2025

GIKI Prospectus Q&A Chatbot using Retrieval-Augmented Generation (RAG)

PROJECT REPORT

MUHAMMAD MUDASIR YASIN MUGHAL

GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES AND TECHNOLOGY

Table of Contents

Brief Overview

This project implements a **Retrieval-Augmented Generation (RAG)** chatbot tailored for GIKI-related documents (prospectus, fee structure, academic rules, student handbook, advisory handbook, etc.). The system allows users to upload up to **five** documents (PDF, DOCX, TXT), builds a searchable vector index over semantically meaningful chunks, and answers user queries by retrieving relevant chunks and conditioning an LLM to produce concise, source-cited responses.

Primary goals

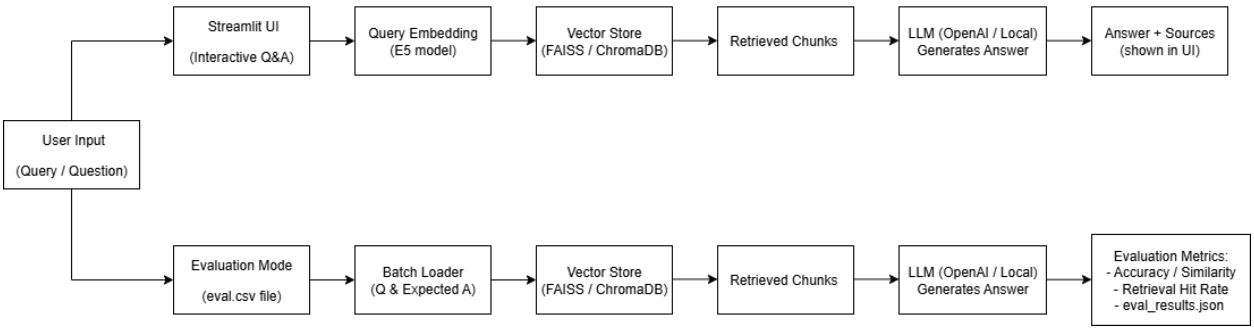
- Accept and parse multiple document formats.
- Chunk content into retrieval-friendly passages with overlap.
- Create embeddings and store them in a vector index (FAISS).
- Retrieve the top-k relevant chunks for a query, optionally re-rank (MMR).
- Generate natural language answers using a pretrained LLM (OpenAI models when available; local Flan-T5 fallback).
- Provide a Streamlit web interface with language toggle (English/Urdu), source display, and downloadable conversation history and evaluation results.

Key implementation choices

- Embeddings: `intfloat/multilingual-e5-base` (E5-style prefix handling) or `all-MiniLM-L6-v2` as alternative.
- Vector DB: FAISS `IndexFlatIP` with normalized embeddings (cosine similarity via inner product).
- Chunking: sentence-aware splitting and greedy char-length chunking (`target_chars=1800`, `overlap_chars=300`) — a proxy for ~400–500 tokens per chunk.
- Generator: `OpenAI: gpt-4o-mini` (or `gpt-4o`) when API key present; fallback `google/flan-t5-base` local pipeline otherwise.
- UI: Streamlit app supporting upload (≤5 files), index build/reset, chat, language toggle, evaluation CSV runner, and downloads.

System Architecture

The following diagram illustrates the end-to-end workflow of the GIKI Prospectus Q&A Chatbot, showing how documents are processed, embedded, stored, and queried through the RAG pipeline.



Processes and Methods

Below we describe each major pipeline component and the design decisions made in the code.

Document ingestion

Supported file types: `.pdf`, `.docx`, `.txt`.

- **PDF**: parsed via PyMuPDF (`fitz`) using `page.get_text("text")`.
  - Pros: fast, extracts layout text for typical born-digital PDFs.
  - Caveat: scanned or image PDFs require OCR (not implemented by default).
- **DOCX**: parsed via `python-docx`, concatenating paragraph text.
- **TXT**: decoded (UTF-8 fallback) and split by lines.

For each page or document, text is preprocessed into sentences using a simple regex-based sentence splitter and further prepared for chunking.

Chunking strategy

- **Sentence-aware splitting**: initial segmentation into sentences (punctuation-based) and line breaks.
- **Greedy chunk assembly**: collects sentences until a character-length threshold (`target_chars=1800`) is reached, then emits a chunk and continues. A small overlap (`overlap_chars=300`) is kept between chunks to preserve cross-boundary context.

Rationale: token-counting libraries (e.g., `tiktoken`) are ideal but the implemented char-length heuristic provides a simple, robust proxy for ~400–500 token chunks suitable for retrieval and context injection.

Each chunk is stored with metadata: `source` (filename) and `page` (where applicable), and `text` (the chunk itself).

Embedding & vector store

- Embeddings generated with Sentence-Transformers (`SentenceTransformer`) using `intfloat/multilingual-e5-base` by default, which helps support multilingual content (English/Urdu).
- E5 models expect `passage:` / `query:` prefixes for passages vs queries — the code automatically applies these prefixes.
- Embeddings are L2-normalized and stored as `float32` vectors.
- Vector store: FAISS `IndexFlatIP` (inner product) with normalized vectors equals cosine similarity.

Design notes

- FAISS index is built in memory and stored in Streamlit session state; it does not persist across process restarts in the current implementation.
- Each vector's textual metadata is stored separately in `st.session_state.meta` for later display and context assembly.

Retrieval and ranking

- **Initial retrieval**: FAISS `index.search(query_vec, k_candidates)` returns candidate indices.
- **Optional MMR re-ranking**: to diversify the selected passages, the code implements a small MMR routine. Because FAISS in this usage doesn't provide the stored vectors back, the routine **re-embeds candidate passages at query time** (costly but acceptable for small demos).
- Final top-k indices and their metadata are returned for context assembly.

Prompting & generation

Prompt structure

- A short system prompt instructs the assistant to answer only from the provided context, include inline citations like `[source: <filename> p.<page>]`, and to respond in English or Urdu per user choice.
- The user prompt combines the retrieved `context` (concatenated chunks) and the natural user `Question` with the instruction to use only the context.

Generator choices

- **Primary**: OpenAI via the new SDK (`OpenAI` client). Models supported in the UI: `gpt-4o-mini` and `gpt-4o`.

- **Fallback**: Transformers `google/flan-t5-base` pipeline for text2text generation when no OpenAI key is present.

Notes on generation:

- Temperature defaults to 0.0 for deterministic answers; user can adjust.
- The LLM should be used primarily to *rephrase and synthesize* retrieved passages; the system prompt explicitly restricts it to context-only responses to reduce hallucinations.

UI / UX

Built with Streamlit, the app provides:

- Sidebar settings: select embedding model, generator model, creativity (temperature), top-k.
- Upload control for up to 5 files and buttons to build/reset index.
- Chat interface with stream of messages, source expander for each answer, and download options (Markdown & PDF).
- Evaluation panel: upload a CSV (`question, expected`) and run the evaluation, producing semantic similarity and retrieval hit metrics, downloadable as CSV.

Evaluation pipeline

The code includes a lightweight evaluation routine:

- For each row with `question` and `expected`, the system obtains an `answer` using the current pipeline.
- **Semantic similarity**: embeds `expected` and `answer`, then computes the dot product (cosine) between normalized vectors.
- **Retrieval hit**: a heuristic using **6-word n-gram overlap** between expected answer and any retrieved chunk.

Outputs per test sample: question, expected, answer, semantic_sim, retrieval_hit, sources.

Challenges and Hurdles

This section documents practical challenges encountered when building and running this RAG pipeline.

Parsing & data quality

- **Scanned PDFs**: born-image PDFs (scans) require OCR (Tesseract or cloud OCR) to extract text; current pipeline will return empty pages.
- **Complex DOCX structures**: tables, footnotes, and headers aren't fully preserved by paragraph-based parsing — you may lose structural cues.

Chunking & context window

- Char-based chunking is a heuristic; token-accurate chunking (via `tiktoken`) is preferable to ensure chunks fit model context limits precisely.
- Overlap size is a tunable hyperparameter — too small loses continuity, too large increases retrieval redundancy and cost.

Retrieval & reranking inefficiency

Current MMR implementation **re-embeds** candidate passages at query time because the FAISS index in use does not directly expose stored vectors. This increases CPU cost and latency for each query. A production system should either store vectors externally (e.g., in a DB) or use an index that returns vectors.

Model quality & hallucinations

LLMs can still hallucinate even if constrained by context. The system prompt restricts answers to the uploaded content, but verification steps (e.g., cross-encoder verification or exact phrase matching) can be added as safeguards.

Resource & scalability

FAISS in-memory index is fine for demos but not ideal for large-scale or multi-user deployments. For larger doc collections, use managed vector stores (Pinecone, Chroma, Weaviate) or persistent FAISS with on-disk indexes.

Multilingual (Urdu) quality

Multilingual E5 embeddings support Urdu for retrieval, but **generation quality** in Urdu depends on the LLM used; local Flan-T5 performs poorly in Urdu compared to OpenAI models.

Results and Metrics

How to measure

Use the built-in evaluation pipeline; key metrics:

- **Semantic similarity (cosine)** between expected answers and generated answers — computed using embedding dot product. Range: `[-1, 1]` (with normalized vectors). Higher = closer.
- **Retrieval hit (boolean)**: whether any retrieved chunk contained a 6-word sequence overlapping with the expected answer (heuristic). This approximates whether retrieved context contained a direct match.
- **Latency**: time per query (split into retrieval time and LLM response time). Useful for UX and cost planning.
- **Index size & chunk counts**: total chunks indexed, average tokens per chunk (approximate), and storage/memory footprint.

Optional advanced metrics:

- **Exact-match / F1** (if expected answers are short facts) — useful for factoid-style Q&A.
- **ROUGE / BLEU** for longer paraphrase comparisons.
- **Human relevance judgments** (1–5 rating) for a sample of answers.

Example evaluation protocol

- Prepare `eval.csv` with `question,expected` rows (50–200 test pairs covering common, edge, and tricky questions).
- Upload your 5 GIKI docs, build index.
- Run the evaluation from the sidebar.
- For each sample collect: `semantic_sim`, `retrieval_hit`, `sources`.
- Aggregate:
    - Mean semantic similarity
    - Fraction of samples with `retrieval_hit == True`
    - Average time per query
    - Human-rated accuracy

Suggested thresholds

- **Mean semantic similarity ≥ 0.65**: good alignment between generated answer and expected.
- **Retrieval hit ≥ 0.80**: good retrieval coverage (i.e., most expected answers exist in retrieved chunks).
- **Average latency < 2s** (server-side embedding + FAISS + local model) or **< 3s–4s** when calling OpenAI (network + model time).

These thresholds are rough — actual numbers depend on document length, query complexity, and model choice.

What the code currently reports (automatic outputs)

- Indexing summary: number of chunks indexed and number of files processed (displayed after Build Index).
- Evaluation CSV outputs a table with `semantic_sim` and `retrieval_hit` per question and a downloadable `eval_results.csv`.

Limitations and Risks

Hallucination & overconfidence

LLMs may output plausible-sounding statements not present in the documents. Mitigations: strong system prompts, citation enforcement, cross-checking answers against retrieved chunks, and returning verbatim spans rather than freeform summaries for high-risk facts.

Data privacy & leakage

Users upload potentially sensitive documents. The current demo uses local memory; if deployed to a cloud service (or using OpenAI API), ensure compliance: encryption at rest/in transit, access controls, and data retention policies.

OCR & data loss

Scanned/photographed docs require an OCR pre-step. Without it, important content will be missing from the index.

Language limitations

Urdu generation quality depends on generator model. Local Flan-T5 is limited; OpenAI models perform better but add cost. Translation pipelines can mitigate this.

Cost and latency

OpenAI calls introduce cost per token and variable latency. Large retrieved contexts increase input tokens and cost.

Persistence & multi-user concerns

Streamlit session-state and in-memory FAISS are not persistent. Multi-user environments need a server-backed store and concurrency management.

Future Improvements

Below are prioritized improvements that will raise robustness, performance, and usability.

Better chunking & token accounting

Use tokenizers (`tiktoken` or model-specific tokenizers) to create exact token-sized chunks that fit target model context windows.

Persistent, production-grade vector store

Replace in-memory FAISS with a managed vector DB (Pinecone, Chroma, Weaviate) or persistent FAISS index on disk; store vectors alongside metadata for fast re-ranking.

Avoid re-embedding during reranking

Preserve passage embeddings at index build time (store them in the meta store and/or the vector DB) so MMR and cross-encoder re-rankers can reuse them without re-computation.

Add OCR pipeline

Integrate Tesseract or cloud OCR (Google Vision / AWS Textract) for scanned PDFs.

Hybrid retrieval & reranking

Combine sparse (BM25) and dense retrieval for robust recall, and use a cross-encoder re-ranker for precision.

Improve Urdu support

Use a stronger multilingual generator or a translation step: (i) generate in English, (ii) translate to Urdu using a translation API or a fine-tuned model.

Answer grounding & provenance

Return verbatim snippets when answers rely heavily on facts and provide clear, clickable citations that open the source page/offset.

Monitoring, logging & CI

Add telemetry for latency, error rates, cost per request; unit tests and integration tests for ingestion, indexing, retrieval, and generation.

Security & governance

Add authentication/authorization, ephemeral keys, encrypted storage, and an admin console for data retention policies.

UI/UX enhancements

Progressive indexing, chunk preview, adjustable chunk size, multi-lingual UI, answer confidence scores, and user feedback (thumbs up/down to collect labels for retraining).

Appendix

Deployed Application

Streamlit App: https://giki-prospectus-chatbot-cbhvkj2em2w7s9vgqqwfhs.streamlit.app

Source Code Repository

GitHub Repository: https://github.com/mudasiryasin/giki-prospectus-chatbot