Project HL architectural design

Below is a high-level application architecture outlining the major modules, their responsibilities, interactions, dependencies, and a recommended project structure. This design is tailored for a local Ubuntu deployment of a RAPTOR-powered knowledge base built using LangChain, FAISS, and Hugging Face’s multilingual-e5-large embedding model.

**Overview Diagram**

┌────────────────────┐

│ User / Client │

└────────────────────┘

│

▼

┌──────────────────────┐

│ Query Service │

│ (API / Web Frontend) │

└──────────────────────┘

│

▼

┌──────────────────────┐

│ Retrieval Module │

│ (Vector Store Search)│

└──────────────────────┘

│

▼

┌──────────────────────┐

│ Vector Store (FAISS)│

└──────────────────────┘

▲

│

┌──────────────┬─────────────┼─────────────┬──────────────┐

│ │ │ │

▼ ▼ ▼ ▼

┌─────────┐ ┌───────────────┐ ┌─────────────┐ ┌───────────────────┐

│ Document│ │ Embedding │ │ RAPTOR Tree │ │ Metadata / Config│

│ Ingestion│ │ Generator │ │ Generator │ │ Module │

│ Module │ │ Module │ │ Module │ └───────────────────┘

└─────────┘ └───────────────┘ └─────────────┘

│ │ │

└─────► (LangChain Documents) ◄────────────┘

**Module Descriptions & Responsibilities**

1. **Document Ingestion Module**  
   **Scope:**
   * Discover and load various file types (PDFs, DOCX, TXT, etc.) from specified sources (directories, network locations, etc.).
   * Use LangChain’s built-in file loaders and/or custom loaders to parse and extract text from documents.
   * Perform pre-processing such as cleaning and formatting.
   * Convert the raw text into LangChain Document objects, attaching metadata (source file name, page number, document type, language, etc.).

**Key Functions:**

* + load\_documents(path: str) -> List[Document]
  + parse\_file(file\_path: str) -> Document
  + preprocess\_document(document: Document) -> Document

**Dependencies:**

* + LangChain document loaders, libraries for file handling (e.g. PyPDF2, Unstructured), and any specialized extraction libraries.

1. **Document Chunking Module**  
   **Scope:**
   * Split long documents into manageable, semantically coherent chunks to optimize for embedding generation and later retrieval.
   * Implement strategies like overlapping contexts to preserve information at boundaries.

**Key Functions:**

* + chunk\_document(document: Document, chunk\_size: int, overlap: int) -> List[Document]

**Dependencies:**

* + LangChain’s text splitters such as RecursiveCharacterTextSplitter or similar.

1. **Embedding Generator Module**  
   **Scope:**
   * Load the Hugging Face intfloat/multilingual-e5-large (or a chosen alternative) for generating embeddings from text.
   * Apply the appropriate prefixes ("passage:" for document chunks, "query:" for queries) to match the model’s training context.
   * Batch process texts for efficiency.

**Key Functions:**

* + get\_embedding(text: str) -> np.array
  + embed\_documents(documents: List[Document]) -> List[Embedding]

**Dependencies:**

* + Hugging Face Transformers, PyTorch/TensorFlow, and LangChain integration wrappers for embeddings.

1. **RAPTOR Tree (Clustering and Summarization) Module**  
   **Scope:**
   * **Clustering:** Use algorithms such as UMAP for dimensionality reduction followed by clustering (e.g., GMM or K-Means) to group similar document chunks.
   * **Summarization:** For each cluster, leverage a Large Language Model (LLM) to produce an abstractive summary that captures the essence of all chunks in that cluster.
   * **Recursion:** Optionally, repeat clustering and summarization on summary nodes to build a multi-level (hierarchical) representation (i.e., a tree) of the knowledge base.

**Key Functions:**

* + cluster\_documents(embeddings: List[Embedding]) -> Dict[cluster\_id, List[Document]]
  + summarize\_cluster(documents: List[Document]) -> Document
  + build\_raptor\_tree(documents: List[Document]) -> List[Document] (returns enriched list of both original chunks and summary nodes)

**Dependencies:**

* + Libraries such as UMAP, scikit-learn for clustering, and LangChain’s LLM wrappers for summarization.

1. **Vector Store Module (FAISS Integration)**  
   **Scope:**
   * Store the embeddings of document chunks and summary nodes in a FAISS index.
   * Provide efficient similarity search and retrieval functionality.
   * Offer functionality to persist (save/load) the index so the embedding computation need not be repeated at each service start-up.

**Key Functions:**

* + build\_faiss\_index(documents: List[Document]) -> FAISSIndex
  + save\_index(index: FAISSIndex, filepath: str)
  + load\_index(filepath: str) -> FAISSIndex
  + query\_index(query\_embedding: np.array, k: int) -> List[Document]

**Dependencies:**

* + FAISS library (either CPU or GPU version), numpy, LangChain’s vector store abstraction.

1. **Query Service Module**  
   **Scope:**
   * Expose an interface (e.g., using FastAPI or Flask) to accept user questions.
   * Preprocess and embed incoming queries using the same embedding model (with "query:" prefix).
   * Retrieve relevant nodes from the FAISS index using the vector store module with collapsed tree retrieval (i.e., a unified index of chunks and summaries).
   * Construct and send a complete prompt (question + retrieved context) to an LLM to generate a final answer.
   * Return the answer (with optional source metadata) back to the user.

**Key Functions / Components:**

* + REST endpoints for query submission (e.g. /ask).
  + Function handle\_query(query: str) -> str that ties together embedding, retrieval, prompt assembly, and LLM call.

**Dependencies:**

* + FastAPI/Flask (for web API), LangChain QA chains, embedding and retrieval modules, and optionally an LLM client (OpenAI API, local LLM, etc.).

1. **Configuration and Metadata Module**  
   **Scope:**
   * Centralized configuration for the entire application (e.g. model paths, chunk sizes, retrieval parameters, API keys, etc.).
   * Optionally, manage metadata schema for documents and summaries so provenance is maintained.

**Key Files:**

* + config.py or similar configuration file.

**Dependencies:**

* + Standard Python configuration libraries (e.g. configparser, pydantic).

1. **Main Application Driver**  
   **Scope:**
   * Orchestrate the overall pipeline or provide command-line interfaces (CLIs) for various tasks (ingestion, indexing, updating the index, and starting the query server).

**Key Functions:**

* + A “build index” routine: invoke ingestion, chunking, embedding, RAPTOR tree building, and vector store construction/saving.
  + A “run server” routine: load vector store and start the query API service.

**Recommended Project Structure**

A suggested directory layout might look like this:

knowledgebase-raptor/

├── ingestion/

│ ├── \_\_init\_\_.py

│ ├── loader.py # File loading functions for PDFs, DOCX, TXT, etc.

│ ├── chunker.py # Document chunking utilities

│ └── utils.py # Additional helper functions

├── embeddings/

│ ├── \_\_init\_\_.py

│ └── embedder.py # Wrappers for HuggingFace embedding model calls

├── raptor/

│ ├── \_\_init\_\_.py

│ ├── clustering.py # Clustering functions using UMAP and GMM/K-Means

│ ├── summarizer.py # Functions to call an LLM for summarization

│ └── tree\_builder.py # Orchestrator for building the RAPTOR tree

├── vector\_store/

│ ├── \_\_init\_\_.py

│ └── faiss\_store.py # Functions to build, save, load, and query the FAISS index

├── query\_service/

│ ├── \_\_init\_\_.py

│ └── api.py # FastAPI (or Flask) server exposing query endpoints

├── config.py # Central configuration (e.g., chunk sizes, model names)

├── main.py # Main driver for building index or starting service

├── requirements.txt # Python dependencies (langchain, transformers, faiss-cpu, umap-learn, scikit-learn, fastapi, etc.)

└── README.md # Project overview and instructions

**Module Interactions**

1. **Pipeline Build (Indexing Phase):**
   * **Ingestion Module** loads files from disk and returns a list of Document objects.
   * **Chunking Module** processes each document into smaller logical chunks.
   * **Embedding Module** then computes embeddings for all chunks.
   * **RAPTOR Tree Module** takes the embedded chunks, clusters them, and uses summarization to create higher-level nodes – outputting a unified set of documents (original chunks plus summary nodes).
   * **Vector Store Module** builds a FAISS index from the entire set and persists it to disk.
2. **Query Phase (Online):**
   * The **Query Service Module** (e.g. a FastAPI app) loads the persisted FAISS index on startup.
   * When a query is received, it uses the **Embedding Module** to transform the query text.
   * The **Vector Store Module** is then called to perform a similarity search (collapsed retrieval) across the full index.
   * Retrieved context (both detailed and abstracted nodes) is formatted into a prompt and sent to an LLM via a **LangChain QA chain** component (potentially also using the **RAPTOR Tree Module** if drill-down is needed).
   * The answer is returned to the client.
3. **Configuration & Metadata:**
   * All modules read from a centralized configuration (config.py) for consistency in parameters like chunk sizes, clustering options, embedding model names, API keys, etc.
   * Metadata attached to documents during ingestion is carried through the pipeline. It’s used to trace sources back during retrieval and for filtering if needed.

**Dependencies**

* **LangChain:** Core for document abstraction, chains, and integration with loaders, LLM, and vector store.
* **Transformers (Hugging Face):** For loading and running the multilingual-e5-large embedding model.
* **FAISS:** For efficient vector similarity search.
* **UMAP, scikit-learn:** For dimensionality reduction and clustering in the RAPTOR tree construction.
* **FastAPI/Flask:** For building the query service API.
* **LLM Provider (Optional):** E.g., OpenAI API or a local LLM for summarization and question answering.
* **Other Utilities:** Standard Python libraries such as numpy, pandas (if needed for data manipulation), and any specialized libraries required by Unstructured, file loaders, etc.

This high-level architectural design emphasizes modularity, allowing you to independently develop, test, and scale each component. Whether you update the document ingestion logic or swap out the embedding model, the clear separation of concerns and well-defined interfaces between modules will make maintenance and future enhancement straightforward.