ChatGPT

Graph-Enhanced Dynamic Scoring Plan for LLMC Retrieval System

Executive Summary

Recommended Architecture: We propose a multi-stage **graph-augmented retrieval pipeline** for LLMC that dynamically adjusts scoring per query. The design integrates **hybrid retrieval (lexical + dense)** as a first stage, followed by **graph-based candidate expansion** and **LLM-powered re-ranking**. The retrieval scoring will be **adaptive** – for example, a code-centric query triggers higher weights for code index results and graph neighbors, whereas documentation queries favor textual explanations. A lightweight gating mechanism (initially heuristic, later a learned classifier) will choose or blend routes (code vs. docs) per query. Graph signals from LLMC's repository graph (call graphs, "where-used" references, doc links) will enhance both retrieval and re-ranking: top results' neighbors in the graph will be included or boosted, and a graph-based re-ranker (e.g. a small Graph Neural Network or link-analysis algorithm) will refine the ordering [1] [2]. Finally, a **human feedback loop** will be established to continuously evaluate and adjust the system—using graded relevance labels and preference judgments to fine-tune weights or train learned rankers. Crucially, all components map onto existing LLMC interfaces (routing, scoring, fusion, etc.), ensuring feasibility within LLMC's local-first design.

Why This Fits LLMC: LLMC already separates code/document embeddings, has a routing module, and expects a `.llmc/rag_graph.json` graph artifact. Our plan leverages these: we add graph-based expansion in `search_spans()` after initial retrieval, incorporate graph-based features in `Scorer` (e.g. boost if a file is linked to a top result), and enhance `fusion.py` with more robust rank fusion. Because LLMC's focus is local repositories on modest hardware, we favor methods that are **lightweight and efficient**. For example, rather than calling huge external models, we use local 4–7B LLMs (like Qwen-3 4B or Vicuna 7B) for re-ranking, which research shows can approach GPT-3.5 performance in ranking [3]. Graph algorithms (like PageRank on a small subgraph of results) and small cross-encoders can run within milliseconds to tens of milliseconds [4], balancing quality gains with latency. The architecture is modular: each enhancement (graph expansion, reranker, fusion change, etc.) can be toggled off via config to fall back to the simpler baseline if needed.

Expected Gains: This plan directly attacks the observed *"docs-over-code"* bias in implementation queries. By infusing code connectivity and intent-aware scoring, code files that implement a function will rank higher when the query implies *"where/how is this implemented"*. Graph expansions mitigate missing recall – if a relevant code file wasn't retrieved by embeddings, it might be pulled in via a connected doc or test that was retrieved. We anticipate **higher recall of code references in code-intent queries** (measured by a new Code@k metric) and improved overall relevance (nDCG, MRR). Prior studies show graph-enhanced reranking can boost nDCG by ~5–8% and recall ~6% in text retrieval [5], and LLM rerankers can add another few points of improvement in complex queries [6]. Our approach should substantially reduce cases where users have to manually search for code after getting only docs. At the same time, for documentation-oriented queries, we preserve quality by still retrieving top docs (graph signals will mostly provide additional context or be disabled when not beneficial). Using an **adaptive approach per query**, we avoid one-size-fits-all tuning: the system can route and score differently for “How is X implemented?” vs “How to use X library?”.

Risks and Mitigations: Key risks include **score scale mismatches** (dense vs. BM25 vs. graph scores), **graph noise** (irrelevant neighbors diluting precision), and the **cost/latency of LLM reranking**. We mitigate these by careful calibration and toggle-able components. For example, we will implement **score normalization or rank-fusion** to safely merge different sources, and impose thresholds on graph expansions (only add neighbors if initial confidence is high, to avoid spurious hops). The LLM reranker will run only on a small candidate set (e.g. top 5–10) and can be disabled for very short queries or if latency is critical. To prevent over-bias toward code, the gating logic will fall back to a mixed retrieval if confidence in code-intent is low. We will also invest in **evaluation**: an offline test set with human relevance labels (especially for code queries) will allow us to tune weights (like `code_boost`) and catch failure modes (e.g. if an irrelevant code snippet consistently gets boosted due to a graph link, we’ll adjust the algorithm or graph data). An online A/B test can ensure that user satisfaction (measured by e.g. fewer follow-up queries or higher success rates) improves. The plan includes continuous monitoring of retrieval quality and periodic re-calibration, especially when the embeddings or LLM components are updated (to avoid drift).

Phase-wise Implementation: We outline a phased roadmap. **Phase 1 (Quick Wins):** Introduce reciprocal rank fusion and intent-based rerouting using simple rules – minimal changes to config (e.g. adjust `code_boost` when code keywords detected) and adding an RRF option in `fusion.py`. This yields immediate improvements in code result presence without heavy computation. **Phase 2 (Intermediate):** Integrate the repository graph: load `.llmc/rag_graph.json` and use it to pull in connected files for top results, plus implement a lightweight graph-based reranker (e.g. scoring connected clusters higher). Also, deploy a small local LLM (or cross-encoder) to rerank top-k results with a carefully designed prompt – improving fine-grained relevance. **Phase 3 (Best Quality):** Train a learned router (classifier) for query intent on labeled data, and possibly a learned ranker (e.g. a Graph Neural Network or LambdaMART model) using features from embeddings, BM25, and graph connectivity. Incorporate human feedback at this stage to fine-tune these models. Each phase builds on LLMC’s existing architecture, adding new config flags (ensuring **backward compatibility** if a feature is turned off). By the end, LLMC will have a **dynamic multi-scoring system** that is *“graph-aware”*, *“LLM-informed”*, and continuously *“human-aligned”*.

Must-Do Now vs. Later: Initially, we recommend focusing on **high-ROI, low complexity changes**: e.g., enable **multi-route retrieval with learned weighting** (possibly learning an optimal code/doc weight), and use **rank fusion** to merge dense and lexical results in a robust way. These can be done in weeks and directly address the immediate problem (code being under-ranked). In parallel, start collecting human judgments on a sample of queries—this will guide further tuning and provide training data. **Later (medium-term)**, implement graph-based augmentation and the LLM reranker when we have the capacity to evaluate them thoroughly (and perhaps after iterating on the graph generation to ensure it’s reliable). **Long-term**, invest in the fully learned components (classifier, learned ranker) once enough data is gathered; these offer the highest upside in optimizing the system but also require the most effort and risk. By phasing the implementation, we ensure that LLMC sees incremental improvements quickly, while laying the groundwork for more sophisticated enhancements as validation data and confidence in the approach grow.

Taxonomy of Methods

In approaching dynamic multi-scoring for Graph-RAG (graph-enhanced Retrieval-Augmented Generation), we categorize relevant methods as follows:

- **Graph Construction & Utilization:** Techniques to create and leverage graphs from data for retrieval. This includes **document structure graphs** (nodes might be sections, code blocks, etc., with edges for hierarchy or references), **knowledge graphs** linking entities or concepts, **citation or hyperlink graphs**, **code dependency graphs** (call graphs, import graphs), and **cross-modal graphs** linking code and docs. Some methods explicitly traverse these graphs to find relevant nodes (e.g. entity->related entity->document chains) [2], while others encode graph context in embeddings or use graph algorithms to rank results (like PageRank on the citation network). *Multi-hop retrieval* is often enabled by graphs: a query might require joining two pieces of info, and graph links can guide the retriever to hop between related content that a flat search would miss [2]. For example, GraphRAG methods in knowledge bases perform BFS/DFS along edges that match the query's relations to find an answer path [7] [8]. In code search, a graph could allow jumping from a documentation page to the actual code implementation via a link edge.

- **Retrieval Stages in Pipelines:** Modern retrieval often occurs in **multiple stages**:

- A **first-stage retriever** uses inexpensive methods (e.g. BM25 or dense embeddings) to get an initial candidate list from the entire corpus.
- A **second-stage reranker** then refines that list with more sophisticated analysis (cross-encoders, LLMs, etc.).

- Sometimes a **third stage** or additional rerankers follow, especially in high-precision scenarios. Within first-stage, one can also have a **hybrid approach** (dense + lexical) to improve recall. There are specialized retrievers like **multi-vector models** (ColBERT, SPLADE) that capture multiple aspects of text but at higher cost. Also, **query expansion** (either via pseudo-relevance feedback or using LLMs to generate reformulations) can be considered a stage: expanding the query and re-searching to retrieve what initial query missed [9] [10]. Each added stage can improve quality but adds latency and complexity. For example, BM25 (lexical) recall is high for exact matches, dense retrieval finds semantic matches, and combining them covers more ground at the cost of combining scores.

- **Re-ranking Paradigms:** After initial retrieval, **rerankers** determine the final order. Paradigms include:

- **Pointwise reranking:** Score each candidate independently for relevance (e.g. a cross-encoder that outputs a relevance score given query and doc).
- **Pairwise reranking:** Compare candidates in pairs, often used in learning-to-rank (e.g. LambdaRank) – model learns to say which of two is more relevant.
- **Listwise or Setwise reranking:** Consider the entire set of top-k results and produce an optimal ordering. LLM-based rerankers often operate in a listwise manner by taking multiple documents as input and outputting an ordered list [11] [12]. This can capture inter-document comparisons ("Doc A covers the answer better than Doc B…").
- **Graph-based reranking:** Here, candidates are treated as nodes in a graph with edges representing relationships (similar content, hyperlinks, shared entities, etc.) [13]. A **GNN (Graph Neural Network)**

or algorithm propagates scores through this graph, so that mutually reinforcing candidates (e.g. several results pointing to the same function or concept) boost each other [14]. This can also include **cluster-based rerankers** (find clusters of related results and favor the central ones) or classical link analysis like HITS/PageRank on the candidate set [15].

- **Fusion-based reranking:** Not a reranker per se, but merging results from different retrieval methods into one ranked list. Techniques like **Reciprocal Rank Fusion (RRF)** rank results by a combination of their rank positions in each list, without an explicit learned model [16]. Learned fusion (see below) can also be seen as a reranker that takes multiple scores as input.

- **Score Fusion & Calibration:** When combining different retrieval routes (lexical vs. semantic, code vs. docs), raw scores are often **incomparable** – e.g., cosine similarity from embeddings vs. BM25 scores. Fusion methods can be:

- **Rank-based fusion:** e.g., **RRF** assigns a score like $\sum_{sys} \frac{1}{k + \text{rank}_{sys}}$ for each candidate [16], making fusion robust to score scales. **Borda count** or other voting schemes also fall here. These require no training and often improve recall by giving any method's top results a chance.
- **Score normalization:** calibrate scores into a common range or probability. For example, one can apply a sigmoid to dense similarities or use min-max scaling on BM25 scores per query. **Z-score or quantile normalization** across a result list is another approach (though sensitive to outliers). Proper calibration can allow a fixed weight combination (e.g. 70% dense + 30% BM25 after normalization).

- **Convex combination vs. learned fusion:** A simple approach is a weighted sum of scores with a fixed weight (which could be tuned on a validation set) [17]. More advanced is **learning to fuse**, e.g. train a regression model that takes features like BM25 score, dense score, document type (code or doc), etc., and outputs a combined relevance score (KnowledgeIntegration's **K^2** model follows this idea with calibrated scoring). Learned fusion can be as light as one parameter (the weight between systems) or a full ML model. Research indicates that a well-tuned linear combination can outperform naive RRF and is sample-efficient to learn (only a small labeled set needed to set the weight) [17]. However, it requires relevance judgments for training.

- **Dynamic Scoring & Gating:** Instead of one static scoring formula, dynamic methods adapt to the query:

- **Intent detection/gating:** First, classify the query (e.g. code-related, documentation, configuration query, test-related, etc.). Then choose a specialized retriever or adjust weights. LLMC's router is a heuristic form of this (keywords trigger code route). A learned approach could use a classifier or even an LLM prompt to predict intent ("Is the user likely asking for code implementation?"). In retrieval research, mixture-of-experts models like **RouterRetriever** use a gating network to pick among multiple expert retrievers per query [18] [19]. In our context, the "experts" are the code vs. docs indices or even different embedding models for code, docs, config files, etc. Dynamic gating can also mean using a confidence measure: e.g., if the query embedding is very close to some code vectors (high max cosine), then clearly it matches code content, so boost code results; if it's mostly natural language, rely on docs.
- **Uncertainty-aware scoring:** If the system is unsure about query intent or finds only low-confidence matches, it might broaden retrieval (use both routes equally, include more diverse results). Alternatively, it might engage an LLM to reformulate the query or might rank by diversification

(ensuring both code and docs appear in results to hedge bets). This ties into **bandit algorithms or online learning**: over time, if users consistently click code for a certain query pattern, the system should learn to favor code for that pattern. Contextual bandits could adjust retrieval strategy based on real-time feedback, gradually shifting weights.

- **Mixture-of-experts retrieval:** This is a formal approach where multiple scoring experts (say, BM25, dense, and graph) each give a score, and a gating mechanism decides how to mix them per query. One can imagine training a small network that given query features outputs weights for each scoring component. This is dynamic fusion learned from data.

- **Calibration for routing/reranking:** This refers to ensuring that intermediate decisions (like the router's confidence) and final scores are on a meaningful scale. For instance, using a **Platt-scaled probability** from an intent classifier to decide route weights, or using temperature scaling on LLM's relevance scores to avoid overconfident extremes. Calibration helps in not over-committing to one route unless sufficiently confident, and in making scores between routes more comparable.

- **Human-in-the-Loop Paradigms:** Approaches that incorporate human feedback either offline (labeling data to influence models) or online (real-time preferences).

- **Graded relevance judgments:** Using human annotators to label retrieval results on an ordinal scale (e.g. 0 – irrelevant, 1 – somewhat, 2 – highly relevant) for a set of queries. This forms a "gold set" to tune and evaluate the system. It's a classic approach to IR evaluation, adapted here to code/docs context.
- **Preference judgments:** Instead of absolute ratings, ask which of two results is more helpful, or whether adding a certain context improved the answer. Pairwise preferences can be used to train ranking models (via e.g. Bradley-Terry or RankNet loss) and may be easier for humans than consistent absolute grading.
- **LLM-assisted judging:** Using an LLM to automate some relevance assessments (like a "AI judge"). For example, given a query and a document, have an LLM read them and assign a relevance score [20]. This can accelerate labeling but needs careful prompt/rubric design to avoid LLM errors. Often, a human can then audit a subset or difficult cases.
- **Active learning for labeling:** Instead of labeling a random sample of query-results, use active learning to pick the most informative items – e.g. results where two ranking methods disagree, or queries where the system is uncertain. This ensures human effort has maximal impact on improving the model.
- **Continuous feedback integration:** Capture implicit user feedback (clicks, time spent, success in follow-up questions) and explicit feedback (thumbs up/down on answers). These signals can then inform retrieval adjustments. For example, if users often scroll past the first snippet and click the second, that might indicate a re-ranking is needed.
- **Alignment and safety:** In contexts where the LLM uses retrieved info, human oversight ensures the retrieval promotes **faithfulness** (the model shouldn't cite irrelevant code) and respects user intent. Human-in-loop can help catch cases where an LLM reranker might "hallucinate" a document's relevance. Additionally, human curators might enforce that certain sensitive files (e.g. containing credentials) are never retrieved, aligning with policy.

This taxonomy guided our plan: we combine graph construction (code and docs graph) with multi-stage retrieval (dense+lexical first stage, LLM/GNN second stage), using both score fusion and dynamic gating, and we close the loop with human feedback. Each component has prior art and evidence backing its inclusion, as detailed next.

“Top 10 Most Promising Techniques” Shortlist

From the above categories, we highlight ten techniques likely to yield significant gains for LLMC’s retrieval, with an assessment of each:

1. **Graph-Based Candidate Expansion (Query-guided)**
2. *What it is:* Using the repository graph to **augment retrieval results with related nodes**. After initial retrieval, identify top results and fetch directly connected files (neighbors in call graph, references, lineage). For instance, if a user asks about function `foo()`, and a documentation page about `foo` is retrieved, the code file defining `foo` (linked via the graph) would be added to the candidates. This is analogous to pseudo-relevance feedback but uses graph links instead of term overlap [21].
3. *Why it matters:* It tackles the “missing code” issue. If the first-pass retriever misses the relevant code but finds something related (a test or a doc), the graph ensures the code isn’t overlooked. This dynamic expansion improves **recall of truly relevant items**. Graph traversal is powerful for **multi-hop queries** or implementation questions where a doc might mention a feature and the graph leads to its implementation [2]. It can also reveal connected context (e.g. related functions) that enriches answer completeness.
4. *Evidence strength:* **Strong.** In knowledge base QA and academic search, graph-based expansion greatly improved recall [22] [2]. A recent study on listwise LLM rerankers shows merging initial results with graph-neighbor results improved nDCG@10 by up to 13% and recall by 28% [23]. The concept stems from **pseudo-relevance feedback** and clustering hypothesis, which are well-established in IR: documents similar to a known relevant document are likely relevant [24]. These principles carry over to code/docs via repository graphs.
5. *Integration complexity:* **Medium.** We can implement this in LLMC’s `search_spans()`: after initial search, for each top-N result (configurable, e.g. $N=3$), load its neighbors via `graph_index.where_used_files()` or `lineage_files()`. Merge those neighbors into the result list (ensuring no duplicates). This requires the `.llmc/rag_graph.json` artifact to encode at least file-to-file edges (function calls, references, etc.). We’ll need to decide scoring for new neighbors – e.g. inherit a fraction of the source’s score or use a default lower score. Config knobs: `graph_expansion.enable` (bool) and perhaps `graph_expansion.top_n_neighbors = k`. The main complexity is ensuring we **don’t blow up** the candidate list: we might limit to adding at most e.g. 5 neighbors total. This is a straightforward addition data-wise, but requires careful score normalization (graph-added items shouldn’t unfairly outrank original hits unless appropriate).
6. *Failure modes:* If the graph is noisy or a top result is only tangentially related, expansion could pull in irrelevant files (“graph drift”). For example, a broad doc page linking to many code files might flood results with off-topic code. Also, if a malicious or irrelevant link exists (e.g., a test that imports many modules), it could drag those in. There’s also risk of **duplicating** what dense retrieval already found, offering little new info but complicating ranking.

7. *When NOT to use it:* If the query is very precise (e.g. explicitly names a function that is directly retrieved in top 1), expansion might be unnecessary and could introduce confusion. Also, for purely conceptual questions with no clear graph ties (e.g. “what is polymorphism” in docs), graph expansion may add noise. We should disable or minimize expansion when the router detects a **broad natural language question** where textual relevance is more important than code connectivity. We can also condition on the graph’s coverage: if the graph is incomplete or unreliable for a repository, better to avoid using it until it’s improved.

8. **Graph-Based Re-ranking with GNN or Link Analysis**

9. *What it is:* Building a small graph among the retrieved candidates (and possibly their immediate neighbors) and using a **Graph Neural Network or PageRank-like algorithm** to adjust scores. Each candidate is a node; edges could be: “file A imports file B”, “doc X references function in file Y”, or content-similarity above a threshold. A GNN (e.g. Graph Convolutional Network or Graph Attention Network) can propagate relevance signals: if two results support each other (connected), they reinforce each other’s score [14]. Alternatively, treat it like a network of votes: perform a few iterations of a random-walk score propagation (akin to PageRank/HITS) so that *hubs* (files referenced by many others) or *authority* nodes gain score [15].
10. *Why it matters:* This method **captures interdependencies** in results that a linear ranker can miss. For example, if several top results all link to a particular code file, that code file likely is a central answer (maybe a core function used everywhere). Graph reranking will boost that central node. It also helps diversify or remove redundancy: if many results are clones (e.g., multiple tests calling the same function), the graph structure can identify that cluster and perhaps up-rank one representative while down-ranking duplicates. Overall, it can improve precision by **elevating “consensus” relevant nodes** and improve recall by considering the broader subgraph of the query’s context [14] [1].
11. *Evidence strength:* **Medium-Strong.** Graph neural re-ranking (GNRR) is a recent innovation shown to improve ranking metrics with modest overhead. Di Francesco et al. (2024) achieved +5.8% average precision improvement using a corpus graph GNN reranker in <40ms/query [5]. MacAvaney et al. (2022) similarly saw ~8% nDCG gains by reranking with a kNN graph among candidates [5]. The survey of graph-based reranking reports consistent gains across domains, including **passage retrieval** and recommendation [25]. This indicates the approach is sound. However, most published examples are on text or multimodal data; our scenario (code+docs) is a bit special, but the relational nature is if anything more pronounced (code often has explicit links).
12. *Integration complexity:* **High.** Implementing a GNN reranker requires model code (potentially using a library like DGL or PyTorch Geometric) and possibly training data to learn weights. A lighter alternative is implementing a simple link-analysis heuristic: e.g., build an adjacency matrix of candidates and perform two iterations of score smoothing (each node gets some score = base_score + α * sum(neighbor_scores)). This could be done without learned parameters, just a formula (α can be a config). That would fit in `llmc/rag/rerank.py` after initial scoring. A full GNN with training would require an **offline training pipeline** with labeled data (which we might get from human judgements). Integration points: we’d extend the `Scorer` or a new `GraphScorer` to use graph context. We also need to parse `.llmc/rag_graph.json` to get the edges between candidates (could pre-index an adjacency list for each file). Config: `graph_rerank.enable=true`, and possibly parameters like α or the number of iterations.
13. *Failure modes:* **Over-smoothing** – a known issue where GNNs can make everything seem similar, hurting discriminative power [26]. In our case, if not careful, a highly connected but not relevant file might accumulate score from others. For example, a central utility file (logging, constants) linked by many might erroneously get boosted even if the query is about a specific feature. There’s also cost: constructing and processing a graph each query adds overhead (though for ~10–20 candidates it’s small). If the graph data has stale or wrong links, we could boost the wrong things. Additionally, if we already expanded neighbors, we must ensure not to double-count that in reranking – perhaps use one or the other.

14. *When NOT to use it:* If the candidate set is very small (e.g. only 1–3 results), graph reranking does nothing useful. If the graph relationships are sparse or irrelevant for the type of query (e.g. conceptual docs with few cross-links), a GNN won’t help and might just add noise. Also, for extremely

time-sensitive contexts (where every millisecond counts), we might skip GNN due to complexity. We can have a fallback: if `len(candidates) < X or no significant edges found`, then bypass graph rerank. Essentially, use it when we have a rich local graph among results, otherwise default to base scores.

15. **Reciprocal Rank Fusion (RRF) for Hybrid Retrieval**

16. *What it is:* A **rank-based fusion** method that merges results from multiple systems by their rank positions [16]. In practice, we'd run (a) the dense embedding search on code, (b) dense search on docs, and possibly (c) BM25 (lexical) on combined or separate indices. Instead of relying on raw score comparison, we assign each result a fused score like $\frac{1}{k + \text{rank}*{dense}} + \frac{1}{k + \text{rank}*$ (for example with k=60 as a common choice). This way, an item highly placed in any list gets a boost. We then sort by this fused score. RRF is simple: it doesn't need training or even parameter tuning except the constant k. }
17. *Why it matters:* RRF is **robust to score scale issues**. LLMC's current approach uses a weighted max of normalized scores, which can be brittle if one score dominates. RRF instead treats each rank list democratically, often yielding a good mix of results. It's particularly useful for **diverse signals**: here, lexical search might catch exact keyword matches (e.g. an error message string in code) that dense misses, while dense finds semantic matches. RRF ensures neither is lost. It has a track record of strong performance in IR competitions as a strong baseline for fusion. For our use, it's an easy way to incorporate multi-route fan-out (code and docs routes) such that even if the doc similarity scores are higher on a numeric scale, a code result that was rank 1 in the code route will still appear near the top in the fusion.
18. *Evidence strength:* **Strong (baseline)**. RRF has been known in IR for over a decade as a near-"oracle" combination method that is hard to beat without careful tuning [17]. Contrary to some assumptions, a recent analysis found RRF's effectiveness can depend on parameter choices, but overall it's competitive, and a properly tuned linear fusion can slightly outperform it in domain-specific cases [17]. In absence of training data for a custom fusion weight, RRF is a **safe default**. Many production systems use rank-fusion or score normalization plus sum. Our expectation is that RRF (with k ~ 50-60) would significantly improve code query success by preventing relevant code (even if lower scoring) from being overshadowed by docs.
19. *Integration complexity:* **Low.** We can implement RRF in the `fusion.py::fuse_scores` function as an option. If `fusion.mode = 'rrf'` in config, then instead of current max-weight logic, do: for each route's result list, assign a score = 1/(k+rank). Sum these for each document (document identified perhaps by file path). Then produce a ranked list. We should be careful: currently `fuse_scores` deals with maybe two lists (code vs docs). RRF naturally extends to any number. We'll need to handle the case where the same file appears in both lists (in which case it gets contributions from both). We could still sum them. This is straightforward and doesn't require additional data structures beyond collecting results.
20. *Failure modes:* If one retrieval method is very poor (retrieves mostly junk), RRF might still inject that junk because it "fairly" gives everyone a shot. For example, if we always do a code search even for clearly doc questions, RRF might unnecessarily include a random code file at rank ~5. This could hurt precision for doc queries. To mitigate, we might include only reasonably high scoring candidates or use a cutoff (e.g., only fuse top 10 from each route). Another subtlety: if there's a large difference in list lengths, e.g. BM25 returns 100 items but dense only 10, the tail of the BM25 list might unfairly get small but nonzero scores that push irrelevant items up. A common heuristic is to truncate each list to some top-M before fusing. We should monitor if fusion brings any off-topic results.

21. *When NOT to use it:* If we have reliable training data to learn an optimal weight, a simple weighted fusion might outperform RRF (as noted by Pinecone's study) [17]. So eventually, we might replace RRF with a learned combination once labels exist. Also, RRF can be skipped when only one retrieval source is used (obviously). If the code and docs indexes have already been jointly embedded (not in our case, but hypothetically), a single model might be better fused by score. But given our current architecture (separate indices), RRF (or its learned variant) is almost always beneficial. We might disable RRF in extreme cases where one route is known to be much noisier – but a better solution is to filter that route's output rather than abandon RRF.

22. **Learned Hybrid Scoring (Calibrated Score Combination)**

23. *What it is:* Instead of a fixed heuristic fusion, use a **learned model to combine scores or features**. For example, train a logistic regression or small neural network that takes as input: dense similarity, BM25 score, file type (code/doc), query length, etc., and outputs a combined relevance score. This requires a training set of query-document relevance (from human labels or implicit feedback). The simplest form is learning a **convex weight** λ such that score = $\lambda \cdot \text{dense} + (1-\lambda)\cdot \text{BM25}$ maximizes some metric. More advanced, one can include interaction terms or even learn separate weights for code vs docs (e.g. effectively learning `code_boost` rather than setting it manually). **Calibration** is key: we may first transform scores to a common scale (e.g. convert BM25 to a 0-1 normalized range and same for dense) before feeding them to the model [27].
24. *Why it matters:* This promises the **best of both worlds** once sufficient data is present. Learned fusion can directly optimize for our target metric (say nDCG on a validation set). It can account for quirks: e.g., maybe BM25 is generally weaker except when query contains certain keywords, etc. A learned model can pick up on patterns like "if query contains `error:` token, give more weight to code search because it likely appears in code logs". It also future-proofs: if we change embedding models, we can recalibrate with a small dataset rather than guess weights. Ultimately, this can outperform static methods – Pinecone's research noted that a learned convex combo outperformed RRF in several scenarios [17].
25. *Evidence strength:* **Medium.** There's long history of learning-to-rank in IR, and hybrid search tuning is well-studied. Many systems report that even a tiny amount of training data (dozens of queries with judgments) suffices to significantly improve ranking by adjusting weights [17]. E.g., tuning a single parameter λ on a dev set can yield gains over untuned fusion. That said, evidence is only as strong as the data: if our labeled set is small or not representative, learned weights might overfit or degrade generality (the Pinecone study found convex combos needed only a small set to train, implying sample efficiency [17], but that's in a text domain – likely similar for us).
26. *Integration complexity:* **Medium.** Implementing the combination in code is trivial (once weights are decided). The challenge is **obtaining training data and performing the optimization**. We'd need to label perhaps a few hundred query results (especially focusing on code queries). Then use a script or even manual trial (grid search for λ) to find the best weighting. We might integrate this as part of `llmc.toml` config: e.g. `scoring.dense_weight = 0.7` learned from data, replacing the current heuristic `code_boost` logic. If using multiple features, we might embed a small model (which could be as simple as a linear model) inside the retrieval scoring – that's more complex to maintain (would need to load model weights, etc.). Probably we start with learning a few scalar weights that we then hard-code or put in config. This is essentially extending what LLMC already does with `code_boost` and `doc_penalty` but in a data-driven way.

27. *Failure modes:* If the training judgments are biased or sparse, the learned weights might be suboptimal. E.g., if most labeled queries were code-intent, the learned model might overweight code results and hurt docs queries (or vice versa). Also, a linear model may not capture all interactions – though our scenario is simple enough that linear might suffice. Another risk is **distribution shift**: if we tune weights on one repository or dataset, it might not transfer to another project's corpus characteristics. We might need per-repo calibration if they differ widely (perhaps mitigated by making the classifier depend on repository meta features, or simply accept some performance loss across repos).

28. *When NOT to use it:* Initially, when we have **no labels** (right now, that is the case), a guessed weight or RRF is safer. We should not prematurely "learn" from too little data. Also, if real-time adaptivity is needed (varied queries over time), a static learned weight might not always be optimal – but that's where dynamic gating and more advanced models come in later. So, we'd not use a blindly learned static weight if we detect it's failing on some query type (we'd either segment by query type or fall back to dynamic rules). Essentially, use learned fusion once we're confident in the training set's quality and representative coverage.

29. **LLM-based Listwise Reranking**

30. *What it is:* Utilizing a **Large Language Model to evaluate and reorder a set of retrieved candidates**. In practice, we take the top-k results (maybe after initial scoring or expansion) and form a prompt for an LLM: e.g., "You are a ranking assistant. The user asked: `<query>`. Here are 5 candidate answers (with snippets). Rank them from most to least relevant." The LLM then outputs an ordered list or scores. This is a *listwise* approach because the LLM sees all candidates and can directly compare them [11]. Variants: one can ask the LLM to just pick the best, or assign each a relevance grade, but listwise sorting yields a full ordering. We can use either the same local model powering LLMC (if it's capable enough) or a specialized reranker model (like *RankVicuna*, an open 7B model fine-tuned for ranking).
31. *Why it matters:* LLMs can capture subtle relevance signals that embedding similarity might miss – understanding of context, semantics, even user intent nuances. For instance, an LLM can judge that "Function `foo` doesn't appear to handle X based on its code, so it might not answer the query asking about X" – a subtle reasoning that pure similarity can't do. It can also leverage real language understanding to handle tricky queries (e.g., negations or multi-part questions). Listwise format ensures the LLM knows the context of other candidates, reducing randomness (it can explicitly compare and see which is more complete). This often leads to **improved ranking quality, especially on complex questions** [6]. Additionally, LLM rerankers bring *explainability*: they could provide a rationale (though in practice we might not use that to avoid hallucinations, see guardrails below).
32. *Evidence strength:* **Strong for quality, Medium for cost.** Research has shown GPT-4 zero-shot reranking can surpass even fine-tuned cross-encoders, improving nDCG by ~2-3 points on benchmarks [6]. The "Is ChatGPT Good at Search?" study found GPT-4 in listwise mode outperformed prior SOTA rankers on multiple datasets [6]. Moreover, RankVicuna (Pradeep et al. 2023) demonstrated that a 7B open model can achieve near GPT-3.5 performance in reranking [3]. This is promising for local use (since a 7B model can run on a single GPU). However, these results are on text benchmarks; for code/doc retrieval, we expect similar benefits as the LLM can reason about code relevance too (assuming it's been trained on code). The caveat is cost: running an LLM for every query adds latency and resource usage, which must be justified by the complexity of the query or the importance of getting the absolute best context.

33. *Integration complexity:* **Medium.** LLMC's architecture can accommodate an LLM rerank in the `rerank.py` stage. We'd likely implement a function to format the prompt with the query and the snippets (from `spans` content), call the local LLM (which might be Qwen-3 4B or similar loaded in memory), and parse its output. We need to ensure deterministic prompts for consistency (the prompt engineering is crucial: instruct it not to use outside knowledge, only the given text, and possibly to output just an ordered list or scores to avoid variability). We must also guard cost: e.g., only rerank top 5 or so, not 20, to keep prompt length manageable. Config: `llm_reranker.enable=true`, `llm_reranker.model=qwen-3`, `llm_reranker.max_candidates=5`. If we use a small model, we might need to fine-tune it for ranking (or use something like RankVicuna's weights). Another approach is pairwise prompting (multiple prompts comparing pairs), but listwise is more efficient albeit requiring fitting multiple docs in context. We should also run the LLM with temperature 0 for consistency.
34. *Failure modes:* The LLM might **hallucinate judgments** – e.g., it could favor a document that "sounds" more relevant but actually isn't, if its internal knowledge interferes. If not instructed well, it might leak answer information (like if it knows the answer from training data and then judges a doc that contains it implicitly as relevant even if the snippet didn't show it). There's also run-to-run variability if not using a deterministic setting. Performance-wise, a 4B LLM might simply not be very accurate at fine distinctions; if it's not as good as GPT-4, its ranking might sometimes degrade things. We have to carefully evaluate if the reranker is actually improving relevance or just reordering arbitrarily.

35. *When NOT to use it:* For **simple queries** where lexical or embedding signals clearly suffice (e.g. "error code 1234" – BM25 will find the exact line in code), an LLM rerank is overkill and only adds latency. We can gate the usage: perhaps only use LLM rerank if the top results' scores are close or if the initial retrieval has a certain uncertainty (like low confidence). Also, if latency budget is tight (e.g. user expects near-instant answers), we might disable LLM rerank except for hard queries. In an online setting, we could trigger it on a second try or when the user explicitly asks for more precision. Essentially, use it selectively to maximize ROI. A gating approach could be: if query is long/natural language or if initial retrieval has mix of code/docs that need sorting, then use LLM; if query is a straightforward keyword or identifier, skip LLM rerank.

36. **Lightweight Cross-Encoder Reranker**

37. *What it is:* Instead of a full LLM, use a smaller **cross-encoder model** (e.g. a MiniLM or mpnet-based model fine-tuned on QA retrieval) to score each (query, document) pair. This is a pointwise reranker that reads the query and a candidate's text (or code) and outputs a relevance score (often via a classification head or regression). For example, MonoT5 (a T5-based reranker) was popular: it re-phrases "Query: ... Document: ... Relevant?" and outputs a score. We could fine-tune or use an off-the-shelf model for code search if available (or even a sentence-transformer cross-encoder).
38. *Why it matters:* Cross-encoders often outperform bi-encoders (dense retrieval models) because they consider the full interaction between query and doc tokens, not just aggregate embeddings. They can catch if, say, the doc indeed answers the query or just mentions keywords. They're much faster to run than an LLM and deterministic. For LLMC, a small reranker could be run on the top 50 candidates in a few hundred milliseconds or less if optimized, which might be acceptable. It won't have the reasoning depth of an LLM but will reliably boost relevant items and demote false positives (like a code file that happened to share some tokens but isn't actually about the query). Think of it as a middle-ground: better scoring than raw embeddings, cheaper than GPT.

39. *Evidence strength:* **Strong (traditional IR).** Systems like BM25 -> cross-encoder (monoBERT, monoT5) have been the winning approach in many benchmarks [28]. They consistently add significant gains (often 5-10% absolute on metrics like nDCG) over first-stage retrieval. For code, Microsoft's CodeBERT and related models have cross-encoder versions that improved code search results in evaluations. The principle is well-established: pointwise learned rankers can effectively model relevance given training data. The only limitation: they require training (so we need data or to use a model pre-trained on similar data). If using a general cross-encoder (like one trained on MS MARCO), it might not understand code well unless fine-tuned. We might exploit our human judgments to fine-tune one for our domain.
40. *Integration complexity:* **Medium.** It involves adding another model to our pipeline. If using a pre-trained one from HuggingFace, we need to load it (which might be a few hundred MB) and run inference on each candidate. We'd integrate that likely in `rerank.py`, after initial scoring but before final fusion. Because it's pointwise, it's parallelizable: we can score each candidate independently (batching on GPU). Config: `cross_encoder.model_name = "...",` `cross_encoder.top_k = 20`. We'd need to ensure the text we pass (for code, maybe include some code context up to a limit, and for docs, the snippet). We may use the snippet content that LLMC already stored. One complexity: if a candidate is a long file, the cross-encoder can't take all tokens – we might use the most relevant span (LLMC's spans) or start of file. Possibly we feed the *extracted span* that triggered the retrieval (since LLMC indexes spans). That actually aligns well: we have short chunks (spans) that were retrieved, so a cross-encoder can score those chunks with the query.
41. *Failure modes:* If not well-trained on our data, it might give wrong preferences – e.g., focusing too much on superficial similarity. There's also a risk of **topical drift**: cross-encoders sometimes overly favor documents that are *query-similar* even if they don't contain the answer (they might pick up on query words but not truly answering context, especially if binary-trained on "does this doc contain an answer?" vs. graded relevance). Also, a cross-encoder might not be easily interpretable; if it makes a mistake, we might not know why. In terms of performance, applying it adds latency roughly proportional to number of candidates * model size. A 100M parameter model on 20 candidates might be fine (~20x forward passes), but on CPU that might be slow – better to use GPU if available.

42. *When NOT to use it:* If we have an LLM reranker active for a query, a cross-encoder may be redundant (LLM likely does as well or better). Also, if we lack any data to trust a cross-encoder, using a pre-trained one not tuned to code might be ineffective – possibly skip until fine-tuned. It might be unnecessary for queries where the initial rank is already almost perfect (simple look-ups). We could enable the cross-encoder only for queries classified as complex or when initial rank confidence is low. Another scenario: if the repository is small and retrieval is already high precision, a cross-encoder might be overkill. But in large repos with many similar functions, it can be quite helpful to pinpoint the right context.

43. **Intent-Aware Routing & Query Gating**

44. *What it is:* A mechanism to dynamically choose the retrieval strategy based on the query's **predicted intent**. Concretely for LLMC: a classifier (heuristic or learned) decides whether a query is asking for *code implementation* or *conceptual documentation*, or both, etc. Based on that, we set different weights or even choose different pipelines. E.g., if code-intent with high confidence, we might **only search code index or give it a heavy boost**; if docs-intent, stick primarily to docs index; if unsure or mixed, do multi-route with intermediate weights. This gating can also extend beyond code vs docs: e.g., route query about error messages to a logs/FAQ index if one exists, or queries about function

usage to example code vs definition. Essentially, treat "which expert should handle this query?" as a prediction problem. Over time, this can evolve into a learned mixture-of-experts as seen in RouterRetriever [18], but initially we can implement rules or a lightweight model.

45. *Why it matters:* The current static multi-route (with fixed weights like docs_primary route pulling some code at 1.0 weight) isn't optimal for every query. Some queries clearly require code, yet the system might still retrieve many docs (diluting precision). Others are clearly doc questions but code pops up because of overlapping terms. By gating properly, we improve precision by **targeting the right index**. It also saves computation; e.g., if a query is definitely code-related, we might skip running a heavy BM25 on all docs. Additionally, it helps avoid user frustration (no more "why am I seeing docs when I asked for code?" or vice versa). In effect, it **aligns retrieval with user intent**, which is fundamental for user satisfaction.
46. *Evidence strength:* **Medium.** Query intent classification is a standard component in enterprise search (like routing queries to product vs help center). Literature on **vertical selection** in search engines shows benefits of using query classifiers to route to different indexes (e.g., news vs web vs images). In our context, anecdotal evidence is clear (the problem statement itself notes code queries returning docs). RouterRetriever research provides evidence that choosing the right expert model per query yields better results than one-size-fits-all [18]. They improved multi-domain retrieval by a learned router that picks the right domain expert for each query, significantly boosting nDCG on BEIR benchmark [18]. This is analogous to us picking code vs docs expert. So while we may not have a paper specifically on code-vs-doc routing, general IR practice and that research suggest a solid benefit.
47. *Integration complexity:* **Low to Medium.** On the low end, we can enhance `llmc/routing/router.py` with more robust heuristics (maybe regex for common code question patterns like "implementation of", "where is X defined", detection of camelCase tokens suggesting code, etc.). Also, utilize features: if the user query contains backticks or looks like code snippet, route to code. LLMC already has some heuristics in `code_heuristics.py`; we can iterate on those easily (this is low-hanging fruit). Next, medium effort: train a simple classifier. For example, gather a dataset of queries labeled as code-intent or doc-intent. Train a small logistic regression or fine-tune a BERT on those (could even use the embedding model itself by feeding query text and looking at nearest neighbors categories as weak labels). Then integrate this model in router (maybe as a small PyTorch model or even a set of keywords weighted). The classifier can output a probability of code vs doc intent. Then we map that to route weights: e.g., if P(code)=0.9, set code route weight high. The config can allow enabling this learned intent detection (`enable_intent_model=true`). We'll also add logging to `search_spans` or router to record decisions and the confidence, for debugging and analytics (e.g., output: "Router: classified query as code-centric (0.85), applying code_boost=1.5"). The medium part is training the classifier – but we can start with heuristics and later replace with learned.
48. *Failure modes:* Misclassification is the obvious one. If we wrongly label a docs query as code, we might return code files which frustrate the user. If uncertain, the system should ideally hedge (e.g., still include some docs). There's also the nuance of **mixed intent** – e.g., "What does function X do and how to use it?" might need both the code (for what it does) and docs (for usage). A binary classifier might not capture that; we might need multi-label or a rule to allow multi-route when both probabilities are moderate. Another risk: if the classifier is too confident based on spurious cues (like any query containing "API" it might assume docs, even if sometimes code is needed). So continuous monitoring of misroutes is needed. Fortunately, since we will log decisions, we can manually review and adjust rules or retrain on errors.

49. *When NOT to use it:* If the router confidence is low (around 0.5), it might be safer to use multi-route blending rather than a hard route. So we should not hard-switch to single-route unless very sure. Additionally, in early deployment, if we haven't thoroughly tested the classifier, we might run it in shadow mode (deciding but not fully affecting results) to evaluate. But conceptually, some form of intent-aware routing should almost always improve things, as long as it's not overly aggressive. We wouldn't disable it entirely unless it proves unreliable – instead, we'd fallback to default multi-route when unsure rather than turning gating off. In summary, always *consider* intent, but don't overfit to it if confidence is not there.

50. **Human-in-the-Loop Relevance Feedback & Labeling**

51. *What it is:* Systematically involving human evaluators to label query results and using those labels to refine the system. This includes designing a **relevance rubric** for code/doc queries (e.g., 0–3 relevance scale defined clearly for each category of query), having humans annotate a set of query-result pairs, and analyzing these judgments to identify weaknesses (e.g., "docs often got a 2 when code should have been 3 – meaning code missing"). Additionally, using **preference feedback**: present two versions of context (say, old vs new retrieval) to internal evaluators or even end-users to choose which is better. The human feedback then guides adjustments – whether manually (tweaking config) or through model training (fine-tuning rerankers, training the fusion weight, etc.). In an ongoing loop, we also plan **active learning**: focusing labeling on queries where the system is uncertain or performing poorly. For instance, if the LLM reranker disagrees with the initial rank, that's a good query to get human judgment on, to see which is correct.
52. *Why it matters:* Human judgment is the **ground truth** that ultimately defines success. Especially in a code assistant context, automated metrics only go so far – we need to ensure that what we retrieve is actually helpful to users. Humans can catch subtle issues (like "this code is relevant but is from the wrong version of the API" – something models might miss). By having a solid evaluation set, we can tune our methods (like learned weights, thresholds) with confidence. Moreover, human feedback is essential for **alignment**: if the system's idea of relevance diverges from user needs, only human input can correct that. Also, in tricky scenarios (like a result that's technically relevant but too complex to be useful), humans can provide nuanced feedback that direct metrics (which might just consider it "contains answer = relevant") don't. Incorporating human preferences (which context leads to a better final answer) helps optimize not just retrieval metrics but actual end-to-end usefulness.
53. *Evidence strength:* **Strong (for evaluation), Medium (for direct improvement)**. The practice of human relevance evaluation is standard in IR (TREC competitions rely on it). Studies consistently show that optimizing to human-annotated relevance improves user satisfaction more than optimizing proxy metrics. On using preferences: the success of RLHF (reinforcement learning from human feedback) in aligning LLMs (like ChatGPT's training) is a big testament – models optimized on human preference can vastly improve quality. We can analogously train our rerankers or at least tune weights from preference data. Active learning has empirical support in reducing labeling effort while maximizing model improvement [29]. There's less direct literature on human-in-loop for retrieval augmented systems specifically, but generally teams building search systems (e.g. Google's raters, etc.) rely heavily on human judgments to refine algorithms. So this is more of a best practice than a hypothesis.
54. *Integration complexity:* **Medium.** This is more process than code, but we need to create the **infrastructure for collecting and applying feedback**. Steps: Define a clear labeling guideline (the rubric for code/doc relevance). Use a tool (even a spreadsheet, or a simple web app) to let

annotators (could be ourselves or hired labelers) rate results. We might store these in a JSON or database, and write evaluation scripts to compute metrics like nDCG. Integration to the pipeline is indirect: e.g., we might periodically retrain the intent classifier or adjust `code_boost` based on trends in the labeled data. If we incorporate preferences to train a reranker (like via pairwise learning-to-rank), that's an additional model training component. Possibly, we can loop the human feedback to an LLM for rapid iteration (like ask an LLM to summarize common errors from human labels). In terms of code changes: minimal – mainly in eval scripts and maybe a small interface to log when user provides explicit feedback.

55. *Failure modes:* **Labeling bias or inconsistency** – if rubrics aren't clear, different annotators may disagree (low inter-annotator agreement). This can lead to wrong conclusions (tuning to noise). We mitigate by training annotators with examples and measuring agreement (aim for at least moderate κ). Another issue: focusing too much on the labeled set can cause overfitting the system to those specific queries ("teaching to the test"). We ensure diversity in queries and continuously update the set. Also, human preferences can sometimes be short-sighted: a user might prefer a doc because it's easier to read even if the code had the answer. We have to interpret preferences carefully (maybe the correct solution is to retrieve code *and also* provide explanation, which is an answer-generation issue beyond retrieval). Lastly, collecting feedback takes time and money, so there's a risk of not having enough data – we should use it wisely for the most impactful adjustments (like training high-level parameters first, not trying to micro-optimize without sufficient data).

56. *When NOT to use it:* There's essentially no scenario where *ignoring* human feedback is a good idea if available. The more interesting question is how to use it. We shouldn't blindly trust a single user's click as a judgment of relevance (could be random or exploratory). So implicit feedback should be aggregated and treated carefully. But for explicit labeled data, we always use it for evaluation at least. Perhaps one caution: do not inject humans in the loop *at query time* for end-users (too slow) – our human-in-loop is for training/validation, not a production step. So aside from evaluation speed constraints, human oversight should always be considered a key part of the development cycle, especially for aligning to what *we* consider relevant in code assistant context (which might differ from generic IR judgments).

57. **Active Learning and Model Adaptation**

58. *What it is:* A strategy to **continuously improve models (like classifiers or rankers) using new data** gathered from usage and targeted labeling. Active learning involves selecting the most informative new examples to label – e.g., queries where the current model is least confident or frequently wrong. Over time, as the system is used, we can log difficult queries (like those with low-scoring results, or user dissatisfaction signals) and feed them into the next round of model training. This way, models (like the intent router or reranker) are not static; they adapt to the peculiarities of our domain and evolving usage patterns. This also covers **bandit approaches** for online fine-tuning: using implicit user clicks to slightly adjust ranking weights in real-time (with safeguards to not drift too far without confirmation).
59. *Why it matters:* Our system's optimal settings might change – e.g., if the codebase grows, or if users start asking different types of questions. Active learning ensures we don't have to label everything – just the most impactful cases. By focusing human effort on the "edges" where the model struggles, we get the most improvement per label. Online adaptation (with guardrails) could catch when, say, a particular boost is hurting more than helping as content changes. This leads to a system that **keeps getting better** and doesn't stagnate or degrade when conditions shift. It's essentially implementing a feedback loop at the algorithmic level, not just human oversight.

60. *Evidence strength:* **Medium.** Active learning is a well-researched area; for instance, in search, some studies show it can reduce labeling needs by ~30% for the same model accuracy [29]. In practice, teams do iterative training with new data rather than one-shot training. Contextual bandits (for search ranking) have had mixed success but are conceptually appealing for personalization and fine-tuning ranking weights on the fly. Given our moderate user base (likely not massive web traffic), a conservative approach is needed. But even periodically retraining models with new data (offline) has proven benefits – models improve significantly after a few iterations with new labeled examples that specifically target prior errors.
61. *Integration complexity:* **Medium.** We need to build some plumbing to capture data: e.g., log queries and results, along with any signals (did the user click a suggestion, copy code, etc.). Then a process to analyze those logs and perhaps choose a subset for labeling (which could be done by us reviewing them). Then update the models. Technically, retraining a classifier or regressor is straightforward once data is prepared. For bandit-style online tweaks, that's more complex: we might implement a simple mechanism where we maintain a small delta to e.g. code_boost that increments or decrements if we see sustained patterns of user behavior (but doing this robustly is tricky and might be deferred). Initially, the "active" loop might just be manual: each month, gather new tricky queries, label them, retrain the intent classifier, adjust any weights. Over time, we can automate parts (like a script to auto-select candidates for labeling). This requires moderate engineering but mostly on the data side (ensuring logs are stored and accessible for analysis).
62. *Failure modes:* One risk is **biased feedback** – maybe users rarely explicitly give feedback unless it's very bad, so an active strategy might only see those extreme cases and over-adjust. Or if using implicit signals (clicks), they might be misleading (maybe a user clicked a doc not because it was relevant but just checking; absence of click on code doesn't always mean it's irrelevant if the answer was in the doc snippet above). Bandit approaches might chase clicks in a way that doesn't always align with true relevance (e.g., a flashy doc title might get clicks over a dull code snippet, even if code had the needed info). Also, continuous adaptation can lead to instability if not monitored – changes in one iteration could degrade something else (thus always keep a baseline and evaluate on known labeled set when updating).

63. *When NOT to use it:* We should be cautious not to let the system "learn itself" into a bad state without oversight. So we wouldn't do fully autonomous online learning unless we have strong validation gates. In low-traffic scenarios, it might be fine to just periodically retrain offline rather than try constant online updates. If we notice the model thrashing (oscillating decisions due to noise in feedback), we'd dial back and require more stable data to make changes. Essentially, active learning is always useful offline; online bandit learning we would only deploy if we have enough interaction data and have tested it thoroughly in simulation. Initially, focus on active learning for labeling (choosing what to label next), which is low risk, and hold off on automated weight tweaking until comfortable.

64. **Rubric-Based Evaluation & Metrics for RAG**

    - *What it is:* Establishing **custom metrics and evaluation criteria** tailored to our use-case. This includes defining a graded relevance rubric for code/doc retrieval (as mentioned under human loop), and computing metrics like **Code@k** – e.g., percentage of code-intent queries where at least one relevant code snippet is in top k results (a recall measure for code). Also metrics like **nDCG** (Normalized Discounted Cumulative Gain) which uses graded relevance to reward getting highly relevant items ranked higher – this requires our rubric to assign, say, 3 for "exact code", 2 for "related doc", etc. We also consider **Context Precision/Recall**: out of the retrieved context given to the LLM, how much was actually needed or used to produce a correct answer [30]. Another RAG-specific metric is **Answer faithfulness** – did the LLM's answer only use the provided context? While not directly a retrieval metric, it's influenced by retrieval quality (bad retrieval -> hallucination). We might use a proxy like whether the final answer contained a citation to the retrieved snippet or whether humans judged the answer correct with evidence. Designing a metric like **CodePrecision@k** (of top k retrieved, what fraction are truly relevant code) can measure precision for code queries specifically, ensuring we're not just throwing code in but throwing the right code.

- *Why it matters:* Without good metrics, we can't be sure our changes are improvements. The standard IR metrics (Precision, Recall@k, MRR, nDCG) need appropriate definition of "relevance" in our context – which might be multi-dimensional (code vs doc difference). By having Code@k, we directly quantify the "doc dominance" issue and track progress on it. Graded metrics (nDCG) let us credit partial relevance: a doc might be somewhat useful even if not the ideal answer for a code query, so we give it a lower grade rather than binary 0/1. This nuance guides tuning (so we don't over-penalize including docs, just we prefer code a bit more). Context precision/recall goes a step further to measure the impact on answer quality – ultimately we care if the user got their answer correctly and easily. These metrics will show if adding graph info yields more complete context (higher context recall) or if adding more results starts to add irrelevant info (hurting context precision). Essentially, a comprehensive metric suite prevents optimizing one thing at the expense of another (e.g., we could always boost code and get 100% Code@3, but if those code snippets were not actually relevant, overall usefulness drops – which nDCG and context precision would catch).
- *Evidence strength:* **Strong (conceptually).** The RAG community is actively discussing metrics beyond just retrieval because the end goal is a correct answer. For instance, the RAGAS framework defines context precision/recall, answer correctness, etc., to evaluate RAG systems holistically [30]. Using such metrics has revealed cases where simply improving retrieval recall doesn't always improve answer quality if irrelevant info is introduced. By monitoring these, one can ensure alignment between retrieval and generation performance. Code@k is our custom metric addressing a specific problem; it's logically sound that if Code@3 goes up while relevance of those code results is maintained, users should benefit. We'll still verify with direct user study or answer eval. But historically, nDCG with a good rubric correlates well with user satisfaction in search. So we trust these metrics as necessary indicators (though they are only as good as our relevance labels, hence the rubric importance).
- *Integration complexity:* **Low.** This just requires implementing calculations in our eval scripts or possibly hooking into an existing library (maybe integrate with RAGAS metrics). We define the rubric (like: for code queries – code implementing asked feature = 3, test covering feature = 2, relevant doc = 2, related but not directly answering = 1, irrelevant = 0; for doc queries – correct doc = 3, code example = 2 if helps, etc.). Then we label some evaluation set accordingly. Then computing nDCG@k or Recall@k is straightforward. Code@k for, say, k=3 can be computed if we label each result as code or not and relevant or not. For context metrics, if we have a reference answer or at least knowledge of what was needed, we can tag which retrieved items were needed (context recall = fraction of needed pieces present) and which retrieved items were unnecessary (context precision = fraction of retrieved items that were actually used in answer). This might require separate annotation of which snippet answered the question. It's a bit of effort to set up annotation for context usage, but even a heuristic approach (was the answer text present in the snippet?) can approximate it. Alternatively, we measure final answer correctness (via human or automated QA) as the ultimate metric – and see how retrieval changes affect that. All these can be done offline with

our annotated data. Online, we might simplify to track something like user follow-up rate (if they often have to ask another question, maybe retrieval wasn't great). But offline metrics suffice for most tuning.

    - *Failure modes:* Metrics can be misinterpreted or mis-optimized if not designed carefully. For example, Code@k going up is only good if those code results are indeed relevant; if not, an unwise algorithm could game the metric by surfacing any code (which a poor labeling schema might count as success if we only cared about type). We avoid that by coupling type with relevance in the rubric (a code file that doesn't answer is still irrelevant). Another hazard: optimizing for context recall might encourage retrieving too many items, hurting readability – we need to balance precision and recall. That's why we will always look at multiple metrics (e.g., ensure context precision doesn't plummet while context recall improves). Overfitting to a specific metric is also possible (we won't, say, tune parameters purely to maximize one metric on a small eval set without cross-validation or further sanity checks). Essentially, the failure is not in using metrics but in using them blindly. We will mitigate by using a diverse set (as listed) and preferring improvements that show up consistently across them (or at least don't degrade one drastically for a gain in another, unless that trade-off is deliberate and acceptable).
    - *When NOT to use it:* We should always use metrics for evaluation, but we shouldn't restrict ourselves to only these if new insights come. For instance, if we find a certain failure mode not captured by current metrics, we might introduce a new metric or at least track that scenario separately. One could argue that ultimately user satisfaction is the metric; since we can't measure that directly all the time, these proxies are the best we have. So, no scenario to not use metrics – just ensure we're using the right ones. Maybe one caution: do not drive the **user interface** by these metrics in a vacuum (e.g., if Code@k is low but maybe users don't actually need code for those queries – context matters). Always interpret metrics in context of the actual queries and outcomes, not in isolation.

Each of these techniques offers a piece of the overall solution. **Summary of fit:** Graph expansions (Technique 1) and graph reranking (2) directly leverage LLMC's graph artifact to fix the code retrieval gap. Rank fusion (3) and learned scoring (4) ensure our multi-modal retrieval is robust and tunable. LLM rerankers (5) and cross-encoders (6) boost quality of top results, especially on complex queries, using advanced language understanding. Intent routing (7) tackles the specific docs-vs-code routing problem head-on. Human feedback (8) through labeling is our compass to ensure all these changes truly help users, and active learning (9) means we won't stagnate or mis-calibrate. Finally, a good metrics framework (10) lets us quantify improvements and catch regressions scientifically.

Importantly, **when not to use** or when to be cautious is noted for each – e.g., avoid graph expansion if it adds noise, or skip LLM rerank on trivial queries – to make sure we apply the right tools at the right times. This multi-faceted toolkit gives us flexibility: we have "cheap" options (like RRF or heuristics) and "expensive" ones (like LLM rerank or GNN) that we can deploy selectively. Next, we detail how these pieces come together in a cohesive scoring architecture for LLMC.

Proposed Scoring Architecture (Blueprint)

In this section, we outline a concrete blueprint for an upgraded LLMC retrieval scoring system, aligning each component with LLMC's existing modules and data structures. The design is modular, so features can be toggled or tuned via config. We break it down into the major pipeline stages:

4.1 Candidate Generation (Retrieval Stage 1)

Multi-Route Lexical + Dense Retrieval:
We will continue to use dual indices: one for code (`emb_code` table) and one for docs (`embeddings` table), as LLMC does currently. On a query, *both* routes are executed (if enabled) – this yields two lists of candidates with embedding similarity scores. Additionally, we introduce a **lexical retrieval** component using BM25 (or even simple keyword matching on the text and code). Concretely: - Use the existing `deterministic_router` (modified with intent gating, see 4.5) to determine if both routes should be used. In most cases, we'll use both with weights, except maybe extreme confidence in single-route. - The dense retrieval is done as now (e.g. FAISS or SQLite VSS query for top N from each index). - In parallel, perform a BM25 search on the combined corpus or separately on code and docs. We can leverage an existing library (perhaps Whoosh or just a simple TF-IDF in memory since data size is manageable). For efficiency, we might restrict BM25 to top M (like 100) and then it will be fused later. This BM25 is particularly useful for catching exact matches (error codes, function names, etc.). - Optionally, use the query to search *multiple fields*: e.g. code search could also look at file names (since LLMC's scorer already has filename matching boosts). We will incorporate that as part of scoring rather than separate retrieval (filename matching can be a feature). - **Why both?** Dense retrieval is great for semantic matches but can miss rare tokens, while BM25 will catch those exact token matches even if semantics differ [31]. Combining them improves recall by 15–30% in hybrid systems [32].

Graph-Based Query Expansion (Pre-retrieval):
Before executing searches, we can apply any trivial query rewrites. For example, if the query is identified as code-intent and contains a CamelCase identifier, we might automatically also search for it split by underscores or with different casing (to catch cases where code tokenization differs). If the query is documentation-style but mentions a class name, we might add the class name as a keyword to ensure code is considered. These are minor expansions to help retrieval catch variants. We will not do heavy LLM-based query expansion initially (to avoid query drift) [9], but keep the door open. For instance, if the query is very short or ambiguous, an LLM (like GPT-3.5) could be prompted in the background to suggest additional keywords or synonyms, which we then include in a second round retrieval (that is a PRF-like strategy). This is optional and can be config-controlled (`query_expansion.enable`). Initially, focus on straightforward expansions like synonyms from a glossary or known alias (like "config" vs "configuration file").

Route-specific filtering:
LLMC already maps file types to routes (via `routing.slice_type_to_route`). We will ensure that when retrieving from code index, we restrict to code file types (maybe based on extension), and similarly docs route. This was likely already done, but we'll double-check config `code_extensions` vs `doc_extensions`. This avoids irrelevant cross-type results at source.

Fan-out and result pooling:
After retrieval, we will have possibly three lists: Code-dense, Doc-dense, Lexical (mixed). We **pool** these for the next stage. Pooling can be done by simply concatenating with tags or by an initial fusion: - We propose to do an initial fusion of dense+lexical via rank fusion or learned weight (the hybrid approach). For now, RRF is simplest: combine code-dense, doc-dense, and lexical lists with RRF to produce one unified candidate list with a fused score [16]. This ensures no good candidate from any source is lost. - Alternatively, we keep them separate a bit longer and let the reranker handle it. But a unified list with some initial fusion score is fine. We will include in that score also any prior boosts (like file name match bonus or extension boost) from `Scorer`. - The **max number of candidates** to carry forward can be config

(`search.top_k_candidates`). Perhaps ~50 total is a good starting point (experiments can tune this). That ensures the reranking stage has enough to find the gems but not too many to slow down.

LLMC Fit: - The changes happen in `llmc/rag/search/search_spans()`. We integrate BM25 by calling a function (possibly implemented in `llmc/rag/lexical_search.py`) if enabled. - We adjust how `fusion.py::fuse_scores` works: if `fusion.mode="rrf"`, do RRF combination of the multiple lists. We provide `route_weights` as config (perhaps learned later, but initially can keep 1.0 for both if using RRF). - The router logic is extended (in `router.py`) to output not just one route, but potentially triggers for lexical search as well (or we just always do lexical as a supplement if enabled, as it's cheap). Possibly treat lexical as another "route" with weight. For simplicity, treat BM25 as another retrieval that always contributes (like a multi-route where one is "text" using BM25). - We will log in verbose mode: e.g., "Retrieved 50 candidates (20 code, 20 docs, 10 from lexical overlap)." and possibly details for debugging.

4.2 Graph Augmentation of Candidates

Once we have the initial candidate list, we apply **graph-based augmentation**: - **Neighbor Inclusion:** For each candidate in the top N (say top 5) of the fused list, retrieve its neighbors from the repository graph. The graph (`rag_graph.json`) likely contains edges like: - `calls` or `imports`: code file A calls B. - `defines`: doc D references function in code C. - `tests`: test T is linked to code file X. - We load these via `graph_index.where_used_files(file)` for usage, and `lineage_files(file)` for perhaps parent references, etc. For each neighbor not already in the candidate set, we add it with a base score. - **Scoring Neighbors:** How to score added nodes? One approach: use a **decay factor** relative to the source node's score. E.g., if doc D with score 0.8 is in top 5 and code file C is linked from D, we add C with score 0.8 * α (where α maybe 0.5 as a config). If multiple top nodes link to the same neighbor, we could boost it more (maybe sum contributions). This is akin to one step of spreading activation on the graph [8]. We will cap the score so that a neighbor doesn't outrank its sources unless it also had some direct retrieval score of its own (if it did, it'd already be in the set). Essentially, treat this as "bring it to attention" rather than fully determine rank. - We will differentiate edge types for possible filtering: e.g., if a doc links to 5 code files, maybe only add the one most relevant by context (if graph stores contextual similarity or we could quickly check embedding similarity of the neighbor to query as a secondary filter). - **Graph Schema Expectations:** The `.rag_graph.json` should represent a graph of files/nodes. We expect nodes identified by some key (file path or ID) and edges listing relations. We should clarify its schema (perhaps it's an array of edges or an adjacency list mapping). We'll design our augmentation to be flexible: e.g., have config to choose which relations to use (maybe `[graph.use_relations] = ["calls","defines","tests"]`). Then in augmentation, we consider those edges. This way, if the graph contains extraneous relations (like "similar content"), we could include or exclude. We also ensure not to go beyond 1-hop at this stage (multi-hop traversal can explode results). - **Graph Traversal Option:** In advanced config, we could allow limited multi-hop retrieval: e.g., if query explicitly asks a multi-hop question, we might do BFS up to 2 hops. But likely unnecessary for code (most answers found in 1 hop). - By including neighbors, we handle the scenario: user asks "Where is function Foo called?", dense retrieval might get the definition of Foo; our graph augmentation would find all usage locations and include those code references. Conversely, ask "How to do X per docs?", if a code snippet is found, we might include the doc that references that snippet.

Graph-Based Scoring Adjustments:
Even for candidates already in the list, graph info can adjust their scores: - If a code file and a doc are connected and both retrieved, we might want to bump the code file's score if the doc had high score. This is like a mutual reinforcement. Implementation: after initial scoring, for each pair of candidates that are

connected, add a small bonus to each (or to one type, e.g., boost code if it has a doc neighbor in top results). This could be configured as `scoring.graph_neighbor_boost = 0.1` etc. - If a candidate has many incoming links (degree centrality), we might incorporate a prior score for that. E.g., `score += β * log(1 + degree(node))`. This accounts for the intuition that a function used widely might be more important (though not always relevant to query – careful!). We might specifically use centrality if we are dealing with ambiguous short queries (like if user just says "authorization" – maybe boost the central "auth.py" module). - We can also *penalize* isolated nodes if desired (opposite of above, but probably not needed unless noise). - All these adjustments happen in `Scorer` (we can extend `scoring.py`). We have file-type specific boosts already; we can add e.g. `if enable_graph_priors: score += code_degree[file]*beta`. We would compute degree from the graph index on load and store in a dict for quick lookup.

Avoid Graph-Noise Amplification:
We set safeguards: - Only augment for top results that have a reasonably high initial score. If something barely made it to rank 20 with a low score, we avoid using it to pull neighbors (to prevent a chain of marginal relevances). - Limit number of neighbors added per source (e.g., if a doc links to 10 files, maybe add at most top 2 by some criterion, or just skip if too many). - Possibly require some semantic similarity filter: we could compute the embedding of the query and of the neighbor (embedding likely in our index anyway) – if neighbor's embedding isn't somewhat similar to query, skip adding it. This ensures we don't add completely off-topic neighbors. E.g., doc page referencing a common utility might not be relevant to the question asked. - If graph data is outdated (we should ensure the graph is regenerated when code updates), we might risk broken links. Ideally, `rag_graph.json` is up-to-date with the indexed content. We can double-check node existence when adding neighbors (skip if neighbor file isn't in index, or has been filtered out by route – e.g., don't add a binary file etc.).

LLMC Fit: - Primary code in `search_spans` after retrieval: incorporate a call to a function `graph_expand_candidates(initial_candidates)` that returns augmented list. - Use existing `graph_index` API from `llmc/rag/graph_index.py`. The methods `where_used_files` and `lineage_files` likely return lists of file paths. We will call those and interpret results as neighbors. - Possibly extend the graph loading to include edge types. If needed, we can adjust `graph_store.py` or how the JSON is read to capture relation labels. - Config additions: under `[graph]` e.g. `enable_expansion=true`, `neighbor_score_factor=0.5`, `max_neighbors_per_node=3`. Also `use_relations=["calls","defines"]` to control which edges count. - If `rag_graph.json` is missing or not configured, the system should still run (just skip augmentation). So our code should detect graph availability and only execute if present.

4.3 Reranking Stage (Second-Stage Ranking)

At this point, we have a candidate list (perhaps 30–50 items) with initial fused scores (embedding/BM25 and graph adjustments). Now we apply more expensive rerankers to fine-tune the ordering.

Efficient Cross-Encoder Reranker:
We introduce a small cross-attention model (like MiniLM or mpnet, possibly fine-tuned on MS MARCO or code search data). It will score each candidate's content against the query: - We feed the query and the candidate snippet (the text of the span or code chunk) into the model. For code, if the candidate is code, we might also feed some surrounding context or docstring if available. - The model outputs a relevance score (logit). We normalize these (maybe using sigmoid to [0,1]). - This is done for top K_x candidates (maybe

20). That K_x can be configured or adaptive (e.g., if we have 50 candidates, maybe cross-encode 30). - The cross-encoder scores will then replace or augment the previous score. E.g., we could set a new score = cross_enc_score * w + old_score * (1-w). If the cross-encoder is well-trained, we might trust it entirely and sort by its score; but combining might be safer if it's not perfect. We will experiment on dev set to see. - If we fine-tune this model on our labeled data, great. If not, we might use an off-the-shelf. For instance, `microsoft/MiniLM-L12-H384` trained on MS MARCO (there are such models for passage ranking). - Integration: implement this in `llmc/rag/rerank.py` as a new reranker class. Possibly behind `enable_cross_encoder` flag. - We will run it once per query, so performance depends on model size * candidate count. A small model on CPU might do ~50 ms for 20 candidates, which could be okay. On GPU, even faster. - If no GPU, we might consider running it half precision or some optimization.

LLM Listwise Reranker (Optional High-Quality path):
For queries where we need maximum precision or when the above steps are inconclusive (e.g., many candidates of similar score), we can invoke an LLM (like our local Qwen 4B or an optional call to GPT-4 if allowed, but since local-first is key, we focus on local). - This would involve constructing a prompt with the query and, say, the top 5 candidates after cross-encoder rerank. We might include short snippets from each (few lines or summary if long). - The prompt might be:

```
You are an expert code assistant. Rank the following answers to the question:
"<query>".
Documents:
[1] <snippet of candidate1>
[2] <snippet of candidate2>
...
Output format: an ordered list of document numbers from most relevant to least
relevant.
```

- We run the model with temperature 0 to get a deterministic rank. Parse it. - Then reorder those top 5 accordingly. We could leave lower-ranked ones as is or not include them to the LLM at all. So essentially, LLM final say on the top few. - If we want, we could instead have it assign a score to each ("relevance: X") and use that, but sorting is easier and yields relative judgements. - We absolutely include guardrails in the prompt: instruct it to only use given info, not outside knowledge (to avoid it ranking based on hallucinated info). - Possibly use something like "If any document contains the direct answer or code in question, that should be top. If a document is not relevant, it should be last." to shape its criteria. - We will design prompt templates and test them internally with some known cases. - Integration: This happens in `rerank.py` after cross-encoder. Use an LLM client (or local model loaded via huggingface/transformers or similar). - Config: `llm_rerank.enable` and maybe a threshold on when to use (like `min_query_length` or `use_for_queries_matching_pattern`). - For performance reasons, we likely don't do this for every query by default. Perhaps only on queries that are long/formulated (which often indicate complexity), or perhaps an interactive approach (LLMC could retrieve context, see if answer was found; if not, maybe then use an LLM to improve context – though that blurs into generation stage). - Given local model is smaller, it might be fine though. We can benchmark Qwen 4B or Vicuna 7B on ~2000 token input (query+5*snippets); if it's like 5 seconds, that might be borderline but maybe acceptable for hard queries.

Anti-Hallucination & Calibration in Reranking:
We implement some rules when using LLM: - For example, a "rationale-free" mode: we specifically prompt

for just the final ranking without a verbose explanation (explanations can sometimes include content from the documents which could inadvertently confirm biases). - To ensure consistent output, we might use few-shot prompting: e.g., give an example with dummy documents and correct ranking to set the pattern. - We also double-check the LLM output. If it does something unexpected (like returns a doc number not in the list, or ties), we have fallback: e.g., default to cross-encoder ranking for safety. - If the LLM seems to strongly contradict the cross-encoder (e.g., it ranks something very low that cross-encoder had high), that's fine—it's possibly correct (maybe cross-encoder fell for a trap word). But if we notice systematically that the LLM makes odd decisions on easy cases, we restrict its use or improve the prompt.

Combining Rerankers:
We might end up with multiple scores: original fused score, cross-encoder score, LLM ordering. We need a final decision: - Likely, if LLM is applied, we take its ordering for the top chunk and then append others after. If not applied, use cross-encoder's sorted order. - Cross-encoder vs. original: we will predominantly trust cross-encoder for ordering among what it scored. But maybe keep original scores for those it didn't score (positions beyond K_x). - We should also propagate any *grouping* constraints: e.g., if the same file appears multiple times (different spans) in candidates, we might want to group or choose the best span to avoid redundancy. Possibly as a post-process: if top 3 results are from the same file, that might not be optimal to present. We can merge them or pick one (this might be more relevant for answer presentation than retrieval per se, but worth noting). - The multi-stage nature is: initial rank by similarity -> adjustments by graph -> reorder by cross-encoder -> optionally reorder by LLM. Each stage is designed to improve fidelity, so we normally replace the prior ranking with the new one (rather than combine scores) except where specified (like combining cross-encoder and original might be weighted if cross-encoder is not fully trusted).

LLMC Fit: - The cross-encoder model can be integrated via `transformers` library loaded at startup of LLMC (perhaps in rag context initialization). We might add a lazy loader in `rag/__init__.py` for the model if enabled. - The LLM (like Qwen 4B) might already be loaded as part of the main LLM for answering. If so, we can call it directly (maybe via an internal API or just through a prompt to itself). However, careful: using the same LLM that's handling the conversation to do rerank might interfere with its state. Ideally, we load a separate instance or use the same model but ensure context is separate (like a system prompt to do the ranking as an isolated task). If that's complicated, we might instead fine-tune a separate instance specifically for rerank (like Vicuna 7B as we mentioned). We'll specify either using the main model in a controlled way or using a dedicated model. - Code changes in `rerank.py`: add methods `apply_crossencoder()` and `apply_llm_rerank()`. Ensure we handle token limits (if a snippet is too long, truncate it smartly – e.g., prioritize first lines of a doc or function signature). - Also integrate with `scoring.py::Scorer` if needed to incorporate crossencoder outputs (though likely easier to do separate). - All new knobs go into config: e.g., `[rerank] enable_crossencoder, crossencoder_model="name", enable_llm=true, llm_model="path-or-name", llm_candidates=5, ...`.

4.4 Fusion and Final Scoring

After reranking, we need to produce the final ordered list of contexts to return (and possibly their scores for the LLM to decide how much to show, etc.). Two fusion modes will be provided:

Rank-based Fusion (RRF or similar):
We maintain an option to use RRF as the merging strategy if we do multi-route without heavy rerankers. Actually, this would have been applied in stage 4.1 as described. But at the final stage, if for some reason

we still have separate streams (like if someone turned off cross-encoder and wants to just merge code/doc lists by rank), RRF is the go-to. It's simple and effective in absence of training [17] . This mode can be selected via config (`fusion.mode = 'rrf'`), and then even after rerank (if rerank is off), it ensures combined results.

Calibrated/Weighted Fusion (learned):
We implement a mode where fusion uses a learned weight or score scaling: - E.g., `fusion.mode = 'weighted'` with `fusion.weight_code = 1.2` and `fusion.weight_docs = 1.0` (just as an example). Then in `fuse_scores`, we would multiply scores from code index by 1.2 before merging. This effectively is a static learned fusion. We can fill these from our training experiments (e.g., if we find best λ for dense vs lexical). - Additionally, we consider **score normalization**: e.g., convert cosine similarities to a 0–100 range by mapping the top result to 100, etc., before applying weights, to reduce sensitivity to absolute values. Or use something like *Reciprocal rank normalized scores* if needed. - If we pursue a learned model approach (like a linear model that uses multiple features), that may not be easy to implement purely in config. Instead, we might precompute the combination externally and just set a weight as above. For instance, if dense vs BM25 weight is 70/30, that can be done by scaling one and summing. - This mode is useful once we have a stable known good combination; RRF is fallback if not.

Duplicate Handling & Consolidation:
If the same file appears multiple times (due to multiple span entries in the DB), LLMC currently uses max score. We will keep that logic: in `fuse_scores`, if a file is already in results, take the max score (or perhaps sum – but max is fine to avoid doubling influence). We will maintain that for fairness. - But one improvement: if two top spans from same file are both high, after final selection we might merge them into one for output, but that's more a presentation detail. Possibly we let the answer generator see both if they are far apart in file.

When to use which fusion:
- If no training data or uncertain environment, RRF (rank fusion) is default as it's robust to scaling issues [16] .
- When we have calibrated weights from some tuning, use weighted mode for slightly better performance. - In case we incorporate more than 2 sources (say code dense, doc dense, BM25, graph suggestions as separate), RRF can handle multiple easily, whereas learned weights for 3+ might need more data. - We can also support a "hybrid" approach: e.g., do learned fusion of dense+BM25 for each route, then RRF across routes; but this might be overkill.

LLMC Fit: - `llmc/routing/fusion.py::fuse_scores` will be modified to handle these modes. Possibly refactor to a strategy pattern: if mode=='rrf' do rrf; if 'weighted' do weighted sum; if 'max' (current) do current behavior for backward compat. - Add config in `llmc.toml` [fusion] section for mode and any parameters (k for RRF if we allow customizing it, default 60 as often used; weights for weighted mode). - Confirm the interface from search_spans uses fuse_scores appropriately with route weight (we might override route weights usage depending on mode). - Test that duplicates (same span) merging still works in all modes.

Note on presenting results to LLM:
Even though final fusion gives us an order, we might still hand off multiple top contexts to the answer generator. So the order matters in terms of which ones we include or prioritize if there's a token limit. Likely LLMC will take top few and maybe drop low ones if too many tokens. So getting the order right is crucial; beyond that, the actual numeric score might not be needed except maybe to decide cut-off (like don't

include contexts below a certain score gap). We could set a rule like include contexts until cumulative coverage or until a big score drop, etc. But that's beyond retrieval – more on answer assembly. We can incorporate a simple cut: e.g., if there's a sharp drop in scores after rank 3, stop there to avoid adding marginal context (improves precision). This can be derived from our graded relevance (if we see after certain rank everything is low relevant, no need to feed to LLM). Possibly implement as: if score at rank k < X% of score at rank1, don't include beyond k.

4.5 Dynamic Scoring & Routing (Intent Gating and Adaptation)

This component wraps around the whole retrieval pipeline, making query-specific decisions:

Heuristic Gating (Phase 1):
Initially, implement improved heuristics: - In `router.py`, before doing anything, run through a list of checks: - If query matches a "code query" pattern (contains words like "implement", "source code", or matches a regex like `function[\w,]*` indicating code syntax, etc.), then set route weights: code route weight = 1.0 (maybe even exclusively code) and docs route weight = 0.2 (just in case). - If query contains triple-backticks or looks like it pasted code snippet with a question, route to code (likely debugging scenario). - If query contains words like "documentation", "guide", "explain how", likely docs-intent: weight docs route higher (1.0 docs, 0.5 code). - If query is just a symbol name or file name, that's often code (looking for definition), so code primary. - We also use file extension hints: e.g., if user mentions `.json` or config file, maybe route to config docs if we have them. - Many of these exist; we refine thresholds and maybe add new ones discovered from user logs. - Implement a config `[routing.heuristics]` where specific keywords or regex can be listed for code vs docs (so it's easily adjustable without code changes).

Learned Classifier (Phase 2):
Once we accumulate some labeled data of query->intent, we train a classifier: - Possibly a small BERT fine-tuned on our queries (if we have, say, a few hundred examples). If not enough data, even a logistic regression on bag-of-words might do surprisingly okay given obvious tokens. - This classifier could have classes: {code, docs, both}. Or maybe a probability continuous that we threshold. - We integrate it by loading the model at startup (could use a lightweight library or our own inference). - Use it to get P(code). If P > 0.7, treat as code-intent, if <0.3 as doc-intent, else mixed. - These thresholds can be tuned to ensure precision (we don't want false code intents, so maybe high threshold). - The classifier can also output other categories if needed (like maybe detect if it's a configuration question vs test question in the future to route to specific sub-index or apply test penalties differently). - Logging: we output in debug log: "Intent classifier: Code probability 0.85 -> routing to code index primarily."

Mixture Routing (Parallel or Weighted):
If intent is uncertain, we use multi-route with weights (as currently but dynamic). If very certain, we might still retrieve from both but heavily down-weight the other route, or possibly skip the other route entirely to reduce noise. We likely won't skip entirely (except maybe in extreme obvious cases) because of the possibility of a little relevant info in docs even for code questions, but we might if we are confident to save compute (like don't query docs index for every obviously code question). - Multi-route fan-out is configured in `llmc.toml` (like code_primary fan-out to docs with weight 0.5). We will programmatically override those weights per query according to classifier output rather than static. - Implementation: after classifier/heuristics decide weights, pass them to `search_spans` which currently uses `route_weights` to combine scores. Or we may directly apply them when merging results. Actually, `fusion.py` uses route weights to scale scores. That's fine – we can feed in dynamic weights to it. - The router might currently just

pick one route; we need to extend it to handle “deterministic router” giving multiple routes. Possibly simpler: always retrieve both but multiply scores from one route by the weight (this is how multi-route fusion is done by weight). - If classifier says code:docs weight 1.0:0.2, we multiply all doc scores by 0.2 before fusion (or equivalently treat that as route weight). - If classifier says both equal, then normal (1.0,1.0). - Possibly incorporate other dynamic factors: like query length – longer queries might need more docs context as code search on a long natural language query may fail, so lean docs for those. Short query that is a single token – likely code symbol, lean code. - Also, if initial retrieval from one side yields nothing useful (like no code hits above a score threshold), maybe automatically fall back to the other route fully. E.g., if code route found nothing with similarity > 0.3, but docs had some, then the query might not match code well, so rely on docs.

Calibration & Confidence in Routing:
We plan to calibrate the intent model scores. For example, if using logistic regression, ensure the probabilities reflect reality (maybe Platt scaling on a validation set). This prevents overconfident gating. If uncertain (like 0.55 vs 0.45), we treat as mixed. - We can implement a small temperature or sigmoid adjustment in config if needed (like `intent_threshold_high=0.8, low=0.2`). - Also calibrate across embedding: sometimes the content of results could inform intent (if top doc results all have very low scores but code have a decent match, that might indicate it was a code query that docs couldn’t answer). We can consider a second-pass: check retrieved results, and if the initially favored route yielded poor results, adjust. This is a bit complex (like a feedback loop), but could be done. It’s similar to slide-adaptive retrieval in the LLM listwise paper [22]: they use reranker feedback to decide to fetch more. We could do: if after initial retrieval top docs are low-similarity but there’s a moderately high-sim code result down the list, maybe upweight code more and re-fuse or even re-query code with broader search. This might be a later improvement.

Logging and Debugging Gating:
We’ll add verbose logs for each query: which route(s) were used, what weights, and why (heuristic matched X, or classifier output Y). Possibly add this info to the output context for debugging (maybe not to user, but to a dev mode). This is vital to build trust in the gating.

LLMC Fit: - Modify `router.py` in `DeterministicRouter.route(query)` to incorporate our logic. It might currently return a route name or list – we will adapt it to produce a structured output that includes recommended weights for each route. - `search_spans` will use those weights when calling `fuse_scores`. - Possibly maintain backward compatibility: if gating disabled, it works as before from config. - Config: `[routing] method = "learned"` vs "heuristic". If learned, we load the classifier (point to a model path perhaps). - Classifier can be implemented as a small function call (if logistic, could even inline weights formula from config if we don’t want a heavy dependency). - For learned approach, we might have to include an inference library or we can pre-generate a list of keywords with weights from logistic regression (like essentially a fancy weighted keyword match – that could approximate a trained model). But better to just load a tiny model if possible.

Fail-safe and Fallbacks:
If for some reason the classifier is unsure or fails, we fallback to default multi-route with equal weights (as LLMC does by default, or maybe slight bias to docs as safe side if unknown). - If an extreme gating returns zero results on main route (like we only searched code and found nothing), we could then do a second attempt searching docs – basically avoid zero-results situations by broadening query or route if initial attempt fails. That ensures recall.

By implementing dynamic routing and scoring adjustments per query, we get a system that can be “biased” when it needs to (to overcome the code vs doc imbalance) yet remain neutral or broad when uncertainty exists. This gating combined with the rest of the pipeline addresses the core problem (the wrong index dominating due to fixed weights).

With the blueprint above: - **Candidate generation** ensures a strong initial recall (dense + lexical). - **Graph augmentation** injects crucial missing pieces (connecting code with docs). - **Reranking** (cross-encoder + LLM) polishes the ranking to maximize relevance. - **Fusion** yields a coherent final list combining modalities fairly. - **Dynamic gating** ensures all the scoring behaves optimally per query category.

Each component maps to specific LLMC modules and can be developed and tested in isolation, then tuned to work together.

Now we move to the practical plan of implementing these improvements in phases, plus how we will evaluate and monitor them.

Implementation Plan (Engineering-Ready)

To implement the above architecture in LLMC, we propose a phased approach with clear milestones, balancing quick wins and longer-term investments. Below we outline the phases, tasks, and estimated effort, referencing the relevant modules and config keys for each.

Phase 0: Baseline Evaluation Setup (1 week)
Goal: Establish current baseline metrics and gather data to guide improvements.
- **Tasks:** - Define a set of representative queries (perhaps ~50, covering code-intent, docs-intent, ambiguous). Possibly include known problematic ones from user feedback. - Get current retrieval results for these queries using LLMC’s default (enable_multi_route with its default weights). - Have team members label these results with relevance (according to a draft rubric). This gives us an initial gold set. - Compute baseline metrics (MRR, nDCG, and specifically measure how often code file was ranked top for code queries). - Set up the evaluation harness: use `llmc/rag/eval/routing_eval.py` as a starting point, extend it to calculate our metrics like Code@3, etc. - **Outcome:** A baseline report (how many queries are failing due to doc dominance, etc.), which will later validate improvements. - **Engineering:** Mostly scripting outside the core code. Ensure `routing_eval.py` can accept different config toggles to simulate our upcoming changes. No production changes yet, except maybe minor logging improvements if needed to capture details.

Phase 1: Quick Wins (2–3 weeks)
Goal: Address the immediate “docs before code” issue with minimal risk changes and introduce better fusion.
- **Task 1.1: Reciprocal Rank Fusion Implementation**
- Modify `llmc/routing/fusion.py::fuse_scores` to add RRF. Actually, we create a new function or parameter. - Config: Add `fusion.mode = "rrf"` or `"max"` (default). If "rrf", apply formula: for each route’s result list, use score = 1/(k + rank). Use k=60 (configurable). - This is straightforward. Test it on a known case: e.g., if code route had a relevant item at rank 1 in code list but rank ~5 in combined originally, ensure with RRF it comes up near top. - **Modules:** `fusion.py`. Also update docs to describe new config in `llmc.toml.example` if exists. - **Estimate:** 1-2 days (including unit test). - **Task 1.2: Heuristic Intent

Routing Improvements
- Expand `llmc/routing/code_heuristics.py` with more patterns. E.g., detect “implement” or presence of `()`. Fine-tune thresholds or list of code keywords. - Add similar for docs keywords (maybe in a new `docs_heuristics.py`). - Integrate these in `DeterministicRouter`: e.g., if code heuristic triggers strongly, choose code_primary route (which via config already sets multi-route with code heavy). If docs triggers, choose docs_primary route. - Actually, simpler: adjust config weights: currently `[routing.multi_route.docs_primary]` fans out to code with weight 1.0. We might reduce that to 0.5 by default (since the problem is doc queries pulling code too strongly). Conversely, for code_primary, ensure docs weight is modest (0.3-0.5). - This can be done in `llmc.toml`. But to be dynamic, maybe leave config as is and rely on router logic to override weights in code vs docs cases. Possibly easier now to just adjust config defaults for immediate effect: e.g., set code_primary to doc weight 0.5 (so code queries still get some docs but less), and docs_primary to code weight 0.3 (so docs queries show code only if very relevant). - These config changes are low effort and can be done immediately. (We must be careful: if code embeddings are significantly different scale, weight=0.3 might hide code completely; but initial tests will show). - **Modules:** `llmc.toml` (or corresponding default values in code). - **Estimate:** 1 day to tweak and test on baseline queries. - **Task 1.3: Scorer Boost Adjustments**
- Increase `scoring.code_boost` slightly (if it’s a factor applied to code files). Similarly, ensure any `doc_penalty` is set (the config exists but possibly 1.0 by default). For a quick win, we might do something like code_boost = 1.2, doc_penalty = 0.9 as default. These static boosts can push code up a bit. - This is a blunt instrument but easy. Do it in `llmc.toml [scoring]`. - Also ensure test files are penalized (maybe test_penalty exists). If not, we could identify by path like `/test/` and penalize slightly, because tests often match query words but aren’t what user wants (depending on user’s question). - **Modules:** `scoring.py` (if code to apply these factors, already present likely). - **Estimate:** <1 day (just config changes, minimal code). - **Task 1.4: Lexical Search Integration** (optional in Phase1 if quick): - Use Python’s Whoosh or similar to create an index on the fly for the repo text (or just SQLite FTS module). - If time allows, integrate a simple keyword search: In `search_spans`, after dense results, do an FTS query on `spans.content` for top 5 results. Many doc strings or code comments might be captured by this. - This result can be fused via RRF with others. - If complexity is high, skip to Phase2. But likely using SQLite FTS5 (since we already use SQLite for storage) is feasible. - **Modules:** Possibly extend `llmc/rag/database.py` to create FTS index for spans or use existing one if any. Then query via SQL. - **Estimate:** 2-3 days (index + query integration). - **Testing & Release of Phase1:**
- After these, run the evaluation set again. Expect to see improvements: e.g., Code@3 should improve (more code in top 3 for code queries). - Ensure no major regressions on docs queries (maybe slight drop in docs rank if code intrudes, but check that). - If something off, adjust weights. This phase is mostly config tuning, so iterate quickly. - Then document these new defaults and maybe provide an option to revert if needed (just in case).

Phase 2: Intermediate Improvements (4–6 weeks)
Goal: Introduce graph-based enhancements and a basic learned component (intent classifier), plus the cross-encoder reranker. This phase requires more implementation and possibly training data.

- **Task 2.1: Graph Neighbor Augmentation**
- Implement `graph_expand_candidates` as per blueprint.
- Steps: Load `rag_graph.json` in `llmc/rag/graph_index.py::load_indices` (likely already done on startup, ensure it's accessible).
- Provide utility to get neighbors: maybe unify `where_used_files` and `lineage_files` calls into one function that collects all relevant neighbors. Or call both if needed. (We should find what exactly

those functions return; likely one is “who uses this file” and one is “what does this file use”. We might want both directions.)

- In `search_spans`, after initial retrieval, do:

```
if config.graph.enable_expansion:
    candidates = graph_expand_candidates(candidates, top_n=X)
```

- Implement `graph_expand_candidates(list, top_n)`:
    - For each candidate in list[:top_n]:
    - find neighbors via graph_index (depending on type: if it’s code file, we might find docs that mention it or code that calls it; if it’s doc, find code it refers to).
    - For each neighbor not already in list, create a candidate entry with: content snippet maybe empty or we can fetch the first span of that file from the DB to have something. score = candidate.score * alpha (e.g., 0.5). a flag or metadata “source=graph” to identify it (maybe for logging).
    - Append them to list.
    - (Optionally, to avoid too large list, we can directly apply cross-encoder on them later to filter, but we can just include and let rerank sort them).
- Then proceed with scoring.
- **Edge cases:** ensure not to add if neighbor is identical to candidate (like linking to self).
- **Modules:** `graph_index.py`, `search.py`.
- **Estimate:** 1 week (including testing on some known relationships).
- **Task 2.2: Graph-Based Scoring Adjustments**
- Add graph degree boost: In `Scorer.score(span)` we have info about file path. We can maintain a global dict `graph_degree` (load at startup) that counts edges for each file. Then: if `enable_graph_degree_boost` in config, do `if span.file in graph_degree: span.score += beta * log(1+degree)`.
- Also possibly if a span’s file is present as neighbor of another high-scoring file, but that scenario is handled by neighbor injection anyway.
- Essentially, implement one or two simple graph priors and test if they help or not (we can keep them configurable and measure).
- E.g., `graph.centrality_boost = 0.1` parameter.
- **Modules:** `scoring.py`.
- **Estimate:** 2 days (calc and integration).
- **Task 2.3: Intent Classifier Integration**
- Based on labeled data from Phase0/1 or even heuristic outputs, train a classifier offline (could fine-tune a DistilBERT on a small dataset of queries we have, labeling them code vs docs).
- For now, perhaps implement a simple approach: logistic regression on unigrams as a proof of concept (we can code that quickly and even embed the coefficients for most predictive terms in config).
- But assuming we can fine-tune a small transformer (maybe use Hugging Face Trainer for a couple of epochs), result is a model file.
- Integrate by loading this model in `router.py` if `routing.options.enable_intent_model` is true.
- Could use `transformers` pipeline for text classification if ease, or do manual tokenization + our small model forward pass.

- If a heavy model (not likely, can be tiny ~50MB), maybe load at startup outside main LLM to not hog memory. But probably fine.
- Implement `predict_intent(query)` that returns probability or label.
- Then in router, override route weights accordingly: e.g., set `route_weights = {'code': p_code * 1.0, 'docs': (1-p_code)*1.0}` or more extreme if needed.
- Might also set a threshold to do exclusive: if p_code>0.9, do code route only (though we must ensure at least some docs search happens unless absolutely sure).
- We'll refine as we test on data.
- **Modules:** `router.py`, plus maybe create a new file for model (like `intent_model.py`) to keep it separate.
- **Estimate:** 1-2 weeks (including collecting training data, training, integration and testing).
- **Task 2.4: Cross-Encoder Reranker**
- Choose a model: e.g. `cross-encoder/ms-marco-MiniLM-L-6-v2` (a 6-layer MiniLM, fast) or a CodeBERT cross-encoder if exists. Alternatively, fine-tune one on our own small dataset if possible (less likely if not enough data yet).
- Implement in `rerank.py`:
    - Load model (using `transformers.AutoModelForSequenceClassification` and tokenizer).
    - For each candidate span (text snippet) in top K:
    - form input like `[CLS] Query [SEP] Snippet [SEP]`.
    - get output score.
    - Could batch them to speed up.
    - Replace their score with this output (or combine: maybe to start, just replace, trusting the cross-encoder).
    - Then sort candidates by new score.
    - This yields reranked list.
- Work out truncation: ensure snippet + query fits model's max length (likely 512 tokens). If snippet is longer, maybe take first 256 tokens of it. That's usually fine since snippet likely from DB chunk anyway.
- Add config: `rerank.crossencoder_model = 'name'`, `rerank.crossencoder_top_k = N`.
- Ensure this step is optional (if model not loaded or disabled, skip).
- Test on a few queries to see that it e.g. pushes truly relevant up if initial rank missed them.
- **Modules:** `rerank.py`, maybe `rag/__init__.py` to load model at startup or on first use.
- **Dependencies:** Need `transformers` library if not already in environment; assume that's available or can be added.
- **Estimate:** 1 week (model loading, coding, testing on examples).
- **Task 2.5: Additional Graph-Based Reranking (simplified)**
- If time permits, try a simplified graph rerank: after cross-encoder, if some results are connected, give a small boost to connected pairs.
- But perhaps cross-encoder will implicitly handle that by understanding content. So this might be optional.
- Possibly skip or keep minimal (like if multiple results from same file or same module, ensure they are not all at bottom).
- We likely skip this explicit step in Phase2 to avoid over-complicating until we see need.
- **Testing & Tuning Phase2:**

- Now we have multiple new components. We test each in isolation first: e.g., run retrieval with graph expansion on some known case to see neighbors appear; test intent classifier by printing what it predicts for some queries; test cross-encoder by comparing its ranking vs initial.
- Then do an end-to-end run on evaluation queries:
    - Compare metrics from Phase1 vs Phase2. Ideally see further improvement in code queries (with graph and cross-encoder, code files that were previously slightly lower should move up more). Also overall nDCG hopefully up due to cross-encoder fine discrimination.
    - Check any doc query regressions. If cross-encoder is general, it might actually improve doc queries as well by cleaning false positives.
- Fine-tune: adjust `alpha` for neighbor score if needed, adjust top_n for expansion to avoid too many extras, adjust classifier threshold if noticing misroutes.
- Check performance: measure average retrieval time. Cross-encoder will add some ms; check it's within acceptable range. If too slow, consider lowering top_k or switching to a faster model.
- At end of Phase2, we should have a robust retrieval that solves the main issues and is already learning-based to some extent.

Phase 3: Advanced Enhancements (6–8 weeks or more)
Goal: Achieve best possible performance with heavy but optional components (LLM reranking, learned ranker training, human feedback integration) and cement the evaluation/feedback loop.

- **Task 3.1: LLM Listwise Reranker**
- Integrate a local LLM for reranking top results. Possibly fine-tune one (if we have resources, e.g., fine-tune a 7B LLaMA on a few hundred ranking examples with a ranking loss). But fine-tuning LLM is heavy; instead we can prompt-engineer.
- Develop prompt template as earlier described. Perhaps create a function `llm_rerank(query, candidates)` in `rerank.py` which returns reordered list.
- Use the already loaded main LLM (if it's accessible programmatically). Since LLMC uses an LLM to answer, we may need to instantiate another pipeline to call the same model for our prompt (to not interfere with user session). Or if using an open model, we could load it again (memory heavy though).
- Alternatively, use an external call to GPT-4 as an experimental feature behind a flag (since local is priority, but perhaps allow it if user has API key).
- Implementation complexity is moderate: formatting string, sending to model, parsing output. The key is to ensure deterministic parse. Possibly ask for output as JSON or a sorted list like "Ranked: 3>1>2>...".
- We need to manage context length – limit candidates to maybe 5, and ensure snippet lengths such that combined prompt ~ < 2000 tokens.
- Put this behind config `llm_rerank.enable` default false (since not all users can run this smoothly).
- Testing: feed some known cases where cross-encoder struggled (if any) to see if LLM fixes it. Also test that it doesn't hallucinate weird order.
- Evaluate difference on metrics if possible (though with small dataset, might be minor).
- **Modules:** `rerank.py` and possibly an `llm_client` util if needed.
- **Estimate:** 2 weeks (including prompt refinement and performance tuning).
- **Task 3.2: Graph Neural Reranker (Research-heavy)**

- If resources allow, attempt training a GNN on the retrieval graph. This would involve:
    - Construct training instances from our dataset: treat the top 50 retrieved as a graph (with edges from `rag_graph` between them), use our relevance labels as supervision (which ones are relevant).
    - Possibly use a GNN library to train a model that takes this graph and outputs a score for each node, optimizing nDCG or similar.
    - This is a significant research effort. Given limited data, maybe not feasible to fully realize. Alternatively, implement an unsupervised graph rerank like the L2G approach (Log-based graph in Yoon et al. 2025) which was basically using co-occurrence without training [1] [4]. But we don't have user logs of co-occurrence yet.
    - Perhaps skip this or do a simplified variant: e.g., use PageRank on the subgraph of candidates. That we can do easily: treat each candidate as node, edges from graph, run a few iterations of PageRank, use resulting score (combined with initial).
    - Implement this as an alternative rerank method in `rerank.py` (`graph_rerank.enable`).
    - Without training, results may not be clearly better than simpler methods. But could be a fun experiment on our eval set. We monitor if it helps.
- **Modules:** Could utilize `networkx` to run PageRank on a small graph.
- **Estimate:** 3-4 weeks if doing full training, but if doing unsupervised small algorithm, 1 week.
- **Task 3.3: Learned Rank Fusion Model**
- Using the judgments from Phase2 (which hopefully we have more of by now), train a simple linear model to combine scores/features. Possibly redundant if cross-encoder and LLM do the heavy work, but if we want an automated tuning of weights for dense vs lexical vs graph:
    - Use our labeled queries, and maybe do a grid search or machine learning to find best route weight or BM25 weight.
    - For example, if we find that 70% dense + 30% BM25 is optimal, set that in config and possibly remove RRF in favor of weighted.
    - This is minor but ensures our static combination is optimal.
- **Modules:** primarily offline/training. Then update `fusion.mode="weighted"` and weights in config.
- **Estimate:** 1 week (analysis + applying).
- **Task 3.4: Human Feedback System**
- Develop a simple interface or process for continuous labeling:
    - E.g., every week, take 10 new queries from logs (especially those where user behavior suggests dissatisfaction: multiple rephrases, long time, etc.), have a team member label their results.
    - Add to our eval set. Recompute metrics over time.
    - Not strictly coding, more process. But we might automate some part:
    - Logging user interactions: ensure we capture queries and maybe which retrieved snippet user clicked (if UI allows clicking to open file, etc.). If not, maybe see if user had to ask follow-up.
    - If we have these signals, we can define triggers for queries to review.
    - Also, incorporate a feature for user to give explicit feedback ("The assistant gave irrelevant info because..."). If LLMC UI can be extended, perhaps a simple feedback command that logs the query and feedback.
    - For now, probably keep it internal.

- The main coding might be adding more comprehensive logging. Possibly in `search_spans` we can log each query, chosen route, top results (anonymized), and eventual user rating if available (this could be a field in our judgment collection, not in code).
- Additionally, ensure the evaluation harness (perhaps using RAGAS library or our own) can compute our metrics easily. We might integrate RAGAS to compute context precision/recall once we have some answer annotation.

- **Estimate:** 2 weeks (in parallel to others, since it involves setting up labeling workflows).

- **Task 3.5: Performance and Sharp Edges**

- Before final deployment, we spend some time profiling. See if any step is too slow:
    - Cross-encoder: if running on CPU and taking >0.5s per query, consider optimizing (maybe reduce top_k or use quantized model).
    - LLM rerank: likely disabled by default, but if enabled, warn users about latency.
    - Graph expansion: see if loading or using graph is fast (maybe cache neighbor lists in memory to avoid repeated JSON parsing).
    - Memory: loading multiple models (embedding model, cross-encoder, LLM) might be heavy. If memory is an issue on smaller hardware, consider config to turn off cross-encoder and rely on just LLM or vice versa. Provide guidelines in docs.
- Implement any needed caching: e.g., keep cross-encoder model on GPU if possible, re-use it for subsequent queries to avoid re-loading (should already if loaded globally).
- Test concurrency if relevant (maybe not heavily for local use, but at least ensure no major issues if two queries come simultaneously – e.g., LLM reranker might need separate instance or locking).

- **Estimate:** 1 week for thorough testing and fixes.

- **Task 3.6: Documentation & Fallbacks**

- Update LLMC's documentation to describe new options (graph, rerankers, etc.), so users know how to enable/disable.
- Ensure that each new feature can be turned off to get back to a simpler baseline:
    - If `enable_graph=false`, our code should skip those steps cleanly.
    - If `enable_crossencoder=false`, skip that.
    - If `enable_intent_model=false`, use default routing weights from config.
    - All these fallbacks basically revert to a Phase1-like behavior. We should test that toggling them indeed returns near baseline performance to ensure no hidden coupling.
- Provide default config values that are safe (maybe leave LLM rerank off by default, leave cross-encoder on if we assume local model is provided).
- **Estimate:** 3 days for writing and final polish.

Phase 4: (Optional) Online A/B Testing and Continuous Improvement (ongoing)
- After release, gather usage data. Possibly run an A/B if we have enough users: e.g., some queries with new system, some with old (if we can split based on a flag). - Or at least monitor feedback if any user flags outputs. - Use that to further refine. This is less a scheduled phase, more an ongoing cycle (especially with active learning). - Could involve slight tweaks like adjusting graph expansion if we see it occasionally adds noise (maybe gating that also by an "expansion confidence" threshold, like only expand if top doc was very relevant). - This phase is indefinite, basically maintenance and improvement.

Time/Cost Estimates Summary:
- Phase 0 & 1: ~4 weeks (mostly one engineer tweaking config and verifying). - Phase 2: ~5 weeks (involves some model training for classifier and integration of cross-encoder). - Phase 3: ~8+ weeks (LLM integration, advanced modeling – could be more if research heavy tasks extend). - Overall ~3-4 months to implement all to full extent, but Phase1 and 2 deliver the key improvements in ~2 months. We can deliver value incrementally: - After Phase1, push an update (immediate relief for code queries). - After Phase2, another update (with graph and better ranking, which likely covers most issues). - Phase3 features might be provided as experimental or power-user options given their complexity and resource needs.

Resource considerations:
- We might need a GPU for training classifier (though could also do on CPU if small data) and for possibly running cross-encoder/LLM. If targeting local usage, we assume users might have at least one GPU for the LLM itself. If not, our cross-encoder should be light enough for CPU. The classifier overhead is negligible at runtime (very fast). - Memory: storing the graph – `rag_graph.json` maybe a few MBs (depending on repo size, edges likely in the thousands, which is fine). Loading cross-encoder model ~100MB. Acceptable on most dev machines. - The biggest is if user enables LLM rerank with a large model (like if they configured to call GPT-4 via API – cost money and slow). We will not do that by default; it's just an option. - Human labeling cost: initially internal so just our time, later if we want more could consider crowdsourcing but domain knowledge is needed, so likely keep it in team or use experienced annotators. Volume is small (maybe a few hundred judgments).

Module Summary of Changes:

- `llmc.toml`: new keys in [scoring], [fusion], [routing], [rerank], [graph].
- `llmc/routing/router.py`: incorporate classifier and advanced logic.
- `llmc/routing/fusion.py`: new fusion modes (rrf, weighted).
- `llmc/rag/search.py` (`search_spans`): incorporate lexical search call, call graph expansion, apply rerankers.
- `llmc/rag/graph_index.py`: ensure methods for neighbor lookup, possibly precompute degree.
- `llmc/rag/scoring.py`: add graph-based score tweaks.
- `llmc/rag/rerank.py`: implement cross-encoder and possibly LLM rerank logic.
- Possibly new file like `llmc/rag/lexical.py` or integrate lexical in `search.py` depending on design.
- Add any dependencies in `requirements.txt` if not present (like `rank_bm25` or `transformers` if not already required).
- Test cases: update or add unit tests for:
- Fusion (given toy inputs, does rrf and weighted produce expected order?).
- Router (given some artificial queries, does it choose expected route? We can simulate classifier by monkeypatching).
- Graph expand (if we set up a small graph in memory, does expansion add correct neighbors?).
- Rerankers (maybe more integration test than unit; ensure it runs without error on a sample).
- Monitoring plan: We'll include debug logs as mentioned. Also, after deployment, we'll periodically run our eval set through the system to catch any drift (especially if embedding model changes, we re-evaluate if weights still good).

Each feature is controllable, so if any issues occur in production (like performance), users can disable that feature via config to fall back. For example, if cross-encoder is too slow on someone's machine, they can set `enable_crossencoder=false` and still benefit from other improvements.

Thus, the implementation plan delivers incremental upgrades to LLMC: - Immediate config tuning (Phase1) for quick impact. - Medium-term structural improvements (Phase2) with graph and ML models. - Long-term best quality (Phase3) adding cutting-edge rerank and continuous learning, carefully integrated as optional to not disrupt baseline usage for those who can't afford extra cost.

Evaluation Plan (Offline + Online)

A rigorous evaluation plan is crucial to verify improvements and avoid regressions. We describe both offline evaluation with labeled data and an approach to online evaluation (if applicable), including metrics, significance testing, and test scenarios.

Offline Evaluation

Datasets and Query Buckets:
We will assemble a test suite of queries categorized by intent: - **Code-Intent Queries:** e.g., "Where is `getUserData` implemented?", "How does the system calculate the checksum?" – expecting a code answer. We gather these from actual usage logs (if available) or create them based on project knowledge. Aim for at least 50 queries in this category. - **Docs-Intent Queries:** e.g., "How to configure the max retry setting?", "What does error code 4041 mean?" – expecting an explanation or documentation snippet. Again ~50 queries. - **Mixed/Uncertain Queries:** e.g., "Authentication flow" (could be code or docs), or "Initialize database (example)" – ambiguous or broad. ~20–30 queries. - **Edge/Adversarial Queries:** e.g., queries that contain both code-like and doc-like terms ("function X memory leak documentation"), or queries with misleading terms (a code-looking token that's actually generic). Also include queries with terms that appear in tests a lot to see if system avoids test trap. Perhaps 20 of these.

Additionally, if possible, include some multi-hop questions (if any exist, e.g., "Which module calls function X?" – requires linking code relationships).

We will have humans label the *relevance of each retrieved item* for these queries. We use the rubric: - 3 = Highly relevant (direct answer or contains the needed info). - 2 = Partially relevant (on topic but maybe not complete or secondary info). - 1 = Marginally relevant (mentions terms but not really useful). - 0 = Not relevant.

Also label whether an item is Code or Doc (to compute Code@k).

We'll target at least the top 5–10 results from each system variant for labeling, since beyond that likely not needed for nDCG@10 and such.

Metrics: - **Recall@k (especially k=3 or 5):** Measures if at least one relevant item is in top k. We'll break this down: e.g., for code-intent queries, measure recall of any code file relevant (CodeRecall@3). For docs queries, measure recall of relevant doc. - **Precision@k:** Out of top k results, how many are relevant? We might use Precision@3 or 5 to ensure we're not adding junk. - **nDCG@k:** Using the graded relevance, to evaluate ranking quality with position consideration (nDCG@5 or @10 typically). We will compute this overall and by category (e.g., nDCG@5 for code queries). - **MRR (Mean Reciprocal Rank):** Good for evaluating if the first relevant result is high. Particularly for code queries, MRR should improve if the relevant code moves from rank 4 to rank 1 etc. - **Code@k metric:** We define, for code-intent queries, the fraction where at least one code result is in the top k. Essentially this is the recall of relevant code as primary. We may also track the average rank of first relevant code (a kind of MRR specifically for code presence). - **Context Precision/Recall (for RAG):** If we can, we'll attempt to measure these. To do so, we need to define what info is needed to answer the query. For each query, we can identify the *ground truth answer or source.* For instance, if a question is "Where is X implemented?", the ground truth might be "in file Y, function Z". So if our retrieved contexts included file Y, then context recall = 1 (it had what was needed). If it also included a bunch of irrelevant stuff, context precision might be lower. We can approximate: - Context Recall = 1 if among the retrieved top k there was a snippet that contains or directly leads to the answer, else 0. (We can glean this from whether a highly relevant item was retrieved). - Context Precision = (# of retrieved items actually used in answering) / (# of retrieved items used). This is tricky to get without generating answers. Alternatively, measure the proportion of retrieved items that were judged relevant (since if they are relevant, they presumably contribute or at least not distract). - So maybe use % of top 5 that have relevance >=2 as a proxy for context precision. - **Answer correctness / faithfulness (optional):** Ultimately, after retrieval, the LLM produces an answer. We could evaluate the final answers for correctness to see if retrieval improvements correlate. However, doing a full answer eval might be complex and introduces LLM variability. Initially, focus on retrieval metrics. Later, perhaps choose a subset of queries to actually run through the LLM with old vs new retrieval and have evaluators judge answers (or see if the answer now includes correct code reference vs previously not). - Also track **ranking of ground-truth**: for queries where we know which file or doc is the answer (like we framed it ourselves), check its rank. That's akin to recall/precision but sometimes easier: we want that ground truth in top 3 ideally.

We will compute these metrics for: - Baseline (original system). - After Phase1 changes. - After Phase2 changes. - After Phase3 changes (if applicable). This shows progress.

Statistical Significance:
Given the number of queries (~100-150), we can use paired significance tests: - For metrics like MRR or nDCG per query, we can do a paired t-test or Wilcoxon signed-rank between baseline and new system results. - For categorical metrics like recall@k (success rate), we can use McNemar's test (since it's paired yes/no outcomes per query for baseline vs new). - We aim for $p < 0.05$ to claim improvement is significant. Given the improvements expected are relatively large (based on intuitive examples), we anticipate many will be significant, but we confirm. - We also look at effect sizes (e.g., how many queries improved vs degraded). Ideally, improvements in code queries should not come at large expense of doc queries (maybe slight trade-off but hope not). We will quantify: e.g., out of 50 doc queries, baseline vs new MRR difference and see if any drop is significant (shouldn't be, but we check). - If a trade-off is found (e.g., code queries much better, doc queries a bit worse), we have to decide if that's acceptable. Possibly tweak weights to mitigate.

Robustness tests:
- **Embedding model drift:** If in future the embedding model is changed (say user upgrades to a different vector model), we should test if our learned weights and classifier still perform. We can simulate this by intentionally perturbing scores or trying an alternative embedding (if available) to see if our pipeline still holds. We can also design our pipeline to be robust: e.g., RRF doesn't care about absolute scale, so if new embed model has different range, RRF still works. The cross-encoder and rerank stage ensure final sorting by actual content, mitigating differences in initial ranking. So we hypothesize the system will adapt relatively

well, but we’d re-run the evaluation with the new model if such an event happens and possibly re-tune some weights. - **Adversarial queries:** We will test queries that might trick naive systems: - e.g., Query that contains a common doc term but actually needs code: “Show me the implementation of login (not the documentation)” – our intent classifier should catch “not the documentation” and route to code. We’ll verify it does. - Query that looks like code (has parentheses) but is actually a general question: e.g., “What is best practice for function main() usage?” – likely more a docs question about usage patterns, our system might mis-route to code by seeing `()`. We test such and see if results are still reasonable. Perhaps our multi-route ensures docs still appear. If we find failure, we refine heuristics or allow the LLM reranker to correct (LLM might realize none of the code answers the “best practice” and might rank a doc higher if present). - Queries with contradictory signals or very broad: ensure system doesn’t crash or do something weird. Possibly queries with no good answer in corpus – system should just retrieve whatever closest and the LLM might say “I couldn’t find info”. That’s okay; just ensure the pipeline doesn’t break if retrieval returns empty (shouldn’t, dense always returns something albeit low score). - **Failure mode analysis:** For any query where new system didn’t improve or got worse, analyze why. E.g., did graph expansion add a wrong neighbor that got ranked high? Did the intent classifier misroute? This analysis will feed back into refining those components. This is part of offline evaluation – basically error analysis on our labeled set.

Ablation Studies:
To justify each component, we’ll do some ablations: - Run system with graph features off vs on (but other things same) to isolate graph’s contribution. - With cross-encoder off vs on, measure effect. - With intent routing off vs on. - This helps confirm that each part is beneficial. If any part shows minimal gain for complexity, we might reconsider using it (e.g., if cross-encoder yields only 1% better and costs 300ms, maybe rely on LLM or vice versa). - Likely outcomes: graph expansion significantly improves code recall; cross-encoder improves nDCG a bit; intent routing mainly improves precision (removing some wrong-route results). - We will document these improvements with citations to numbers (for internal report, but summary for final).

Online Evaluation (Optional/Planned)

If LLMC is used interactively by users, we can collect some online signals: - **User click/dwell:** If UI allows user to click on a retrieved snippet to open full file or highlight, we can track how often the user clicked a code snippet vs a doc. Ideally, for code queries, we want to see more code snippet clicks (assuming a click means it was relevant to user). - **Follow-up queries:** If a user immediately asks a follow-up like “No, I meant the implementation” after seeing an answer (or uses a refine query feature), that indicates the initial retrieval might have been off. We can compare frequency of such events before vs after. - **User ratings:** If the system has a thumbs-up/down on answers, that could indirectly reflect retrieval quality. Not perfectly (could also be generation quality), but if we see a trend of improved ratings post-improvement, that’s good. - We might run an A/B test: half of queries use baseline retrieval, half use new (if we have enough usage to split). Then compare outcomes: which got more positive feedback or less follow-ups. Because our user base might not be huge, this A/B might be informal or time-sliced (e.g., one week baseline, next week new, under assumption of similar query distribution). - **Metrics online:** Possibly compute *success rate* as: was the user satisfied in one turn (didn’t re-ask the same question or rephrase). Track that ratio historically and see if it improves. - Also measure latency: ensure the more complex pipeline doesn’t degrade responsiveness too much. If we find average response time went up, see if still acceptable. Possibly measure tokens in prompt or generation because more context means longer answers maybe – keep an eye on that.

Regression Gates:
We will implement monitors that if something goes wrong in retrieval, we catch it: - e.g., if retrieval returns extremely low similarity (embedding) for top results consistently, perhaps the embedding model changed or something – we should log warning. - If the pipeline took > X seconds for a query, log that along with features enabled to identify bottleneck. - We won't automatically disable features in production on the fly, but at least logs alert dev to issues so we can patch. - Another gate: if the LLM reranker output doesn't parse or seems ill-formed, we can catch that and fall back to cross-encoder result for safety (so user isn't affected by a parsing error). - We could consider a sanity check on final results: if top results all have extremely low scores or are all irrelevant by some heuristic, maybe something went wrong (but that scenario unlikely if pipeline works).

Benchmarking vs Standard Datasets:
Optionally, we might test our pipeline on a public dataset like CodeSearchNet or a subset of StackOverflow Q&A to gauge performance relative to known baselines. For example, if project has Q&A pairs, we can see if the relevant code is retrieved by our system. This is not directly necessary, but could provide additional confidence that the approach is generally strong. However, our primary focus is on this specific codebase context (where doc and code are tightly coupled), so we emphasize custom evaluation.

To conclude, the evaluation plan ensures: - We quantitatively verify each improvement (with metrics and significance). - We identify any negative impacts and address them. - We have a continuous loop (with human labels and user signals) to refine the system further. - By defining code-specific metrics (like Code@k), we directly measure the solution of the problem at hand (docs dominating code queries) [33] . If Code@3 goes from, say, 50% to 90% for code queries, that's a clear win. If nDCG for docs queries stays within 1-2% of baseline, we know we haven't hurt docs much.

We will present these evaluation results in reports as we progress through phases, ensuring stakeholders can see the tangible gains at each step. This data-driven approach will guide final tuning decisions (e.g., how high to set code_boost) and ensure the final deployed system is objectively better for users in the ways that matter.

Human Scoring System Design

To reliably evaluate and improve retrieval relevance, we will implement a structured human scoring system. This involves a clear **labeling schema (rubric)**, training for annotators, and a process to incorporate these labels into model development. We also plan to leverage LLM assistance for labeling in a limited capacity, with human validation, to increase efficiency.

Label Schema & Rubric:
We define separate but compatible rubrics for code-intent and docs-intent queries (since relevance can look slightly different between them):

- **Relevance Levels:** We use a 4-point scale (0 to 3):
- **3 (Highly Relevant):** For code-intent: the snippet contains the implementation or directly answers the query (e.g., the function definition asked for, or code that clearly performs the requested operation). For docs-intent: the text directly answers the question or explains the concept asked (e.g., a paragraph describing the exact feature or error code).

- **2 (Relevant):** Partially answers the question or is on the topic but maybe not complete. Code-intent: perhaps a usage example of the function rather than its definition, or a closely related function implementation. Docs-intent: a page that discusses the topic but not specifically the asked detail, or perhaps requires inference.
- **1 (Slightly Relevant):** Touches on the query's keywords but is mostly off-target. Code-intent: code that mentions the term but in a different context (e.g., the term appears in comments or in an unrelated function). Docs-intent: a page that includes the term in passing or a related concept but isn't helpful to answer the query.
- **0 (Not Relevant):** Completely unrelated to the query (no useful information). This includes results that just match common words or are wrong domain (like returning a generic library file for a very specific question).

These definitions will be documented in a guideline document. We will include examples for each: - e.g., Query: "How is encryption key generated?" – Show a code snippet that generates a key (relevance 3), a doc page describing keys (3 if direct, maybe 2 if general), a code snippet just referencing "key" in variable name (1 or 0 depending). - We'll clarify that for code queries, usually only the actual implementation is a 3, closely related usage or definition could be 2, etc.

We also define how to treat multiple spans from same file: If a file definitely contains the answer but maybe the snippet doesn't show it fully, we might still give it a 3 if we believe the file is the right one (since presumably the assistant could use more of it). We'll instruct annotators to base on snippet but also consider the file context if obvious (they can click to see more if needed).

Additionally, each labeled item will get a tag for type: Code vs Doc (we can infer from file extension typically, but we can also explicitly mark it). This is for metrics like Code@k.

Handling Mixed Queries:
If a query legitimately needs both explanation and code (rare, but e.g. "What does function X do and where is it implemented?"), we can label relevance accordingly (both code and doc might be highly relevant). But mostly we categorize query by primary need and label accordingly.

Annotator Training:
We will have a small team (perhaps ourselves and a few colleagues familiar with the codebase) do the annotations. We'll conduct a calibration exercise: - Everyone labels the same 10 queries, then compare. - Discuss any disagreements and refine rubric definitions. For instance, if one person gave a 3 to something and another gave 2, figure out why and clarify guidelines. - Aim for consistent interpretation especially of 2 vs 3 (which can be subjective). A rule might be: If the snippet alone fully answers the question, it's a 3. If you'd need to read a bit more or it's relevant but not directly the answer, it's a 2.

We set a target inter-annotator agreement (IAA). Using weighted kappa for ordinal labels or simply percentage agreement on relevant vs not. We aim for at least $\kappa > 0.7$ on the scale (substantial agreement). If lower, we revisit training.

Sampling Strategy for Labeling:
We can't label everything, so we choose wisely: - Use stratified samples: some easy queries, some hard known problematic ones. - Use active learning cues: queries where our new system and baseline differ – these are critical to label to see which is correct. - Also queries where the system's confidence (like classifier

output ~0.5 or low scores) is low – likely challenging queries, good to label and see what's happening. - Over time, as we gather logs, we might pick up new queries that come in. - Ensure each new feature is covered: e.g., queries that would exercise graph connections, ones that rely on lexical matches, etc.

Labeling Process:
We'll likely use a simple tool: maybe an Excel or Google Sheet listing Query, Candidate snippet, (maybe highlight the matching part). Annotators put score and any comments. - Alternatively, use an existing tool like Labelbox or even a custom script to present query and top results and record ratings. - Since our scale is small, a spreadsheet can suffice initially.

We'll label initial set for evaluation and continue to label new queries as needed for retraining models: - e.g., if we want to improve the intent classifier, we label more queries for that specifically.

LLM-Assisted Labeling:
We might try using an LLM to pre-score some results to save time, especially for obvious cases: - For each query+snippet, we could prompt GPT-4: "Q: ...; Snippet: ...; Is this relevant? (choices 0-3)". Actually, GPT-4 might be quite good at this given it can read code and English. - We'd use chain-of-thought prompting (maybe provide rubric in prompt) to have it simulate a judge [34]. For example: - "You are an expert programmer. The user question is X. I retrieved this code snippet: ... . According to the criteria (explain criteria), how relevant is it? answer just a number." - Let GPT-4 label a batch. Then humans review those labels rather than doing from scratch. They can probably very quickly spot if GPT was wrong (like if it hallucinated context). - This could speed up labeling but we must be careful: GPT-4 might give something a 2 when truly it's 3 or vice versa if it misinterprets snippet. So we'd treat it as a helpful second opinion, not ground truth. - Particularly for borderline cases, we have humans decide.

We likely use LLM assistance for volume if needed, but given our dataset is not huge, humans can manage. However, for continuous labeling at scale (if we integrate in a bigger product), this approach could scale up (with human verification of a sample to ensure quality remains high).

Active Learning & Iterative Labeling:
As mentioned in Phase 3, we will use active learning: - The system flags queries where, for example, the intent classifier was uncertain or where the LLM reranker gave a different order than cross-encoder (indicative of a conflict). Those queries we prioritize for human labeling to see which method was correct. - Another strategy: any query where user gave negative feedback, label that to diagnose retrieval issue. - If we ever incorporate a learned ranker or fine-tune cross-encoder on our data, we can use active learning to choose which additional query results to label to improve it (like focusing on cases it got wrong in validation). - Maintain an ongoing list of "queries to label" and schedule regular labeling sessions.

Quality Assurance in Labeling:
We will periodically measure IAA: - If multiple annotators, double-label ~10% of the data to ensure consistency. Compute kappa or percent agreement. If we see drift (like one annotator being more lenient), we sync up and adjust. - Also implement simple checks: if an annotator marks something 3, ensure it indeed contains answer. Possibly have a second person quickly glance at all 3's to confirm (since 3 means presumably solved). - We keep a log of labeling decisions and rationales (maybe annotators can write a note if something is borderline). This helps when training models – we can see if model might struggle on certain patterns.

Incorporating Labels into System:
- **Intent Classifier Training:** Use labeled query intents (which we derive from context: for each query in our set, we know whether the desired answer was code or doc from the highest relevant items). We can label query intent directly as well as derive from which results were marked highly relevant. Then train classifier (like we did). - **Learning to Rank:** If we decide to train a learning-to-rank model (like LambdaMART on features or fine-tune cross-encoder), these labels are the training target. We would, e.g., use all query-doc pairs with their 0-3 labels to train a model to predict those. Possibly convert to pairwise preferences (all pairs where one has higher label). - **Evaluation & Leaderboard:** Maintain a "leaderboard" of different system configs on our labeled set to track progress. E.g., baseline nDCG vs current. This labeled set effectively becomes our internal benchmark. We should be careful that as we tune system to this set, we don't overfit – but since queries are real ones, overfitting just means it's very good on those, which likely correlates with being good in general. We can always reserve a few queries as a final test not to tune on (like a dev/test split). - **Preference Modeling:** Another interesting use: we can do side-by-side comparisons. For example, show an annotator the results of baseline vs results of new system for a query, and ask which set is better (or if new fixed something). This is more of a QA of improvement rather than a label for a model, but helps confirm we're making progress. If we systematically see new system preferred in, say, 80% of cases by humans, that's strong evidence of success.

Examples in the Rubric Document:
We will include concrete examples in our annotator guide: - Query: "Where is the user password hashed?" - Snippet A: `function hashPassword(...) { ... }` (actual implementation) – definitely a 3 for code-intent. - Snippet B: Documentation paragraph: "Our system hashes passwords using bcrypt in module Auth." – If code-intent, this is 2 (descriptive but not code); if docs-intent (like if user wanted explanation), this would be 3 for docs query. - Snippet C: A test case setting a dummy password and calling hash (just usage) – probably 1 (not explaining or showing implementation). - We'll do similarly for a docs query.

Edge Cases Clarification: - If a code snippet is relevant but maybe incomplete (like part of function shown, but presumably the rest off-screen) – we still give it 3 if it's the correct function, because the system can fetch the rest or we assume the chunk is representative. - If the snippet is relevant but outdated info (like referencing an old version function name not used now) – that's tricky. Possibly mark lower because it might mislead. But if in the project context it's irrelevant because it's old code, then it's irrelevant in current context. We'll avoid such issues by focusing on current project state in labeling. - If multiple snippets are needed together (say a function and its documentation both needed to answer) – each can be marked relevant individually. The metrics like context recall/precision handle combination.

Integration with LLM-assisted search and QA checks:
We might also incorporate LLM in QA: - For example, after labeling, we could prompt an LLM to check consistency: "Given the query and these top results with their relevance scores, do the labels make sense?" This could catch obvious mistakes (like if an annotator accidentally marked something 0 that clearly contained the answer text). - It's experimental, but could use it as a second pair of eyes for glaring errors.

Inter-Annotator Agreement Targets:
As mentioned, we aim $\kappa > 0.7$. If after initial calibration it's, say, 0.6 (some disagreement on fine points), we'll refine. It's important because inconsistent labeling can confuse learning models and metrics. We might drop to simpler binary relevant/not relevant if needed for certain analysis (like compute precision at k by counting anything >=2 as relevant). But graded is better if consistent.

Active Learning Implementation:
We'll schedule periodic rounds. For instance, every two weeks: - Gather new candidate queries (via logs or new feature corner cases). - Label them (with at least two annotators verifying). - Add to training sets and retrain any models if enough new data (maybe accumulate until have, say, +20% data, then retrain). - Also update our evaluation metrics continuously.

This human-in-the-loop system ensures our retrieval stays aligned to human judgment, and as the codebase or usage evolves, we adapt. It also provides a safety check: humans reviewing outputs can catch if, say, the system starts doing something undesired (like over-emphasizing some frequent but not relevant patterns) and we can correct course, either by adjusting weights or collecting more data to retrain components.

In summary, **the human scoring system** provides the ground truth backbone for our evaluation (explicit graded judgments), guides the training of learned components (classifier, reranker), and gives continuous feedback for improvements. By combining careful rubric design, training for annotators, and selective labeling strategies, we aim to maximize label quality and utility. LLM assistance will be used pragmatically to boost efficiency, but final judgments rely on human validation to ensure reliability [20]. This approach will result in a robust set of relevance judgments that keep the LLMC retrieval tuned to what users truly find relevant and useful.

Risks, Failure Modes, and Sharp Edges

No system upgrade is without risks. We enumerate the potential failure modes and issues in our proposed approach, along with mitigation strategies:

1. Score Scale Mismatch:
When combining different scoring systems (dense, BM25, cross-encoder, etc.), their scales differ. A dense similarity might range 0–1, BM25 could go up to 10s or more, cross-encoder logits could be uncalibrated. This can lead to one source dominating if not handled. For example, initially we saw doc embeddings often outrank code because maybe doc embeddings had higher absolute similarity for semantic matches. - *Mitigation:* We introduced rank-based fusion (RRF) specifically to avoid raw score comparisons [16]. RRF is insensitive to absolute values, focusing on rank order from each source. This greatly reduces risk of one source overpowering due to scale. Additionally, we apply normalization: e.g., for weighted fusion, we might normalize each list's scores to [0,1] by dividing by max or using a logistic transform. We can also calibrate via a small set (as done in Pinecone's analysis) to set weights properly [17]. - We'll thoroughly test edge cases: e.g., a query with no lexical matches (BM25 scores all zero) – ensure that doesn't produce weird fusion result (we can set a default so if BM25 all zeros, the weighted sum doesn't just zero out everything). - If needed, implement a safety: check if final fused scores look extreme (like all 0 except one route), and in that case revert to rank merge.

2. Reranker Instability and Hallucination:
Using an LLM to rerank (or even to generate rationales) can introduce unpredictability. It might rank incorrectly by using outside knowledge (e.g., the LLM "knows" the answer and thus favors a doc that it thinks matches its knowledge, even if snippet doesn't). Or it might produce non-deterministic results if prompt not tightly controlled. - *Mitigation:* We use zero-temperature and fixed prompts to make LLM reranking as deterministic as possible. We instruct it to only consider provided content. We also validate LLM output format. If it outputs something invalid (or if it seems to be basing decision on knowledge not

present), we have a fallback: default to cross-encoder rank. Essentially, LLM rerank is an enhancement, but if we detect any anomaly (via regex or sanity checks), we log a warning and ignore that output. - Additionally, we don't rely on LLM giving a "reasoning chain" for ranking (that could hallucinate content from outside), we ask directly for final ranking to minimize unnecessary text generation that could go awry [35]. - We'll test with known tricky cases to ensure LLM isn't biased incorrectly (like queries about nonexistent things – LLM might have seen something in training and hallucinate relevance; our guard is that if none of the snippets mention that thing, LLM should rank them low; if it doesn't, that's a hallucination sign). - Also, we keep LLM rerank optional – conservative users might keep it off if they worry about stability and just use cross-encoder + graph which are more stable, albeit possibly slightly less accurate.

3. Graph Noise Amplification:
Graph expansion could bring in irrelevant nodes. For instance, a popular utility module might be linked to many files and could appear as neighbor often without being relevant to the specific query. If our algorithm boosts connected nodes, a highly connected but off-topic node might bubble up (like a logging module appearing in results just because everything calls it). - *Mitigation:* We set rules: only expand from top relevant items (so if a doc is relevant, its neighbors likely related; but if the doc was a broad page linking to many, we might restrict that). We also introduced semantic filtering – checking neighbor's embedding similarity to query before adding. If the neighbor doesn't share keywords or vector similarity with query, skip it, even if it's connected. - We also down-weight expansions (score * α) to ensure they don't outrank the origin unless clearly needed. This means a neighbor typically comes in a bit lower unless it's also directly somewhat relevant. - Over-connection issue: we might add a rule to ignore extremely high-degree nodes unless the query specifically suggests them. E.g., if a node is connected to >50 files (likely a generic util), we require a stronger direct match to include it. - On reranking, the cross-encoder should further filter these: a logging module snippet likely won't be rated highly relevant to a specific functional query by the cross-encoder model because the query context doesn't match. So second-stage reranking will naturally push truly irrelevant graph additions down. This reduces harm of adding a somewhat irrelevant neighbor – it might be last rank and not chosen in final context. - We will examine any cases where a graph addition appears in top 3 to ensure it was indeed relevant (if not, adjust parameters). - Worst-case, if graph info turned out too noisy for a particular repo, user can disable it (setting `enable_graph=false` would revert to plain retrieval).

4. Over-Routing to Code (or Docs):
The intent classifier or heuristics might misfire, leading to missing relevant info. For example, classify a query as code intent and thus only bring code results, but actually the answer was in documentation or requires a combination. Then the user might not get a complete answer. - *Mitigation:* We rarely absolutely exclude a route; even for code-intent, we still do a docs search with some weight (unless extremely confident). So something relevant in docs should still appear albeit a bit lower. - We also tune thresholds to avoid aggressive gating when not sure. If the classifier is 60% code/40% docs, we'll treat that as mixed and not suppress docs results much. Only when it's, say, 90/10 would we heavily bias. - Multi-route with weights ensures even in a biased scenario, the secondary route can contribute if it has a very relevant result. E.g., if docs route finds an exact match phrase, even scaled down it might surface. - Additionally, after retrieval, if we see that one side returned nothing useful, we can pivot. For instance, if code was prioritized but top code results have super low scores and doc results have high, that indicates misrouting. We can incorporate a check: if the highest doc score is significantly above highest code score (after initial retrieval), then perhaps treat it as docs query after all. In SlideGar's adaptive retrieval, they merge results from feedback documents in a second round [22] – analogous to that, we can dynamically adjust. This is complex to

implement perfectly, but even a heuristic like "if code route top < threshold and docs top > threshold, switch weights" could help. We'll be cautious with that though to avoid flip-flopping incorrectly. - Through evaluation, we'll see if any docs query got worse. If so, maybe the heuristic was too aggressive. We'll refine pattern lists (maybe remove or soften a rule if it caused misrouting). - We will include in logs cases where classifier was uncertain or where one route had no results, so we can analyze and refine logic.

5. Embedding Model Upgrades and Drift:
If the embedding model (for dense retrieval) is changed or fine-tuned later, it could change similarity distributions. Our calibrated weights and intent detection (if it uses embedding signals) might become suboptimal. - *Mitigation:* Our pipeline is designed to be somewhat model-agnostic: RRF doesn't care about absolute values, cross-encoder re-evaluates actual content, LLM does the final sanity check. So even if embeddings change, the later stages catch a lot. But for best performance, we'd recalibrate. We can quickly re-run our evaluation with the new embeddings and see if performance changed. If so, we adjust e.g. `code_boost` or retrain the classifier on any new patterns (the queries wouldn't change, but maybe the classifier that uses text won't be affected. If it used embedding differences to detect intent (we haven't done that, but if we did), we'd need to retrain those thresholds). - In sum, if embedding model updates, schedule a mini-phase of re-tuning: - Redo the weight tuning (like Pinecone's λ search) on our validation set to see if we need to change the blend of dense vs lexical or code vs docs weight. - Check that the ranker (cross-encoder) still performing (it wouldn't be directly affected by embedding changes). - Keep the old model around for A/B test if needed to confirm improvements or catch regressions.

6. Increased Latency and Cost:
Each new component (BM25, graph, cross-encoder, LLM) adds processing. There's a risk the system becomes too slow for interactive use, or uses too much memory or GPU, etc. - *Mitigation:* We modularized such that heavy components can be disabled by config if a user is on limited hardware. For example, `enable_llm_rerank` default might be off, given it's heavy. Cross-encoder we choose a small fast model. - We will measure typical latency. Suppose baseline was 0.5s per query. With cross-encoder and graph, maybe it becomes 0.7s. We think that's okay for most. The LLM rerank might add a couple seconds if used – we'll emphasize that's optional and perhaps only for high-importance queries (maybe even triggered by user like "improve answer quality" button). - Memory: A 7B model for rerank is memory heavy. If a user doesn't have a second GPU, they might not be able to load it concurrently with main LLM. So again, likely off by default. Or if the main model can double as reranker via prompt (no separate load), we can do that to save memory at cost of some prompt tokens. - Another approach: pipeline triggers. For straightforward queries, skip cross-encoder to save time. We can set a threshold: if the top embedding score is extremely high and the query is short, maybe it's an easy direct hit – just serve it. Only do full rerank if needed. This is a future optimization if latency is an issue. - Also, we plan to use knowledge distillation or smaller models: e.g., RankVicuna shows a 7B can match GPT-3.5 ranker [3]. If we fine-tune a 3-7B model for reranking, that might run locally much faster than GPT-4 API and still improve over no reranker. We can explore that if needed. - For now, to ensure user experience, we'll keep user-facing note: if retrieval is slow, consider disabling X or upgrade hardware. In our internal environment, we'll profile each part.

7. Complex Graph Maintenance:
LLMC's graph artifact might need updating whenever code changes. If outdated, retrieval might link to things that no longer exist or miss new links. - *Mitigation:* That's more an ops issue: ensure `.llmc/rag_graph.json` is regenerated when indexing new code. Usually, LLMC likely generates it via a script. We should note in docs that for graph-augmented retrieval, up-to-date graph is crucial. - If graph is missing or fails to load (maybe some format issue), our code should detect and skip graph usage gracefully, logging a

warning. So a risk is a broken or old graph causing weird retrieval. But we have safety: if graph returns neighbors that don't exist in index (because code removed), we skip them. If it returns something irrelevant, cross-encoder likely will demote it. So the impact of an outdated graph is hopefully limited to maybe missing some new connections (performance not as good as could be, but not catastrophic). - We also might add a version check: if the graph's timestamp is much older than the file index timestamp, log that it might be stale. Up to the user to update, but at least warn.

8. Overfitting to Labeled Data (Distribution Mismatch):
Our evaluation queries might not cover all real user queries (there's risk we tune too much for them and not generalize). For instance, maybe our test queries are mostly function lookups, and we optimize for those, but users also ask architectural questions which we under-evaluated. - *Mitigation:* We will expand our labeled set progressively by sampling actual queries. We will try not to micro-optimize for small quirks unless they appear in real usage. That's where capturing real user questions for labeling is important; we avoid just theoretical ones. - Also, having diverse team members generate queries can reduce bias (some think of different tasks). - If possible, once the system is live, gather some random sample of usage (with permission) and evaluate performance qualitatively on them to ensure no blind spots. - The use of pairwise preferences (like checking new vs old on queries outside our initial set) will also guard that improvements hold broadly.

9. Known Hard Cases Not Solved:
There might be queries that require multi-hop reasoning (like "Which functions call X and then Y?") or extremely fuzzy matching that our system still can't get easily. These could still fail and be seen as retrieval failures or LLM hallucinations if LLM tries to answer without proper context. - *Mitigation:* We acknowledge that dynamic retrieval can't solve everything. For multi-hop, our graph helps some, but very complex logical queries might require query decomposition (which is beyond our current scope). If such queries arise, the risk is the assistant might give incomplete answer. - Our plan: identify these and perhaps handle at answer level with LLM reasoning. But retrieval-wise, as long as we return individually relevant pieces, the LLM might piece them together. We can also advise user to ask in smaller steps if needed. - Not really a "failure" of our pipeline but a limitation to be aware of.

10. User Confusion or Trust Issues:
If the system changes what it retrieves, users might notice differences. Ideally it's better, but there's a slight risk: say previously it always showed docs first, now it shows code – some users might not want code. However, if their query was clearly about implementation, code is correct. Possibly a user expected a more human-readable answer (docs) but got code which they find harder to parse. - *Mitigation:* This is more of a UX consideration. The assistant ultimately uses the retrieved snippets to answer in natural language, so even if we retrieve code, the LLM will explain it. So the user sees an answer referencing code (maybe citing the file). That should be fine as long as the answer is well-formed. If the user specifically didn't want code, they likely wouldn't ask how it's implemented. So probably not an issue. - We should ensure the LLM's final formatting of answer handles code context well (like maybe include the code snippet in markdown if relevant). But that's generation side. - If any user feedback suggests "I got too much code, I wanted an explanation", then it might be a sign of mis-identifying their intent. We'd revisit such cases and possibly refine the router or simply instruct the LLM to provide explanation even if code is given (the LLM can use the code to derive explanation). - Also, to maintain trust, we should avoid glaring mistakes in retrieval: our improvements aim to reduce irrelevancy, which should improve user trust (less hallucinated answers since context is correct). If any new error is introduced (like retrieving a wrong code piece that leads LLM to a wrong conclusion), that's a trust hit. That's why evaluation and careful adjustments are needed.

11. Integration Bugs:
As we modify many modules, there's risk of new bugs (e.g., graph index not loaded properly leading to exceptions, or concurrency issues if two threads use the cross-encoder, etc.). - *Mitigation:* Write unit tests for new functions (like graph_expand, fusion logic, classifier output). Also integration test manually with a variety of queries. We'll do a dry run with known answers to make sure pipeline flows. - Also keep config toggles such that if something breaks, user can turn it off easily as a workaround until fix. - We'll use try-except around non-critical parts (like if the cross-encoder fails for some reason, catch it and just proceed with original scores to still return something rather than crash). Log the error for us to fix, but user still gets an answer (maybe slightly worse but not nothing).

12. Cost of Human Labeling / Maintenance:
Our plan relies on continued human evaluation and tuning. If that effort lags, the models might become stale or misaligned with new features. - *Mitigation:* We will institutionalize the labeling as part of development cycles. Perhaps not a direct user risk but a project risk. Ensuring management is on board to allocate time for evaluation is key. Possibly automate parts (via LLM assistance) to reduce manual effort. - Always have at least a small gold set to quickly sanity-check any changes.

In conclusion, we've identified known challenges and put in place multiple safety nets: - Use robust fusion to handle scoring differences. - Keep human oversight in the loop to catch and correct issues like misrouting or irrelevant expansions. - Maintain toggles for features, allowing graceful degradation to simpler retrieval if something goes wrong. - Extensive testing on different query types to ensure no major category of query got worse.

By addressing these risks proactively, we aim to deliver a retrieval system that is not only more effective but also reliable and maintainable. Many of the mitigations (like RRF, dynamic gating calibration, etc.) are drawn from best practices and literature, giving confidence that we can manage these sharp edges [14] [22]. We will remain vigilant during deployment, using logging and user feedback to quickly spot any unforeseen failure modes and respond accordingly with config tweaks or patches. The modular design ensures we can isolate and roll back any one part without dismantling the whole system if needed, thereby limiting the blast radius of any single component's failure. Ultimately, these precautions will help us safely harness the advanced techniques for significant gains in retrieval performance.

Appendix

References

1. Han et al. 2025 – *"Retrieval-Augmented Generation with Graphs (GraphRAG)"*, arXiv 2501.00309. (Provides a survey of graph-enhanced RAG and examples of traversing knowledge graphs for multi-hop queries) [2] [36]

2. MacAvaney et al. 2022 – *"Graph-based Retrieval for Passage Ranking"* (in Graph Adaptive Reranking, yields ~8% nDCG improvements by reranking with candidate graph) [5]

3. Di Francesco et al. 2024 – *"Graph Neural Re-Ranking via Corpus Graph"* (demonstrated +5.8% AP using GNN reranker under 40ms, highlighting benefit of considering relationships among top docs) [5]

4. Bruch et al. 2023 (Pinecone) – *"An Analysis of Fusion Functions for Hybrid Retrieval"*, CIKM 2023. (Found that learned weighted fusion outperforms RRF slightly and that RRF, while robust, is sensitive to its parameter; suggests small training set can tune weights effectively) [17]

5. Rathee et al. 2025 – *"Guiding Retrieval using LLM-based Listwise Rankers"*, EMNLP 2025. (Introduces SlideGAR algorithm; improves nDCG@10 by up to 13% and recall by 28% by merging initial results with feedback from reranker in listwise LLM setting) [23]

6. Pradeep et al. 2023 – *"RankVicuna: Zero-Shot Listwise Document Reranking with Open-Source LLMs"* (Showed a fine-tuned 7B Vicuna model can achieve ranking quality on par with ChatGPT on TREC DL benchmarks, proving smaller local models can be effective for rerank) [3]

7. Liu et al. 2020 – *"GLOW: Global Weighted Self-Attention Network for Web Search Reranking"*, CIKM 2020. (Example of a residual re-ranking approach, relevant to mention that graph rerank and learned rankers can be combined as in MPGraf etc.) [37]

8. Gao et al. 2024 – *"LLM4Rerank: LLM-driven multi-objective reranking"* (Described in EmergentMind summary: uses LLM to iteratively refine ranking with prompts controlling aspects, demonstrating integration of LLM in ranking pipeline) [38] [39]

9. Yoon et al. 2025 – *"L2G: Learning to Rerank with document Graphs from Logs"*, under review 2025. (Proposed using co-occurrence in reranker outputs to build affinity graphs for propagation, achieving close-to-oracle performance at low cost) [1] [25]

10. Sachan et al. 2022 – *"Improving Passage Retrieval with Zero-Shot Rerankers"*, SIGIR 2022. (One of early works using GPT-3 for reranking, highlighting bounded recall problem and inspiring adaptive retrieval to expand result set) [22]

11. Liang et al. 2022 – *"Effective prompting for LLM-based relevance assessment"*, arXiv 2022. (Studied instructional prompts for ChatGPT reranking, found listwise output of permutations yields best results) [40] [6]

12. Sourcegraph (Hartman) 2025 – *"Lessons from Building AI Coding Assistants: Context Retrieval and Evaluation"*, RecSys'24 paper. (Industry experience: underscores difficulty of evaluating context relevance and necessity of custom metrics for code assistant) [41] [42]

13. RAGAS Documentation – 2023. (Defines metrics like ContextPrecision and ContextRecall to evaluate RAG pipeline: context precision = fraction of retrieved context that was actually used to answer, context recall = whether retrieved context contained the ground truth answer) [43] [33]

14. Turnbull 2015 – *"Relevant Search"*. (On judgment lists and graded relevance for search evaluation; reinforces need for clear rubrics and consistent judgments in IR eval) [44]

15. OpenAI 2023 – *OpenAI's evals (G-Eval)*. (Using GPT-4 as a judge for LLM outputs; demonstrates that LLMs can assist in evaluation when given a rubric, though caution needed) [20] .

(Note: The reference numbers above correspond to in-text citations like [2]. Those citations point to supporting lines in the source material provided via the browsing tool, which substantiate the statements.)

Key Papers and Contributions (Summary Table):

Technique/ Paper	Contribution	Integration Insight
Han et al. (GraphRAG Survey, 2025)	Survey of Graph-enhanced RAG, holistic framework and domain-specific challenges [2]. Emphasizes graph traversal and GNNs capturing relational knowledge.	Use GraphRAG patterns: e.g. BFS on code graph for multi-hop questions; ensure retriever & generator adapt to structure.
MacAvaney et al. (GAR, 2022)	Graph-based reranker improves BM25→T5 by +8% nDCG [5]. Constructs kNN graph of candidates and propagates scores.	Validates adding graph relationships among candidates improves ranking. Implement lightweight propagation among top results.
Di Francesco et al. (GNRR, 2024)	GNN re-ranking via corpus graph, +5.8% AP in 40ms [5]. Uses a learned GAT on candidate graph.	Shows a learned graph model yields gains quickly. Suggests even a static graph algorithm might help if model not available.
Pinecone (Bruch et al., 2023)	Comparison of hybrid fusion methods [17]. Found tuned weighted sum outperforms RRF slightly; RRF still good default.	We will implement RRF first, and consider simple weight tuning via a small labeled set to fine-tune if possible.
Rathee et al. (Adaptive LLM Rerank, 2025)	Solves bounded recall: merges initial hits with similar docs of top ones using a graph (corpus kNN) [22]. Achieved large recall jump with minimal LLM overhead.	Inspiration for graph expansion: precompute similarity graph offline, then during query, pull neighbors of top docs to improve recall. We'll do similar via code graph.
Pradeep et al. (RankVicuna, 2023)	Open 7B LLM matched ChatGPT for rerank (zero-shot) [3]. Demonstrates local models can suffice for reranking with proper fine-tune.	We can aim to fine-tune a local 7B (if needed) for rerank or trust that using one zero-shot (with good prompting) will be effective.
Sourcegraph (Hartman, 2025)	Noted lack of ground truth and need for multi-faceted eval for coding assistant [42]. Used small internal sets + open data; context relevant if it leads to correct answer.	Confirms our plan to craft a custom evaluation (Code@k, context recall) is necessary. Also warns that final answer quality is ultimate measure, so tie retrieval eval to that where possible.

Technique/ Paper	Contribution	Integration Insight
RAGAS Metrics (2023)	Defines context precision/recall, answer faithfulness [43] [33]. Integrates with LLM eval for answers.	We'll adopt context precision/recall to measure how well our retrieved snippets cover the needed info, and watch answer correctness (maybe via separate QA check).
OpenAI Evals (G-Eval, 2023)	Using LLMs to automate eval with rubrics [20]. LLMs can provide initial relevance judgments which humans refine.	We will try using GPT-4 to pre-label some query-doc pairs to save time, but will validate carefully.

Pseudocode: Dynamic Multi-Stage Retrieval

```
# Pseudocode for search_spans with dynamic multi-scoring
def search_spans(query):
    # 1. Intent routing
    route = deterministic_router.route(query)
    # route might contain route_weights if multi-route
    routes_to_search = route.routes  # e.g. ['code','docs']
    weights = route.weights          # e.g. {'code':1.0,'docs':0.5}

    # 2. Retrieve initial candidates
    results = {}  # {route: [(span, base_score), ...]}
    for r in routes_to_search:
        if r == 'code':
            results[r] = dense_search(index_code, query, top_k = K1)
        elif r == 'docs':
            results[r] = dense_search(index_docs, query, top_k = K1)
    if config.enable_lexical:
        lex_results = bm25_search(corpus, query, top_k = K2)
        results['bm25'] = lex_results

    # 3. Initial fusion of results
    all_candidates = fuse_results(results, mode=config.fusion.mode,
weights=weights)
    # all_candidates: list of (span, fused_score)

    # 4. Graph-based augmentation
    if config.graph.enable_expansion:
        neighbors = []
        for span, score in all_candidates[:config.graph.augment_top_n]:
            for neigh in graph_index.get_neighbors(span.file,
types=config.graph.relations):
                if neigh not in [cand.file for cand in all_candidates]:
```

```
# fetch a representative span from neigh (e.g. first span or one containing
reference if known)
                    neigh_span = database.get_span(neigh, 0)
                    # compute neighbor relevance filter
                    if is_semantically_related(query, neigh_span):
                        neigh_score = score * config.graph.neighbor_weight
                        neighbors.append((neigh_span, neigh_score))
        all_candidates.extend(neighbors)

    # 5. Scoring adjustments (e.g. code boost, file name boost)
    for span, score in all_candidates:
        if span.file_extension in config.scoring.code_extensions:
            score *= config.scoring.code_boost
        if span.file in filename_matches(query):
            score += config.scoring.filename_bonus
        # Graph degree prior
        if config.graph.enable_degree_boost:
            deg = graph_index.degree.get(span.file, 0)
            score += config.graph.degree_beta * log(1+deg)
        span.score = score

    # 6. Second-stage Reranking
    rerank_candidates = sorted(all_candidates, key=lambda x: x[1], reverse=True)
[:config.rerank.top_n]
    if config.rerank.enable_crossencoder:
        model = crossencoder_model  # loaded globally
        for span, base_score in rerank_candidates:
            features = tokenizer.encode(query, span.text)
            ce_score = model.predict(features)
            span.rerank_score = ce_score
        rerank_candidates.sort(key=lambda x: x[0].rerank_score, reverse=True)
    if config.rerank.enable_llm:
        llm_list = rerank_candidates[:config.rerank.llm_top_n]
        ranked_order = call_llm_rerank(query, llm_list)
        rerank_candidates = ranked_order + [c for c in rerank_candidates if c
not in ranked_order]

    final_candidates = rerank_candidates  # already sorted

    # 7. Select top spans for output (consider cut-off by score or length)
    output_spans = []
    token_budget = config.output.max_tokens
    for span, score in final_candidates:
        if token_budget <= 0: break
        output_spans.append(span)
        token_budget -= len(tokenizer.tokenize(span.text))
        # if token_budget small, maybe break early or drop low relevances
    return output_spans  # these are fed to LLM as context
```

(The above pseudocode is a simplified sketch – actual implementation will handle data structures and integration with LLMC's classes.)

Benchmark & Dataset Suggestions:

- **Internal Repo Queries:** Our own curated set as discussed, derived from actual usage patterns.
- **Public Code QA datasets:** e.g., *CodeSearchNet* (has function descriptions and code, but that's code-to-doc retrieval), or *Stack Overflow Q&A* for natural language queries and code answers. We can adapt a few to test generalizability.
- **Knowledge Base QA for Graph testing:** e.g., an academic KB query (like the example in GraphRAG paper about drugs and genes) [2] . Not directly our domain, but we could simulate with our graph (like "Which module calls function X that triggers Y?" to test multi-hop).
- **TREC Deep Learning 2019 (MS MARCO):** not code, but we could use a subset to ensure our reranking methods are on par with known results (for sanity check of cross-encoder and LLM rerank performance).
- Ultimately, focus on our domain is more relevant, but cross-checking with a known IR dataset with metrics can validate that, e.g., our cross-encoder usage is correct (should reproduce known gains on MS MARCO dev set for instance).

Additional Tables:

Top 10 Promising Techniques vs LLMC Fit:

Technique	LLMC Modules Affected	Config Keys Added	Fallback if disabled
Graph Neighbor Expansion (GraphRAG)	`rag_graph.json` via `graph_index.py`, integration in `search_spans`	`graph.enable_expansion`, `graph.neighbor_weight`, `graph.augment_top_n`	System just uses initial retrieved results without graph neighbors (slightly lower recall for code).
Graph-Based Rerank (PageRank or GNN)	Possibly new `rerank_graph.py` or inside `rerank.py`	`rerank.graph_enable`	If off, default to scoring without graph propagation (just cross-encoder/LLM).

Technique	LLMC Modules Affected	Config Keys Added	Fallback if disabled
Reciprocal Rank Fusion (Hybrid)	`fusion.py`	`fusion.mode` (set to 'rrf' or 'weighted')	If off (mode 'max'), uses original max score fusion (could revert if needed).
LLM Listwise Rerank (RankGPT-style)	`rerank.py` calling LLM	`rerank.enable_llm`, `rerank.llm_top_n`	If disabled, rely on cross-encoder or initial rank. LLM not invoked.
Cross-encoder Pointwise Rerank	`rerank.py` with model loaded	`rerank.enable_crossencoder`, `rerank.model_name`	If off, skip directly to LLM or final fusion.
Intent Classification (Router)	`router.py`, possibly `intent_model.py`	`routing.enable_intent_model`	If off, uses static multi-route weights from config (like old behavior).
Dynamic Multi-Route Gating	`router.py` (logic within)	(Same keys as above; uses thresholds)	If disabled, routes are used equally (or as per basic config).
Human-in-loop Feedback (not code)	(external process, influences retraining of models and config)	N/A (maybe logging toggles)	If not utilized, risk of stale tuning – no immediate fallback, but system still runs with last known tuning.
Active Learning (process)	(primarily external/ offline)	N/A	Without it, models might not improve or adapt – essentially freeze in time.

Technique	LLMC Modules Affected	Config Keys Added	Fallback if disabled
Rank Fusion Learned Weight (K²)	`fusion.py` (weighted mode)	`fusion.weight_code`, etc.	If disabled or no data, can use RRF or equal weights as safe default.

Each can be independently turned off. In our final config, we'll likely enable many by default except maybe LLM rerank for resource reasons. But if any cause issues, users/devs can disable that feature without breaking others.

Final Thoughts:

This plan aligns with LLMC's architecture by plugging into key extension points (routing, scoring, reranking) without major refactoring of core logic. By evaluating each addition thoroughly, we reduce the risk of negative side-effects. The result will be a more **intent-aware, graph-savvy retrieval system** that better serves code assistant use-cases, confirmed by both **quantitative metrics** (improved Code@k, nDCG) and **human judgment** (annotators and users seeing more relevant code when needed and no loss for docs queries).

[1] [4] [5] [13] [14] [15] [25] [26] [37] [38] [39] Graph-Based Re-Ranking Strategy
https://www.emergentmind.com/topics/graph-based-re-ranking-strategy

[2] [7] [8] [36] [2501.00309] Retrieval-Augmented Generation with Graphs (GraphRAG)
https://ar5iv.labs.arxiv.org/html/2501.00309v2

[3] [2309.15088] RankVicuna: Zero-Shot Listwise Document Reranking with Open-Source Large Language Models
https://arxiv.org/abs/2309.15088

[6] [35] [40] aclanthology.org
https://aclanthology.org/2023.emnlp-main.923.pdf

[9] [10] [11] [12] [21] [22] [23] [24] [28] Guiding Retrieval using LLM-based Listwise Rankers
https://arxiv.org/html/2501.09186v1

[16] [17] An Analysis of Fusion Functions for Hybrid Retrieval | Pinecone
https://www.pinecone.io/research/an-analysis-of-fusion-functions-for-hybrid-retrieval/

[18] [19] RouterRetriever: Routing over a Mixture of Expert Embedding Models
https://arxiv.org/html/2409.02685v2

[20] [34] A Comparison of LLM-based Relevance Assessment Methods - arXiv
https://arxiv.org/html/2504.12558v1

[27] [31] [32] Mastering Hybrid Retrieval Strategies in RAG: Combining the Best of ...
https://medium.com/@sahin.samia/mastering-hybrid-retrieval-strategies-in-rag-combining-the-best-of-multiple-approaches-3153caa18b01

[29] Label shift conditioned hybrid querying for deep active learning
https://www.sciencedirect.com/science/article/abs/pii/S0950705123003660

[30] Metrics - Ragas
https://docs.ragas.io/en/v0.1.21/concepts/metrics/

[33] RAG Evaluation Metrics: Best Practices for Evaluating RAG Systems
https://www.patronus.ai/llm-testing/rag-evaluation-metrics

[41] [42] Lessons from Building AI Coding Assistants: Context Retrieval and Evaluation | Sourcegraph Blog
https://sourcegraph.com/blog/lessons-from-building-ai-coding-assistants-context-retrieval-and-evaluation

[43] Context Precision - Ragas
https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/context_precision/

[44] Cracking the code on search quality: The role of judgment lists - Elastic
https://www.elastic.co/search-labs/blog/judgment-lists