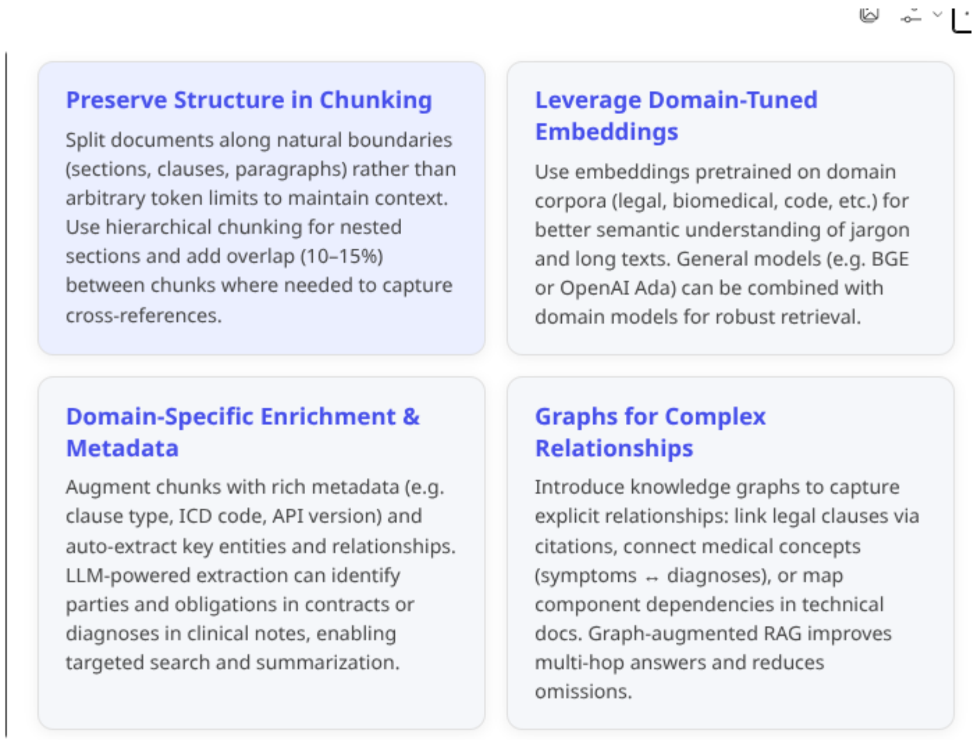

Preserve Structure in Chunking

Split documents along natural boundaries (sections, clauses, paragraphs) rather than arbitrary token limits to maintain context. Use hierarchical chunking for nested sections and add overlap (10–15%) between chunks where needed to capture cross-references.

Leverage Domain-Tuned Embeddings

Use embeddings pretrained on domain corpora (legal, biomedical, code, etc.) for better semantic understanding of jargon and long texts. General models (e.g. BGE or OpenAI Ada) can be combined with domain models for robust retrieval.

Domain-Specific Enrichment & Metadata

Augment chunks with rich metadata (e.g. clause type, ICD code, API version) and auto-extract key entities and relationships. LLM-powered extraction can identify parties and obligations in contracts or diagnoses in clinical notes, enabling targeted search and summarization.

Graphs for Complex Relationships

Introduce knowledge graphs to capture explicit relationships: link legal clauses via citations, connect medical concepts (symptoms ↔ diagnoses), or map component dependencies in technical docs. Graph-augmented RAG improves multi-hop answers and reduces omissions.

Extending LLMC to Domain-Specific Documents: Research Findings

Local-first Retrieval-Augmented Generation (RAG) can be successfully extended beyond code to handle legal, medical, technical, and general enterprise documents. This report summarizes **how to adapt LLMC's architecture** – originally built for code – to these domains, focusing on chunking strategies, structure parsing, domain-specific enrichment, embeddings, knowledge graphs, and evaluation methods. We emphasize solutions suited for **privacy-sensitive environments** (e.g. healthcare, legal, government) where **local models** (potentially up to ~70–80B parameters on 128GB VRAM) are preferred over cloud services. Below, we detail architecture adaptations and state-of-the-art techniques to ensure accurate, secure, and efficient RAG for each domain.

1. Document Structure Extraction & Chunking

Key Idea: Chunk documents at **meaningful semantic units** (sections, clauses, paragraphs) rather than arbitrary lengths. This preserves context and logical flow, crucial for accurate retrieval. Each domain has its own structural markers that we can exploit: [milvus.io]

- **Legal Documents:** Legal texts like contracts and statutes have well-defined hierarchy (articles, sections, clauses, sub-clauses). Use natural breaks such as clause titles, numbering, or headings as chunk boundaries. For example, a contract might be chunked into sections like *Recitals*, *Definitions*, *Payment Terms*, *Termination*, etc., with each clause or subsection as a chunk. Many contracts use numbered outlines (1, 1.1, 1.1.1) – these indicate nesting. A **hierarchical chunking** approach can parse these to build a tree of sections, then treat the smallest leaves as chunks. This was successfully done in a 2025 legal IR study: they split long articles into clauses, built a tree with the article title as root, and **appended the section path as metadata** on each chunk to preserve context. If a clause is still too long (e.g. >1024 tokens), further split it into overlapping segments (e.g. 50% overlap). Use overlap sparingly – around 10–15% of content – to ensure that if a clause references something defined in a previous clause, the context is carried forward. Avoid splitting tables or bullet lists in half; treat them as atomic units to keep their meaning intact. Tools: layout-aware PDF parsers (like PyMuPDF or pdfplumber) can extract text along with headings and indentations, which helps identify section breaks. For markup like HTML or DOCX, use heading styles or regex patterns (e.g. ^\d+\. for numbered clauses) to split accordingly. *Nested structure:* Retain the hierarchy by storing something like section_path="Section 5 > 5.2 > (a)" as metadata in each chunk. This way, if the QA needs more context, the system can surface sibling or parent clauses. Also extract any **titles or headings** as part of the chunk text or metadata – e.g. prefix a chunk with "**Termination:** (text of termination clause)" so that search can match section names. [milvus.io] [aclanthology.org]
- **Medical Documents:** Clinical notes and reports often follow conventions such as SOAP format (Subjective, Objective, Assessment, Plan) or have section headers (e.g. *History of Present Illness*, *Medications*, *Lab Results*, *Impression*). Use these domain headers as chunk boundaries. For example, treat the *Assessment/Plan* section of a doctor's note as one chunk if it's reasonably sized, since it contains the physician's conclusions and plans. For longer sections, chunk by paragraphs or subheadings (many EHR notes list each problem separately under Assessment/Plan). **Preserve list structures**: if a medication list or lab results are given as a list, keep the whole list in one chunk instead of breaking it mid-list – splitting a list can cause loss of meaning (e.g. a medication without its dosage line). *Nested structure:* Medical documents can have sub-sections (e.g. within "Objective", there might be "Vital Signs", "Physical Exam"). Where these are identifiable (through keywords or formatting), they can

form a hierarchy. You could store metadata like section="Physical Exam", parent_section="Objective" on chunks. **Metadata** extraction is important: for each note, capture attributes like date, clinic or provider, patient ID (in a de-identified form), and document type (e.g. Discharge Summary vs. Progress Note). This allows filtering (e.g. restrict retrieval to the same patient or same document type). *Tools:* If notes are in semi-structured text, simple rules or ML classifiers can detect section headers (e.g. a regex for "Assessment:" at start of line). Some NLP libraries (like **ClinicalBERT-based section taggers**) have been used to label sections in clinical notes automatically. For PDF reports, use layout parsing to find bolded or underlined headings common in that hospital's template. [milvus.io]

- **Technical Documentation:** User manuals, API docs, and specs are usually structured by chapters, sections, and subsections (often reflected by headings or HTML tags). Leverage these: each top-level section or sub-section can be a chunk if not too large. For example, an API reference might naturally chunk by each function or class – the function name, its description, parameters, and examples form one coherent chunk. If that exceeds token limits, split further: e.g. separate chunks for a function's description vs. its examples. Always include the function or section name in the chunk text/metadata to give context ("Chunk title: *Function X() description*"). Technical docs also contain *lists of steps, code snippets, and tables*. Keep a step list in one chunk to preserve order. Keep code examples intact with their explanation as one unit if possible, since the code and its description complement each other. *Metadata:* Tag chunks with things like the manual name or version, the software product or module, and the section title (e.g. {"product": "WidgetPro", "version": "3.1", "section": "Installation > Prerequisites"}). This enables queries like "installation prerequisites for WidgetPro" to filter relevant chunks. *Nested structure:* Similar to legal, you can represent the hierarchy of headings. Many modern docs are in markdown or HTML – use that structure. E.g., an <h2> denotes a section, <h3> a sub-section; you can capture the path "Guide > Chapter 2 > Section 2.1" as a context for each chunk. This was found to improve retrieval in long manuals by providing anchor points for search. [aclanthology.org]
- **General Enterprise Documents:** These include policies, SOPs, memos, and reports which may be long-form text but often with some headings or logical segments. For a long policy document, use its internal headings (e.g. *Purpose*, *Scope*, *Policy Statement*, *Procedures*) as split points. Each section can be chunked into paragraphs if needed, but try to keep each chunk self-contained in meaning. Business memos or emails can often be short enough to leave as one chunk, or split by paragraph if very lengthy. **Paragraph-level chunking** is a reliable default for unstructured text, as paragraphs naturally encapsulate one idea. Make sure to include preceding context if a paragraph is too short to stand alone (e.g. if a bullet list item is only one line, include the list's introductory line in that chunk for context). *Metadata:* For enterprise content, capturing document title, date, author, department, or classification

(confidential, public) is useful. E.g., a chunk from an HR policy might carry doc_type: "HR Policy", title: "Leave Policy", effective_date: 2025-01-01. Such metadata can aid filtering and also let the answer cite the source properly ("according to the 2025 Leave Policy…").

Chunk Size Guidelines: In all domains, aim to keep chunks within the input limits of your embedding model (often 512 tokens for BERT-based, or up to 8000+ for newer ones). Empirical research in 2025 showed chunk size impacts retrieval: **smaller chunks (~100 tokens) excel when answers are very specific facts, whereas larger chunks (500+ tokens) work better for broader questions requiring more context**. Thus, consider an adaptive strategy: default to ~300–500 token chunks for rich context, but if you have a FAQ-style repository (short Q&A pairs), smaller chunks might yield higher precision answers. Always **validate** chunking with test queries. If a query about "termination notice period" isn't retrieving the right clause, it might be that the clause was split such that "notice period" is in a separate chunk from "termination." In that case, adjust by merging those or increasing overlap. Tools like LangChain's recursive text splitter or Unstructured.io's document loader can automate splitting by headings, then by paragraph, etc., and allow configuring overlaps. [arxiv.org]

Overlap & Context Windows: Where content naturally flows across sections, include overlap. For legal and policy docs, a good practice is to repeat the section heading or a summary sentence in the next chunk so that context carries. One source suggests including ~10% of the previous chunk's text at the start of the next. However, keep chunks **de-duplicated** – too much overlap can confuse the retriever and waste index space. A sliding window approach is useful if there are no clear cut boundaries, but prefer semantic boundaries first. [milvus.io]

Preserving Formatting: In some cases (like tables or lists), retrieving the answer requires seeing the format (e.g. a policy table of allowed vs. disallowed actions). While vector search loses formatting, you can mitigate this by storing an HTML snippet or a reference to the page. For instance, you might index the text of a table but also keep an HTML version to display. This is more of a front-end consideration, but mention it in design.

Summary: By chunking along domain-specific structures and enriching chunks with contextual metadata, we create slices of text that are semantically meaningful and retrievable. This minimizes the chance of losing critical context during retrieval. As a best practice, **chunk by structure first, then by size**, and verify that each chunk "makes sense" on its own (e.g. reading it answers a sub-question or addresses a sub-topic clearly). This structured slicing sets the foundation for accurate domain RAG indexing.

Chunking Across Domains – Comparison

Domain	Typical Chunk Units	Overlap Strategy	Special Handling	Metadata to Capture

Legal	Sections/Clauses (e.g. *Definitions*, 5.2(a) clause) [milvus.io]	10–15% overlap if clauses reference prior text [milvus.io]	Don't split numbered lists or tables mid-way [milvus.io]. Maintain clause numbering in text.	Section title, numbering (hierarchy path) [aclanthology.org], parties involved, document name, effective date.
Medical	Note sections (SOAP), paragraphs within sections	Minimal overlap (sections often self-contained). Possibly overlap if a finding continues into plan.	Keep lists (medications, labs) intact [milvus.io]. Watch for broken sentences at boundaries.	Document type (e.g. Discharge Summary), date, patient ID (anon), section name (*Assessment*, *Plan*).
Technical	Manual chapters or subsections; API entry per chunk	Overlap at section boundaries if a concept continues. Include preceding sentence of prior section if needed.	Preserve code blocks with their explanation together. Keep step-by-step procedures in one chunk.	Document title, version, section headings (hierarchy), perhaps product/module name, and any reference IDs.
Enterprise	Policy sections; report sections; or paragraphs	Low overlap (generally distinct paragraphs). Repeat a lead-in line if a	Keep bullet lists as one unit. Ensure each chunk has enough context to be	Doc type (policy, memo, email), title/subject, author, date, department, classification level.

		section break is mid-topic.	understood alone.	

2. Domain-Specific Embedding Models & Retrieval

Key Idea: Use **specialized embedding models** that understand each domain's vocabulary and semantics for vectorizing chunks, while leveraging general models where appropriate. LLMC's architecture supports multiple embedding profiles and indices (via content-type routing), so we can assign a suited model per domain. This improves retrieval relevance, as shown by extensive research that domain-tuned models outperform general ones on in-domain queries. [arxiv.org]

- **Legal Text Embeddings:** Legal documents contain jargon (e.g. *force majeure*, *indemnity*, citations of laws) and long sentences with precise meaning. **Legal-BERT** and its variants are strong choices. For instance, *LEGAL-BERT* (Chalkidis et al., 2020) was pretrained on EU and US legal corpora and can interpret legal terms far better than a generic model. Similarly, **CaseLaw-BERT** was tuned on U.S. court opinions, and **Contracts-BERT** on contract text – these models excel at understanding statute citations or typical clause language. In fact, a 2025 study found that these legal-specific models **achieved state-of-the-art** on contract understanding tasks, outperforming even much larger general models (they beat a RoBERTa-large on legal tasks despite 69% fewer parameters). For our retrieval, using legal-tuned embeddings means queries like "supplier shall indemnify" will match the contract clause with that obligation, even if different wording is used (thanks to the model's semantic grasp of indemnification context). On HuggingFace, models like nlpaueb/legal-bert-base-uncased are readily available. We can further fine-tune these on our specific document set (e.g. fine-tune on a set of our company's contracts Q&A pairs) to sharpen understanding of our domain. **General embeddings** can still play a role: for broader semantic similarity or cross-domain queries, models like **BAAI's BGE** or OpenAI's Ada embedding are known to work well even on legal text. BGE (General Embedding model) has multi-lingual capability and handles long inputs (up to 8192 tokens), useful if we have multi-language contracts or very long provisions. We could adopt a *hybrid retrieval*: index with a legal-specific model and a general model in parallel, then fuse results. Alternatively, use a **re-ranker**: first retrieve top N by a fast model, then re-rank with a powerful cross-encoder like **ColBERT** tuned for legal text. ColBERT (Khattab et al.) uses late interaction and can significantly improve precision for matching legal passages by focusing on keyword alignments (useful for long contracts where specific phrasing matters). In practice, a pipeline might embed chunks with Legal-BERT (for accurate recall of legal specifics) and also with a general model (to catch more conceptual matches), then if our hardware allows, use a fine-tuned MiniLM or legal SBERT cross-encoder to score the top candidates. **Local model considerations:** Several legal models are relatively small (110M to 340M parameters) and can run on CPU/GPU easily. For larger options, **LexLM** (2023) is a legal-specific model built on RoBERTa that could be used if available. The user's current use of Qwen-3 4B suggests we have access to Alibaba's Qwen models – notably, **Qwen-3-Embedding-4B** could be a strong general-purpose embedder that is also multilingual and even code-aware, which might cover legal language well. It supports instructions and can produce 768-D or higher vectors. While not specifically trained on law, Qwen's versatility might allow consolidating routes (if we prefer one model for all docs for simplicity) – though for maximum accuracy, specialized models as described are recommended for legal content. [zilliz.com] [zilliz.com], [arxiv.org] [arxiv.org], [arxiv.org] [bentoml.com] [arxiv.org]

- **Medical & Biomedical Embeddings:** The medical domain is heavy with technical terminology, abbreviations (*HTN* for hypertension), and hierarchical relationships (diseases, symptoms, drugs). Several domain-specific models exist to handle this:
  - **BioBERT**: A BERT-based model pretrained on millions of PubMed abstracts. It understands biomedical terminology and context, making it effective for literature and even clinical text. For example, BioBERT will know *EGFR* can mean a gene or a lab value and embed accordingly, whereas a generic model might be clueless. [zilliz.com]
  - **ClinicalBERT**: This is tuned on clinical narratives (MIMIC-III ICU notes). It picks up on shorthand and patient-centric language ("Pt denies chest pain" etc.). This is ideal for EHR notes because it learned the style of doctor notes, including how certain findings are phrased. For instance, ClinicalBERT would closely embed "no evidence of pneumonia" with "denies pneumonia" recognizing the negation context, which is important for retrieval relevance in clinical QA. [zilliz.com]
  - **PubMedBERT**: Even more specialized, trained exclusively on PubMed abstracts (and full texts) – it often outperforms general models in biomedical QA and relation extraction tasks. For example, linking a question about a drug's side effect to a research paper paragraph. [zilliz.com]
  - **SapBERT**: A model that integrates **UMLS** medical knowledge by aligning synonyms during training. SapBERT is great for concept-level retrieval – e.g., it will embed "heart attack" and "myocardial infarction" nearly identically, which is incredibly useful in medical search where many terms have synonyms. This could be leveraged to improve recall in our system: we could either use SapBERT directly for embedding, or use it to normalize query and chunk text (find UMLS concepts and normalize terms). [zilliz.com]
  - **SciBERT**: Trained on 1.14M scientific papers across disciplines, it is adept at general scientific language. It can be useful for medical research documents or any scientific

reports our system handles. SciBERT has its own vocabulary tailored to science texts (e.g., it will have seen terms like “DNA polymerase” frequently). [github.com]

Empirical insights: a 2024 study found that **generalist embedding models can rival or beat specialized ones for short clinical text**, whereas specialized models shine on longer or domain-intensive queries. This suggests that if our queries are simple (single medical terms), even a strong general model (like SBERT) could retrieve relevant text. But for complex or lengthy inputs (like a paragraph of patient symptoms), a specialized model’s understanding yields better retrieval. Given we aim for high accuracy, using domain models is safer. We can route all “medical” content to a **medical embedding index** with BioBERT or ClinicalBERT. If multilingual support is needed (e.g., if handling non-English medical docs), models like **XLM-R** or multilingual BioBERT variants exist, or even BGE-M3 which covers 100+ languages including medical terms. [bentoml.com]

Another promising model is **MedCPT** (2023) – a *contrastively pretrained* model on large-scale PubMed search logs. This model was designed to improve zero-shot biomedical retrieval by learning from what people search vs. click (so it embeds queries and articles in a way that reflects relevance). Using MedCPT’s query encoder for questions and article encoder for documents could boost our medical Q&A scenarios, since it explicitly learned to handle “lay query -> scholarly article” matching. It’s public (from NCBI) and ~110M parameters, so running locally is feasible.

Local deployment: BioBERT and kin are base-size transformers (~110M params) – easy to run on CPU with moderate latency or on a GPU for speed. For larger context or better performance, we could consider newer *Sentence Transformers* derived from those (e.g. sentence-transformers/microsoft/BiomedVLP-CXR-BERT-general for clinical note similarity). If we have the computing budget, an alternative is to use a powerful general model like **Qwen-3-Embedding-4B** for everything, but instruct it for domain needs (the Qwen embed models are instruction-aware, so possibly we can prepend something like “Domain: medical. Task: embed the following for retrieval.” to the chunk text to steer it). However, evaluating on a small validation set of medical Q&A would confirm if this matches the performance of BioBERT. In academia and industry, **Ensemble retrieval** is common: e.g., some systems retrieve with multiple models and unify results. We can mirror that if needed (but at cost of complexity). [bentoml.com]

- **Technical Documentation Embeddings:** Technical content can be a mix of natural language and code or config snippets. Two types of models are relevant:
    - **Code-aware text models:** If the documentation includes code examples or mentions API symbols, using a model trained on code+text pairs is beneficial. **CodeBERT** (Microsoft) or **UniXcoder** are pre-trained on source code and related comments/docs. They will understand that List.append() is a method name and relate it to “adds an item to list” description. So a developer’s query “how to append to a list” would vector-similar to a chunk containing “append() – adds an item to the list”. General embeddings might miss that alignment. If our technical docs are API

references or developer guides with lots of identifiers, routing them through a code-oriented embedding model will likely improve hit rates. [zilliz.com]

    - **General sentence embeddings for prose:** For conceptual or explanatory documentation (like design docs, how-to guides), generic sentence embeddings do well. Models like **Sentence-BERT (SBERT)** variants (e.g., all-MiniLM-L6-v2 or all-mpnet-base-v2) have strong performance on semantic similarity for everyday language. They are lightweight (MiniLM has ~22M params) yet achieve high accuracy, and mpnet-base (110M params) is one of the top downloaded because it balances speed and quality. These can capture meaning of technical sentences like "Ensure the service account has read permissions" in a way that matches a user's question "Which permissions are needed for the service account?". [zilliz.com], [bentoml.com] [bentoml.com]

Multilingual or long docs: If documentation might include multiple human languages (e.g., an international company knowledge base) or very long sections, consider models built for those. **LaBSE** (Language-agnostic BERT Sentence Embedding) is a multilingual model covering 100 languages – useful if technical docs come in English, French, Chinese, etc., because it maps semantically similar cross-lingual sentences together. For long documents, models like **Longformer** have extended context; however, for embedding, splitting into chunks handles length. Alternately, OpenAI's text-embedding-ada-002 (now called engine *text-embedding-3* family) can handle ~8192 tokens and provides a single 1536-D embedding – one might chunk less and embed bigger pieces with that for efficiency, but local deployment of such a powerful model is not feasible (it's a cloud API). Instead, one could use the **BGE-M3** model which supports up to 8192 token input and multi-vector retrieval. BGE-M3 (2024, 7B+ parameters) is heavier but could run on a 48GB GPU in 4-bit. It has the advantage of multi-granularity training: it can represent both short and long passages effectively. [zilliz.com] [bentoml.com]

Resource trade-offs: For local-first, transformer models up to 1B parameters are fine on a single GPU. Our environment with 128GB VRAM could host even a 32B model embedding if needed, but typically retrieval uses smaller models for speed. If we needed extreme speed and had CPU-only, fallback options like **FastText or GloVe** embeddings were mentioned in literature – these are static embeddings (no context) and generally not as accurate for nuanced search, so they're not ideal for Q&A. A better lightweight option is **Universal Sentence Encoder (USE) Lite** or **distilRoBERTa-based SBERT**, which can run on CPU with low latency. They lose some precision but might suffice for less complex content. We should test recall@K on our data to see if the smaller model misses relevant info. If so, stick to a transformer on GPU. Given the hardware, using something like **Qwen-3-Embedding-4B** across technical and enterprise content might be attractive – Qwen3 embedding supports **over 100 programming languages** as well, meaning it was trained to handle code tokens, which is perfect for technical docs. It's also instruction-tunable, so we could incorporate domain hints. [zilliz.com] [bentoml.com]

- **General Enterprise Documents Embeddings:** For documents that are not highly technical or domain-specific (e.g. general corporate policies, wiki pages, etc.), high-quality general-purpose embedding models are suitable:
    - **All-mpnet-base-v2:** from Sentence-Transformers, 768-dim embeddings with excellent performance on semantic search tasks (it often tops benchmarks like MTEB for general domains). It's a good default for English documents. [bentoml.com]
    - **E5-large (Extensible Embeddings)**: a newer class of embedding models (e.g. intfloat/e5-large-v2) trained on data like MS MARCO, which are explicitly tuned for retrieval tasks. Many teams report strong results with E5 on Q&A similarity and it's available open-source (~300M params).
    - **OpenAI Ada v2** (if considering non-local for a moment) sets a high bar for quality – our system being local means we want analogues: Embeddings from **Gemini** or **GPT-3 family**. Notably, **EmbeddingGemma-300M** by Google (from the Gemma 3 model) is optimized for local deployment and rivals larger models on benchmarks. It's multilingual and very compact (300M) due to an efficient architecture, making it a candidate if we need to support many languages on edge devices. [bentoml.com]

In practice, for enterprise text, we could standardize on one strong general model to handle everything that isn't clearly legal/medical/tech. The **routing config** in LLMC can map multiple content types to the same "docs" route – e.g. HR policies and marketing plans both use the "default_docs" embedding route (with a model like all-mpnet or Qwen). This keeps things simpler unless there's a justification to break them out. If some enterprise documents are very domain-specific (say, a collection of financial regulations), we could add a custom route with a finance-oriented model.

Indexing & Retrieval: Each domain's chunks will be indexed in a vector store (LLMC uses SQLite+FAISS by default, which can handle moderate scale). For large volumes (e.g. millions of medical notes), consider a dedicated vector DB like Milvus or Qdrant which scale to billions of embeddings and offer efficient ANN search. These can run local (Milvus can run with GPU acceleration for ANN). The legal pipeline example used Milvus with HNSW index to store 2K Vietnamese legal docs' chunks and got query latencies <50ms. We can achieve similar with FAISS in-memory for our scale if under a few hundred thousand chunks, or use disk-based if larger. [aclanthology.org]

There's also the concept of **multi-vector retrieval** for complex documents (as BGE-M3 and Jina v4 support). For instance, a long policy might be better represented by multiple embeddings (one per section) under a single document ID. Our system currently works at chunk level, which is simpler, but in future, multi-vector per document could capture more nuance. [bentoml.com], [bentoml.com]

Summary: Pick **embedding models that align with content**: Legal-BERT and friends for contracts and laws, Bio/ClinicalBERT for medical text, code-aware models for API-heavy docs, and strong general sentence models for everything else. This ensures that the vector similarity search retrieves truly relevant context. We will configure LLMC's routing (as shown in the prompt's TOML snippet) such that slice_type="legal_contract" uses embeddings.routes.legal (with, say, CaseLawBERT embeddings into index emb_legal), slice_type="medical_clinical" uses embeddings.routes.medical (with ClinicalBERT into emb_medical index), and so on. [zilliz.com] [zilliz.com] [zilliz.com]
One can further improve retrieval by adding a **reranking stage**: e.g. after initial vector search, use a small domain-specific cross-attention model to score top results. There are lightweight rerankers like **BGE-Reranker-v2-M3** (multilingual) used in legal IR, or one can fine-tune MiniLM or Electra on a small QA dataset for each domain. This can significantly boost precision (in the Vietnamese legal DRiLL challenge, reranking + clustering improved correctness of retrieved articles). [aclanthology.org] [aclanthology.org], [aclanthology.org]
All embedding and retrieval steps remain **local-first**. Models like Qwen-3 or BGE can be run on our GPU server, and vector search can happen in-process or via a local DB. This retains security (no external API calls with sensitive data). With domain-optimized embeddings in place, we expect higher relevance: e.g., a query "What is the governing law of the agreement?" will vector-match to the chunk naming the governing law clause, because the legal embedder understands that concept, whereas a generic embedder might miss it if wording differs.

3. Domain-Specific Enrichment & Metadata Augmentation

Key Idea: Beyond raw text, enrich each chunk/document with **metadata and structured summaries** that capture important domain-specific information. This enrichment serves two purposes: (1) It enables more precise querying via metadata filters or by providing additional context to the LLM (for instance, including a summary of a contract clause can help the LLM generate a better answer). (2) It allows building knowledge on top of the text – e.g., linking defined terms in a contract or identifying medications in a clinical note – which can feed into a knowledge graph or help with direct Q&A (like "list all medications for this patient").
We tailor enrichment to each domain:

- **Legal Document Enrichment:** Legal texts, especially contracts and regulations, benefit from extracting key entities and clause-level insights:
    - **Parties & Roles:** Identify the parties involved in a contract and their roles (e.g. *Buyer*, *Seller*). Also capture other proper nouns (companies, persons) and defined terms. Often contracts explicitly define parties at the start (preamble) and throughout refer to them (like "the Company shall..."). We can use NER or simple regex (the parties are usually capitalized terms in quotes in the first paragraph) to get <Party1: ABC Corp

(Licensor)>, <Party2: XYZ Ltd (Licensee)>, etc. This can be stored as metadata on the document and on relevant chunks (e.g., obligations chunks could list which party is obligated).

- **Clause Classification:** Classify or label each clause by type (e.g., *Payment Terms*, *Confidentiality*, *Termination*, *Indemnification*). This could be done via keywords or an LLM prompt that reads a clause and outputs a tag. Having clause types allows users to query things like "What are the termination conditions?" and the system can filter or boost chunks labeled *Termination*. The CUAD dataset (Contracts AI) defined 13 important clause types (like Governing Law, Assignment, Liability, etc.) – those could guide what to label.
- **Defined Terms:** Contracts often have a Definitions section. Extract each defined term and its definition. This can be done by finding the typical pattern (words in quotes followed by "means…"). Storing a dictionary of defined terms is hugely helpful; if a later clause mentions a term, the system can provide the definition context. For RAG, one could treat each definition as a mini-chunk and also link it to where it's used (via graph edges).
- **Obligations & Rights:** It's useful to extract who must do what (obligations) and who has what rights. A generative model can be prompted: *"Extract any obligations in this clause, in the form: {obligor} – {obligation}."* For example, *"Seller – must deliver 100 units by Jan 1."* Likewise for rights: *"Buyer – is entitled to 5% discount if shipment is late."* These extractions can flag critical duties and help answer specific queries ("What does the Seller have to do if delivery is late?").
- **Dates & Deadlines:** Identify any dates (e.g. effective date, closing date) and time periods (e.g. "within 30 days"). This is straightforward with date regex or spaCy's date parser. Tagging clauses with any timeline can help queries about "notice period" or "term duration".
- **Summarize Clauses:** Use an LLM to generate a concise summary of each clause or section. This is especially helpful for very long clauses – the summary can be stored as metadata (e.g. summary field) that the retriever doesn't use for similarity but can be appended to the chunk text at answer time or used in a reranker prompt. For instance, summarizing a dense indemnity clause in one sentence ("Indemnitor will compensate Indemnitee for any third-party IP claims") helps a non-lawyer user understand the answer snippet. [llm.co]
- **Risk Flags:** Automatically flag clauses that are unusual or high-risk. Many legal AI tools do this – e.g., flag if there is no limitation of liability clause (missing provision), or if the indemnity clause is one-sided. We can implement simple checks or use an LLM with a prompt: "Does this clause deviate from standard practice? Identify any

risky or non-standard language." For example, if a non-compete clause had an indefinite duration, that could be flagged.

These enrichments can be performed by a combination of **rule-based NLP and LLM-based extraction**. Libraries like *spaCy* with legal NER models might identify laws, dates, orgs. For complex info (like obligations), an LLM prompt is effective. Notably, a contract review platform described using **clause-level extraction, risk flagging, and RAG-based summaries** to accelerate legal review. They extract clauses like indemnification, summarize them, and highlight deviations. We can mirror this: e.g., after indexing, run a background job that uses GPT-4 (locally, perhaps via Llama2 70B fine-tuned on legal tasks) to annotate each clause. The results (like "ClauseType=Termination; Summary=Either party may terminate if the other breaches with 30 days cure period; RiskFlag=None") can be stored in the metadata DB. [llm.co] [llm.co], [llm.co]

All this metadata is then used at query time. For instance, if the user asks, *"Is there an indemnification clause and what does it cover?"*, the system can directly jump to chunks labeled Indemnification. Or incorporate the clause summaries into the answer.

Additionally, **citations** or references in legal docs (like references to other contracts, or to statutes) should be extracted. If a contract says "subject to Section 12.3 above" or a law says "per 15 U.S.C. §77", capturing those as structured links can feed a graph (discussed in section 4).

- **Medical Document Enrichment:** Clinical data is often unstructured text that contains mentions of standardized concepts. Enriching involves extracting those concepts and any relevant attributes:
    - **Medical Entities (NER):** Extract key medical entities: diagnoses, symptoms, medications, procedures, test results, etc. There are specialized NER models (e.g. from **Spark NLP Healthcare** or **cTAKES**). For example, identify "Type 2 diabetes mellitus" as a diagnosis, "Metformin" as a medication, "HbA1c 8.5%" as a lab result, "MRI of the lumbar spine" as a procedure. Modern clinical NLP pipelines can map these to ontologies like **ICD-10 codes** for diagnoses, **SNOMED CT** for conditions, **RxNorm** for drugs, **CPT** for procedures. If we extract and normalize, we could attach a metadata like icd10: E11.9 on the chunk discussing diabetes, or rxnorm: 8600 on "metformin". This allows query like "patients with diabetes" to match the note even if the note used a synonym (because both map to E11.9).
    - **Anonymization:** Ensure removal or tagging of Protected Health Information (PHI) in text (names, IDs, etc.). This is more about privacy than RAG quality, but it's crucial for compliance – any pipeline processing patient notes should either work on de-identified data or perform on-the-fly de-identification (with tools like phfilter or spacy PII pipeline). We might not expose PHI anyway, but just as a step.
    - **Sections & Attributes:** Mark sections such as *Family History*, *Allergies*, *Review of Systems*. If not already chunked that way, at least label text spans. Also extract

attributes like *Patient Demographics* (age, gender) if present ("42-year-old male") – these could be metadata fields.

- **Clinical Summaries:** Use an LLM to summarize a patient's record or a long note. For instance, generate a brief "Patient Summary" that lists major problems and current treatment plan. This could be stored and used to answer broad queries like "What's the overview of this patient's case?". Abstractive summarization of medical notes is an active area of research – one must ensure factual accuracy (the summary must not hallucinate conditions not in the note). If using LLMs like GPT-4 (or a fine-tuned Flan-XL), one can prompt it to only use info from the text. There are also specific models (e.g. finetuned T5 for discharge summary summarization). We would validate these summaries with clinicians if possible. [mdpi.com]
- **Medical Codes and Abbreviations:** Expand abbreviations (maybe in metadata or hover hints). E.g., "HTN" -> "Hypertension". And identify important codes mentioned. If the text says "Procedure: LMWH 40mg sc daily", one might tag "LMWH" (a med) with its generic name or code.
- **Critical Values/Findings:** One can set up automated alerts for certain findings. For example, if a radiology note says "critical result: intracranial hemorrhage", tag that finding as critical. Or if a lab result is flagged (e.g. "K+ = 6.5 (H)"), identify it as high critical lab. This could be used to answer "Were there any critical labs for this patient?" reliably.
- **Contraindications/Interactions:** A more advanced enrichment is checking for drug interactions or contraindications mentioned. For example, if the patient is on Drug A and has Condition B that is contraindicated, one could flag it. This might involve external knowledge (drug interaction databases). Possibly out of scope for initial implementation, but worth noting for safety monitoring use-cases.

For implementing, **John Snow Labs' Spark NLP for Healthcare** provides pre-trained pipelines that do a lot of the above (NER with mapping to ICD10, RxNorm, etc.). We could integrate that in pipeline if Java-based tools are okay; otherwise, use HuggingFace models like *zurichat/bluebert* or *BioClinicalBERT* for NER and then a dictionary lookup for coding. Another route is to prompt an LLM: *"Extract the following from the note: Diagnoses, Medications (with dosage), Procedures, Allergies."* LLMs (like GPT-4 or a medically-tuned one) can produce a JSON of these. A 2024 study showed LLMs can be good medical coders when given tools or retrieval support. We can do a Retrieve&Rank approach for coding: retrieve likely codes for a mention and have the LLM pick the right one, as in some research on ICD coding.

The enriched data allows powerful queries: "What medications is this patient on?" can be answered by directly listing the medications metadata. Or the system could use both vector search and metadata filtering (find chunks of type *Medications List*).

Also, **patient-level aggregation** is a factor: One note might not list all of a patient's diagnoses (some are in past notes). A future extension is to compile a patient problem list by aggregating across notes via patient ID. For now, our RAG can retrieve across all notes for the patient if a query comes (since we tag patient ID).

- **Technical Documentation Enrichment:** Tech docs can be semi-structured (like an API reference has field names for parameters, return values, etc.). Key enrichments:
    - **API/Function Signatures:** Identify the name of functions, classes, API endpoints etc. If the doc is an API reference, likely each chunk already centers on one function. But it helps to parse out the *signature* (e.g., function name and arguments) and store it. Also note the return type or exceptions thrown. This is analogous to how LLMC already extracts function definitions from code. Here we do it from doc text: regex patterns (like FunctionName() or formatting cues (code font) can locate them.
    - **Parameter definitions:** If a chunk describes multiple parameters, consider splitting or at least enumerating them in metadata. E.g., parameters: ["timeout": "in seconds", "retries": "number of retry attempts"]. This could allow answering "What does the timeout parameter do?" by directly matching timeout metadata.
    - **Configuration Options:** Many tech docs list settings and their values. We can structure those into key-value pairs. E.g., {"setting": "EnableSSL", "default": false, "description": "Turns on SSL encryption."}.
    - **Warnings and Notes:** Extract any warnings, cautions, or important notes (often prefaced by "Note:" or indicated by an icon). Tag chunks that contain critical warnings so that if a user asks "Are there any safety warnings?", we can filter to those. Also, when answering a how-to, the assistant can append, "Note: ..." if relevant.
    - **Dependencies and Requirements:** If a doc mentions prerequisites (e.g. "Requires Python 3.8" or "This API is available in version >= 5.0"), structure that info. E.g. add min_version: 5.0 metadata or prerequisites: "Install library X".
    - **Cross-references:** Technical docs often say "See also: XYZ". We can capture these references (like a link between two topics in the documentation). This is fodder for the knowledge graph as well, but even as metadata we could list related topics.
    - **Summarize Procedures:** If there is a multi-step procedure, consider generating a summary or a one-line purpose. For example, "This section describes how to configure backup schedules." as a summary of a whole section. This can be done with either an LLM or by taking the first line of the section which often is a summary.
    - **Tag by Category:** If the documentation covers different categories (like "Admin Guide" vs "User Guide", or modules of a product), tag chunks accordingly to restrict queries. For instance, label something as audience: developer vs audience: end-user if such distinction exists in docs.

Implementation can rely on parsing the markup of docs (if we have HTML/MD). For example, identify <table> for configuration options and extract rows into a structured form. Or use simple patterns: "**Parameter**:" appears, then next line is explanation.
LLMs could help e.g., *"Extract all the commands mentioned and their purpose."* if dealing with a CLI guide. But rule-based might suffice for many as the docs are often formal.
The enriched info allows answering very specific questions without full generation. E.g., "What does --force flag do?" – if we stored that as metadata for the --force option ("bypass confirmation prompts"), we can directly respond with that snippet.

- **General Enterprise Documents Enrichment:** For corporate docs that don't fall in the above domains, enrichment focuses on classification and summarization:
    - **Document Classification:** Label documents by type (Policy, Report, Email, Meeting Minutes, etc.). This can be done by filename or location, or an LLM that reads a snippet. We might integrate the content source: e.g., if it comes from an email server, mark it as type: email. This is helpful so that if a query says "Summarize emails from my manager", the system knows to filter on type=email AND sender=manager.
    - **Summary & Keywords:** Generate a short summary of each document (especially if long). Also extract keywords or key phrases. For example, for a 10-page policy, have a 100-word summary stored. This can be used to answer "What is this document about?" quickly, or for the system to decide relevance. Key phrases (e.g. using RAKE or TextRank algorithms) can be added as metadata tags.
    - **Entities:** Identify people, organizations, projects mentioned. For instance, tag that a memo references project "Apollo" or person "Jane Doe". This aids search queries like "memos about Project Apollo". Standard NER with spaCy or transformers can do person/org extraction; we just must be careful with false positives if names coincide with common words.
    - **Sensitive Content Markers:** Perhaps detect if a document is marked confidential, or contains sensitive terms (like "salary", "SSN"). Not directly for retrieval, but for compliance (maybe to avoid showing some content unless authorized).
    - **Dates and Numbers:** If a report has key numbers (sales totals, etc.), one might extract those and label them so that queries like "Q4 revenue" can be answered via data lookup. This borders on structured data extraction.

Since enterprise docs can be quite heterogeneous, a one-size approach is tough. We can apply a general LLM to read and output a JSON of important fields. Because content might be internal, using a local LLM (maybe Llama-2 13B tuned on summarization) is advisable for summarization tasks. There have been open models like **GPT4All-J** that, if fine-tuned on instructions, can manage simple extraction tasks with high accuracy.

Using Domain LLMs for Enrichment: The question asks if enrichment should use domain-specific LLMs. If available, **domain-tuned LLMs can improve accuracy** of extraction. For legal, an LLM fine-tuned on contracts (like *InLegalBERT* or a larger GPT-3.5 fine-tuned on legal Q&A) would better identify clause roles than a generic model. For medical, Google's **Med-PaLM** (which is based on PaLM 2 large and further trained on medical QA) demonstrates near-expert performance in answering medical questions, implying it has strong grasp of clinical language. A model like that might extract a patient summary more accurately and safely than a general model. However, Med-PaLM isn't open-source. We do have some open medical LLMs (e.g. *ChatDoctor*, *GatorTron* (a 8B biomedical model), or LLaMA-based models fine-tuned on medical dialogs). These can be integrated if needed. For technical, one might use something like *StarCoder* or *CodeLlama* (for code comprehension) but those are more for generation than info extraction.

In practice, **a strong general model with careful prompting often suffices**, especially if we do one document at a time (reducing risk of confusion). For example, GPT-4 (closed) or Llama2 70B (open) will do a decent job extracting contract fields if given a structured prompt. Domain LLMs shine when the text is very domain-specific (like dense legal arguments or doctor shorthand). If using smaller local models for automation, it might be worth fine-tuning one on a small supervised dataset for each domain – e.g., fine-tune Llama-2 on a few hundred labeled contract clauses for clause classification. This yields a custom model with potentially higher precision on that task. But this is an advanced step. Initially, we can prompt engineer:

- e.g., *"You are LegalExtractGPT. Identify: Clause type (if any of Payment,Termination, etc), Obligations (who does what), and any Risk Flag in the following clause: ..."* – and parse its answer.

We also keep enrichment **modular and optional** – via config flags, so that one can turn off heavy LLM-based enrichment if not needed or if using a smaller setup. The architecture could have an "enrichment pipeline" that cascades: first do cheap regex/NER for obvious items, then if configured, call an LLM for deeper extraction.

Summary: Enriching the raw text with structured information transforms unstructured documents into a more **knowledge-base-like form**. This not only boosts retrieval (via metadata filtering and additional context for re-rankers) but also enables new capabilities like targeted questioning (e.g., "What are the obligations of each party?" can be answered by looking at extracted obligations rather than free-form search). All extraction can be done locally – using either small domain models or larger quantized models – to maintain privacy. Table below highlights some enrichment elements per domain:

Domain Enrichment Highlights

- *Legal:* Parties, defined terms, key clause types (Termination, Liability, etc.), clause summaries, and flags for missing or unusual clauses. E.g., label *Clause 5. Indemnification* – parties: Supplier indemnifies Client; flag: no cap on liability (risk). [llm.co]

- *Medical:* Problems/diagnoses with codes (ICD-10), Medications with dosages (RxNorm), procedures (CPT), allergies, vital signs. Anonymize PHI. Summarize clinical notes (with focus on Assessment & Plan). Flag critical findings (e.g. "acute stroke" in Impression). [zilliz.com]
- *Technical:* Identify API names, parameters, return values in text. Tag prerequisites and outcomes. Extract configuration options into key-value form. Mark important "Note/Warning" blocks. Link "See also" references between topics. Possibly maintain a mini index of error codes to error messages for quick lookup.
- *Enterprise:* Summarize documents. Tag doc type and key topics (via keywords or taxonomy). Extract action items in meeting minutes, or decisions. Identify persons or teams responsible if mentioned (for workflow queries like "Who is accountable for X in this document?").

All these enriched data should be stored in the index or a sidecar database and utilized in retrieval or answer generation. For instance, during retrieval we might boost chunks where clause_type = query_type (if user query classified as asking about termination, boost termination clauses). During generation, we can have the LLM incorporate structured info: *"The relevant clause (Section 10) says: . Note: This clause is governed by New York law."* The LLM.co platform underscores that having structured clause data plus retrieval leads to comprehensive answers with citations. We aim to achieve similar depth in answers by preparing the data accordingly. [llm.co]

4. Graph-Based Relationship Extraction (GraphRAG for Documents)

Key Idea: Extend LLMC's **Schema Graph (GraphRAG)** concept from code to text documents by constructing knowledge graphs of entities and their relationships found in the documents. Many non-code documents have implicit structures and references that can be made explicit with a graph. This graph can then be used to enrich retrieval (via additional context or multi-hop traversal) and provide **explainable connections** in answers ("Clause 5 refers to Clause 7, which defines X").
By domain:

- **Legal Knowledge Graph:**

Legal documents are highly interconnected:

  - **Cross-reference Graph:** Contracts and laws often refer internally to other sections (e.g. "as provided in Section 12.3 above"). We can parse these references and create directed edges between clause nodes. For example, node "Clause 12.3" ←ref– "Clause 5.1". Then, if a question is about Clause 5.1 obligations, the system can quickly retrieve Clause 12.3 as well due to the link (Graph traversal). This prevents missing context that spans clauses.

- **Document-to-Document Citations:** Legal opinions cite other cases, statutes, or regulations. For instance, a court decision might cite *Brown v. Board (1954)* or quote *15 U.S.C. § 77*. We can detect citations by regex or known patterns (case names in italics, "§" for statutes). Then build a graph where each document node (case or law) has edges linking to those it cites. In a legal RAG context, if a user asks "what did case X say about Y?", the system might retrieve case X and also automatically include cases that case X cites on Y, using the graph for context. Neo4j is often used in legal tech to store such citation networks. For example, researchers created knowledge graphs of German laws to assist LLMs, and projects like CaseLawKG exist. We can represent nodes for each legal doc in our corpus and edges: [arxiv.org]
    - *Case → cites → Case*
    - *Case → cites → Statute* (with a property like which section of the statute).
    - *Contract → amends → Contract* (if we have contract versions).
- **Entity Relationships:** If we extract parties and their obligations from contracts (from section 3), we can form a graph: Party A —[obligation: pay]→ Party B, or Party X —[is affiliate of]→ Party Y (if defined in contract). Another example: *"This lease is between Landlord and Tenant."* Create relationship: Tenant--(leases)->Property--(owned_by)->Landlord. If the user asks, "Who is the tenant and what property do they lease?", a graph query can directly answer if these triples are stored. In practice, we might not go so deep initially, but it's possible.
- **Defined Term Usage:** We could have nodes for each **Defined Term** in a contract, and edges from that term to clauses where it's used. E.g., Term "Effective Date" –used_in→ Clause 3, Clause 7. Conversely Clause 1.1 (Definitions) –defines→ "Effective Date". This graph helps if a question is "When is the Effective Date?" The system can traverse from the term node to its definition clause quickly.
- **Legal Concept Graphs:** In law, concepts like "duty of care" or "consideration" are mentioned across cases. One could create nodes for legal concepts and link cases that talk about them. However, that requires NLP to identify concepts, which is a bigger effort (maybe using topic modeling or NMF as per that LANL paper). They did Hierarchical NMF to discover topics in legal text and integrated that with KG and vectors: e.g., a latent topic representing "intellectual property" linking cases, which can augment search. This is advanced but interesting for trend analysis queries. [arxiv.org]

Utility of GraphRAG in legal: Graphs enable **multi-hop reasoning**. For instance, a user asks, "Does this agreement conflict with the master agreement from 2023?" If we link a contract and its master agreement (maybe via a property or an "amends" edge), the system can retrieve both and use an

LLM to compare clauses. Also, graphs help in **explainability** – an answer can say, "Clause 5 of the Amendment references Clause 12 of the Master Agreement, which states ...". [arxiv.org]
Implementation: Use a graph database (Neo4j can run locally; or even an RDF triple store) or an in-memory networkx graph for smaller scale. As we parse docs:

- internal refs: we can annotate chunk metadata with "refs_to: [section 12.3]" then batch-resolve these to actual chunk IDs and store edges.
- doc citations: maintain a dictionary of doc titles/ids so when parser finds "Roe v. Wade", it links to our node for that case (assuming the case is in our corpus; if not, maybe link to an external placeholder node).
- party relationships: from extraction, if we have a structured representation like {PartyA: "Alice Corp", PartyB: "Bob LLC", relationship: "agrees to purchase"} we can add edges in a graph like Alice -[purchases]-> Bob (with contract id as context).

There are libraries for contract knowledge extraction that could help (some research, e.g., **LeGen** 2023 uses generative models to extract complex legal relations).
At query time, GraphRAG integration can work like: use the user's question to identify if a graph traversal is needed (maybe via pattern or a separate "graph query planner"). Or simply, when retrieving via vectors, also retrieve related nodes via graph: e.g., if chunk X is retrieved and it has outgoing ref edges, consider pulling those target chunks into context (with lower weight). This was how GraphRAG for code worked (pulling neighbor nodes).

- **Medical Knowledge Graph:**

Healthcare data already has rich ontologies (UMLS unifies diseases, drugs, procedures, etc.). We can leverage those as a backbone:

- **Patient-centric graph:** If we have data across multiple notes or data types for a patient, we could create a node for the patient, link to nodes for each encounter (visit or note), and further to findings or diagnoses. For example, Patient John Doe → has_diagnosis → Diabetes (ICD-10: E11) and John Doe → had_lab_test → HbA1c (value node 8.5%). If a query asks "What were John Doe's labs over time?", a graph query could assemble them. However, if our system is primarily focusing on document QA rather than a fully structured EHR, we might not build a full patient graph initially.
- **Knowledge Graph of Medical Concepts:** Great value lies in integrating general medical knowledge. For instance, a knowledge graph could link *Disease* → *symptoms*, *Disease* → *treatments*, *Drug* → *interactions* → *Drug*. There are public knowledge bases for this (like the graph encoded in UMLS or more specific ones: e.g. DrugBank for interactions, or a simple curated graph: Fever, Cough → COVID-19). The **MedRAG** paper (2025) built a four-tier diagnostic graph and integrated it with retrieval. They captured differential diagnosis relationships – essentially mapping which diseases

have overlapping vs distinguishing symptoms, and fed that into the LLM's reasoning to improve diagnostic accuracy. In our context, we could use a pre-existing graph: e.g. for any disease mentioned, retrieve known risk factors or treatments from a knowledge base. This could help answer questions that combine patient info with general knowledge ("Given the patient has X and Y conditions, what drug should be avoided?" – the system might know via graph that drug Z is contraindicated with condition Y). [arxiv.org]

  - **Document linkage:** If one note says "see radiology report from 2023-01-01", link that note node to the radiology report node. Also linking notes in chronological order (a chain of notes for a patient). That way, if a query refers to "the previous visit" and we have a graph connecting visits, we know which to retrieve.
  - **Clinical entities relationships from text:** We can extract relations like *Medication – for – Condition*, *Test – result – value*, *Condition – causes – Symptom*. Some of these can be identified with existing models (e.g., SBERT-based relation extraction or prompt LLM). The **RadGraph** system defines a graph of findings and their attributes in radiology reports (with RadGraph-F1 as metric). For example, in a chest x-ray report, they connect "Nodule" –[location]→ "right lung". We could adopt similar approach: in any text, if we identify an entity and an attribute, link them. This structured info ensures that if a question asks "Where is the nodule?", the graph has (Nodule, location, right lung) to answer. [mdpi.com]
  - **Patient timeline graph:** Represent each patient's timeline as a sequence graph: Node1: "Clinic Visit Jan1" -> Node2: "ER visit Feb5" -> Node3: "Surgery Mar10". Each node connects to a document (note or summary). A query "what happened after the ER visit?" would hint at traversing that link to find the surgery note.

Implementing a large medical knowledge graph might be outside scope (it could have millions of nodes if we incorporate all diseases), but we can integrate *existing knowledge sources via retrieval* – e.g., have a separate index for a medical knowledge base and do retrieval from it as well (multi-hop RAG). But since the question focuses on local documents, our main graph focus is likely on internal relationships (e.g., between notes or within them) rather than all external knowledge. Still, a limited built-in knowledge graph (like a dictionary of medication interactions) stored locally could be useful for proactive warnings in answers.

Summarily, GraphRAG for medical could help in **cohort or trend queries** (if we had multiple patients, which we might not in one user's context) and in ensuring consistency (like linking a test result mention to its normal range node to check if it's high).

- **Technical Documentation Graph:**

Tech docs often describe systems with components and their relations:

- **API dependency graph:** If documentation is for an SDK or library, we can build a graph of classes, methods, modules, and their relationships. This is analogous to code GraphRAG (which tracks function calls, class inheritance, etc.). We might derive some of this from code itself (if we have the code, LLMC's existing code graph covers imports, calls, class hierarchies). But even from docs, we can glean relations: e.g. "Class B inherits from A" – make an edge B -inherits-> A. Or "Component X requires service Y to be running" – an edge X -depends_on-> Y. If our documentation describes an architecture, we might create a node for each component and link them. [arxiv.org]
- **Topic linkage:** Link related sections in docs. For example, user guide chapter about "Authentication" and API doc section on "Auth endpoints" are connected by topic. We can detect related topics by similar titles or explicit references. Also, many docs have a "Related Articles" list (which we can grab as edges).
- **Configuration graph:** In IT docs, you might have a graph of settings to outcomes. Or error codes to troubleshooting steps. E.g., node "Error 123" –resolved_by→ "Apply Patch X". Then a question "How to fix error 123?" can retrieve that via the graph connection.
- **Prerequisite graph:** Represent prerequisites as edges: e.g., "Install Java" –needed_for→ "Run Application Server". If a query is "Why can't I run the server?", the graph might suggest checking if Java is installed.

Essentially, any explicit relationship mentioned (depends on, part of, similar to, deprecates, etc.) can be captured. We can use a combination of NLP and manual config. For instance, parse sentences for keywords: "requires", "depends on", "see also", "replaced by", etc., to assign edge types. Or use an LLM: *"In this text: X, detect if there's a relationship like dependency or reference and output triple."*. Graph construction can be incremental.

Graph-aided retrieval: If a user asks a question about a class, we might also fetch its parent class documentation via the graph, anticipating that might contain inherited behavior info. Or if they ask "What changed from v1 to v2?", if we have a version difference graph (like each API method node might have edges to its older version), we can retrieve the diff nodes.

- **General Enterprise Graph:**

For enterprise docs, graphs might model:

- Organizational structure (if documents mention departments or people, a graph of Person->Department or Org chart could be present; linking that could help e.g. "Who approved this policy?" if we link to their role in the org).
- Project relationships: if we know certain files belong to Project Alpha, we can connect them. So a query "status of Project Alpha" can gather all related docs easily.

- Document lineage: if a policy supersedes an older one, link them (like *Policy 2021 → superseded_by → Policy 2023*). Then a search for the latest version can traverse.
- Meeting references: If meeting minutes mention follow-ups in another meeting, link those meetings.

Many enterprise relationships might be simpler to handle with metadata filters (like all docs tagged Project Alpha) rather than a complex graph. However, a graph could help with questions that require joining info across documents.

Graph Integration into RAG:

LLMC's current GraphRAG for code uses relationships like calls, imports to fetch connected code slices on a query. We will do analogous: when a chunk is retrieved, we look up its node in the graph and consider adding neighboring chunks. For example, if a contract clause chunk is retrieved and it **refs**→ another clause, we pull that clause's text as supporting context (with a lower score). Or if a medical note chunk is retrieved that mentions a follow-up test, and we have an edge to the actual test report, we might fetch that report. [arxiv.org]

We could also do **graph-based query expansion**: For instance, if user asks "Is there a conflict between Clause 5 of the amendment and Clause 12 of the base contract?", our system could use the graph: find Clause 5 (amendment) node, see it refers to Clause 12 (base). We retrieve both clauses and then use the LLM to compare. Without graph, it might have retrieved clause 5 but not know to bring clause 12. Graph solved the multi-hop.

Another approach: directly querying the graph for certain structured questions. If a question is recognized as a graph query (e.g. "Who are all the subcontractors in these contracts?"), we could query the graph of Parties (if we stored relationships like *Contract X –hasParty→ Subcontractor Y*). Then respond from that structured result. This requires a query classifier to decide if we should do a Cypher/Gremlin query instead of vector search. That might be beyond MVP, but it's an ultimate goal for an **AI that not only searches text but also reasons over the knowledge graph**.

Quality and Maintenance: Creating the graph relies on extraction correctness. We should monitor precision of those extractions (especially with LLM). In legal and medical, errors in relationships could mislead. Therefore, start with high-confidence relations (like explicit section refs, obvious citations) which are clearly parseable, then gradually add more nuanced ones when confident (perhaps have a human review loop for critical ones). The graph can be updated incrementally as new docs come in or old ones change.

In summary, adding a graph layer helps capture **non-local context** that pure embeddings might miss. It makes the system more **explainable** (since it can trace through relationships) and can improve **completeness** of answers on complex queries. Academia and industry are actively exploring this: e.g., a 2024 paper combined RAG with a knowledge graph and showed improved performance on complex QA, and an open-source project GraphAugmented-Legal-RAG is pursuing building a KG from legal text for retrieval. Our design would allow toggling this feature via config if needed (since not all use-cases require the graph – simple QAs might be fine with just vector search).

5. Evaluation Methodologies for Domain RAG Quality

Implementing these domain-specific RAG extensions requires robust evaluation to ensure the system is accurate, relevant, and safe. We need to evaluate both **retrieval performance** (are we getting the right documents/snippets for a query?) and **end-to-end QA performance** (are the answers correct, complete, and not hallucinated?). Moreover, domain-specific challenges mean we should use specialized benchmarks.

Retrieval Performance Metrics:

Standard IR metrics apply, measured on a test set of queries with known relevant documents:

- **Recall@K:** For each query, the fraction of truly relevant documents present in the top-K retrieved results. In high-stakes domains, high recall is critical – missing a relevant clause or note can be dangerous. We likely aim for Recall@5 or @10 to be very high (>0.9) on internal test queries. [mdpi.com]
- **MRR (Mean Reciprocal Rank):** The average of the reciprocal of the rank of the first relevant document for each query. This captures how high the first hit tends to be; important for user experience (the sooner a relevant snippet appears in the answer, the better). [mdpi.com]
- **nDCG (Normalized Discounted Cumulative Gain):** a weighted measure of ranking quality that accounts for graded relevances. If we have e.g. primary vs secondary relevance judgments, nDCG@K will reward ordering the more relevant ones higher. For instance, in a contract QA, the exact clause answering the question is "highly relevant", whereas a cited related clause is "somewhat relevant". nDCG would measure if the system ranks the exact clause above the tangential one. [mdpi.com]
- **Precision@K** can also be looked at (especially if we worry about false positives distracting the answer). However, with RAG the model can ignore irrelevant retrieved text, but if too much noise is retrieved it might confuse the LLM or waste context window.

To evaluate these, we should create a set of **test queries for each domain**. Ideally, these come from real user needs:

- For legal, we can use something like the **COLIEE competition dataset** which has legal IR queries (e.g. find Japanese civil code articles relevant to a query) to evaluate our legal retrieval. Or construct our own: take a contract and form questions like "What is the limitation of liability?" with the known relevant clause as ground truth. [arxiv.org]
- For medical, there's **emrQA** (a question set on clinical notes) which we can adapt. Also, the **USE Cases** from MIMIC (some tasks where given a patient note, find info about allergies, etc.).

- For technical, perhaps take sample Q&A from documentation forums or create them (e.g. from an internal FAQ).
- For general enterprise, if possible, gather some actual user queries and have SMEs identify which documents/chunks should answer them.

We can then compute recall, MRR, etc., on those queries by seeing if our retrieval (embedding + any rerank) surfaces the expected text. If below target, we iterate (maybe adjust chunking or embed model).

Quality of Answers / Generation:

Since RAG ultimately provides answers via an LLM synthesizing retrieved info, we need to measure correctness and completeness of answers:

- **Factual Accuracy Metrics:** There are domain-specific measures:
  - In medical, metrics like **FactScore** check if all facts in the answer are supported by sources. Or **RadGraph-F1** (for radiology) measures if the model correctly stated the relationships (like identifying correct findings and attributes). **MED-F1** is another proposed metric checking medical terminology alignment. We could use FactScore: basically, a score that penalizes unsupported statements. Tools like SciFact or even a second LLM can compute it by checking answer vs retrieved docs for entailment. [mdpi.com]
  - In legal, one might measure **accuracy of legal reasoning** or use entailment metrics: e.g. does the answer correctly entail the question given the law? Possibly use a model to verify or do human evaluation since legal nuance is tricky. There's no standard automatic metric yet, but one can do a **reference comparison** if we have an ideal answer.
  - For general QA, metrics like **Exact Match** and **F1** (token overlap with a gold answer) are common, e.g., as used in SQuAD or Natural Questions. If we create gold answers for our test queries, we can use those. But often our answers may be longer explanations, so BLEU/ROUGE may be applied to summaries.
  - **ROUGE**: if summarizing, e.g. a discharge summary, we can compare to a reference summary with ROUGE-L. But as noted in the healthcare RAG review, ROUGE can be misleading if it misses critical info. [mdpi.com]
  - Another approach: have domain experts rate answers for correctness, completeness, and coherence (Likert scale or so). Human eval is gold for quality but is expensive. We might do it on a small set periodically (like lawyers review 20 Q&A from the system).
  - **Hallucination Rate:** measure how often the answer includes information not present in retrieved documents. We can test this by deliberately asking at least one question where answer is not in the data; the correct behavior is to say “no answer found.” If the system makes one up, that’s a hallucination. We want a very low hallucination

rate, especially in legal/medical. We enforce citation of source for every claim – if an answer sentence has no supporting chunk, that's a red flag. We can automatically check that each sentence of answer overlaps significantly with some retrieved text (embedding approach or lexical).

   - **"No Answer > Wrong Answer" compliance:** Evaluate cases where the correct answer is "There is none in the docs." LLMC's philosophy is to prefer saying *"I don't have information"* over fabricating. We test queries with no support and see if the answer correctly refuses. A metric could be the false answer rate on unanswerable questions (should be low). [aclanthology.org]

Domain-specific Benchmarks & Datasets:
Mentioned in the user prompt, a few relevant benchmarks:

- **LegalBench** – a collection of tasks for legal LLMs (covering contract understanding, case QA, etc.). This can give examples for evaluation, though many tasks are generation or classification. E.g., *ContractNLI* is a dataset for determining if a contract clause entails a statement (natural language inference) – a system could use RAG to find the clause and then judge entailment. We might not directly implement NLI, but we can test if our retrieval finds the clause needed for the entailment task and then perhaps use a separate model to do the entailment check.
- **COLIEE** (Legal IR and QA competition) – has a statute law retrieval task and a legal QA task. We can use the retrieval task to validate our legal embedding approach by seeing how it performs retrieving relevant articles given questions (if our data included those laws).
- **CaseHOLD** – a dataset of case law holdings for retrieval/classification.
- **PubMedQA** – Q&A where answer is yes/no/maybe based on PubMed abstracts. We could simulate it: give our system the question + our indexed literature, and see if it finds and concludes correctly.
- **MedQA (USMLE)** – It's multiple-choice exam questions. Our system could attempt to answer by retrieving facts from a medical corpus. This is a tough challenge; it might need chain-of-thought. If we restrict scope to retrieval, we can check if for each question, the relevant text containing the answer is retrieved.
- **emrQA** – specifically built from clinical notes with queries. We can try our system on emrQA (the dataset provides questions and requires retrieving the snippet in the note that answers it). We could compute F1/EM of the answer string vs gold.
- **MIMIC-III/IV** – could create internal challenges like: given a synthetic patient profile, did the system correctly summarize or extract allergies? If yes, measure by human or by checking if all allergies were mentioned.
- **NaturalQuestions, TriviaQA** – these are open-domain general QA. They aren't domain-specific, but they test general knowledge retrieval. If our enterprise includes general

knowledge (e.g. a company wiki that might have trivia-like info about the company), we could adapt some to ensure the system can do open-domain lookup if needed. But largely, our focus is domain docs, so these might be less relevant except for baseline comparisons.

Freshness & Staleness: If documents update frequently (like laws amended, or new policy versions), we need to ensure the RAG answers from the current version. We thus track document version or last modified time in metadata. As part of evaluation, we simulate scenario: an old version is indexed, then it's updated and re-indexed – confirm that queries now pull the updated info and not stale slices. Also test the fallback: if a query is about something added in the latest commit that might not be embedded yet, LLMC in code could detect "slice stale" and read live file. We might implement similar: if user asks about a document and the index is older than the doc's timestamp, we either re-index on the fly or read the latest file content directly. Our evaluation can include a case where we intentionally leave an index stale and see if the system warns or updates (depending on design).

Safety Evaluation:

For medical especially, ensure no harmful advice. Possibly include some test cases where an incorrect answer could cause harm if hallucinated (like a medication dosage). Ideally, the system should refuse or express uncertainty if it's not sure. We might run a subset of **MedMCQA** or **MultiMedQA** questions that require reasoning and see if the system sticks to sources or goes off-track.

User Feedback Loop: In deployment, gather feedback from users (thumbs up/down on answers, or corrected answers) to continuously measure and improve. For now, offline eval suffices, but in production, implement metrics like:

- percentage of sessions where user had to rephrase (as a proxy for failure to answer initially).
- any occurrences of the assistant apologizing or correcting itself due to uncertainty (should ideally only when appropriate).

Cost & Latency Monitoring: Another aspect of "quality" is performance in practical terms. We should measure average retrieval time and answer generation time. With local models, ensure it's within acceptable limits (maybe <2-3 seconds for query for a good UX). If using larger local models (like Qwen-3 32B) for generation, we might hit slower responses. We can mitigate via quantization. But evaluation should track this: e.g., run 100 queries and log 95th percentile latency.

To illustrate the importance of RAG vs just stuffing context, consider the recent 1M-token context models. OpenAI and others are touting huge context windows, but **evaluation shows RAG is still crucial**. Huge contexts incur massive latency and cost – one demo had a 456K token prompt taking 76 seconds and ~$2 in API cost. In contrast, a typical RAG query sending maybe 1K of relevant text is near-instant and cents cost. Also, RAG provides source citations (crucial for trust in legal/medical) whereas giant context answers still don't cite. And you simply can't put *billions* of tokens of enterprise data into a prompt – RAG scales better by selecting slices. This perspective is supported by industry analysis: even with 1M-token LLMs emerging, for non-trivial corpora, retrieval remains the only viable solution. [dev.to]

To drive this point, the graph below contrasts the **cost and latency** of using RAG vs. extremely long context for a query:

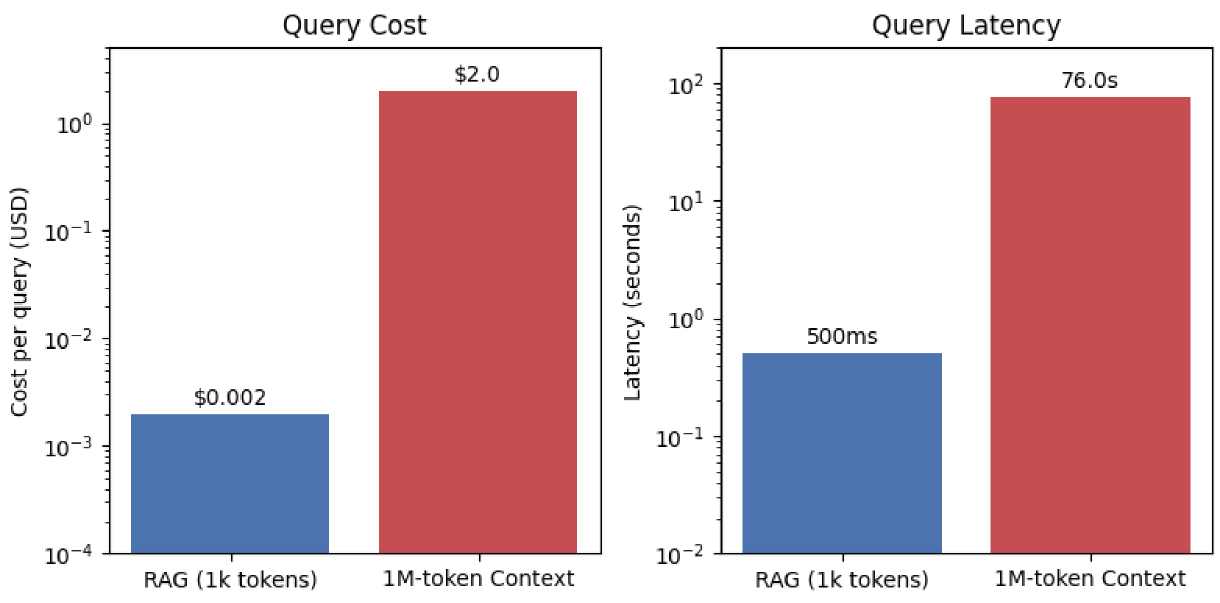

Figure: Query cost and latency – RAG vs. 1M-token context. A retrieval approach processing ~1,000 tokens costs fractions of a cent and responds in under a second, whereas sending a 500K token context to an LLM can cost dollars and take over a minute. Additionally, RAG provides source citations for transparency, which long-context alone lacks. [dev.to]

Evaluation Process: We will perform iterative evaluation:

1. **Unit tests on retrieval:** using curated query->doc pairs (for each domain). Use scripts to calculate recall, MRR, etc. Also test specific scenarios (e.g., a query referencing a defined term – does retrieval get the definition clause?).
2. **Simulated Q&A tasks:** Run the full system on benchmark questions (like from LegalBench, emrQA) and compare answers to ground truth or have them scored by domain metrics (e.g., BLEU for a known answer, or a specialized metric if available).
3. **Human review:** For each domain, have an expert read ~10 sample Q&A from the system. They check if any important point from the documents was missing or wrong in the answer. They also gauge if the tone and clarity are appropriate (less an issue with just retrieval, but for full answer, it matters).
4. **Safety checks:** Intentionally test some edge queries: in legal, ask something outside the document's scope (the system should not hallucinate law that isn't there). In medical, ask

"Should the patient take double dose?" expecting the system to either find relevant guidance or caution that it cannot advise beyond provided info.

5. **Graph evaluation:** If we implement graph, test a query which needs the graph (like a cross-ref question) with graph on vs off and see difference. Verify that graph augmentation does bring in the needed info.

Continuous evaluation: Set up automated regression tests so that as we tweak chunking or models, we see if accuracy metrics improve or degrade.

In summary, we will use **information retrieval metrics (Recall@K, MRR, nDCG)** to fine-tune the retriever for each domain, and domain-specific QA benchmarks (LegalBench tasks, medical QA, etc.) to ensure the end-to-end system actually answers correctly. We will also monitor for **hallucinations and stale info usage** to uphold the "no wrong answer" policy. Domain experts and targeted datasets will be an integral part of the evaluation strategy, given the nuance in legal and medical responses that automatic metrics alone might not fully capture. [mdpi.com]

6. Prior Art and State of the Art in Domain RAG

Lastly, it's useful to position our approach relative to existing systems and research:

- **Commercial Systems in Legal AI:** Tools like **Harvey.AI** and **Casetext's CoCounsel** have garnered attention in legal circles. Harvey (deployed at some big firms) uses GPT-4 under the hood, augmented with the firm's documents, to answer legal questions and assist with contract analysis. CoCounsel (by Casetext, now acquired by Thomson Reuters) similarly performs contract analysis, legal research Q&A, and document review with a GPT-4 based RAG approach. They likely use embeddings for case law retrieval and then prompt GPT-4 to reason (Casetext was known for its legal search engine). Our approach differs in being local-first, but we mirror their emphasis on citation and accuracy. Notably, CoCounsel advertises handling tasks like identifying risky clauses or summarizing depositions – indicating heavy use of domain prompts and possibly fine-tuned models for summarization. There's also **Lexis+ AI** and **Westlaw Precision** exploring AI to retrieve and summarize legal authorities; presumably they use vector search on their huge legal databases.

Another, **LLM.co's Contract Review** (an enterprise solution) directly aligns with what we propose for legal: they boast *"clause-level extraction, risk flagging, and retrieval-augmented summaries"* entirely on-premises, using models trained on legal language and fine-tuned on the client's own playbooks. This validates our direction – combining extraction (to identify clauses, deviations) with RAG for grounded Q&A. We're essentially building a similar capability within LLMC. [llm.co]

- **Medical AI Systems: Google's Med-PaLM 2** is a leading example of a domain-specific LLM (not retrieval-based, but fine-tuned on medical QA) that reached about 86.5% on medical exam questions, nearly expert level. It shows that incorporating vast domain knowledge (through training) yields strong results. However, Med-PaLM still can make mistakes and it doesn't cite sources, since it's end-to-end neural. Google is likely exploring retrieval for healthcare too (there are papers on retrieving patient info to help answer patient-specific questions). IBM Watson Health attempted something similar earlier (Watson for Oncology) but that was more knowledge base and not so much modern RAG.

On the open side, projects like **SymptomGPT** or **ClinicalCamel** are fine-tuning LLaMA on medical dialogs and instructions. These could be integrated into our pipeline for improved generation of answers, but with caution and thorough validation.

There's also specialized software in hospitals now: e.g., **EpIC** (Evidence-based Interpretation of Clinical records) that uses retrieval to assist clinicians in summarizing patient history from EHR – essentially a RAG system in healthcare. And smaller startups working on summarizing doctor-patient conversations into notes (some use RAG to pull in guidelines).

- **Technical Domain Systems:** Developer assistants like **GitHub Copilot**, **Cursor**, or **Amazon CodeWhisperer** focus on code, but also help with docs by retrieving relevant code examples or documentation. **NVIDIA's NeMo** has a "documentation QA" pipeline where they fine-tune an embedding model (Llama 2 based) specifically for technical Q&A, reportedly boosting accuracy by a large margin. They refer to it as Llama 3.2 NeMo retriever in an NVIDIA blog. Also, **Notion AI** and **Confluence AI** allow querying your knowledge base; under the hood, they use vector search on your pages and an LLM to answer, quite analogous to our plan for enterprise docs. These tools highlight the need for good chunking (they often use paragraph-level chunking and metadata like page titles, similar to our approach).
- **Academic Research:**
  - The original RAG paper by Facebook (Lewis et al., 2020) introduced combining DPR embeddings with generative models for open-domain QA. It sparked this whole line of systems. [arxiv.org]
  - There have been many domain adaptations: e.g., **"Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (Lewis 2020)】 is the seminal one, and others extended it to domains like finance or multi-modal.
  - **Legal RAG papers:** We saw one that combined vector search, knowledge graph, and NMF topic modeling for legal docs. Another 2025 paper, *"Graph RAG for Legal Norms: A Hierarchical and Temporal Approach"*, likely addresses tracking legal changes over time and using a graph of norms. This suggests our inclusion of version tracking and hierarchical linking is in line with cutting-edge ideas. [arxiv.org]
  - **Domain-specific QA**: There are works like *"CaseLaw QA with RAG"* or *"Contract Understanding via QA"* which find that retrieval improves accuracy on tasks like identifying unfair terms (compared to not retrieving).

  - **Chunking research:** The arXiv paper we cited systematically analyzed chunk sizes across datasets, highlighting the trade-offs. Another relevant one from ACL 2023: *"Document Segmentation Matters for RAG"*, which likely discusses intelligent splitting for better answers. Our strategy is informed by such research: essentially, segment semantically and sometimes hierarchically for best results. [arxiv.org]
  - **Long context vs RAG:** Several blog posts and at least one LinkedIn/ACL article debate if long context LLMs obsolete RAG. The consensus so far is exactly what we reasoned: RAG remains more practical for now, and **hybrid approaches** might emerge (like using shorter contexts with retrieval for some parts and long context for others). One ACL 2024 paper (EMNLP Industry Track) is titled "Retrieval Augmented Generation or Long-Context LLMs? A Comprehensive Study and Hybrid Approach", indicating the field is actively benchmarking these approaches. Likely they conclude a combination is best depending on scenario. [dev.to]
  - **LlamaIndex / LangChain:** These open-source frameworks have modules for many data types (PDF loader, SQL loader, etc.). LlamaIndex (formerly GPT Index) in particular is exploring *data connectors* and *query planning* where the system decides which indices to query (similar to our content routing idea). They've introduced the concept of a *ComposedGraph* that can mix knowledge graph and vector index in queries – very akin to GraphRAG. In fact, the Neo4j example we saw is from LlamaIndex's documentation (LlamaIndex Cookbook) connecting LlamaParse, LlamaExtract, and Neo4j. This means that open tooling is aligning well with our design: we might leverage LlamaIndex components for parsing and extraction, then feed into LLMC's indexing pipeline. [neo4j.com]
  - **Haystack** by Deepset is another open-source QA framework. It supports pipelines with retrievers and readers, and has models like **BioGPT** integrated for biomedical QA. We can glean best practices from their docs (they often mention chunking by paragraphs and using domain-specific readers like a BioBERT QA model for medical).
- **Multimodal and Others:** There's a trend to incorporate images or scanned docs. For instance, *LayoutLM* series uses both text and layout info for document understanding. In medical, systems like *EMERGE* mix clinical text and images (like radiology) in RAG. If our enterprise has diagrams or forms, maybe in the future we add a vision component (but for now, we assume text).

In summary, **our approach is at the intersection of many state-of-the-art ideas**: it combines proven techniques (structured chunking, domain embeddings, RAG) with emerging enhancements (knowledge graphs, domain-specific LLM integration). No single off-the-shelf solution does all this locally, but elements are found across projects:

- LLM.co demonstrates the viability of an on-prem domain-tuned RAG for contracts. [llm.co]

- Research like MedRAG and Legal GraphRAG shows effectiveness of injecting knowledge graphs for reasoning. [arxiv.org], [arxiv.org]
- Zilliz/Milvus AI references give concrete best practices for chunking and embeddings which we have adopted. [milvus.io], [zilliz.com], [zilliz.com], [zilliz.com]

We'll align our implementation with these insights, essentially building a system that stands on the shoulders of these efforts but tailored to our specific local data and requirements (e.g., privacy).

7. Implementation Considerations & Architecture Adaptations

Finally, we outline how to implement these ideas in LLMC's architecture, ensuring the solution is **modular, configurable, and reusable** across domains.
Building on the given LLMC v0.6.6 stack, which has components for indexing, embedding, enrichment, graph, retrieval, etc., we will generalize or extend each:

- **File Format Handling:** Unlike code (mostly plain text files), enterprise documents come in PDFs, Word, Excel, etc. We'll integrate robust parsers:
  - **PDF:** Use a library like PyMuPDF (MuPDF) or pdfplumber to extract text while preserving layout info (hierarchy of headings, reading order). We must handle multi-column layouts or footnotes (MuPDF can help identify columns and separate footnotes). If PDFs contain scanned pages (images), integrate an OCR step using pytesseract or an OCR model to get text. This could be toggled by a config flag for images. We should also consider PDF metadata (title, author) and include that. [milvus.io]
  - **Word Documents (.docx):** Use python-docx to read DOCX files directly. This gives us the text along with style information (like what's a Heading1 vs Normal text). We can map heading styles to our chunk hierarchy logic. For older binary .doc, maybe convert to docx via LibreOffice in an automated way if needed.
  - **Excel/CSV:** If some knowledge is in spreadsheets (like an Excel of policy data), we might either exclude those or convert them to CSV and parse. Possibly treat each row as a chunk if the user might query tabular data (though RAG on pure table data could be tricky—an alternative is to load it into a small database and let the assistant query that via a tool, but that's beyond standard RAG).
  - **HTML/Markdown:** Many technical docs or internal wikis are HTML/MD. We can use BeautifulSoup for HTML to extract text by sections (use heading tags to split). For Markdown, use a parser like markdown-it to identify headings, lists, etc. The output we feed to embedding should ideally be cleaned of HTML tags but we can keep some markers for structure (like special tokens for headings that the model might pick up).

    - **Emails:** Often in enterprise, data might include email archives (e.g., EML files). We should parse those (extract fields: To, From, Date, Subject, Body). We can treat email threads as a document with sections by date or treat each email separately. Include context like “From: X, Date: Y” in the chunk text to help answer queries like "emails from my manager last week".
    - **Scanned documents/OCR:** If dealing with legacy scanned records, apply OCR. Using something like Tesseract yields text but loses structure; optionally, one could use ML-based **layout extraction** (like LayoutLM or DocTron) to reconstruct some structure from the image. This might be advanced, so initially ensure text is at least searchable.

Each of these format handlers can be modular. We could implement them as "document loaders" similar to LangChain’s loaders, and auto-select based on file extension. The system should be able to watch a folder and when a new PDF or DOCX appears, use appropriate loader to get text and metadata for indexing.

- **Privacy and Security Considerations:**
    - **PII/PHI Redaction:** Especially for medical data, ensure no sensitive identifiers are inadvertently exposed. If our user is authorized to see the data, we can keep PHI in context; but for extra safety, we might apply an automatic anonymization to the text and possibly re-identify after answer if needed. We can integrate a list of patient names or use an NER model to blank names, IDs, etc., in the index. This avoids any chance of an LLM leaking a name where not appropriate.
    - **Confidential Docs:** Some enterprise docs might be highly confidential. If the RAG system is used by different users with different permissions, we need **access control** at retrieval time. We can include in metadata the access level or user roles allowed. Then, the retrieval component should filter out chunks the current user shouldn’t see. For example, if an HR policy is confidential to HR, and an engineering user asks a question, the system should not retrieve that content. This implies passing user context into the search query (supported by vector DBs via filtered search on metadata).
    - **Encryption:** The index (SQLite or vector DB) might contain sensitive text embeddings which could be reversed to text to some extent. If this is a concern, store data on encrypted disk. Also secure the knowledge graph store if it contains sensitive relationships. Ensure any backups are handled securely too.
    - **No External Calls:** Given local-first, we avoid external APIs. But if we ever consider using something like OpenAI for a particularly tough query, that must be off by default for sensitive domains. Perhaps provide a config to allow a cloud LLM fallback for less sensitive data (some companies do a tiered approach: try local model, if not

confident and if data is non-sensitive, call cloud GPT). In our case, likely we keep everything internal to be safe.

  - **Monitoring and Logs:** Logging should be careful not to log sensitive content in plaintext. We might log query and maybe document IDs retrieved, but not the full content.

- **Scalability (Data Volume and Model Size):**
  - If we have tens of thousands of documents (like a whole contract repository or thousands of patient notes), our chunk count might be in the hundreds of thousands. SQLite with FAISS index can handle that on a single machine (FAISS is C++ optimized for millions of vectors). But beyond that, consider external vector stores:
    - **Milvus** or its cloud cousin Zilliz are built for scalability (billions of vectors) and support filtering. They can run on our hardware (with GPU accel). We saw an example where Milvus easily handled 2k docs, but it's known to scale to much more. [aclanthology.org]
    - **FAISS** itself can handle up to millions in memory. On 128GB we could store many embeddings even at 768 dims. For instance, 1 million embeddings of dim 768 float32 ~ 3GB, which is okay. The challenge is realtime updates – FAISS IVF indexes are not as easily updateable for dynamic data, but if our data is mostly append-only or small updates, it's fine. There's FAISS "IndexHNSW" which allows insertions.
    - **Qdrant** or **Pinecone** are other vector DBs; Qdrant is open source and can be self-hosted. Could consider if needing more features. However, to keep dependencies minimal, we might stick to FAISS or Milvus.
  - **Index Sharding:** We may separate indices by domain (already doing by route). That also partitions the data which improves search speed by focusing on smaller indexes for each domain. If a query clearly falls under legal, we query only legal index.
  - **Memory for Models:** Running multiple domain models could be heavy (e.g., a Legal-BERT and a ClinicalBERT in memory simultaneously). But since they are base-size (~110M each), that's manageable (couple hundred MB each). The bigger ones like Qwen-3 4B for embedding – 4B parameters for an embedding model is large (~8 GB in FP16, or ~4 GB in int8). If we use it across all content, okay. If we use different models for each domain, that's multiple 4B models which might strain GPU memory. We'll need to allocate carefully or possibly run some on CPU if real-time speed isn't needed for all (embedding new docs can be slower if on CPU).
  - **Batching and Async:** To index a lot of docs, do in batches to utilize the GPU (most embedding models support batching multiple texts per forward pass). Also use

asynchronous processing for file watching: LLMC's daemon can enqueue new files and process in background so that user queries aren't blocked by indexing jobs.

- **Incremental Updates and Versioning:**
    - Use content hashing (like SHA256 already in LLMC) to detect changes in files. For each document, store a hash of its text. On re-index runs, if hash changed, update that doc's chunks in the index (remove old ones, add new ones). Since documents have stable identifiers or paths, we can tie chunks to a doc ID for easy purge and replace. [aclanthology.org]
    - If maintaining **historical versions**, we could either treat the old version as a separate document (with metadata version=old, maybe archived) or replace it. If the use-case requires comparing versions (like comparing two versions of a contract), we likely need to keep both and have an edge between them. Configuration could allow turning on retention of old versions in the index but marked as such.
    - Consider real-time search for content not yet indexed: If user uploads a new file and immediately asks a question, our system should either (a) index it near real-time (with file watchers, indexing might complete within seconds of save), or (b) have a fallback to directly read the file. LLMC does fallback to live file read if RAG is stale. For docs, maybe easier to ensure indexing is run immediately on file change events. We can leverage FS events on the document repository. The architecture already has event-driven file watching that triggers re-index. [arxiv.org]
    - **Superseded documents:** If we mark a doc as superseded by another, our retrieval should de-prioritize or exclude superseded ones unless explicitly asked. We could implement a metadata flag and a query filter to skip superseded if a newer exists.
- **Configuration and Flexibility:**
    - All domain-specific behaviors (which model, which chunking splitter, which enrichment steps on or off, which graph relations to use) should be configurable via the llmc.toml or a similar config. For example:

TOML
```
[domains.legal]

enabled = true

embedding_model = "nlpaueb/legal-bert-base-uncased"

use_clause_extraction = true
```

```
use_graph = true


[domains.medical]

enabled = true

embedding_model = "emilyalsentzer/Bio_ClinicalBERT"

use_summary = true
```

Show more lines

This way, switching a methodology is as easy as flipping a flag. The code should be modular: e.g., a function enrich_legal(document) that calls subroutines if corresponding flags are true. We avoid hardcoding things in one big pipeline.

    - Possibly allow *pluggable prompts* for LLM enrichment so that users can tweak how extraction is done (some might prefer to fine-tune a model rather than rely on prompt, etc.).
    - By designing around interfaces (e.g., a generic DocChunker interface implemented by LegalChunker, MedicalChunker, etc.), we make it easy to add a new domain in future (like "financial reports" domain with its own logic).

- **Reuse of Code:** Many functions will be similar across domains (e.g. splitting by headings, summarizing sections). We should factor common utilities (like a generic function to split by headings for any doc with Markdown headings). Domain-specific code extends or configures these utilities. For instance, we might have a generic regex splitter but feed it domain-specific patterns. Reuse ensures we're not writing completely separate logic unnecessarily.
- **Integration into LLMC Stack:**
    - **Routing:** Expand the slice_type_to_route map with our new domain types (as shown in the prompt snippet). Content classification: Instead of just relying on file extension, we might do a quick classification of text to decide if something is legal vs general docs (e.g., if it contains "WHEREAS" or many section numbers -> legal). This can be a simple heuristic or a small ML classifier. Alternatively, user might put different kinds of files in different folder paths which we know (like /contracts vs /policies), and assign domain based on path.
    - **Index Metadata Store:** LLMC uses SQLite to store metadata about each chunk (file path, etc.). We'll extend the schema to store domain-specific metadata we generate (like clause type tags, patient id, etc.). Possibly a flexible key-value table for chunk

metadata, or store as JSON in a column if that's easier. We need to ensure efficient filtering: maybe add indices on common fields like patient_id for speed.

- **Enrichment Pipeline:** We likely implement this as part of indexing: once chunks are made, run enrichment (some can be done on entire document before chunking too, like classifying doc type). This yields extra metadata or even additional synthesized chunks (like a contract summary chunk, or a list of all obligations as a pseudo-chunk). In the stack diagram, enrichment was separate stage after embedding – but if our enrichment yields extra text (like summaries), we might embed those too or at least attach to the corresponding chunk.
  - The LLMC backend cascade idea (local -> cloud) can apply if e.g. we try a local small model to generate a summary, if it fails or is low confidence, fallback to a bigger model, etc.. We can incorporate that to control cost. [arxiv.org]
  - Also include any **safety filtering** as done for code (they had "Latin-1 safety and garbage filtering" to remove weird data). For text, perhaps filter chunks that are extremely short or meaningless (these likely come from OCR noise). [aclanthology.org]
- **Graph storage:** Where to store the graph? Could leverage SQLite to store edges too (since SQLite can be used for small graph queries via recursive CTEs). But better to use a graph DB for complex queries. However, for MVP, a simple approach: implement a set of adjacency lists in Python or in memory (since our graph is not enormous likely) and use it in retrieval phase. For example, maintain a dict: refs_index: {chunk_id: [referenced_chunk_ids...]} and when retrieval results come in, extend the set with those references. This is simple but covers the main internal cross-ref use case with minimal infra. For more advanced queries or large graphs (like thousands of nodes of case law network), a Neo4j instance accessible via Cypher query from Python would be beneficial.
- **MCP/Agent tool integration:** The architecture mentions *MCP/agent tool integration* in service layer. Possibly that stands for a protocol to allow the LLM to call tools. In our context, one might allow the LLM to explicitly query the graph or lookup a definition via a tool. For example, an agent could decide: "I have a question about a defined term, let me call a definition_lookup(term) tool that queries the knowledge graph or metadata DB for that term's definition." This is an advanced use of LLM as an agent utilizing the structured data we've built. We could design some basic tools (like a function to fetch specific metadata) and allow the LLM (if using a capable model) to use them. However, this complicates the conversation management. It

might be optional for later – initial system can do straightforward retrieve-then-answer without dynamic tool use. But it's good to note the possibility. [arxiv.org]

- **Choice of Local Models for Generation:** The user specifically mentioned Qwen-3 4B being used currently (presumably for code tasks). For our domains, we have options:
    - Possibly use the same Qwen-3 4B or 32B for answer generation across all domains, since Qwen (by Alibaba) is known to be a general-purpose chat model like ChatGPT. It likely wasn't specifically tuned on legal/medical, but with good context (provided by RAG) and instructions, it might perform decently. Qwen-3 4B is quite small though – might struggle with very complex legal reasoning or medical correctness. If we have the compute (128GB could host a larger model), maybe consider **Llama-2 70B** or **Qwen-3 32B** as the answer generator for better quality. Qwen-3 32B in 4-bit would be ~8GB, which is fine. That model (if as good as GPT-3.5) could be a workhorse. Also, there's **Code Llama 34B** or others if needed for technical answers that might involve code (though primarily we are doing retrieval, not writing code).
    - For especially domain-specific answers, one could fine-tune a LoRA on top of these for tone or style (like making a "Legal-Qwen" by fine-tuning on some legal Q&A to sound more authoritative or cite properly). But initially, base models with system prompts to enforce citing should do.
- **Testing Plan**: Use the evaluation methods from section 5 during implementation to test each piece. E.g., after building chunker, test on sample documents that it splits correctly. After embedding integration, compute similarity on a known pair to ensure model is working. After full pipeline, run on known QA pairs.

MVP Scope and Flags:

For an MVP (Minimum Viable Product), we might **choose one domain to fully implement first**, to validate the approach:

- A strong candidate is the **legal domain (contracts)** because it's self-contained, text-heavy, and high value. We would implement chunking by clauses, legal embedding, extraction of clause types and maybe one or two things (like party names), and basic cross-ref graph. Then test with some queries (like from an NDA or MSA).
- Alternatively, if the user urgent need is healthcare, we start with **medical** on a sample set of de-identified notes, implement section chunking and med embeddings.

However, since user didn't prioritize a single domain, we can do a thin slice of each to prove generality:

- Maybe implement basic chunk & retrieval for all, but enrichment and graph for just 1 or 2 as a demonstration (because enrichment/graph is most complex to do for all at once).

The MVP should allow easily **switching on/off features**:

- e.g., run in a mode with graph disabled to compare.

- Or skip enrichment to see how much it affects results (maybe as an ablation).

Success Criteria:

- The system should answer questions with factual correctness using information from the documents, with citations, in each domain scenario.
- For an example query: *"What is the penalty for late delivery in Contract X?"*, it should return the relevant clause text (or summary) and cite it, rather than a generic answer. Or a medical example: *"Does the patient have any drug allergies?"* -> it should find that in the notes and respond accordingly, citing the note.
- Quantitatively, we might target something like >90% Recall@5 on our test queries and no major hallucinations in a set of evaluated answers.

By meeting those, we ensure the extended LLMC meets the needs of domain experts relying on it. The design is flexible enough to incorporate further advancements (like new embedding models or adding a new domain like finance or manufacturing manuals) by just adding new configurations and maybe extraction rules, not rewriting the core.

Conclusion: We have detailed how to extend LLMC's code-centric RAG to handle legal, medical, technical, and enterprise text corpora. We will preserve semantic integrity through intelligent chunking, leverage domain-specific embeddings for better recall, enrich documents with structured data (from clause labels to ICD codes) for targeted querying, introduce knowledge graphs to map relationships beyond the surface text, and rigorously evaluate the system on domain-specific criteria. All of this while keeping the system local, secure, and configurable. The outcome will be a powerful, multipurpose RAG system enabling users to query and reason over their unstructured documents with confidence and accuracy, much like they've been able to do with code using LLMC, now across all these rich text domains. [milvus.io] [zilliz.com], [zilliz.com] [llm.co], [zilliz.com] [arxiv.org]