# Technical Report: Sanskrit Document Retrieval-Augmented Generation (RAG) System

## 1. Introduction & Objective

This project implements a minimal, end-to-end **Retrieval-Augmented Generation (RAG)** pipeline designed to answer user queries using a small collection of **Sanskrit documents**, with an explicit constraint of **CPU-only inference**.

The primary objective is to:

- Ingest Sanskrit documents from a local folder
- Preprocess and chunk the documents for retrieval
- Retrieve the most relevant chunks for a user query
- Produce an answer either **extractively** (selecting a Sanskrit sentence from retrieved text) or **generatively** (using a small local/CPU-friendly Transformer model)

CPU-only operation is achieved by:

- Pinning a CPU build of PyTorch (`torch==2.10.0+cpu`)
- Loading HuggingFace models without CUDA usage
- Using lightweight retrieval (TF-IDF) and small generation models

## 2. System Architecture & Workflow

### 2.1 High-level pipeline

The implementation follows a standard RAG workflow split into three runnable steps:

1. **Ingestion + Chunking** (`code/ingest.py`) → writes `artifacts/chunks.jsonl`
2. **Indexing** (`code/index.py`) → writes `artifacts/tfidf_index.joblib`
3. **Retrieval + Answering (RAG CLI)** (`code/rag_cli.py`) → retrieves contexts and answers

### 2.2 Architecture diagram (derived from code)

```
+-------------------+
|   data/           |
|  .txt/.pdf/.docx  |
+---------+---------+
          |
          v
+-------------------+         writes
| code/ingest.py     |--------------------> artifacts/chunks.jsonl
| - read_any()       |
| - normalize_text() |
| - chunk_text()    |
+---------+---------+
          |
          v
+-------------------+         writes
| code/index.py     |-------------------> artifacts/tfidf_index.joblib
| - TF-IDF (char)   |
| - joblib dump     |
+---------+---------+
          |
          v
+-------------------+
| code/rag_cli.py   |
| - query normalize |
| - retrieve top-k  |
| - build prompt    |
| - answer (mode)   |
+-------------------+
```

### 2.3 Retriever vs Generator separation

The separation is implemented at the **functional** level:

- **Retriever responsibilities**
  - Loading index (`joblib.load`) and chunk store (`chunks.jsonl`)
  - Transforming query and scoring chunks via TF-IDF similarity (`retrieve()`)
- **Generator responsibilities**
  - Constructing a prompt from retrieved contexts (`build_prompt()`)
  - Generating an answer using either:
    * Extractive mode (`extractive_sanskrit_answer()` and additional heuristics)
    * Generative mode (`generate_flan_t5()` or `generate_gpt2()`)

Although these components live in the same script (`rag_cli.py`), they are decoupled in the sense that retrieval is performed first and the generator consumes only the retrieved contexts + query.

## 3. Sanskrit Documents Used

### 3.1 What is present in this repository

The current repository contains **one document** under `data/`:

- `data/Rag-docs.docx`

No `.txt` or `.pdf` files are currently present in `data/`.

### 3.2 Supported formats by the implementation

Even though the current `data/` folder contains a `.docx`, the ingestion code supports:

- `.txt` via `read_txt()`
- `.pdf` via `read_pdf()` (uses `pypdf`)
- `.docx` via `read_docx()` (uses `python-docx`)

### 3.3 Domain/type of Sanskrit content

From the code and README examples (queries about , , and speaker identification like " "), the dataset appears to contain **Sanskrit prose/verse snippets and factual/attribution-style statements** suitable for short QA.

**Note:** This section is limited by the repository contents and does not assume additional unseen corpora.

## 4. Preprocessing Pipeline

Preprocessing is split between ingestion-time cleaning/chunking and query-time normalization.

### 4.1 Text extraction

Implemented in `code/ingest.py`:

- **TXT**: read using UTF-8 with error ignoring
- **DOCX**: extracted paragraph-wise using `python-docx`
- **PDF**: extracted page-wise using `pypdf.PdfReader` and `page.extract_text()`

### 4.2 Cleaning and normalization

Implemented in `code/ingest.py`:

- Normalizes newline conventions (`\r\n` → `\n`)
- Strips whitespace per line
- Collapses excessive blank lines

Query-side normalization in `code/rag_cli.py`:

- Removes common punctuation, danda ( ) handling
- Whitespace normalization
- A matching-normalization step removes some diacritics ( , , ) for more forgiving overlap checks

### 4.3 Chunking strategy

Implemented in `code/ingest.py`:

- Fixed-size character chunking
  - Default `chunk_size=900` characters
  - Default overlap `overlap=150` characters

Chunks are stored as JSON Lines (`artifacts/chunks.jsonl`) with metadata:

- `doc_id`, `source_path`, `chunk_id`, and `text`

### 4.4 Devanagari vs Roman/transliterated input

Implemented in `code/rag_cli.py`:

- If the query contains Latin letters, the CLI attempts transliteration to Devanagari using `indic_transliteration`.
- It tries multiple roman schemes (IAST/ITRANS/HK) and uses the first successful conversion.

## 5. Retrieval Mechanism

### 5.1 Retriever type

The retriever is **vector-based** using classical IR features:

- **TF-IDF character n-grams** (3–5) using scikit-learn's `TfidfVectorizer(analyzer="char")`

This is implemented in `code/index.py`.

### 5.2 Indexing method

- The vectorizer is fit on all chunk texts
- The TF-IDF sparse matrix is stored
- Both are serialized into `artifacts/tfidf_index.joblib` via `joblib.dump`

### 5.3 Similarity/search strategy

- Query is transformed by the same vectorizer
- Similarities are computed via `sklearn.metrics.pairwise.linear_kernel` (cosine similarity for L2-normalized TF-IDF)

In `rag_cli.py`, a small heuristic **keyword overlap boost** is added to improve relevance for some queries.

### 5.4 Why this approach fits Sanskrit documents

This repository uses character n-grams, which is a pragmatic choice for Sanskrit because:

- It is robust to inflectional variation and tokenization issues
- It works with Devanagari Unicode without needing a specialized tokenizer
- It is CPU-efficient and avoids heavy embedding models

## 6. Generation Mechanism (LLM)

### 6.1 Answer modes

Implemented in `code/rag_cli.py`:

- **Extractive (default)**: selects a Sanskrit sentence from the retrieved contexts
- **Generative**: constructs a prompt and generates an answer using a Transformer model

### 6.2 Models supported

- **Seq2Seq backend** (`--backend flan_t5`):
  - `AutoModelForSeq2SeqLM` (default `--gen_model google/mt5-small` as shown in README)
- **Causal LM backend** (`--backend gpt2`):
  - `AutoModelForCausalLM` loaded from a local directory (`model/`)

### 6.3 How retrieved context is passed

- The prompt is constructed as:
  - An instruction header
  - The question
  - A concatenated context block with top-k retrieved chunks (with scores and source metadata)

### 6.4 CPU compatibility considerations

- Uses `@torch.inference_mode()` for generation functions
- Loads models without CUDA calls
- Provides a stable extractive mode to avoid heavy generation costs
- Truncates inputs (e.g., GPT-2 path computes safe max input tokens)

## 7. Performance & CPU Optimization

This repository does not include benchmark scripts or logged measurements, so exact performance numbers are not claimed here.

However, based on the implementation, the following practical observations apply:

- **Retrieval latency**: typically low (sparse matrix similarity on CPU), scaling with number of chunks.
- **Generation latency**: dominated by Transformer inference.
  - `google/mt5-small` is relatively small but can still be slow on CPU for large `max_new_tokens`.
  - Local GPT-2 generation uses sampling, which can be slower/unpredictable.

CPU-oriented optimizations present in the code:

- Lightweight TF-IDF retrieval (no neural embeddings)
- Optional extractive answering (fast, deterministic)
- Inference-only mode (`torch.inference_mode()`)
- Input truncation for GPT-2 to avoid exceeding model context

Recommended measurements (future work):

- Time each pipeline stage (ingest/index/retrieve/generate)
- Track memory via `psutil` or Task Manager during generation

## 8. Limitations & Future Improvements

### 8.1 Current limitations (observed from code/repo)

- **Dataset format mismatch with assignment**: `data/` currently contains only a `.docx` file; no `.txt/.pdf` examples are present in the repository.
- **PDF limitations**: `pypdf` extraction works for text-based PDFs; scanned PDFs would require OCR.
- **Preprocessing is minimal**: no advanced Sanskrit normalization (Unicode canonicalization, sandhi-aware processing, sentence segmentation).
- **No evaluation suite**: no automated tests or retrieval/generation quality benchmarks.
- **Modularity**: retriever and generator are logically separated but live in one CLI script (`rag_cli.py`).

### 8.2 Realistic future enhancements

- Add a `data/` sample set in `.txt` and `.pdf` to match assignment deliverables
- Add stronger normalization (Unicode NFC/NFKC, danda-aware sentence splitting)
- Add citations in output (show which chunk(s) support the answer)
- Implement a vector-embedding retriever (still CPU-friendly) as an optional backend
- Split code into modules: `loader.py`, `preprocess.py`, `retriever.py`, `generator.py`
- Add a small benchmarking script and a JSONL evaluation set

## 9. Conclusion

This project delivers a runnable, CPU-only Sanskrit RAG proof-of-concept with:

- Multi-format ingestion (`.txt`, `.pdf`, `.docx`) and chunking
- TF-IDF character n-gram retrieval over chunked text
- A query interface that can accept Devanagari and attempts transliteration from roman input
- Extractive and generative answering paths using CPU-compatible Transformer models

Relative to the original assignment, the core architecture (Loader $\rightarrow$ Retriever $\rightarrow$ Generator) is implemented and runnable. The largest gap is not in the pipeline code itself, but in the **currently included corpus samples**, since the provided `data/` folder contains only a `.docx` document rather than `.txt/.pdf` examples.