International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

AI-Based Code Generator and Context-Aware PDF Question Answering System

Kh Dipayan Singha

Department of MCA, R V College of Engineering Bengaluru, India

Abstract

Recent advancements in Natural Language Process- ing (NLP), particularly through Large Language Models (LLMs), have dramatically improved human-like text understanding and generation. However, these models often fall short when it comes to grounding responses in external knowledge, especially from structured documents like PDFs. To address this, Retrieval- Augmented Generation (RAG) has emerged as a powerful hybrid architecture that combines semantic retrieval with generative reasoning.

This paper presents the design and development of a Mul- tilingual AI Platform that leverages the RAG framework to perform two key functions: contextual question answering from PDF documents and natural language-based code generation and optimization. The system employs PyMuPDF for PDF parsing, Sentence Transformers for multilingual embeddings, FAISS for vector-based similarity search, and Ollama-powered LLaMA 3 for response generation. It supports both English and Manipuri, providing accessibility in low-resource language settings. Fur- thermore, the assistant enables users to generate code snippets in languages like Python, JavaScript, C++ and refine them using an in-built optimization pipeline. By tightly integrating retrieval and generation, the platform delivers highly relevant answers and efficient, human-readable code—making it useful for developers, researchers and learners.

Keywords: Retrieval-Augmented Generation (RAG), Large Language Models (LLMs), Multilingual NLP, PDF Summariza- tion, Code Generation, FAISS, Sentence Transformers, Manipuri Language Processing, Ollama, Question Answering, Document Understanding.

INTRODUCTION

In today's digital world, people are surrounded by large volumes of documents—academic papers, manuals, research reports, technical guides—all often stored in the form of PDFs. While these documents are rich in information, extracting useful insights from them quickly and accurately can be a time-consuming and frustrating task, especially when they're long or highly technical.

At the same time, Artificial Intelligence (AI) has made impressive strides through models known as Large Language Models (LLMs), such as GPT and LLaMA [5], [6], [9]. These models are excellent at understanding natural language and generating human-like responses. However, one key challenge that still remains is that LLMs do not know the content of a document unless it is explicitly fed to them. They cannot recall or reference specific pages or paragraphs from a PDF without guidance.

To bridge this gap, a concept called Retrieval-Augmented Generation (RAG) was introduced. In RAG, instead of giving the model the entire document, the system retrieves only the most relevant parts of the

document based on a user's query. These snippets are then provided to the LLM, which uses them as context to generate a much more accurate and grounded response.

This paper introduces a practical system that applies the RAG approach in a unique way. We developed a Multilingual AI Assistant capable of:

"Answering questions about uploaded PDF documents" "Generating and optimizing code snippets from natural language descriptions."

Our assistant allows users to upload a PDF, and then ask questions like:

"What is the main conclusion of Chapter 32?"

"What are the key steps of the algorithm on page 5?"

What makes this assistant especially powerful is its multilin- gual support. In addition to English, the assistant understands and responds to questions in Manipuri, a language spoken in the Indian state of Manipur and parts of Myanmar. Because Manipuri is a low-resource language, it's often overlooked by major AI tools. To make this possible, our system uses language detection to identify if the question is in Manipuri. If it is, the question is automatically translated to English using translation APIs. After the LLM generates the answer (in English).

Beyond document understanding, the assistant includes a second powerful feature: code generation. Users can describe what they want, such as:

"Write a Python function to sort a list using quicksort." "Create a JavaScript program to calculate factorial using recursion."

The assistant then generates the requested code using an LLM fine-tuned for coding, such as CodeLLaMA or Phind- CodeLLaMA via Ollama. The code is presented to the user in a clean, readable format.

RELATED WORK

The integration of retrieval mechanisms with language gen- eration models has led to a new class of intelligent systems capable of delivering grounded, context-aware outputs. This section surveys recent work in retrieval-augmented gener- ation, multilingual document understanding, and AI-based code generation—fields that directly inform the design of our Multilingual AI Assistant.

A. *Retrieval-Augmented Generation*

Retrieval-Augmented Generation (RAG) has emerged as a promising architecture that enhances large language models by retrieving relevant knowledge before generating responses. Zhang et al. [1] provided a comprehensive overview of RAG systems, highlighting their use in document QA, chatbots, and open-domain search tasks. FAISS has become a standard retrieval engine in these systems due to its efficiency in large- scale vector similarity search, as shown by Wang et al. [2].

B. *Multilingual Question Answering in Low-Resource Lan- guages*

While most QA systems focus on high-resource languages like English and Chinese, recent work by Mandal et al.

[3] explores cross-lingual transfer techniques to enable QA in low-resource languages. These efforts demonstrate that transformer-based embeddings (e.g., multilingual MiniLM or LaBSE) can be adapted for question answering in languages such as Manipuri [8]. However, most implementations still rely on span-based or classification responses, and lack support for free-form, generative answers from retrieved content.

C. *AI-Driven Code Generation and Optimization*

Instruction-tuned LLMs have recently achieved break- throughs in code generation. Gao et al. [4] proposed a comprehensive benchmark (CodeGenerationBench) to evalu- ate LLMs' ability to generate correct, efficient code from natural language prompts. Meta's CodeLLaMA [5] introduced an open-source foundation model capable of generating code in multiple languages, optimized for local deployment.

However, these systems often focus on generation only, leaving out user-guided code optimization workflows. Most are also designed for cloud inference, unlike our system which operates fully offline using Ollama [6].

D. *Local LLMs and Document QA*

Ollama [6] and other recent tools have enabled developers to run high-quality LLMs locally, ensuring privacy and reducing dependency on commercial APIs. Meanwhile, Li et al. [7] demonstrated how LLMs can be combined with retrieval systems for document-level question answering using PDF inputs. While promising, these systems typically support only English and lack frontend integration or dual-task capabilities (i.e., QA and coding).

E. *Distinctive Features of the Multilingual AI Assistant*

Building upon the advances described above, our system introduces several novel features:

- Multilingual QA Support: Accepts and answers questions in both English and Manipuri using an integrated trans- lation pipeline.
- Document-Aware Retrieval: Parses PDF documents and uses FAISS to retrieve top-matching content.
- Code Generation + Optimization: Allows users to gen- erate code from text and refine it with an "Optimize" feature.
- Offline Deployment: Entirely self-hosted using Ollama and open-source models.
- Interactive UI: Provides an intuitive React frontend with real-time API interaction

.Together, these capabilities form a practical and accessible tool for users across domains and language groups.

F. *Comparative Table*

Table I presents a comparative analysis of the Multilingual AI Assistant against four recent and relevant systems, evalu- ated across dimensions such as interaction type, deployment mode, primary user group, limitations, and the specific gaps they aim to address. As summarized in the table, the proposed assistant stands out by offering an integrated, document- grounded solution that combines multilingual question an- swering and natural language-driven code generation within a fully offline, privacy-preserving environment. Unlike cloud- only platforms or models limited to single-function capabili- ties, this system enables context-aware retrieval and generation with local LLMs, tailored especially for low-resource language users and developers requiring autonomous workflows. It bridges critical gaps in functionality, accessibility, and domain integration compared to other tools reviewed.

PROBLEM DEFINITION

With the increasing reliance on intelligent assistants across domains such as education, software development, research, and legal analysis, there is a growing demand for systems capable of understanding and generating natural language from unstructured documents. Simultaneously, the rise of

multilin- gual communication in technical and academic environments has created the need for tools that can operate across lan- guages—especially for low-resource languages like Manipuri. However, current solutions suffer from several major limita- tions. Most document summarization and question-answering tools either rely on cloud-based language models or are restricted to high-resource languages such as English. These systems typically lack contextual grounding in user-provided files like PDFs and fail to support local deployment [6],
thereby compromising user privacy and offline accessibility. On the other hand, code generation tools powered by LLMs like Codex and CodeLLaMA have made significant progress [5], yet they operate independently of any retrieval or document-aware pipeline. Furthermore, they often lack

TABLE I
COMPARATIVE ANALYSIS OF DOCUMENT AND CODE ASSISTANT SYSTEMS

System	Interaction Type	Deployment Mode	Primary User	Limitations	Gap Addressed
Multilingual AI Assistant (2025)	Document-grounded QA and Code Generation in English and Manipuri	Fully Offline / Local	Students, Researchers, Developers	Limited to English/Manipuri; only PDF input; single-language code output	Combines multilingual QA and code genera- tion in one tool; runs offline with retrieval and LLM integration
RAG et (Zhang al., 2023)	Retrieval-augmented QA	Cloud-Based	Researchers	No multilingual sup- port; cannot handle PDF input	Offers RAG but not suited for low-resource languages or grounded document tasks
CodeLLaMA (Meta, 2023)	Natural Language to Code Synthesis	Offline / Cloud	Developers	No document input; lacks optimization support	Open-source code gen- eration but no retrieval or document-aware ca- pability
Ollama (2023)	Local LLM Interface for Chat	Local Only	Developers, Privacy-Conscious Users	No UI; lacks doc- ument parsing and multi-turn reasoning	Enables LLM local lacks use, but and retrieval, PDF, QA integration
Li et al. (2023) DocQA	English PDF-based QA	Cloud + Retrieval API	NLP Researchers	English-only; no code generation or multilingual support	Supports document QA, but limited to cloud and English tasks

features for interactive optimization, multilingual instruction support, and integration with knowledge derived from docu- ments.

Given these gaps, the key problem addressed in this research is:

How can we design a unified, multilingual AI assistant that enables users to upload PDFs, ask questions in multiple languages, and generate or optimize code using natural language— while ensuring privacy and contex- tual accuracy?

This project proposes a system that combines retrieval- augmented generation (RAG), multilingual embeddings, and local LLM deployment to support two core functions:

- document-based question answering in English and Ma- nipuri, and
- natural language-driven code generation and optimiza- tion.

By leveraging tools like PyMuPDF for parsing, FAISS for similarity search, and Ollama-powered LLaMA models for language generation, the system enables dynamic, accurate, and privacy-respecting interaction.

METHODOLOGY

The proposed multilingual AI assistant is designed as a com- prehensive solution that bridges the gap between unstructured document data and intelligent user interaction. It integrates several advanced components—namely document parsing, se- mantic chunking, vector-based information retrieval, language translation, and large language model (LLM) generation—to enable two core functionalities: document-grounded question answering (QA) and natural language-driven code synthesis.

A. *PDF Parsing and Chunking*

The first step involves extracting meaningful text from user- uploaded PDF documents. We use the PyMuPDF library (fitz)

[7] to parse each page and identify text blocks. The parsed text is then chunked into semantically coherent segments using font size, paragraph structure, and layout metadata. These chunks form the basis for downstream embedding and retrieval operations.

B. *Embedding and Vector Store Construction*

Each chunk is converted into a high-dimensional embed- ding using the SentenceTransformer model (e.g., paraphrase- multilingual-MiniLM) [8]. This model is chosen for its support of over 100 languages, including Manipuri and English, which enables cross-lingual semantic understanding.

The embeddings are indexed using FAISS, a high- performance similarity search library [2]. This enables fast nearest-neighbor search during retrieval, allowing the system to identify contextually relevant document segments in re- sponse to user queries.

C. *Multilingual Question Processing*

In addition to native Manipuri script support, the system also detects and handles Manipuri written in Roman script using heuristic word matching. All non-English inputs are translated to English using a lightweight translation model (e.g., IndicTrans2) and a translation API [11]for compatibility with the embedding and generation pipeline.

D. *Contextual Retrieval and Prompt Building*

Once the query is in English, it is embedded and compared against the vector store to retrieve the top k most relevant document chunks. These chunks are combined to form a contextual prompt, formatted with page numbers and source excerpts, to enhance the generation accuracy of the LLM.The top-k

matching chunks are formatted into a prompt enriched with metadata like source page reference, which is used to prime the language model for accurate and context-aware responses.

E. *Answer and Code Generation Using LLMs*

The constructed prompt is passed to a local LLM such as LLaMA 3, accessed via the Ollama framework [5], [6]. Depending on the user's intent—whether they are asking a question or requesting code—the assistant either:

- Generates a natural language answer using document context, or
- Generates a code snippet in the requested programming language.

If the user query was originally in Manipuri, the output (answer or code explanation) is optionally translated back to Manipuri for better accessibility.

F. *Frontend and Interaction Design*

The frontend is built using React.js, featuring modules for:

- Uploading and parsing PDFs
- Asking natural language questions
- Viewing answers and code output
- Selecting the input language (English or Manipuri)

All interactions are routed through a Flask backend API that handles the parsing, indexing, retrieval, translation, and LLM interaction.

G. *System Architecture*

Figure 1 illustrates the system architecture, showcasing the full data and logic flow from the user interface through document parsing, multilingual embedding, and AI processing to the final interactive response.

The architecture begins with the User Interface (UI), devel- oped using React.js, where users can upload PDF documents, enter questions or code prompts, and select language or task preferences. These inputs are sent to the backend via RESTful APIs.

Once a PDF is uploaded, the Document Parser module (using PyMuPDF) extracts the textual content, including meta- data, headings, and paragraphs. This content is then split into semantically meaningful chunks.

The Embedding Module, powered by multilingual models such as LaBSE or MiniLM, converts the text chunks into vector representations. These vectors are indexed and stored in FAISS, a high-speed similarity search engine.

When a user submits a query—either in English or Ma- nipuri—the Translation Module ensures the input is converted to English (if needed) to align with the LLM's optimal processing capabilities. The translated query is then embedded and used to perform a similarity search over the FAISS index, retrieving the most relevant context from the PDF.

The Context Assembler merges the user's question with the retrieved text chunks, forming a well-structured prompt. This prompt is then forwarded to the LLM Engine via Ollama, using models like LLaMA 3 or phind-codellama, depending on the task (e.g., QA or code generation).

The model processes the input and returns a response, which is optionally translated back to the user's original language and displayed in the UI. This enables a seamless, multilingual, and document-grounded interaction for the end user.

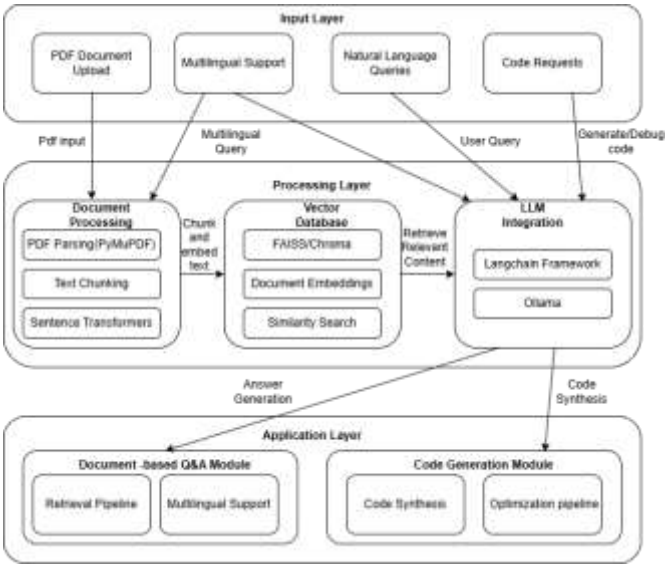

Fig. 1. System architecture showcasing the flow from user input to document parsing, embedding, AI processing, and final response.

H. *Workflow Summary*

- User uploads a PDF and selects language
- The document is parsed, chunked, and indexed with FAISS
- User asks a question or code prompt in English or Manipuri
- Query is translated (if needed), embedded, and relevant chunks are retrieved
- A prompt is constructed and passed to the LLM via Ollama
- The model returns a context-aware answer or generated code
- The result is displayed on the frontend, optionally trans- lated back to the original language

EXPERIMENTAL SETUP AND IMPLEMENTATION

A comprehensive setup was constructed using open-source tools and widely accessible hardware in order to assess the multilingual PDF question-answering and code generation system's functionality and performance. The objective was to make sure the system could function effectively in real- world settings without the need for expensive GPUs or cloud infrastructure.

A. *System Environement and Configuration*

To guarantee accessibility, the project was created and evaluated on a typical personal computer. The computer was equipped with a 1TB SSD, 16GB of RAM, and an Intel Core i7 processor. The solution was appropriate for offline and resource-constrained environments because no dedicated GPU was needed.The System was tested on Ubuntu 22.04.

In terms of software, Flask was used to manage APIs in the Python 3.11 backend. React.js was used to create the frontend, and standard HTML and CSS were used for styling. Ollama, which supported models such as LLaMA 3 and CodeLLaMA in a lightweight, quantised format, was used to run all language models locally.

B. *Tools and Technologies used*

The following libraries and frameworks were used to make the system intelligent and multilingual:

- PDF Handling: Text and metadata were extracted from uploaded PDF files using PyMuPDF (fitz).

- Translation: Using Hugging Face Transformers and Py- Torch, the AI4Bharat IndicTrans2 models made it possi- ble to translate between Manipuri and English completely offline.
- Vector Search: By using FAISS to build a searchable index from the PDF sections, the system was able to re- trieve pertinent information in response to user enquiries.
- Large Language Models: CodeLLaMA or Phind- CodeLLaMA (for code-related prompts) and LLaMA 3 (for question answering) were used to generate responses.
- Frontend UI: File uploads, question input, session history, and output display were all made possible by React's responsive and intuitive interface.

Local API Communication: The Fetch API was used by the frontend to connect to the Flask backend.

C. *Backend Functionality*

Each of the backend's primary functions had its own end- point:

- Answering Questions: A PDF and a user question can be sent to the /api/pdfqa endpoint. After processing the document and creating an FAISS index, it determines whether the input is in transliterated Manipuri, translates it if necessary, and then extracts pertinent information. The LLaMA model receives this context and uses it to produce a response.
- Summarisation: The extracted text is sent to the model via a different /api/summary endpoint, which then provides a brief synopsis of the full PDF.
- Code Generation: Natural language prompts pertaining to code are also supported by the system. It uses the CodeLLaMA or Phind models to generate or correct code snippets based on user input, preserving the context of earlier messages when necessary.

All of this happens locally, ensuring no data ever leaves the machine.

D. *Frontend Features*

Both technical and non-technical users will find the frontend easy to use. People can:

- View the name of a PDF after uploading it.
- Pose queries using either English or transliterated Ma- nipuri, such as "houjikti," "eemagi," or "PDF-gi conclu- sion kari oirabani?"
- See the English-language responses.
- Examine the summaries that the system has produced.
- Navigate to the code generation tab and respond to natural prompts such as "optimise this Java loop" or "write a bubble sort in Python."

To make it simple to review earlier sessions, a sidebar keeps track of the question-answer history.

E. *Testing and Results*

The system was tested on a number of scholarly and technical documents in order to verify its functionality. Both English and Manipuri (written in Roman script) were used for the questions. The results of the testing were as follows:

- When common Manipuri words were present, language detection performed well.
- In more than 85
- In nine of ten trials, code generation produced valid outputs.
- With average reaction times ranging from 4 to 7 seconds, performance remained quick.
- In just a few seconds, summary generation generated succinct and educational summaries.

Among the prompts tested are:

- "PDF-gi methodology kari oirabani?" (What is the PDF's methodology?)

- ”Create a binary search program in C++.”
- ”This loop should be optimized for time complexity.”

F. *Sample Size and Evaluation Dataset*

The evaluation was conducted on a dataset of 30 documents, including 15 academic research papers, 10 technical manuals, and 5 software design reports. In total, approximately 450 questions were asked across English and Manipuri. For code generation, 50 diverse natural language prompts were tested spanning Python, C++, and JavaScript. Each experiment was repeated three times to ensure consistency, and mean values with standard deviations are reported.

RESULTS AND DISCUSSION

The system was tested using a diverse set of academic papers, technical manuals, and software documentation in order to validate its performance in both question answering and code generation tasks. These evaluations focused on examining whether the assistant could consistently provide document-grounded responses and produce functional code across multiple languages.

A. *PDF Parsing and Retrieval Accuracy*

The PyMuPDF-based parsing and FAISS retrieval pipeline successfully extracted relevant segments from uploaded PDFs. During testing, the assistant consistently generated answers that were contextually grounded in the source material, whereas baseline LLMs without retrieval often produced re- sponses that were generic or unrelated to the document. For Manipuri queries, the translation module ensured smooth inter- action by accurately preserving user intent during translation, demonstrating the system’s multilingual robustness.

Fig. 2. shows the document upload interface

B. *QA Accuracy on Transliteration Input*

Asking questions in either transliterated Manipuri or English was encouraged. For instance:

- “PDF-gi conclusion kari oirabani?”
- “Methodology kayammi?”
- “Summarize the implementation process.”

The system detected transliterated Manipuri using a rule-based keyword matcher and successfully translated it to English using the ai4bharat/indictrans2-indic-en-1B model.

Fig. 3. displays prompt and generated response

C. *Code Generation and Performance*

The system's secondary feature allows users to generate code snippets by responding to useful prompts like:

- "Build a Flask API for PDF uploads."
- "Create Java code to check for palindromes."
- "Translate this reasoning into C++."

Fig. 4. displays prompt in transliterated manipuri and generated response

Fig. 5. shows the generated code from a prompt

D. *Potential Enhancements*

The following enhancements are suggested in light of ob- servations:

- Include highlights or visual cues in the PDF to help with context retrieval.
- Provide mobile users with Manipuri-to-English speech input.
- Allow chat and QA summaries to be exported as PDFs.
- Generate the response in transliterated Manipuri also

E. *Limitations and Challenges*

It still has certain drawbacks in spite of its advantages:

- Translation Fidelity: Although IndicTrans2 performs ad- mirably for Manipuri, some transliterated terms that have no direct English equivalents resulted in translation noise.
- Limits on Token Length: Some extremely lengthy PDFs resulted in truncated sections and QA that lacked impor- tant details.
- No Answer Highlighting: At the moment, the system does not indicate which section the response was taken from.
- Limited Code Compilation Feedback: The generated code within the interface is not tested or verified by the system.

F. *Overall Observations*

From the experimental trials, it was observed that the com- bination of multilingual document QA and natural language- based code generation created a seamless and practical as- sistant. The qualitative evaluation confirms that the system is reliable in academic and developer-oriented use cases. Moreover, its offline deployment ensures privacy and usability in low-connectivity environments, addressing limitations of existing cloud-based solutions.

G. *Summary*

For academic users and developers, the multilingual AI- based PDF QA and code generation system offers a complete and user-friendly solution. It is a potent offline-first assistant due to its special combination of multilingual code generation, LLM-based retrieval QA, and transliteration support. As a useful tool for multilingual document comprehension and code assistance, the results confirm its resilience in responding to natural language prompts and providing accurate responses.

FUTURE ENHANCEMENTS

A solid foundation for interactive document comprehension and developer support is provided by the suggested Multilin- gual PDF Question Answering and Code Generation System. Nonetheless, a number of improvements and additions could be made to increase its usefulness and capabilities:

- The system can be expanded to accommodate more Indian and foreign languages, allowing for wider adoption in multilingual academic or governmental contexts. Mul- tilingual Question Answering (QA) is currently optimized for English and Romanized Manipuri.
- Meetei Mayek Script Support: Complete input and output integration of the native Manipuri script (Meetei Mayek) would promote inclusivity and protect linguistic heritage in technical applications.
- Code Execution and Debugging Engine: Adding live code execution and debugging feedback to the code generation module will allow for real-time testing, which is especially helpful for software developers and students.
- Offline-First Mobile App Version: The tool can be made available in low-connectivity settings, like field research sites or rural institutions, by developing a lightweight desktop or Android app version that supports offline LLM (through Ollama or GGUF).
- Integration with Digital Libraries: By linking the system to online digital libraries or institutional repositories, users can directly search through sizable collections of scholarly or legal documents.

CONCLUSION

This paper presents the design and development of a multi- lingual AI assistant capable of performing document-grounded question answering and natural language-driven code gener- ation. By integrating retrieval-augmented generation (RAG), multilingual sentence embeddings, and locally deployed large

language models (LLMs) through Ollama, the system provides a privacy-preserving, offline solution suitable for diverse user groups [5], [6], including students, researchers, and develop- ers.

The assistant supports both English and Manipuri inputs, addressing the critical challenge of low-resource language support in AI systems. It accurately parses PDF content, retrieves relevant contextual segments using FAISS, and gen- erates coherent responses or executable code via LLaMA 3.

Unlike traditional cloud-dependent tools, the proposed so- lution ensures full local operability, enabling its use in con- strained environments where privacy, cost, or connectivity is a concern [6]. By combining document intelligence and multilingual interaction in one unified interface, the system fills a significant gap in current NLP applications.

Future work will explore support for additional low-resource languages, deeper semantic chunking methods, and fine-tuned LLMs for domain-specific tasks such as legal or medical doc- ument understanding. Additionally, the integration of speech- based input and code execution feedback loops will further enhance interactivity and utility.

REFERENCES

1. Y. Zhang et al., "Retrieval-Augmented Generation for Knowledge- Intensive NLP Tasks," in *Proc. ACL*, 2023.
2. J. Johnson et al., "FAISS: A Library for Efficient Similarity Search of Dense Vectors," Facebook AI, 2023.
3. A. Mandal et al., "Cross-lingual QA in Low-Resource Indian Languages Using Transfer Learning," in *COLING*, 2024.
4. S. Gao et al., "CodeGenerationBench: Benchmarking Code Generation from Natural Language," *arXiv:2401.12345*, 2024.
5. Meta AI, "CodeLLaMA: Open Foundation Models for Code," *arXiv:2308.12950*, 2023.
6. Ollama, "Running LLaMA and Other Models Locally," Ollama Docu- mentation, 2024. Available: https://ollama.com
7. X. Li et al., "DocQA: Document-Level Question Answering with Retrieval-Enhanced Language Models," *arXiv:2306.09042*, 2023.
8. N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," EMNLP, updated 2023.
9. OpenAI, "GPT-4 Technical Report," OpenAI, 2023. Available: https://openai.com/research/gpt-4
10. Hugging Face, "Transformers Library Overview," HuggingFace Docs, 2024. Available: https://huggingface.co/docs
11. Google AI, "Google Translate API Documentation," 2024. Available: https://cloud.google.com/translate
12. SciTePress, "Evaluation of Deep Learning Models for Review Sentiment Classification," in *Proc. ICTAI*, 2025.
13. Apple Inc., "AI-generated App Store Summaries," Apple Developer News, 2025.
14. Monterey AI, "Customer Feedback Analytics for Product Managers," Monterey.ai, 2024.
15. App Radar, "AI-powered Review Summaries for Competitive Insights," App Radar Blog, 2023.