ID or key anywhere, ensure we don't prefix in those places by accident. (We likely only prefix right at embedding time and not persist the prefix in storage, except in vector values which is fine).

- [ ] Possibly **abstract prefix logic**: We could implement a wrapper embedding function that automatically adds prefix based on a parameter (is_query bool). This reduces code duplication. E.g., `embed_text(text, role="query"| "passage")`.

- **Vector Store Schema & Index Handling:**

- [ ] **Adjust index dimension** in the code that initializes the index. For example, if using FAISS:

```
index = faiss.IndexFlatIP(768)
```

  instead of 384. If using Chroma:

```
chroma_collection = chroma_client.create_collection(name="llmc_v2",
dimension=768)
```

  If the dimension is not explicitly set, ensure the first vector added is of length 768 so that it infers correctly. If we have an assertion in code that checks vector length (to avoid inconsistency), update it to expected new value when using new model.
- [ ] **Implement separate index name or path:**
  - Introduce `INDEX_NAME` or `INDEX_PATH` config. E.g., in `.env` or settings: `INDEX_PATH=".rag/index_v2.db"` for new. In code where the index is loaded (maybe in `tools/rag/vectorstore.py` or at app startup), use this path. Probably we have something like:

    ```
    index_path = os.getenv("INDEX_PATH", ".rag/index.db")
    ```

    Update accordingly for new default later.
  - Alternatively, if index is loaded via Chroma by name, use a name from config.

- Ensure the system can still fall back to old path if needed (so likely have both values configurable).

- [ ] **Dual Index integration:**

  - In `tools/rag/retriever.py` (or pipeline), implement logic:

    ```
    if DUAL_INDEX_ENABLED:
        results_primary = primary_index.query(q_emb)
        results_secondary = secondary_index.query(q_emb)  # this might
    call openai or old index
        results = merge_results(results_primary, results_secondary)
    else:
        results = primary_index.query(q_emb)
    ```

    You might need to set up `secondary_index` earlier based on `DUAL_INDEX_PROVIDER`. E.g., if provider is "openai", define a function to call OpenAI's API with q_emb or query text; if "voyage", maybe instantiate a second local index from old data.
  - Implement a simple `merge_results(list1, list2)` using RRF or rank merging. Possibly, if time is short, a simpler union sorted by score (assuming scores comparable if both are cosine on same model – but if one is different model, scores not directly comparable). RRF is safer for disparate sources [13]. We can do:

    ```
    def merge_results(res1, res2, K=50):
        merged = {}
        for i, (doc, score) in enumerate(res1, start=1):
            merged[doc] = merged.get(doc, 0) + 1.0/(i+5)  # 5 or some
    constant to dampen
        for j, (doc, score) in enumerate(res2, start=1):
            merged[doc] = merged.get(doc, 0) + 1.0/(j+5)
        sorted_docs = sorted(merged.items(), key=lambda x: x[1],
    reverse=True)
        return sorted_docs[:K]
    ```

    This is a simple RRF with k=5 (common).
  - Also, load/initialize secondary index based on provider:
  - If "voyage": this might be a pointer to old local index. We could instantiate it similarly to primary but from old data. Maybe have it as `old_index = load_index(file=".rag/index.db")`.
  - If "openai": define how to retrieve. Maybe call OpenAI's vector search if something like Pinecone is set up (not specified). Possibly it's out of scope to fully implement now; at least leave a stub or log "not implemented".
  - Keep dual off by default, so this code might not run unless turned on.

- **Re-ranker Integration:**

- [ ] **Verify cross-encoder model loading:** Ensure the code that loads the reranker uses the correct model name from config (`RERANK_MODEL`). If "mini_lm", presumably we load `CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')`. If "bge_base", ensure we know what model that refers to (maybe `cross-encoder/ms-marco-MiniLM-L-12-v2` or some BGE cross model if exists). If not sure, consider `bge_base` meant for a bi-encoder usage; but given config, leave as option.

- [ ] **In retrieval pipeline**, after initial retrieval:

```
if RERANK_ENABLED:
    candidates = results[:RERANK_K_IN]
    reranked = reranker.rerank(query, candidates)  # implement this
    final_results = reranked[:RERANK_K_OUT]
else:
    final_results = results[:DESIRED_K]
```

The reranker logic might exist in `tools/rag/rerank.py` or similar. Check if there's an existing function. Possibly named something like `apply_rerank(query, docs)`. If not, implement using CrossEncoder:

```
pairs = [(query, doc['text']) for doc in candidates]
scores = reranker.predict(pairs)  # returns array of scores
scored_docs = list(zip(candidates, scores))
scored_docs.sort(key=lambda x: x[1], reverse=True)
filtered = [doc for doc, score in scored_docs if score >=
RERANK_CONFIDENCE_TAU]
```

then take top K_OUT from filtered (or if filtered is empty, consider fallback as discussed).

  - [ ] Ensure to **normalize cross-encoder scores** if needed. Some cross-encoders output logits that aren't probabilities. If our tau=0.32 is intended as a probability, we might need to apply sigmoid to scores if the model outputs logits. Check model documentation or output range:
  - The cross-encoder miniLM model likely outputs a single score that's not bounded 0-1 (but we could fine-tune or calibrate it). In practice, we might have to adjust tau accordingly or simply use relative ranking and not absolute threshold. However, since tau is given, maybe it's known to correspond roughly to a probability. Possibly the cross-encoder could be one that was trained for classification (relevant vs not) and outputs a probability.
  - For now, implement threshold on raw score as given, with understanding that it was probably chosen empirically.
  - [ ] Add logging for rerank decisions (like how many filtered, etc., as planned in Observability).
  - [ ] Ensure the reranker uses the **raw query text** (without "query:") because cross-encoder doesn't expect that prefix (prefix is only for bi-encoder E5).
  - [ ] Ensure reranker uses chunk text properly: we might exclude some of the metadata in the chunk text when feeding cross-encoder, if that metadata is not helpful for sentence relevance. Possibly the function already handles that (like maybe it just uses chunk['body'] or so).
  - [ ] Test reranking on a sample query in dev to ensure it returns reasonable ordering.

- **Semantic Cache & Caching Adjustments:**

- [ ] **Keying cache by model/index version:** If there's a cache structure (maybe in `tools/rag/cache.py` or within retriever), update it:
  - For embedding cache, could be a dict like `embed_cache = { (model_name, text): vector }`.
  - For query result cache, `result_cache = { (index_version, query): [results] }`.
  - If using an LRU cache object, we might incorporate model as part of key by concatenating.
  - Alternatively, a simpler approach: whenever we flip model, we just clear these caches (since mixing would be wrong). We can implement a function to clear caches on startup of new model or when config changes. But since config change triggers restart anyway, caches start empty naturally.
- [ ] **Invalidate old cache entries** on deployment of new model. Could do a version number in cache implementation. Possibly not needed if restart flushes memory. For persistent caches (if any), better to differentiate by key.

- [ ] Ensure the caching logic still wraps around retrieval properly. If we have new intermediate steps (like dual index), the cache probably should store final results after merging, not one index's result. But since dual is off, not a big worry. If on, caching might need to incorporate which combination of indices were used.

- **Index Building Script:**

- [ ] In `scripts/reindex_embeddings.py` (if it exists) or wherever `make reindex.e5` points:
  - Implement loading of the corpus. Possibly, we can leverage `corpus.jsonl`: If present, read each line's text and metadata. If not, perhaps the script knows how to fetch all content (maybe scanning the repository for all files, etc. – likely out of scope here to reimplement, so probably easier is reading the prepared corpus file).
  - Use the new embedding model (with prefix "passage:") to embed all items.
  - Store them in a new collection or DB:
  - If SQLite, do `INSERT` for each vector and metadata.
  - If Chroma, use `collection.add()` with embeddings and metadatas in batches.
  - If Faiss flat file, accumulate in a numpy array and then write out using Faiss write index.
  - After adding all, save/persist if needed (Chroma `collection.persist()`).
  - Print summary (like `N vectors indexed`).
  - Handle any chunking if needed (but likely corpus.jsonl is already chunked content).
- [ ] If `make reindex.e5` is not yet in the Makefile, add it:

```
reindex.e5:
    @echo "Reindexing corpus with E5 embeddings..."
    python scripts/reindex_embeddings.py --model intfloat/e5-base-v2 --
output .rag/index_v2.db
```

This is pseudocode; actual might vary. If no CLI args, use env: `EMBEDDINGS_MODEL_NAME=intfloat/e5-base-v2 INDEX_PATH=.rag/index_v2.db python scripts/reindex_embeddings.py`.

- [ ] Ensure that the script can run to completion in a reasonable time. If corpus is huge, consider adding a progress output or multiprocessing. But likely manageable.

- [ ] We should also consider updating any ingestion script that runs continuously (if any) to now index new documents with E5. If such exists (like if we have a `tools/rag/indexer` that adds docs on the fly), update it to use new model (once switched over). But since we are mostly doing a one-time reindex, we might not worry about live ingestion until after rollout (then ensure all ingestion uses new pipeline by default).

- **Testing and Validation:**

- [ ] **Unit tests for embedding dimensions:** If there are tests in `tests/` that assert the vector dimension or recall values, update them. For example, a test might have expected the embedding dimension to be 384; now it should expect 768 when E5 is configured.
- [ ] **Update eval tests**: Possibly we have tests that run a mini evaluation. We will add our `queries.jsonl` etc., so maybe no code but ensure the harness scripts themselves work (perhaps write a small test to open qrels and ensure no formatting issues).
- [ ] **New tests**:
    - Could add a test to ensure prefixing works: e.g., a small function that calls embed_query on "hello" returns same vector as if we manually did "query: hello" through model (but that requires model loaded, which might be heavy for a test context, so might skip).
    - At least test that our `merge_results` logic returns expected output given sample inputs.
    - If possible, test that switching the env from old to new yields different results for a known query (like simulate that known relevant doc now appears).

- [ ] **Manual tests**:

    - Run the application locally with the new model and ask a question, verify results.
    - Possibly use the evaluation harness manually to double-check recall on a few known items.
    - If a UI is available (like a chat frontend), try queries before/after.

- **Logging & Observability Code:**

- [ ] Insert logging statements as per Observability plan:
    - In retrieval function, after embedding query: log (maybe at debug/info) the time taken. Use Python `time.perf_counter()` around steps.
    - After vector search: log time and top similarity (and perhaps an excerpt of top result for debugging).
    - If dual index: log how many results from each and maybe one from each top.
    - After reranking (if used): log top cross score and any filtered count.
    - Use a consistent format and include a query ID or something (if not easily available, maybe a truncated query string).
    - Ensure not to log entire user query in production logs if that's a privacy concern – but probably fine if internal. Alternatively, log just a hash of it.
    - E.g.,

```
logger.info(f"Retrieval: query='{q[:20]}...', index=E5,
embed_time={embed_ms}ms, search_time={search_ms}ms,
```

```
top_sim={top_sim:.3f}, rerank={'Y' if used_rerank else 'N'}
(rerank_time={rerank_ms}ms)")
```

    - Additionally, log any errors caught (like if embedding fails).
- [ ] Possibly integrate with an existing metrics collector: if we use Prometheus or similar, we might expose counters/histograms for these times. That might be beyond scope, so logs might suffice for now.

- [ ] Monitor printing: ensure log statements won't flood too much. They could be at debug level for per-query detail. We might keep them at debug and elevate them to info on canary for analysis.

- **Housekeeping:**

- [ ] **Roadmap.md** update: prepare an entry describing the change (see next section).
- [ ] **GitHub issues**: Create issues for tasks not done yet or future improvements (like enabling rerank by default, or refactoring code).
- [ ] **Remove any temporary code**: If we added support for both 384 and 768 models in the same code, after full rollout we might remove 384-specific branches. But likely keep until we decide to drop old path entirely. Possibly mark TODO to remove in future.
- [ ] **Documentation**: If there's a user-facing config or difference (maybe none, since internal), just ensure devs know about new env vars via README.
- [ ] **Verify no security/compatibility issues**: E5 is an open model, license MIT, so fine. Just note if using it fits any compliance (should be okay).

Each of these tasks should be checked off during development. Upon completion, we will run the evaluation and proceed with rollout as per plan.

Changelog & Issue Tracking

Changelog Snippet for `DOCS/Roadmap.md`:

- *2025-11-XX*: **Semantic Search Embeddings Upgrade** – Upgraded the vector search embedding model from MiniLM-L6-v2 (384-dim) to E5-base-v2 (768-dim). This change improves retrieval relevance (Recall@5 boosted by ≥10% in tests) and enhances the system's ability to find pertinent code and documents. The new approach uses **"query:"/"passage:" prefixing** for better model understanding, stores normalized embeddings for cosine similarity, and integrates an optional cross-encoder reranker for improved precision. The vector index schema has been versioned to support 768-dimensional vectors (existing index preserved for rollback). Observability has been enhanced to monitor embedding latency and retrieval confidence. This upgrade is behind a feature flag initially and will be rolled out progressively (P0→P2) to ensure performance stability. Developers can reindex existing content using `make reindex.e5` and evaluate via `make eval.post`. *(Ref: Embeddings Upgrade SDD)*

Proposed GitHub Issue List (by Priority):

- **P0:**

- *Upgrade Embedding Model to E5-base-v2 (768-dim) with Prefixing*: Implement new embedding pipeline using `intfloat/e5-base-v2`. Add "query:"/"passage:" prefixes and ensure L2 normalization. Update model loading code and verify embedding output dimension is 768.
- *Re-index Corpus with New Embeddings*: Create script/Make target to build a new vector index (`.rag/index_v2.db`) using E5 embeddings. Populate with all existing content chunks, preserving metadata. Ensure new index is stored without affecting the old index.
- *Adapt Retrieval Logic for E5 & Index V2*: Update retrieval code to use the correct index based on config flag. Handle 768-dim vectors in similarity search (Faiss/Chroma). Maintain backward compatibility so old index can still be used if needed.
- *Implement Semantic Cache Versioning*: Prevent cache collisions between old/new embeddings by keying caches with model or index version. Clear or segregate caches on model switch.

- *Evaluation Harness & Testing*: Set up `tests/eval/embeddings` with queries, corpus, qrels. Add Makefile targets (`eval.pre`, `eval.post`) to compute Recall@5. Validate that new model achieves ≥10% higher Recall@5 than baseline. Ensure all unit tests pass (update expected dims, etc.).

- **P1:**

- *Deploy & Monitor Embedding Upgrade (Staging/Canary)*: Roll out the new embedding model in a staging environment, then as a canary in production. Monitor latency (TTFT), memory usage, and query result quality. Gradually increase traffic to new index when metrics are acceptable.
- *Observability & Logging for Retrieval Pipeline*: Add detailed logging/metrics for embedding latency, vector search time, cache hits, reranker usage, and result similarities. Ensure these appear in logs/monitoring dashboards for analysis during rollout.
- *Dual-Index Support for Hybrid Retrieval*: Finalize support for `DUAL_INDEX_ENABLED`. Allow querying secondary index (e.g., old index or external provider) in parallel. Implement result merging (e.g., RRF) so that if enabled, both indexes contribute to final results. (This feature is off by default but should be functional for experimentation.)

- *Reranker Integration & Threshold Tuning*: Verify cross-encoder reranker works with new embeddings. Conduct a test enabling `RERANK_ENABLED=1` in staging to evaluate impact on precision. Tune `RERANK_CONFIDENCE_TAU` if necessary. (Default remains off; create plan to enable later if beneficial.)

- **P2:**

- *Full Rollout & Old Index Deprecation*: Enable the E5-based embedding index as the default in production (100% traffic). Once stable, decommission the old MiniLM index. Remove or archive `.rag/index.db` and any code branches for the old embedding (if no longer needed for rollback). Ensure new index is kept up-to-date for any content changes going forward.
- *Performance Optimization (ANN Index)*: If needed, implement approximate nearest neighbor index (HNSW or similar) for the new embeddings to improve query latency at scale. Benchmark recall vs. brute-force to ensure no significant drop. Switch to ANN index if it provides clear latency gains under load.
- *Enable Cross-Encoder Reranking by Default*: Based on post-rollout analysis, decide whether to turn on the reranker to improve answer precision. If enabled, monitor impact on latency and quality. Adjust

`K_IN`, `K_OUT`, or tau as needed. Update default config (`RERANK_ENABLED=1`) if it consistently improves results.

- *Documentation & Cleanup*: Update internal documentation (README/Architecture docs) to reflect new embedding model and usage of prefixes. Remove feature flags and conditional logic for old model if we are confident in the new setup. Close out all related tasks and document the achieved recall improvement and any follow-up actions.

These issues cover the implementation, rollout, and follow-up work for the embedding strategy upgrade, categorized by priority. Each issue corresponds to a deliverable in the design and will be tracked to completion.

Exact Reindex Command:

To rebuild the vector index with E5 embeddings from scratch, run the following command in the project root:

```
$ make reindex.e5
```

This Make target will execute the indexing script using the new model and output to the designated new index file. Under the hood, it sets the necessary environment and calls the Python script, for example:

```
EMBEDDINGS_MODEL_NAME="intfloat/e5-base-v2" INDEX_PATH=".rag/index_v2.db"
python scripts/reindex_embeddings.py
```

This will ingest the entire corpus, apply the `"passage: "` prefix to each chunk, compute 768-dimensional E5 embeddings (with L2 normalization), and store them in `.rag/index_v2.db` (or the configured collection). The result is a fresh index ready for use in retrieval. After running this, you can proceed with `make eval.post` to verify the recall improvements, and then update the application config to point to the new index for serving queries.

---

1 2 8 9 10 11 intfloat/e5-base-v2 · Hugging Face
https://huggingface.co/intfloat/e5-base-v2

3 sentence-transformers/all-MiniLM-L6-v2 · Hugging Face
https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

4 7 Open-source embedding models: which one to use? : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1nrgklt/opensource_embedding_models_which_one_to_use/

5 6 12 13 14 Architecture of RAG Building Reliable Retrieval Augmented AI - CPI Consulting
https://www.cloudproinc.com.au/index.php/2025/09/15/architecture-of-rag-building-reliable-retrieval-augmented-ai/