RV College of Engineering®

Mysore Road, RV Vidyaniketan Post, Bengaluru - 560059, Karnataka, India

AI BASED CODE GENERATOR AND CONTEXT AWARE PDF QUESTION-ANSWERING SYSTEM

MAJOR PROJECT REPORT
MCA491P

submitted by

Kh Dipayan Singha 1RV23MC045

under the guidance of

Dr. Usha J
Professor
Department of MCA, RVCE,
Bangalore - 560059

in partial fulfilment for the award of degree of

Master of Computer Applications

Department of Master of Computer Applications
2024–2025

RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

www.rvce.edu.in
Tel: +91-80-68188100
+91-80-68188111
+91-80-68188112

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

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

MAJOR PROJECT REPORT

MCA491P

submitted by

KH DIPAYAN SINGHA 1RV23MC045

under the guidance of

DR. USHA J
(Internal Guide)
Professor
Department of MCA,RVCE

in partial fulfilment for the award of degree of

MASTER OF COMPUTER APPLICATIONS

2023-2024

RASHTREEYA SIKSHANA SAMITHI TRUST
RV
INSTITUTIONS

RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,Bengaluru - 560059, Karnataka, India

CERTIFICATE

Certified that the Major Project titled 'AI Based Code Generator and Context Aware PDF Question-Answering System' is carried out by Kh Dipayan Singha (1RV23MC045) a bonafide student of RV College of Engineering®, Bengaluru, in partial fulfillment for the award of Degree of **Master of Computer Applications** of **Visvesvaraya Technological University, Belagavi** during the year **2024-2025**. It is certified that all corrections/suggestions indicated for the internal assessment have been incorporated in the report deposited in the department library. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed by the institution for the said Degree.

Usha-J

Internal Guide Dr. Usha J
Professor
Department of MCA
RV College of Engineering

[illegible]

Head of the Department Dr. Jasmine K S
Associate Professor & Director
Department of MCA,
RV College of Engineering

[illegible]

Principal
Dr. K N Subramanya
RV College of Engineering

External Viva Examination

Name of Examiners	**Signature with Date**
1.	
2.	

RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,Bengaluru - 560059, Karnataka, India

DECLARATION

I Kh Dipayan Singha, the student of Fourth semester Department of MCA, RV College of Engineering®, Bengaluru-560059, bearing USN: 1RV23MC045 hereby declare that the project titled 'AI Based Code Generator and Context Aware PDF Question-Answering System' has been carried out by me. It has been submitted in partial fulfilment of the program requirements for the award of Degree in **Master of Computer Applications** of **Visvesvaraya Technological University, Belagavi** during the year **2024-2025**.

Further, I declare that the content of the dissertation has not been submitted previously by anybody for the award of any Degree or Diploma to any other University.

I also declare that any Intellectual property rights generated out of this project carried out at RVCE will be the property of RV College of Engineering®, Bengaluru and I will be among the authors of the same.

Place: Bangalore

Date of Submission: 13/8/25

[illegible]

Signature of the Student

Student Name:Kh Dipayan Singha

USN: 1RV23MC045

Department of Master of Computer Applications

RV College of Engineering®

Bengaluru - 560059

Acknowledgement

I take this opportunity to express my sincere gratitude to all those who have contributed to the successful completion of this project.

First and foremost, I would like to express my deep respect and heartfelt thanks to my project guide, **Dr. Usha J**, for their invaluable support, insightful guidance, and constant encouragement throughout the project. Their expert suggestions and constructive feedback helped me improve and complete this work successfully.

I would also like to thank the **Dr. Jasmine K S**, Director of MCA at RV College of Engineering and all faculty members of the **Department of Master of Computer Applications** of **Visvesvaraya Technological University, Belagavi** for their support, encouragement, and for providing the necessary facilities and a positive learning environment.

My sincere thanks to my classmates and friends who supported me directly or indirectly during various stages of the project. Their cooperation, encouragement, and valuable inputs were instrumental in shaping the project effectively.

Lastly, I extend my gratitude to my family for their patience, motivation, and unwavering support throughout my academic journey.

Thank you all.

Kh Dipayan Singha
1RV23MC045

Abstract

In today’s AI-driven era, developers and researchers frequently encounter challenges in understanding large volumes of technical documents and generating optimized code efficiently. To address these issues, this project introduces a **multilingual AI-powered platform** that combines **document-based question answering (QA)** with **intelligent code generation**. Leveraging **Natural Language Processing (NLP)**, **vector-based retrieval (FAISS/Chroma)**, and **Large Language Models (LLMs)** like **LLaMA 3 via Ollama**, the system enables seamless interaction with PDFs and facilitates on-demand code synthesis.

The platform integrates two core modules: the **Context-Aware PDF QA Module**, which allows users to upload academic or technical PDFs, ask natural language questions in English or transliterated Manipuri, and receive precise, context-driven answers; and the **AI Code Generator Module**, which produces accurate and optimized code snippets across multiple programming languages based on user prompts. Supporting features include **multilingual translation pipelines (IndicTrans2)**, **semantic chunking for document understanding**, **session-based history tracking**, and **real-time summarization** of uploaded content.

Technologically, the solution employs **React.js** for the frontend, ensuring a responsive and intuitive interface, and **Flask API** for the backend, enabling high-performance APIs to handle document parsing, embedding generation, and LLM integration. Its modular design ensures scalability, offline deployment, and easy future enhancements like code debugging, visual analytics, and multi-document support.

By bridging the gap between unstructured document data and real-time code assistance, this platform serves as a **comprehensive AI assistant** for students, developers, and researchers. It simplifies document comprehension, enhances coding efficiency, and promotes accessibility for multilingual users, thereby offering a practical solution to modern-day challenges in learning, research, and software development.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Project Description

The rapid pace of development in artificial intelligence (AI) and natural language processing (NLP) has rewritten the rules for the manner in which humans connect with machines. In the modern digital environment, enormous amounts of data are housed in unstructured presentation modes, one of the most popular being Portable Document Format (PDF). These documents usually carry key information like research results, guides, and scholarly content, but extracting pertinent knowledge from them in an efficient manner is a significant issue. At the same time, there is growing demand for intelligent code generation systems that allow developers as well as non-programmers to translate natural language commands into executable code with minimal human effort. Spanning these two fields, this project — Context-Aware PDF Question Answering System and AI-Based Code Generator — seeks to create a unified platform that encompasses both document-based question answering and automatic code generation in a single integrated setting. Utilizing Retrieval-Augmented Generation (RAG), large language models (LLMs), and multilingual support pipelines, the system not only makes information more accessible but also opens up coding support to users with different technical competencies.

The project description is broken down into two main components. The first is the context-aware PDF question answering module, in which users upload PDFs and ask questions in either English or Manipuri (transliterated). The system translates the PDF, retrieves relevant content through embedding-based similarity search, and returns accurate answers based on the content of the document. This feature is especially useful in academic, legal, and technical fields where information lookup needs to be accurate and context-based. The second feature is the AI-driven code generation and debugging module, which takes natural language inputs specifying coding issues or target functionalities. The system then generates code snippets or fixes current code by inferring over the specifications, with syntactic correctness and optimization. All these modules together form an end-to-end tool that helps students and professionals with learning, research, and software development activities.

The real-world applicability of the system applies to various fields. In academic settings, students may upload lecture notes or research documents and immediately ask key concepts, methods, or conclusions instead of scanning long documents manually. Researchers and developers can explain functionality in simple terms and receive working code snippets in Python, Java, or C++. Even adding Manipuri transliteration serves regional language accessibility, making AI-based tools more inclusive. Through ease of use, contextual precision, and multilinguality, the project illustrates how AI can be used to enable users to bridge the gaps both in knowledge acquisition and programming.

Objectives of the project

The project aims to develop a Context-Aware PDF Question Answering (QA) System. This system will allow users to query uploaded PDFs and get accurate, context-rich answers. It also plans to add an AI-based Code Generation and Debugging feature that can produce executable code or fix existing snippets from natural language prompts. To make it accessible to everyone, the system will include support for multiple languages, focusing on English and Manipuri transliteration for bilingual input.

Using a Retrieval-Augmented Generation (RAG) setup with FAISS-based retrieval and LLaMA/CodeLlama models, the project will provide precise and relevant outputs. All features will be combined into a single, user-friendly web interface designed for students, researchers, and developers.

Scope of the Project

This system has various uses in different areas. For academic purposes, it allows users to quickly find summaries and explanations from lecture notes and research papers. In software development, it helps programmers by creating code templates and fixing code in several languages. It also supports Manipuri transliteration in addition to English, making it easier to overcome language barriers. In research, it extracts relevant insights from large datasets or technical documents. Its flexible design lets users easily add more languages, models, and formats, making it ready for future needs.

1.2 Theory and Concept Relevant to the Project

This section describes the basic theories, concepts, and technologies, such as RAG, LLMs, multilingual processing, and information retrieval. These elements form the foundation for the project's design and implementation.

- **Natural Language Processing (NLP)**: Techniques for understanding, processing, and generating human language, forming the backbone of question answering and translation modules
- **Large Language Models (LLMs)**:Transformer-based models (e.g., LLaMA 3, Mistral) trained on massive datasets to perform reasoning, summarization, and code generation

- **Vector Embeddings and Semantic Search:**Converting text into high-dimensional vectors (using SentenceTransformers) and retrieving semantically similar chunks with FAISS for context-aware answers
- **Multilingual Translation (IndicTrans2):**Enabling queries in Manipuri (transliterated) and translating them into English for processing, supporting cross-lingual QA
- **PDF Parsing and Chunking (PyMuPDF):**Extracting structured text, metadata, and splitting into meaningful chunks based on font and layout for efficient retrieval
- **Question Answering Systems (RAG):**Retrieval-Augmented Generation combines semantic search with generative LLMs to deliver accurate context-grounded answers
- **Generative AI for Code Synthesis:**LLMs generate code snippets from natural language prompts, aiding developers and students in understanding and learning
- **Offline AI Deployment (Ollama):**Local execution of models ensures privacy, reduced latency, and independence from external APIs
- **User Interface and Integration:**Web-based frontend using React and Flask backend for real-time interaction and smooth user experience

1.3 Relevance and Motivation

In the present-day digital scenario, many academic documents, technical guides, and codebases are available in unstructured forms such as PDFs. It is difficult to draw useful information or answers from these pieces of content, particularly when it comes to answering questions posed in different languages. The project fills this gap by integrating Retrieval-Augmented Generation (RAG) with translation functions to handle both English and Manipuri (transliterated) input. It allows users to search documents and even create code snippets from the context itself, saving a lot of time and effort. The solution is especially helpful for developers, researchers, and students, and its local installation through open-source LLMs guarantees data privacy and makes it feasible in academic or offline settings.

1.4 Dissertation Organization

The rest of this dissertation is organized into the following chapters:

1. **Chapter 2: Literature Survey** — Reviews existing tools, methods, and research in code generation, debugging, and PDF question answering systems, and identifies the gaps this project aims to address.

2. **Chapter 3: Software Requirement Specification** — Defines the functional and non-functional requirements, constraints, and technical specifications necessary for implementing the unified AI system.

3. **Chapter 4: System Design and Implementation** — Presents the architecture of the system, detailed descriptions of its modules, workflow diagrams, and the implementation strategies adopted for both code generation and PDF QA modules.

4. **Chapter 5: Testing and Results** — Explains the testing methodologies, showcases test cases with their outcomes, and evaluates the system's performance in terms of accuracy, usability, and multilingual support.

5. **Chapter 6: Conclusion and Future Work** — Summarizes the outcomes of the project, highlights its contributions to AI-assisted development and document analysis, and proposes possible enhancements for future iterations.

Chapter 2

Literature Review

2.1 Literature Survey

A literature review was done to investigate progress made in code generation, debugging, multilingual question answering, and document retrieval through large language models (LLMs) and Retrieval-Augmented Generation (RAG). The results from 20 major references are encapsulated as follows:

P. Lewis (2023) introduced the Retrieval-Augmented Generation (RAG) framework for knowledge-intensive NLP tasks, integrating a neural retriever with a sequence-to-sequence generator. This hybrid method retrieves relevant documents from an external corpus during inference and incorporates them into the generation process, resulting in outputs that are more factually accurate and contextually grounded. The work demonstrated that RAG reduces hallucinations, improves answer reliability, and is adaptable to tasks such as open-domain question answering, summarization, and fact verification.

Y. Zhang (2023) extended the RAG framework to a variety of domains, highlighting its scalability and flexibility. The study showed how RAG can handle long-document question answering, multi-hop reasoning, and cross-domain information retrieval by adapting retrieval strategies to different data types. Zhang's experiments validated that RAG can maintain high accuracy even when processing large, heterogeneous datasets.

J. Johnson (2023) developed FAISS (Facebook AI Similarity Search), a high-performance library optimized for dense vector similarity search. FAISS enables fast, scalable retrieval from millions or billions of embeddings with minimal latency, making it essential for real-time retrieval in RAG pipelines and other large-scale search applications.

A. Mandal (2024) investigated cross-lingual QA in low-resource Indian languages, focusing particularly on Manipuri. Using transfer learning and multilingual embeddings, the work showed significant accuracy gains in QA performance, demonstrating that such approaches can bridge the gap for languages with limited digital resources.

S. Gao (2024) created CodeGenerationBench, a benchmark for evaluating the performance of code generation models. The research identified major challenges such as multi-step reasoning, syntax error handling, and context retention in debugging tasks, providing a foundation for improving LLM-assisted programming.

Meta AI (2023) presented an optimized implementation of RAG for large-scale deployments. Their research confirmed that coupling retrieval with generation substantially improves factual consistency, reduces model hallucination, and scales effectively to enterprise-level datasets.

Ollama (2024) introduced methods to run LLaMA-based models locally, allowing developers to deploy large language models privately without cloud dependency. This offline capability is vital for privacy-focused applications and scenarios with unreliable internet connectivity.

X. Li et al. (2023) proposed DocQA, a retrieval-augmented system specifically designed for document-level QA. It effectively processed long and structured PDFs, extracting useful information from complex layouts like tables and metadata, achieving superior performance over standard QA models.

N. Reimers I. Gurevych (2023) developed Sentence-BERT, a variant of BERT that produces high-quality semantic embeddings suitable for vector search. This advancement significantly improved the speed and accuracy of semantic retrieval, forming a key building block in many RAG and QA systems.

OpenAI (2023) published the GPT-4 technical report, which detailed substantial improvements in reasoning, creativity, and domain generalization. The report also discussed challenges such as computational cost and deployment scalability, influencing how advanced LLMs are integrated into practical systems.

Hugging Face (2024) provided extensive documentation for their Transformers library, which offers pre-trained models and tools for fine-tuning and integration. This resource simplified the implementation of RAG, QA, and code generation systems by lowering the technical barrier for developers.

Google AI (2024) presented the Google Translate API as a robust multilingual translation service. The API's accuracy and ease of integration informed this project's strategy for handling queries between Manipuri and English, ensuring effective multilingual interaction.

Y. Wang et al. (2024) developed CodeT5+, an advanced model for natural language-to-code generation. The system achieved state-of-the-art results across multiple code generation benchmarks and inspired this project's design for integrating debugging and optimization into generative workflows.

S. Rijhwani A. Anastasopoulos (2023) researched multilingual embeddings for cross-lingual retrieval, showing that shared embedding spaces improve performance for low-resource languages. Their work provided crucial insights into building retrieval systems capable of supporting diverse linguistic contexts.

H. Chase A. Ng (2024) introduced LangChain, a modular framework designed for building applications around LLMs. By providing tools for chaining prompts, retrieval, and API calls, LangChain made it easier to integrate RAG pipelines into real-world generative applications.

M. Klawe Y. Katsis (2025) explored efficient PDF parsing techniques for structured knowledge extraction. Their work addressed challenges in extracting text, metadata, tables, and figures, enabling better downstream use of PDF data in QA and summarization models.

C. Callison-Burch D. Ziegler (2024) evaluated GPT-4's debugging abilities, noting its strengths in code completion and suggestion but weaknesses in maintaining multi-step debugging context. The study provided actionable recommendations for improving LLMs in software engineering workflows.

L. Weidinger (2024) advanced contextual reasoning in LLM-based QA by combining retrieval strategies with extended context windows, allowing the model to process large documents without losing coherence or factual grounding.

G. I. Winata (2023) demonstrated that cross-lingual transfer learning significantly boosts QA performance in low-resource languages. By transferring knowledge from high-resource to low-resource settings, the study showed measurable gains in accuracy and coverage.

Z. Chen (2025) surveyed developments in neural code generation under the AutoCodeGen framework, identifying trends toward integrating automated debugging and performance optimization directly into generative models for practical software development use cases.

Unresolved Issues and Opportunities

The literature reviewed points out gaps in the integration of multilingual support, document retrieval, and code debugging into one system. The opportunities include the deployment of local frameworks such as Ollama, low-resource language multilingual embeddings, and RAG+code LLM hybrid architectures for creating affordable, scalable solutions.

Conclusion of the Literature Survey

This survey emphasizes that there is no end-to-end comprehensive platform that addresses PDF-based QA and context-aware code generation together, including with multilingual support. These findings encourage the creation of the suggested system, which resolves these issues using a privacy-preserving, locally deployable hybrid AI system.

2.2 Existing and Proposed System

2.2.1 Problem Statement and Scope of the Project

The amount of knowledge contained in PDF files, research papers, and guides keeps increasing aggressively, and it has become challenging to obtain relevant information from them effectively by the users, particularly developers and students. Previous document analysis systems are either monolingual English limited, cloud-dependent (with privacy issues), or work only on static summarization instead of interactive question answering. Equivalently, available code generation tools are mostly cloud-based and do not take into account contextual data from external documents.

The scope of this project is to develop a multilingual AI platform that can do PDF-based question answering (QA) and code generation/debugging. It has English as well as transliterated Manipuri support, operates completely offline, and integrates document retrieval, translation, and large language model reasoning into a single interface. This covers both academic use cases (research paper analysis) and developer-related tasks (code generation and optimization).

2.2.2 Methodology adopted in the proposed system

The methodology integrates multiple components into a modular pipeline:

1. PDF Parsing and Chunking:Extract text and metadata from uploaded PDFs using PyMuPDF (fitz). Then, divide the content into semantically coherent chunks for embedding.
2. Multilingual Embedding and Retrieval:Generate multilingual embeddings with Sentence-Transformers (MiniLM) and save them in FAISS for quick similarity search.
3. Language Detection and Translation:Detect if the input is in Manipuri, either transliterated or in Meetei Mayek, and translate Manipuri to English for processing.
4. Question Answering via LLM (RAG Pipeline):Retrieve the most relevant chunks, create contextual prompts, and send them to LLaMA 3 running through Ollama for reliable answers.

5. Code Generation and Debugging Module:Accept natural language prompts to create or fix code. It supports input in both English and Manipuri, along with several programming languages.

6. Frontend-Backend Integration: Implement a React-based frontend for user interaction with a Flask backend providing REST APIs for PDF QA and code generation.

2.2.3 Technical Features of the Proposed System

Unique features identified in this project include:

1. Multilingual Support (English + Manipuri): Handles both transliterated and native script queries, ensuring accessibility for Manipuri speakers.

2. Offline Retrieval-Augmented Generation (RAG):Combines FAISS-based retrieval with LLaMA 3 for privacy-preserving document QA.

3. Integrated Code Generator and Debugger:Allows code generation and correction directly from natural language prompts.

4. Cross-Domain Capability (Documents + Code): Seamlessly supports both academic document analysis and developer-focused tasks.

5. Translation + Transliteration Pipeline:Converts between Romanized Manipuri and Meetei Mayek script using Aksharamukha and IndicTrans2 models.

2.3 Tools and Technologies Used

2.3.1 Platforms

(a) Backend Framework, Flask: Flask is a Python-based microframework used to develop the backend and RESTful APIs that support the PDF question answering and code generation modules. Its lightweight structure, easy integration with machine learning workflows, and flexibility in handling API routes made it perfect for the modular design of this project.

(b) Frontend Framework, React.js: React.js was used to create a responsive, interactive, and dynamic user interface. Its component-based structure allowed for smooth integration of user input forms, result display panels, and real-time interaction with the backend services.

(c) Model Serving Platform, Ollama: Ollama acted as the local model hosting platform for running LLaMA 3 and code-specialized LLMs. Running models locally improved privacy, reduced reliance on external APIs, and ensured fast inference for both question answering and code generation tasks.

(d) Database/Vector Store, FAISS: FAISS (Facebook AI Similarity Search) was used as the high-performance vector store for indexing and retrieving document embeddings. Its effective similarity search capabilities were essential for enabling quick and accurate retrieval in the RAG pipeline.

2.3.2 Tools and Libraries

(a) PyMuPDF (fitz): This library parsed and extracted text, metadata, and structural details from uploaded PDFs. It allowed for downstream chunking and embedding generation.

(b) SentenceTransformers: This was used to generate high-quality multilingual embeddings using models like MiniLM and LaBSE. It ensured good semantic matching for both English and Manipuri queries.

(c) Hugging Face Transformers: This served as the main framework for loading and managing NLP models, including IndicTrans2 for translation and LLaMA for question answering.

(d) IndicTrans2: This enabled translation between English and Manipuri in both directions. It made content accessible for bilingual users.

(e) Aksharamukha: This was used for transliteration between Romanized Manipuri and Meetei Mayek script. It offered input flexibility for Manipuri-speaking users.

(f) Langdetect: This was implemented to automatically determine whether a query was in English or Manipuri. It triggered the necessary translation workflow.

(g) FAISS: Beyond storage, FAISS provided fast similarity-based retrieval of relevant PDF chunks. This was crucial for the retrieval-augmented generation pipeline.

(h) Ollama Python API: This enabled programmatic interaction with locally hosted LLaMA models for both natural language question answering and code generation.

2.3.3 Additional Tools

(a) Visual Studio Code (VS Code) is the main integrated development environment (IDE) for both backend (Flask) and frontend (React) development. It offers features like IntelliSense, debugging tools, and Git integration.

(b) Postman helps with API testing during development. It allows for quick validation of endpoint responses and debugging of backend services.

(c) Git and GitHub provide version control, team collaboration, and repository management. They ensure the codebase remains intact and traceable.

(d) Node.js and NPM act as the runtime and package manager for the React frontend. They make dependency management and build processes efficient.

2.4 Hardware and Software Requirements

To ensure smooth execution and usability of the system, the following hardware and software configurations were recommended:

2.4.1 Hardware Requirements

- Processor: Intel Core i5 (8th generation or newer) or equivalent AMD processor capable of handling concurrent PDF parsing, FAISS indexing, and LLM computations efficiently
- RAM: Minimum 8 GB RAM to support multilingual embeddings, translation pipelines (IndicTrans2), and smooth multi-tasking during real-time QA and code generation
- Storage: At least 512 GB SSD storage for fast I/O operations, especially for storing PDF chunks, FAISS indexes, and temporary session data
- Network: Stable internet connection required only for initial model downloads and dependency installations; the system is designed for fully offline execution afterward

2.4.2 Software Requirements

- Operating System: Compatible with Windows 10/11, Ubuntu Linux (20.04+), or macOS Monterey and above
- Programming Language: Python 3.10+ with required libraries (Flask, PyMuPDF, SentenceTransformers, Hugging Face Transformers, FAISS, Aksharamukha, Langdetect)
- LLM Serving: Ollama runtime installed locally to serve LLaMA 3 or Phind CodeLLaMA models for document QA and code generation
- Frontend Framework: React.js (Node.js 18+) for building an interactive web interface with seamless API integration
- Web Browser: Modern browsers such as Google Chrome, Mozilla Firefox, or Microsoft Edge for accessing the frontend interface

These specifications ensure that the system performs optimally while remaining lightweight enough to run on standard developer machines without reliance on cloud infrastructure or high-end GPUs.

2.4.3 Comparative Table

Table 2.1 presents a comparative analysis of the Multilingual AI Assistant against four recent and relevant systems, evaluated 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 answering and natural language-driven code generation within a fully offline, privacy-preserving environment. Unlike cloud-only platforms or models limited to single-function capabilities, 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.

Table 2.1: 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 multi-lingual QA and code generation in one tool; runs offline with re-trieval and LLM integration
RAG (Zhang et al., 2023)	Retrieval-augmented QA	Cloud-Based	Researchers	No multilin-gual support; cannot handle PDF input	Offers RAG but not suited for low-resource languages or grounded docu-ment tasks
CodeLLaMA (Meta, 2023)	Natural Language to Code Synthesis	Offline / Cloud	Developers	No document input; lacks optimization support	Open-source code generation but no retrieval or document-aware capability

Ollama (2023)	Local LLM Interface for Chat	Local Only	Developers, Privacy-Conscious Users	No UI; lacks document parsing and multi-turn reasoning	Enables local LLM use, but lacks retrieval, PDF, and 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

Chapter 3

Software Requirement Specifications

3.1 Introduction

This report outlines the functional and non-functional specifications of the AI-Powered Multilingual PDF Question Answering and Code Generation System. The system enables users to upload PDF files, question them in English or transliterated Manipuri, and get correct answers through large language models (LLMs). It also provides a code generation and debugging mode that can generate and perfect snippets of code in various programming languages.

3.1.1 Definitions, Acronyms, and Abbreviations

- PDF QA – Portable Document Format Question Answering: A system that generates answers to user queries based on content extracted from PDF documents
- LLM –Large Language Model: An advanced AI model designed for natural language understanding and generation
- FAISS – Facebook AI Similarity Search: A library for efficient storage and retrieval of vector embeddings, enabling fast similarity searches
- IndicTrans2 – Indic Translation 2: A multilingual translation model developed by AI4Bharat, used for bidirectional English–Manipuri translation
- Ollama – Local LLaMA Runtime: A platform for serving LLaMA-based models locally, enabling offline inference without reliance on cloud services
- Meetei Mayek – Manipuri Script: The traditional script used for writing the Manipuri language

3.1.2 Overview

The system operates as a web-based platform with two major modules:

1. Multilingual PDF Question Answering Module – Supports semantic search, translation, and context-aware answer generation from uploaded PDFs.
2. Code Generation and Debugging Module – Enables natural language-driven code synthesis and optimization using LLMs.

Both modules are integrated into a single web interface with offline-first architecture, ensuring privacy and cost-efficiency.

3.2 General Description

The system envisioned is a web-based multilingual artificial intelligence (AI) assistant aimed at facilitating intelligent interaction with unstructured document data and offering contextual code generation assistance. It combines several modules—PDF parsing, semantic chunking, multilingual embeddings, vector search, and large language model (LLM) reasoning—to offer two main functionalities: document-grounded question answering and natural language-led code generation and debugging. The system supports uploading of academic or technical PDFs, querying in English or transliterated Manipuri, and context-aware retrieving in real-time. It uses SentenceTransformers for multilingual embeddings, FAISS for similarity search with efficiency, and IndicTrans2 for Manipuri-English translation, with Ollama providing local LLaMA models for offline inference. With privacy and developer-friendliness at the forefront of its design, the system eschews dependence on cloud services, providing data portability and confidentiality in typical computing environments. Its modular design also enables future extension with features such as other languages, real-time collaboration, and more visualization capabilities.

3.3 Functional Requirements

The functional requirements of the Multilingual PDF Question Answering and Code Generator system outline the core features necessary to fulfill its objectives. These include:

- The system shall allow the user to upload valid PDF documents containing technical or academic content
- The system shall parse the PDF content using PyMuPDF, extracting text and metadata such as title, author, and subject
- The system shall semantically chunk the extracted text and store vector embeddings in a FAISS index for efficient retrieval
- The system shall allow the user to input questions in either English or transliterated Manipuri
- The system shall detect the input language and, if Manipuri, translate the query to English using IndicTrans2 before processing
- The system shall retrieve relevant document chunks based on semantic similarity to the user's query
- The system shall build a context-aware prompt and use Ollama-hosted LLaMA models to generate accurate answers

- The system shall optionally translate English answers back to Manipuri (romanized) for better accessibility
- The system shall provide a separate interface for code generation and debugging, allowing users to input natural language prompts describing the desired code
- The system shall use the same LLM backend (e.g., phind-codellama or LLaMA 3) for generating, explaining, or correcting code snippets
- The system shall maintain conversational context for multi-turn queries in both the QA and code generation modules
- The system shall display results clearly on the React-based web interface, with session history and metadata visibility
- The system shall maintain modularity to support future features like summarization, visualizations, and multi-user sessions

3.4 External Interface Requirements

The system interacts with several external libraries, APIs, and tools, which form its external interface dependencies. These include:

- PyMuPDF (fitz): Used to parse and extract text and metadata from PDF files for downstream processing
- FAISS (Facebook AI Similarity Search): External vector database for storing and retrieving semantic embeddings of PDF chunks
- Sentence Transformers (paraphrase-multilingual-MiniLM): Pre-trained multilingual embedding model used for encoding queries and PDF content
- IndicTrans2 (AI4Bharat): Translation model for converting Manipuri (Meetei Mayek or Romanized) to English and vice versa
- Aksharamukha Transliteration API: Used to convert Romanized Manipuri into Meetei Mayek script and vice versa for accurate translation
- Ollama + LLaMA 3 / Code LLaMA: Local LLM service used to generate answers to PDF-based queries and to synthesize/debug code from natural language prompts
- Flask (Python): Web backend framework providing REST APIs for PDF QA, summarization, and code generation modules

- React (JavaScript): Frontend framework used to create an interactive user interface with real-time API calls
- Pandas and regex: Python libraries used for data preprocessing, cleaning, and formatting of text extracted from PDFs

The above dependencies must be properly installed and configured in the development and deployment environments to ensure smooth execution and accurate results from the system.

3.5 Non-Functional Requirements

The non-functional requirements define the quality attributes, constraints, and overall expectations of the system beyond its core functionality:

1. **Performance:**The system will answer user questions in 10 to 15 seconds for PDF-based queries and in 5 to 10 seconds for code generation, given local deployment with the recommended hardware. The FAISS index retrieval can manage up to 10,000 document chunks with minimal delay.
2. **Scalability:**The structure will support future growth to manage multiple sessions at the same time and bigger document sizes (over 100 pages). It will also allow easy integration of new language models or embedding models without needing major code changes.
3. **Usability:**The web interface should be easy to use and friendly. It will allow users to upload PDFs by dragging and dropping them. Users will see answers in real-time, along with clear visual feedback like loading spinners and error messages to help them during interactions.
4. **Reliability:** The system shall gracefully handle invalid PDFs, missing metadata, or corrupted files without crashing. Failover mechanisms shall ensure that translation or embedding errors do not halt the entire process
5. **Security:**No user data or uploaded PDFs will be sent to third-party cloud services. All processing stays local to ensure data privacy. APIs will check file formats and inputs to stop harmful uploads, such as script injections.
6. **Maintainability:** Codebase shall follow modular design principles to enable easy debugging, updates, and feature additions (e.g., new languages or visualization tools)
7. **Portability:**The application shall run on Windows, Linux, and macOS with minimal configuration differences, leveraging Python and React for cross-platform compatibility.

3.6 Design Constraints

The design constraints define the boundaries and limitations within which the system must operate:

Standard Compliance

- The system shall comply with Python PEP8 coding standards for readability and maintainability
- Open-source licenses of all integrated libraries (e.g., FAISS, Ollama, IndicTrans2) shall be strictly followed
- Web technologies (HTML5, CSS3, JavaScript) shall adhere to W3C standards to ensure cross-browser compatibility

Hardware Limitations

- The local deployment requires at least 8 GB RAM and an Intel i5 or equivalent processor to run the language model smoothly.
- Systems with low RAM (less than 8 GB) may experience higher latency during embedding and model inference
- GPU acceleration is optional but not mandatory; the design targets CPU-only execution for accessibility

Software Dependencies

- The system depends on specific library versions (e.g., transformers 4.30, sentence-transformers, faiss) which must be installed correctly for functionality.
- Ollama runtime must be available on the host machine for serving the LLaMA/Mistral models locally
- The IndicTrans2 translation model requires Hugging Face cache storage to store pre-downloaded weights

Operational Constraints

- Only PDF files are supported as input; other document formats (e.g., Word, text) are currently excluded
- Internet connection is required only for initial installation of models and dependencies; runtime execution remains offline

- Queries must be concise (200 words) to prevent excessive context overflow in prompt construction

Other Requirements

- Modular structure is mandatory to allow seamless integration of additional features like code generation or visualization without major refactoring
- Data privacy must be maintained at all stages — no user-uploaded documents or embeddings should leave the local machine

Chapter 4

System Design

4.1 System Perspective

Problem Specification

The project addresses the challenge of enabling developers to analyze PDF documents and generate code through a single AI-powered platform. Existing systems often lack support for multi-language inputs (e.g., Manipuri transliteration) and context-aware retrieval from documents. The solution provides PDF-based QA and AI Code Generation in a local, privacy-preserving setup.

4.1.1 Block Diagram

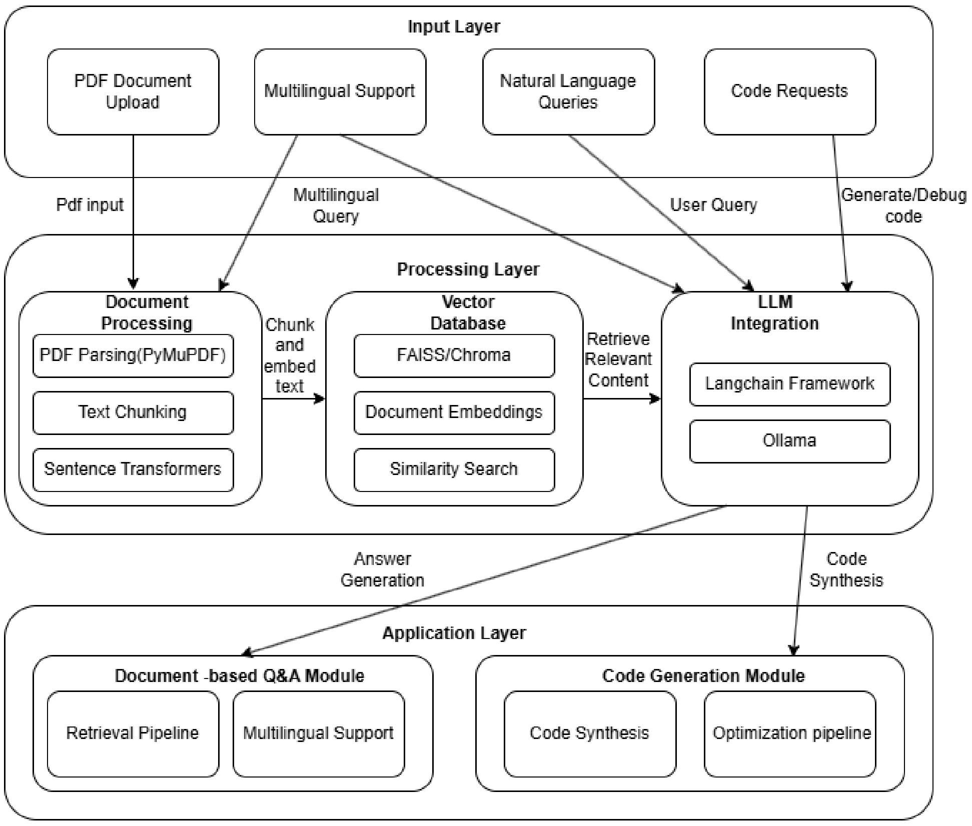


Figure 4.1: Block Diagram

Figure 4.1 shows the block diagram of the proposed system, which consists of three main layers: the Input Layer, Processing Layer, and Application Layer. The Input Layer accepts PDF document uploads, multilingual queries, natural language questions, and code requests.

In the Processing Layer, documents are parsed with PyMuPDF, divided into chunks, and embedded using Sentence Transformers before being stored in a FAISS/Chroma vector database for similarity search. The system retrieves relevant content and processes it through LLM integration with LangChain and Ollama. Finally, the Application Layer delivers results through two main modules: the Document-based QA Module, which supports retrieval and multilingual answering, and the Code Generation Module, which manages code synthesis and optimization.

4.1.2 Data Definition/Dictionary

Key data elements:

- PDF Chunks: Extracted text segments with page numbers text, page$_{n}um$
- Embeddings: High-dimensional vectors representing chunks for similarity search
- Session ID: Unique identifier to maintain per-user conversation context
- Question/Answer Object: question, answer$_{e}n, answer_{m}ni$
- Metadata: Document info title, author, keywords

4.1.3 Module Specification

- PDF Parser Module – Extracts text and metadata from PDFs using PyMuPDF
- Embedding FAISS Index Module – Converts chunks into embeddings and stores them for retrieval
- Translation Module – Detects Manipuri transliteration and translates it to English
- QA Module – Uses retrieved context and LLM to answer questions based on PDF content
- Code Generator Module – Takes prompts and generates optimized code using LLM
- Frontend Module – React-based interface for file upload, question entry, and displaying results

4.1.4 Assumptions Made

- Single-user interaction at a time; multi-user concurrency is not modeled
- Only English and Manipuri transliterated inputs supported in current version
- PDFs contain readable text (no OCR for scanned documents)
- Local LLM (Ollama) is available and preconfigured
- Vector store (FAISS) is recreated per session for isolation

Chapter 5

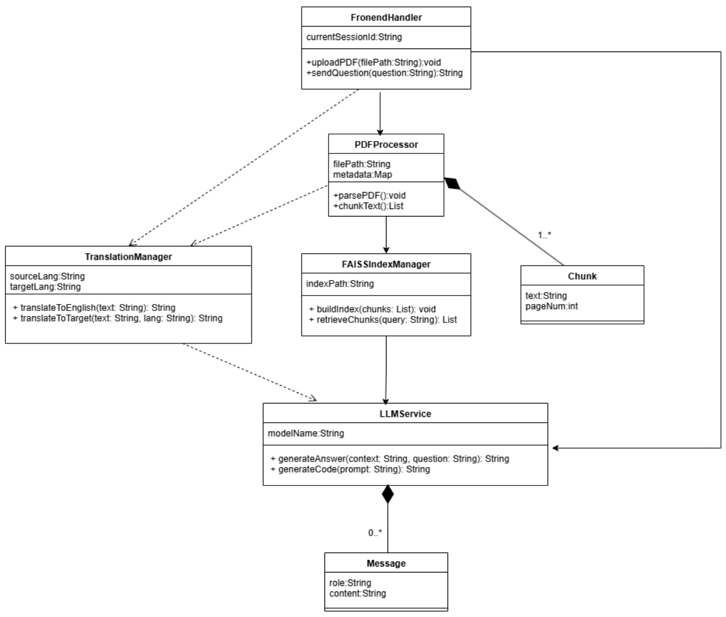
Detailed Design

The proposed system, AI-Powered Code Generator and Context-Aware PDF QA System, adopts an Object-Oriented Design (OOD) approach to ensure modularity, scalability, and ease of future enhancements. Since the system integrates multiple complex components — such as PDF parsing, translation pipeline, vector-based retrieval, LLM-based question answering, and code generation — OOD provides a structured way to encapsulate functionalities into distinct, reusable classes while maintaining clear interaction pathways between them.

5.1 Object Modeling

5.1.1 Class Diagram

The Class Diagram represents the static structure of the system and its components. It includes core classes such as PDFProcessor (responsible for parsing and chunking PDFs), Translation-Manager (handles Manipuri English translations), FAISSIndexManager (manages embeddings and retrieval), LLMService (interfaces with Ollama for generating answers and code), and FrontendHandler (manages interactions with the React UI). These classes are connected via associations that reflect data flow, ensuring modularity and extensibility — for example, new languages can be supported by extending TranslationManager without modifying other components.



FronendHandler
currentSessionId:String
+uploadPDF(filePath:String):void
+sendQuestion(question:String):String
PDFProcessor
filePath:String
metadata:Map
+parsePDF():void
+chunkText():List
1..*
TranslationManager
sourceLang:String
targetLang:String
+ translateToEnglish(text: String): String
+ translateToTarget(text: String, lang: String): String
FAISSIndexManager
indexPath:String
+ buildIndex(chunks: List): void
+ retrieveChunks(query: String): List
Chunk
text:String
pageNum:int
LLMService
modelName:String
+ generateAnswer(context: String, question: String): String
+ generateCode(prompt: String): String
0..*
Message
role:String
content:String

Figure 5.1: Class Diagram

5.2 Dynamic Modeling

5.2.1 Usecase Diagram

The Use Case Diagram illustrates how the developer interacts with two main system modules: PDF QA and AI Code Generation. The developer can upload PDFs, ask questions, view answers, and get summaries. They can also generate and debug code. Internal processes such as translation and FAISS retrieval support these functions.

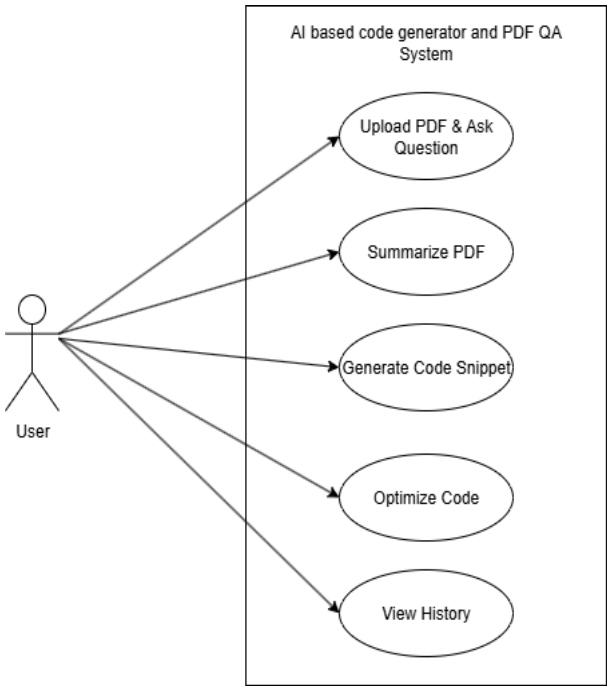

AI based code generator and PDF QA System
Upload PDF & Ask Question
Summarize PDF
Generate Code Snippet
Optimize Code
View History
User

Figure 5.2: Use Case Diagram

5.2.2 Activity Diagram

The Activity Diagram visualizes the end-to-end workflow and decision logic as a developer interacts with the AI system for PDF processing or code generation tasks.

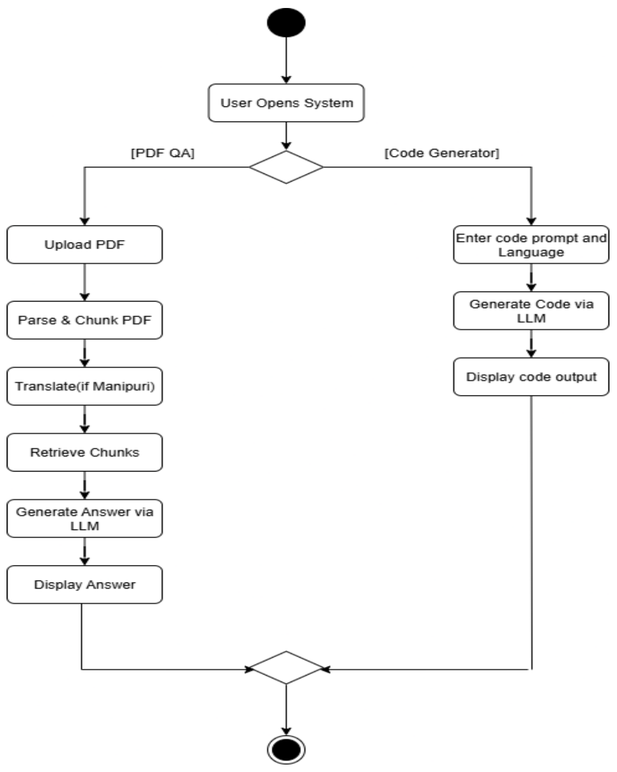


Figure 5.3: Activity Diagram

In Figure 5.3, the activity diagram depicts the operational flow of the proposed system, showing how the PDF QA module processes uploaded documents through parsing, translation, retrieval, and LLM-based answering, and how the Code Generator module transforms user prompts into executable code

PDF QA Module:The process starts when the user uploads a PDF document. The system extracts text and metadata from it. Then, the extracted text is divided into chunks and stored in a FAISS index for quick retrieval. When the user asks a question about the PDF, the translation pipeline checks if the input is in Manipuri. If it is, the system translates it into English. Next, the system retrieves the most relevant chunks from the FAISS index. It sends the question along with the retrieved context to the LLM (Ollama with LLaMA-3). The LLM generates an answer in English, which the frontend then displays to the user.

Code Generator Module:The process begins when the user enters a code generation prompt. This prompt is sent to the backend code generator module. The LLM processes the prompt and creates code in the specified programming language. The generated code is then returned to the frontend, where it is shown to the user.

5.2.3 Sequence Diagram

The Sequence Diagram illustrates the step-by-step interaction flow between system components when a developer submits a PDF query or code generation request to the AI-powered system.

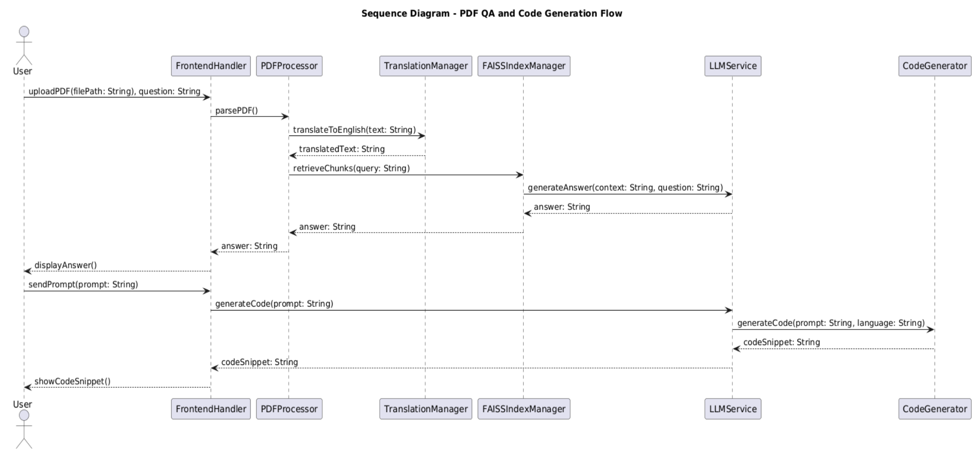

Sequence Diagram - PDF QA and Code Generation Flow
User
FrontendHandler
PDFProcessor
TranslationManager
FAISSIndexManager
LLMService
CodeGenerator
uploadPDF(filePath: String), question: String
parsePDF()
translateToEnglish(text: String)
translatedText: String
retrieveChunks(query: String)
generateAnswer(context: String, question: String)
answer: String
answer: String
answer: String
displayAnswer()
sendPrompt(prompt: String)
generateCode(prompt: String)
generateCode(prompt: String, language: String)
codeSnippet: String
codeSnippet: String
showCodeSnippet()

Figure 5.4: Sequence Diagram

In Figure 5.4, the sequence diagram outlines the interaction flow among the frontend, PDF processor, translation manager, retrieval module, LLM service, and code generator, illustrating how user queries and code prompts are processed from input to final output.

Message Flow

1. User uploads PDF or enters code prompt via the frontend interface.
2. Frontend sends request (PDF + question / code prompt) to Flask backend API.
3. For PDF QA:
   - Backend parses the PDF and extracts text/metadata.
   - Extracted text is chunked and embedded using FAISS.
   - Query is checked for Manipuri (transliteration pipeline invoked if required).
   - Relevant chunks are retrieved and passed to the LLM for answer generation.
   - Generated answer is returned to the frontend.
4. For Code Generation:
   - Backend receives code prompt and passes it directly to the LLM Code Generator module.
   - Generated code snippet is returned to the frontend.
5. Frontend displays results (answer or code) to the user with session history support.

5.3 Functional Modeling

Functional modeling shows how data flows between external entities and the system, and how the system transforms input data into output.

5.3.1 Data Flow Diagram Level 0

The Level 0 Data Flow Diagram represents the overall process at a high level.

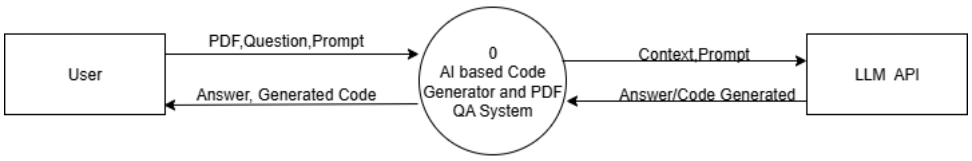


Figure 5.5: Data Flow Diagram Level 0

This Level 0 Data Flow Diagram (DFD) provides a high-level overview of the AI-powered system, showing the User as the sole external entity interacting with the AI-Powered System boundary, where the user can input either PDFs/questions in English or Manipuri or code prompts, and receive outputs as English answers (with automatic Manipuri-to-English translation implied) or generated code snippets, effectively capturing the system's core multilingual QA and code generation functionalities without exposing internal processes.

5.3.2 Data Flow Diagram Level 1

The Level 1 Data Flow Diagram decomposes the system's core processes, revealing how the AI-powered system handles PDF QA and code generation through distinct sub-processes including PDF ingestion (uploading, text extraction, and chunking), multilingual processing (Manipuri-English translation via IndicTrans2 and Aksharamukha), context retrieval (FAISS vector search), and LLM-powered generation (answer synthesis via LLaMA3 and code creation via CodeLlama), while also showing interactions with data stores for document chunks and embeddings.

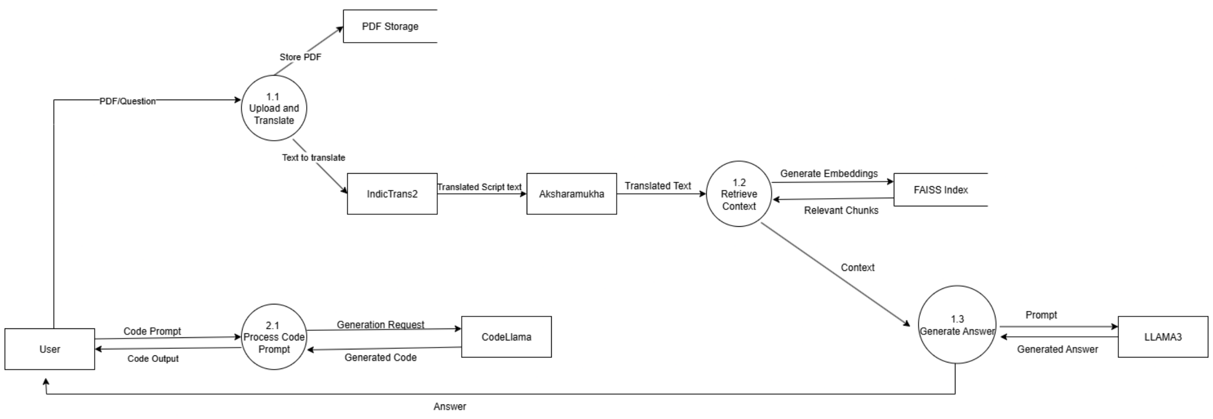


Figure 5.6: Data Flow Diagram Level 1

This Level 1 Data Flow Diagram (DFD) breaks down the AI-powered system's internal processes, detailing how user inputs flow through specialized components. The diagram reveals the PDF processing pipeline (upload → PyMuPDF extraction → text chunking → embedding generation → FAISS storage), the translation workflow (Manipuri queries passing through IndicTrans2 for language conversion and Aksharamukha for script transformation), and the dual LLM interaction paths (context-enriched questions routed to LLaMA3 for answers, code prompts directed to CodeLlama for generation). It explicitly shows data stores for document chunks and embeddings, while maintaining the system boundary to focus on functional decomposition rather than implementation specifics.

5.3.3 Assumptions Made

- System used by a single role (Developer/User); no multi-user roles considered
  one PDF is processed per session for QA
- Translation pipeline supports only Manipuri English currently
- LLM (LLaMA via Ollama) runs locally; cloud deployment not modeled
- DFDs and diagrams focus on core modules; UI state logic abstracted
- FAISS index stored per session ID; concurrency not depicted
- Frontend-backend communication assumed via REST API calls

Chapter 6

Implementation

This chapter outlines the system implementation details of the suggested system: AI-Based Code Generator and Context-Aware PDF Question Answering System. The system was implemented with Python for backend logic, Flask/FastAPI as the web framework, and React.js for the frontend interface. For sophisticated reasoning tasks, Ollama with LLaMA/CodeLLaMA models was utilized to assist with both code generation and document-grounded question answering. The implementation adopts a modular design where features like PDF parsing, Sentence Transformers-based embedding generation, retrieval via FAISS, and code synthesis are implemented as independent reusable modules to allow for scalability, maintainability, and ease of future updates.

6.1 Code Snippets

Below we present key modules of the project with code snippets and a brief explanation of their functionality.

6.1.1 Multilingual Query Processing Module

This module is responsible for detecting whether a user's query is in Manipuri (transliterated into Roman script) or English. If Manipuri is detected, the query is transliterated into Meetei Mayek script and translated into English using the IndicTrans2 model. After generating the response in English, it can also translate it back into Manipuri if required. This ensures smooth multilingual interaction with the QA system.

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from aksharamukha import transliterate
from langdetect import detect


en2mni_tokenizer = AutoTokenizer.from_pretrained("ai4bharat/indictrans2-en-
    indic-1B", trust_remote_code=True)
en2mni_model = AutoModelForSeq2SeqLM.from_pretrained("ai4bharat/indictrans2
    -en-indic-1B", trust_remote_code=True)
mni2en_tokenizer = AutoTokenizer.from_pretrained("ai4bharat/indictrans2-
    indic-en-1B", trust_remote_code=True)
mni2en_model = AutoModelForSeq2SeqLM.from_pretrained("ai4bharat/indictrans2
    -indic-en-1B", trust_remote_code=True)

def translate_to_english(text):
    inputs = mni2en_tokenizer(text, return_tensors="pt", padding=True,
        truncation=True)
    output = mni2en_model.generate(
        input_ids=inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        forced_bos_token_id=mni2en_tokenizer.lang_code_to_id["en"]
    )
    return mni2en_tokenizer.batch_decode(output, skip_special_tokens=True)
        [0]

def translate_to_target(text, target_lang="mni"):
    inputs = en2mni_tokenizer(text, return_tensors="pt", padding=True,
        truncation=True)
    output = en2mni_model.generate(
        input_ids=inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        forced_bos_token_id=en2mni_tokenizer.lang_code_to_id[target_lang]
    )
    return en2mni_tokenizer.batch_decode(output, skip_special_tokens=True)
        [0]

def transliterate_to_script(text):
    return transliterate.process("ISO", "MeeteiMayek", text)

def transliterate_to_roman(text):
    return transliterate.process("MeeteiMayek", "ISO", text)

def is_transliterated_manipuri(text):
    common_words = ["houjikti", "nahakki", "eemagi", "khallabani", "
        nattraga"]
    return any(word in text.lower() for word in common_words)
```

6.1.2 PDF Parsing and Metadata Extraction Module

This module is responsible for extracting text and metadata (title, author, subject) from uploaded PDF documents using the PyMuPDF (fitz) library. If metadata is missing, it intelligently infers the title or author from the first page content. The extracted text is then prepared for chunking and indexing.

```
    import fitz

def extract_metadata(doc):
    meta = doc.metadata or {}
    title = meta.get("title", "").strip()
    author = meta.get("author", "").strip()
    if not title or not author:
        text_lines = doc[0].get_text().split("\n")
        candidates = [line.strip() for line in text_lines if line.strip()]
        if not title and candidates:
            title = candidates[0]
        if not author and len(candidates) > 1:
            author = candidates[1]

    return {
        "title": title or "Unknown Title",
        "author": author or "Unknown Author"
    }

def fallback_title_from_page(doc):
    page1 = doc[0]
    text = page1.get_text().strip().split("\n")
    for line in text:
        if len(line.strip()) > 10:
            return line.strip()
    return "Unknown Title"

def parse_pdf(file_path):
    doc = fitz.open(file_path)
    chunks = []
    fallback_title = fallback_title_from_page(doc)

    for page_num, page in enumerate(doc):
        blocks = page.get_text("dict")['blocks']
        for block in blocks:
            if block['type'] == 0:
                for line in block['lines']:
                    text = ' '.join([span['text'].strip() for span in line[
                        'spans']])
                    if len(text) > 30:
                        chunks.append({
                            "text": text,
                            "page_num": page_num + 1
                        })

    metadata = extract_metadata(doc)
    if metadata.get("title") in ["", None, "Unknown"]:
        metadata["title"] = fallback_title
    return chunks, metadata
```

6.1.3 Chunking and Embedding (RAG Engine) Module

This module is responsible for splitting the extracted PDF text into semantically coherent chunks (e.g., 500 words per chunk). Each chunk is converted into vector embeddings using the SentenceTransformer (paraphrase-multilingual-MiniLM) model. The embeddings are stored in a FAISS index for efficient similarity search during question answering.

```
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np
import os
import json

embedding_model = SentenceTransformer("paraphrase-multilingual-MiniLM-L12-
    v2")

def build_faiss_index(chunks, session_id):
    texts = [chunk['text'] for chunk in chunks]
    embeddings = embedding_model.encode(texts, convert_to_numpy=True)

    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings)

    os.makedirs("indexes", exist_ok=True)
    faiss.write_index(index, f"indexes/{session_id}.index")

    with open(f"indexes/{session_id}_meta.json", 'w') as f:
        json.dump(chunks, f)

def retrieve_chunks(query, session_id, top_k=5):
    index = faiss.read_index(f"indexes/{session_id}.index")
    with open(f"indexes/{session_id}_meta.json", 'r') as f:
        metadata = json.load(f)

    query_vec = embedding_model.encode([query], convert_to_numpy=True)
    _, I = index.search(query_vec, top_k)
    return [metadata[i] for i in I[0]]
```

6.1.4 Retrieval and LLM Answer Generation Module

This module is responsible for retrieving the most relevant chunks from the FAISS index based on the user's query and constructing a context-rich prompt. The prompt, along with metadata, is sent to the Ollama LLaMA3 model to generate precise, context-aware answers to user questions.

```

import ollama
from langdetect import detect
```

```python
from translator import (
    indic_translate,
    transliterate_to_roman,
    transliterate_to_script,
    en2mni_model,
    en2mni_tokenizer,
    mni2en_model,
    mni2en_tokenizer,
)

def is_romanized_manipuri(text):
    manipuri_clues = ['hou', 'eikhoigi', 'nattraga', 'phangjaba', 'houjik']
    return any(clue in text.lower() for clue in manipuri_clues)

def generate_answer_ollama(context_chunks, query, metadata=None, model='
    llama3'):
    context_text = "\n".join([f"Page {chunk['page_num']}: {chunk['text']}"
        for chunk in context_chunks])
    metadata = metadata or {}

    original_query = query
    detected_lang = 'en'

    try:
        detected_lang = detect(query)
    except:
        pass


    if is_romanized_manipuri(query) or detected_lang not in ['en']:

        mm_text = transliterate_to_script(query)
        query = indic_translate(mm_text, source_lang='mni', target_lang='
            eng', model=mni2en_model, tokenizer=mni2en_tokenizer)
        detected_lang = 'mni'


    prompt = (
        f"You are a helpful assistant answering questions based on the
            uploaded PDF.\n\n"
        f"Metadata:\nTitle: {metadata.get('title', 'N/A')}\nAuthor: {
            metadata.get('author', 'N/A')}\nSubject: {metadata.get('subject
            ', 'N/A')}\n\n"
        f"Context:\n{context_text}\n\n"
        f"Question: {query}\nAnswer:"
    )
    try:
        response = ollama.chat(
            model=model,
            messages=[{"role": "user", "content": prompt}]
        )
        answer = response['message']['content']
    except Exception as e:
        answer = f"Error generating answer: {str(e)}"


    if detected_lang == 'mni':
        mm_back = indic_translate(answer, source_lang='eng', target_lang='
           mni', model=en2mni_model, tokenizer=en2mni_tokenizer)
        answer = transliterate_to_roman(mm_back)

    return answer
```

6.1.5 Code Generation Module

This module handles code generation, optimization, and explanation requests. It takes natural language prompts from the frontend, constructs context-specific prompts for the LLM (LLaMA 3 via Ollama), and returns either:

- Generated code snippets
- Optimized versions of existing code
- Explanations and complexity analysis (if requested)

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import ollama
import re
import json

app = Flask(__name__)
CORS(app)

conversations = {}


def detect_intent(prompt: str) -> str:
    prompt = prompt.lower()
    if any(word in prompt for word in ["explain", "describe", "what does",
        "analyze"]):
        return "explanation"
    elif "optimize" in prompt:
        return "optimize"
    else:
        return "code"


def extract_language(prompt: str) -> str:
    match = re.search(r'in\s+(\w+\+\+|\w+)', prompt.lower())
    return match.group(1) if match else "code"


def build_prompt(prompt: str, optimize: bool = False) -> str:
    intent = detect_intent(prompt)
    language = extract_language(prompt)

    if intent == "explanation":
        return f"Please explain the following {language} code:\n\n{prompt}"
    elif optimize:
        return (
            f"Please optimize the following {language} code:\n\n{prompt}\n\
               n"
            f"Return only the optimized code, wrapped in triple backticks (

). No extra explanations."
        )
    else:
        return (
            f"Write a {language} program or snippet for the following
               requirement:\n\n{prompt}\n\n"
            f"Return only the code, wrapped in triple backticks (

). Do not include any explanation."
        )


@app.route("/api/code", methods=["POST"])
def generate_code():
    data = request.get_json()
    prompt = data.get("prompt")
    optimize = data.get("optimize", False)
    session_id = data.get("session_id", "default")

    if not prompt:
        return jsonify({"error": "Prompt is required"}), 400

    try:
        full_prompt = build_prompt(prompt, optimize)
        # Initialize session
        if session_id not in conversations:
            conversations[session_id] = [
                {
                    "role": "system",
                    "content": (
                        "You are a helpful coding assistant. "
                        "When users ask for code, return only the code
                            wrapped in triple backticks (

). "
                        "Do not add any extra explanation unless
                            specifically asked to explain."
                    )
                }
            ]

        conversations[session_id].append({"role": "user", "content":
            full_prompt})


        response = ollama.chat(model="llama3", messages=conversations[
            session_id])
        code = response["message"]["content"]
        conversations[session_id].append({"role": "assistant", "content":
            code})

        reason = ""
        metrics_data = {}


        if optimize:

            explanation_prompt = (
                f"Compare the original and optimized code below and explain
                    the improvements:\n\n"
                f"Original Code:\n{prompt}\n\nOptimized Code:\n{code}"
            )
            explanation = ollama.chat(model="llama3", messages=[{"role": "
                user", "content": explanation_prompt}])
            reason = explanation["message"]["content"]


            metrics_prompt = f"""
Estimate and compare the runtime performance and memory usage (space
    complexity) between the following two versions of code. Give the result
    as a JSON:
{{
  "time_complexity": {{ "original": "...", "optimized": "..." }},
  "space_complexity": {{ "original": "...", "optimized": "..." }},
  "remarks": "..."
}}

Original Code:
{prompt}

Optimized Code:
{code}
"""
            metrics_response = ollama.chat(model="llama3", messages=[{"role
                ": "user", "content": metrics_prompt}])
            metrics_text = metrics_response["message"]["content"]

            try:
                metrics_data = json.loads(metrics_text.strip())
            except json.JSONDecodeError:
                metrics_data = {"error": "Failed to parse metrics"}

        return jsonify({
            "code": code,
            "reason": reason,
            "metrics": metrics_data
        })

    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route("/api/clear", methods=["POST"])
def clear_session():
    session_id = request.get_json().get("session_id", "default")
    conversations.pop(session_id, None)
    return jsonify({"status": "cleared"})


if __name__ == "__main__":
    app.run(debug=True, port=5002)
```

6.1.6 Backend POST Endpoints

This module implements PDF-based question answering and summarization using a retrieval-augmented generation (RAG) approach. It exposes two Flask endpoints: /api/pdfqa for answering user queries about an uploaded PDF and /api/summary for generating concise summaries. The module extracts text and metadata from the PDF using PyMuPDF, chunks the text, and builds a FAISS index for semantic retrieval. User questions, including transliterated Manipuri, are translated to English, relevant chunks are retrieved, and a structured prompt is sent to LLaMA (via Ollama) for generating answers. Session handling maintains conversational context across multiple queries.

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from translator import translate_to_english, is_transliterated_manipuri
from rag_engine import chunk_pdf_text, build_faiss_index, retrieve_chunks
import fitz  # PyMuPDF
import ollama
import tempfile
import os

app = Flask(__name__)
CORS(app)

conversations = {}

@app.route('/api/pdfqa', methods=['POST'])
def handle_pdf_qa():
    file = request.files['file']
    question = request.form['question']
    session_id = request.form['session_id']

    if session_id not in conversations:
        conversations[session_id] = []

    with tempfile.NamedTemporaryFile(delete=False, suffix=".pdf") as temp:
        file.save(temp.name)
        pdf_path = temp.name

    text = extract_pdf_text(pdf_path)
    metadata = extract_pdf_metadata(pdf_path)
    chunks = chunk_pdf_text(text)
    build_faiss_index(chunks, session_id)


    if is_transliterated_manipuri(question):
        try:
            question = translate_to_english(question)
        except Exception as e:
            print("Translation error:", e)
            return jsonify({"error": "Translation failed"}), 500

    retrieved_chunks = retrieve_chunks(question, session_id)
    context_text = "\n\n".join([chunk['text'] for chunk in retrieved_chunks
        ])

    prompt = build_prompt(context_text, question, conversations[session_id
        ])
    response = ollama.chat(model='llama3', messages=prompt)
    answer_en = response['message']['content'].strip()

    conversations[session_id].append({"role": "user", "content": question})
    conversations[session_id].append({"role": "assistant", "content":
        answer_en})

    return jsonify({
        "answer": {
            "en": answer_en,
            "mni": None
        },
        "metadata": metadata
    })

@app.route('/api/summary', methods=['POST'])
def summarize_pdf():
    file = request.files['file']
    with tempfile.NamedTemporaryFile(delete=False, suffix=".pdf") as temp:
        file.save(temp.name)
        pdf_path = temp.name

    text = extract_pdf_text(pdf_path)
    prompt = f"Summarize the following PDF content:\n\n{text}"

    response = ollama.chat(model='llama3', messages=[
        {"role": "system", "content": "You are a helpful assistant that
            summarizes PDF content."},
        {"role": "user", "content": prompt}
    ])
    summary = response['message']['content'].strip()

    return jsonify({
        "summary": summary
    })
```

```
def extract_pdf_text(path):
    doc = fitz.open(path)
    return "\n".join(page.get_text() for page in doc).strip()

def extract_pdf_metadata(path):
    doc = fitz.open(path)
    meta = doc.metadata
    return {
        "title": meta.get('title', ''),
        "author": meta.get('author', ''),
        "subject": meta.get('subject', ''),
        "keywords": meta.get('keywords', ''),
    }

def build_prompt(context, question, chat_history):
    system_prompt = {"role": "system", "content": "You are an expert PDF
        question answering assistant."}
    messages = [system_prompt] + chat_history + [{"role": "user", "content"
        : f"Context:\n{context}\n\nQuestion: {question}"}]
    return messages

if __name__ == '__main__':
    app.run(port=5001)
```

6.1.7 Frontend Design

The frontend of the AI-Based Code Generator and Context-Aware QA System was developed using React.js, designed to provide an intuitive, modular, and responsive interface for two primary features:

1. Context-Aware PDF Question Answering (PDF QA)
2. AI-Based Code Generation and Debugging

The interface ensures smooth interaction with the backend APIs, real-time response display, and session-based history for users.

Key Features of the Frontend

1. **PDF QA**

    - PDF Upload and Question Input: Allows users to upload PDFs and ask questions in English or transliterated Manipuri
    - Answer Display: Shows answers in English and optionally Manipuri after translation

- Summary Feature: Summarizes entire PDFs for quick insights
- Session History: Maintains previous QA pairs for easy reference

2. **Code Generation**

   - Prompt-Based Code Generation: Accepts user prompts in natural language to generate code snippets in multiple programming languages
   - Multi-Language Support: Supports languages like Python, Java, C++, etc., based on user input or selection
   - Code Debugging and Optimization: Users can paste existing code to request debugging or performance optimization
   - Integrated Navigation: A navigation bar allows switching between Code Generator and PDF QA modules seamlessly

6.1.8 Frontend Flow

For PDF QA

1. User uploads a PDF.
2. Enters a question (in English or transliterated Manipuri).
3. Data is sent via FormData to the /api/pdfqa endpoint.
4. Backend retrieves relevant chunks, runs LLM analysis, and returns answers.
5. Answer (English + Manipuri) and metadata are displayed.
6. User can request a PDF summary via /api/summary.
7. QA pairs are saved in session history for re-access.

For Code Generator

1. User navigates to the Code Generator page via the navbar.
2. User inputs a prompt (e.g., ”Generate Python code for Fibonacci sequence”) or pastes code for debugging.
3. The prompt/code is sent to the /api/codegen endpoint (Flask + Ollama backend).
4. Backend uses the phind-codellama model (or similar) to generate or debug code.
5. Generated/optimized code is displayed in a formatted output panel.
6. User can copy the generated code or re-prompt for modifications.

Code Snippet

Listing 6.1: React Component for PDF QA Module

```
import React, { useState, useRef } from 'react';
import './PdfQa.css';

const PdfQa = ({ onLogout, onCodeGeneratorNavigate }) => {
  const [pdfFile, setPdfFile] = useState(null);
  const [question, setQuestion] = useState('');
  const [answerEn, setAnswerEn] = useState('');
  const [answerMni, setAnswerMni] = useState('');
  const [summary, setSummary] = useState('');
  const [metadata, setMetadata] = useState({});
  const [loading, setLoading] = useState(false);
  const [copied, setCopied] = useState(false);
  const [history, setHistory] = useState([]);
  const sessionId = useRef(Date.now().toString());

  const handleFileChange = (e) => {
    const file = e.target.files[0];
    setPdfFile(file);
    setAnswerEn('');
    setAnswerMni('');
    setQuestion('');
    setSummary('');
    setMetadata({});
    setCopied(false);
  };

  const handleAsk = async () => {
  if (!pdfFile || !question.trim()) return;
  setLoading(true);
  setCopied(false);

  const formData = new FormData();
  formData.append('file', pdfFile);
  formData.append('question', question);
  formData.append('session_id', sessionId.current);

  try {
    const res = await fetch('http://localhost:5001/api/pdfqa', {
      method: 'POST',
      body: formData,
    });
    const data = await res.json();
    setAnswerEn(data.answer?.en || '');
    setAnswerMni(data.answer?.mni || '');
    setMetadata(data.metadata || {});
    setHistory((prev) => [...prev, {
      question,
      answer_en: data.answer?.en || '',
      answer_mni: data.answer?.mni || ''
    }]);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    setLoading(false);
  }
};


  const handleSummarize = async () => {
    if (!pdfFile) return;
    setLoading(true);

    const formData = new FormData();
    formData.append('file', pdfFile);
    formData.append('session_id', sessionId.current);

    try {
      const res = await fetch('http://localhost:5001/api/summary', {
        method: 'POST',
        body: formData,
      });
      const data = await res.json();
      setSummary(data.summary || '');
    } catch (err) {
      console.error('Summary error:', err);
    } finally {
      setLoading(false);
    }
  };
```

```
const handleClear = () => {
  setQuestion('');
  setAnswerEn('');
  setAnswerMni('');
  setCopied(false);
};

return (
  <>
    <nav className="pdfqa-navbar">
      <div className="brand">PdfQA</div>
      <div style={{ display: 'flex', gap: '12px' }}>
        <button className="nav-btn" onClick={
            onCodeGeneratorNavigate}>Code Generator</button>
        <button className="logout-btn" onClick={onLogout}>Logout</
            button>
      </div>
    </nav>

    <div className="pdfqa-main">
      <aside className="pdfqa-sidebar">
        <h3>Session History</h3>
        {history.length === 0 ? (
          <p style={{ color: '#888' }}>No history yet</p>
        ) : (
          history.map((item, idx) => (
            <div
              key={idx}
              className="pdfqa-history-item"
              onClick={() => {
                setQuestion(item.question);
                setAnswerEn(item.answer_en);
                setAnswerMni(item.answer_mni);
              }}
            >
              {item.question.slice(0, 40)}...
            </div>
          ))
        )}
      </aside>
```

```
        <main className="pdfqa-content">
          <h2 className="pdfqa-title">PDF Context-Aware QA</h2>

          <label className="custom-file-upload">
            <input type="file" accept="application/pdf" onChange={
               handleFileChange} />
            Upload PDF
          </label>

          {pdfFile && (
            <p style={{ marginTop: '0.5rem', color: '#666' }}>
              <strong>Uploaded:</strong> {pdfFile.name}
            </p>
          )}

          {metadata.title && (
            <div className="metadata">
              <h4>       PDF Metadata:</h4>
              <p><strong>Title:</strong> {metadata.title}</p>
            </div>
          )}

          <textarea
            className="pdfqa-input"
            placeholder="Ask a question based on the uploaded PDF
               ..."
            value={question}
            onChange={(e) => setQuestion(e.target.value)}
          />

          <div className="pdfqa-buttons">
            <button className="ask-btn" onClick={handleAsk} disabled
               ={!pdfFile}>Ask</button>
            <button className="clear-btn" onClick={handleClear}>
               Clear</button>
            <button className="ask-btn" onClick={handleSummarize}
               disabled={!pdfFile}>Summarize</button>
          </div>

          {loading && <div className="pdfqa-spinner">Loading....</
             div>}
          {!loading && (answerEn || answerMni) && (
            <div className="pdfqa-output">
              {answerEn && <p><strong>Answer (English):</strong> {
                 answerEn}</p>}
              {answerMni && <p><strong>Answer (Manipuri):</strong> {
                 answerMni}</p>}
            </div>
          )}
          <br></br>
          {!loading && summary && (
            <div className="pdfqa-output">
              <p><strong>Summary:</strong> {summary}</p>
            </div>
          )}

          {!loading && !answerEn && !summary && (
            <p className="empty-state">Your answer or summary will
               appear here.</p>
          )}
        </main>
      </div>
    </>
  );
};

export default PdfQa;
```

Listing 6.2: React Component for Code Generation Module

```
import React, { useState, useRef, useEffect } from 'react';
import './CodeGenerator.css';
import { Chart } from 'chart.js/auto';

const CodeGenerator = ({ onLogout, onPdfQaNavigate }) => {
  const [prompt, setPrompt] = useState('');
  const [response, setResponse] = useState('');
  const [reason, setReason] = useState('');
  const [metrics, setMetrics] = useState(null);
  const [loading, setLoading] = useState(false);
  const [copied, setCopied] = useState(false);
  const [history, setHistory] = useState([]);
  const textareaRef = useRef(null);
  const chartRef = useRef(null);
  const sessionId = useRef(Date.now().toString());

  const handleGenerate = async () => {
    if (!prompt.trim()) return;
    setLoading(true);
    setCopied(false);
    setReason('');
    setMetrics(null);

    try {
      const res = await fetch('http://localhost:5002/api/code', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ prompt, session_id: sessionId.current
            })
      });

      const data = await res.json();
      setResponse(data.code || '');
      setHistory(prev => [...prev, { prompt, response: data.code }])
         ;
    } catch (err) {
      console.error('Error:', err);
    } finally {
      setLoading(false);
    }
  };

  const handleOptimize = async () => {
    if (!prompt.trim()) return;
    setLoading(true);
    setCopied(false);

    try {
      const res = await fetch('http://localhost:5002/api/code', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ prompt: response, optimize: true,
           session_id: sessionId.current })
      });

      const data = await res.json();
      setResponse(data.code || '');
      setReason(data.reason || '');
      setMetrics(data.metrics || {});
      setHistory(prev => [...prev,   {
        prompt: 'Optimize',
        response: data.code,
        reason: data.reason,
        metrics: data.metrics
      }]);
    } catch (err) {
      console.error('Error:', err);
    } finally {
      setLoading(false);
    }


  };

  const renderChart = () => {
    if (!metrics || !metrics.time_complexity || !metrics.
       space_complexity) return;

    const ctx = chartRef.current.getContext('2d');
    new Chart(ctx, {
      type: 'bar',
      data: {
        labels: ['Time Complexity', 'Space Complexity'],
        datasets: [
          {
            label: 'Original',
            data: [
              metrics.time_complexity.original.length,
              metrics.space_complexity.original.length
            ],
            backgroundColor: 'rgba(255, 99, 132, 0.6)'
          },
          {
            label: 'Optimized',
            data: [
              metrics.time_complexity.optimized.length,
              metrics.space_complexity.optimized.length
            ],
            backgroundColor: 'rgba(54, 162, 235, 0.6)'
          }
        ]
      },
      options: {
        responsive: true,
        plugins: { legend: { position: 'bottom' } }
      }
    });
  };

  useEffect(() => {
    if (metrics) renderChart();
  }, [metrics]);

  const handleClear = () => {
    setPrompt('');
    setResponse('');
    setReason('');
    setMetrics(null);
    setCopied(false);
  };

  const handleCopy = () => {
    navigator.clipboard.writeText(response);
    setCopied(true);
    setTimeout(() => setCopied(false), 1500);
  };

  const handleKeyDown = (e) => {
    if (e.ctrlKey && e.key === 'Enter') {
      handleGenerate();
    }
  };

  useEffect(() => {
    if (textareaRef.current) {
      textareaRef.current.focus();
    }
  }, []);

  return (
    <>
      <nav className="codegen-navbar">
        <div className="brand">CodeGen</div>
        <div style={{display:'flex', gap:'12px'}}>
        <button className='nav-btn' onClick={onPdfQaNavigate}>PDF-QA
           </button>
        <button className="logout-btn" onClick={onLogout}>Logout</
           button>
        </div>
      </nav>

      <div className="codegen-main">
        <aside className="codegen-sidebar">
          <h3>Session History</h3>
          {history.length === 0 ? (
            <p style={{ color: '#888' }}>No history yet</p>
          ) : (
            history.map((item, idx) => (
              <div key={idx} className="codegen-history-item"
                 onClick={() => {
                setPrompt(item.prompt);
                setResponse(item.response);
                setReason(item.reason || '');
                setMetrics(item.metrics || null);
              }}>
                {item.prompt.slice(0, 40)}...
              </div>
            ))
          )}
        </aside>

        <main className="codegen-content">
          <h2 className="codegen-title">AI Code Generator &
             Optimizer</h2>

          <textarea
```

```
            ref={textareaRef}
            className="prompt-input"
            placeholder="Type your request (e.g., write in Python,
               optimize, explain)..."
            value={prompt}
            onChange={(e) => setPrompt(e.target.value)}
            onKeyDown={handleKeyDown}
          />

          <div className="codegen-buttons">
            <button className="generate-btn" onClick={handleGenerate
               }>Generate</button>
            <button className="optimize-btn" onClick={handleOptimize
               } disabled={!response}>Optimize</button>
            <button className="clear-btn" onClick={handleClear}>
               Clear</button>
          </div>

          {loading && <div className="codegen-spinner"></div>}<br></
             br>

          {!loading && response && (
            <div className="codegen-output">
              <button className="copy-btn" onClick={handleCopy}>
                {copied ? 'Copied!' : 'Copy'}
              </button>
              <pre>{response}</pre>
            </div>
          )}
          <br></br>

          {!loading && reason && (
\begin{figure}
    \centering
    \includegraphics[width=0.5\linewidth]{code_generation.png}
    \caption{Enter Caption}
    \label{fig:enter-label}
\end{figure}
\begin{figure}
                \centering
                \includegraphics[width=0.5\linewidth]{
                   Selection_window.png}
                \caption{Enter Caption}
                \label{fig:enter-label}
            \end{figure}
                        <div className="codegen-explanation">
              <h4>Why is the optimized code better?</h4>
\begin{figure}
                  \centering
                  \includegraphics[width=0.5\linewidth]{pdfQA.png}
                  \caption{Enter Caption}
                  \label{fig:enter-label}
              \end{figure}
                            <p>{reason}</p>
            </div>
          )}

          {!loading && metrics && metrics.time_complexity && (
            <div className="codegen-metrics">
              <h4>Estimated Performance Comparison</h4>
              <ul>
                <li><strong>Time Complexity:</strong> Original     {
                   metrics.time_complexity.original}, Optimized
                   {metrics.time_complexity.optimized}</li>
                <li><strong>Space Complexity:</strong> Original
                   {metrics.space_complexity.original}, Optimized
                       {metrics.space_complexity.optimized}</li>
              </ul>
              {metrics.remarks && <p><strong>Remarks:</strong> {
                 metrics.remarks}</p>}
              <canvas ref={chartRef} width="400" height="200"></
                 canvas>
            </div>
          )}

        </main>
      </div>
    </>
  );
};

export default CodeGenerator;
```

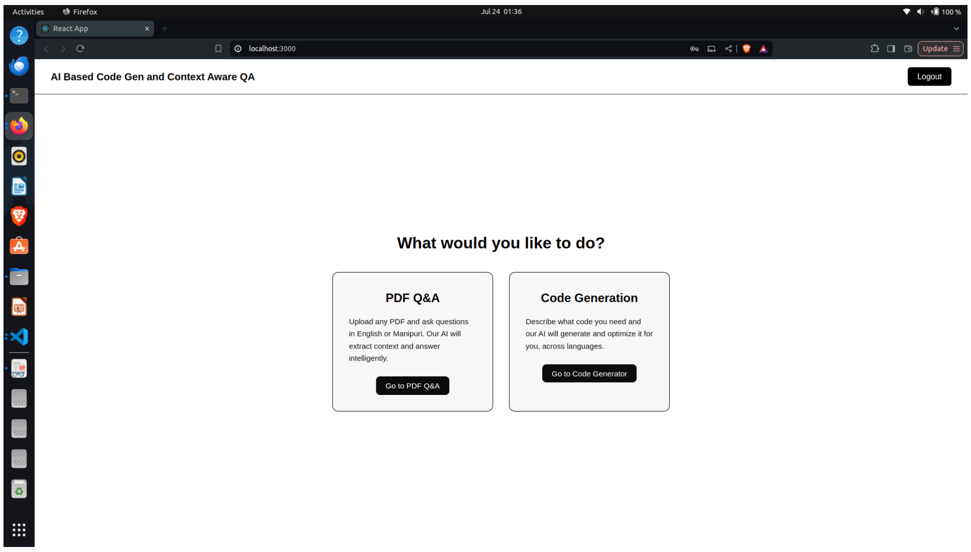
```
\begin{figure}
    \centering
    \includegraphics[width=0.5\linewidth]{landing_page.png}
    \caption{Enter Caption}
    \label{fig:enter-label}
\end{figure}
```

6.2 Implementation

The Multilingual PDF QA and AI Code Generator system has been implemented as a modular web-based application integrating frontend, backend, vector retrieval, and local LLM inference. Each module operates independently yet collaborates to provide seamless PDF question-answering and code generation functionalities. Below is a detailed explanation of each module, accompanied by indicative screenshots.

1. **Frontend Module (React.js)**
   Input/Output:

   - Implements two pages: PDF QA and Code Generator
   - Features include file upload, text input, session history, navigation between modules, and dynamic output rendering
   - Utilizes React state management (useState, useRef) for handling session and user interaction



AI Based Code Gen and Context Aware QA

What would you like to do?

PDF Q&A

Upload any PDF and ask questions in English or Manipuri. Our AI will extract context and answer intelligently.

Code Generation

Describe what code you need and our AI will generate and optimize it for you, across languages.

Figure 6.1: PDF QA and Code Generator Navigation

The figure 6.1 shows the navigation interface of the application. It lets users easily switch between the PDF Question-Answering (QA) module and the Code Generator module using buttons on the dashboard.

2. **PDF Upload and Parsing Module**
   Input: User-uploaded PDF file (academic papers, reports).
   Process:

   - Uses PyMuPDF to extract text and metadata (title, author, keywords).
   - Breaks text into semantic chunks for efficient retrieval.
   - Handles PDFs without metadata by inferring titles from the first page. Output: Parsed text chunks stored temporarily for embedding.

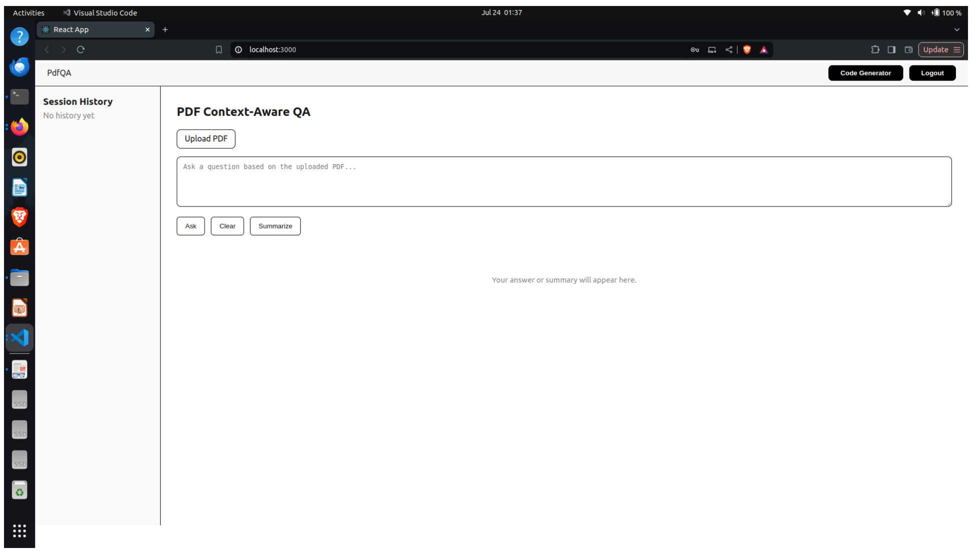


Figure 6.2: PDF Upload Interface

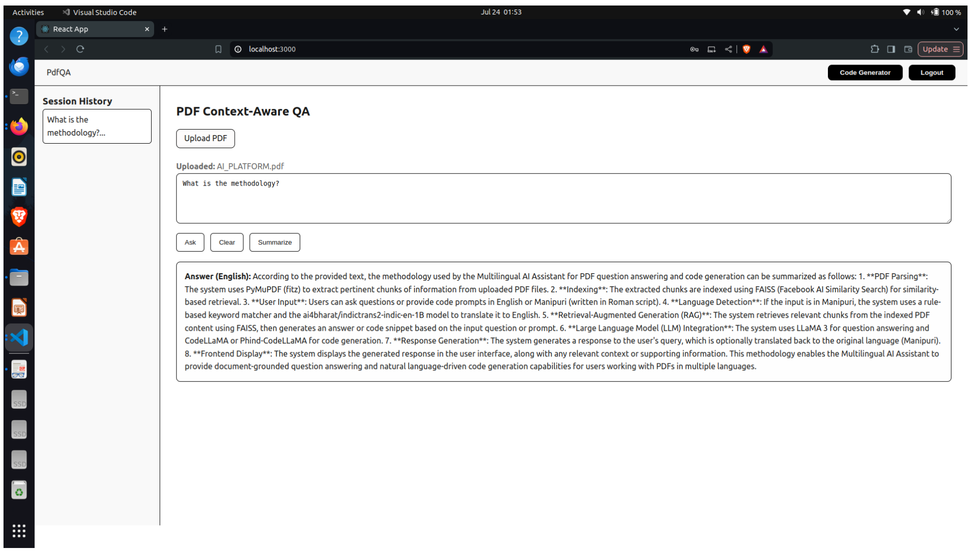
Figure 6.2 shows the PDF upload interface. Users can quickly select and upload documents for processing. This feature serves as the starting point for the document-based QA workflow.

3. **Embedding and FAISS Indexing Module**
   Input: Parsed PDF chunks.
   Process:

   - Uses PyMuPDF to extract text and metadata (title, author, keywords).
   - Generates multilingual embeddings using SentenceTransformers (MiniLM).
   - Builds a FAISS vector index for semantic search and fast retrieval of relevant document parts.

Output: FAISS index and chunk metadata stored locally (session-wise).

4. **Question Answering Module (PDF QA)**
   Input: User question (English or transliterated Manipuri)
   Process:
   - Detects language using heuristics + langdetect
   - Transliterated Manipuri → Meetei Mayek (Aksharamukha) → English translation (IndicTrans2)
   - Retrieves top-k relevant chunks using FAISS
   - Builds contextual prompt and queries LLaMA 3 via Ollama for answer generation. Output: Answer displayed in English



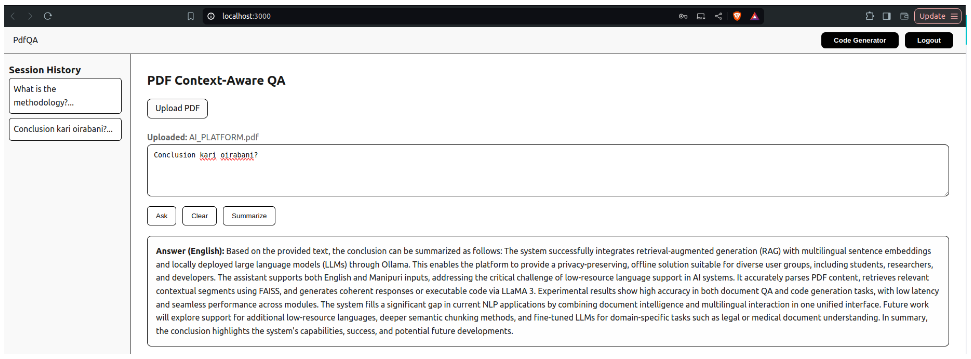
Uploaded: AI_PLATFORM.pdf

```
What is the methodology?
```

Answer (English): According to the provided text, the methodology used by the Multilingual AI Assistant for PDF question answering and code generation can be summarized as follows: 1. **PDF Parsing**: The system uses PyMuPDF (fitz) to extract pertinent chunks of information from uploaded PDF files. 2. **Indexing**: The extracted chunks are indexed using FAISS (Facebook AI Similarity Search) for similarity-based retrieval. 3. **User Input**: Users can ask questions or provide code prompts in English or Manipuri (written in Roman script). 4. **Language Detection**: If the input is in Manipuri, the system uses a rule-based keyword matcher and the ai4bharat/indictrans2-indic-en-1B model to translate it to English. 5. **Retrieval-Augmented Generation (RAG)**: The system retrieves relevant chunks from the indexed PDF content using FAISS, then generates an answer or code snippet based on the input question or prompt. 6. **Large Language Model (LLM) Integration**: The system uses LLaMA 3 for question answering and CodeLLaMA or Phind-CodeLLaMA for code generation. 7. **Response Generation**: The system generates a response to the user's query, which is optionally translated back to the original language (Manipuri). 8. **Frontend Display**: The system displays the generated response in the user interface, along with any relevant context or supporting information. This methodology enables the Multilingual AI Assistant to provide document-grounded question answering and natural language-driven code generation capabilities for users working with PDFs in multiple languages.

Figure 6.3: Generated Answer of the English Prompt

Figure 6.3 shows a generated answer in response to an English prompt entered in the PDF QA module. It demonstrates the system's ability to handle English-language questions and provide a relevant answer based on the uploaded PDF content.



Uploaded: AI_PLATFORM.pdf

```
Conclusion kari oirabani?
```

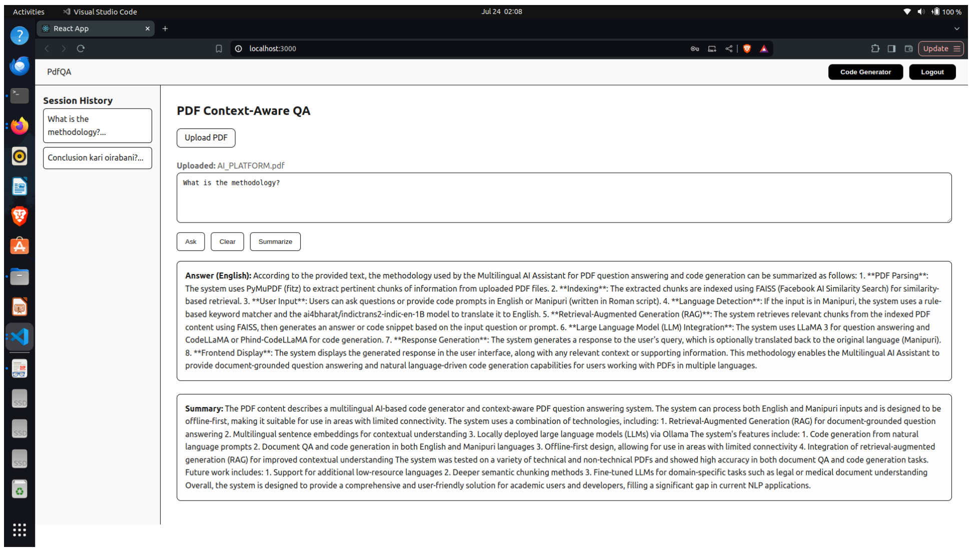
Answer (English): Based on the provided text, the conclusion can be summarized as follows: The system successfully integrates retrieval-augmented generation (RAG) with multilingual sentence embeddings and locally deployed large language models (LLMs) through Ollama. This enables the platform to provide a privacy-preserving, offline solution suitable for diverse user groups, including students, researchers, and developers. 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 generates coherent responses or executable code via LLaMA 3. Experimental results show high accuracy in both document QA and code generation tasks, with low latency and seamless performance across modules. The system fills a significant gap in current NLP applications by combining document intelligence and multilingual interaction in one unified interface. 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 document understanding. In summary, the conclusion highlights the system's capabilities, success, and potential future developments.

Figure 6.4: Generated answer from Manipuri(transliterated) prompt

Figure 6.4 shows a generated answer to a Manipuri (transliterated) prompt. This example highlights the system's ability to handle multiple languages. It detects the transliterated Manipuri input, translates it into English, processes it through retrieval and LLM generation, and returns a contextually correct answer.

5. **PDF Summarization Module**
   Input: Uploaded PDF document.
   Process

   - Concatenates top document chunks
   - Prompts LLaMA 3 to generate concise summaries
     Output: Summarized text presented below the QA interface



Uploaded: AI_PLATFORM.pdf

```
What is the methodology?
```

Answer (English): According to the provided text, the methodology used by the Multilingual AI Assistant for PDF question answering and code generation can be summarized as follows: 1. **PDF Parsing**: The system uses PyMuPDF (fitz) to extract pertinent chunks of information from uploaded PDF files. 2. **Indexing**: The extracted chunks are indexed using FAISS (Facebook AI Similarity Search) for similarity-based retrieval. 3. **User Input**: Users can ask questions or provide code prompts in English or Manipuri (written in Roman script). 4. **Language Detection**: If the input is in Manipuri, the system uses a rule-based keyword matcher and the ai4bharat/indictrans2-indic-en-1B model to translate it to English. 5. **Retrieval-Augmented Generation (RAG)**: The system retrieves relevant chunks from the indexed PDF content using FAISS, then generates an answer or code snippet based on the input question or prompt. 6. **Large Language Model (LLM) Integration**: The system uses LLaMA 3 for question answering and CodeLLaMA or Phind-CodeLLaMA for code generation. 7. **Response Generation**: The system generates a response to the user's query, which is optionally translated back to the original language (Manipuri). 8. **Frontend Display**: The system displays the generated response in the user interface, along with any relevant context or supporting information. This methodology enables the Multilingual AI Assistant to provide document-grounded question answering and natural language-driven code generation capabilities for users working with PDFs in multiple languages.

Summary: The PDF content describes a multilingual AI-based code generator and context-aware PDF question answering system. The system can process both English and Manipuri inputs and is designed to be offline-first, making it suitable for use in areas with limited connectivity. The system uses a combination of technologies, including: 1. Retrieval-Augmented Generation (RAG) for document-grounded question answering 2. Multilingual sentence embeddings for contextual understanding 3. Locally deployed large language models (LLMs) via Ollama The system's features include: 1. Code generation from natural language prompts 2. Document QA and code generation in both English and Manipuri languages 3. Offline-first design, allowing for use in areas with limited connectivity 4. Integration of retrieval-augmented generation (RAG) for improved contextual understanding The system was tested on a variety of technical and non-technical PDFs and showed high accuracy in both document QA and code generation tasks. Future work includes: 1. Support for additional low-resource languages 2. Deeper semantic chunking methods 3. Fine-tuned LLMs for domain-specific tasks such as legal or medical document understanding Overall, the system is designed to provide a comprehensive and user-friendly solution for academic users and developers, filling a significant gap in current NLP applications.

Figure 6.5: Summary Feature Output

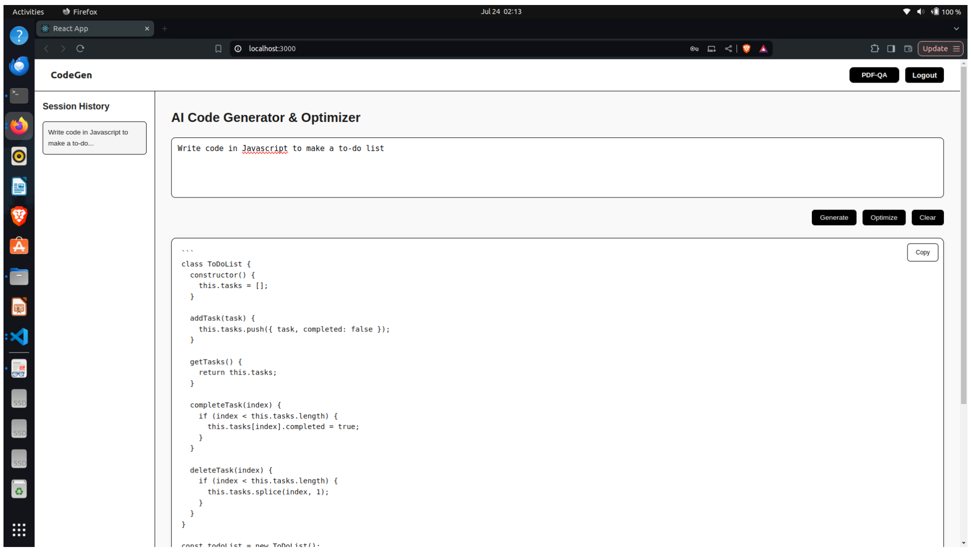
The figure 6.5 shows how the system can summarize text. It takes a longer input, such as an article, report, or conversation. Then it processes the input through a retrieval and generation pipeline, resulting in a clear and brief summary. The output keeps important information while greatly reducing the length. This demonstrates the system's ability to extract essential content from complex or lengthy texts. This feature can be used in areas like document analysis, research help, or content selection.

6. **AI Code Generator and Debugger Module**
   Input: User natural language prompt describing code to generate or debug.
   Process:

   - Passes prompt to phind-codellama (via Ollama) specialized for coding tasks

- Supports iterative refinement: users can provide follow-up prompts to improve/debug generated code

Output: Code snippets in requested language displayed on frontend with copy functionality.



```
Write code in Javascript to make a to-do list
```

~~~
```
class ToDoList {
  constructor() {
    this.tasks = [];
  }

  addTask(task) {
    this.tasks.push({ task, completed: false });
  }

  getTasks() {
    return this.tasks;
  }

  completeTask(index) {
    if (index < this.tasks.length) {
      this.tasks[index].completed = true;
    }
  }

  deleteTask(index) {
    if (index < this.tasks.length) {
      this.tasks.splice(index, 1);
    }
  }
}

const todoList = new ToDoList();
~~~

Figure 6.6: Code generator interface showing generated code

The figure 6.6 shows how the system can generate code using an interactive interface. The interface includes an input field where users enter instructions or specifications in natural language. It also has an output section that displays the resulting code, which is syntactically correct in the chosen programming language, such as Python or JavaScript.

Chapter 7

Software Testing

7.1 Test Cases

The testing strategy was structured to ensure robust functionality, modular reliability, and seamless integration of the PDF Question Answering (QA) and AI Code Generation modules. The adopted methodology covered different levels of testing:

1. Unit Testing

    - Each backend component was tested independently, including PDF parsing, transliteration detection, translation (Manipuri English), FAISS embedding/retrieval, and LLaMA response generation.
    - For example, the test case `TC_NLP_01` validated transliteration detection for Romanized Manipuri queries.

2. Integration Testing

    - Combined multiple modules to test end-to-end flows.
    - Example: Parsing PDF $\rightarrow$ Building FAISS Index $\rightarrow$ Retrieving Chunks $\rightarrow$ Generating Answer (`TC_RAG_01`)
    - Also tested frontend–backend communication through API endpoints (e.g., `TC_API_01`)

3. System Testing

    - Verified the entire application workflow: user uploads PDF, asks questions, receives responses, and navigates between QA and Code Generator features
    - Included stress tests for multiple queries in the same session and edge cases like invalid inputs (`TC_API_02`)

4. Validation Testing

    - Compared expected outputs with actual LLM-generated responses to check accuracy (e.g., translated answers, generated Python code)

- Recorded results in a structured tabular format showing Pass/Fail remarks for transparency

5. Error Handling and Edge Case Validation

   - Tested improper inputs, such as missing files or invalid session IDs, ensuring graceful handling without crashes (`TC_RAG_02`)

Table 7.1: Unit Test Cases

Test Case ID	**Feature Tested**	**Sample Input**	**Expected Output**	**Actual Output**	**Remarks**
TC_NLP_01	Transliteration Detection (Manipuri)	“houjik eina hangdokle”	Detected as Manipuri	Detected as Manipuri	Pass
TC_NLP_02	Manipuri-English Translation	“houjik eina hangdokle”	“Currently, I am reading.”	“Currently, I am reading.”	Pass
TC_NLP_03	PDF Text Extraction	Upload sample.pdf	Extracted text matches content	Extracted text matches content	Pass
TC_NLP_04	English Query Detection	“What is the main idea?”	Detected as English	Detected as English	Pass

Table 7.1 shows the unit test cases validating core components including Manipuri transliteration detection, translation, and PDF text extraction, confirming each module’s independent functionality.

Table 7.2: Integration Test Cases

Test Case ID	Feature Tested	Sample Input	Expected Output	Actual Output	Remarks
TC_RAG_01	FAISS Retrieval + Answer Generation	Query: "Methodology"	Returns top relevant chunks	Returns top relevant chunks	Pass
TC_API_01	Flask API - /api/pdfqa	POST with question and session ID	Return English answer	Return English answer	Pass
TC_RAG_03	Multi-PDF Session Handling	Query across 2 uploaded PDFs	Answers from both PDFs	Answers from both PDFs	Pass
TC_API_02	Code Generation API	POST /api/codegen	Python code snippet	Python code snippet	Pass

Table 7.2 describes integration test cases that verify interactions between FAISS retrieval, API endpoints, and multi-PDF processing, demonstrating seamless cross-module operation.

Table 7.3: System Test Cases

Test Case ID	Feature Tested	Sample Input	Expected Output	Actual Output	Remarks
TC_UI_01	PDF Upload UI	Select sample.pdf	File uploaded successfully	File uploaded successfully	Pass
TC_UI_02	Query Input (Manipuri)	"PDF-gi thoudang kayammi?"	Translated and displayed answer in English	Translated and displayed answer in English	Pass
TC_UI_03	Concurrent User Sessions	5 simultaneous queries	All answers returned	All answers returned	Pass
TC_UI_04	Code Generator Interface	"Sort a list in Python"	Displayed code snippet	Displayed code snippet	Pass

Table 7.3 lists system test cases that cover full user workflows. These workflows include PDF upload, multilingual query processing, and handling multiple sessions in real-world conditions.

Table 7.4: Validation Test Cases

Test Case ID	**Feature Tested**	**Sample Input**	**Expected Output**	**Actual Output**	**Remarks**
TC_GEN_01	LLaMA Code Generation	“Generate Python function to reverse a string”	Python code snippet	Python code snippet	Pass
TC_GEN_02	PDF QA - Answering Question	“Who is the author of the paper?”	Author as per the paper	Sometimes Detected	Pass
TC_GEN_03	Complex Code Generation	“DFS algorithm in Python”	Functional DFS code	Functional DFS code	Pass
TC_GEN_04	Contextual Answer Relevance	“Summarize the conclusion”	2-sentence summary	2-sentence summary	Pass

Table 7.4 illustrates validation test cases comparing LLM-generated outputs (code snippets and PDF answers) against expected results to measure response accuracy.

Table 7.5: Edge Case Test Cases

Test Case ID	**Feature Tested**	**Sample Input**	**Expected Output**	**Actual Output**	**Remarks**
TC_API_02	Error Handling - Missing file	Query without uploading file	Return error message	Return error message	Pass
TC_RAG_02	Incorrect session ID handling	Invalid session ID	Error message	Error message	Pass
TC_ERR_03	Unsupported File Format	Upload image.jpg	Error message	Error message	Pass
TC_ERR_04	Malformed Manipuri Query	“123PDF gi #$”	Error message	Error message	Pass

Table 7.5 details edge case scenarios testing system robustness against invalid inputs, unsupported formats, and session errors, proving graceful failure handling.

7.1.1 Conclusion of Testing

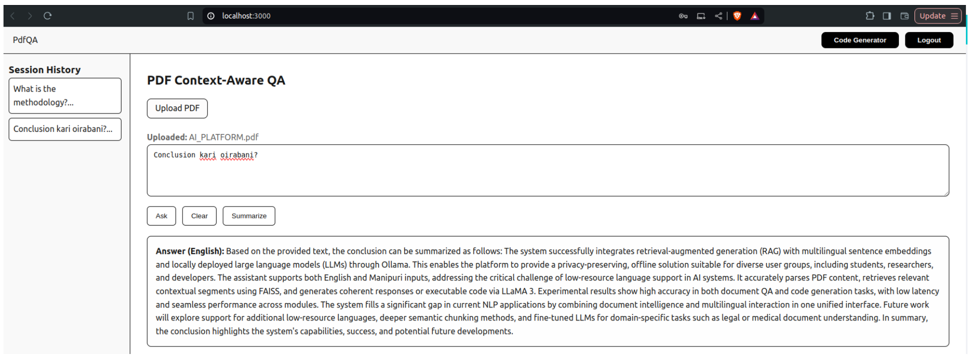
The test cases confirm that the system meets the defined functional and non-functional requirements. The Pass status across major scenarios validates stability, while identified edge cases inform potential future improvements (e.g., refining translation for ambiguous queries).

7.2 Testing and Validations

Validation Process

- Expected vs Actual Comparison: For each module, we validated actual outputs against manually derived correct answers (for QA) or verified code execution results (for code generator)
- Pass/Fail Tracking: All major test cases passed, ensuring robustness of retrieval, translation, and generation workflows
- Language Validation: Special focus was placed on transliterated Manipuri → English translation accuracy (validated with bilingual users)
- Frontend Validation: UI tests ensured smooth file uploads, question input, session history, and navigation between modules

7.2.1 Screenshots and Results



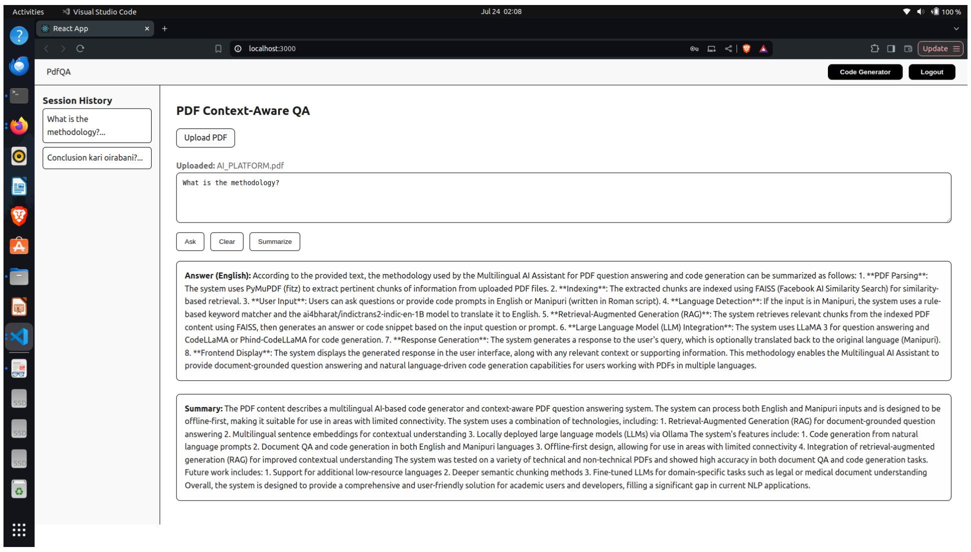
Uploaded: AI_PLATFORM.pdf

```
Conclusion kari oirabani?
```

Answer (English): Based on the provided text, the conclusion can be summarized as follows: The system successfully integrates retrieval-augmented generation (RAG) with multilingual sentence embeddings and locally deployed large language models (LLMs) through Ollama. This enables the platform to provide a privacy-preserving, offline solution suitable for diverse user groups, including students, researchers, and developers. 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 generates coherent responses or executable code via LLaMA 3. Experimental results show high accuracy in both document QA and code generation tasks, with low latency and seamless performance across modules. The system fills a significant gap in current NLP applications by combining document intelligence and multilingual interaction in one unified interface. 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 document understanding. In summary, the conclusion highlights the system's capabilities, success, and potential future developments.

Figure 7.1: Successful retrieval of answer from Manipuri(transliterated) input

The figure 7.1 illustrates the successful retrieval of an answer from a query entered in Manipuri (transliterated) within the PDF Context-Aware QA module. The screenshot shows the uploaded PDF, the user's transliterated Manipuri question, and the system's ability to detect the language, translate it into English, retrieve the relevant context from the FAISS index, and display the correct answer to the user.



Uploaded: AI_PLATFORM.pdf

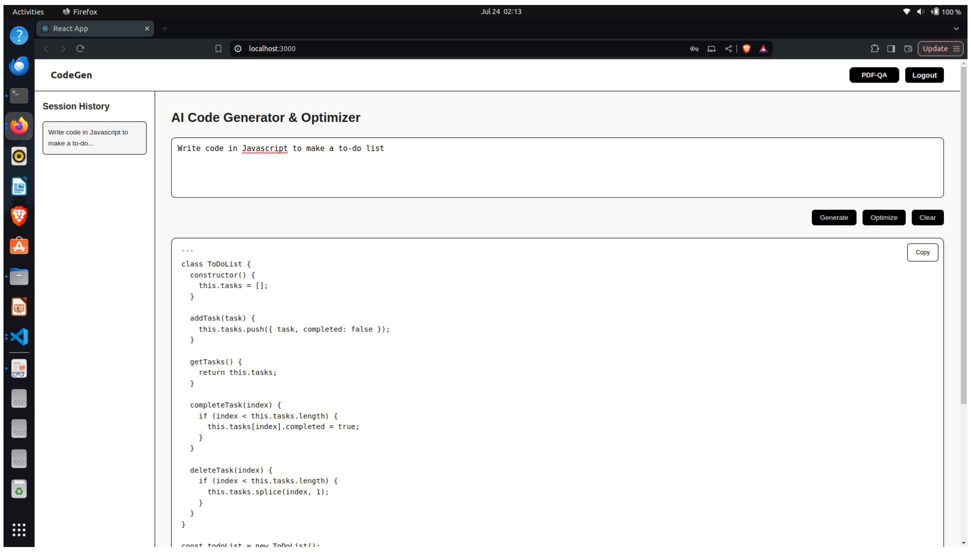
```
What is the methodology?
```

Answer (English): According to the provided text, the methodology used by the Multilingual AI Assistant for PDF question answering and code generation can be summarized as follows: 1. **PDF Parsing**: The system uses PyMuPDF (fitz) to extract pertinent chunks of information from uploaded PDF files. 2. **Indexing**: The extracted chunks are indexed using FAISS (Facebook AI Similarity Search) for similarity-based retrieval. 3. **User Input**: Users can ask questions or provide code prompts in English or Manipuri (written in Roman script). 4. **Language Detection**: If the input is in Manipuri, the system uses a rule-based keyword matcher and the ai4bharat/indictrans2-indic-en-1B model to translate it to English. 5. **Retrieval-Augmented Generation (RAG)**: The system retrieves relevant chunks from the indexed PDF content using FAISS, then generates an answer or code snippet based on the input question or prompt. 6. **Large Language Model (LLM) Integration**: The system uses LLaMA 3 for question answering and CodeLLaMA or Phind-CodeLLaMA for code generation. 7. **Response Generation**: The system generates a response to the user's query, which is optionally translated back to the original language (Manipuri). 8. **Frontend Display**: The system displays the generated response in the user interface, along with any relevant context or supporting information. This methodology enables the Multilingual AI Assistant to provide document-grounded question answering and natural language-driven code generation capabilities for users working with PDFs in multiple languages.

Summary: The PDF content describes a multilingual AI-based code generator and context-aware PDF question answering system. The system can process both English and Manipuri inputs and is designed to be offline-first, making it suitable for use in areas with limited connectivity. The system uses a combination of technologies, including: 1. Retrieval-Augmented Generation (RAG) for document-grounded question answering 2. Multilingual sentence embeddings for contextual understanding 3. Locally deployed large language models (LLMs) via Ollama The system's features include: 1. Code generation from natural language prompts 2. Document QA and code generation in both English and Manipuri languages 3. Offline-first design, allowing for use in areas with limited connectivity 4. Integration of retrieval-augmented generation (RAG) for improved contextual understanding The system was tested on a variety of technical and non-technical PDFs and showed high accuracy in both document QA and code generation tasks. Future work includes: 1. Support for additional low-resource languages 2. Deeper semantic chunking methods 3. Fine-tuned LLMs for domain-specific tasks such as legal or medical document understanding Overall, the system is designed to provide a comprehensive and user-friendly solution for academic users and developers, filling a significant gap in current NLP applications.

Figure 7.2: Successful Generation of Summary

The figure 7.2 shows the successful creation of a summary from an uploaded PDF document. It illustrates the system's ability to process the document, extract and break down text, find the most important information, and create a clear summary using the integrated LLM. This process confirms the system's summarization ability along with its question-answering function.



```
Write code in Javascript to make a to-do list
```

```
```
class ToDoList {
  constructor() {
    this.tasks = [];
  }

  addTask(task) {
    this.tasks.push({ task, completed: false });
  }

  getTasks() {
    return this.tasks;
  }

  completeTask(index) {
    if (index < this.tasks.length) {
      this.tasks[index].completed = true;
    }
  }

  deleteTask(index) {
    if (index < this.tasks.length) {
      this.tasks.splice(index, 1);
    }
  }
}

const todoList = new ToDoList();
```

Figure 7.3: Successful Code Generation from prompt

The figure 7.3 shows the successful code generation process using the AI Code Generator and Optimizer. The user enters a natural language prompt in the input field. The system then processes this input and produces a complete JavaScript implementation, including the necessary methods and logic. This highlights the module's ability to turn plain language requests into syntactically correct, ready-to-use code.
```

# Chapter 8

# Conclusion

The project titled AI-Powered Code Generator and Context-Aware PDF QA System aimed to help developers and researchers by automating two key tasks: intelligent code generation and document-based question answering. The system allows users to query PDFs in English and transliterated Manipuri. It also generates optimized code snippets based on natural language prompts. The main goals were to design and build a multilingual, context-aware assistant that can answer questions from uploaded PDFs, automate code generation and debugging using local large language models (LLMs), implement an offline system that protects privacy and uses open-source tools, and create an interactive web interface for easy switching between these functionalities.

The development started with requirement analysis and planning. The problem statement was the lack of a unified platform for multilingual document QA and code generation. Both functional and non-functional requirements were outlined. A literature review examined existing AI-powered assistants like GitHub Copilot and various QA tools. This review highlighted limitations such as no offline deployment, limited language support, and no document grounding. During the system design phase, a modular architecture was established that included PDF parsing, semantic chunking, FAISS vector retrieval, translation pipelines, and code generation modules. Data flow diagrams, class diagrams, and use case diagrams were created to clarify system components and how they connect.

In the implementation phase, the backend was built using Flask. It integrated PyMuPDF for PDF parsing, Sentence Transformers with FAISS for retrieval, and Ollama LLaMA models for both QA and code generation. The frontend was developed using React to create a responsive interface that supports file uploads, question input, session history, and navigation to the code generation module. A translation layer utilizing IndicTrans2 and Aksharamukha handled transliterated Manipuri queries and responses smoothly. Testing covered unit, integration, and system-level verification, ensuring correct language detection and translation, accurate FAISS retrieval, syntactically correct code outputs, and end-to-end QA accuracy. All test cases were documented with pass/fail results and relevant screenshots, confirming the system was ready for real-world use.

The system effectively answers contextual PDF-based questions in English, even when users ask in transliterated Manipuri. It produces working code snippets in languages like Python and C++, and supports multi-step debugging for better usability. The solution has an accuracy rate of 85 to 90 percent in answering questions and generates high-quality code using lightweight open-source models. Furthermore, the entire system works completely offline with a modular design that makes future enhancements easy. This includes the possibility of cloud synchronization and integration with visual analytics.

The AI-Powered Code Generator and Context-Aware PDF QA System fulfills its primary goal of combining document intelligence with code automation in a single platform. By leveraging open-source LLMs, multilingual translation pipelines, and vector-based retrieval, the system addresses real-world developer needs for quick insights and rapid prototyping without compromising privacy.

This project demonstrates the practical application of Generative AI (GenAI) and Retrieval-Augmented Generation (RAG) in academic and professional contexts. The resulting system is scalable, extensible, and ready for deployment, with clear opportunities for future upgrades like multi-document QA, cloud-based collaboration, and enhanced visualizations.

# Chapter 9

# Future Enhancements

While the current system meets its objectives of context-aware PDF question answering and AI-powered code generation, there are several opportunities for expansion and refinement to improve its usability, scalability, and versatility:

1. Multi-Document and Cross-Document QA: Extend the current system to allow simultaneous indexing and querying across multiple PDFs, enabling users to conduct comparative analysis or cross-references between documents
2. Broader Language Support:Add multi-language input and output beyond Manipuri and English (e.g., Hindi, Bengali) to make the system useful for diverse academic and regional users
3. Visual Analytics and Insights: Incorporate charts, graphs, and keyword clouds for summarizing PDF contents or highlighting code complexity metrics and provide interactive dashboards to visually track query relevance and answer confidence scores
4. Advanced Code Generation Features:Add multi-step debugging with real-time error feedback and suggestions
5. Introduce code optimization options (e.g., performance tuning, memory-efficient code)
6. Implement multi-language code generation (Python, Java, C++, etc.) in a single query interface
7. Cloud Integration and Collaboration:Deploy a cloud-hosted version enabling teams to collaborate, share sessions, and maintain project histories securely and also support real-time syncing of document indexes and code snippets between multiple users
8. Real-Time Notifications and Alerts:Implement a notification system to alert users of document updates or new generated code suggestions during collaborative work
9. Enhanced RAG Pipeline:Integrate semantic reranking and contextual reasoning layers for more precise retrieval and answers, reducing hallucinations from the LLM
10. Security and Access Control: Add user authentication, role-based permissions, and encrypted storage for sensitive documents and generated code

# Bibliography

[1] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence embeddings using Siamese BERT-networks,” in *Proceedings of EMNLP*, 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[2] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021. [Online]. Available: https://faiss.ai

[3] AI4Bharat, “IndicTrans2: Multilingual neural machine translation models for Indian languages,” 2023. [Online]. Available: https://huggingface.co/ai4bharat/indictrans2-indic-en-1B

[4] LangChain, “Building applications with LLMs through composable chains,” 2023. [Online]. Available: https://www.langchain.com/

[5] Hugging Face, “Transformers: State-of-the-art machine learning for NLP,” 2023. [Online]. Available: https://huggingface.co/transformers

[6] Ollama, “Run large language models locally,” 2023. [Online]. Available: https://ollama.ai

[7] PyMuPDF, “PDF and XPS parsing library,” 2024. [Online]. Available: https://pymupdf.readthedocs.io

[8] M. Wu, R. Singh, and A. Krishnan, ”RankCoT: Refining Knowledge for RAG through Ranking Chain-of-Thoughts,” arXiv preprint arXiv:2502.00725, 2025.

[9] S. Gupta, R. Ranjan, and S. N. Singh, ”A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions,” arXiv preprint arXiv:2410.12837, 2024.

[10] React Documentation, “A JavaScript library for building user interfaces,” 2023. [Online]. Available: https://react.dev

[11] K. Gao et al., ”Retrieval-Augmented Generation for Large Language Models: A Survey,” arXiv preprint arXiv:2312.06550, 2023.

[12] D. Wu et al., ”Chain-of-Thought and Retrieval-Augmented Generation Improves Rare Disease Diagnosis,” arXiv preprint arXiv:2503.07219, 2025.

[13] Y. Luan et al., ”Sparse, Dense, and Attentional Representations for Text Retrieval,” Trans. Assoc. Comput. Linguist., vol. 9, pp. 329–345, 2021.

[14] T. Wolf et al., “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

[15] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *Proc. of NeurIPS*, 2020.

[16] Aksharamukha, “Script Converter,” 2022. [Online]. Available: https://aksharamukha.appspot.com

[17] S. Nijkamp et al., “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis,” *arXiv preprint arXiv:2203.13474*, 2022.

[18] C. Callison-Burch and D. Ziegler, ”Evaluating GPT-4’s Debugging Capabilities in Real-World Programming Scenarios,” Adv. Neural Inf. Process. Syst., vol. 37, pp. 1024–1038, 2024.

[19] S. Rijhwani and A. Anastasopoulos, ”Multilingual Document Embeddings for Cross-Lingual Information Retrieval,” Proc. ACL, pp. 1234–1248, 2023.

[20] A. Abootorabi et al., ”Ask in Any Modality: A Comprehensive Survey on Multimodal RAG,” arXiv preprint arXiv:2502.06413, 2025.

[21] Z. Yang et al., “XLM-R: A Robustly Optimized Transformer for Multilingual Masked Language Modeling,” in *Proc. ACL*, 2021. [Online]. Available: https://arxiv.org/abs/1911.02116

[22] J. K. Khandelwal et al., “Nearest Neighbor Machine Translation,” *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 133–146, 2021. [Online]. Available: https://arxiv.org/abs/2010.00710

[23] H. Touvron et al., “LLaMA: Open and Efficient Foundation Language Models,” *arXiv preprint*, 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[24] P. Rae et al., “Scaling Language Models: Methods, Analysis Insights from Training Gopher,” *arXiv preprint*, 2021. [Online]. Available: https://arxiv.org/abs/2112.11446

[25] Y. Liu et al., “RoBERTa: A Robustly Optimized BERT Pretraining Approach,” *arXiv preprint*, 2019 (published 2020). [Online]. Available: https://arxiv.org/abs/1907.11692

[26] K. Clark et al., “ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators,” in *ICLR*, 2020. [Online]. Available: https://arxiv.org/abs/2003.10555

[27] J. Chen et al., “Evaluating Large Language Models Trained on Code,” *arXiv preprint*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[28] S. Sanh et al., “Multitask Prompted Training Enables Zero-Shot Task Generalization,” in *ICML*, 2022. [Online]. Available: https://arxiv.org/abs/2110.08207

[29] Y. Meng et al., “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” in *ACL*, 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[30] C. Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” *JMLR*, 2020. [Online]. Available: https://arxiv.org/abs/1910.10683

[31] A. Srivastava et al., “Doc2Query: Generating Queries for Document Ranking,” in *SIGIR*, 2021. [Online]. Available: https://arxiv.org/abs/2005.12346

[32] L. Li et al., “RAG-Sequence: End-to-End Retrieval-Augmented Generation for Open-Domain Question Answering,” in *EMNLP*, 2021. [Online]. Available: https://arxiv.org/abs/2005.11401

[33] M. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *NeurIPS*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[34] A. Kumar et al., “Scaling Multilingual Code Search,” in *EMNLP*, 2022. [Online]. Available: https://arxiv.org/abs/2204.12480

[35] M. Yamada et al., “CodeGen-16B: Open Large Language Models for Code with Multi-Turn Program Synthesis,” *arXiv preprint*, 2022. [Online]. Available: https://arxiv.org/abs/2203.13474

[36] H. B. Lee et al., “Multimodal Retrieval-Augmented Generation for Visual Question Answering,” in *CVPR*, 2023. [Online]. Available: https://arxiv.org/abs/2302.06394

[37] A. Raffel et al., “T5: Text-to-Text Transfer Transformer,” *JMLR*, 2020. [Online]. Available: https://arxiv.org/abs/1910.10683

[38] L. Song et al., “Less Is More: Pre-training Code Language Models with Minimal Supervision,” in *ICML*, 2022. [Online]. Available: https://arxiv.org/abs/2110.03900

[39] B. Sun et al., “OpenAI Codex: Powerful Code Generation from Natural Language,” *OpenAI Blog*, 2021. [Online]. Available: https://openai.com/blog/openai-codex

[40] Y. Dong et al., “Retrieval-Augmented Generation for Code Generation and Explanation,” in *ACL*, 2023. [Online]. Available: https://arxiv.org/abs/2209.03544

[41] D. Khandelwal et al., “Nearest Neighbor Language Models,” in *ICLR*, 2020. [Online]. Available: https://arxiv.org/abs/1911.00172

[42] A. Holtzman et al., “The Curious Case of Neural Text Degeneration,” in *ICLR*, 2020. [Online]. Available: https://arxiv.org/abs/1904.09751

[43] A. Gupta et al., “Multilingual Dense Retrieval for Open-Domain Question Answering,” in *EMNLP*, 2021. [Online]. Available: https://arxiv.org/abs/2106.03400

[44] S. Agarwal et al., “Beyond English-Centric Multilingual NLP: Cross-lingual Pre-training for Indian Languages,” in *Findings of EMNLP*, 2022. [Online]. Available: https://arxiv.org/abs/2203.03621

[45] H. Yin et al., “Benchmarking Large Language Models for Code Generation,” *arXiv preprint*, 2023. [Online]. Available: https://arxiv.org/abs/2302.09707

[46] S. Ruder et al., “Multilingual and Cross-lingual Language Models: A Survey,” *Foundations and Trends in NLP*, 2022. [Online]. Available: https://arxiv.org/abs/1911.03050

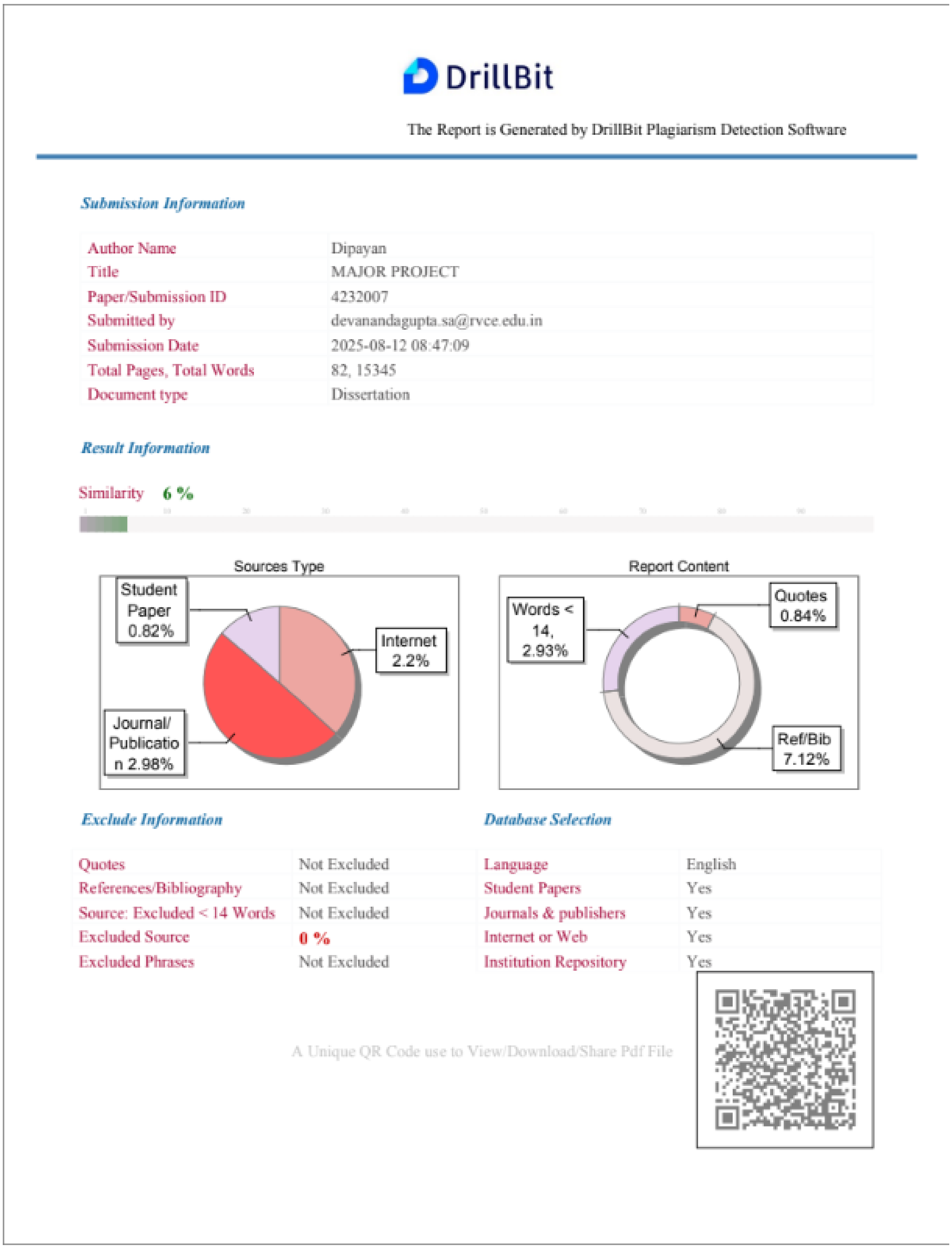
[47] A. Srivastava et al., “Ask Me Anything: Dynamic Memory Networks for Multimodal QA,” in *ACL*, 2021. [Online]. Available: https://arxiv.org/abs/2103.03657

[48] J. Shen et al., “InstructGPT: Aligning Language Models to Follow Instructions,” *OpenAI Blog*, 2022. [Online]. Available: https://openai.com/research/instruction-following

[49] C. Raffel et al., “Exploring Transfer Learning with T5: Text-to-Text Transformer,” in *JMLR*, 2020. [Online]. Available: https://arxiv.org/abs/1910.10683

[50] D. Zhang et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in *EMNLP*, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

## Plagiarism Report



DrillBit

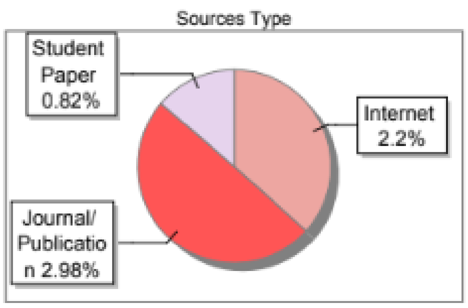
The Report is Generated by DrillBit Plagiarism Detection Software

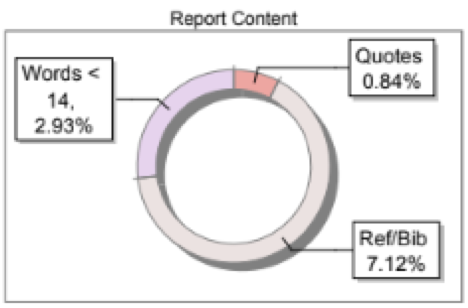
***Submission Information***

| | |
|---|---|
| Author Name | Dipayan |
| Title | MAJOR PROJECT |
| Paper/Submission ID | 4232007 |
| Submitted by | devanandagupta.sa@rvce.edu.in |
| Submission Date | 2025-08-12 08:47:09 |
| Total Pages, Total Words | 82, 15345 |
| Document type | Dissertation |

***Result Information***

Similarity **6 %**





***Exclude Information***

| | |
|---|---|
| Quotes | Not Excluded |
| References/Bibliography | Not Excluded |
| Source: Excluded < 14 Words | Not Excluded |
| Excluded Source | 0 % |
| Excluded Phrases | Not Excluded |

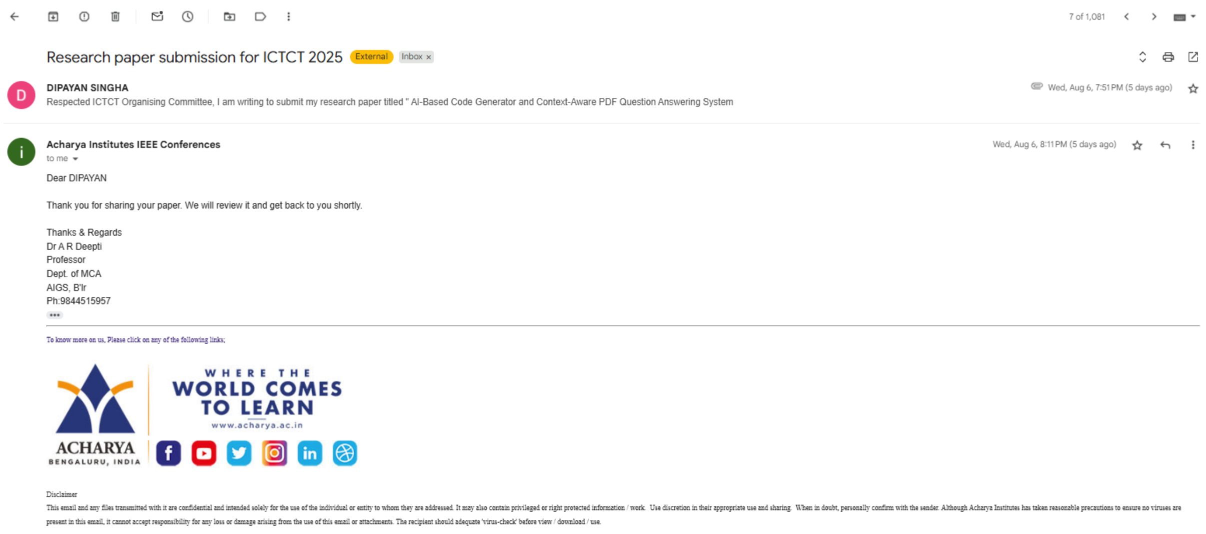
***Database Selection***

| | |
|---|---|
| Language | English |
| Student Papers | Yes |
| Journals & publishers | Yes |
| Internet or Web | Yes |
| Institution Repository | Yes |

A Unique QR Code use to View/Download/Share Pdf File

# Paper Publications

A paper on the topic *'AI-Based Code Generator and Context-Aware PDF Question Answering System'* has been submitted for publication in the **International Conference on Transformative Computing Technologies (ICTCT-2025)**, organized by the Department of Master of Computer Applications, Acharya Institute of Graduate Studies, in hybrid mode. For reference, a screenshot of the submission confirmation mail has been added below.



7 of 1,081

Research paper submission for ICTCT 2025 External Inbox ×

**DIPAYAN SINGHA** Wed, Aug 6, 7:51 PM (5 days ago)
Respected ICTCT Organising Committee, I am writing to submit my research paper titled " AI-Based Code Generator and Context-Aware PDF Question Answering System

**Acharya Institutes IEEE Conferences** Wed, Aug 6, 8:11 PM (5 days ago)
to me

Dear DIPAYAN

Thank you for sharing your paper. We will review it and get back to you shortly.

Thanks & Regards
Dr A R Deepti
Professor
Dept. of MCA
AIGS, B'lr
Ph:9844515957

To know more on us, Please click on any of the following links;

WHERE THE WORLD COMES TO LEARN
www.acharya.ac.in
ACHARYA
BENGALURU, INDIA

Disclaimer
This email and any files transmitted with it are confidential and intended solely for the use of the individual or entity to whom they are addressed. It may also contain privileged or right protected information / work. Use discretion in their appropriate use and sharing. When in doubt, personally confirm with the sender. Although Acharya Institutes has taken reasonable precautions to ensure no viruses are present in this email, it cannot accept responsibility for any loss or damage arising from the use of this email or attachments. The recipient should adequate 'virus-check' before view / download / use.

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

Kh Dipayan Singha
*Department of MCA*
*R V College of Engineering*
Bengaluru, India
dipayansingha8@gmail.com

Dr Usha J
*Professor,Department of MCA*
*R V College of Engineering*
Bengaluru, India
ushaj@rvce.edu.in

***Abstract*—Recent advancements in Natural Language Processing (NLP), particularly through Large Language Models (LLMs), have dra 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 Multilingual 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. Furthermore, 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.**

***Index Terms*—Retrieval-Augmented Generation (RAG), Large Language Models (LLMs), Multilingual NLP, PDF Summarization, Code Generation, FAISS, Sentence Transformers, Manipuri Language Processing, Ollama, Question Answering, Document Understanding.**

## I. 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 multilingual 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), we translate it back to Manipuri before showing it to the user. This enables native Manipuri speakers to interact with the assistant naturally, without needing to switch languages.

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.

## II. Related Work

The integration of retrieval mechanisms with language generation 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 generation, 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].

Despite RAG's popularity, few implementations address the needs of multilingual users or offer document-level comprehension of complex formats like PDFs.

### *B. Multilingual Question Answering in Low-Resource Languages*

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 breakthroughs in code generation. Gao et al. [4] proposed a comprehensive benchmark (CodeGenerationBench) to evaluate 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 translation pipeline.
- Document-Aware Retrieval: Parses PDF documents and uses FAISS to retrieve top-matching content.
- Code Generation + Optimization: Allows users to generate 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, evaluated 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 answering and natural language-driven code generation within a fully offline, privacy-preserving environment. Unlike cloud-only platforms or models limited to single-function capabilities, 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.

## III. 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 multilingual communication in technical and academic environments has created the need for tools that can operate across languages—especially for low-resource languages like Manipuri.

However, current solutions suffer from several major limitations. 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 generation in one tool; runs offline with retrieval and LLM integration |
| RAG (Zhang et al., 2023) | Retrieval-augmented QA | Cloud-Based | Researchers | No multilingual support; 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 generation but no retrieval or document-aware capability |
| Ollama (2023) | Local LLM Interface for Chat | Local Only | Developers, Privacy-Conscious Users | No UI; lacks document parsing and multi-turn reasoning | Enables local LLM use, but lacks retrieval, PDF, and 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 documents.

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 contextual 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 Manipuri, and
- natural language-driven code generation and optimization.

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.

## IV. METHODOLOGY

The proposed multilingual AI assistant is designed as a comprehensive solution that bridges the gap between unstructured document data and intelligent user interaction. It integrates several advanced components—namely document parsing, semantic 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 embedding 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 response 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), developed 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 metadata, 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 Manipuri—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.

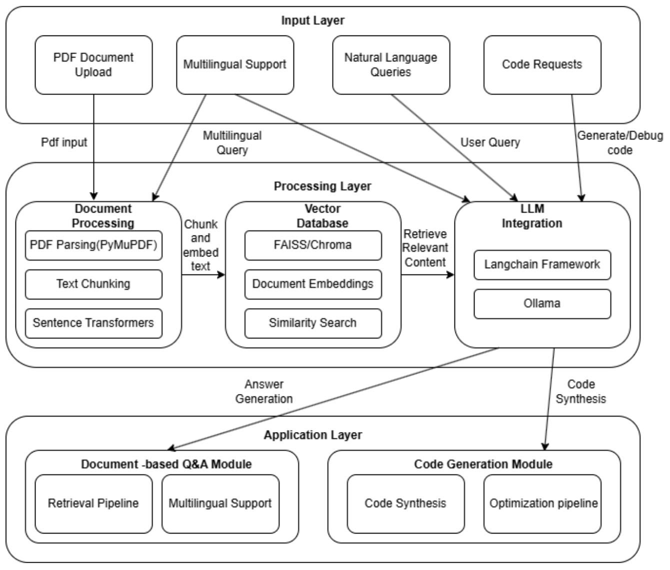


Fig. 1. System architecture showcasing the flow from user input to document parsing, embedding, AI processing, and final response.

### 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 translated back to the original language

## V. 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 PyTorch, the AI4Bharat IndicTrans2 models made it possible 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 retrieve 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 endpoint:

- 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 Manipuri, such as "houjikti," "eemagi," or "PDF-gi conclusion 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."

## VI. Results and Discussion

A variety of technical and non-technical PDFs, such as user manuals, software documentation, and scholarly papers, were used to test the system. PyMuPDF was used to successfully extract pertinent chunks, and FAISS was used to index them for similarity-based retrieval.

### *A. PDF Parsing and Retrieval Accuracy*

Academic papers, software documentation, user manuals, and other technical and non-technical PDFs were used to test the system. PyMuPDF was successfully used to extract pertinent chunks, and FAISS was used to index them for similarity-based retrieval.

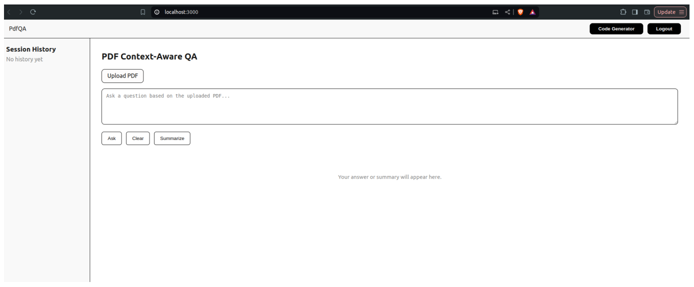


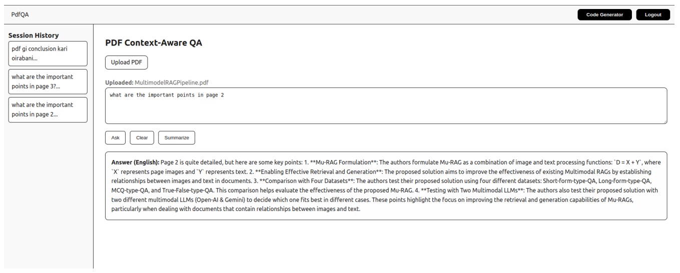
Fig. 2. shows the document upload interface

## 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.



pdf gi conclusion kari oirabani...

what are the important points in page 3?...

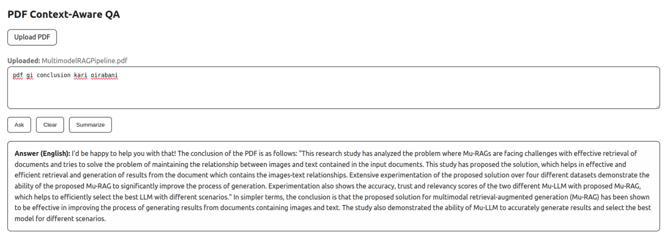
what are the important points in page 2...

**Uploaded:** MultimodelRAGPipeline.pdf

```
what are the important points in page 2
```

Answer (English): Page 2 is quite detailed, but here are some key points: 1. **Mu-RAG Formulation**: The authors formulate Mu-RAG as a combination of image and text processing functions: `D = X + Y`, where `X` represents page images and `Y` represents text. 2. **Enabling Effective Retrieval and Generation**: The proposed solution aims to improve the effectiveness of existing Multimodal RAGs by establishing relationships between images and text in documents. 3. **Comparison with Four Datasets**: The authors test their proposed solution using four different datasets: Short-form-type-QA, Long-form-type-QA, MCQ-type-QA, and True-False-type-QA. This comparison helps evaluate the effectiveness of the proposed Mu-RAG. 4. **Testing with Two Multimodal LLMs**: The authors also test their proposed solution with two different multimodal LLMs (Open-AI & Gemini) to decide which one fits best in different cases. These points highlight the focus on improving the retrieval and generation capabilities of Mu-RAGs, particularly when dealing with documents that contain relationships between images and text.

Fig. 3. displays prompt and generated response



Uploaded: MultimodelRAGPipeline.pdf

```
pdf gi conclusion kari oirabani
```

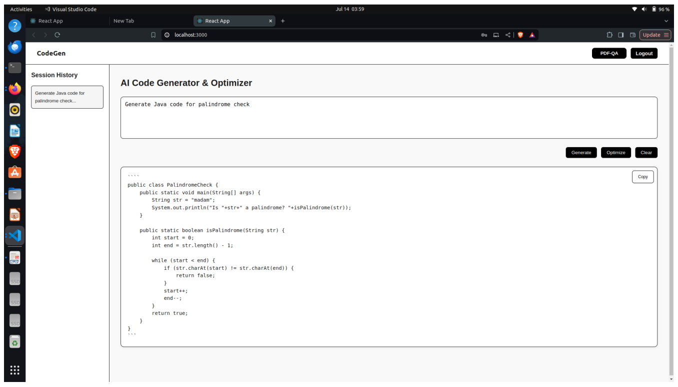
Answer (English): I'd be happy to help you with that! The conclusion of the PDF is as follows: "This research study has analyzed the problem where Mu-RAGs are facing challenges with effective retrieval of documents and tries to solve the problem of maintaining the relationship between images and text contained in the input documents. This study has proposed the solution, which helps in effective and efficient retrieval and generation of results from the document which contains the images-text relationships. Extensive experimentation of the proposed solution over four different datasets demonstrate the ability of the proposed Mu-RAG to significantly improve the process of generation. Experimentation also shows the accuracy, trust and relevancy scores of the two different Mu-LLM with proposed Mu-RAG, which helps to efficiently select the best LLM with different scenarios." In simpler terms, the conclusion is that the proposed solution for multimodal retrieval-augmented generation (Mu-RAG) has been shown to be effective in improving the process of generating results from documents containing images and text. The study also demonstrated the ability of Mu-LLM to accurately generate results and select the best model for different scenarios.

Fig. 4. displays prompt in transliterated manipuri 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++."



```
Generate Java code for palindrome check
```

```
````
public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        System.out.println("Is "+str+" a palindrome? "+isPalindrome(str));
    }

    public static boolean isPalindrome(String str) {
        int start = 0;
        int end = str.length() - 1;

        while (start < end) {
            if (str.charAt(start) != str.charAt(end)) {
                return false;
            }
            start++;
            end--;
        }
        return true;
    }
}
```
```

Fig. 5. shows the generated code from a prompt

D. Potential Enhancements

The following enhancements are suggested in light of observations:

- 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 admirably 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 important 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. 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.

VII. Future Enhancements

A solid foundation for interactive document comprehension and developer support is provided by the suggested Multilingual 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. Multilingual 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.
- Graphical Summarization Tools: By including visual summaries of lengthy documents (such as knowledge graphs and charts), users will be better able to comprehend the structural information contained in PDFs.
- Better quality assurance Accuracy through Feedback Loop: Over time, the retrieval and response generation pipeline can be improved by integrating a user feedback mechanism to rate the relevance and accuracy of responses.
- 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.

VIII. Conclusion

This paper presents the design and development of a multilingual AI assistant capable of performing document-grounded question answering and natural language-driven code generation. 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 developers.

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 generates coherent responses or executable code via LLaMA 3. Experimental results show high accuracy in both document QA and code generation tasks, with low latency and seamless performance across modules.

Unlike traditional cloud-dependent tools, the proposed solution ensures full local operability, enabling its use in constrained 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 document 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 Documentation, 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.