

Executive Summary (Key Findings and Recommendations)

Optimizing recall in enterprise **Retrieval-Augmented Generation (RAG)** systems requires improvements across data chunking, retrieval algorithms, embeddings, context handling, and infrastructure. **Advanced chunking strategies** like semantic or proposition-based chunking yield more meaningful text segments than naive fixed-size chunks, boosting retrieval recall by up to ~5–9% in benchmarks. However, these come with higher computational cost (e.g. embedding every sentence or invoking LLMs for chunking). In production, start with reliable **recursive or semantic chunking**, and consider agentic (LLM-guided) chunking for critical documents despite its latency/cost.

On the retrieval side, **hybrid architectures** that combine dense vector search with sparse keyword matching significantly improve recall by covering both semantic and exact term matches. Leading companies use multi-stage pipelines: an initial dense+sparse retrieval fused via rank merging (e.g. RRF), followed by a **re-ranking** stage (cross-encoder or even an LLM) to boost precision. This approach improved top-5 recall by ~26% in one case, albeit with ~3× higher latency. **Embedding model choice** is critical: newer models like OpenAI's *text-embedding-3* or Cohere v3 outperform the popular Ada-002 by large margins on recall benchmarks. For instance, OpenAI's 3-small model achieved ~44% vs Ada-002's 31% recall on a multilingual test. Fine-tuning open-source embeddings on domain data can further lift recall of domain-specific content (e.g. FinBERT for financial texts improved accuracy on finance queries). **Context window utilization** is another key: with 100K-token models now available, one can supply more documents per query, but must mitigate the "lost-in-the-middle" effect where models underweight middle context. We recommend ordering retrieved chunks by relevance and importance so critical facts appear early in the prompt.

Advanced patterns like **multi-hop retrieval** (iteratively retrieving for complex queries) and **self-reflective RAG** (the LLM critiques and refetches missing info) are emerging to maximize recall on complex questions. Rich metadata (document type, timestamps, topics) should be leveraged for filtering and query enrichment to narrow search space and isolate tenants in multi-tenant deployments. In evaluation, go beyond Recall@K – use metrics like **MRR and NDCG** to capture ranking quality, and build synthetic query sets with LLMs to quantitatively stress-test recall. In production, tackle document updates with incremental indexing and use similarity-based de-duplication to avoid redundant entries. **Scalability** solutions (ANN indexes, sharding, caching) will ensure recall doesn't drop off as you scale to millions of documents. Finally, recent RAG research (2024–2025) shows >10% gains from techniques like agentic retrieval and graph-augmented search, pointing to future systems that combine text and knowledge graphs for even higher recall.

Recommendations: Begin with a hybrid dense+BM25 retrieval baseline using semantic chunking and a strong embedding model (consider OpenAI or a fine-tuned open-source model). Layer on a lightweight re-ranker (e.g. MiniLM cross-encoder) to boost precision without extreme latency. Incorporate metadata-based filters for scope (e.g. by document type or date) to reduce false negatives. Continuously evaluate recall on realistic queries – including multi-facet questions – and iterate with advanced methods (query reformulation like HyDE, neighbor chunk expansion, multi-hop agent strategies) as needed. This phased approach will

yield an enterprise RAG system with high recall of relevant knowledge, robust to growing content and query complexity.

1. Advanced Chunking Strategies Beyond Basics

Proper document **chunking** is foundational for high recall: chunks must be large enough to contain complete facts, yet small enough to stay relevant. Beyond basic fixed-length splitting, new strategies aim to preserve semantic integrity:

- **Semantic Chunking:** Uses embedding similarity to group sentences by topic rather than by raw length. This approach detects topic shifts (e.g. a sudden drop in similarity between consecutive sentences) and splits there, yielding coherent topical chunks. It often produces fewer, larger chunks (of varying sizes) – in one example, only 8 semantically-coherent chunks covered a document, some up to 8k characters. The benefit is **higher recall**: Chunks are more likely to fully answer a query if relevant. A 2024 study found semantic chunking improved retrieval recall by ~2–3 percentage points over standard recursive splitting, and up to 9 points in certain cases. The trade-off is **cost** – every sentence needs embedding, meaning hundreds of model inferences per document. It also requires tuning a similarity threshold (percentile, std dev, etc.) to decide splits. Semantic chunking excels for dense text (research papers, technical docs) where natural section breaks are absent or subtle. When recall is top priority and budget allows, this method yields the most context-complete chunks.
- **Proposition-Based Chunking (Agentic Chunking):** This two-step LLM-driven approach breaks text into atomic propositions (standalone factual statements), then uses an agent (LLM) to **regroup** these propositions into chunks. The idea, introduced in 2024 by FullStackRetrieval et al., is that each proposition expresses one fact, and the LLM can decide which facts belong together in a chunk. This retains fine-grained semantic units without losing context. **Pros:** Extremely high semantic coherence – each chunk may represent a self-contained idea or subtopic, which can boost answer recall and reduce irrelevant text. In experiments, LLM-guided proposition grouping achieved the highest accuracy (chunks yielding 91.9% recall in one benchmark with an LLM-enhanced variant). **Cons:** It's expensive and complex. It requires crafting prompts for the LLM to extract propositions and cluster them, and results depend on LLM quality. There is also latency overhead since chunking each document involves one or multiple LLM calls. Use this for critical knowledge bases where maximum recall is worth the cost – e.g. policy documents where missing a clause has severe consequences.
- **Graph-Based Chunking:** Instead of linear text splitting, this approach builds a **graph of concepts or references** within the document/corpus and defines chunks via graph structures. For example, one might connect sentences or paragraphs by their semantic relatedness or hyperlinks, then extract subgraphs as chunks. This is closely tied to **Graph RAG** paradigms where documents are seen as nodes in a knowledge graph ¹. Graph-based chunking preserves relationships (like “this paragraph cites that figure/table” or “concept X appears in these three places”). It can improve recall for data that is inherently graph-structured (e.g. documentation with many cross-references, or FAQ answers linked to product pages). **Trade-offs:** Graph chunking can capture global context better (preventing “context fragmentation” across chunks), but constructing and traversing the graph adds complexity. Also, not all content is easily represented as a graph unless explicit links or ontology exist. This is an emerging area – research prototypes (e.g. GraphRAG 2024) show improved multi-hop question answering by using graph-structured chunks ¹, but there isn't yet a one-size-fits-all toolkit.

Consider graph chunking for content like procedure manuals or academic papers, where sections naturally link; it may yield chunks that ensure all related info is retrieved together.

- **“Agentic” Adaptive Chunking:** A general approach where an AI agent dynamically chooses a chunking strategy per document. For instance, trivial formats (like short emails) might just use fixed split, whereas a long legal contract triggers a semantic+proposition strategy. The agent analyzes the document structure and content, possibly via a prompt, and selects an optimal splitting plan. In practice, this could mean using an LLM to determine section boundaries or to recursively split difficult passages. The advantage is flexibility – it attempts to get “the best of all worlds” by tailoring chunking to each document’s nature. However, it inherits the downsides of LLM calls (cost, variability) and adds engineering overhead to implement the decision logic. Use cases would be heterogeneous corpora (e.g. a mix of code, PDFs, plain text) where no single chunking method works universally.

Chunk Size & Overlap Considerations: Regardless of method, chunk size is a key lever for recall vs precision. Smaller chunks (e.g. 50–100 words) mean more pieces to index, which increases chance that one piece exactly matches a query (high recall), but they risk losing context that might be needed to fully answer the query (lower precision if the model only sees a fragment) ² ³ . Larger chunks (500+ words) carry more context (good precision) but fewer of them will exactly hit the query terms (risking recall loss). A **balanced approach** is often ~200–300 words with 10–15% overlap, or ~800–1000 characters with ~100 char overlap, as a starting point. Overlap ensures if an important sentence lies at a boundary, it appears in both neighboring chunks. Empirically, overlapping chunks can improve answer recall by ensuring context isn’t split up irrecoverably. One 2024 NVIDIA benchmark found an interesting sweet spot: for PDF documents, **page-level chunks** (each page as one chunk) gave the highest accuracy (0.648) and consistency – essentially because each page kept its full context (especially helpful for tables/figures that span a page). But this only applies when page breaks coincide with semantic units (e.g. slides, forms). In general, it’s wise to evaluate a few chunk sizes/strategies on a validation set: measure if answers are found in retrieved chunks. Many RAG engineers start with **LangChain’s RecursiveCharacterTextSplitter** (which tries sensible breakpoints like newlines, sentences, etc.) as a baseline, then graduate to semantic or LLM-based chunking if recall is insufficient. Remember that **chunking is an “offline” cost** – you pay for it once during indexing – so investing in a better chunking strategy (even if computationally heavy) can pay off with permanently better retrieval performance.

Best Strategies by Content Type: Different content benefits from different chunking tactics:

- *Technical documentation* (with sections, code snippets, etc.): Use **semantic or document-structure chunking**. Preserve section headers and code blocks as atomic units. Including section titles in the chunk text or metadata is recommended so that context isn’t lost (e.g. a chunk of code gets the heading “Initialization Function” prepended). Semantic grouping can merge small sub-sections that are contextually related. If the docs are in Markdown or have clear headings, leverage that (many frameworks have Markdown/HTML splitters that keep content under each `<h2>` together). This ensures that, say, a function description and its parameter table end up in one chunk for retrieval.
- *Code repositories:* Treat code differently from prose. A logical chunk is often a function or class definition. Tools like *LangChain’s PythonCodeTextSplitter* split code by syntax (function boundaries). This is crucial for recall – if a query asks “What does function X do?”, retrieving the entire function body chunk is ideal. Also, code tokens are dense; embedding models may have limited context window for embeddings (many models cap at 512 tokens for embedding generation). In such cases, **hierarchical chunking** helps: embed each function, but also perhaps embed file-level summaries. This is akin to Matryoshka embeddings (discussed later) where you have nested representations. For code, ensure overlapping context (include

import statements or class context in each chunk) to not lose needed info.

- *Time-series data or structured logs*: This is tricky – chunking might not apply if data is not textual. If you have time-stamped text (like support chat logs, chronological events), consider **windowing by time** (e.g. one day or one conversation per chunk) or by event count. Graph-based chunking could be relevant (e.g. linking events that are causally related). But often, for logs, better to preprocess into a summary or anomaly events and chunk those. If it's textual time-series (like a weekly report archive), you may chunk by each report but ensure the date is in metadata for temporal filtering.

- *Multi-modal content*: If documents include images or figures with captions, chunking should keep references with text. One strategy: use an OCR or caption extraction on images and insert that text into the document before chunking. Alternatively, for PDFs with images, use page-level chunking (so images+captions stay with surrounding text). If dealing with truly multi-modal embedding (e.g. using CLIP for images and text), you might create separate vector indices but that goes beyond chunking. Generally, chunk multi-modal docs in a way that *textual description of visuals stays in the same chunk as the visual*. This might mean one chunk per figure+paragraph. Some pipelines generate a textual summary of images (using GPT-4 Vision or similar) and append it to the text before embedding. This ensures that queries about an image can retrieve the chunk via the generated text.

In summary, **chunking impacts recall** by determining what the retriever can “grab” as the atomic unit. Advanced chunking (semantic, agentic, structural) yields chunks that better align with how questions are asked, thereby increasing the likelihood that a single chunk fully contains the answer. The trade-off is always between *recall vs. cost/latency*: more sophisticated chunking may use more preprocessing time and compute. Our recommendation is to iterate: start simple (e.g. 300-word chunks with overlap), evaluate recall on a sample of queries, then introduce semantic or LLM-driven chunking for documents where you observe failures (e.g. answers split across two chunks frequently). This targeted adoption maximizes benefit while controlling cost.

2. Retrieval Architecture Deep Dive

Modern high-recall RAG retrieval pipelines are typically **multi-stage and hybrid**, combining different retrieval methods and reranking to capture as many relevant results as possible:

- **Hybrid Sparse+Dense Search**: It's widely proven that fusing lexical search (e.g. BM25 or keyword TF-IDF) with semantic vector search improves recall. Sparse retrieval excels at exact matches (names, error codes, jargon) that a dense model might miss, while dense retrieval finds paraphrased or conceptually related text. Many enterprise systems implement hybrid search via **Reciprocal Rank Fusion (RRF)** or similar merging of two result lists. For example, a pipeline might query both a PGVector index (with embeddings) and an Elasticsearch index (with BM25), then merge results so that a document highly ranked in either gets a boost. This approach is robust and easy to tune (RRF has one parameter usually). In one report, a hybrid RRF baseline significantly outperformed using dense or sparse alone in early precision. The recall gains come especially for “mismatch” cases – e.g. user uses a different term than the document. Dense can bridge synonyms in those cases, whereas lexical catches exact terms like proper nouns or code references that dense might fail to encode. **Trade-offs**: Running two searches has slight overhead, but for moderate corpus sizes (tens of thousands of docs), this is usually tens of milliseconds extra – well worth the recall boost. For very large corpora, more complex fusion like *SPLADE* (which generates sparse “expanded” vectors from a transformer, combining aspects of both approaches) might be used ⁴, but RRF is a solid starting

point. The key is to carefully *normalize scores* from different systems (embedding distances vs BM25 scores are not directly comparable; RRF sidesteps this by using rank positions).

- **Multi-Vector Retrieval (ColBERT and Late Interaction):** Traditional dense retrieval uses a single vector per chunk. Multi-vector approaches store multiple embeddings per document to capture different aspects or sub-components. **ColBERT** (Khattab & Zaharia 2020) is a prime example – each token (or each meaningful token) in a passage is embedded, and at query time a BERT-based late interaction scoring computes relevance. This yields very fine-grained matching (the model can, for instance, find that a query token “Washington” aligns with a particular token in the document embedding that corresponds to “Washington (state)” while other parts of the query align elsewhere). The result is typically **higher recall and precision** on granular QA tasks – ColBERT v2 has shown state-of-the-art results on MSMARCO passage retrieval with significantly improved recall over single-vector bi-encoders, while also compressing index size. In RAG, ColBERT can ensure that even if the relevant sentence is a small part of a chunk, the token-level interaction can surface that chunk. **Cons:** Multi-vector methods greatly increase index size and retrieval complexity. Instead of one 768-d vector per chunk, you might store 100+ smaller vectors. Many vector databases do not natively support multi-vector querying (though workarounds exist). There’s also a latency hit: late interaction means the query has to compare against every token’s vector of candidate passages (ColBERT uses clever pruning and GPU acceleration to mitigate this). In practice, multi-vector retrieval is used when **relevance at a very fine scale** is required – e.g. precise FAQ matching where a single word mismatch matters. For enterprise wikis or code search, multi-vector can drastically improve exactness, but ensure your infrastructure (and budget) can handle the bigger index. A compromise is **multiple representation indexing**, where you store a few vectors per doc, each representing a different facet (like one for title, one for body, one for conclusion). This is lighter than full ColBERT but still covers more angles than a single embedding.
- **Query Transformation Techniques:** These aim to bridge the gap between the user’s query wording and the documents’ wording, thereby retrieving relevant docs that a direct query might miss. One popular method is **Hypothetical Document Embeddings (HyDE)**. With HyDE, you prompt an LLM (e.g. GPT-3.5 or 4) with the user question to generate a *fake document or answer* to that question. That generated text (which hopefully uses terms and context similar to what a correct answer would) is then embedded and used as the query vector to find real documents. Essentially, the LLM “imagines” the answer, which expands the query’s semantics, and the vector search uses that to grab actual supporting passages. HyDE has been shown to outperform standard query embedding in zero-shot scenarios, often increasing recall substantially when the original query is short or under-specified. Another approach is **multi-query generation** – generate several rephrasings or expansions of the query (diverse phrasings, or sub-questions for each aspect of a complex query). Then perform retrieval for each and merge results. This can catch different relevant documents that a single phrasing might not. Research like *DMQR (Diverse Multi-Query Rewriting) 2024* has shown gains in retrieval performance by covering multiple query variants simultaneously. The downside is obvious: more queries mean more load on the system and more results to merge – but for high-stakes queries, this can be worth it. One must also have a strategy to *combine* the results (e.g. union them or take top from each), and to avoid just getting the same document multiple times. In practice, multi-query gen can be triggered only for certain queries (like those identified as complex or ambiguous). Simpler query rewriting includes synonym expansion (e.g. add common synonyms or acronyms of query terms as OR clauses in a sparse search) and **Query autorewriting** using user behavior (if many users reformulate a particular query, use those learnings). Another notable

technique: **LLM-based query expansion** where you ask an LLM: “Rephrase this question using terminology from the documentation.” If you have domain-specific vocabulary, this can dramatically increase recall by aligning the query to the corpus vocabulary.

- **Re-ranking Architectures:** After an initial retrieval (often called the candidate generation stage), a re-ranker can refine the ordering of results to improve precision without losing recall. Two common re-rankers are **cross-encoders** and **LLM-based rankers**. A cross-encoder, such as *MS MARCO MiniLM cross-encoder*, takes a query and a candidate passage together as input and produces a relevance score. It “sees” the actual text of both, allowing it to catch nuances that bi-encoder embeddings might miss. Adding a cross-encoder to re-rank top 20 or 50 results can significantly boost precision@K and recall@K (since some relevant items that were ranked lower by the first stage get moved up). In Yash Mhaskar’s 2025 study, adding a MiniLM cross-encoder to re-rank the top 20 hybrid results resolved many lexical mismatch issues and improved Precision@5 by 29% (from 0.40 to ~0.517) and Recall@5 by 26%. The key is that the cross-encoder can align the query with passages in a more fine-grained way, effectively “rescoring” relevance. The cost is additional latency – cross-encoders are BERT models that might add tens or hundreds of milliseconds per query (though small ones like MiniLM are quite fast, and batching can be used). **LLM-as-reranker:** Here you would prompt an LLM (like GPT-4) with the query and a list of retrieved snippets, asking it to judge which snippets are most relevant or to pick the best K. This can capture even deeper understanding (LLMs might use world knowledge plus the snippet to infer relevance), but it’s expensive and slow (hundreds of tokens of input to a large model). It could be used offline to periodically evaluate or fine-tune other re-rankers, rather than in real-time. Another re-rank approach is **learn-to-rank (LTR)** using gradient boosted trees or neural networks on features. For instance, one could train an XGBoost model to rank results using features like BM25 score, embedding cosine similarity, document popularity, and query-document metadata overlap. Traditional LTR can incorporate business logic (like prefer more recent docs). If labeled data (query -> relevant doc pairs) is available, an LTR model can improve ordering and recall by learning patterns beyond pure semantic similarity. However, LTR requires maintenance and may not generalize beyond the training set domain. In summary, re-ranking is essential when you care about the **quality of the top 3-5 results**, which in RAG you do (because an LLM can only ingest so many). It’s common to retrieve, say, top 50 with a high-recall method, then re-rank to ensure the truly relevant 5 are at the top. This dramatically improves the chance that the final selected context contains the answer (improving effective recall from the LLM’s perspective). Re-ranking does add latency: in one pipeline, adding cross-encoder and neighbor expansion made queries ~3.5 seconds slower (3.2× slower than baseline). Teams mitigate this by using faster rerankers (e.g. knowledge distillation to a smaller model like **MiniLM** or even a one-layer transformer called “FlashRank”), caching frequent query results, or doing reranking asynchronously. Depending on throughput needs, you might apply rerank only when the initial confidence is low. But for most enterprise QA, a ~2-3 second retrieval stage is acceptable if it ensures the LLM gets the right info.

Other architectural considerations include **neighbor expansion** (as used by Yash’s pipeline): attaching the adjacent chunks of any top result to provide local context. This can improve recall of multi-paragraph answers when relevant content spans chunk boundaries. In his pipeline, each retrieved chunk brought in its immediate neighbors with a decayed weight, and during reranking the model considered $(\text{previous} + \text{current} + \text{next})$ chunk as one context. This yielded better recall for answers split across chunks. The cost is more tokens per candidate and some complexity in deduplication (ensuring a neighbor that was

itself retrieved separately isn't double-counted). Nonetheless, it's a handy trick to counter chunk fragmentation issues.

Finally, note that **retrieval doesn't always have to be one-shot**. Some systems iterate: initial retrieval -> partial answer -> retrieve again for missing pieces. We'll cover this under multi-hop, but architecturally it means designing your system to allow an LLM to call the retriever multiple times (which can be done via tools or an Agent loop in frameworks like LangChain). Architectures that incorporate feedback loops tend to achieve higher recall on complex queries at the cost of more compute.

Latency vs Recall: It's worth summarizing numbers: A simple dense retrieval (embedding query + ANN search) can be as low as 50ms. Adding BM25 might add 20ms. Merging results negligible. A cross-encoder on top 20 might add 50–200ms (depending on model and hardware). So a well-optimized hybrid pipeline can still be sub-second. However, if you start adding GPT-4 rerankers or multi-hop with multiple LLM calls, latency can jump to several seconds. The good news is that improving recall of the *top few* results has an outsized benefit – as noted by Mhaskar, improving early precision/recall helped with the LLM's lost-in-the-middle issue. They also found recall gains plateaued after $k=10$ – meaning the main gains were in the very top ranks. This suggests a strategy: **spend compute to get the top-5 as relevant as possible (via rerank, etc.), rather than naively retrieving 100 docs hoping the answer is somewhere**. A small number of high-quality retrieved chunks is better for downstream LLM performance than a flood of somewhat-relevant chunks.

3. Embedding Strategy Optimization

The choice and usage of **embedding models** directly affects recall, as it determines how well semantic similarity correlates with “relevance.” Key considerations are model selection, fine-tuning, dimensionality, and special techniques for hierarchical or multilingual data:

- **Comparing Embedding Models:** As of 2024–2025, there are many options beyond OpenAI's ubiquitous `text-embedding-ada-002`. In fact, newer models significantly outperform Ada-002 on retrieval benchmarks. OpenAI's latest text-embedding-3 series (available in small and large variants) shows notable gains: e.g. `text-embedding-3-large` improved multilingual retrieval by 23.5% and English retrieval by ~3.6% over Ada. The small variant also outperforms Ada; one report cites ~44% vs 31% recall on a MIRACL multilingual test (small model vs Ada). Cohere's `embed-english-v3.0` and `embed-multilingual-v3.0` are strong contenders too – for instance, Cohere's multilingual v3 scored 64.0 on MTEB (Massive Text Embedding Benchmark) and 54.6 on BEIR, which are state-of-the-art levels. Meanwhile, open-source models like **E5** (by Intfloat) offer surprisingly good performance – `e5-base-v2` (which is a ~110M param model) often matches or beats Ada on retrieval tasks⁵. According to Pinecone's tests, many proprietary models now “far outperform Ada” in accuracy. So if recall is paramount, one should strongly consider these newer models. For example, OpenAI's `text-embedding-3-large` has up to 3072 dimensions (Ada was 1536) and captures more information per vector. It costs more (and has a bigger index footprint), but ideal when you need the highest quality. If cost is a concern, OpenAI's `text-embedding-3-small` is 5× cheaper than Ada-002 per token and still more accurate – a likely “sweet spot” for many. Cohere's v3 is another excellent choice; it supports up to 4096-token inputs for long texts and had competitive performance. Also notable is **Google's Gecko** embedding (released via Vertex AI) – it's distilled from a large model and reportedly achieves SOTA retrieval scores, but it's newer and slightly less accessible in tooling. When comparing, consider dimensions (higher dims may give better

separation but at cost of storage/query speed), model size (impacts encoding throughput), and any specialization (some models are better at short text vs long documents). **Benchmark results (MTEB, BEIR)** are good starting points but should be taken with a grain of salt, as some open models might be overfitted to those benchmarks ⁶. Ultimately, test on a subset of your data if possible.

- **Fine-Tuning and Domain Adaptation:** If your enterprise content is domain-specific (legal contracts, medical texts, programming code), fine-tuning an embedding model can markedly improve recall for that domain. OpenAI and Cohere's hosted models currently *do not allow user fine-tuning* of embeddings (OpenAI only allows some projection fine-tuning to reduce dimensions). So for domain adaptation, open-source is the way. Techniques include: fine-tuning a model like SBERT or E5 on a dataset of Q&A or search click data from your domain, or training with **Generalized Contrastive Learning (GCL)** that incorporates relevance feedback ⁷ ⁸. For example, in finance, FinBERT models fine-tuned on financial texts significantly outperformed generic BERT on financial NER, QA, etc.. In a RAG context, using FinBERT embeddings would likely recall specific financial jargon better than a generic model. Another approach is **instructor models** (like text-instructor by HKUST) which allow you to prompt the embedding model with a task description (like "Represent this sentence for legal case retrieval") – effectively a form of embedding prompt-engineering rather than weight fine-tuning. These have shown boosts in specific scenarios. There's also an emerging practice of **continual fine-tuning**: periodically retraining or fine-tuning the embedding model on a growing set of domain questions and relevant document pairs (some companies generate synthetic Q&A from new documents to fine-tune the encoder such that those docs become more findable). The caution: fine-tuning requires a decent amount of data (or very careful regularization) not to overfit and distort the vector space too much. It also can break the model for out-of-domain queries. A compromise is multi-vector indexing or ensemble: keep a generic embedding for broad semantic recall, but also index with a specialized embedding and merge results. If the domain is truly narrow (like all documents are about internal company policies), a single domain-tuned model can outperform a general model by a large margin in recall, as it knows the terminology extremely well.
- **Matryoshka Embeddings (Hierarchical Retrieval):** This is a novel technique to enable **variable-length or multi-scale embeddings**, analogous to Russian nesting dolls ⁹. The idea is to train a model to produce embeddings that can be truncated at various lengths while still being meaningful. For example, a 1024-d vector where the first 256 dims alone represent a high-level summary, the next 256 add more detail, etc. This allows a "funnel search" approach ¹⁰: use fewer dimensions (faster search) to get candidate docs, then refine with the full vector if needed. It can also mean generating separate embeddings for different granularities: e.g. an embedding for the whole document, and embeddings for each section, linking them together. Matryoshka-style models effectively compress multiple representations into one. A 2025 blog noted that truncation-aware training can yield **2–4× smaller embeddings with no loss in fidelity** ¹¹, which is a big win for speed and memory. In practice, one could index both coarse and fine vectors: first retrieve at coarse level (to get relevant documents broadly), then within those retrieve relevant sections with fine vectors – this improves scalability and possibly recall (fewer missed documents, since coarse retrieval casts a wide net without examining all fine details). Some vector databases (like LanceDB) are exploring storing hierarchical embeddings for each record ¹². For an enterprise with diverse content lengths, Matryoshka embeddings (or generally, hierarchical indexing) could ensure that both long and short documents are handled efficiently – long documents won't dominate the recall just because of their length, and short snippets won't be overlooked.

- **Multilingual and Cross-Lingual Retrieval:** In many enterprises, knowledge might be in multiple languages. If users query in one language and content is in another, you need an embedding model that aligns languages in the same vector space. Models like **Cohere multilingual v3** (supports 100+ languages) or multilingual E5, or SBERT models like LaBSE, are designed for this. If you use a monolingual model like Ada for English queries but half your docs are in Spanish, recall will suffer dramatically because the model isn't aligned for Spanish (it might still embed Spanish but likely not well). So, ensure to pick a multilingual embedding if needed. Cross-lingual retrieval recall can also be boosted by translating queries or documents into one common language as a pre-processing step, but that introduces translation errors. A strong multilingual embedding (e.g. LaBSE or Instructor-xl multilingual) can directly handle it. Keep in mind that multilingual models might have slightly lower raw recall on English than an English-only model of the same size (due to capacity dilution), but the benefit of cross-language capability outweighs that if your use case needs it.
- **Embedding Dimension and Index Considerations:** Higher dimensional embeddings can capture more information but also can increase the chance of irrelevant “noise” in similarity. There's a subtle effect: as dimension grows, distance metrics concentrate (curse of dimensionality). In some cases, using *too high* dimension can make it harder to distinguish truly relevant vs just somewhat similar, without large datasets to train on. However, in practice, the models we discuss are trained to optimize semantic similarity, so using the full dimension is usually best. OpenAI Ada is 1536-d; Cohere v3 is 768-d for English (which is smaller but still performs very well). If index memory is a concern, one could reduce dimensionality (e.g. PCA or random projection down to 256 or 300) – this will typically hurt recall a bit but might be acceptable if it's easier to scale. Some OpenAI users project Ada's 1536 down to e.g. 768 using PCA to halve storage with minimal loss. But **quantization** is often a better route: e.g. Faiss has IVF PQ that can compress vectors by 4× or 8× with tiny recall loss, which helps with large corpora. Also, **embedding normalization** (most use cosine similarity so the vectors are normalized already) can matter – ensure you use the similarity metric appropriate for the embedding (OpenAI's are usually cosine). And note embedding models have context length limits: e.g. Ada-002 can embed texts up to ~8191 tokens; Cohere v3 up to 4096 tokens. If documents exceed that, you must chunk or truncate before embedding. Long embedding models (OpenAI has text-embedding-ada-002-v2 with 16k token context rumored, or use summarization first). The good news is many enterprise docs are not huge (and if they are, chunking is anyway needed for relevance).

Recommendations: Evaluate your current embedding's effectiveness by measuring recall@K on a set of known query-document pairs. If it's low, consider upgrading the model before other tweaks. For an **English enterprise wiki**, OpenAI's `text-embedding-ada-002` is a decent baseline, but `text-embedding-3-small` would give a boost and cut costs. If you want to avoid external APIs or reduce costs long-term, try `intfloat/e5-base-v2` or the larger E5 variants – these have surprisingly strong performance (some E5 models claim >0.70 MTEB average, close to top proprietary models) ⁵. Just remember to allocate a GPU/CPU for inference (E5-base can encode ~20 docs/second on CPU). For **domain-specific** content (legal, finance, science), a fine-tuned model is worth the effort – e.g. `lawyer-embed-boost` (hypothetically) fine-tuned on legal Q&A will likely recall case law far better than a generic model that hasn't seen much legal text. If multilingual queries or documents are in play, ensure to use a multilingual embedding or do query translation as needed. Aligning languages can improve cross-lingual recall from near-zero (if using wrong model) to respectable (Cohere multilingual or LaBSE have strong cross-lingual recall). Lastly, keep an eye on **embedding model updates** – 2024 saw big leaps, and by 2025 we even have community models that rival or beat Ada. For example, *BGE (Baidu General Embedder)* models or **Gemini** (if released by Google) might

shift the landscape. Being modular in your pipeline (ability to swap embeddings easily) ensures you can take advantage of these to continually boost recall without changing other components.

4. Context Window Utilization

Large context windows (such as 100k-token models like GPT-4-128K or Anthropic Claude 100K) promise that we can stuff *many* retrieved documents into the prompt. In theory, this reduces the need for extremely selective retrieval – you could just retrieve 20–50 relevant chunks and feed them all. However, in practice, **effective use of large context is non-trivial**. Issues like the *lost-in-the-middle* effect mean that blindly adding more context doesn't linearly improve performance. Here's how to optimize context usage for recall:

- **Optimal Context Packing:** With 128k tokens, one might fit dozens of chunks. But you should prioritize **quality over quantity**. It's often better to feed, say, 10 highly relevant chunks than 50 mixed-relevance chunks, even if all 50 are somewhat related. The LLM's attention is a limited resource – it tends to pay most attention to the beginning and end of the prompt, while the middle may receive less focus (hence “lost in the middle”). To mitigate this, we recommend ordering retrieved contexts by **descending relevance or confidence**. That way, the most crucial evidence sits at the top where the model's attention is strongest. Empirical studies (Liu et al. 2023) have shown LLMs often favor the first few and last few segments in a long prompt, sometimes ignoring middle info. So, if you have 20 chunks, consider splitting into two prompts or ensure a summary of each chunk is also provided near the beginning.
- **Relevance-Based vs Chronological Ordering:** Generally, sort by relevance score (e.g. similarity or reranker score) for maximum chance that important info is noticed. However, if the query is time-sensitive (e.g. “how did metrics change over Q1 to Q4?”), chronological ordering might make more sense to preserve a timeline. In such cases, you might group by relevance first, then order chronologically within top group. Another approach is **hierarchical ordering**: group chunks by their source document or topic, and then order groups by relevance. This can help the model digest information in coherent blocks (document by document) rather than a random mix. Coherent grouping might reduce confusion and help the model synthesize within each group before comparing across groups. If you do this, it can be useful to insert brief separators or labels, e.g. “**Document A:** [chunk text] ... **Document B:** [chunk text] ...”, so the model knows which info came from where.
- **Token Budget Allocation:** In a prompt, you have to allocate tokens between: the system/message instructions, the user query, the retrieved context, and potentially other prompts (like chain-of-thought primes). **Maximizing recall** means maximizing the amount of relevant context provided, but blindly cramming everything can backfire (the model might get overwhelmed or dilute key facts). A smart allocation might be: *system prompt (e.g. 500 tokens) + user query (e.g. 50 tokens) + retrieved content (remaining ~95% of tokens)*. If you have 100k tokens for content, you could include ~90k of retrieved text comfortably. But you should consider diminishing returns – the more you put, the less each additional chunk contributes. One strategy: include the top N chunks verbatim, then for the next M chunks include *summaries* instead of full text. For example, after the first 10 chunks, you might insert a note “Further relevant info: (summary of other minor relevant points)”. This way, you surface secondary info without spending as many tokens. Another approach is **progressive disclosure**: give the model the most relevant info first, let it reason or produce an intermediate answer, then in a follow-up prompt provide additional context if needed. This is a form of iterative

prompting that uses context window in stages rather than all at once. It can sometimes yield better results than a single giant prompt.

- **Lost-in-the-Middle Mitigation:** Research by Liu et al. (2023) found models often fail to utilize middle parts of very long contexts. Some mitigation techniques: (1) **Splitting context into multiple queries** – e.g. if you have 30 chunks that are relevant, split into two prompts of 15 chunks each and ask the model to answer the query from each subset, then merge or have a final reasoning step. This ensures none of the context is truly “middle” because each prompt is shorter. (2) **Salience flags** – explicitly instruct the model to pay attention to certain parts: e.g. insert “*Important: ...*” before critical pieces, or ask the model at the end “Did you use all the provided information? If not, revisit the middle sections.” This is prompt-engineering to force attention. (3) **Reiteration** – if a crucial fact is in the middle of context, you might repeat it or summarize it again at the end. This of course uses more tokens, but ensures it’s seen at the end where attention spikes. Anecdotally, this reduces misses for key facts that otherwise got buried. However, use with caution: repeating too much can confuse the model or make it think there’s emphasis, but one well-placed reiteration of a critical sentence can be worthwhile.
- **Extremely Large Windows (128k+):** When you have a model like GPT-4-128k, you might consider simply throwing entire documents at it instead of chunking at all. For instance, if the question might require info from anywhere in a 50-page PDF, one could just feed the whole PDF in one go. The advantage is you avoid chunking errors (the model sees everything in original context). The disadvantage is cost and lost-in-middle – if the answer is on page 30, will the model find it? Some experiments suggest that while the model *can* read that much, its accuracy on QA doesn’t infinitely scale with context size – it still often needs the relevant part to be highlighted or near the prompt edges. **Retrieval is still useful** even with long context models. A sensible compromise: retrieve normally but allow a higher k (maybe 20–30) for a long-context model, because you can fit it, but don’t just feed unfiltered 500 pages. The retrieval step ensures the bulk of irrelevant sections are left out, focusing the model. This is why RAG remains relevant; as one article title put it, long-context LLMs can’t fully replace retrieval – because the model isn’t guaranteed to locate the needle in the haystack by itself. By pre-selecting with retrieval, you give it a smaller haystack.
- **Context Window vs Generation Trade-offs:** If you have a fixed total token limit per API call, using more for context means less for the model’s answer. But typically queries are short and answers are reasonably sized (the user likely doesn’t want a 50k-token answer). So allocating, say, 90% of tokens to context is fine if needed. Just ensure the *instruction* (system prompt) is not cut too short – it’s important to still remind the model to use the context and to cite or whatever format you need. A common pitfall is to overflow the model’s input and cause truncation of either the beginning (system prompt) or end (some context or user question). Always check the prompt assembly logic to preserve the critical parts.

One should also consider **how to prompt the model to utilize context**. For example, explicitly instruct: “The following is relevant information from our knowledge base. Use it to answer the question. If something appears unrelated, you can ignore it.” This helps the model not get distracted by peripheral info in a huge context. Without guidance, large context can sometimes confuse the model (it might try to connect disparate pieces incoherently).

Chronological or hierarchical ordering of context can be useful if the question demands it. For instance, a query like “Summarize the evolution of policy X from 2019 to 2021” implies time order is needed. In such cases, retrieving docs from 2019, 2020, 2021 and ordering them chronologically yields a better answer structure. Hierarchical context windows (used in tools like HyDE’s multi-turn or the LongRAG approach) might first provide the model a high-level summary of each document (global context) and then the details. E.g., present: “Doc1 summary: ...; Doc2 summary: ...; Now details: ...” – this way the model knows where to focus. This dual perspective (global then local) was shown in LongRAG to improve accuracy by ~6–17% over vanilla RAG on long-document QA, precisely by addressing the lost-in-middle and incomplete context issues.

In summary, treat the context window as a **valuable but limited resource** – even if it’s 100k tokens. Fill it with information in a structured, intentional way. Emphasize the most important facts by placement or even slight repetition. Use the window to include as **complete an answer context as possible** (context completeness), but avoid drowning the model in superfluous text. It’s often a game of diminishing returns: the first 2–3 relevant chunks give a huge boost; the next 5 moderate boost; beyond 10–15 chunks, you might get very marginal gains in answer recall. Use validation QA to find that sweet spot. If you notice answers are missing secondary facts, that’s the cue to add more context or change ordering. If answers get incorrect due to being distracted by irrelevant context, that’s the cue to prune or reorder context. Large context is powerful, but only when wielded judiciously.

5. Advanced Retrieval Patterns (Multi-hop, Self-Correction, etc.)

Simple one-shot retrieval may fall short on complex queries that involve multiple pieces of information or reasoning over several documents. Advanced retrieval patterns address these scenarios:

- **Parent-Child Chunking (Hierarchical Retrieval):** This pattern keeps hierarchical relationships between chunks. For example, each chunk might store metadata of its parent section or document summary. Instead of treating all chunks as flat, the retriever can use parent context to scope searches. Implementation: when indexing, you can concatenate section titles or document titles to the chunk text (or as metadata). This way, a chunk “inherits” context – e.g., a chunk of a legal clause might carry “Contract 2021 – Section 5: Liabilities” as part of its content or metadata. This improves retrieval recall because even if the query doesn’t match the chunk’s raw text, it might match the parent context. It also helps disambiguate chunks that are similar – the model knows which doc they came from. Another approach is two-stage retrieval: first retrieve relevant *documents* or sections (parents) using a coarse index (like document embeddings), then retrieve specific *chunks* (children) from those parents using a fine-grained index. This is a **hierarchical retrieval** approach common in systems like DrQA and multi-step retrievers. It can improve recall because even if the exact answer chunk isn’t found in a top-k global search, the system might pick the right document, then within that document find the answer. The cost is a more complex pipeline and slightly more latency (two searches instead of one). However, for large collections, this is also more efficient (pre-filtering by document can reduce false positives).
- **Contextual or Augmented Retrieval:** This involves enriching the embedding of a chunk with additional context – essentially the chunk knows about its neighbors or metadata. One example: prepend the section heading and document title to the chunk text before embedding (or to the embedding vector). This ensures the vector captures broader context. For instance, a chunk that just says “It uses a gradient boosting algorithm” is ambiguous alone, but if embedded with “Methodology – Model: It uses a gradient boosting algorithm”, it will be retrieved by queries about

gradient boosting in that methodology context. Some academic works (Poliakov & Shvai 2024) attach domain-specific keywords as metadata to each chunk to improve recall – like tagging all chunks about “GDPR” explicitly. When combined with a vector, those tags act like anchor points for similarity. This approach is relatively low-cost and often yields better recall **and** precision (since fewer irrelevant chunks surface). It’s advisable for enterprise content to include at least the doc title and top-level section in each chunk’s metadata or text.

- **Temporal-Aware Retrieval:** For content that changes over time or where queries refer to time (“as of 2022, what was X?”), retrieval should respect temporal context. This can be achieved via metadata filtering (only consider documents or versions from that time) and via time-aware embeddings. A straightforward method: include the date/year in the metadata and filter based on query clues. E.g., if query contains “2023”, restrict retrieval to content dated 2023 or earlier. This avoids high-recall mistakes like retrieving an outdated answer when a newer one exists or vice versa. Some advanced systems maintain **multiple indexes partitioned by date** (like one per year or per quarter) so that they can direct the query to the right temporal slice. If user specifically asks historical questions, having an index of archived versions can be useful. But a more dynamic approach is at retrieval time: add a keyword filter “year:2020” if detected. Also, note that some embedding models might not inherently know about time, so a vector similarity alone might surface the wrong era. That’s why coupling vector search with a temporal filter (or an attribute in the scoring function) can improve recall of the *correct* answer. In summary, if your knowledge base is time-indexed (wikis with version history, financial filings by year, etc.), design your retrieval to **isolate relevant time periods**. It can be as simple as storing `year` and doing a where clause before vector search. The trade-off is if time isn’t explicitly asked, you might need logic not to filter erroneously; often letting the model see multiple versions can confuse it. So it’s a bit of a challenge – experiments might be needed to balance this.
- **Multi-hop Reasoning Retrieval:** Some questions require information from multiple documents or multiple parts of one document that are not contiguous. For example, “Did person A ever meet person B? Provide details” might require finding a document about A that mentions an event, and another about B at that event. No single chunk has the full answer. Multi-hop retrieval handles this by iterative querying. A common technique is the **“self-ask” chain**: The system or LLM breaks the query into sub-questions, retrieves for each, and then combines them. One 2022 method, Self-Ask with Search, had the LLM generate follow-up queries (“Who is person A? [retrieve] Where was person A on Jan 1? [retrieve] Who else was at that location? [retrieve]”). Each retrieval informs the next. Graph-based retrieval like GraphRAG also naturally does multi-hop by following edges (e.g. find entity nodes then related entity nodes) ¹. To implement multi-hop in an enterprise setting, you can either use an agentic LLM approach or pre-compute links between documents (like a knowledge graph of references). A simple start: if a query has two distinct entities or concepts, do two separate queries and then feed both sets of retrieved info to the LLM. This at least ensures both pieces come into context. A more advanced approach uses the LLM’s reasoning: e.g., instruct the LLM “If the question is complex, you may ask me (the retriever) intermediate questions.” LangChain or LlamaIndex agents allow an LLM to call a search tool multiple times – that’s a form of multi-hop RAG. The benefit is dramatically higher recall on complex Qs that aren’t answerable from one snippet. The challenge is complexity and potential for error propagation (if the first hop retrieves something slightly off, subsequent hops might go astray). Nonetheless, benchmarks like HotpotQA have shown that explicitly doing multi-hop retrieval steps leads to higher correctness. In internal tests, multi-hop pipelines often recall supporting evidence from 2+ docs ~30% more often than single-hop.

Frameworks like **MultiHop-RAG** (a proposed dataset and approach) encourage retrieving multiple documents for multi-part questions.

- **Self-RAG (Self-Reflective Retrieval-Augmented Generation):** This is a recently introduced paradigm (Asai et al. 2023) where the LLM not only generates an answer but also **critiques its own answer and triggers additional retrieval if needed**. In Self-RAG, the model is taught to emit a special token or instruction like “<search>” when it realizes it needs more info. The process is: the model tries to answer with the current retrieved context; if during generation it becomes unsure or identifies a missing piece (“I haven’t seen info about X”), it can halt and request retrieval on that missing piece, then continue. This essentially gives the LLM a second (or third...) chance to fetch relevant info it didn’t get initially, improving recall of facts that were not in the first retrieval batch. Asai et al. reported that Self-RAG outperformed even retrieval-augmented ChatGPT on several knowledge tasks. Implementation-wise, it requires the ability to interpret the model’s output and feed back into retrieval mid-generation. You also need to prompt the model in a special way so it knows it can ask for more info (maybe with a phrase like: “If you are unsure or the context is insufficient, say: ‘Searching for XYZ’”). Self-RAG is powerful for high-recall requirements because it reduces the chance of “giving up” with whatever was initially retrieved. Instead, the model actively fills the gaps. One risk is infinite loops or unnecessary searches, so the system has to be designed to avoid that (e.g. limit to 1–2 additional searches). Another related concept is **Corrective RAG** – after the model gives an answer, it checks the answer against the sources or via a verifier, and if it finds missing or contradictory info, it goes back to retrieve more and then correct the answer. This could be done with an LLM “judge” that decides if the answer is fully supported; if not, more retrieval happens. These reflective mechanisms are in early stages, but they point toward more resilient high-recall systems that don’t fail just because the initial retrieval missed something.

In practice, adding a self-reflection loop tends to increase recall of supporting evidence by ensuring any *initial recall miss* can be caught. For example, if the model answered partly but then says “I’m not sure about X”, that second retrieval might fetch the missing piece. This is especially useful in **open-ended Q&A** where the space of possible relevant info is large.

- **Multi-step Knowledge Integration:** This pattern goes beyond just retrieval to using retrieval results in intermediate reasoning steps. For instance, an agent might retrieve info, use it to derive a new query, retrieve again, and so forth (like chain-of-thought with retrieval in between). One can design prompt chains such that the LLM first lists what needs to be found (“Step 1: find A; Step 2: find B”), then the system executes those as searches, then the LLM gets all results to produce the final answer. This explicit planning can improve recall of all necessary pieces because the model articulates what it needs. There is academic evidence that such **decomposition** improves answer accuracy on multi-hop QA. GraphSearch (2025) extended this idea to an agent that performs *Query Decomposition -> Context Refinement -> Query Grounding ->...-> Evidence Verification -> etc.* in a structured loop. They found this agentic workflow, combined with dual retrieval on text and graph, significantly improved answer recall and accuracy in multi-hop tasks.

In summary, **advanced retrieval patterns aim to retrieve all pieces of relevant info, not just one chunk**. Parent-child and contextual chunking ensure each chunk retrieved is rich and self-contained. Multi-hop and self-reflective retrieval ensure if one chunk can’t answer everything, the system can fetch additional ones. These patterns increase the likelihood that by the time the LLM generates its final answer, it has seen every necessary fact (context completeness). For an engineering team, implementing these might mean moving from a simple query->vector search to a mini dialogue between LLM and retriever. It adds complexity, but if

you need that last 10-15% of recall (particularly for complex queries or critical domains), these approaches are worth it. Start by instrumenting your system to detect unanswered aspects of questions (maybe via LLM feedback or by checking if the answer contains unknown placeholders). That will tell you if multi-hop or self-RAG is needed. Then you can incorporate something like Self-Ask prompting or an agent with a search tool. This is an active research area, so expect rapid evolution – the good news is early results show **big gains in challenging queries** (e.g. LongRAG’s dual retrieval had +17% recall over vanilla on multi-hop QA, and Self-RAG improved factual accuracy on knowledge tasks). Adopting these cutting-edge methods could give a production system a notable edge in recall and reliability.

6. Metadata and Filtering Architecture

Leveraging **metadata** (structured information about documents) and proper filtering is an often underestimated strategy for boosting retrieval precision *and* recall. A well-designed metadata schema allows the retriever or user to slice the knowledge base so that only relevant portions are searched, thus avoiding diluting recall with irrelevant results. Key points:

- **Metadata Schema Design:** In an enterprise context, documents come with valuable attributes: document type (policy, ticket, spec), author or team, creation date, product or project name, sensitivity level, etc. Decide which of these matter for retrieval. An optimal schema includes metadata that correlates with how users might restrict queries. For example, if users often ask “according to policy” vs “according to technical docs”, having a `category: policy` vs `category: technical` metadata is useful. If certain content is confidential or user-specific, include access control tags (though those are more for security than recall). Also consider the format – for code, you might tag the programming language; for support tickets, tag the issue type or status. Essentially, any property that helps partition the data into more homogeneous subsets can improve retrieval when used right ¹³ ¹⁴. For instance, you might store `product_line` metadata on knowledge base articles; if a query mentions a product, you filter to that subset, immediately eliminating irrelevant articles and **improving recall within the relevant slice**. Without that filter, relevant results might be drowned out by irrelevant ones from other products. However, be careful: too strict filtering can remove relevant docs mistakenly, so filters should usually be applied when high confidence (like an explicit mention of a product name).
- **Pre-filtering vs Post-filtering:** **Pre-filtering** means applying metadata filters *before* retrieval (e.g. query only a subset of the index or add a clause to the search query). **Post-filtering** means retrieving broadly, then discarding any results that don’t match a metadata criterion. Pre-filtering tends to improve precision and speed (you search a smaller index) but can hurt recall if your filter is off. Post-filtering preserves recall (you didn’t miss anything in search) but you might retrieve a lot of junk only to drop it, wasting some effort. Example: Suppose a query is “2021 revenue of ACME Corp”. A pre-filter could restrict to documents from 2021 (metadata year=2021) and perhaps financial reports. This is good if your date extraction is accurate. But if a relevant document is an early 2022 summary of 2021 numbers, a strict year=2021 filter would exclude it (hurting recall). A post-filter approach would retrieve anything mentioning ACME and revenue, then among results prefer those with year=2021. Perhaps a hybrid: use year filter but also allow a small chance to search adjacent year. In general, use pre-filters for **hard constraints** (like tenant isolation – see below, or known language, or filetype if the question implies it wants a code snippet vs a PDF). Use post-filter or rerank for **soft constraints** (like user asks “recent”, you might boost recent docs but not exclude older if they are highly relevant). Some vector DBs allow applying filters at search time (e.g.

Pinecone's filter query, Weaviate where filter narrows vectors considered) – these are effectively pre-filters. Others rely on you to filter after retrieving by ID. If using hybrid retrieval, you can also filter the sparse part (like an SQL “WHERE category=Policy” in BM25 query) and restrict vector search collection accordingly.

- **Namespace/Tenant Isolation:** In multi-tenant systems (where each client's data must be isolated), metadata is essential. The simplest safe strategy is **completely separate indexes per tenant** – then there's zero risk of cross-talk and no complex filtering needed at query time (since a given query handler only touches one index). However, managing many indexes (if you have thousands of tenants) might be operationally heavy. An alternative is a single index with a `tenant_id` field on each vector and always applying `tenant_id == X` as a filter on queries. Many enterprise vector DBs support that. It's critical to enforce this at every retrieval call to avoid any data leak. In terms of recall, isolation actually makes the retrieval problem easier (each tenant's data is a smaller set, so recall@k is inherently higher within that set). So, high recall is easier to achieve per tenant. The challenge is purely making sure multi-tenancy doesn't break things. If there are common relevant pieces across tenants (e.g. some generic knowledge shared by all), you could either duplicate that content in each tenant index or use a separate “global” index that is searched in parallel (with proper permission checks). For example, if a company's wiki shares some public documentation, you might combine tenant-specific search with a global search. But typically, treat each tenant entirely separately for simplicity. Note: If using open-source tools, make sure they have capability for filtering or you might inadvertently retrieve someone else's data. Most modern vector DBs (Pinecone, Weaviate, Vespa, Milvus etc.) support filtering on fields like `tenant_id` natively.
- **Metadata for Boosting Ranking:** Beyond binary filtering, metadata can be used as signals. For instance, document popularity or user rating could boost certain results. Or freshness could be a factor: if two docs are equally relevant textually, prefer the newer one (to improve likely correctness). You can incorporate this via a reranker or custom scoring function – e.g. $score = 0.7 * semantic_score + 0.3 * freshness_score$. Some systems convert important metadata into additional embedding dimensions or separate vectors (a multimodal approach). A simpler approach: include metadata text in the document content. For example, append “Written in 2015” to the text so that a query with “2015” matches it more. However, that's a bit of a hack and can confuse the model, so use carefully.
- **Tag Taxonomy Design:** If you tag documents with topics or categories, those tags effectively create a taxonomy that can be exploited for retrieval. For maximum precision, the taxonomy should be **mutually exclusive and collectively exhaustive** at the level you plan to filter. If a doc is tagged “database” and “network” and a user asks for “network latency issues”, ideally you filter to network or at least boost network-tagged docs. Overlapping tags or overly broad tags reduce their utility (if everything is tagged “internal” that doesn't help discriminate). A hierarchical taxonomy (e.g. Product -> Feature -> Issue Type) can allow hierarchical filtering: query mentions a product -> filter all content of that product, improving recall since irrelevant products are gone. But if the taxonomy is too fine-grained, you risk missing relevant docs that were mis-tagged. So design tags in line with query patterns. One strategy: analyze logs to see if users often mention certain categories or keywords – if yes, make those explicit filters. E.g., if many queries say “in API X” or “in project Y”, ensure those correspond to metadata fields you can filter.
- **Metadata-Enriched Indexing Pipelines:** Some cutting-edge approaches use LLMs to generate metadata like summaries, keywords, or question-answer pairs for each document offline. For

example, *Dadopoulos et al. (2024)* created document-level summaries and section-level keywords for financial reports, then used those in a multi-stage retrieval (first filter by summary using a similarity or keyword match, then retrieve chunks). They also embedded chunks concatenated with metadata (“contextual chunks”) to produce richer vectors. This improved their QA accuracy significantly, showing that spending effort to enhance metadata (even via LLM annotation) is worthwhile ¹⁵. For instance, auto-generated **keywords** can act as tags where none existed. If done well, a query that matches a keyword can directly retrieve the associated chunk with high recall. Consider this if your documents lack structure – you can prompt an LLM: “List 5 key topics discussed in this document” and store those as tags. Then the retriever can match on those tags. The cost is upfront API usage, but you only do it once per doc.

- **Pre vs Post Filtering Performance:** A quick performance note: filtering at query time (especially on large indexes) can be an expensive operation in some systems if not indexed. Some vector DBs index certain fields for filtering, making it efficient (O(1) check per vector considered). Others might scan results and filter, which if k is small is fine, but if filter is selective the ANN search might as well have not considered those. In extreme cases (like 1M docs, filter narrows to 1k docs), it’s best to either maintain separate indexes or use partitioning (e.g. Milvus partitions or Elastic indices per category). Partitioning is basically an index-level pre-filter, highly efficient. Tools like Milvus allow you to partition by tenant or category and automatically route queries ¹⁴.

In summary, **metadata is your friend** for achieving high precision *without sacrificing recall*. By narrowing the search space to the “right” slice, you effectively increase recall *within* that slice. The key is you assume the query provides some hint (explicit or implicit) to that slice. Many production failures of QA systems happen because they looked in the wrong place – metadata helps avoid that. A practical tip: start by identifying at least one or two metadata dimensions to incorporate (e.g. doc type and product line). Implement filtering or boosting on those and measure the difference. Often you’ll see that previously missed answers (low recall) were because the system was too busy looking at wrong categories. With filtering, those answers pop up.

Finally, ensure that metadata usage is configurable – you may need to adjust weights or turn off a filter if it’s harming certain queries. Monitor queries where zero results are found; it could be an overzealous filter. Likewise, monitor if irrelevant results from a certain category keep showing up; that suggests maybe a filter should be applied for those query types. An *adaptive retrieval* system might even classify the query type (e.g. legal vs technical question) and apply filters accordingly (some research has LLMs deciding which knowledge source or index to query – a form of metadata routing). All of this falls under the umbrella of **structured retrieval** which, when combined with unstructured embedding retrieval, makes an enterprise RAG system far more powerful and context-aware.

7. Evaluation and Metrics

To optimize and ensure high recall, you need robust evaluation methodologies. Traditional IR metrics apply, but RAG brings some twists. Here’s how to evaluate and monitor retrieval quality:

- **Recall@K:** This is the primary metric for retrieval – the fraction of queries for which the relevant document (or answer-containing chunk) is present in the top K results. In RAG, typically K is the number of chunks you feed to the LLM (say 5 or 10). You want recall@K to be as high as possible, ideally 100% for K equal to your context size. If recall@K is low, the LLM never even sees the answer.

However, recall@K alone doesn't account for rank ordering which matters (having the answer at rank 10 vs rank 1 can affect generation due to context position). So we pair recall with metrics like **MRR** and **NDCG**:

- **MRR (Mean Reciprocal Rank)**: MRR gives weight to how high the first relevant result is ranked. If the answer chunk is always the first result, MRR = 1. If sometimes second or third, it's lower. In RAG, a high MRR@K is nice because it means the model sees the answer right at the top of context (less chance of it being overlooked). If recall@10 is high but MRR is low (meaning relevant doc is often like 8th or 9th), that indicates the retriever finds the info but doesn't rank it well – this could cause lost-in-the-middle issues. A cross-encoder reranker often boosts MRR by rearranging relevant items to the front.
- **NDCG (Normalized Discounted Cumulative Gain)**: NDCG is useful if you have graded relevance or multiple relevant documents. For instance, some chunks might contain partial answers or supporting facts. NDCG@K will reward having multiple relevant chunks high up. This metric is common in web search eval and is relevant to RAG if your queries potentially need multi-document evidence. A high NDCG@5 means not only did you retrieve something relevant at rank 1, but ranks 2–5 were also helpful and not junk. That correlates with the LLM getting a fuller picture (context completeness).
- **MAP (Mean Average Precision)**: MAP is like an average of precision values at all recall points, basically measuring overall ranking quality across the whole list. It's somewhat less interpretable for RAG since we usually care about top K, but it's still a good summary if you have binary relevance for each doc.

Each metric highlights a different aspect: recall@K = did we get it in there at all; MRR = how early; NDCG/MAP = how well ranked overall especially if multiple relevant pieces. It's wise to track a suite of these. For example, in Yash's evaluation they looked at Precision@5, Recall@5, MRR – they saw recall improvements but interestingly MRR didn't improve as much in some cases, meaning sometimes the first relevant item wasn't rank1. That insight can drive specific tuning (e.g. tune the reranker to push that first relevant higher).

- **"Context Completeness" / Coverage Metrics**: A concept especially for RAG is measuring if all necessary context was retrieved. If a question needs two documents, did we get both? This is like multi-relevance recall. You can simulate this by marking all chunks that contain an answer or part of an answer as relevant, then computing recall@K for **each relevant piece**. Alternatively, use NDCG since it would give a gain for each relevant doc retrieved. If your use case demands comprehensive answers with multiple sources, you'll want a metric that penalizes retrieving only one of the needed pieces. For instance, if a question is "Compare product A and B," relevant info about A and about B must be retrieved. If you only get A's info, traditional recall (looking for any relevant) might count it, but the answer will be incomplete. So in evaluation, define both A-doc and B-doc as needed; only if both in top K do you consider it success. This can be encoded in a custom metric or simply manually analyzed for multi-facet queries.
- **Synthetic Evaluation Datasets**: Getting real labeled data (queries with ground-truth relevant docs) in an enterprise setting is hard (unless you have search logs or known Q&A pairs). A popular solution is **synthetic data generation**. You can use an LLM to create a question from a document: prompt it like "Read this document and come up with a hard question that it answers." Then you know that document should be retrieved for that question. By generating dozens or hundreds of Q's per document across your corpus, you build an evaluation set. Amazon's AWS blog (2023) describes using Anthropic's Claude to generate evaluation queries for RAG, and NVIDIA released a synthetic data generation tool for retrieval eval. Once you have this synthetic Q&A or Q->doc mapping, you

can compute recall etc. The caveat: LLMs might generate unnatural or overly easy questions. You have to curate a bit, maybe filter out trivial ones. But it's extremely useful for catching retrieval gaps. For example, you might find your retriever fails on certain phrasings the LLM came up with. You can then improve embeddings or add synonyms to fix that. Also, synthetic sets allow quick comparison of models: run your current retriever vs a new one on the same generated queries and see which gets higher recall.

- **Human Evaluation:** Ultimately, human judgment is gold for whether the RAG system is performing in production. Human eval for retrieval usually means: given a user query and the documents, have humans identify which documents actually contain the correct answer or are useful. This can be done as a side-effect of answer validation – e.g. label which sources are needed to answer the question. Another angle: have humans rate the final answers with and without retrieval. If the version with retrieval is correct X% more often, that implies good retrieval recall and usage. But specifically isolating retrieval quality, one can do blind comparisons: show a subject the top 5 retrieved snippets from System A vs System B for the same query, ask which set is more likely to answer the question (or even let them try to answer from those snippets). This requires skilled raters with domain knowledge. It's expensive, so usually used to validate major changes.
- **A/B Testing in Production:** When you make retrieval improvements (new model, new chunking, etc.), it's wise to A/B test if possible. That means splitting live traffic so some fraction of queries use the new retrieval pipeline while others use the old, and comparing outcomes. Outcomes could be explicit user ratings ("Was this answer helpful?" thumbs-up/down), or implicit signals like whether the user asked a follow-up (maybe indicating the first answer was incomplete), or if they clicked a source link, or if they rephrased the question (indicating dissatisfaction). You gather stats, and if the new pipeline shows higher success (say, higher user helpfulness ratings or fewer follow-ups), then it's a win. It's important to run enough queries to get statistical confidence. Also, ensure both groups get the same LLM and prompt, only difference is retrieval, otherwise you confound variables. A/B is the gold standard to catch things metrics can't – maybe the new retriever brings more relevant info but also more irrelevant noise that confuses the LLM, leading to worse answers. Pure recall metrics might say it's better, but A/B shows overall answer quality dropped. So always validate retrieval changes end-to-end if you can.
- **LLM-as-Evaluator:** A novel approach is using an LLM to judge retrieval results. For example, after retrieval, you can prompt an LLM: "Here's the question and the top 5 results. Does one of these results answer the question? Yes/No, which one?" This can automate recall measurement on unlabeled queries. If the LLM is reliable, it could essentially tell you if the answer was found. This is the idea behind some tools (e.g. Ares by Stanford) which combine synthetic queries with an LLM judge for evaluation. It's not perfect, but can scale to many queries cheaply. One must be cautious – the LLM might hallucinate relevance. But with careful prompting (maybe ask it to quote which snippet has the answer), it can approximate human eval.
- **User-Facing Metrics:** In production, track metrics like *answer success rate* (percentage of answers that contain correct info – perhaps approximated by how often users don't re-ask). Also track *coverage*: fraction of user queries that had at least one source returned (if your system has a fallback like "I don't know" when nothing found). If coverage goes up after a retrieval tweak, likely recall improved. Another metric is *time to answer* or *tokens used* – if retrieval allows a shorter answer or faster answer, that's good (though mostly a generation metric).

To summarize, we recommend maintaining a **ground truth dataset** of queries with known relevant docs (even if partially synthetic or small). Compute Recall@K, MRR, NDCG on it regularly. Use that for model comparisons. Also perform **manual spot-checks** – often just reviewing 20 random queries and their retrieved results can highlight issues (like seeing irrelevant doc at rank1 for a query – why did that happen?). For continuous monitoring, if your system is live, implement logging of cases where the LLM gave a wrong answer that was due to retrieval – maybe log instances where user corrects it or where a human-in-the-loop flags it. Then examine if the correct info was retrieved. Over time, you'll gather examples of recall failures to fix.

Finally, one innovative practice: **generate adversarial or challenging queries** to push the system's limits. E.g., take a document and ask something that requires info from two parts of it, or ask a very specific detail only mentioned once. These can test if chunking or retrieval is losing that detail. By continuously evolving your evaluation set (with real queries, synthetic queries, diverse scenarios), you ensure your optimization efforts are actually moving the needle on recall in meaningful ways, not just overfitting to a narrow set.

8. Production Challenges and Solutions

Building a high-recall RAG system in production entails more than just algorithms – you must handle content updates, versioning, duplicates, scale, and cost constraints:

- **Handling Document Updates (Incremental Indexing):** In an enterprise, knowledge is not static – pages get updated, new docs added, old ones removed. Re-indexing everything nightly might be too slow or costly as data grows. Aim for **incremental indexing**: when a document changes, update its chunks in the vector store. Many vector DBs support upsert operations. The tricky part is detecting what changed. If a small section was modified, ideally you'd only re-embed that section's chunk(s). One strategy: chunk the doc as usual, but compute a hash for each chunk's text. On update, re-chunk the doc and compare chunk hashes to the old ones; only re-embed and replace those that changed or new chunks. This avoids re-embedding the entire doc if only one paragraph changed. If documents have stable IDs, you can update by ID (some DBs allow updating the vector in place; others you might delete old and add new). Also consider **timeliness**: do you need the search index updated in real-time (seconds) or is hourly okay? For a support knowledge base, maybe hourly is fine. For something like incident logs, maybe you need minute-level updates. Ensure your pipeline can handle continuous ingestion – possibly using a message queue or event trigger whenever a source doc is edited. One challenge is if the content is user-generated (like conversations, forum posts), volume can be high – you might need to batch updates or prioritize certain content. A robust incremental pipeline will monitor for failures (e.g. if embedding API fails for an update, retry it, etc.).
- **Version Control of Content:** Sometimes you have multiple versions of the same document (e.g. an API v1 vs v2, or yearly policy documents). If a user question doesn't specify version, you usually want the latest. But if they do ("according to the 2020 policy..."), you need that specific version. Solutions: include version info in metadata (like `policy_id=123, version=2020`). If query specifies, filter by it. If not, default to current. You could also have only current in the main index and archive older versions separately that are searched only when needed. But that risks missing answers if people ask historical questions. Another strategy: let the LLM handle it – e.g. retrieve all versions and let model pick the relevant info. But that could be confusing. It's safer to **isolate versions** with metadata and use query understanding to pick one. Training an NER or using an LLM to detect year or version mentions in query is doable. On the flip side, if a document changes and you want the

RAG system to immediately stop using old info, you have to **remove or deprecate old chunks**. Some vector DBs support soft deletes or timestamp-based filtering (so you can mark old ones as archived). Always ensure no stale info lingers beyond its valid date. For instance, if a policy changed, you don't want the old policy chunk showing up. We can mitigate by filtering on an "active" flag. Real-world example: Imagine a pricing sheet changed yesterday; you'd update the index with new prices and remove the old. But if removal doesn't propagate due to a bug, the system might retrieve outdated prices (dangerous!). So testing removal flows is as important as adding.

- **Deduplication:** Duplicate or highly similar content is common in enterprises (the same paragraph might appear in multiple documents, or minor variations across versions). Without handling this, you risk the retriever returning essentially the same answer multiple times, wasting slots. Also, duplicates can bias the LLM – if it sees the same text twice, it might overemphasize it. There are a few levels of deduplication:
 - **Exact duplicates:** If two chunks are identical text, you really only need one in the index. You can hash chunk text and skip indexing if hash seen before. Or do this periodically: cluster vectors that are extremely close (cosine sim > 0.99) and remove extras. If they belong to different documents, you may keep one copy but note the multiple sources in metadata. For example, if a FAQ entry is copied into a user guide, you could keep one chunk but list both source IDs. Then whichever source user expects can be cited.
 - **Near duplicates:** e.g. "Turn power off and on" vs "Turn the power off, then on" – not exact but essentially same. Embedding similarity can catch these. You could use a second embedding (like a MinHash or a locality-sensitive hash) to detect near duplicates at scale. Another approach is **clustering** chunks by similarity and picking a representative. Some vector DBs allow storing multiple IDs per vector, which could represent merged duplicates. Alternatively, at query time, you can filter out results that are too similar to higher-ranked ones (this is like result diversification). For instance, if top1 and top3 have cosine sim 0.99 to each other, maybe drop one of them (or replace one with next unique). This ensures the user/LLM sees a variety of info. In high-recall contexts, you do want maybe duplicates from independent sources as cross-verification (e.g. two sources saying the same thing can increase confidence), but it rarely helps to show both explicitly to the model. One is enough unless asked for comparative answers.
 - **Conflicting duplicates:** If duplicates have slight differences (like one says "cost is \$100", another says "cost is \$90" due to being updated), this is more dangerous – it's not true duplicates but out-of-sync info. Ideally, your indexing should treat the older one as superseded (maybe removed if you trust the update, or at least the metadata says "old version"). The LLM seeing both might be confused or give a wrong answer. So this ties into version control – best to not have conflicting info in the retrieved set if you can avoid it (via filtering out outdated ones).

The good practice is to at least remove verbatim duplicates, which can dramatically shrink index size and improve retrieval focus. Many have found that eliminating redundant chunks speeds up searches and can slightly improve recall (because the vector index has less clutter). For near-duplicates, you might not drop

them but you can implement a **diversity penalty** in retrieval (some rerankers penalize documents that are too similar to ones already selected). This increases the chance of covering multiple aspects in top results.

- **Scalability (10M+ documents):** Scaling to tens or hundreds of millions of docs introduces performance and cost challenges but also recall implications. As the corpus grows, **maintaining high recall requires retrieving more candidates or using better embeddings**, because with more possible items, the chance that a relevant item is low-ranked grows. For example, recall@5 might drop as you go from 100k docs to 10M docs if you keep everything else the same, simply because there are more "distractors". Solutions: increase **K** (retrieve say top 20 and let the model sort out which to use), or tighten chunking (make each chunk more content-dense so it's easier to match). Also, **ANN index parameters** need tuning: for HNSW, you might need a higher **ef** or **M** to keep recall high at large scale, which increases memory or query time ¹⁶ ¹⁷. Check how recall in ANN (approximate nearest neighbor) compares to brute-force on a sample; as you scale up, you often need to allocate more resources to keep that gap small.

Architecturally, for 10M+ items, you'll likely use **sharding** across servers. Ensure your vector DB can do that (most can). A query will then fan out to shards – latency might increase but still under control (with 10 shards maybe each does 1/10th of work). You also need to consider memory: 10M * 1536-d float32 ~ 60 GB just for vectors, plus index overhead possibly 2-3× that for HNSW (so maybe 180 GB). That often means multi-node or using disk-based indices if memory is an issue. Solutions like **Faiss IVF** can use disk (slower but scales), or **ScaNN** on disk, or managed services like Pinecone that handle it under the hood. If using disk ANN, monitor recall because sometimes disk indices trade some accuracy for space.

Another approach for scale is **hierarchical indexing** (mentioned earlier with parent-child) – e.g. first index by document or category to reduce candidates. This is effectively what Google does (two-pass retrieval). It's necessary when brute-forcing everything isn't feasible. For RAG, a common scaled solution is: retrieve top N docs by BM25 (which is quite scalable using inverted indices), then vector search only within those docs' chunks. This hybrid cascade can improve throughput while keeping recall if your initial filter isn't too coarse. In fact, a 2024 study found that a combined full-text + dense + sparse pipeline (a "blended" search) outperformed pure vector search on both accuracy and efficiency for large corpora.

Additionally, at 10M scale, you want robust **monitoring**: track query latency (p95, p99), index refresh times, memory usage, etc. A spike might indicate some segment of the index not fitting in RAM causing recall to drop. And design for scaling incrementally – e.g. test at 1M, 5M, tune parameters, then go to 10M.

- **Cost Optimization:** Running a RAG system has a few cost drivers: embedding API calls, vector storage/compute, and LLM calls. For retrieval specifically:
- **Embedding costs:** If using API (like OpenAI) to embed documents, this can be high upfront. For example, 10k documents averaging 1000 tokens is 10M tokens, which at \$0.0004/1K = \$4 – trivial. But 1M docs would be \$400, and 10M docs \$4,000. Manageable perhaps, but if docs are longer or if you update frequently, it adds up. Using newer cheaper models (OpenAI's 3-small is \$0.00002/1K, 20x cheaper than Ada) or open-source (cost is infrastructure – e.g. running a GPU for a few days to embed everything). If you have the engineering capacity, embedding large corpora with an open model like E5-large (which might run at, say, 100 tokens/sec on a GPU – do the math for your corpus) can avoid API costs entirely. Many companies embed on CPU farm to avoid GPU costs for large batch jobs (since embedding models are not huge, CPU can suffice albeit slower). Also consider incremental embedding – don't re-embed the whole corpus if 5% changed. Maintain an "embedded"

flag or use content hash to skip unchanged content. Another trick: some content might be available in vector form already (e.g. if it's image captions or something, maybe you have tags – not common though).

- *Vector DB costs:* Managed services often charge by vector count and dimension (and maybe query count). E.g. Pinecone might charge a certain amount for 1M 1536-dim vectors storage. If cost is an issue, you can reduce dimension (1536→768) or compress vectors (some DBs allow storing 8-bit quantized vectors, cutting memory by 4×). That might reduce recall slightly (quantization error), but often not dramatically if 8-bit. Another cost aspect: **index build time** – if you have to rebuild indexes often, it can use a lot of compute. Some systems support dynamic indexing (like HNSW can insert on the fly). That's preferable to periodic rebuild. Also consider cheaper storage tiers – some DBs have an option to store on SSD vs RAM which is cheaper but with some latency penalty. If top-notch recall is needed, you might stick to RAM. But you could store older or less-used portions on disk.
- *Compute for retrieval queries:* Each query uses CPU (or occasionally GPU if using GPU-accelerated search) cycles. If QPS (queries/sec) is high, and you use heavy rerankers (which might be a small BERT run for each query), that can add the need for some GPU or CPU servers. Optimize by using smallest effective reranker (MiniLM vs full BERT vs GPT-4). Caching can also cut costs: many queries repeat or are similar. If you cache retrieval results for popular queries, you save doing embedding + ANN search again. However, careful with caching if underlying data changes – ensure to invalidate caches on updates.
- *Overall pipeline:* Sometimes to save cost, teams consider skipping retrieval for certain queries and just rely on the LLM's knowledge (especially if the question is something the base model likely knows). This is risky for factuality, but if you have good confidence classification (like the system detects if a question is about internal data vs common knowledge), you might not invoke vector search for obviously general questions, saving those ops. Similarly, if user asks something that historically retrieval hasn't helped (like an opinion question), maybe you skip retrieval.

One concrete cost reduction in practice: OpenAI's new pricing made embedding very cheap compared to LLM calls. So retrieval portion of cost might be <5% of total (LLM dominates). That means you should not cut corners in retrieval quality just to save pennies – a single GPT-4 answer might cost \$0.05-\$0.10 in tokens, whereas retrieving vectors is maybe \$0.001. So from a cost/benefit view, prioritize *accuracy (recall)* over minimal retrieval cost. However, if using an extremely large custom model for rerank (like running a local 7B model for rerank), that might have hardware costs – try smaller models first.

To summarize, **scalability and cost** considerations enforce careful engineering but there are known strategies: use approximate indexes to keep latency manageable, partition data to keep recall high, compress/quantize to lower memory cost, prefer open-source for large batch embedding, and always measure the recall impact of any optimization. Many retrieval quality issues only show up at scale (like index parameter tuning), so plan to iterate as your corpus grows. Also, be mindful of the maintenance aspect – e.g. do you have processes for index rebuild if it crashes, backup of vector data, etc. A robust system will treat the vector index somewhat like a database – with backup/restore, monitoring, and fallbacks (maybe if vector search fails, fall back to keyword search so user still gets something, albeit with lower recall possibly).

9. Recent Research and Innovations (2024–2025)

The field of RAG is rapidly evolving. Notable recent research themes aimed at improving recall and overall effectiveness include:

- **Agentic RAG and Iterative Retrieval:** Inspired by the success of LLM “agents” (like AutoGPT), researchers are giving LLMs more autonomy in the retrieval process. Rather than a single retrieval step, the LLM can plan multiple steps, ask follow-up queries, and even perform reasoning in between. We touched on Self-RAG and multi-hop – these are instances of the broader trend of *agentic retrieval*, where the LLM essentially acts as a search agent. One example, the **GraphSearch** paper (Yang et al. 2025), uses an agent with six modules that decompose queries, reflect, verify evidence, and issue searches in dual channels (text and graph). It showed superior performance on complex multi-hop QA versus one-shot GraphRAG. Another example is **LlamaIndex’s query planning**: they’ve demonstrated LLMs that decide which index or which tool to query first, effectively doing a higher-level retrieval planning. The benefit is higher recall on complex queries and more efficient searching, since the agent can ignore irrelevant sources and focus on relevant ones using reasoning. Expect more frameworks that integrate an LLM-driven controller that dynamically interacts with the vector DB.
- **Graph-augmented RAG (GraphRAG):** Traditional RAG uses unstructured text retrieval. GraphRAG incorporates knowledge graphs or structured data to improve recall and reasoning. Several papers in 2024 have shown that augmenting text retrieval with a graph of entities/relations can improve answering complex queries that involve relationships or multi-step logic ¹. For example, a graph might connect Person -> Company -> Location, so a question “Where was the founder of X born?” can navigate: X (company) -> founder (person) -> born in (location). Without a graph, the system might retrieve an article about the founder, and the LLM would scan for birthplace – which might work, but graph can guide retrieval to exactly the needed node. Innovations include building heterogeneous graphs (combining multiple types of relations), and using **path-based retrieval** – ensuring the evidence covers a path of reasoning. The GraphSearch agent we discussed is one such innovation, showing that using relational queries alongside semantic ones improved answer accuracy. In practice, to leverage this, enterprises might build a knowledge graph of their content (e.g. linking related concepts, taxonomy, references). It’s an additional investment, but pays off for complex domain questions. There’s also research into **knowledge graph self-play** where the LLM can query a graph database in addition to vector search. Industry-wise, some companies incorporate product knowledge graphs into their chatbots to ensure consistency and better recall on factual lookups.
- **Reasoning-Enhanced Retrieval:** Another trend is injecting more “smarts” into the retrieval stage itself. Rather than purely similarity, retrieval models are being trained to *reason* about queries – essentially bridging IR with NLP reasoning. A paper by Shen et al. (2024) even suggests letting a large language model itself serve as a dense retriever by generating some latent representation on the fly. They found a GPT-based retriever could tie or beat traditional dual encoders in some cases, likely because the GPT could do a form of reasoning about the query. There’s also work on **late-interaction with reasoning**: e.g. ColBERT with added contextual expansion, or hybrid models that feed the top retrieved doc back into the query embedding (iterative refinement). One example: **HyDE** (already mentioned) is a kind of reasoning – the LLM “reasons out” a hypothetical answer then embeds it. Another is **REA (Relevance Estimation with Attribution)** where the retriever tries to

ensure the retrieved text can directly answer (a step towards guaranteeing answer recall). We also see research on chain-of-thought prompting to improve retrieval – e.g. instruct the LLM to think of related terms or sub-questions (like “what could be another way the answer is phrased?”) then retrieve. This merges retrieval with the LLM’s reasoning.

- **Retrieval-augmented generation in LLM training:** Instead of just using retrieval at inference, some research integrates retrieval in the training of the model. For example, Meta’s **RETRO** (2022) showed training a model that can attend to retrieved neighbors improved factual accuracy. More recent work in 2024/25 likely builds on this – e.g. fine-tuning LLMs to use tools or to incorporate retrieval into their forward pass. This might produce models that are inherently better at using context (reducing lost-in-the-middle perhaps). While not directly an “enterprise” setting thing you can implement, these advances may trickle into APIs (we might get models that are retrieval-native). This is relevant because it may shift the optimal strategies – e.g. a RETRO-like model might require a specific chunk format or length for best effect.
- **Larger and Specialized Embeddings:** New embedding models keep coming. For instance, OpenAI’s rumored next version might have an even larger vector or multi-vector outputs. Also, companies like Together and Vectara are releasing models claiming >10% improvements over Ada and Cohere on key benchmarks. One example: **Boomerang embeddings** by Vectara, which claim to incorporate some cross-attention from the document to better align query/doc space. The “Matryoshka embeddings” we discussed are also part of recent innovation – e.g. LanceDB demonstrating variable-length vectors. These aren’t just academic; they’re making their way into products (Galileo, Milvus, etc., are discussing funnel search, etc. ¹⁸ ¹⁰). For an engineering team, staying updated means periodically re-evaluating the embedding model choice every 6–12 months, as something might offer a >10% recall gain essentially “for free” (just by swapping model, as happened when Ada-002 replaced Ada-001, etc.).
- **Industry Case Studies:** In 2024-2025, many companies have shared their RAG system experiences. For example:
 - *Meta* has their open-source system combining Wikipedia retrieval with a tuned language model for WikiQA.
 - *Microsoft* integrated RAG in Bing (GPT-4 with Bing is essentially a RAG system searching the web). They’ve likely done a ton of work on caching, freshness (web content updates), and prompt formatting to maximize usage of retrieved data (since browsing is slow, they likely optimize recall by carefully selecting what to retrieve).
 - *OpenAI* Plugins / Tools: OpenAI enabling the model to call a search API is RAG; user experiences from plugins show that formulating the right search query via an LLM agent was tricky but improved with prompt engineering.
 - *Databricks* (or other data companies) using RAG for documentation assistants – they reported that RAG reduced hallucinations by significant amounts in their docs QA, highlighting how retrieval recall directly affects final answer correctness.
 - *Financial and legal firms* deploying RAG have shared that combining symbolic search (like regex for numbers or legal citations) with vector search gave better recall on those specific queries. The lesson is customizing retrieval to domain quirks matters.

One case study: **NVIDIA** (2024) evaluated chunking strategies across datasets (we saw that page-level chunking result). They also likely explored different retrievers. The takeaway from their work is the importance of matching retrieval approach to data type – e.g. for code, using a code-specific model increased recall by ~15% in their tests. So specialized models (like CodeBERT for code, SciBERT for scientific text) are an industry-proven trick to boost recall on niche corpora.

- **Emerging Trends:** We mentioned agentic and graph, another trend is **multi-modal RAG**. E.g., retrieval that can fetch images or tables, not just text. Systems like **Multimodal GPT-4** use text and image context – there’s research on how to best retrieve relevant diagrams for a question (embedding images into the vector DB as well). For an enterprise with diagrams or screenshots, this could be relevant (imagine searching design documents with UI images). Techniques include using CLIP embeddings for images and concatenating with text vectors, or having separate indices and letting the LLM decide which to query. It’s nascent but growing.

Another interesting idea is **personalized or user-aware RAG** – using user’s past behavior as metadata to boost recall of what’s relevant to them. E.g., if an employee from HR is asking a question, boost results from HR docs. This crosses into recommendation, but from a recall perspective, it can reduce noise (the HR person likely doesn’t want deep engineering docs in their answers). Some case studies (perhaps at EMNLP 2025) might show improved helpdesk responses by incorporating user profile in retrieval.

Finally, **evaluation innovations**: The community is publishing datasets and leaderboards for RAG (e.g. the *ARES* framework and *Mossi* at NeurIPS 2024). These focus on factual QA with retrieval. They often reveal weaknesses in recall – e.g. ARES might ask an obscure fact and check if RAG finds it. By following these, one stays aware of what techniques are improving recall (for example, many top entries use rerankers or special negative mining in training to up recall).

In summary, the cutting edge (2024–2025) is moving towards **LLM-driven retrieval loops**, **integration of structured knowledge**, and **embedding improvements**. Many of these show double-digit percentage improvements over prior art on benchmarks. An engineering team should consider which of these are feasible to adopt. For instance, deploying a full agentic multi-step retrieval might be complex for production right now, but maybe a simplified two-step search could be done. Or incorporating a small knowledge graph for key entities in your domain. At minimum, keep the system modular so you can plug in improved components (e.g. if a new embedding model is clearly better, you can re-embed and swap it; if a new reranker is published by MS Research, you can try it). Given how fast things are evolving, a **continuous improvement mindset** is key – what gives state-of-the-art recall today might be eclipsed in a year. (For example, if someone releases a 20B parameter dual encoder next year that has near human-level semantic recall, you’d want to leverage that.)

10. Concrete Implementation Recommendations

Let’s put it all together for a hypothetical scenario: a **10,000-document enterprise wiki** (say technical + policy docs) and design an optimal RAG stack for maximum recall, with justified choices:

Vector Database: For ~10k docs, almost any vector store (open-source or managed) can handle it. If ease of setup is priority, I’d recommend using an open-source library like **Faiss** or **Chroma** in initial development (since 10k is small, even a simple in-memory index is fine). For production, if you want enterprise features (monitoring, scaling as docs grow), consider **Pinecone** or **Weaviate** or **Milvus**. Weaviate, for instance, has

hybrid search built-in and filters, which could be handy. Pinecone has a robust cloud service and can easily scale if you later go to millions of docs. If you anticipate growth, starting with Pinecone (or Qdrant Cloud which is similar) might save migration later. But if cost is a concern, running **Qdrant** yourself (open source, written in Rust) is an excellent choice; it supports metadata filtering natively, and ~10k docs is trivial for it in terms of performance. Index type: use **HNSW** (the default in many). With 10k vectors, you could even do brute-force (flat index) and get <50 ms query times, but HNSW gives future scalability. Set **M** (connectivity) and **ef** (search accuracy) such that you get ~99% of brute-force recall (shouldn't be hard at this scale).

Chunking Parameters: Based on typical wiki content (mix of text, bullet lists, some code maybe), a good starting chunk size is **~500 tokens with 50 token overlap** (or ~350 words with 15% overlap). This often balances context completeness and specificity. It aligns with the example we saw where 500-char chunks with overlaps gave good results. Use a **recursive splitter** that tries to break at paragraph or sentence boundaries; this ensures chunks aren't cut mid-sentence unnecessarily. Overlap is important in case an answer straddles a boundary. 50 tokens (~1-2 sentences) overlap is usually sufficient. You can adjust chunk size based on document structure: if the wiki has sections with ~200 words each, you might just chunk by those sections (no need to arbitrarily cut them). But if some pages are long, chunk within them. After chunking, verify: average chunk length, and whether Q&A pairs in your validation set fall mostly within single chunks (if you find many answers need two chunks, you might increase chunk size or overlap).

Embedding Model: I recommend **OpenAI's text-embedding-ada-002 or text-embedding-3-small** for ease and strong performance. Since this is enterprise wiki (domain-specific but likely broad English text), Ada-002 is a proven baseline. However, given OpenAI's new pricing and improved model, **text-embedding-3-small** would be ideal – it's much cheaper and from reports has better multilingual and similar English performance. It's 768-dimensional (assuming "small" is smaller dimension) which saves space. If using OpenAI API is acceptable (data privacy, etc.), go for it. If not, a local alternative is **SentenceTransformers** model like **all-MiniLM-L12-v2** or the newer **all-mpnet-base-v2**. Those are 384-d and reasonably good (MIRACL ~30% vs Ada's 31% recall, according to some Reddit discussions). Or use **E5-base-v2** which we know is strong. If the wiki includes code snippets or very technical jargon, consider a model tuned for that (OpenAI embeddings handle code somewhat, but there are CodeBERT or Unixcoder for heavy code content). To start, Ada-002 is fine. Fine-tuning isn't necessary unless you observe clearly that many misses are due to domain synonyms the model doesn't get.

Hybrid Retrieval Setup: I strongly suggest using **hybrid retrieval** here. The wiki likely has unique terms (acronyms, error codes) where exact match is crucial. So incorporate a sparse search. If using Weaviate or Vespa, they can do hybrid natively (like BM25 + vector fusion). If using Pinecone or Qdrant, you might have to do it manually: e.g. also index docs in Elasticsearch or use an open-source full-text like **Lunar** or simply in-memory Lucene if small scale. For simplicity, maybe use **LlamaIndex or LangChain** which can do a BM25 lookup (they sometimes use Whoosh or similar) alongside vector search. Given only 10k docs, even just scanning all text for keyword matches is not too slow (<500ms easily). But better to use an inverted index. So architecture: Query comes in -> do a BM25 query on wiki texts (could be through Elasticsearch if you have it) -> do a vector query via Pinecone -> take top results from both (say 10 from each) -> combine. Use **Reciprocal Rank Fusion (RRF)** or simpler: sort by a weighted sum of normalized scores. Or even simpler: if BM25 top result looks extremely relevant (exact matches), you might ensure it's included. In many Q&A, dense and sparse results will overlap if both are good; but RRF ensures if one misses, the other picks up. This hybrid step is relatively easy to implement and will catch those edge cases, so recall improves (as Rohan Paul noted, hybrid improves recall by capturing both exact and semantic matches).

Re-Ranking: For 10k docs, initial retrieval is fast, so spending some time re-ranking ~20 candidates is fine. I recommend a **cross-encoder reranker** like `cross-encoder/ms-marco-MiniLM-L-6-v2` (which Yash used) or the slightly larger `-L-12-v2` if needed for quality. This will re-evaluate each (query, chunk) pair and give a relevance score. Integrate it such that it takes e.g. top 10 from the fused list and outputs a sorted top 5. This should boost precision and ensure the best answer chunk is ranked first if possible. Latency wise, MiniLM can score ~100 passages in ~50ms on a GPU or a few hundred ms on CPU – very manageable. If implementing in Python, you can use Huggingface Transformers with that model. Alternatively, OpenAI just launched a rerank API (though cost might not justify it here). Cross-encoder not only improves accuracy, but also mitigates any trade-off between dense vs sparse scoring scales. It's like an insurance that the final set is optimal. If you cannot host a model, another approach is to use the LLM itself as a reranker by asking it to choose the most relevant snippets. But that's slower and costly (though maybe interesting to experiment with GPT-3.5 doing it, as it might cost \$0.002 per rerank, not too bad).

Metadata Schema: Suppose the wiki has categories like "Policies", "Technical Guides", "HR info". We create a metadata field `category`. Additionally, possibly `author` or `team` if that correlates with content type. Also `created_date` if needed for recency queries. Use metadata mainly for filtering if query clearly indicates. E.g., if query contains "policy" or is asked by HR (if we know user role), filter to `category:Policy`. But by default, maybe we don't filter except to exclude irrelevant categories if we detect context. This can be done simply by keyword rules or a classifier. Another metadata might be `source_doc` name, which we'll include for answer citation or to group results from the same document.

Pipeline Architecture Diagram: *(In text form)* The flow:

1. **Query understanding:** The raw user query goes to a small module that (a) identifies if any filters apply (e.g. query mentions year => record that, or certain domain words map to a category filter), and (b) perhaps expands the query for BM25 (e.g. add synonyms if known). This could be as simple as some regex or a lookup table for product names to categories.
2. **Embedding & Sparse encoding:** Simultaneously, embed the query into the vector space using the chosen model. Also, form a sparse query (could be just the raw query terms for BM25).
3. **Hybrid Retrieval:** Query the vector index for top `k1` (say 10) and the BM25 index for top `k2` (say 10). Apply any metadata pre-filters here if determined (e.g. if user said "in 2022", filter vector search to documents from 2022, and filter BM25 results similarly).
4. **Merge Results:** Combine the results lists. For RRF, you could use formula $\text{score} = 1/(\text{rank_dense} + 1) + 1/(\text{rank_bm25} + 1)$ for those that appear in both, etc.. Or simpler, maintain two lists and take the top 5 from dense and top 5 from BM25 (ensuring no duplicates) to pass to next stage. (We choose top 8 unique overall, for example).
5. **Neighbor Expansion:** Optionally, for each retrieved chunk, fetch its immediate neighbors (previous and next chunk in the source doc). This is easy if you stored chunk indices or have a way to get them (maybe store a `chunk_id` and also a link to next/prev ids). Attach them temporarily with a slight lower priority. This yields a set of candidate chunks (maybe 12-15 total).
6. **Re-ranking:** Take these candidates and run the cross-encoder reranker. It will score each for the query. Sort by that score and take the top 5 (or whatever fits in prompt).
7. **Final Context Assembly:** These top chunks are then concatenated (with maybe a separator and source metadata) into the prompt, after a system message instructing how to use them.
8. **LLM Generation:** The prompt goes to the LLM (e.g. GPT-4 or 3.5) to produce the answer, ideally citing sources (we'd instruct it to mention which document or provide references if multiple sources).
9. **Response:** Return answer to user, possibly with those citations.

All steps from 2 to 7 are typically under a second or two with this setup on modest hardware.

Expected Performance Metrics: With this pipeline, I'd expect very high recall on answer-containing documents. Qualitatively: likely **Recall@5 > 90%** on internal Q&A if the content is in the wiki (because hybrid + rerank is very robust). Precision@5 should also be high (since reranker will push true positives up). Latency: the retrieval pipeline ~100ms (vector + BM25) + ~50-100ms rerank = ~200ms, which is negligible compared to LLM response time (e.g. GPT-4 might take a few seconds to generate the answer). So overall, user sees maybe 2-5 seconds depending on answer length, dominated by the model's generation. Cost per query: - OpenAI embedding of query: ~0.0001 USD (if 50 tokens at \$0.0004/1K for Ada) or even less for 3-small. - Vector DB lookup: usually charged by index size, not per query, but let's say negligible per query (maybe \$0.00001). - BM25 search: if self-hosted, just compute cost; if using Azure Cognitive Search or something, minor cost too. - Cross-encoder rerank: if using your own model, just infra cost (maybe fractional cent of GPU time). If using an API (not typical for rerank), then might be \$0.001 or so via OpenAI if using an embedding model as judge. - LLM answer: if using GPT-4 8k context and we pass, say, 5 chunks ~ 500 tokens = 2500 tokens context + answer ~500 tokens = 3000 tokens, that's ~\$0.09 per query (at \$0.03/1K input + \$0.06/1K output). If using GPT-3.5, it's far cheaper (~\$0.002).

So likely the main cost is the LLM. The retrieval adds maybe ~5-10% overhead to that in cost terms, which is well worth it for accuracy.

Specific Tools: To implement, one could use **LangChain** or **LlamaIndex** since they provide abstractions for hybrid search and rerank. For example, LlamaIndex has a `CompositeRetriever` that could combine keyword and vector retrievers, and a `RAGReducer` that can do reranking with an LLM or a provided model. However, building the pipeline manually is also fine (and often more transparent for tuning).

Monitoring in Production: We'd log queries and the retrieved docs chosen, user feedback if any. Also perhaps periodically run the synthetic eval to catch recall dips (especially after data updates or model changes).

Diagram (logical):

```
User Query
└─> [Query Analyzer] -> {filter criteria: None or category X, etc.}
    └─> [Vector Embed] (embedding model) -> query vector q
    └─> [BM25 Query] (e.g. Elastic) -> top_k2 doc IDs
    └─> [Metadata Filter] (if any criteria) applied to both retrievals
└─> [Vector DB Search] q -> top_k1 chunks (with scores)
└─> [BM25 Results] -> top_k2 chunks (or docs then chunks)
└─> [Combine Results] -> candidate chunks list
    └─> [Neighbor Expansion] (fetch neighbors for each candidate chunk)
        -> expanded candidate list
└─> [Cross-Encoder Rerank] (query, chunk) for each candidate -> rerank scores
└─> [Top N Selector] -> final top_n chunks
└─> [Prompt Assembler] -> system msg + user query + formatted top_n chunks
```

```
└> [LLM Generation] -> answer (with references)
└> [Return Answer to User]
```

If we include an “embedding store” step: we would have pre-embedded all wiki chunks and loaded into the vector DB (with metadata like chunk_id, doc_id, text). Possibly also an index of doc titles for BM25 if we choose to search titles or content.

Given this pipeline, for a typical straightforward query, the retrieval should surface the relevant info in those top chunks. For a multi-part query, our system might retrieve pieces on both parts due to hybrid and multiple candidates, and hopefully reranker keeps both if needed (or the LLM can be prompted to ask again if it finds something missing – that could be a future upgrade using self-ask).

In conclusion, this stack is **production-ready**: Pinecone/Qdrant for vector store with filtering, an embedding model (OpenAI or local SBERT), a lightweight BM25 (maybe just use Whoosh or Elastic), and a reranker. All components are well-established, and combining them addresses recall from multiple angles: lexical recall, semantic recall, and ranking precision. We'd also add **monitoring hooks** to measure retrieval hits: e.g. after answer generation, possibly verify if the answer text matches a source chunk (which could hint if retrieval missed something). If we find retrieval didn't have the needed info for some queries, we'd analyze why – maybe it was chunked out or a term mismatch – and iterate (maybe add that term to a synonym list or break that chunk differently).

By implementing this system, we expect to maximize recall for the enterprise wiki: virtually every answerable question should be grounded in a retrieved chunk. The LLM then has the best shot at providing correct, context-complete answers, fulfilling the goal of an enterprise-grade RAG system.

Bibliography:

1. Yash Mhaskar (Nov 2025). *“Improving RAG Retrieval: Lessons from Building and Evaluating a Hybrid Pipeline.”* Medium.
2. Laxmikant Dange (May 2025). *“Comprehensive Analysis of Optimal Embedding Model for RAG.”* Medium.
3. Benjamin Chu (Oct 2024). *“Search Smarter, Not Harder: Redefining Retrieval-Augmented Generation.”* Towards AI. [9](#) [11](#)
4. *“Best Chunking Strategies for RAG in 2025.”* Firecrawl Blog (2025).
5. Anurag Mishra (2024). *“Five Levels of Chunking Strategies in RAG (Notes from Greg's Video).”* Medium.
6. Zhao et al. (Nov 2024). *“LongRAG: Dual-Perspective Retrieval-Augmented Generation for Long-Context QA.”* EMNLP 2024.
7. Asai et al. (2023). *“Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection.”* (Preprint).
8. Yang et al. (Sept 2025). *“GraphSearch: An Agentic Deep Searching Workflow for Graph-RAG.”* arXiv 2509.22009.
9. Dadopoulos et al. (Oct 2025). *“Metadata-Driven Retrieval-Augmented Generation for Financial QA.”* arXiv 2510.24402.
10. NVIDIA (2024). *Internal Benchmark on Chunking Strategies* – reported in Firecrawl blog.
11. AWS Machine Learning Blog (2023). *“Generate synthetic data for evaluating RAG with Amazon Bedrock Claude.”*
12. Milvus (2024). *“Master RAG Optimization: Key Strategies for AI Engineers.”* Galileo.ai (mentions hybrid search and hierarchical indices). [19](#)
13. Rohan Paul (2023). *“How does hybrid search work?”* (Blog) – notes on improved recall with hybrid.

(Dates are included where applicable. Some sources are blogs and preprints from 2024–2025, reflecting the latest developments.)

Open Questions: Despite advancements, a few uncertainties remain:

- **Recall vs Scale:** How will retrieval quality hold up as data scales to hundreds of millions of documents? We expect some degradation or need for more complex hierarchical retrieval. Techniques like distributed ANN or layered retrieval help, but it's not fully known if ultra-scale corpora (think internet-level) can be handled with near-perfect recall without enormous compute. More research into efficient negative sampling and index structures (like learned indexes) is needed to keep recall high at low cost for very large collections.

- **Retrieval Failure Modes:** What causes most failures in production? We know some (e.g. query–document vocabulary mismatch, chunk boundary issues, outdated info). But systematically, there could be other patterns – e.g. overlapping concepts causing confusion. Understanding when the retriever fails (and possibly having the system auto-detect uncertainty) is an open challenge. Can we have the system know it might have missed something (some measure of query drift or low confidence)? This ties into self-reflection triggers.

- **Contradictory Information:** Handling conflicting sources is tricky. If two retrieved docs disagree, current LLMs might either pick one arbitrarily or produce a hedged answer. Open question: can we develop a module to compare sources for consistency or choose based on reliability metadata? Possibly a future approach is to assign confidence/trust scores to sources (like source A is official, source B is user forum) and bias the answer accordingly. There's research on truthfulness that might integrate with RAG, but no clear solution yet.

- **Prompt Engineering for Recall:** We do prompt the LLM to use retrieved data, but to what extent can prompt engineering make the LLM compensate for retrieval misses? For example, prompting it to reformulate queries (HyDE does that) or to explicitly list what info is needed. There's room to improve how the LLM and retriever interact via prompts (like meta-prompts that encourage multi-hop questioning). As new LLM capabilities emerge (like GPT-4's planning abilities), how to harness them without overly complicating the pipeline remains open.

- **Context Completeness Metrics:** We touched on evaluating if all needed context is retrieved, but formal metrics or automated ways to ensure “completeness” are not fully developed. One idea: use the LLM to simulate an answer and see if any aspect of the answer lacks support from retrieved text. If yes, then retrieval wasn't complete. This could be made into a feedback loop. But designing that reliably is an open problem (you don't want false alarms from hallucinated needs).

- **Domain Adaptation Efficacy:** While we can fine-tune embeddings, how far can we push that? Could a small fine-tuned model outperform a huge generic model for a niche domain? This could reduce dependency on big APIs. There's ongoing exploration here; a gap remains in understanding the limits of fine-tuning for recall (e.g. risk of overfitting so it recalls training examples but not truly generalizing).

- **Integration with Knowledge Bases:** As companies often have both text docs and databases/KBs, how to seamlessly integrate SQL or graph queries with vector retrieval? Some solutions (like prompt the LLM to decide between them) exist, but an optimal, unified retrieval interface is not fully solved. A related open question: can we automatically build a knowledge graph from documents to enhance retrieval (some early works, but not plug-and-play yet)?

- **User Feedback Loop:** How to best use user feedback to improve recall? If users click certain results or provide corrections, can we feed that into retriever learning effectively (active learning for the retriever)? Many current systems don't close that loop beyond manual tuning. Perhaps reinforcement learning on retrieval selection could be tried – that area is open.

Addressing these questions will likely be the focus of the next wave of RAG improvements, bringing us closer to systems that have **comprehensive recall** and deep understanding across ever-growing enterprise knowledge.

1 GraphSearch: An Agentic Deep Searching Workflow for Graph Retrieval-Augmented Generation

<https://arxiv.org/html/2509.22009v1>

2 Indexing in RAG Systems: Building the Brain Behind Knowledge ...

<https://medium.com/@alameerashraf/indexing-in-rag-systems-building-the-brain-behind-knowledge-augmented-ai-d9b89758c734>

3 (PDF) Analyzing the Impact of Various Indexing Techniques on ...

https://www.researchgate.net/publication/394810149_Analyzing_the_Impact_of_Various_Indexing_Techniques_on_Retrieval-Augmented_Generation_RAG_Performance_in_Closed-Domain_Question_Answering

4 9 10 13 14 16 17 Search Smarter and Not Harder: Redefining Retrieval Augmented Generation | by Benjamin Chu (周明祥) | Towards AI

<https://pub.towardsai.net/search-smarter-and-not-harder-redefining-retrieval-augmented-generation-108dff0f1f89?gi=666547c14ed1>

5 6 Choosing an Embedding Model | Pinecone

<https://www.pinecone.io/learn/series/rag/embedding-models-rundown/>

7 8 11 18 A Different Angle: Retrieval Optimized Embedding Models - Zilliz blog

<https://zilliz.com/blog/a-different-angle-retrieval-optimized-embedding-models>

12 RAG Tutorials - LanceDB

<https://lancedb.com/docs/tutorials/rag/>

15 Metadata-Driven Retrieval-Augmented Generation for Financial Question Answering

<https://arxiv.org/html/2510.24402v1>

19 Master RAG Optimization: Key Strategies for AI Engineers - Galileo AI

<https://galileo.ai/blog/rag-performance-optimization>