ImmverseAI AI/ML Intern Assignment
Sanskrit Document Retrieval-Augmented Generation (RAG) System

Name : Toshik Choudhary

Date: 11 / 2 / 2026

1. Introduction

As mentioned in the ImmverseAI AI/ML intern assignment, the goal of this project is to design and implement a CPU-only Retrieval-Augmented Generation (RAG) system for Sanskrit documents. After ingesting a small corpus of Sanskrit stories, the system uses semantic embeddings to index them, retrieves the most pertinent passages for a user query, and then uses a lightweight multilingual sequence-to-sequence model that runs solely on CPU to generate an answer. In order to allow for the replacement or improvement of individual components in subsequent work, the architecture is purposefully modular, with distinct components for document loading and preprocessing, retrieval, and generation. Rather than attaining state-of-the-art natural language fluency, the main focus is on creating a transparent, repeatable pipeline and showcasing end-to-end functionality on Sanskrit text.

2. Dataset: Sanskrit Documents

The dataset consists of Sanskrit prose passages provided in the assignment document, including stories such as **“मूर्खभृत्यस्य”**, **“चतुरस्य कालीदासस्य”**, **“वृद्धायाः चार्तुयम्”**, and a narrative about a devotee who expects divine help without personal effort. These stories are written primarily in classical Sanskrit, with occasional English glosses or translations in brackets to aid understanding.

The original DOCX file was converted into UTF-8 encoded .txt files, which are stored in the project’s data/ directory and serve as the source corpus for the RAG pipeline. Together, these texts span a few thousand words and cover everyday narrative scenes, moral lessons, and dialogues. This size is small enough to allow efficient CPU-only indexing and retrieval, while still being rich enough to test question-answering over multiple documents.

3. System Architecture

The system follows the standard RAG architecture, with a clear separation between the **retriever** and **generator** components. At a high level, the pipeline consists of four stages: (1) document ingestion and preprocessing, (2) embedding and indexing of text chunks, (3) retrieval of the most relevant chunks for a given query, and (4) answer generation conditioned on the retrieved context. All components are implemented in Python and orchestrated through a single console application entry point.

The code is organized into modular files under code/rag_sanskrit/.
The ingest.py module loads Sanskrit .txt files and splits them into overlapping chunks. The vectorstore.py module computes dense embeddings using a sentence-transformer model and stores them in a FAISS index, together with the original chunks. The generator.py module wraps a CPU-only multilingual seq2seq model and exposes a simple generate_answer() interface. The pipeline.py module coordinates these components: it builds or loads the index, accepts user queries, performs retrieval, calls the generator, and prints both the answer and a preview of the retrieved context.

4. Preprocessing Pipeline

The preprocessing pipeline starts by scanning the data/ directory for all UTF-8 Sanskrit .txt files. Each file is read into memory and its full text is appended to a corpus list. Since the documents are narrative prose, a simple character-level chunking strategy is used instead of sentence segmentation, which can be challenging and error-prone for Sanskrit. The chunking function creates overlapping windows of fixed length (e.g., 400 characters) with a configurable overlap (e.g., 50 characters) to preserve context across boundaries.

This approach ensures that each chunk remains small enough to be efficiently embedded and stored in memory, while the overlap reduces the risk that important information is split across chunks in a way that harms retrieval. All chunking parameters, such as CHUNK_SIZE and CHUNK_OVERLAP, are defined centrally in config.py so they can be tuned without changing the rest of the code. The result of preprocessing is a list of text chunks that covers the entire Sanskrit corpus and serves as input to the embedding and indexing stage.

5. Retrieval Component

The retrieval component is implemented in the vectorstore.py module using a **dense vector search** approach. Each text chunk produced by the preprocessing pipeline is encoded into a fixed-dimensional embedding using the pretrained model sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2, which supports many languages and scripts, including Indic scripts. These embeddings are stored in a FAISS IndexFlatL2 index, which allows efficient nearest-neighbour search based on Euclidean distance while remaining lightweight enough for CPU-only execution.

During a query, the user's question is encoded with the same sentence-transformer model, and the FAISS index is searched for the top-K most similar chunks. The system returns both the text of these chunks and their distances, which are then passed to the generation component. To avoid recomputing embeddings for every run, the FAISS index and the list of chunks are persisted to disk under data/index/, and can be reloaded on subsequent runs. This design provides fast semantic search over the Sanskrit corpus and cleanly isolates retrieval from generation.

6. Generation Component

The generation component is implemented in the generator.py module and is responsible for producing an answer conditioned on the retrieved Sanskrit passages and the user's query. It uses the pretrained model **google/byt5-small**, a byte-level multilingual encoder–decoder model that can run on CPU without requiring a GPU. The model and its tokenizer are loaded via the Hugging Face Transformers library and explicitly moved to the CPU device. A compact prompt is constructed that includes the retrieved Sanskrit passages, the user's question, and an instruction to answer in simple Sanskrit based only on the given text.

For each query, the system concatenates the top-K retrieved chunks into a single context block and feeds this, together with the question, into the ByT5 model. Generation parameters such as max_new_tokens and temperature are kept modest to control output length and randomness. Because google/byt5-small is a relatively small general-purpose multilingual model, its answers for complex classical Sanskrit can be noisy or partially ungrammatical. This limitation is documented as a conscious trade-off in favour of a lightweight, CPU-friendly solution; the generator module remains modular so that a larger or Sanskrit-specific model could be substituted in future work if more compute is available.

7. CPU Performance and Resource Usage

The entire pipeline is designed to operate on a standard CPU-only laptop environment. Both the sentence-transformer retriever model and the ByT5 generator run on CPU, and no GPU APIs are used. On first run, the main cost is downloading model weights and computing embeddings for all chunks; this is a one-time overhead. Once the FAISS index and chunk metadata are stored on disk, subsequent runs can reload the index and start answering queries more quickly.

In informal tests on a typical laptop, index building over the small Sanskrit corpus completes in a few seconds after the initial model download. Individual query latency is dominated by the generator and is typically on the order of a few seconds per question, which is acceptable for an interactive console tool. Memory usage remains within the capacity of a standard development machine because the corpus is small and both models are compact. These characteristics satisfy the assignment's requirement of efficient CPU-only inference without specialised hardware.

8. Limitations and Future Work

The current implementation focuses on correctness of the RAG pipeline and CPU feasibility rather than on perfect natural language fluency. The main limitation is the quality of generated answers: while retrieval reliably brings back relevant Sanskrit passages, the google/byt5-small model sometimes produces partially garbled or repetitive text when asked to answer detailed questions about classical Sanskrit narratives. This behavior stems from using a small, generic multilingual model that has not been specifically fine-tuned for Sanskrit question answering.

Several directions can address these limitations in future work. First, one could experiment with larger multilingual or Indic-centric models, or with Sanskrit-fine-tuned models, and plug them into the existing Sanskrit Generator interface. Second, more sophisticated preprocessing—such as sentence-level segmentation, morphological analysis, or sandhi-splitting—could improve both retrieval granularity and prompt quality. Finally, a lightweight web or GUI interface, along with caching of generated answers, could provide a more user-friendly experience on top of the existing console application.

9. How to Run the System

To run the system, the user first clones or downloads the repository and installs the Python dependencies listed in requirements.txt using pip install -r requirements.txt. The provided Sanskrit .txt files, converted from the assignment document, should be placed in the data/ directory. No GPU configuration is required; a standard Python 3.10+ environment with internet access for the initial model downloads is sufficient.

After setup, the user runs the command python main.py from the project root. On first execution, the script ingests the Sanskrit documents, creates overlapping text chunks, computes embeddings, builds the FAISS index, and downloads the necessary Hugging Face models. Once the index is ready, the program enters an interactive loop that repeatedly prompts for a query in Sanskrit or English. For each query, it retrieves the most relevant chunks, generates an answer, and prints both the answer and a snippet of the retrieved context. The user can exit the application at any time by typing exit or quit.

In addition to the console interface, a minimal web UI is implemented using Streamlit (app.py). Running streamlit run app.py starts a browser-based interface where the user can upload extra Sanskrit .txt files and submit queries via a text area. The backend logic is identical to the console pipeline: uploaded files are saved into data/, the index is rebuilt, and answers are generated on CPU.

The complete codebase, including the console pipeline, Streamlit UI, configuration, and report, is available at: https://github.com/tochWolf/RAG_sanskrit