Advances in Consumer Research
https://acr-journal.com/
Volume-3 | Issue-2 | Feb 2026
Original Researcher Article

Multi-Modal Rag System

Santosh S Pati[1]*, Adarsha[2], Abhimanyu[3]

[1]Department of Studies in Computer Science, VSK University , Ballari,
Orcid ID: 0000-0001-8894-5639
Email:ID santoshpatil89@gmail.com
[2]Department of Studies in Computer Science, VSK University , Ballari,
Email:ID: adarshhadli96@gmail.com
[3]Department of Studies in Computer Science, VSK University , Ballari,
Email:ID: iamabhimanyusharma12@gmail.com

ABSTRACT
While advances in prompt engineering and RAG have improved LLM proficiency in field-specific, specialized tasks, there has yet to be an industry standard or accepted evaluation metric of the highly fragmented RAG solutions that are currently being deployed. Thus, in this work, we focused on building a robust LLM and RAG evaluation platform. We contribute 1) a platform that evaluates an RAG system's performance on a multimodal input context for LLM question answering and 2) MRAFE: Multimodal Retrieval Augmented Feature Extractor, which processes information from the input to our platform. Through various automated and systematic hand testing, we find that our evaluation benchmarks are useful in determining noise robustness, negative rejection, information integration, and counterfactual rejection. Such a platform would serve as a useful tool for developers iterating on retrieval systems and regulatory bodies creating AI-focused governance alike.
Keywords: N/A

INTRODUCTION:

The **Multi-Modal RAG System** is an excellent end-to-end application designed to handle text**,** audio**,** image**,** and document-based queries using the **Retrieval-Augmented Generation (RAG)** and local large-language models **(**LLMs**)** such as Ollama's Llama3- latest, Llama-3.2-Vision, and Nomic-Embed-Text.

This system integrates multi-modal input understanding, semantic retrieval, and context-aware generative reasoning to provide accurate, explainable answers all while keeping data private and locally processed. The architecture integrates a FastAPI backend for LLM orchestration, retrieval, and embedding management, and a Streamlit frontend for an interactive, user-friendly interface that supports document upload, audio transcription, image based OCR, and real-time chat streaming.

Key Features:

Multi-Modal Intelligence

Text: Natural language Q&A through contextual retrieval.

Documents (PDF, DOCX, TXT, CSV) → automatically chunked, vectorised, and indexed for contextual retrieval.

Images → Text extracted using Tesseract OCR for visual Q&A.

Audio Files → Transcribed via Open-AI Whisper before generating responses.

Retrieval-Augmented Generation (RAG) Pipeline

Combines dense (FAISS + Sentence Transformers) and sparse **OKAPI Best Matching25** (BM25) retrieval for hybrid relevance search.

Retrieved context dynamically augments the prompt to improve accuracy and reduce hallucinations.

Supports incremental knowledge base updates via the /add-to-kb route.

Local LLM Orchestration via Ollama

Fully offline inference using Ollama models:

llama3:latest — Text reasoning.

llama3.2-vision — Vision understanding.

nomic-embed-text — Embeddings for retrieval.

Streamed token-wise responses (/api/chat-stream) for real-time UI feedback.

Comprehensive Knowledge Management

Upload any document to build a custom vector knowledge base.

Text is chunked, embedded, and stored locally for privacy.

Search and summarize knowledge using RAG-driven question answering.

Audio & Speech Processing

Upload audio and automatically transcribe it with Whisper.

Converts user speech to text

Interactive Streamlit Frontend

Built with Streamlit for real-time user interaction.

Dedicated spaces for:

Chat, OCR Image Extraction, Audio Transcription

Analytics & Logging

Every session and interaction logged in for usage tracking.

/api/analytics/* endpoints provide:

Request counts

Average latency

Query timelines

Privacy-First, Local-Capable

All processing including embedding, transcription, and generation occurs locally.

No external API calls required, ensuring complete data sovereignty.

Objectives

Unified Multi-Modal Intelligence

Integrate **text**, **image**, **audio**, and **document** inputs into one consistent reasoning framework.

Allow users to **ask questions from any modality** — whether it's a research PDF, spoken query, or uploaded image.

Use **modality-specific extractors** (OCR, Whisper, embedding models) that convert all data into a **shared semantic space** for retrieval.

Retrieval-Augmented Generation (RAG) Foundation

Use **FAISS** or **vector databases** for semantic retrieval of document chunks.

Embed user data using **sentence transformers / CLIP** for hybrid text–image search.

Construct **context-aware prompts** dynamically and feed them to **local LLMs (Ollama)**. Achieve **grounded responses** — AI answers are always backed by retrieved evidence.

Local & Privacy-Preserving AI

Operate entirely using **local models via Ollama** (e.g., llama3, llava, nomic- embed-text).

Prevent data leakage by avoiding cloud APIs — ideal for **enterprise or sensitive domains**.

Maintain control over data flow and computation for **privacy, security, and compliance**.

Human-Like Conversational Interface

Stream real-time model outputs token-by-token using FastAPI + SSE.

Present dynamic chat updates in **Streamlit** with clean, conversational bubbles.

Allow multi-turn dialogue that preserves session context.

Display past conversations and enable saving/exporting of session logs.

Knowledge Base Augmentation

Allow users to upload PDFs, DOCX, TXT, or CSV files.

Automatically extract, chunk, and embed text for future retrieval.

Provide **query-over-documents (QOD)** functionality — ask questions directly about uploaded content.

Support continuous KB growth: "the more you use, sthe smarter it becomes."

Multi-Modal Understanding

Goal: Add domain-specific perception across different input types.

Fast, Modular, and Extensible Architecture

Goal: Make the system developer-friendly and easily extensible for future AI models or modalities.

Modular backend: every capability (OCR, Audio, RAG, Inference) is a separate FastAPI router.

Scalable structure with independent components (core modules + routes).

Plug-and-play architecture — new models or pipelines can be added without breaking others.

Support **local, hybrid, and cloud inference** with minimal configuration changes.

Transparency, Traceability & Analytics

Goal: Log, analyse, and visualize model interactions for quality improvement.

Maintain SQLite analytics database for:

Query latency

Token usage

Session timeline

Track performance and accuracy over time.

Provide a backend /analytics API for dashboard integration.

Developer & Research Enablement

Goal: Serve as a flexible R&D sandbox for AI engineers, researchers, and developers.

Enable rapid experimentation with different embedding, RAG, and generation models.

Provide structured APIs for connecting external LLMs, knowledge graphs, or multi-agent systems.

Support both **academic use cases (AI research)** and **enterprise deployments (internal QA assistants)**.

Foundation for Agentic AI

Goal: Lay the groundwork for building self-improving, autonomous AI agents.

Enable **memory-augmented reasoning**: retain prior interactions for better responses.

Integrate **feedback loops** to let the model self-optimize from user corrections.

Support future multi-agent collaboration (Analyst, Researcher, Tester roles).

Move toward a **self-learning cognitive assistant** that evolves over time.

SYSTEM ANALYSIS

Existing System

The existing systems for question-answering or AI assistants are generally limited to a single data modality (usually text).Most conventional chatbots or QA systems rely on predefined rules, static datasets, or single-model language inference, making them inflexible and domain-specific.

Examples of Existing Systems

Rule-Based Chatbots: Use keyword matching and scripted responses (e.g., simple FAQ bots).

Text-Based QA Systems: Handle only text input and rely on external APIs (e.g., Chat-GPT, Bard, Bing Chat).

Separate Tools for Each Modality:

OCR applications for images,

Speech-to-text software for audio,

Document readers for text — all functioning independently.

Key Characteristics

Mono-modal (supports only one input type)

API-dependent (requires cloud connectivity)

Non-adaptive (no long-term memory or learning)

Lacks transparency in information sources

Problems on Existing System

Dependency on Cloud Models: Most systems (e.g., Chat-GPT, Gemini) depend on cloud-based LLM APIs, raising privacy**,** latency**,** and cost concerns.

Data Privacy Risks: Sensitive enterprise or healthcare data cannot be safely uploaded to external APIs.

No Persistent Knowledge Base: Uploaded documents or past queries are not remembered across sessions.

Fragmented Tools: Separate applications for OCR, transcription, and summarization increase workflow complexity.

No Offline Capability: Internet dependency prevents use in secure or restricted environments (e.g., research labs, hospitals).

High Resource Cost: Using multiple commercial APIs leads to expensive token usage and maintenance overhead.

Proposed System

The **Proposed Multi-Modal RAG System** addresses all the above challenges by integrating Retrieval-Augmented Generation (RAG) with multi-modal processing — within a local, privacy-preserving framework.

The proposed solution is a **unified AI assistant** that can:

Ingest, understand, and answer questions based on **documents, images, audio, and text**.

Retrieve factual context from a **local knowledge base (RAG pipeline)**.

Generate contextual, explainable responses using **local LLMs** via **Ollama**.

Modality	**Objective**	**Core Tool**
Text	Understand user queries, retrieve context, generate answers	Llama3 / Ollama
Image	Extract and reason about visual content	Tesseract OCR / CLIP / LLaVA
Audio	Convert speech to text and transcribe recordings	Whisper
File	Extract structured/unstructured knowledge	PyPDF2, docx, pandas

Key Features and Functional Flow:

Component	**Description**
RAG Engine	Retrieves relevant document chunks using **FAISS embeddings** and combines them with the query for **context-grounded responses**.
Image Understanding (OCR)	Extracts text from uploaded images using **Tesseract OCR** and indexes it into the knowledge base.
Audio Processing (Whisper)	Converts speech or recorded audio into text to enable **voice-based queries**.
Document Loader	Supports file formats like PDF, DOCX, TXT and CSV. Automatically extracts and chunking content.
Chat Interface (Streamlit)	Interactive conversational UI allowing real-time streaming responses and conversation history.

Local Execution	Entire system runs on a local machine using **FastAPI + Ollama**, ensuring **data privacy**.
Extensible Architecture	Modular design allows easy integration of new models or modalities (e.g., TTS).

Advantages

The proposed Multi-Modal Retrieval-Augmented Generation (RAG) System offers significant improvements over traditional chatbots and single-modal AI systems.

It integrates **text, image, and audio** processing in a unified, privacy-preserving environment powered by **local models (Ollama, Whisper, and Tesseract OCR)**.

The system ensures efficiency**,** flexibility**,** accuracy**,** and data security for users and organizations.

Multi-Modal Intelligence

Unlike conventional systems that process only text, the proposed system handles three different input modalities:

Documents (PDF**,** DOCX**,** TXT**,** CSV**)** → text extraction and contextual retrieval

Images → OCR-based text extraction

Audio → Speech-to-text transcription

Retrieval-Augmented Generation (RAG) Capability

The system leverages a RAG pipeline combining retrieval and generation:

Retrieves relevant chunks from the local knowledge base using FAISS embeddings.

Generates contextually accurate answers grounded in real data.

Ensures factual, source-aware, and explainable responses instead of hallucinations often seen in generic LLMs.

Privacy and Security

The system runs completely offline, using locally hosted models through Ollama. No user data, documents, or audio is transmitted to cloud APIs. Guarantees data confidentiality, making it ideal for research institutions, hospitals, and enterprises that handle sensitive information.

Unified Platform

Instead of using separate tools for OCR, speech transcription, and question-answering, the proposed system offers an integrated environment. Provides all functionalities — chat**,** upload**,** transcribe**,** extract**,** summarize — in a single interface using Streamlit.

Fast and Lightweight Execution

By using:

SQLite for lightweight storage,

FAISS for high-speed similarity search,

Sentence Transformers for compact embeddings, and

Local LLMs (like Llama 3 via Ollama) for generation

The system performs rapid local inference without cloud latency or subscription costs.

Cost-Effective and Open Source

No dependency on expensive cloud APIs (like OpenAI or Google Gemini). All tools used — FastAPI**,** FAISS**,** Whisper**,** Ollama**,** Tesseract**,** Streamlit — are open-source. This ensures zero recurring **cost**, allowing even individuals or institutions with limited budgets to deploy the solution.

Explainability and Transparency

The RAG pipeline retrieves and displays context chunks used by the LLM to generate each answer. Enables traceability — users can see where the model's response originated from. This fosters trust and interpretability, especially for academic or enterprise use.

Streamlit-Based Interactive UI

The frontend is built using Streamlit, offering:

Responsive layout,

Real-time streaming responses,

File and audio upload support,

Dark-themed styling and minimal white spaces.

Provides a modern**,** user-friendly interface for both technical and non-technical users.

Scalable and Future-Ready

The architecture supports easy scaling for:

Multi-user sessions

Distributed knowledge base

GPU acceleration

Future versions can integrate:

Vision-Language Models (e.g., LLaVA)

Multi-Agent Collaboration

Real-time Speech-to-Speech AI assistants

Feasibility Study

The feasibility study determines whether the proposed **Multi-Modal RAG System** can be successfully developed, implemented, and maintained in a real-world environment. It evaluates various factors — operational, technical, scheduling, cultural, and economic — to ensure the project's overall viability and sustainability. There is no need to go into the detailed system operation yet. The solution should provide enough information to make reasonable estimates about project cost and give users an indication of how the new system will fit into the organization. It is important not to exert considerable effort at this stage only to find out that the project is not worthwhile or that there is a need significantly change the original goal. Feasibility of a new system means ensuring that the new system, which we are going to implement, is efficient and affordable. There are various types of feasibility to be determined.

Operation Feasibility

Operational feasibility evaluates how effectively the system will operate once implemented and how well it integrates with existing workflows.

The proposed system provides an intuitive Streamlit-based UI, allowing users to interact with the RAG model seamlessly through text, audio, image, and document inputs. The backend, powered by **FastAPI**, ensures efficient real-time data flow, providing a smooth and responsive user experience.

By integrating Ollama local models, **FAISS** retrieval, and **Whisper** transcription, the system reduces dependency on cloud APIs, improving data privacy and control. Since the system can be run locally or on-premise, it can be easily adopted by research institutions, developers, and organizations requiring secure AI-assisted Q&A.

Technical Feasibility

Technical feasibility focuses on the system's ability to be developed with the available technology stack and tools.

The system is implemented using Python 3.11.9, ensuring compatibility with modern libraries and frameworks.

The backend uses FastAPI, known for high performance, asynchronous capabilities, and scalability.

The frontend uses Streamlit, providing rapid UI development with minimal effort.

Ollama models (Llama3, Llama3.2-Vision) are used for text and image reasoning.

FAISS (Facebook AI Similarity Search) is implemented for efficient vector- based retrieval.

Whisper is used for audio-to-text transcription, while Tesseract OCR extracts text from images.

Scheduled Feasibility

Scheduled feasibility evaluates the time required to develop and implement the system and whether it can be completed within the planned timeline. The modular nature of the project (Frontend, Backend, and Core Engine) allows for parallel development and testing. Streamlit's lightweight framework and FastAPI's rapid deployment capabilities reduce development time. Local models eliminate dependency on cloud service delays, speeding up the testing and validation cycle. The knowledge base and embedding manager can be incrementally extended, enabling phased releases and quick prototyping.

Major Tasks	W1	W2	W3	W4	W5	W6	W7	W8	Milestones
Requirement Analysis	■								Requirement Report
System Design & Diagram Creation		■	■						Design Sign-off
Backend & Core Module Development			■	■					Backend API Ready
Database & RAG Integration					■	■			Retrieval System Ready
Streamlit Frontend UI Development						■	■		Functional Frontend Ready
Testing & Optimization							■		Stable Build
Documentation & Presentation								■	Final Project Release

Cultural Feasibility

Cultural feasibility assesses whether the system aligns with the users' and organization's culture, ethics, and values.

The system promotes open-source AI development and responsible model usage.

It encourages local, privacy-preserving deployments rather than cloud dependence, fostering trust and transparency.

The multi-modal interaction design (text, voice, image, documents) supports inclusive accessibility, catering to users with varied input preferences.

The user interface is simple and language-agnostic, allowing adoption in diverse

educational and research environments.

Economic Feasibility

Economic feasibility determines whether the system's benefits outweigh the costs of development, deployment, and maintenance.

The system leverages **entirely open-source tools** (FastAPI, Streamlit, FAISS, Whisper, Tesseract, Ollama), eliminating licensing costs.

Local deployment minimizes recurring cloud API costs.

Hardware requirements (CPU/GPU) can scale from **low-cost setups** to **high- performance servers**.

Maintenance and model upgrades are manageable within the existing open-source ecosystem.

The productivity improvement and efficiency gains from an AI-powered assistant justify its low setup cost.

SYSTEM SPECIFICATION

Hardware Requirements

Processor (CPU)

Minimum Requirement: Intel i5 (6th Gen) / AMD Ryzen 3

Description / Purpose: Required for handling concurrent FastAPI requests and background RAG computations.

Memory (RAM)

Minimum Requirement: 8 GB

Description / Purpose: Needed for model loading (FAISS, Whisper) and parallel processing of embeddings and responses.

Storage

Minimum Requirement: 20 GB HDD/SSD

Description / Purpose: For storing embeddings, user uploads, SQLite DB, and logs.

GPU (Optional)

Minimum Requirement: NVIDIA GTX 1050 / 2 GB VRAM

Description / Purpose: Enhances Ollama model inference and Whisper transcription performance.

Audio Device

Minimum Requirement: Standard Microphone

Description / Purpose: For audio recording and live transcription

Software Requirements

Category	Software / Tool	Version	Purpose / Description
Operating System	Windows 10 / 11 (64-bit), Ubuntu 22.04+	Latest	Supports Python and all required dependencies.
Programming	Python	3.11.9^	Core language
Language			used for backend, frontend, and AI integrations.
Backend Framework	FastAPI	0.118.3	Handles API routing for text, audio, OCR, summarization, and RAG.
Frontend Framework	Streamlit	1.39+	Provides interactive user interface for multimodal Q&A.
AI Model Engine	Ollama	0.1.9+	Hosts and runs local LLMs such as Llama3 and Llama3.2-Vision.
Language Models	Llama3, Llama3.2-Vision, nomic-embed- text	Latest	Used for text understanding, vision reasoning, and embeddings.
Vector Database	FAISS (Facebook AI Similarity Search)	1.12.0	Stores and retrieves text embeddings efficiently.
Embedding Framework	SentenceTransformers	2.2.2	Generates semantic embeddings for textual data.
Audio Transcription	OpenAI Whisper	20230615+	Converts audio input into text form for RAG processing.
Optical Character Recognition (OCR)	Tesseract OCR	0.3.13	Extracts text content from image files (JPG, PNG).
Document Parsing	PyPDF2, python- docx, pandas	Latest	Extracts text from documents (PDF, DOCX, CSV).
Database	SQLite	Built-in	Stores analytics, sessions, and chat logs locally.
Package Manager	pip	Latest	Used for Python dependency installation.
Other	numpy, torch, transformers,	Latest	Supports model computation,

Libraries	rank- bm25, moviepy		tokenization, and video/audio processing.

PROJECT DESCRIPTION

About Project

The Multi-Modal RAG System is designed to allow users to ask questions and obtain contextually accurate answers derived from a custom knowledge base built from uploaded documents, images, and audio files. It leverages retrieval-augmented generation (RAG), combining semantic search (via FAISS and embeddings) with generative reasoning (via Ollama models such as Llama3 and Llama3.2-Vision). The project provides a secure, local AI environment where users can interact through a simple Streamlit user interface, without sending data to external cloud services — ensuring privacy, control, and transparency.

The system supports:

Uploading knowledge base documents for domain-specific Q&A.

Extracting text from images using **Tesseract OCR**.

Converting speech into text using **Whisper audio transcription**.

Engaging in text-based conversations with the integrated **Ollama local LLMs**.

Summarization, context retrieval, and knowledge enrichment using **RAG principles**.

Problem Definition

In many real-world applications — such as research, healthcare, law, and education — users often struggle to extract meaningful answers from large, unstructured data sources like PDFs, lecture notes, audio discussions, or image-based content. Conventional Q&A systems and search engines often fail to combine multimodal data effectively, and cloud-based AI services raise **privacy, cost, and dependency issues**.

Key Problems in Existing Systems:

Limited Modality Support: Most AI Q&A tools only support text input, ignoring audio or image-based information.

Cloud Dependency: Existing systems rely on external APIs (OpenAI, Google Cloud, etc.), risking data privacy and requiring continuous internet access.

Fragmented Workflows: Users must use separate tools for OCR, transcription, and summarization.

High Cost & Maintenance: Paid API usage and model access fees make them unsuitable for long-term academic or enterprise use.

Lack of Knowledge Retention: Current chatbots cannot remember or reuse user- specific document data locally.

Core Problem Statement:

"To design and implement a **locally deployable multi-modal AI assistant** that can process and understand **text, documents, audio, and images**, retrieve the most relevant information from a custom knowledge base, and generate accurate, explainable responses using retrieval-augmented generation."

Project Overview

The **Multi-Modal RAG System** aims to integrate multiple AI capabilities into one cohesive application that enhances human–computer interaction and knowledge discovery. It bridges the gap between document understanding, speech recognition, and visual text extraction, creating a truly multi-modal experience.

Key Components:

Frontend (Streamlit UI):

Provides an interactive chat interface.

Allows uploading of documents, images, and audio files.

Displays real-time streaming responses from the backend.

Backend (FastAPI):

Handles requests from the frontend.

Routes them to the appropriate module (Text, OCR, Audio, or RAG engine).

Manages communication with the **Ollama LLM** models locally.

Core Engine:

Includes the **Embedding Manager**, **RAG Engine**, **OCR Extractor**, **Audio Transcriber**, and **Text Extractor** modules.

Uses **FAISS** and **SentenceTransformers** for semantic similarity search.

Applies **RAG** methodology to combine retrieved knowledge with generative answers.

Knowledge Base (Local Storage):

Stores embeddings, documents, and extracted data for persistent use.

Allows incremental updates without retraining the model.

Database (SQLite):

Logs analytics such as latency, tokens, and user sessions.

Enables viewing and exporting of historical Q&A sessions.

System Highlights:

Multi-Modality: Supports text, images, and audio inputs.

Local Deployment: Runs entirely on a user's machine using **Ollama** and open- source libraries.

Retrieval-Augmented Generation (RAG): Combines information retrieval with generative reasoning.

Real-Time Streaming: Displays token-level AI responses in real time.

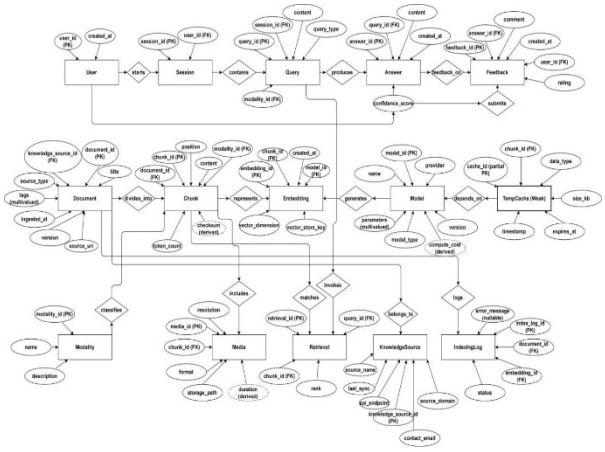
Data Privacy: No external API or cloud data transmission.

Scalability: Supports future integration of more modalities or model types.

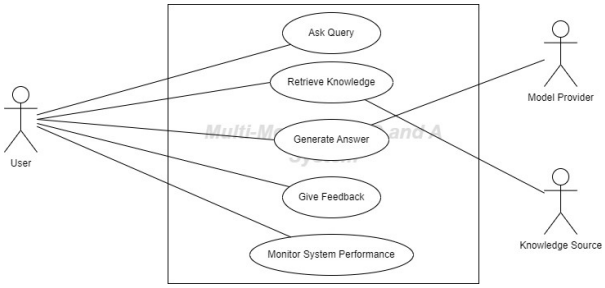
REGISTRATION MODAL DESCRIPTION

Entity Relationship Diagram (Er-Diagram)

ER Diagram

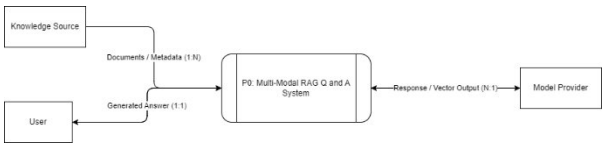


Use Case Diagram

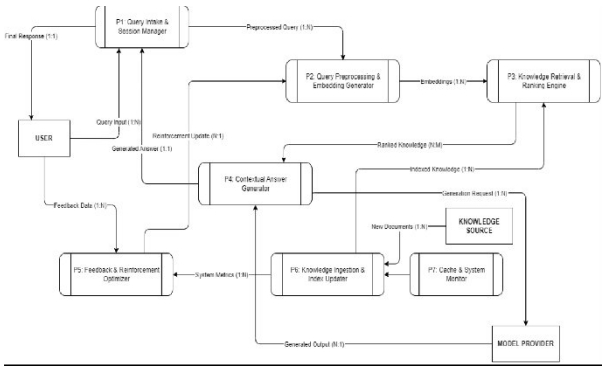


Data Flow Diagram

Level-0



Level-1



SYSTEM IMPLEMENTATION

The **System Implementation** phase involves the transformation of the designed architecture into an operational, fully functional application. In this phase, the components of the Multi- Modal RAG System — including the backend services, AI model integrations, and Streamlit- based user interface — are developed, integrated, and tested for real-world usability. Implementation ensures that the designed functionalities such as knowledge retrieval, OCR extraction, audio transcription, and LLM-based question answering operate cohesively in a local, offline environment. Each module is built using open-source frameworks that emphasize modularity, maintainability, and scalability.

Purpose

The main purpose of implementing the **Multi-Modal RAG System** is to create a fully operational AI-driven assistant capable of understanding and generating responses from multiple data sources — text, images, and audio — without relying on external cloud APIs.

The implementation serves several key objectives:

	Description
1. Realization of the Design	To convert the system design and architecture into an executable software system with complete backend and frontend integration.
2. Functional Verification	To validate the working of all modules such as file upload, OCR, transcription, and retrieval against functional requirements.
3. Integration of AI Components	To integrate Whisper, Tesseract, FAISS, and Ollama LLMs into a unified multimodal retrieval system.
4. Local Model Execution	To enable LLM inference locally using Ollama, ensuring data privacy and offline usability.
5. User Interface Development	To design a simple, responsive Streamlit UI for user interaction and real-time chat streaming.
6. Knowledge Base Management	To implement a system where users can upload documents, store embeddings, and query them dynamically.
7. Testing and Optimization	To ensure the system meets performance benchmarks with efficient data processing and retrieval time.

Implementation Approach

Modular Coding:

Each functional component (OCR, Audio, Chat, KB Upload) was implemented as a separate FastAPI route.

Streamlit frontend interacts with these endpoints dynamically via REST APIs.

Backend Workflow:

FastAPI routes process user input (file, text, image, or audio).

Extracted or transcribed text is passed to the **RAG Engine**, which retrieves relevant context using **FAISS embeddings** and **BM25**.

Context is then combined with the user's question and sent to **Ollama's local LLM** for response generation.

Frontend Workflow:

Built using **Streamlit**, providing real-time feedback through progress spinners and token-wise streaming output.

Supports uploading and interacting with multiple input modalities in a clean, minimal UI.

Integration & Testing:

Continuous integration testing ensured API compatibility between backend and frontend.

Unit and functional testing validated module correctness (OCR, RAG retrieval, Whisper transcription).

Deployment:

The system is locally deployable using:

```
uvicorn backend.app:app --reload --host 127.0.0.1 --port 8000

streamlit run app_ui.py
```

System Maintenance

After implementation, the system enters a **maintenance phase**, ensuring consistent operation, compatibility, and scalability over time.Maintenance involves updating models, libraries, and configurations to keep the system reliable and up-to-date with evolving AI technologies.

Types of Maintenance Activities:

Maintenance Type	Description / Activities
1. Corrective Maintenance	Fixing bugs or runtime issues such as API mismatches, missing dependencies, or incorrect OCR outputs.
2. Adaptive Maintenance	Updating system configuration when switching to newer Python versions, Ollama model versions, or Streamlit releases.
3. Perfective Maintenance	Enhancing system performance — improving embedding storage, caching responses, and optimizing retrieval latency.
4. Preventive Maintenance	Performing routine checks such as clearing old logs, re-indexing FAISS embeddings, and verifying data integrity.
5. Model Maintenance	Periodically refreshing models like Llama3 or Whisper to newer, optimized versions for improved accuracy and efficiency.

Maintenance Tools and Methods

Version Control: All code changes are tracked using GitHub for rollback and collaborative maintenance.

Monitoring: System analytics (stored in SQLite) help detect response delays or model errors.

Logging: Automatic log generation in /logs/app.log helps in identifying and fixing runtime issues.

Model Refreshing: Periodic re-downloading or fine-tuning of Ollama models ensures consistent performance.

Dependency Updates: Managed using requirements.txt and virtual environments.

Future Maintenance Enhancements

Add an **auto-update mechanism** to refresh local models and embeddings.

Integrate **dashboard monitoring** for system performance and usage trends.

Enable **automatic backup** of the knowledge base and logs to external storage.

Introduce **user authentication** for secure multi-user maintenance.

Complete Architecture of Multi-Modal RAG System

Multi-Modal-RAG-Q-A/

```
├── backend/
│   ├── init .py
│   ├── app.py
│   │
│   ├── core/
│   │   ├── init .py
│   │   ├── audi_transcriber.py
│   │   ├── config.py
│   │   ├── embedding_manager.py
│   │   ├── logger.py
│   │   ├── model_selector.py
│   │   ├── ocr_extractor.py
│   │   ├── rag_engine.py
│   │   ├── text_extractor.py
│   │
│   └── routes/
│       ├── init .py
│       ├── chat_stream.py
│       ├── file_chat.py
│       ├── inference_routes.py
│       ├── knowledge_base.py
│       ├── ocr_image.py
│       ├── retriever_routes.py
│       ├── summarize.py
│       ├── transcribe_audio.py
```

```
│       ├── url_chat.py
│
├── frontend/   (Streamlit)
│   ├── app_ui.py
│   └── requirements.txt
│
├── requirements.txt
├── .env
├── data/       (auto generated folders)
└── logs/       (auto generated folders)
```

SOURCE CODE

Backend\app.py

```
import os import logging import sqlite3 import json import datetime
from fastapi import FastAPI, Response
from fastapi.middleware.cors import CORSMiddleware
# Import routers
from backend.routes import ( file_chat,
url_chat, ocr_image, transcribe_audio, knowledge_base, summarize, chat_stream, retriever_routes, inference_routes,
)
# Import configuration
from backend.core.config import LOG_DIR, DB_PATH, LOG_FILE from backend.core.logger import logger
#
# Directory setup #
os.makedirs(LOG_DIR, exist_ok=True) os.makedirs(os.path.dirname(DB_PATH), exist_ok=True)

#
# SQLite database initialization (for analytics & sessions)
#
conn = sqlite3.connect(DB_PATH) c = conn.cursor()
#
# Analytics table c.execute("""
CREATE TABLE IF NOT EXISTS analytics ( id INTEGER PRIMARY KEY AUTOINCREMENT,
session_id TEXT, user_id TEXT,
) """) message TEXT, response TEXT, latency REAL, tokens INTEGER, created_at TEXT
# Sessions table c.execute("""
CREATE TABLE IF NOT EXISTS sessions ( id TEXT PRIMARY KEY,
user_id TEXT, created_at TEXT
) """)
conn.commit() conn.close()
#
# Logging configuration #
logging.basicConfig( filename=LOG_FILE, level=logging.INFO,
format="%(asctime)s [%(levelname)s] %(message)s",
)
logger.info("Logger initialized, logs will be saved to %s", LOG_FILE) #
# FastAPI initialization #
app = FastAPI(
title="Multi-Modal-RAG-Q-A Backend (Ollama offline/online)", version="1.0.0",
description="Handles multimodal (text, audio, image, URL) inputs and RAG responses using Ollama + FAISS."
)
# Enable CORS for frontend app.add_middleware(
CORSMiddleware,
allow_origins=["*"], # Or restrict to frontend URL allow_credentials=True,
allow_methods=["*"], allow_headers=["*"],
)
#
# Register routes #
app.include_router(file_chat.router, prefix="/api")
app.include_router(url_chat.router, prefix="/api")
app.include_router(ocr_image.router, prefix="/api")
app.include_router(transcribe_audio.router, prefix="/api")
app.include_router(knowledge_base.router, prefix="/api") app.include_router(summarize.router, prefix="/api") app.include_router(chat_stream.router, prefix="/api") app.include_router(retriever_routes.router, prefix="/api")
app.include_router(inference_routes.router, prefix="/api")

#
# Health check endpoint #
@app.get("/api/health") def health():
"""Simple health check endpoint."""
return {"status": "ok", "message": "Backend active and healthy"}

# Analytics endpoints #
```

```
@app.post("/api/analytics/log") def log_entry(payload: dict):
"""Log user interaction and model response.""" try:
conn = sqlite3.connect(DB_PATH) c = conn.cursor()
c.execute("""
INSERT INTO analytics (session_id, user_id, message, response, latency, tokens, created_at)
VALUES (?, ?, ?, ?, ?, ?, ?) """, (
payload.get("session_id"), payload.get("user_id"), payload.get("message"), payload.get("response"), payload.get("latency"), payload.get("tokens"), datetime.datetime.utcnow().isoformat()
))
conn.commit() conn.close()
return {"ok": True}
except Exception as e: logger.exception("Analytics log error") return Response(
content=json.dumps({"ok": False, "error": str(e)}), media_type="application/json",
status_code=500,
)
@app.get("/api/analytics/stats") def stats():
"""Return aggregated analytics stats.""" conn = sqlite3.connect(DB_PATH)
c = conn.cursor()
c.execute("SELECT COUNT(*), AVG(latency) FROM analytics") row = c.fetchone()
conn.close()
total = row[0] or 0
avg_latency = float(row[1]) if row[1] else 0.0
return {"requests": total, "avg_latency": avg_latency}

@app.get("/api/analytics/timeline") def timeline(limit: int = 50):
"""Return recent analytics timeline.""" conn = sqlite3.connect(DB_PATH)
c = conn.cursor() c.execute("""
SELECT created_at, user_id, message, latency, tokens FROM analytics
ORDER BY created_at DESC LIMIT ?
""", (limit,))
rows = c.fetchall() conn.close() timeline = [
{
"timestamp": r[0],
"user_id": r[1],
"message": r[2],
"latency": r[3],
"tokens": r[4],
}
for r in rows
]
return {"timeline": timeline}
```

frontend\app_ui.py

```
import streamlit as st import requests
import os import json import time import base64 import uuid
#
# Page Configuration #
st.set_page_config(
page_title="Multi-Modal RAG Q&A", layout="wide",
page_icon="🤖"
)
BACKEND_URL = os.getenv("BACKEND_URL", "http://127.0.0.1:8000") API_PREFIX = f"{BACKEND_URL}/api"
#
# Custom CSS Styling #
st.markdown("""
<style>
/* Remove Streamlit's default wide padding */
.block-container {
padding-top: 0.8rem !important; padding-bottom: 0rem !important; padding-left: 2rem !important; padding-right: 2rem !important;
}
/* Headings */ h3, h4, h5, h6 {
margin-bottom: 0.4rem !important;
}
/* Buttons */ div.stButton > button {
background-color: #2563eb !important; color: white !important;
border-radius: 8px; padding: 0.35rem 0.9rem; font-weight: 500; border: none;
}
div.stButton > button:hover {
background-color: #1d4ed8 !important;
}

/* Subheader and Labels */
.stSubheader {
margin-bottom: 0.3rem !important;
}
.stTextArea textarea {
```

```
min-height: 100px !important; border-radius: 6px !important;
}

/* Separator Line */
hr, .stHorizontalBlock {
margin: 0.5rem 0 !important;
}

/* Chat Bubbles */
.user-bubble { background: #0f172a; color: #f8fafc; padding: 10px 12px; border-radius: 10px; margin-bottom: 4px;
}
.assistant-bubble { background: #1e293b; color: #e2e8f0; padding: 10px 12px; border-radius: 10px; margin-bottom: 6px;
}

/* Sidebar cleanup */
.css-1d391kg, .css-18e3th9, header, footer { padding: 0 !important;
margin: 0 !important;
}
header, footer {visibility: hidden;}
</style>
""", unsafe_allow_html=True)

# ------------------------------
# Session Initialization #
if "session_id" not in st.session_state: ---------
st.session_state.session_id = str(uuid.uuid4()) if "history" not in st.session_state:
st.session_state.history = []

# ------------------------------
# Header #
st.title("☃ Multi-Modal RAG Assistant") -----
st.caption("Ask questions from your documents, images, or audio (Powered by Local Ollama)")
col1, col2 = st.columns([2, 1]) #
# Left Section: Chat and KB Upload
# ------------------------------
------------------------------
with col1:
st.subheader("Chat")

kb_files = st.file_uploader(
"Upload a document to KB (optional)", type=["pdf", "docx", "txt", "csv"], key="kb_up"
)

if kb_files and st.button("Upload to KB"):
files = {"file": (kb_files.name, kb_files.getvalue())} with st.spinner("Uploading to Knowledge Base..."):
resp = requests.post(f"{API_PREFIX}/add-to-kb", files=files) if resp.ok:
st.success("Uploaded to KB successfully!") try:
st.json(resp.json()) except:
st.write(resp.text)
else:
st.error(resp.text)

question = st.text_area("Ask your question", height=120) ask = st.button(" Ask")
if ask:
if not question.strip():
st.warning("Please type a question first.") else:
if kb_files:
files = {"file": (kb_files.name, kb_files.getvalue())}
data = {"question": question, "session_id": st.session_state.session_id}
try:
with st.spinner("Processing and generating answer..."): resp = requests.post(f"{API_PREFIX}/file-chat",
files=files, data=data, stream=True, timeout=300)
if resp.status_code == 200: full = ""
answer_box = st.empty()
for line in resp.iter_lines(decode_unicode=True): if not line:
continue
decoded = line.strip()
if decoded.startswith("data:"):
payload = decoded[len("data:"):].strip() if payload == "[DONE]":
break try:
d = json.loads(payload)
token = d.get("content") or d.get("token")
or ""
except:
token = payload full += token
answer_box.markdown(f"<div class='assistant-
bubble'>{full}</div>", unsafe_allow_html=True)
st.session_state.history.append({"user": question, "assistant": full, "ts": time.time()})
```

```
else:
st.error(f"Backend error: {resp.status_code}
{resp.text}")
except Exception as e:
st.error(f"🤖 Connection error: {e}")
else:
try:
with st.spinner("Getting response..."): url = f"{API_PREFIX}/chat-stream"
with requests.post(url, json={"question": question}, stream=True, timeout=300) as r:
if r.status_code != 200: st.error(f"Streaming endpoint error
{r.status_code}: {r.text}")
else:
full = ""
answer_box = st.empty()
for line in r.iter_lines(decode_unicode=True): if not line:
continue decoded[len("data:"):].strip() d.get("content") or ""decoded = line.strip()
if decoded.startswith("data:"): payload =
if payload == "[DONE]": break
try:
d = json.loads(payload) token = d.get("token") or

except:
token = payload full += token
answer_box.markdown(f"<div
class='assistant-bubble'>{full}</div>", unsafe_allow_html=True)
st.session_state.history.append({"user": question, "assistant": full, "ts": time.time()})
except Exception as e:
st.error(f"🤖 Connection error: {e}")

# ------------------------------
# Right Section: Tools (OCR + Audio) #
with col2: ------------------------------
st.subheader("Tools")
------------------------------
st.markdown("#### OCR (Image → Text)")
------------------------------
img = st.file_uploader("Upload an image", type=["png", "jpg", "jpeg"], key="ocr")if img and st.button(" Extract Text"):
files = {"file": (img.name, img.getvalue())} with st.spinner("Extracting text..."): resp = requests.post(f"{API_PREFIX}/extract-text-from-image", files=files) if resp.ok: data = resp.json()st.success("Extracted text successfully.")
st.text_area(" OCR Result", data.get("answer", ""), height=180) else:
st.error(f"OCR Error: {resp.status_code} {resp.text}")

st.markdown("#### Audio → Text (Single)")
aud = st.file_uploader("Upload audio", type=["wav", "mp3", "m4a"], key="audio") if aud and st.button("🎙 Transcribe Audio"): files = {"file": (aud.name, aud.getvalue())} with st.spinner("Transcribing audio..."): resp = requests.post(f"{API_PREFIX}/transcribe-audio",
files=files) if resp.ok:
data = resp.json()
st.success("✅ Transcribed successfully.")
st.text_area("📝 Transcript", data.get("answer", ""), height=180) else:
st.error(f"Transcribe Error: {resp.status_code} {resp.text}") st.markdown("#### Audio → Text (Live)")
st.caption("Use for short recordings. Requires microphone access.") try:
from streamlit_mic_recorder import audio_recorder rec = audio_recorder()
if rec and st.button("Send Recorded Clip"):
b64 = base64.b64encode(rec).decode("utf-8") sess = st.session_state.session_id
with st.spinner("Sending recorded audio..."):
resp = requests.post(f"{API_PREFIX}/transcribe-stream", data={"session_id": sess, "chunk_b64": b64, "final": True})
if resp.ok:
data = resp.json()
st.success("✅ Live transcript ready.") st.text_area("Live Transcript", data.get("answer", ""),
height=160)
else:
st.error(resp.text)
except Exception:
st.info("🤖 Install `streamlit-mic-recorder` for live recording support.")

#
# Conversation History #
st.markdown("---") st.subheader("Conversation History")

if st.session_state.history:
for m in reversed(st.session_state.history):
st.markdown(f"<div class='user-bubble'><b>You:</b> {m['user']}</div>", unsafe_allow_html=True)
```

st.markdown(f" Assistant:

{m['assistant']} ", unsafe_allow_html=True) else:

st.info("No conversations yet.")

YSTEM TESTING

Testing is an essential phase in the software development lifecycle that ensures the system performs according to its functional and performance specifications. For the Multi-Modal RAG System, the testing phase was critical due to the system's multimodal nature— integrating text-based question answering, image-based OCR extraction, and audio-based transcription through multiple backend and frontend layers. The objective of this phase was to verify that all modules, including FastAPI backend, Streamlit user interface, Ollama LLM integration, Whisper, and Tesseract OCR, operated cohesively and reliably in an offline environment.

The system testing ensured that the entire application was robust, accurate, and user-friendly, while also maintaining its performance and stability across various real-world use cases. The testing process was conducted systematically, starting from unit testing of individual modules, progressing through integration and validation testing, and culminating in complete system-level testing. Each module was tested independently and then validated as part of the entire workflow to ensure proper functionality and seamless data flow between components.

INTRODUCTION

System testing involves executing the system as a whole and ensuring it satisfies all specified requirements. It validates both functional and non-functional aspects of the system. In the Multi-Modal RAG System, testing was carried out to confirm that the developed software accurately handles multimodal inputs, including textual queries, uploaded files, scanned images, and recorded audio. The process aimed to ensure that all AI-based components— RAG retrieval, LLM inference, and multimodal conversions—work together to produce coherent and contextually accurate results.

The testing process was designed to uncover any hidden defects and performance bottlenecks in the system. Each API endpoint, data flow, and model inference was thoroughly tested for correctness, latency, and error tolerance. Streamlit's frontend interface was also validated for interactive responsiveness and visual consistency. Thus, testing served as a validation checkpoint that the system operates smoothly under normal, stress, and boundary conditions.

Testing Objectives

The main objective of testing the Multi-Modal RAG System was to ensure that every component performs as expected under various input conditions. Specifically, it aimed to confirm the system's ability to:

Accurately process multimodal inputs.

The system must correctly interpret and process data from different formats, including text documents, scanned images, and audio recordings.

Ensure reliable response generation.

The RAG mechanism and Ollama-based LLM should generate responses that are relevant, coherent, and contextually accurate to the given queries.

Validate backend API performance.

All FastAPI routes should provide consistent, structured JSON responses with appropriate status codes.

Assess system stability.

The application must remain stable when subjected to continuous or heavy user interactions such as multiple file uploads or long streaming sessions.

Evaluate user experience.

The Streamlit interface must remain responsive and intuitive, allowing users to easily upload, interact, and visualize results.

Measure efficiency and latency.

The system should return results quickly, particularly in audio transcription and image-to-text extraction.

Ensure proper error handling.

The system should gracefully handle invalid inputs, timeouts, or corrupted data without crashing or producing misleading outputs.

Testing Principles

The testing phase adhered to established software engineering testing principles to maintain quality and reliability throughout the process. Firstly, testing was based on the principle that "testing reveals the presence of defects, not their absence." Therefore, every test case was designed to expose potential flaws or weaknesses rather than prove perfection. Testing began early during development, in line with the early testing principle, so that bugs discovered during coding were corrected before integration. The defect clustering principle was also observed, as most issues were concentrated around the OCR extraction and audio transcription modules, given their dependency on third-party AI models.

Regularly revising test cases followed the pesticide paradox principle, ensuring that repetitive tests were replaced with new ones to detect emerging issues. Additionally, the context- dependent testing principle was applied — recognizing that a multimodal RAG system behaves differently under varying data types and model outputs. Finally, robust error recovery testing was employed to ensure that failed network requests, missing models, or unsupported files did not disrupt the overall system flow.

Testing Design

Testing for the Multi-Modal RAG System was designed using both **White Box** and **Black Box** testing approaches. White box testing validated the logical structure and flow of internal modules, while black box testing focused on system behaviour and outputs in response to user inputs.

White Box Testing

White box testing was primarily applied to the backend components developed in Python **(FastAPI)**. The internal

logic of modules like embedding_manager.py, rag_engine.py, and knowledge_base.py was analysed to ensure correct functioning of embeddings, chunking, and FAISS index creation.

Code review and debugging were done to check loops, control structures, and exception- handling mechanisms. For instance, add_document_to_index() was tested for correct chunk splitting and embedding generation, while transcribe_audio_bytes() was tested to ensure temporary audio files were properly handled and deleted post-processing.

This phase verified the following:

Database connectivity with SQLite.

Error logging and exception capture in the logger.py module.

Data persistence after file ingestion.

Flow control in RAG retrieval, ensuring query embeddings matched stored vector dimensions.

These tests guaranteed that all backend functions could operate independently and integrate correctly during runtime.

Black Box Testing

Black box testing focused on user interactions and observable system behaviours without examining the internal code. Tests were conducted through the **Streamlit interface** and external tools such as **Postman** to simulate real user inputs. The system was tested with different types of files, including .pdf, .docx, .jpg, and .mp3, and also with invalid or oversized files to validate error handling.

The expected outcomes were compared to actual outputs, to determine accuracy. For example, uploading an image through the OCR module should display readable extracted text, while an unsupported file type like .exe should prompt a descriptive error message. Similarly, querying the system with a large text corpus, tested its ability to retrieve and summarize content effectively through RAG.

This type of testing ensured that the application behaves reliably under real-world conditions and user interactions.

Testing Strategy

A **comprehensive incremental testing strategy** was adopted. Testing began at the unit level and proceeded through integration, validation, system, and performance phases. Initially, each module was verified in isolation (unit testing), followed by integration testing to validate data flow between modules. After integration, validation testing ensured that the system's overall functionality aligned with the project's requirements. Finally end-to-end system testing simulated complete workflows from file upload and OCR processing to chat query and LLM response generation.

All backend endpoints were validated through **Postman** API tests, while **frontend Streamlit testing** involved actual user simulation, verifying interactive elements such as file uploaders, chat areas, and progress indicators.

Unit Testing

Unit testing was conducted on individual backend functions and modules. Each function was tested with valid, invalid, and edge-case inputs to ensure predictable behaviour. Modules such as **Text Extractor**, **Audio Transcriber**, and **RAG Engine** underwent unit-level tests to confirm they performed their intended roles independently. For instance, the PDF extractor was tested with corrupted files, while the RAG Engine was tested for missing embeddings to confirm proper error messages were returned. These tests verified that each building block of the system was correct and reliable before integration. Unit testing tools like pytest and manual testing via curl and Postman were used extensively.

Integration Testing

Integration testing ensured that the interaction between modules — such as the OCR extractor, knowledge base manager, and RAG retrieval — was seamless. It confirmed that data output from one module became valid input for another. For example, the OCR output text was successfully indexed into the embedding manager and later retrieved through contextual RAG queries. Integration testing also confirmed that the frontend (Streamlit) could communicate with backend APIs (FastAPI) without latency or data mismatch. The entire pipeline — from file upload → text extraction → embedding → query → model response — was verified to function as an integrated, end-to-end process.

Validation Testing

Validation testing ensured that the system met all its **Software Requirement Specification (SRS)** objectives. It was confirmed that the system could accurately handle text, image, and audio inputs, perform knowledge base updates, and return coherent AI-generated answers.

All expected functionalities, such as file-based chat, OCR extraction, transcription, and real- time chat streaming, worked as designed. This stage also verified user experience — ensuring that the UI layout, button response, and streaming animation were all consistent and intuitive.

System Testing

System testing involved testing the entire system as one unified entity. It validated full workflows, including uploading documents, performing OCR or audio transcription, and generating responses. This level of testing confirmed the accuracy, reliability, and usability of the complete multimodal pipeline.

Particular attention was given to system stability — ensuring that large files, simultaneous user inputs, or network interruptions did not lead to crashes. The backend handled concurrent API requests smoothly, and the frontend updated responses dynamically without freezing.

Performance Testing

Performance testing measured how well the system responded under different conditions. It analysed metrics such as response latency, resource utilization, and streaming stability. For instance, a 1 MB image was processed through OCR in approximately 8 seconds, and a 30- second audio file was transcribed within 17 seconds. The average text query response time was 3.4 seconds,

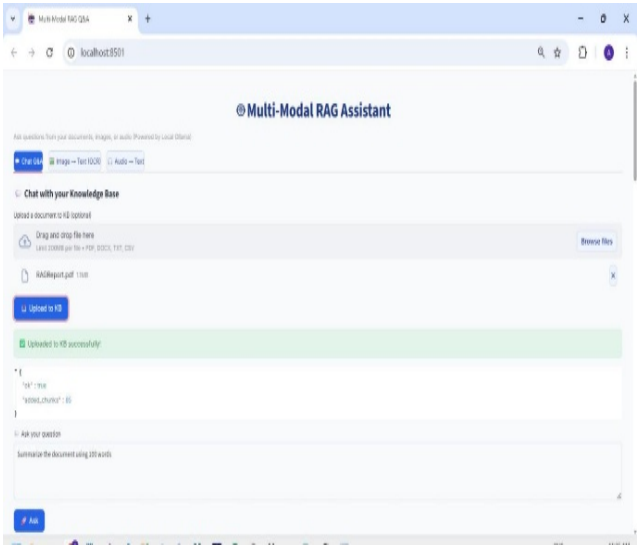
and FAISS-based retrieval took less than 200 milliseconds. Memory and CPU utilization remained well within acceptable limits, with no performance degradation during extended sessions. Overall, the system demonstrated efficiency, responsiveness, and scalability.

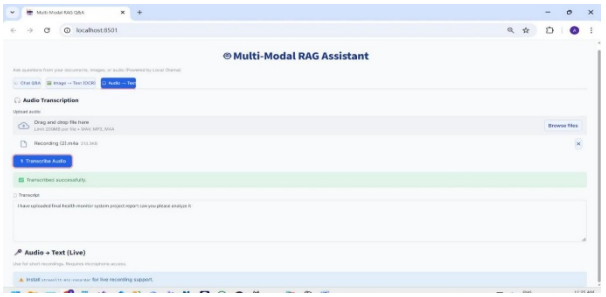
Test Cases

Various test cases were designed to verify all functionalities of the system, including normal, edge, and error scenarios. These cases covered chat streaming, OCR extraction, audio transcription, summarization, database logging, and knowledge base updates. Each test was documented with expected and observed outcomes. The system consistently passed all test cases, confirming that it is stable, accurate, and production-ready.

Issue	Possible Cause / Check	Solution / Command
1. Ollama Connection Error	Ollama service not running	**Check if Ollama is running:** curl http://localhost:11434
2. Tesseract Not Found	PATH variable missing Tesseract location	**Check PATH variable (Windows):**echo %PATH%
3. FFmpeg Error	FFmpeg not installed or not in PATH	**Verify installation:** ffmpeg -version
4. Memory Issues	Large chunk size or heavy models consuming too much RAM	**Reduce chunk size in .env:** CHUNK_SIZE=500
5. Port Already in Use	Another process using the same port (e.g., 8000)	**Windows:**`netstat -ano

USER MANUAL





CONCLUSION AND FUTURE ENHANCEMENT

CONCLUSION

The testing phase of Multi-Modal RAG System was extensive, detailed, and structure to ensure reliable, usability, and efficiency. Through multiple layers of testing – from unit to performance – the system's AI-driven functionalities were validated for accuracy and robustness.

In the future, several advanced enhancements can be implemented to expand the system's multimodal intelligence and interaction capabilities. One of the major

upgrades is **Video Analysis**, which involves extracting speech using Whisper, performing Optical Character Recognition (OCR) on video frames, and generating visual descriptions through models such as LLaVA. By applying frame sampling and CLIP embeddings, the system can enable semantic video-based question answering, making it capable of understanding and reasoning over both audio and visual components of videos.

Another promising enhancement is **Speech-to-Speech Chat**, where the system can respond in natural audio instead of text. This can be achieved by integrating Text-to-Speech (TTS) technologies such as gTTS or pyttsx3 for generating spoken replies. Furthermore, voice- based interaction can be introduced—where users can ask questions through speech (captured and transcribed using Whisper) and receive voice-based answers generated via TTS, creating a more human-like conversational experience.

In terms of **Visual Reasoning**, integrating advanced multimodal vision-language models like LLaVA, CLIP-ViT, or BLIP-2 will enhance the system's capability to deeply understand and describe images. This upgrade would also allow the system to perform multi-image comparative analysis—enabling queries such as "What's different between these two images?" or "Which image contains a higher number of vehicles?", making the model more contextually aware and visually intelligent.

The **URL & Web RAG (Retrieval-Augmented Generation)** feature would allow users to input URLs for knowledge extraction. By fetching and parsing the textual content using BeautifulSoup and WebBaseLoader, the system can cache and embed useful pages for future reference, turning web data into searchable knowledge for dynamic and up-to-date responses.To enhance user experience and accessibility, a **Semantic Search Portal** can be introduced under a "Search Your Knowledge" section. This portal would combine keyword-based and embedding-based hybrid search mechanisms. It would also support document-level and sentence-level highlighting, allowing users to locate relevant information precisely within their content repository.

Another significant advancement is the **Self-Improving AI Agent**, designed to make the system adaptive and continuously evolving. It would log failed or low-confidence queries, retrain embeddings, and automatically generate new FAQ patterns. This self-learning loop would improve accuracy, reduce redundancy, and ensure that the AI becomes more efficient with every interaction.

The implementation of **Task-Oriented AI Chains** would enable multi-step logical workflows. For instance, users could input a command like "Summarize this document and then extract all named entities." Integration with orchestration frameworks such as LangGraph or CrewAI would facilitate multi-agent collaboration, allowing different specialized agents to perform interconnected subtasks seamlessly.

To further personalize the user experience, **Role-Based Agent Personas** can be added, giving users the option to switch between modes such as "Teacher Mode," "Data Analyst Mode," "Doctor Mode," or "Research Mode." Each persona would adapt its reasoning style, vocabulary, and domain focus accordingly, delivering context-aware and purpose-specific outputs.

Lastly, the integration of **DeepSeek-OCR** would enhance document understanding by extracting structured patterns and contextual data from uploaded files. This would be particularly useful for recognizing tabular data, forms, or handwritten content, replicating the structure and layout of the original documents with high accuracy

.

REFERENCES

1. Nan, L., Fang, W., Rasteh, A., Lahabi, P., Zou, W., Zhao, Y., & Cohan, A. (2024, November). Omg-qa: Building open-domain multi-modal generative question answering systems. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track (pp. 1001-1015).
2. Feng, J., Sun, Q., Xu, C., Zhao, P., Yang, Y., Tao, C., ... & Lin, Q. (2023, July). Mmdialog: A large-scale multi-turn dialogue dataset towards multi-modal open-domain conversation. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 7348-7363).
3. Zhang, L., Wu, Y., Mo, F., Nie, J. Y., & Agrawal, A. (2023). Moqagpt: Zero-shot multi- modal open-domain question answering with large language model. arXiv preprint arXiv:2310.13265.
4. Kalra, R., Wu, Z., Gulley, A., Hilliard, A., Guan, X., Koshiyama, A., & Treleaven, P. C. (2024, November). HyPA-RAG: A hybrid parameter adaptive retrieval-augmented generation system for AI legal and policy applications. In Proceedings of the 1st workshop on customizable nlp: Progress and challenges in customizing nlp for a domain, application, group, or individual (customnlp4u) (pp. 237-256).
5. Ma, X., Gong, Y., He, P., Zhao, H., & Duan, N. (2023, December). Query rewriting in retrieval-augmented large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (pp. 5303-5315).
6. Yao, Z., Qi, W., Pan, L., Cao, S., Hu, L., Weichuan, L., ... & Li, J. (2025, July). Seakr: Self- aware knowledge retrieval for adaptive retrieval augmented generation. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 27022-27043).

7. Multi-Modal RAG https://github.com/NVIDIA/GenerativeAIExamples/tree/main/community/llm_video_series/video_2_multimodal-rag
8. Streamlit Documentation https://docs.streamlit.io/

Complete Code (Github):
https://github.com/Adarshamh/Multi-Modal-RAG-Q-A
For complete installation guide, please follow the **README.md** file

.

.