Sanskrit Document Retrieval-Augmented Generation (RAG) System

Name: Rushabh Katekhaye

---

1. Introduction

This project was undertaken to explore whether a modern Retrieval-Augmented Generation (RAG) system can be applied effectively to Sanskrit documents under strict computational constraints.

Unlike English, Sanskrit is a classical language with complex morphology, rich inflection, and script-specific features. Most modern NLP systems are optimized for English and GPU-based environments. Therefore, the primary objective of this project was not only to build a working system, but also to understand the limitations, trade-offs, and practical challenges involved when working with Sanskrit and CPU-only environments.

Rather than assuming a single correct solution, this project followed an experimental approach: testing multiple architectures and models, observing their behavior, documenting failures, and finally selecting the most reliable approach under the given constraints.

.

---

2. Problem Statement

The problem addressed by this project is:

Can we design a system that allows users to ask questions about Sanskrit documents and receive meaningful answers, using only local CPU resources and no external APIs?

The system must:

- Accept Sanskrit text documents as input.
- Allow user queries in Sanskrit or English.

- Retrieve relevant information from documents.
- Provide accurate responses.
- Operate fully locally and on CPU.

---

3. Exploration Strategy

Instead of assuming a single method, the project followed an iterative exploration strategy:

1. Build a retrieval pipeline.
2. Try different generation models.
3. Measure output quality, performance, and stability.
4. Identify practical limitations.
5. Choose the best trade-off between accuracy, reliability, and constraints.

Each iteration was treated as an experiment, not a failure.

---

4. System Overview

The system is structured into four conceptual layers:

1. **Document Processing** – Loading and splitting Sanskrit texts.
2. **Semantic Retrieval** – Converting text into embeddings and performing similarity search.
3. **Answer Generation / Extraction** – Producing answers from retrieved text.
4. **User Interface** – Terminal-based question-answer interaction.

---

5. System Architecture

The system follows the standard RAG architecture:

1. **Document Loader**
2. **Text Preprocessing**

3. **Chunking**
4. **Embedding**
5. **Vector Store (FAISS)**
6. **Retriever**
7. **Generator / Extractive Answering Layer**

User Query
↓
Embedding Model (Multilingual)
↓
Vector Store (FAISS)
↓
Relevant Chunks Retrieved
↓
Answer Generation OR Extractive Output
↓
Final Response to User

Component	Description
Document Loader	Loads Sanskrit text files
Text Splitter	Breaks documents into chunks
Embedding Model	Converts text into vectors
Vector Database	FAISS index for similarity search
Generator (optional)	Generates answers from retrieved context
Extractive Module	Outputs relevant text directly

6.Tools & Libraries

Tool / Library	Purpose
Python	Main programming language
SentenceTransformers	Multilingual embeddings
FAISS	Vector similarity search
Transformers	Model loading
Torch	Neural model backend
Ollama	Local LLM runner

VS Code	Development environment

---

7. Models Evaluated and Their Outcomes

Model	Source	Purpose	Reason for Selection	Observed Result	Reason for Rejection
IndicBART (ai4bharat/IndicBART)	AI4Bharat	Indic language text generation	Trained specifically on Indian languages including Sanskrit	Generated mixed scripts (Devanagari + Tamil/Bengali), output was unreadable	Script mixing, poor control on output
ByT5 (google/byt5-small)	Google	Character-level multilingual generation	Handles Unicode and rare scripts well	Clean characters but weak semantic understanding and poor answers	Lacked reasoning and relevance
mT5 (google/mt5-small)	Google	Multilingual text-to-text generation	Supports 100+ languages including Sanskrit	Extremely slow on CPU, answers inconsistent	Performance too slow and unreliable
Flan-T5 (google/flan-t5-base)	Google	Instruction-tuned QA	Good instruction following	Hallucinated answers not grounded in retrieved context	Violated grounding requirement

Phi (Ollama)	Microsoft	Lightweight CPU LLM	Small, CPU-friendly	Timeout issues, unstable responses	Unreliable under CPU load
Gemma:2B (Ollama)	Google	Local instruction LLM	Better reasoning than Phi	Timeout, memory spikes, occasional hangs	Stability issues on Windows CPU
Claude.ai (web)	Anthropic	Strong reasoning baseline	Used only for conceptual testing	Timeout, not locally deployable	Violates local constraint
SentenceTransformers (MiniLM multilingual)	SBERT	Embedding model	Supports multilingual semantic search	Retrieval worked well (~75% match)	Kept as final retrieval model

8. Experiments and Iterations

Iteration 1 — IndicBART

- Advantage: Indic-specific training
- Issue: Generated mixed Devanagari, Bengali, Tamil characters.

Iteration 2 — ByT5

- Advantage: Character-level tokenization
- Issue: Weak reasoning, repetitive answers, slow CPU inference.

Iteration 3 — mT5

- Advantage: Multilingual capability
- Issue: Very slow on CPU, inconsistent output quality.

Iteration 4 — Ollama (Phi, Gemma)

- Advantage: Fully local generation
- Issues:
    - Model loading delays
    - Timeout errors
    - High RAM consumption
    - Inconsistent response stability

Iteration 5 — Extractive QA (Final)

- Advantage:
    - Clean Sanskrit output
    - No hallucination
    - Very fast
    - Highly reliable
- Tradeoff: No abstract generation, only extraction.

---

9. Issues and Challenges

Category	Problem
CPU Constraint	Large models slow or unstable
Script Handling	Mixed scripts in output
Model Size	Memory limits
Libraries	Version conflicts (huggingface, torch)
Timeouts	Local LLM inference delays
Data Scarcity	Limited Sanskrit corpora

---

10. Results

Metric	Observation
Retrieval Accuracy	70–80% similarity for Sanskrit queries
Response Time	2–3 seconds
Output Quality	Clean Sanskrit text
Reliability	High
Hallucination Risk	None (extractive)

---

11. Conclusion

The project demonstrates that under strict CPU-only and offline constraints, reliable Sanskrit question-answering is achievable through retrieval and extractive techniques. While generative models offer expressive summaries, they are currently impractical for this use case without GPU or cloud resources.

The extractive approach ensures correctness, transparency, and usability, making it suitable for academic and constrained environments.

---

12. Limitations

- No abstractive summarization
- No deep reasoning beyond document content
- Performance bound by CPU hardware
- Limited Sanskrit pretrained models

---

13. Future Scope

- GPU -based inference for generative models
- Fine-tuning on Sanskrit corpora

- Custom Devanagari tokenizers
- Knowledge graph integration
- Hybrid retrieval + reasoning models
