**Concept design**

**Building a Knowledge Base with RAPTOR and LangChain for Complex Documents**

Building an internal knowledge base for complex documents (manuals, SOPs, regulatory texts, FAQs, etc.) can be greatly enhanced by LangChain’s RAPTOR approach. **RAPTOR** (Recursive Abstractive Processing for Tree-Organized Retrieval) augments retrieval-augmented generation (RAG) by hierarchically clustering and summarizing documents into a tree structure ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=RAPTOR%C2%A0introduces%20a%20novel%20approach%20to,limitations%20in%20traditional%20language%20models)). This enables more efficient, context-aware information retrieval across large documents, addressing limitations of traditional flat chunk retrieval. In this guide, we outline best practices for implementing a RAPTOR-powered knowledge base on a local Ubuntu server – from document ingestion and embedding to vector indexing, hierarchical retrieval, and deployment.

**Document Ingestion and Chunking**

**Flexible File Loaders:** Start by ingesting documents in various formats (PDF, DOCX, TXT, etc.) using LangChain’s document loaders. LangChain provides loaders for many file types – e.g. PyPDFLoader or UnstructuredPDFLoader for PDFs, UnstructuredWordDocumentLoader for Word files, TextLoader for plaintext, CSV loaders for spreadsheets, etc. You can implement a *dynamic loader* selection based on file extension or use the GenericLoader utility for a generic solution ([Dynamic document loader based on file type · langchain-ai langchain · Discussion #10507 · GitHub](https://github.com/langchain-ai/langchain/discussions/10507#:~:text=Yes%2C%20LangChain%20does%20provide%20an,JSON%20files%2C%20and%20so%20on)) ([Dynamic document loader based on file type · langchain-ai langchain · Discussion #10507 · GitHub](https://github.com/langchain-ai/langchain/discussions/10507#:~:text=In%20addition%20to%20these%2C%20LangChain,GenericLoader)). For example, GenericLoader.from\_filesystem() can iterate through a folder and apply the appropriate parser to each file type automatically. This modular design lets you easily add new file type support in the future.

**Text Extraction for Complex Content:** Many enterprise documents contain images, tables, and complex layouts. A best practice is to use robust parsing tools (like **Unstructured.io** which LangChain integrates) to extract text from such documents. The LangChain RAPTOR demo explicitly leverages Unstructured for parsing PDFs with tables/figures ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=Python.%20,and%20packaging%20tool%20for%20Python)). This ensures that even complex layouts (e.g. multi-column manuals or forms) are converted into text content for the knowledge base. If tables or images contain critical info, consider specialized extraction (e.g. OCR for images, CSV extraction for tables) or include the parsed output in document metadata so it isn’t lost.

**Chunking Documents:** Once raw text is extracted, split documents into manageable chunks for embedding. Long documents should be broken into smaller *semantic* units (e.g. sections or paragraphs) because language models and embeddings perform better on moderately sized texts. LangChain’s RecursiveCharacterTextSplitter (or MarkdownHeaderTextSplitter if the document has headings) can chunk text while preserving section boundaries. A typical chunk size is on the order of a few hundred words (e.g. ~1,000 tokens), adjustable based on content complexity. In the RAPTOR example, chunks of up to ~2000 tokens were used ([RAPTOR.ipynb](file://file-vddtihmx9rkxzjhcdfqriy%23:~:text=\",/)) – large enough to encompass a full section or topic. You generally want chunks that are self-contained in meaning (to avoid splitting mid-topic) but not so large that they exceed your model context or make clustering difficult.

* **Avoiding Information Loss:** It’s wise to avoid cutting off important sentences or losing context at chunk boundaries. Using the recursive splitter will try larger splits (by paragraph, sentence, etc.) before falling back to character length, which helps maintain coherence. You can also allow a small overlap between chunks (e.g. 50 tokens) to ensure continuity for borderline cases. Each chunk will be stored as a LangChain Document object with relevant metadata (source file name, page number, section title, etc.), which is useful for tracing answers back to origin or filtering by document type.

**Metadata Enrichment:** As you ingest documents, capture key metadata. For instance, tag each chunk with the document title or category (manual vs. policy vs. FAQ), language, creation date, etc. This metadata can later be used to filter search results (e.g. only retrieve from certain manuals) or to provide provenance in answers. Metadata is stored in the Document.metadata field in LangChain. A well-designed ingestion pipeline will allow easy extension of metadata fields as needed.

**Embeddings and Multilingual Retrieval**

**Choosing an Embedding Model:** For a local deployment with multilingual documents, **Hugging Face’s intfloat/multilingual-e5-large** is an excellent choice for the embedding model. The E5 models are state-of-the-art, trained for embedding **queries** and **passages** such that relevant pairs have high cosine similarity. Notably, multilingual-e5-large supports **100+ languages** (being based on XLM-RoBERTa) ([intfloat/multilingual-e5-large · Hugging Face](https://huggingface.co/intfloat/multilingual-e5-large" \l ":~:text=Supported%20Languages)), making it suitable if your knowledge base content or user queries span multiple languages. This model produces 1024-dimensional embeddings (since it’s derived from a RoBERTa-large architecture), providing rich semantic representation.

* **Embedding Usage Tips:** The E5 model expects special prefixes on input text to differentiate query vs. document context. For optimal results, prefix each document chunk with **"passage: "** and each user query with **"query: "** before embedding ([intfloat/multilingual-e5-large · Hugging Face](https://huggingface.co/intfloat/multilingual-e5-large" \l ":~:text=,see%20how%20much%20protein%20you)). This matches the model’s training format and significantly improves retrieval performance by aligning embeddings in the shared vector space. In practice, you can wrap the model with a small helper that adds the prefix and then uses HuggingFace Transformers to compute the embeddings in batches. Batch processing will speed up embedding generation when ingesting many documents.

**Alternatives:** If for some reason E5-large is not ideal (e.g. resource constraints), other options include smaller sentence-transformer models (like all-MiniLM variants) or multilingual instructors. Keep in mind that smaller models (with 384-dimensional embeddings, for example) might be faster but could sacrifice some accuracy ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=OpenAI%20embeddings%20and%20384%20in,source%20small%20embedding%20models)). OpenAI’s text-embedding-ada-002 (1536-dim) is another high-quality choice ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=OpenAI%20embeddings%20and%20384%20in,source%20small%20embedding%20models)), but it requires Internet access/API and primarily supports English (with limited multilingual capability). For an entirely offline system with strong multilingual support, E5-large strikes a good balance. If you need even more efficiency, you could explore quantizing the model or using a GPU to serve embeddings, but these are optimizations rather than fundamental design choices.

**Cross-Lingual Considerations:** With a multilingual embedding, your system can handle cross-lingual retrieval: for example, an English query can retrieve a relevant chunk from a Spanish document if their embeddings are similar. This is very useful for international or multilingual documentation. However, ensure that the language of each chunk is either identified in metadata or the embeddings are truly language-agnostic. The E5 model is trained on many languages, but extremely domain-specific jargon in one language might still pose challenges. In practice, test a few queries in different languages to verify that the retrieval returns expected results across languages.

**Choosing a Vector Store (FAISS vs. Alternatives)**

To enable fast similarity search over the embeddings, you’ll need a vector store or index. The preference here is **FAISS** (Facebook AI Similarity Search), which is a proven library for efficient nearest-neighbor search on dense vectors ([Faiss | ️ LangChain](https://python.langchain.com/docs/integrations/vectorstores/faiss/" \l ":~:text=,See%20The%20FAISS%20Library%20paper)). FAISS is well-suited for local deployment and can handle large collections (millions) of vectors through efficient indexing algorithms. In LangChain, you can use FAISS.from\_documents(documents, embedding) or FAISS.from\_texts(texts, embedding) to create an index, and then call .as\_retriever() on it to integrate with QA chains.

**FAISS Benefits:** FAISS is highly optimized in C++ with Python bindings. It supports various indexing strategies (exact search, IVFFlat for big data, etc.), and importantly it can be saved to disk for persistence (e.g. faiss\_index.save\_local("index\_folder") in LangChain) and loaded back when the server restarts ([How to properly initialize ChromaDB instead of FAISS in order to ...](https://github.com/langchain-ai/langchain/discussions/5682#:~:text=How%20to%20properly%20initialize%20ChromaDB,Here%20is%20a%20small)). Benchmarks have shown that FAISS tends to have excellent **accuracy and speed**. In one comparison, using the same data and queries, FAISS slightly outperformed Chroma in both precision/recall of retrieved context and query latency ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Changing%20the%20vector%20store%20influences,the%20preferred%20choice%20over%20Chroma)). FAISS performed ~17% faster (1.81s vs 2.18s for 50 queries in a test) and retrieved correct context more often, making it a reliable choice ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=As%20indicated%20in%20Table%201%2C,18%20seconds)) ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=differences%20in%20numbers%20are%20marginal,18%20seconds)). In sum, FAISS’s mature and efficient search algorithms often make it the **preferred choice** for local vector retrieval ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Changing%20the%20vector%20store%20influences,the%20preferred%20choice%20over%20Chroma)).

**Alternative Vector Stores:** It’s worth considering alternatives if they offer specific advantages your use-case needs:

* **ChromaDB:** An open-source embedding database that wraps an ANN index (HNSW) with a user-friendly API and persistent storage (using SQLite or DuckDB under the hood). Chroma is easy to use and allows metadata filtering natively. However, by default it uses approximate search, which can introduce slight nondeterminism in results ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Why%20is%20there%20discrepancy%20between,vector%20stores)). For moderate-scale data, FAISS (with exact search or tuned ANN) might be more accurate as noted. Chroma is a fine choice if you want a plug-and-play persistent DB and are okay with its performance trade-offs.
* **Milvus or Weaviate:** These are standalone vector database services that can run on-premise. They are scalable and feature-rich (advanced indexing, filtering, etc.), but they introduce extra moving parts (a separate service to manage). If your knowledge base grows to tens of millions of documents or you need distributed indexing, such systems could be considered. For a simpler local deployment, they may be overkill.
* **Qdrant:** Another lightweight, high-performance vector DB that can run via a single binary (written in Rust). It supports filtering and CRUD on vectors. If you anticipate a need to frequently update the index (add/delete vectors) or use hybrid filtering queries, Qdrant or Chroma might simplify those operations compared to raw FAISS.

In practice, for an internal KB on Ubuntu, many teams start with **FAISS (in-memory)** for its speed and simplicity, and possibly move to an external DB only if scaling demands it. You can also combine approaches: e.g. use FAISS for pure semantic search and then apply metadata filters in application code, or periodically rebuild the FAISS index as documents update. Just remember to persist your index to disk so you don’t have to recompute embeddings on every restart. LangChain’s integration for FAISS and others makes swapping vector stores relatively straightforward if requirements change.

**Hierarchical Clustering and Summarization (RAPTOR Method)**

Once documents are chunked and embedded, **RAPTOR** introduces a hierarchical compression step before final indexing. The core idea is to cluster similar chunks and generate *summaries* for each cluster, then repeat this process to form a tree of summaries. This creates multiple levels of abstraction: leaf nodes are original text chunks, higher-level nodes are summaries that synthesize information from several related chunks ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=In%20short%2C%20the%20intuition%20behind,RAPTOR%20as%20follows)). By doing this, RAPTOR can capture broader context that spans multiple sections, enabling the system to answer high-level questions more effectively (since a summary node might combine info that was scattered in different parts of a manual).

([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/)) **Figure:** A conceptual diagram of the RAPTOR tree architecture. Each colored block (1–5) represents an embedded text chunk from the documents (leaf layer). Similar chunks are **clustered** together (grouped by the clustering algorithm), and an LLM generates an **abstractive summary** of each cluster – shown as yellow nodes (nodes 6, 7, 8 in the middle layer). These summaries become new nodes in the tree with their own embeddings. The process can repeat: the summary nodes themselves are clustered and summarized into higher-level nodes (e.g. root nodes 9, 10), until the tree is built up to a root or until summaries are short enough for the model’s context window. Each node thus contains a condensed representation of the information from its child nodes, allowing multi-scale understanding of the knowledge base.

**Clustering Strategy:** In practice, you will use a clustering algorithm to group embedding vectors. The LangChain RAPTOR example explored using **UMAP** (for dimensionality reduction) followed by **Gaussian Mixture Models (GMM)** for clustering ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20clustering%20approach%20in%20tree,includes%20a%20few%20interesting%20ideas)) ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=GMM%20)). The rationale is that raw embedding vectors might be 384–1536 dimensions, which is computationally heavy for clustering directly; reducing them to a lower dimension (e.g. 5–50 dims) with UMAP can improve clustering efficiency without losing much semantic structure ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=Suppose%20we%20have%208%20document,source%20small%20embedding%20models)). GMM with BIC evaluation can estimate an optimal number of clusters ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=GMM%20)), or you could use simpler methods like K-Means with a fixed k or hierarchical clustering with a distance threshold. The key is to cluster chunks such that those in each cluster are topically related.

* *Example:* If you have a 200-page technical manual, the initial chunks might naturally cluster into groups corresponding to each chapter or section (e.g. all chunks about installation procedures cluster together, safety guideline sections cluster together, etc.). Each cluster will then be summarized.

**Summarization of Clusters:** For each cluster of chunks, use a **large language model** to produce an abstractive summary that captures the key points of that cluster. This is effectively a compression step – condensing, say, 10 chunks (which might be ~10k tokens combined) into a summary of a few hundred tokens that still preserves the important information. The summary content will serve as a new document representing that cluster. You should embed the summary text as well (using the same embedding model) so that it can be treated like any other node in vector search.

* **Prompting the LLM:** When summarizing, prompt engineering is important. Since this is an internal knowledge base, you may want the summaries to be factual and *information-dense* (rather than overly general). A good prompt might be: *"Summarize the following content in a concise paragraph, capturing all important details and terminology. Focus on the key points relevant to X."* You can include the cluster’s combined text (truncated to the model’s input limit). If the cluster text is too long, you might summarize in parts or use a smaller summary first then refine. Ensure the summarizer knows to **preserve critical data** (e.g. regulations or numeric values) to avoid losing important specifics. This step can be done with an API model (like GPT-4 or Claude) or a local model if available (for example, a fine-tuned Llama 2 might handle it if the context fits).

**Recursive Tree Construction:** After the first round of clustering and summarizing (producing first-level summary nodes), you can iterate the process: treat the new summaries as a smaller set of “documents” and again cluster them, summarize to form a second-level of nodes, and so on ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=and%20then%20run%20a%20dimensionality,source%20small%20embedding%20models)). This forms a **tree** where the leaves are original chunks and each higher layer node is the summary of a cluster of nodes below it. You continue until you either (a) reach a single root summary that represents the entire corpus, or (b) the summaries at the current layer are already short enough to fit within the target context window (e.g. each summary is under, say, 1500 tokens, such that a handful of them could be fed into the QA prompt). In practice, you might stop when clusters become very broad or semantically distinct (e.g. you might end up with 2–3 top-level summaries covering very different topics in your knowledge base).

Some key points summarizing the RAPTOR approach ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=In%20short%2C%20the%20intuition%20behind,RAPTOR%20as%20follows)) ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=is%20capable%20of%20synthesizing%20information,answering%20tasks)):

* **Cluster and Summarize:** Group similar document chunks and generate a summary for each cluster using an LLM. This captures the essence of multiple related pieces in one chunk of text.
* **Hierarchical Context:** By repeating this clustering->summarizing process, build a hierarchy of summaries. Higher-level summaries synthesize information from a wider scope, while lower-level (leaf) chunks remain detailed. This tree structure encodes the documents at multiple levels of granularity.
* **Efficient Multi-Document Answers:** RAPTOR enables answering questions that span multiple sections or documents using a **smaller set of summaries**. Instead of retrieving 10 full chunks and throwing them all into a prompt (which might hit context limits), the system can retrieve a couple of summary nodes that already integrate those 10 chunks’ info. This provides relevant context with fewer tokens, making retrieval augmented generation more efficient ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,can%20be%20achieved%20using%20fast)) ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=k,aligning%20with%20model%20context%20constraints)).

It’s important to note that constructing this tree is an **offline** or preprocessing step. It can be computationally intensive (especially if you have thousands of chunks to cluster and call an LLM for each cluster summary), but it only needs to be done once (or whenever you update the knowledge base). The result is a collection of summary documents (nodes) in addition to the original chunks.

After applying RAPTOR, your final set of content to index will include both the original chunks **and** the summary nodes from all levels of the tree. Each of these has an embedding vector and text content. This is what we will feed into the vector store for query-time retrieval.

**Efficient Retrieval Strategy with RAPTOR**

With the RAPTOR tree built, how do we use it during query time? There are two main retrieval strategies:

([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/)) **Figure:** Two retrieval strategies using the RAPTOR tree. **(A) Tree Traversal Retrieval:** The query is embedded (Q) and compared layer-by-layer through the tree structure – e.g. first find the most relevant top-level node, then search within its child nodes, and so on down to leaf chunks. **(B) Collapsed Tree Retrieval:** The query embedding is compared against *all nodes at all levels simultaneously* (a “flattened” index of every summary and chunk), retrieving the top-K most relevant nodes in one step. The retrieved context (pink blocks) is then fed, along with the original query, into the LLM to generate the answer. In practice, the **collapsed tree approach is preferred** because it allows dynamic selection of information from any level of granularity in one search ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,can%20be%20achieved%20using%20fast)). By searching across every node, the retriever can find either a very specific chunk or a high-level summary, whichever is most relevant to the question, without being forced to traverse each layer.

**Why Collapsed Retrieval?** As the LangChain blog notes, collapsed retrieval offers greater flexibility and often **better performance** than strict hierarchical traversal ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,method%20with%202000%20maximum%20tokens)). It effectively treats the entire tree (all summaries + leaves) as one big vector index. The benefit is that the similarity search can surface a broad summary if the query is broad, or a narrow detail if the query is very specific. This improves relevance and comprehensiveness, since RAPTOR can adaptively pick up information from different layers tailored to the question at hand ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,can%20be%20achieved%20using%20fast)). The blog’s authors found this method yielded optimal results when summary nodes were capped at around 2000 tokens each, aligning with typical LLM context constraints ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=k,aligning%20with%20model%20context%20constraints)).

In implementation, using collapsed retrieval means when building the FAISS (or other) vector store, **include all nodes** (original chunks and all summaries). Each node’s text and metadata should indicate its level or source. At query time, you embed the user’s question (remember to prefix as "query:") with the same embedding model, and perform a similarity search in the vector store to fetch, say, the top *k* relevant nodes. The results may be a mix of original text chunks and summary nodes. You can then pass the content of those retrieved nodes into your LLM prompt to answer the question.

**Using Retrieved Summaries:** One nuance with retrieving summary nodes is deciding what to do with them in the final answer generation. Often, you can treat them just like any other context – the summary text itself can be provided to the LLM, which will often be enough to answer a high-level question. However, if the user asks for very detailed information, and the retriever returned only a summary node, you might need to fetch that summary node’s child chunks for details. A simple approach is: if a top-level summary is retrieved and the question is asking for specifics (you could detect this via keywords or the summary being too high-level), you could do a secondary retrieval constrained to that summary’s children. This is essentially a hybrid of traversal (drill down when needed) on top of collapsed retrieval. In many cases though, the combination of summary and original chunks in the top-K results provides a sufficient mix of context. For example, a *regulatory question* might retrieve a summary of a regulation section *and* a specific clause chunk, covering both broad context and exact text.

**Number of Retrieved Documents:** Deciding how many nodes to retrieve (k) is another tuning point. Because RAPTOR nodes pack more information (a summary represents multiple chunks), you may not need as many as you would in a non-hierarchical system. You might start with k=5 (for instance, retrieve the top 5 nodes). If your summaries are well-crafted, even 3 nodes could be enough (as indicated by the RAGAS evaluation in one study where retrieving 3 documents gave high QA scores ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Retrieve%20Multiple%20Context%20Documents))). However, if the user’s query is very specific (requiring a particular detail), ensure that at least one of the retrieved nodes is an original chunk containing that detail. It’s wise to test various queries and adjust k (and the mix of summary vs. leaf nodes if needed) to balance answer completeness vs. prompt length.

In summary, the retrieval phase will use the vector similarity search to identify relevant context quickly. Thanks to RAPTOR, the **retrieved context is both concise and rich** – summaries provide high-level coverage while original chunks provide precision. This approach has been shown to outperform traditional flat retrieval in QA tasks, setting new benchmarks in their experiments ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=is%20capable%20of%20synthesizing%20information,answering%20tasks)).

**System Architecture and Deployment Considerations**

Design the system in a **modular pipeline** fashion for scalability and maintainability. A recommended architecture on a local Ubuntu server is as follows:

1. **Ingestion Module:** A script or service responsible for document loading, parsing, and chunking. This module can scan a directory (or multiple sources), use LangChain loaders to read files, split them into Document chunks with metadata, and store these (perhaps temporarily on disk or directly in memory for the next step). By separating this, you can easily re-run ingestion when new documents are added or existing ones change. Make sure to log or output any chunks and their metadata (for auditing what content is in the knowledge base).
2. **Embedding & Indexing Module:** This component takes the chunks and processes them into the vector store. It will load the embedding model (e.g. the E5-large model via HuggingFace) – ideally once at startup to avoid re-loading weights repeatedly – and then embed all chunks. If using RAPTOR, this module will also perform the clustering & summarization steps:
   * Compute initial chunk embeddings.
   * Run clustering (using libraries like scikit-learn, UMAP, etc.).
   * Call the LLM to summarize each cluster into a new text.
   * Embed those summaries and repeat if doing multiple hierarchy levels.
   * Finally, load all final nodes (original chunks + all summaries) into the vector store (FAISS index).

This can be an offline batch process. For large document sets, it might take some time (embedding thousands of chunks and summarizing clusters is the heavy part), so you might run this as a separate job (perhaps triggered nightly or upon document updates). On Ubuntu, you can orchestrate this with a cron job or a manual trigger. Ensure that the resulting FAISS index is saved to disk (so that it can be reused by the query service without reprocessing). For example, FAISS.from\_documents returns a FAISS object which has a save\_local() method to persist the index and accompanying metadata.

1. **Query Service:** This is the online service (e.g. a FastAPI or Flask app, or a Streamlit interface) that handles user queries in real-time. It will:
   * Load the saved FAISS index (and ensure the embedding model is ready for query embedding).
   * On a query, embed the question using the same embedding model.
   * Perform similarity search on the index to retrieve top-K relevant nodes (using RAPTOR’s collapsed retrieval as discussed).
   * Compose a prompt to an answer-generating LLM that includes the query and the retrieved context. This could be a simple few-shot prompt or a LangChain QA chain (e.g. RetrievalQA chain) that you configure. The prompt might say: *“Using the following documents, answer the question…”* and list the content of the retrieved chunks/summaries.
   * Get the answer from the LLM and return it to the user (optionally with source citations if you include source metadata in the context).

The LLM for answering queries can be an OpenAI API call or a local model (depending on your deployment). If data privacy is a concern and you want to stay fully on-prem, you might use a local LLM like Llama 2 or GPT4All for generation – though their quality may vary. You could also use smaller local models just for the summarization steps and still use an external API for final Q&A if that’s acceptable. This flexibility in model choice is another reason to design these as separate components.

**Multilingual Query Handling:** Because the system uses multilingual embeddings, the query service can accept questions in different languages. You might implement language detection on the query (or even on each chunk at ingestion) to log or route certain behavior, but typically you don’t need to translate queries – just embed and search. If using a multilingual LLM for answering, it could even answer in the language of the query. For internal use, you might restrict answers to a certain language (e.g. always answer in English for consistency) – that can be controlled by the prompt given to the LLM.

**Scalability and Performance:** On a single Ubuntu server, you should be mindful of resource usage:

* The embedding model (E5-large) will consume memory (roughly a few GB for 1k+ model parameters). Make sure the machine has enough RAM, or use a GPU for faster embeddings if available. You can also explore running the embedding model in 8-bit quantized mode to save memory.
* FAISS search is very fast in-memory; the main time cost per query will be embedding the query (which is quick, a few hundred milliseconds) and the LLM generation. To scale to many concurrent queries, you might want to instantiate a pool of models or API calls. If using an OpenAI/Anthropic API, concurrency is handled by the service (but rate limits apply). If using local models (for either summarization or QA), consider running them on GPU and possibly using libraries that support concurrency (or run multiple model instances for multi-threaded serving).
* The clustering and summarization step is the most expensive offline phase. Its scalability will depend on how many chunks you have and how large your model context is for summarization. If you have extremely large documents (hundreds of thousands of chunks), a multi-level hierarchy is essential; you might also use a more sophisticated approach like doing a first pass with embeddings to filter or divide the corpus by topic to make clustering tractable. Fortunately, internal documentation often comes segmented (by department, year, etc.), so you can cluster within subsets to parallelize the work.

**Modularity and Maintainability:** Structure your code so each part (ingestion, embedding, clustering, etc.) is in a separate function or class. This makes testing easier (e.g. you can unit test that PDF loader works, or that summary generation produces something reasonable). It also allows updates – for example, if you decide to switch the embedding model or vector store later, you can do so without overhauling the whole system. Similarly, keep configuration (like chunk size, number of clusters, k for retrieval, model names, etc.) in a config file or easily adjustable section.

**Deployment on Ubuntu:** When deploying on Ubuntu, ensure all dependencies (LangChain, Transformers, FAISS, Unstructured, etc.) are installed and compatible. You might use a virtual environment or Docker to encapsulate the environment. Given the need for packages like faiss-cpu, umap-learn, scikit-learn, and potentially system libs for Unstructured (like poppler for PDFs), using a Dockerfile can be very helpful for reproducibility. The LangChain RAPTOR example installation command shows the necessary libraries: langchain, umap-learn, scikit-learn, langchain\_community (for some loaders), etc. ([RAPTOR.ipynb](file://file-vddtihmx9rkxzjhcdfqriy%23:~:text=\"pip%20install%20,anthropic\"%20]/)). In a production setting, you would start the query service (API or interface) as a persistent process (systemd service or Docker container) so that it loads the index into memory once and serves queries continuously.

**Updating the Knowledge Base:** For a dynamic KB, plan how updates are handled. You might schedule the ingestion + indexing module to run periodically or when documents are added. Since RAPTOR involves summarizing, adding a new document might ideally trigger rebuilding the cluster summaries (at least for the affected cluster or category). In a simple approach, you could just re-run the entire pipeline overnight. If real-time updates are needed, you could insert the new document’s chunks, embed them, then place them into appropriate existing clusters or treat them as new clusters. Re-clustering everything incrementally is a complex problem – a simpler heuristic is to append the new chunks, and maybe generate a summary if they form a standalone cluster (e.g. a new FAQ document might be its own cluster). For most internal use cases, daily or weekly batch updates are acceptable.

In conclusion, by combining LangChain’s flexible document handling with RAPTOR’s hierarchical retrieval strategy, you can build a powerful local knowledge base that returns relevant information efficiently from even the most complex documents. The keys are to **ingest data systematically**, use strong **embeddings** (with multilingual support for global usage), leverage **clustering and summarization** to compress information, and implement a retrieval mechanism that taps into both detailed and high-level representations of your content. With a modular design and attention to the specifics of your documents (length, language, format), this approach will yield a scalable system ready to answer questions accurately and quickly – all hosted on your own infrastructure for full control and privacy.【28†L100# Building a Knowledge Base with RAPTOR and LangChain for Complex Documents

Building an internal knowledge base for complex documents (manuals, SOPs, regulatory texts, FAQs, etc.) can be greatly enhanced by LangChain’s RAPTOR approach. **RAPTOR** (Recursive Abstractive Processing for Tree-Organized Retrieval) augments retrieval-augmented generation (RAG) by hierarchically clustering and summarizing documents into a tree structure ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=RAPTOR%C2%A0introduces%20a%20novel%20approach%20to,limitations%20in%20traditional%20language%20models)). This enables more efficient, context-aware information retrieval across large texts, addressing limitations of traditional flat chunk retrieval. In this guide, we outline best practices for implementing a RAPTOR-powered knowledge base on a local Ubuntu server – from document ingestion and embedding to vector indexing, hierarchical retrieval, and deployment.

**Document Ingestion and Chunking**

**Flexible File Loaders:** Start by ingesting documents in various formats (PDF, DOCX, TXT, etc.) using LangChain’s document loaders. LangChain provides loaders for many file types – e.g. PyPDFLoader or UnstructuredPDFLoader for PDFs, UnstructuredWordDocumentLoader for Word files, TextLoader for plaintext, CSV loaders for spreadsheets, etc. You can implement a *dynamic loader* selection based on file extension or use the GenericLoader utility for a generic solution ([Dynamic document loader based on file type · langchain-ai langchain · Discussion #10507 · GitHub](https://github.com/langchain-ai/langchain/discussions/10507#:~:text=Yes%2C%20LangChain%20does%20provide%20an,JSON%20files%2C%20and%20so%20on)) ([Dynamic document loader based on file type · langchain-ai langchain · Discussion #10507 · GitHub](https://github.com/langchain-ai/langchain/discussions/10507#:~:text=In%20addition%20to%20these%2C%20LangChain,GenericLoader)). For example, GenericLoader.from\_filesystem() can iterate through a folder and apply the appropriate parser to each file type automatically. This modular design lets you easily add new file type support in the future.

**Text Extraction for Complex Content:** Many enterprise documents contain images, tables, and complex layouts. A best practice is to use robust parsing tools (like **Unstructured.io**, which LangChain integrates) to extract text from such documents. The LangChain RAPTOR demo explicitly leverages Unstructured for parsing PDFs with tables/figures ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=Python.%20,and%20packaging%20tool%20for%20Python)). This ensures that even complex formatting (e.g. multi-column manuals or forms) is converted into text for the knowledge base. If tables or images contain critical info, consider specialized extraction (e.g. OCR for images, CSV parsing for tables) or include the parsed output in document metadata so it isn’t lost.

**Chunking Documents:** Once raw text is extracted, split documents into manageable chunks for embedding. Long documents should be broken into smaller *semantic* units (e.g. sections or paragraphs) because language models and embeddings perform better on moderately sized texts. LangChain’s RecursiveCharacterTextSplitter (or MarkdownHeaderTextSplitter if the document has clear headings) can chunk text while preserving structure. A typical chunk size is on the order of a few hundred words (e.g. ~1,000 tokens), adjustable based on content complexity. In the RAPTOR example, chunks of up to ~2000 tokens were used ([RAPTOR.ipynb](file://file-vddtihmx9rkxzjhcdfqriy%23:~:text=\",/)) – large enough to encompass a full section or topic. You generally want chunks that are self-contained in meaning (to avoid splitting mid-topic) but not so large that they exceed your model’s context window or make clustering difficult.

* **Avoiding Information Loss:** It’s wise to avoid cutting off important sentences or losing context at chunk boundaries. Using the recursive splitter will try larger splits (by paragraph, sentence, etc.) before falling back to character length, which helps maintain coherence. You can also allow a small overlap between chunks (e.g. 50 tokens) to ensure continuity for borderline cases. Each chunk will be stored as a LangChain Document object with relevant metadata (source file name, page number, section title, etc.), which is useful for tracing answers back to origin or filtering by document type.

**Metadata Enrichment:** As you ingest documents, capture key metadata. For instance, tag each chunk with the document title or category (manual vs. policy vs. FAQ), author, publication date, language, etc. This metadata can later be used to filter search results (e.g. only retrieve from certain manuals) or to provide provenance in answers. Metadata is stored in the Document.metadata field in LangChain. A well-designed ingestion pipeline will allow easy extension of metadata fields as needed.

**Embeddings and Multilingual Retrieval**

**Choosing an Embedding Model:** For a local deployment with multilingual documents, **Hugging Face’s intfloat/multilingual-e5-large** is an excellent choice for the embedding model. The E5 models are state-of-the-art, trained for embedding **queries** and **passages** such that relevant pairs have high cosine similarity. Notably, multilingual-e5-large supports **100+ languages** (being initialized from XLM-RoBERTa) ([intfloat/multilingual-e5-large · Hugging Face](https://huggingface.co/intfloat/multilingual-e5-large" \l ":~:text=Supported%20Languages)), making it suitable if your knowledge base content or user queries span multiple languages. This model produces 1024-dimensional embeddings (since it’s based on a RoBERTa-large architecture), providing rich semantic representation.

* **Embedding Usage Tips:** The E5 model expects special prefixes on input text to differentiate query vs. document context. For optimal results, prefix each document chunk with **"passage: "** and each user query with **"query: "** before embedding ([intfloat/multilingual-e5-large · Hugging Face](https://huggingface.co/intfloat/multilingual-e5-large" \l ":~:text=,see%20how%20much%20protein%20you)). This matches the model’s training format and significantly improves retrieval performance by aligning embeddings in the shared vector space. In practice, you can wrap the model with a small helper that adds the prefix and then uses HuggingFace Transformers to compute the embeddings in batches. Batch processing will speed up embedding generation when ingesting many documents.

**Alternatives:** If for some reason E5-large is not ideal (e.g. resource constraints), other options include smaller sentence-transformer models (like all-MiniLM variants) or multilingual Instructor models. Keep in mind that smaller models (with 384-dimensional embeddings, for example) might be faster but could sacrifice some accuracy ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=OpenAI%20embeddings%20and%20384%20in,source%20small%20embedding%20models)). OpenAI’s text-embedding-ada-002 (1536-dim) is another high-quality choice, but it requires API access and primarily supports English. For an entirely offline system with strong multilingual support, E5-large strikes a good balance. If you need even more efficiency, you could explore quantizing the model or using a GPU to serve embeddings, but those are optimizations rather than fundamental design choices.

**Cross-Lingual Considerations:** With a multilingual embedding model, the system can handle cross-lingual retrieval: for example, an English query can successfully retrieve a relevant chunk from a Spanish document if their embeddings are similar. This is very useful for international or multilingual documentation. However, ensure that the language of each chunk is either captured in metadata or that the embeddings are truly language-agnostic. The E5 model was trained on many languages, but extremely domain-specific jargon in one language might still pose challenges. In practice, test a few queries in different languages to verify that retrieval returns expected results across languages.

**Choosing a Vector Store (FAISS vs. Alternatives)**

To enable fast similarity search over the embeddings, you’ll need a vector store or index. The preference here is **FAISS** (Facebook AI Similarity Search), which is a proven library for efficient nearest-neighbor search on dense vectors ([Faiss | ️ LangChain](https://python.langchain.com/docs/integrations/vectorstores/faiss/" \l ":~:text=,See%20The%20FAISS%20Library%20paper)). FAISS is well-suited for local deployment and can handle large collections (millions of vectors) through advanced indexing methods. In LangChain, you can use FAISS.from\_documents(doc\_list, embedding) or FAISS.from\_texts(text\_list, embedding) to create an index, and then call .as\_retriever() on it to integrate with your QA chain.

**FAISS Benefits:** FAISS is highly optimized in C++ with Python bindings. It supports various indexing strategies (exact brute-force search, IVFFlat/IVFPQ for huge data, HNSW, etc.), and importantly it can be saved to disk and reloaded for persistence ([How to properly initialize ChromaDB instead of FAISS in order to use it in the AutoGPT example? · langchain-ai langchain · Discussion #5682 · GitHub](https://github.com/langchain-ai/langchain/discussions/5682#:~:text=To%20save%20a%20FAISS%20to,the%20folder%20path%2F%20index%20name)). Benchmarks have shown that FAISS offers excellent **accuracy and speed**. In one comparison of vector stores, using the same knowledge base and queries, FAISS slightly outperformed Chroma in both context precision and recall. It was also faster – taking only ~1.81 seconds to retrieve answers for 50 questions, whereas Chroma took ~2.18 seconds ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=As%20indicated%20in%20Table%201%2C,18%20seconds)). The differences were marginal in many cases, but FAISS returned the correct context more often (in 4 out of 5 discrepant queries) and consistently had lower query latency ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=differences%20in%20numbers%20are%20marginal,18%20seconds)). Overall, FAISS’s efficient search algorithm made it the preferred choice over Chroma in that evaluation ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Changing%20the%20vector%20store%20influences,the%20preferred%20choice%20over%20Chroma)).

**Alternative Vector Stores:** It’s worth considering alternatives if they offer specific advantages your use case needs:

* **ChromaDB:** An open-source vector database that wraps an ANN index (HNSW by default) with a user-friendly API and persistent storage (using SQLite/DuckDB). Chroma is easy to use and allows filtering by metadata out-of-the-box. However, by default it uses approximate search, which can introduce slight nondeterminism in results ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Why%20is%20there%20discrepancy%20between,vector%20stores)). For moderate-scale data, FAISS (with exact search or properly tuned ANN) might be more accurate as noted. Chroma is a fine choice if you want a plug-and-play persistent DB and are okay with its performance trade-offs.
* **Qdrant:** A standalone high-performance vector DB (written in Rust) that can run on-premise. It supports filtering, payload storage, and horizontal scaling. If you anticipate a need to **update** the index frequently (add/delete documents) or complex filtering on metadata, a system like Qdrant or Weaviate might simplify those operations compared to managing raw FAISS indices. They do require running a separate service, though.
* **Other options:** There are many others (Milvus, Weaviate, Elasticsearch’s vector fields, etc.). They generally offer similar ANN capabilities and trade memory for speed. For an initial local deployment, these may be overkill unless you have very large scale or specific integration requirements.

In practice, for an internal KB on Ubuntu, many teams start with **FAISS in-memory** for its speed and simplicity, and possibly move to an external vector DB only if scaling or multi-user access demands it. You can also combine approaches: for example, use FAISS for pure semantic search and then apply metadata filters in application logic, or periodically sync the FAISS index to a disk or cloud storage for persistence. LangChain’s abstraction makes it relatively easy to swap out the vector store backend if requirements change.

**Hierarchical Clustering and Summarization (RAPTOR Method)**

Once documents are chunked and embedded, **RAPTOR** introduces a hierarchical compression step before final indexing. The core idea is to cluster similar chunks and generate *summaries* for each cluster, then repeat this process to form a tree of summaries. This creates multiple levels of abstraction: leaf nodes are original text chunks, higher-level nodes are summaries that synthesize information from several related chunks ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=In%20short%2C%20the%20intuition%20behind,RAPTOR%20as%20follows)). By doing this, RAPTOR can capture broader context that spans multiple sections or documents, enabling the system to answer high-level questions more effectively (since a summary node might combine info that was scattered in different parts of a manual).

([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/)) **Figure:** A conceptual diagram of the RAPTOR tree architecture. Each colored block (1–5) represents an embedded text chunk from the documents (leaf layer). Similar chunks are **clustered** together (grouped by a clustering algorithm), and an LLM generates an **abstractive summary** of each cluster – shown as the yellow nodes (6, 7, 8 in the middle layer). These summaries become new nodes in the tree with their own embeddings. The process can repeat: the summary nodes themselves are clustered and summarized into higher-level nodes (right, e.g. root nodes 9 and 10), until the tree is built up to a root or until summaries are sufficiently compact. Each node thus contains a condensed representation of the information from its child nodes, allowing multi-scale understanding of the knowledge base.

**Clustering Strategy:** In practice, use a clustering algorithm to group embedding vectors of similar content. The LangChain RAPTOR example applied **UMAP** (for dimensionality reduction) followed by **Gaussian Mixture Models (GMM)** for clustering ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20clustering%20approach%20in%20tree,includes%20a%20few%20interesting%20ideas)) ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=GMM%20)). The rationale is that raw embedding vectors might be 384–1536 dimensions, which is computationally heavy for clustering directly; reducing them to a lower dimension (e.g. 5–50 dims) with UMAP can improve clustering efficiency without losing much semantic structure ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=Suppose%20we%20have%208%20document,source%20small%20embedding%20models)). GMM (with BIC to choose the number of clusters) or K-Means are common choices for clustering. The key is to cluster chunks such that those in each group are topically related.

* *Example:* If you have a 200-page technical manual, the initial chunks might naturally cluster into groups corresponding to each chapter or section (e.g. all chunks about installation procedures cluster together, all chunks from the safety guidelines section cluster together, etc.). Each cluster will then be summarized.

**Summarization of Clusters:** For each cluster of chunks, use a **large language model** to produce an *abstractive summary* that captures the key points of that cluster. This is effectively a compression step – condensing, say, 10 chunks (which might be ~5,000 words combined) into a concise summary of a few hundred words that preserves the important information. The summary text will serve as a new document representing that cluster. You should embed the summary as well (using the same embedding model) so that it can be included in the vector index as a node.

* **Prompting the LLM:** When summarizing, prompt engineering is important. Since this is an internal knowledge base, you likely want the summaries to be factual and *information-dense* (rather than overly general). A good prompt might be: *“Summarize the following content in a concise, informative way, preserving all key details and terminology.”* Provide the LLM with the full text of the cluster (if it fits in context) or iterative summarizations if needed. Ensure the summarizer knows to **preserve critical data** (e.g. specific compliance requirements or numeric values) to avoid losing important specifics. Depending on the size of clusters, you might use an 8k or 16k context model for summarization (GPT-4, Claude 2, etc.), or break the cluster and summarize in sections, then merge those summaries. The goal is that each cluster summary is a self-contained, coherent piece of text representing that topic area.

**Recursive Tree Construction:** After the first round of clustering and summarizing (producing first-level summary nodes), you can iterate the process: treat the new summaries as a smaller set of “documents” and again cluster them, then summarize to form second-level nodes, and so on ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=and%20then%20run%20a%20dimensionality,source%20small%20embedding%20models)). This forms a **tree** where the leaves are original chunks and each higher layer node is the summary of a group of nodes below. Continue until you either reach a single root summary for the entire corpus or the summaries at the current layer are already short and distinct enough. In practice, you might end up with a handful of top-level summaries (if the knowledge base has distinct themes). Each node (at all levels) should be stored as a Document with its text and metadata indicating its parent/children or source cluster.

Some key points and benefits of the RAPTOR approach:

* **Multi-Scale Context:** RAPTOR’s recursive summarization creates multiple levels of context. High-level summaries capture the gist of broad topics across many documents, while low-level chunks retain the detailed information. This ensures that whether a question is broad or narrow, the retrieval system has an appropriate representation to draw from.
* **Reduced Context at Query Time:** By clustering and summarizing, the system can answer questions that span many parts of a document using a *smaller set of nodes*. Instead of retrieving a dozen full chunks and concatenating them (which might hit context size limits), the system could retrieve one or two summary nodes that already contain the combined information from those dozen chunks. This greatly reduces the prompt size needed for the LLM, making retrieval and answer generation more efficient ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,can%20be%20achieved%20using%20fast)) ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=k,aligning%20with%20model%20context%20constraints)).
* **Improved Retrieval Performance:** By including summaries in the index, the retriever can find relevant information even if it was split across multiple chunks originally. Our controlled experiments showed that RAPTOR’s method of hierarchical clustering + summarization not only improves answer accuracy but also can set new performance benchmarks on challenging multi-document QA tasks ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=is%20capable%20of%20synthesizing%20information,answering%20tasks)).

Keep in mind that constructing this tree is an **offline preprocessing** step. It can be computationally intensive (especially if you have thousands of chunks and need to call an LLM for each cluster summary), but it only needs to be done when the document set updates. The output is an expanded set of documents (original chunks + various summaries) that you will feed into the vector store.

**Efficient Retrieval Strategy with RAPTOR**

With the RAPTOR tree built, how do we use it during query time? There are two main retrieval strategies to consider:

([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/)) **Figure:** Two retrieval strategies using the RAPTOR tree. **(A) Tree Traversal Retrieval:** The query is embedded (Q) and compared layer-by-layer through the tree structure – for example, first finding the most relevant top-level node, then searching within its child nodes, and so on down to the leaves. **(B) Collapsed Tree Retrieval:** The query embedding is compared against *all nodes at all levels simultaneously* (a “flattened” index of every summary and chunk), retrieving the top-K most relevant nodes in one go. The retrieved context (pink boxes) is then provided, along with the original query, to the LLM to generate the answer. In practice, the **collapsed tree approach is preferred** due to its flexibility and strong performance ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,can%20be%20achieved%20using%20fast)). By searching across every node, the retriever can find either a broad summary or a specific detail as needed, without having to navigate the tree stepwise.

**Why Collapsed Retrieval?** As the LangChain blog notes, collapsed retrieval offers greater flexibility and often **better performance** than strict hierarchical traversal ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=The%20collapsed%20tree%20approach%20is,method%20with%202000%20maximum%20tokens)). It allows dynamic retrieval of information at varying levels of granularity tailored to the question. In other words, RAPTOR can adaptively select nodes from different layers of the tree that best match the query, optimizing relevance and completeness of the answer. This does require searching a larger index (all nodes), but thanks to fast cosine similarity search (FAISS or similar), that's not a problem. In their tests, the **collapsed tree method (with ~2000-token summaries)** provided optimal results, as it accommodated varying amounts of information per node while still fitting within model context constraints ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=k,aligning%20with%20model%20context%20constraints)).

In implementation, using collapsed retrieval is straightforward: when building your vector store, simply **include all nodes** (original chunks and all summary nodes from every level) in the index. At query time, embed the user’s question (with the "query: " prefix for E5) and perform a standard similarity search in this index. The top *K* results might include a mix of detailed chunks and high-level summaries. You can then feed the content of those nodes into the LLM prompt to formulate the answer.

**Handling Retrieved Summaries:** If a retrieved node is a summary rather than original text, you have a couple of options for using it in the answer step. Usually, you can treat the summary text as just another piece of context – it often contains the answer or at least pointers to it. If the question is asking for an overview or a list, a summary node might directly answer it. However, if the question requires a fine detail that was abstracted away in a summary, you might need to pull in the original chunks. One strategy is: when a high-level summary node is retrieved with high relevance, also fetch its child chunks (you likely have a mapping from summary nodes to their source chunks). Then you can give the LLM both the summary *and* a couple of the most relevant underlying chunks to ensure it has the specifics. This two-step fallback can be implemented by storing cluster membership in metadata (e.g. each chunk knows its cluster summary ID, and each summary knows IDs of children). In practice, we often find the combination of summary + a specific chunk in the top-K results naturally covers this, but it’s something to keep in mind for critical precision.

**Tuning k:** Deciding how many nodes to retrieve is a balance between completeness and conciseness. Because RAPTOR nodes pack more information (each summary represents multiple chunks), you may not need as many nodes as a flat retriever would. A common approach is to retrieve e.g. *k=5* nodes. If your summaries are well-made, even 3 nodes could sometimes suffice for broad questions ([Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan | Medium](https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Retrieve%20Multiple%20Context%20Documents)). For very detail-specific queries, those 3 might include at least one original chunk with the answer. It’s wise to experiment: try different k values and evaluate the quality of answers. The RAPTOR paper and demo results can guide you here – they achieved strong results with a handful of retrieved nodes. Also consider using **Maximal Marginal Relevance (MMR)** in the retriever (LangChain supports this) to ensure diversity in the retrieved set – this can help bring in one broad summary and one detailed chunk rather than five very similar chunks.

In summary, the retrieval phase in a RAPTOR-enabled system uses the vector store to surface the most relevant pieces of knowledge, whether they are raw text or summaries. This approach gives the downstream LLM a compact yet information-rich context to work with, which improves efficiency and often answer accuracy. By leveraging both specific and abstracted knowledge, the system can handle a range of question types effectively, as demonstrated by RAPTOR’s superior performance on multi-document QA tasks ([Efficient Information Retrieval RAG for Complex PDFs Using RAPTOR](https://www.langchain.ca/blog/efficient-information-retrieval-from-complex-pdfs-using-raptor-rag/#:~:text=is%20capable%20of%20synthesizing%20information,answering%20tasks)).

**System Architecture and Deployment Considerations**

Design the system in a **modular pipeline** fashion for scalability and maintainability. A recommended architecture on a local Ubuntu server is as follows:

1. **Ingestion Module:** A script or service responsible for document loading, parsing, and chunking. This module can scan a directory (or other sources), use LangChain loaders to read files, split them into Document chunks with metadata, and output these (to memory, disk, or a database). By separating ingestion, you can easily re-run it when new documents are added or existing ones change. In an Ubuntu environment, you might schedule this as a cron job or trigger it via an API when needed. Ensure you log or store the mapping from source documents to generated chunks (for traceability).
2. **Processing & Indexing Module:** This component takes the output of ingestion and builds the knowledge index. It will load the embedding model (e.g. the E5-large via HuggingFace Transformers) – ideally once at startup to avoid reloading weights – and embed all document chunks. If using the RAPTOR approach, this module will also perform the clustering and summarization steps described above:
   * Compute initial embeddings for all chunks.
   * Perform dimensionality reduction (if needed) and clustering to group similar chunks.
   * Invoke the LLM to generate summaries for each cluster (writing these summaries out as new Document objects).
   * Iterate this clustering/summarizing if multi-level hierarchy is needed.
   * Embed all summary nodes as well.
   * Combine the embeddings of original chunks and all summary nodes into the final vector index (FAISS).

This pipeline can be resource-intensive, so you might run it offline. For example, you could run it overnight and have the updated index ready for the next day. Make sure to **persist the vector store** to disk (FAISS index) once built, so that it can be quickly loaded by the query service. LangChain’s FAISS integration allows saving and loading indexes easily ([How to properly initialize ChromaDB instead of FAISS in order to use it in the AutoGPT example? · langchain-ai langchain · Discussion #5682 · GitHub](https://github.com/langchain-ai/langchain/discussions/5682#:~:text=To%20save%20a%20FAISS%20to,the%20folder%20path%2F%20index%20name)). Additionally, you might store the hierarchy mapping (which chunk belongs to which summary cluster) in a simple JSON or database if you plan to use it during retrieval for pulling related chunks.

1. **Query Service:** This is the live service (e.g. a FastAPI app or a Streamlit web UI) that fields user queries using the prepared index. On startup, it will load the persisted FAISS index (and the embedding model, if not kept in memory from before). When a query comes in, the service:
   * Preprocesses the query (e.g. detects language if needed, then applies the "query: " prefix) and embeds it using the same model as was used for the documents.
   * Performs a similarity search in the FAISS index to get top-K relevant nodes (chunks or summaries).
   * Formats a prompt for an LLM that includes the user’s question and the retrieved context. This can be done with a LangChain RetrievalQA chain or manually. Typically, you might use a template like: *“You are an expert assistant. Using the information provided, answer the question. Context:\n{retrieved text}\n\nQuestion: {user question}\nAnswer:”*. The retrieved text can include the content and possibly source tags for each snippet.
   * Sends this prompt to the chosen LLM (could be an OpenAI API like GPT-4, or a local model running on the server).
   * Returns the answer to the user, optionally with references. If you included source metadata (e.g. document titles) in the context, the LLM might naturally cite them. Alternatively, you can post-process the answer and append source attributions based on which documents were retrieved.

These components can be deployed on a single Ubuntu server process, or split into separate processes/services depending on load. For example, the query service could be a continuously running API server, while ingestion/indexing might be run on demand (since it’s not needed for every query).

**Local LLM vs API:** A major decision is whether to use local LLMs for summarization and Q&A or rely on cloud APIs. Running a model like GPT-4-sized locally is not feasible for most, but there are smaller instruction-tuned models that can run on GPU or even CPU (like Llama 2 13B, etc.). They might be sufficient for summarization tasks and even for answering simpler questions, but for high-quality answers on complex text, GPT-4 (via API) or similar is still superior. Since the knowledge base is internal, privacy might be a concern – one compromise is to use local models for summarization (since that involves sending large chunks of internal text to the model), and use an API model for the final Q&A (since the retrieved context is much smaller and already abstracted). If full privacy is required, you’ll need to stick to local models end-to-end, accepting some quality trade-offs. In any case, design your system so that the LLM used for each stage can be configured or swapped out easily (e.g. through an interface in LangChain or a wrapper class).

**Performance Considerations:** On Ubuntu, ensure you have the necessary system libraries installed (for example, faiss-cpu via pip, and any Unstructured dependencies like libmagic for file type detection). Leverage hardware where possible – a GPU can massively speed up embedding computations and LLM inference. If using a GPU for embeddings, the intfloat/multilingual-e5-large model can be loaded with AutoModel.from\_pretrained(..., device\_map="auto") to utilize the GPU. If using CPU, consider using 8-bit quantization for the model (bitsandbytes integration) to make embedding faster. Also, FAISS has a GPU version, but for most cases of a few hundred thousand vectors or less, FAISS on CPU is extremely fast already (query searches in tens of milliseconds).

For the retrieval itself, FAISS can handle very large indexes, but if your corpus grows into the millions of chunks, you may want to use FAISS’s indexing strategies (like IVF) to keep search fast. This can be configured in FAISS and integrated with LangChain by providing a custom FAISS Index. In moderate cases, a flat index is fine. Monitor memory usage: storing a million 1024-dim vectors in RAM is on the order of 4 GB of memory (using 32-bit floats). If memory is a concern, FAISS supports 8-bit or 16-bit compression of vectors or even on-disk indices. Those are advanced optimizations to consider if needed.

**Maintenance and Scalability:** As your knowledge base content evolves, you’ll need a process to update the index. If documents are added incrementally, you could embed those new chunks and *add them* to the existing FAISS index (FAISS supports adding vectors). However, integrating them into the RAPTOR tree is more involved – you would likely need to find which cluster they belong to or create new clusters. In practice, a simpler approach is to periodically rebuild the index from scratch (especially if the corpus is not enormous or changes are batchable). For instance, if you have weekly documentation updates, you might rerun the ingestion and indexing pipeline weekly. This ensures the clustering and summaries remain consistent. If using an external vector DB like Qdrant, you could upsert new documents on the fly, but you’d then lack the updated hierarchical summaries unless you also trigger a re-summarization. This is an area where more automation could be built, but initially, designing for periodic recomputation is straightforward and robust.

Finally, ensure to test the end-to-end system with real-world queries from your users. See if the answers are accurate and if the sources make sense. Pay attention to cases where the LLM might get something wrong – is it due to a missing piece of context (maybe chunk size too large/small, or summarization lost a detail) and refine accordingly. The RAPTOR approach is powerful, but it may require tuning cluster sizes or summary lengths for your specific domain (e.g. regulatory content might require more precise summaries). By following these best practices and iterating, you will have a scalable, efficient knowledge base that leverages LangChain and RAPTOR to deliver quick, context-rich answers from your internal documents, all on your own Ubuntu server.