RAG Architecture Implementation

Context and Problem Statement

The current trend in questions and answering (Q&A) using large language models (LLMs) or other AI related technology is retrieval-augmented-generation (RAG).

RAG is related to Open Generative QA that means LLM (which generates text) is supplied with context (chunks of information extracted from various sources) and then it generates answer.

RAG architecture consists of these steps (simplified):

How source data is processed:

1. **Indexing**: application is supplied with information sources (PDFs, text files, web pages, etc.)
2. **Conversion**: files are converted to string (because LLM works on text data).
3. **Splitting**: the string from previous step is split into parts (because LLM has fixed context window, meaning it cannot handle big documents).
4. **Embedding generation**: a vector consisting of float values is generated out of chunks. This vector represents meaning of text and the main propety of such vectors is that chunks with similar meaning has vectors that are close to. Generation of such a vector is achieved by using a separate model called *embedding model*.
5. **Store**: chunks with relevant metadata (for example, from which document they were generated) and embedding vector are stored in a vector database.

How answer is generated:

1. **Ask**: user asks AI a question.
2. **Question embedding**: an embedding model generates embedding vector of a query.
3. **Data finding**: vector database performs search of most relevant pieces of information (a finite count of pieces). That’s performed by vector similarity: meaning how close are chunk vector with question vector.
4. **Prompt generation**: using a prompt template the user question is *augmented* with found information. Found information is not generally supplied to user, as it may seem strange that a user asked a question that was already supplied with found information. These pieces of text can be either totally ignored or showed separately in UI tab “Sources”.
5. **LLM generation**: LLM generates output.

This ADR concerns about implementation of this architecture.

Decision Drivers

- Prefer good and maintained libraries over self-made solutions for better quality.
- The usage of framework should be easy. It would seem strange when user wants to download a BIB editor, but they are required to install some separate software (or even Python runtime).
- RAG shouldn't provide any additional money costs. Users should pay only for LLM generation.

Considered Options

- Use a hand-crafted RAG
- Use a third-party Java library
- Use a standalone application
- Use an online service

Decision Outcome

Chosen option: mix of "Use a hand-crafted RAG" and "Use a third-party Java library".

Third-party libraries provide excellent resources for connecting to an LLM or extracting text from PDF files. For RAG, we mostly used all the machinery provided by `langchain4j`, but there were moments that should be hand-crafted:

- **LLM connection**: due to https://github.com/langchain4j/langchain4j/issues/1454 (https://github.com/InAnYan/jabref/issues/77) this was delegated to another library `jvm-openai`.
- **Embedding generation**: due to https://github.com/langchain4j/langchain4j/issues/1492 (https://github.com/InAnYan/jabref/issues/79), this was delegated to another library `djl`.
- **Indexing**: `langchain4j` is just a bunch of useful tools, but we still have to orchestrate when indexing should happen and what files should be processed.
- **Vector database**: there seems to be no embedded vector database (except SQLite with `sqlite-vss` extension). We implemented vector database using `MVStore` because that was easy.

Pros and Cons of the Options

Use a hand-crafted RAG

- Good, because we have the full control over generation
- Good, because extendable
- Bad, because LLM connection, embedding models, vector storage, and file conversion should be implemented manually

- Bad, because it’s hard to make a complex RAG architecture

Use a third-party Java library

- Good, because provides well-tested and maintained tools
- Good, because libraries have many LLM integrations, as well as embedding models, vector storage, and file conversion tools
- Good, because they provide complex RAG pipelines and extensions
- Neutral, because they provide many tools and functions, but they should be orchestrated in a real application
- Bad, because some of them are raw and undocumented
- Bad, because they are all similar to `langchain`
- Bad, because they may have bugs

Use a standalone application

- Good, because they provide complex RAG pipelines and extensions
- Good, because no additional code is required (except connecting to API)
- Neutral, because they provide not that many LLM integrations, embedding models, and vector storages
- Bad, because a standalone app running is required. Users may be required to set it up properly
- Bad, because the internal working of app is hidden. Additional agreement to Privacy or Terms of Service is needed
- Bad, because hard to extend

Use an online service

- Good, because all data is processed and stored not on the user’s machine: faster and no memory is used.
- Good, because they provide complex RAG pipelines and extensions
- Good, because no additional code is required (except connecting to API)
- Neutral, because they provide not that many LLM integrations, embedding models, and vector storages
- Bad, because requires connection to Internet
- Bad, because data is processed by a third party company
- Bad, because most of them require additional payment (in fact, it would be impossible to develop a free service like that)
