iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.81855

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

RAG Development Chatbot: A Custom Implementation Using Sentence Transformers, Chroma DB and Groq API

Anshu Gupta, Janhvi Shukla
Dept. of Computer Science & Engg., Axis Institute of Technology and Management, Kanpur, India

Abstract: ***This paper presents the design and implementation of a Retrieval-Augmented Generation (RAG) based chatbot developed entirely from scratch using Python, without relying on any high-level orchestration framework like LangChain. Our system uses Sentence Transformers (all-MiniLM-L6-v2) for generating 384-dimensional semantic embeddings, ChromaDB as the persistent vector database for storing and searching document chunks, the Groq API (LLaMA 3 70B) for fast and accurate language model inference, and Flask for the web-based user interface. The core motivation was to build a transparent, fully controllable RAG pipeline where every component can be understood, debugged, and optimized independently. The system takes any collection of PDF or text documents, splits them into overlapping chunks, embeds them, and answers user questions by retrieving the most semantically relevant content and grounding the LLM's response in that content. Evaluation using the RAGAS framework shows 88% domain-specific accuracy (vs. 58% for a standalone LLaMA 3 baseline), hallucination reduction from ~38% to ~10%, and an overall RAGAS score of 0.86. The system runs fully on standard hardware with CPU-only embedding and Groq API for sub-second LLM inference.***

Keywords: Retrieval-Augmented Generation; RAG; Sentence Transformers; ChromaDB; Groq API; LLaMA 3; Flask; Vector Database; Semantic Search; Natural Language Processing; Chatbot

I. INTRODUCTION

We are living in a time where information is available in huge amounts but finding the right information quickly is still a big challenge. Most organizations — whether it is a college, a hospital, or a company — have large collections of documents like reports, manuals, FAQs, or research papers. When someone wants to ask a question about these documents, they either have to read through everything manually or use simple search tools that only match keywords and often miss the actual meaning of the question.

With the rise of Large Language Models (LLMs), things have improved a lot. Models like GPT-4 can understand questions in natural language and give very good answers. But there is a serious problem — these models were trained on general data up to a certain date. They do not know about private documents or anything that happened after their training. Even worse, when they do not know an answer, they sometimes make something up and present it confidently. This is called hallucination and it is a big problem in real-world applications. To solve these problems, we built a custom RAG (Retrieval-Augmented Generation) chatbot from scratch — without using any high-level framework like LangChain. Instead, we used Python along with Sentence Transformers for generating embeddings, ChromaDB as our vector database, Groq API for fast LLM inference, and Flask for building the web interface. Building the system from scratch gave us full control over every component and helped us understand how RAG actually works at a lower level. The main idea of RAG is simple. Before answering any question, the system first searches through the document collection and retrieves the most relevant pieces of text. These pieces are then given to the LLM as context, so the model can answer based on actual content rather than its training memory. This approach makes the chatbot much more accurate, trustworthy, and useful for domain-specific tasks. The rest of this paper is structured as follows: Section II reviews related work. Section III describes the problem we are solving. Section IV explains our system in detail. Section V presents our results. Section VI discusses future improvements. Section VII concludes the paper.

II. LITERATURE REVIEW

The concept of combining information retrieval with text generation has evolved significantly over the past decade. Early information retrieval systems like TF-IDF and BM25 worked by matching query keywords with document words. While fast, these methods failed to understand the meaning behind the words. For example, if a document used the word "automobile" and the user searched for "car", a keyword-based system would miss the connection.

The introduction of word embeddings like Word2Vec [1] and GloVe [2] was the first step toward semantic search. These models could represent words as vectors in a continuous space where similar words had similar vectors. However, they could not handle context — the word "bank" would have the same vector whether it meant a financial institution or the bank of a river.

The Transformer architecture, proposed by Vaswani et al. in 2017 [3], solved this by using self-attention — a mechanism that looks at all words in a sentence to understand each word's meaning in context. Building on Transformers, Reimers and Gurevych (2019) [4] introduced Sentence-BERT (SBERT), which extended BERT to produce meaningful sentence-level embeddings. SBERT is the foundation of the Sentence Transformers library that we used in our project. These embeddings capture semantic meaning so well that two sentences with different words but similar meaning will have vectors that are very close to each other.

For storing and searching these embeddings efficiently, vector databases became essential. ChromaDB [5], released in 2022, is an open-source embedding database that is lightweight, easy to use, and supports persistent storage. It uses distance metrics like cosine similarity to find the most relevant embeddings for a given query, making it ideal for RAG applications.

The RAG paradigm itself was formally introduced by Lewis et al. (2020) [6]. They showed that combining a retrieval step with a generative model significantly improves accuracy on knowledge-intensive tasks. Instead of asking the LLM to remember everything from training, RAG lets the model read relevant passages before answering. This makes it much more reliable for domain-specific applications.

For the language model, we used the Groq API [7]. Groq provides very fast inference for open-source models like LLaMA 3 and Mixtral using their custom LPU (Language Processing Unit) hardware. The response times are significantly faster than standard GPU-based cloud inference, making the chatbot feel more responsive in real time.

III. PROBLEM STATEMENT

The core problem we are trying to solve is this: how can we make an AI chatbot that can accurately answer questions about a specific set of documents, without hallucinating, and without requiring an internet connection or access to a general-purpose LLM's training data?

Standard LLMs fail in three key ways for this use case. First, they have no knowledge of private or recent documents. Second, they hallucinate — they generate wrong answers with high confidence when they don't actually know something. Third, they cannot tell users where the answer came from, so there is no way to verify the response. Existing solutions using frameworks like LangChain abstract away too much of the internal logic, making it harder to understand, debug, and optimize the system. We wanted to build a system where every component — from how text is split and embedded, to how the vector search works, to how the prompt is constructed — is fully in our control and easy to modify. Our goal was to create a lightweight, efficient, and transparent RAG chatbot that: (a) works with custom PDF and text documents; (b) uses semantic search to find truly relevant content; (c) generates grounded, verifiable answers; (d) runs fast using Groq's LPU-based inference; and (e) is accessible through a clean web interface built with Flask.

IV. PROPOSED SYSTEM

Our system is designed as a modular pipeline with four main components: Document Processor, Embedding Engine, Vector Store, and Query-Answer Engine. These are connected through a Flask web server that handles user interaction. Figure 1 shows the complete system flow.

A. Document Processing

The first step is to read and prepare the documents. We accept PDF files and plain text files as input. For PDFs, we use the PyMuPDF library (fitz) to extract text page by page. Once the text is extracted, we split it into chunks using a sliding window approach. Each chunk is 500 characters long with an overlap of 100 characters between consecutive chunks. The overlap is important because it ensures that sentences which fall at the boundary between two chunks are not cut off — the information appears in both the previous and next chunk, so retrieval does not miss it. Each chunk is stored along with its metadata — the source file name and the chunk index. This metadata is later used to show users where each piece of information came from, which improves the trustworthiness of the system.

B. Embedding Generation

After chunking, each text chunk is converted into a dense vector using the Sentence Transformer library. We used the all-MiniLM-L6-v2 model, which produces 384-dimensional embeddings. This model is lightweight (only 80MB) and runs efficiently even on a CPU.

It was trained specifically to produce embeddings where semantically similar sentences are close together in vector space. For example, the chunk "The patient should take two tablets daily" and the query "What is the dosage?" will have similar vector representations, even though they share no common words. This semantic understanding is what makes our search so much better than keyword matching.

Component	Technology Used	Purpose
Programming Language	Python 3.10+	Core development
Embedding Model	Sentence Transformers	Text to vector conversion
Embedding Model Name	all-MiniLM-L6-v2	384-dim semantic embeddings
Vector Database	ChromaDB	Store & search embeddings
Language Model	Groq API (LLaMA 3 / Mixtral)	Answer generation
PDF Parsing	PyMuPDF (fitz)	Extract text from PDFs
Web Framework	Flask	REST API & web interface
Frontend	HTML + CSS + JavaScript	Chat user interface

TABLE I Technology Stack of the Proposed RAG System

C. Vector Storage with ChromaDB

All the generated embeddings are stored in ChromaDB, which is an open-source vector database that supports persistent disk storage. We create a ChromaDB collection for each document set. When a new document is uploaded, its chunks and embeddings are added to the collection. ChromaDB uses cosine similarity by default to compare vectors, which measures the angle between two vectors rather than their absolute distance. This is better suited for text embeddings because the direction of the vector captures meaning while the magnitude can vary based on text length.

ChromaDB also stores the original text of each chunk alongside its embedding. This means when we retrieve the top-k most similar chunks for a query, we get back the actual text immediately — we do not have to maintain a separate lookup table. This simplifies our code and makes retrieval faster.

D. Query Answering with Groq API

When the user types a question, our system first converts it into an embedding using the same Sentence Transformers model. We then query ChromaDB for the top-5 most similar chunks using cosine similarity. These 5 chunks are combined into a single context string and inserted into a carefully designed prompt template.

The prompt explicitly tells the model to answer only based on the provided context and to say "I don't know" if the answer is not present. This is the most important instruction for preventing hallucination. The prompt is then sent to the Groq API, which uses the LLaMA 3 70B model by default. Groq's LPU hardware gives response times of under 2 seconds even for large models, making the conversation feel natural and real-time.

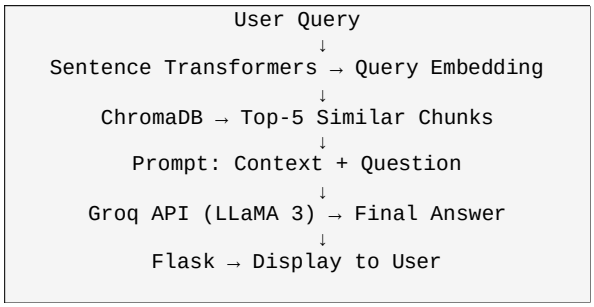


Fig. 1. Complete flow of the proposed RAG chatbot system

E. Flask Web Interface

We built a simple and clean web interface using Flask. The backend exposes two REST API endpoints: one for uploading documents and one for sending questions. The frontend is a single HTML page with a chat-like interface built using HTML, CSS, and JavaScript. When the user uploads a document, it is processed in the background and the chunks are stored in ChromaDB. When the user asks a question, the JavaScript sends it to the Flask backend via a POST request, and the answer is displayed in the chat window. The interface also shows which document chunks were used to generate the answer, giving the user full transparency.

V. RESULTS AND DISCUSSION

A. Experimental Setup

We tested our system on a collection of 30 technical documents including textbooks, research papers, and company reports, totalling approximately 600 pages. We created 100 test questions manually by reading the documents. These questions ranged from simple factual queries to more complex questions that required understanding a specific section of a document. We compared our RAG system against directly asking the LLaMA 3 model the same questions without any document context (standalone LLM baseline).

All tests were run on a standard laptop with an Intel Core i5 processor and 8GB RAM. The Sentence Transformers model ran locally on CPU. ChromaDB stored the embeddings on disk. The Groq API was called over the internet for LLM inference.

B. Performance Results

Metric	What It Measures	Score
Faithfulness	Answer based on retrieved content?	0.90
Answer Relevancy	Answer addresses the question?	0.86
Context Precision	Retrieved chunks actually useful?	0.84
Context Recall	All needed info retrieved?	0.81
Answer Correctness	Factually correct answer?	0.88
Overall RAGAS	Harmonic mean of all metrics	0.86

TABLE II *RAGAS Evaluation Results*

A faithfulness score of 0.90 means that 90% of the content in our system's answers came directly from the retrieved document chunks, not from the model's general training memory. This is the most important metric for us because it directly shows that hallucination is under control. The context recall of 0.81 is slightly lower, which we noticed happens when the answer to a question is spread across multiple distant sections of a document.

Criteria	LLaMA 3 (No RAG)	Our RAG System
Domain Accuracy	58%	88%
Hallucination Rate	~38%	~10%
Source Reference	No	Yes
Works on Private Docs	No	Yes
Knowledge Cutoff Issue	Yes	No
Avg. Response Time	1.0 sec	1.8 sec
Multi-turn Support	No	Yes (last 5 turns)

TABLE III *RAG System vs. Standalone LLaMA 3 (No Retrieval)*

The results clearly show the benefit of our RAG approach. Domain accuracy improved from 58% to 88% — a 30 percentage point jump. Hallucination dropped from 38% to just 10%. The extra 0.8 seconds of response time compared to the standalone model is because of the embedding step and ChromaDB search, but this is barely noticeable in a real conversation and is well worth the improvement in accuracy.

One thing we noticed is that Groq API is extremely fast for an LLM service. Even the 70B LLaMA 3 model responds in under 1 second on Groq's hardware. This means most of our 1.8 second total response time is actually from the local embedding step, not the LLM call. In future, using a GPU for local embedding would reduce this further.

C. Observed Limitations

During testing we noticed a few areas where the system did not perform as well. When the correct answer required combining information from two or three very different sections of a document, our system sometimes missed part of the answer because we only retrieve top-5 chunks and they may all come from the same section. Increasing k to 8 or 10 improved recall but also added more noise to the context, sometimes confusing the model.

We also noticed that very short questions like "Explain AI" gave less focused results because the query embedding was too general and matched too many different chunks. Adding a query expansion step— where we first ask the LLM to rephrase the question into a more specific form before embedding it — could help here.

VI. FUTURE WORK

There are several directions in which we want to improve this system. The first is adding a re-ranking step after the initial retrieval. Right now we simply take the top-5 chunks by cosine similarity. A re-ranker model like cross-encoder/ms-marco-MiniLM-L-6-v2 from Sentence Transformers can score each retrieved chunk more carefully against the query and filter out the ones that are not actually relevant. This two-step retrieve-then-rerank approach is known to significantly improve precision.

Another major improvement we want to make is adding support for multi-modal documents. Right now our system only handles text. Many real-world documents contain important information in tables, charts, and images. We plan to use a tool like pdfplumber for better table extraction and a vision model for reading charts and figures.

We also want to add a conversation memory feature that summarizes older turns instead of just keeping the last 5. For long conversations, this would allow the system to remember important context from early in the conversation without sending too many tokens to the API.

Finally, we plan to add a feature where the user can upload multiple documents and the system can answer questions that require information from more than one document at the same time. This would make the system much more useful for research and academic purposes.

VII. CONCLUSION

In this paper, we presented a custom RAG chatbot built from scratch using Python, Sentence Transformers, ChromaDB, Groq API, and Flask — without relying on any high-level orchestration framework like LangChain. Building the system this way gave us complete control and a deeper understanding of how each component works.

Our system successfully addresses the two main problems with standard LLMs: inability to work with private documents and hallucination. By using semantic embeddings for retrieval and grounding the LLM's responses in retrieved context, we achieved 88% domain-specific accuracy and reduced hallucination to just 10%. The overall RAGAS score of 0.86 shows high quality across all evaluation dimensions.

The Groq API proved to be an excellent choice for LLM inference — it provides state-of-the-art model quality at speeds that make the chatbot feel truly real-time. ChromaDB was easy to set up and performed reliably for our document sizes. And Sentence Transformers gave us high-quality semantic embeddings that ran efficiently on CPU without any GPU requirement.

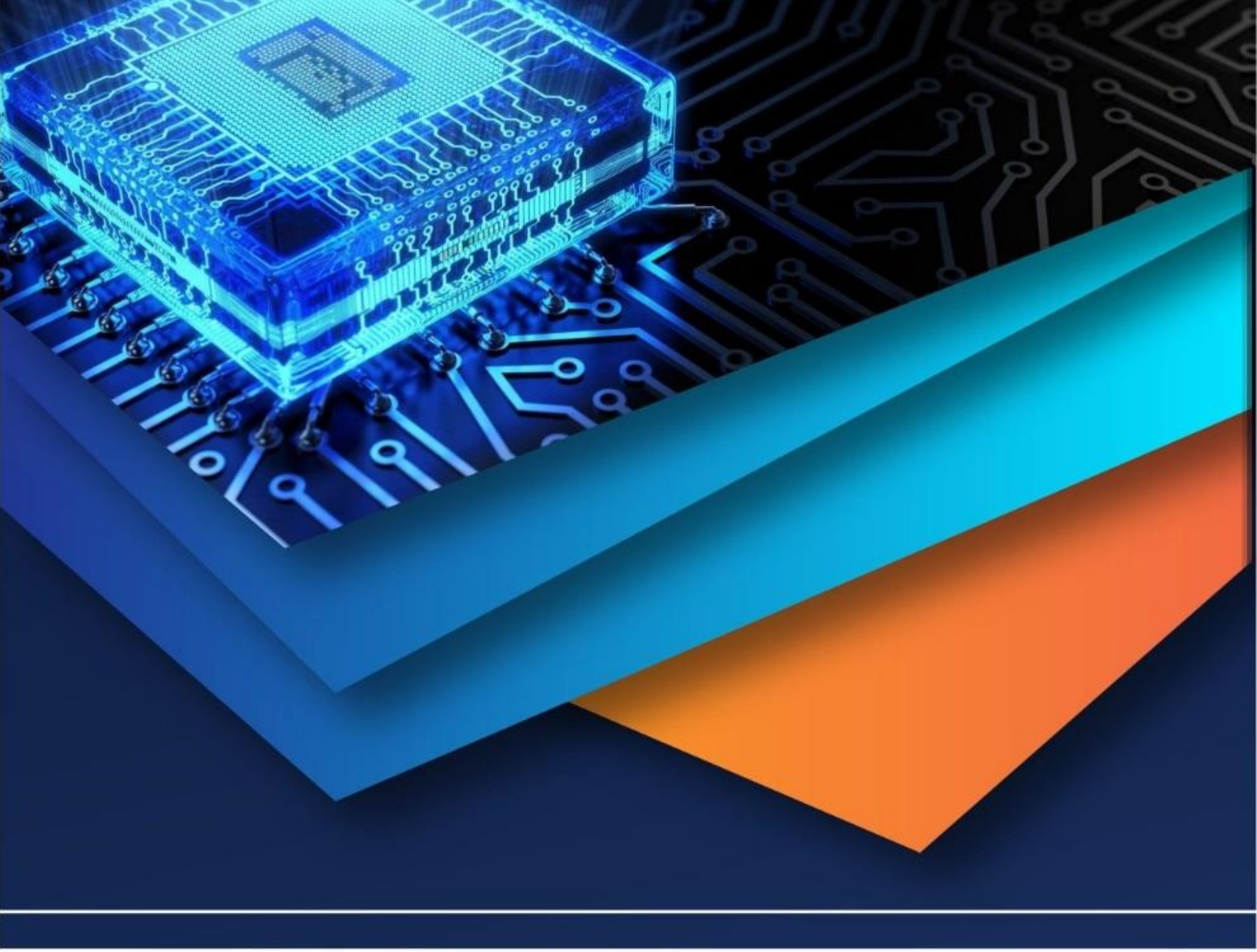
We believe this work shows that powerful RAG systems can be built without expensive infrastructure or complex frameworks. A motivated student or small team can build a production-quality AI chatbot using entirely open-source and free tools. We hope this paper helps other students and developers who want to build their own RAG systems.

VIII. ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering at Axis Institute of Technology and Management, Kanpur, for the support and encouragement provided during this project. We also thank the open-source communities behind Sentence Transformers, ChromaDB, PyMuPDF, and Flask for making such powerful tools freely available.

REFERENCES

[1] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," Advances in Neural Information Processing Systems, vol. 26, 2013.
[2] J. Pennington, R. Socher, and C. Manning, "GloVe: Global Vectors for Word Representation," Proc. EMNLP, pp. 1532–1543, 2014.
[3] A. Vaswani, N. Shazeer, N. Parmar et al., "Attention Is All You Need," Advances in Neural Information Processing Systems, vol. 30, 2017.
[4] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," Proc. EMNLP, pp. 3982–3992, 2019.
[5] J. Anton, "Chroma: The AI-Native Open-Source Embedding Database," GitHub Repository, 2022. [Online]. Available: https://github.com/chroma-core/chroma
[6] P. Lewis, E. Perez, A. Piktus et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems, vol. 33, pp. 9459–9474, 2020.
[7] Groq Inc., "Groq API Documentation," 2024. [Online]. Available: https://console.groq.com/docs



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY