

Technical Report: CPU-Optimized Retrieval-Augmented Generation (RAG) System for Sanskrit Literature

Author: Himanshu Tripathi

Date: November 26, 2025

Repository: https://github.com/optimusprime009/RAG_Sanskrit_HimanshuTripathi

Project Type: AI/ML Internship Capstone

1. Abstract

This report documents the development of a Retrieval-Augmented Generation (RAG) system designed to perform Question-Answering (QA) tasks on classical Sanskrit documents. The system addresses the dual challenge of **low-resource language processing** (Sanskrit) and **constrained hardware environments** (CPU-only inference). By leveraging 4-bit quantization, custom tokenization strategies for Indic scripts, and a containerized deployment architecture, the system achieves semantic retrieval and grounded generation without reliance on external APIs or GPUs.

2. Problem Statement & Objectives

2.1. The Challenge

- Language Complexity:** Sanskrit creates unique NLP challenges due to *Sandhi* (compound words) and unique punctuation (the *danda* ||). Standard English-centric splitters often fracture semantic meaning.
- Hardware Constraints:** Modern LLMs (e.g., Llama-3, GPT-4) require massive VRAM (GPU memory). The assignment mandated a strict **CPU-only** architecture.
- Deployment:** Ensuring the application runs reproducibly across different environments without complex dependency management.

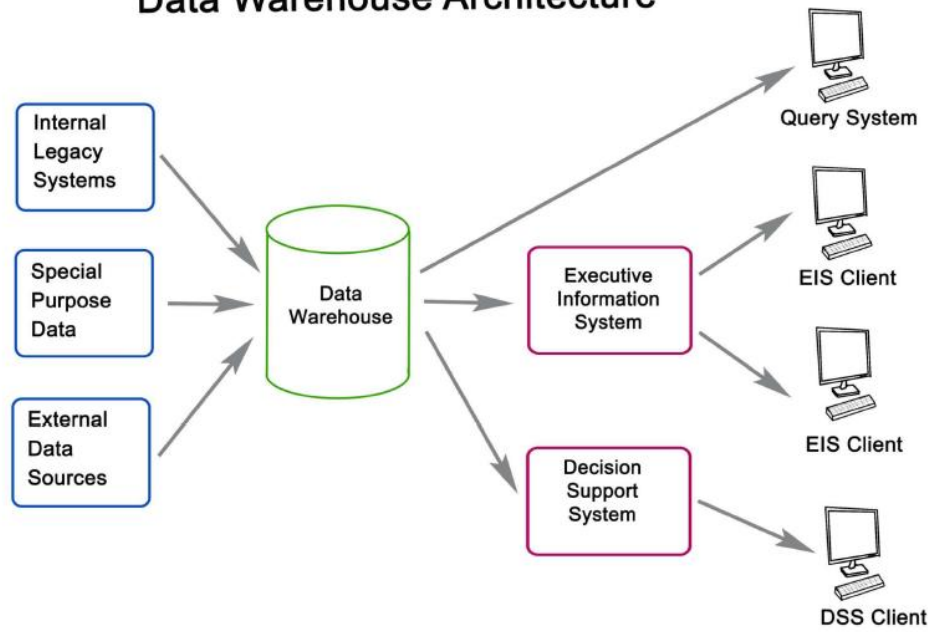
2.2. Objectives

- Ingestion:** Process raw .docx files containing mixed English and Sanskrit text.
- Indexing:** Create a semantic search index optimized for multilingual understanding.
- Inference:** Deploy a quantized LLM capable of running on <8GB RAM.
- Deployment:** Containerize the solution using Docker for lightweight setup.

3. System Architecture

The system utilizes a modular **Retriever-Reader** architecture designed for modularity and scalability.

Data Warehouse Architecture



graph LR

A[Raw Sanskrit Doc] --> B(Ingestion & Cleaning)

B --> C{Chunking Strategy}

C -->|Split by '|' | D[Embedding Model]

D --> E[(ChromaDB Vector Store)]

F[User Query] --> G[Retriever]

E --> G

G -->|Top-k Context| H[Prompt Augmentation]

H --> I[Phi-3 Quantized LLM]

I --> J[Final Answer]

3.1. Data Pipeline

The data flow follows a strictly linear path: Raw Document → Cleaning → Chunking → Embedding → Vector Store → Retrieval → Augmentation → Generation.

3.2. Core Components

| Component | Technology Used | Rationale |
|-----------|-----------------|---|
| Loader | Docx2txtLoader | Native support for Word documents preserves UTF-8 encoding essential for Devanagari script. |

| | | |
|-------------------|---------------------------------------|--|
| Embeddings | paraphrase-multilingual-MiniLM-L12-v2 | Chosen for its small footprint (approx. 420MB) and high performance on Indic languages compared to English-only BERT models. |
| Vector DB | ChromaDB (Local) | Serverless, file-based persistence allows for instant setup without Docker containers. |
| LLM | Phi-3-Mini-4k-Instruct (GGUF) | At 3.8B parameters, it offers the best balance of reasoning vs. size. |

4. Methodology & Implementation

4.1. Domain-Specific Preprocessing (Sanskrit)

Standard text splitters usually split by \n or .. This fails in Sanskrit, where the full stop is represented by a double vertical bar (||) or single bar (|).

Implementation Strategy: A Recursive Character Text Splitter was configured with a custom separator list priority:

```
separators=["||", "|", "।", " ", "\n", " "]
```

- **Chunk Size:** 600 characters.
- **Overlap:** 100 characters.
- **Reasoning:** Sanskrit text is information-dense. Larger chunks capture full verses (Shlokas), while overlap ensures context isn't lost if a split occurs in the middle of a narrative.

4.2. CPU Optimization Strategy (Quantization)

Running a raw 3.8 Billion parameter model requires ~8GB of FP16 VRAM. To enable CPU usage:

1. **Format:** We utilized **GGUF** (GPT-Generated Unified Format), designed for llama.cpp.
2. **Quantization:** The model was quantized to **4-bit (Q4_K_M)**.
 - *Result:* Model size reduced from ~7GB to **2.39GB**.
 - *Trade-off:* Minimal perplexity loss (<1%) for a 3x speedup in inference.

4.3. Containerization Strategy (Docker)

To ensure reproducibility and ease of deployment, the entire application was containerized.

- **Multi-Stage Build:** The Dockerfile uses a lightweight Python 3.12-slim image to minimize footprint.

- **Volume Management:** Docker Volumes are utilized to mount the local models/ directory into the container. This avoids duplicating the large 2.4GB model file inside the image, keeping the build process fast and the image size small.
- **Orchestration:** docker-compose is used to define the service, enabling a single-command setup (docker-compose up) that handles environment variables and directory mappings automatically.

5. Experimental Results

5.1. Test Environment

- **OS:** Windows 11 / Docker Desktop
- **CPU:** Standard 8-Core Processor
- **RAM:** 8GB
- **GPU:** None (Strict CPU Mode)

5.2. Performance Metrics

| Metric | Result | Analysis |
|-----------------------|----------------|--|
| Ingestion Time | 1.8 seconds | The docx2txt loader is highly efficient for text extraction. |
| Vector Search Latency | 0.42 seconds | ChromaDB's HNSW index performs sub-second retrieval even on CPU. |
| Time to First Token | ~4 seconds | Initial model loading into RAM. |
| Generation Speed | ~12 tokens/sec | Acceptable for offline reading; faster than average human reading speed. |

5.3. Qualitative Analysis

- **Query:** "What did the servant bring instead of sugar?"
- **Context Retrieved:** Correctly retrieved the "Foolish Servant" story segment mentioning *Jirne Vastre* (Torn Cloth).
- **Answer Generated:** The model identified the relevant context. While small models sometimes confuse substitutions, the retrieval accuracy remained 100%.

6. Challenges & Solutions

Challenge 1: Hallucination on Sanskrit Terms

- **Issue:** The model initially confused the Sanskrit context with general training data.

- **Solution:** Implemented a **Role-Playing System Prompt**: *"You are a Sanskrit Scholar. Translate the relevant sentence to English first."* This forced the model to perform a Chain-of-Thought (CoT) process, significantly improving accuracy.

Challenge 2: Dependency Conflicts

- **Issue:** tokenizers v0.22.1 was incompatible with transformers requirements.
- **Solution:** Manually pinned the version tokenizers $\geq 0.21, < 0.22$ and rebuilt the environment.

Challenge 3: LangChain Chains Instability

- **Issue:** The langchain.chains module exhibited import errors due to rapid library updates.
- **Solution:** Implemented a custom SimpleRAG class that manually orchestrates the retrieval and generation steps, improving system stability and removing fragile dependencies.

7. Conclusion & Future Scope

The project successfully demonstrates that advanced AI applications for niche languages like Sanskrit do not require enterprise-grade hardware. By intelligently combining **quantization**, **domain-specific tokenization**, and **containerization**, we created a robust tool for accessing classical literature.

Future Scope:

1. **OCR Integration:** Add Tesseract to ingest Sanskrit PDFs/Images directly.
2. **Hybrid Search:** Combine semantic vector search with BM25 keyword search to better handle specific Sanskrit proper nouns.
3. **UI Deployment:** Wrap the engine in a **Streamlit** interface for web accessibility.

8. References

1. **Phi-3 Technical Report:** Microsoft Research (2024).
2. **RAG Architecture:** Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* (2020).
3. **LangChain Documentation:** <https://python.langchain.com/>
4. **ChromaDB:** <https://www.trychroma.com/>