✅ Technical Report

End-to-End Sanskrit Question Answering System using Retrieval-Augmented Generation

---

📌 Project Title

Sanskrit Document Retrieval-Augmented Generation (RAG) System

---

👤 Submitted By

Swaroop Kumbhalwar

📁 Project Folder

RAG_SANSKRIT_SWAROOP

---

1. Introduction

This project focuses on building a Retrieval-Augmented Generation (RAG) system designed specifically for answering queries from a collection of Sanskrit documents.

The complete system is implemented to run fully on CPU-based inference, ensuring it can work efficiently without requiring GPU acceleration.

RAG systems combine two key components:

- Retriever ➔ Finds relevant text passages
- Generator ➔ Produces an answer grounded in retrieved context

This approach improves factual correctness and reduces hallucinations compared to standalone language models.

---

2. System Architecture and Flow

2.1 Overall Workflow

The Sanskrit RAG pipeline follows this structured flow:

1. Load Sanskrit documents from the dataset directory
2. Clean and split documents into smaller chunks
3. Convert chunks into semantic embeddings
4. Store embeddings inside a FAISS vector index
5. Accept user queries from the command line
6. Retrieve the most relevant chunks using similarity search
7. Construct a prompt using query + retrieved context
8. Generate the final answer using a lightweight LLM
9. Display response along with source evidence

2.2 Modular Architecture

The project is designed in two main execution modules:

✅ A. Indexing Module (Offline Stage)

File: build_index.py

This module prepares the retrieval system before runtime:

- Loads all Sanskrit .txt documents from /data
- Applies chunking for better semantic retrieval
- Generates embeddings for each chunk
- Builds a FAISS vector database
- Saves the index to disk for later use

This stage is executed once during setup.

✅ B. Query Module (Online Stage)

File: query.py

This module performs real-time question answering:

- Loads the prebuilt FAISS index
- Converts the user query into embedding space
- Retrieves top-k relevant chunks using similarity search + MMR
- Passes the context into the generator model
- Prints final answer with supporting sources

---

3. Sanskrit Document Dataset

The dataset used in this project consists of multiple Sanskrit stories and subhashitas stored in .txt format inside the /data/ folder.

📄 Documents Included

File Name	Content Description
murkhabhritya.txt	Story of Shankhnad (मूर्खभृत्यस्य)
devbhakta.txt	Story on devotion and effort (देवभक्तः कथा)
ghantakarna.txt	Story of Ghantakarna (घण्टाकर्णः कथा)
kalidasa.txt	Clever Kalidasa story (चतुरस्य कालीदासस्य)
sheetam.txt	Short story about cold hardships (शीतं बहु बाधति)

Each document is stored separately to reduce retrieval confusion and improve grounding quality.

---

4. Sanskrit Document Preprocessing Pipeline

Efficient preprocessing is essential since Sanskrit text often contains complex structure and long sentences.

4.1 Document Loading

The system loads text files using:

- DirectoryLoader
- TextLoader(encoding="utf-8")

UTF-8 encoding ensures correct processing of Devanagari characters.

4.2 Chunking Strategy

Since large documents cannot fit directly into LLM context windows, documents are split into smaller pieces using:

✅ RecursiveCharacterTextSplitter

Configuration

- chunk_size = 350 characters
- chunk_overlap = 50 characters

Chunk overlap helps maintain sentence continuity across boundaries.

4.3 Embedding Generation

To enable semantic retrieval, each chunk is converted into an embedding vector using:

✅ sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2

Reasons for Selection

- Lightweight and CPU-efficient
- Strong multilingual performance
- Suitable for Sanskrit/Hindi-style scripts

---

5. Retrieval and Generation Mechanisms

5.1 Retriever Component

✅ Vector Store

The system uses:

✅ FAISS (Facebook AI Similarity Search)

FAISS enables fast and memory-efficient similarity search on CPU.

✅ Retrieval Strategy

The retriever applies:

✅ MMR (Max Marginal Relevance)

Advantages

- Retrieves relevant and diverse passages
- Avoids redundant repeated context
- Improves answer grounding

5.2 Generator Component

✅ Language Model Used

The answer generation is performed using:

✅ google/flan-t5-small

Reasons

- Small model suitable for CPU inference
- Faster response time

- Effective for QA-style generation tasks

✅ Prompt Design

A constrained prompt template is applied so that:

- Answers are produced only from retrieved context
- If the context lacks the answer, the model returns:

सन्दर्भे उत्तरं न लभ्यते।

6. Performance Observations

6.1 Latency

Component Approximate Time FAISS Retrieval < 1 second LLM Answer Generation Few seconds (CPU dependent)

Retrieval remains extremely fast even on basic machines.

6.2 Resource Usage

The system operates entirely without GPU.

Main memory usage comes from:

Embedding model loading

FAISS index storage

FLAN-T5 inference overhead

Using flan-t5-small keeps RAM usage minimal.

6.3 Retrieval Accuracy / Relevance

System accuracy depends mainly on retrieval quality.

Improvements Observed

Story-wise file separation reduces wrong context mixing

MMR increases relevance diversity

Example

Query: शंखनादः कः? Correctly retrieved from murkhabhritya.txt

Generated response:

शंखनादः गोवर्धनदासस्य भृत्यः अस्ति।

This demonstrates strong retrieval grounding and accurate generation.

7. Conclusion

The Sanskrit Document RAG System was successfully implemented as a fully CPU-based pipeline.

It combines:

Document retrieval using FAISS

Multilingual embeddings

Answer generation using FLAN-T5

The system satisfies all major project requirements:

Sanskrit document processing

Efficient chunk-based retrieval

Context-grounded answer generation

CPU-only deployment feasibility

8. Future Enhancements

Some possible improvements for extending this system in the future are:

- Adding transliteration support so users can ask questions in English/IAST input
- Expanding document support beyond .txt by enabling PDF ingestion
- Exploring Sanskrit-focused embedding models to further improve retrieval accuracy
- Developing a simple web-based interface using Streamlit or Flask for better usability