AI-Powered Website Assistant Using Retrieval-Augmented Generation (RAG)

---

1. Project Overview

This project involves building an AI-powered assistant capable of answering questions based strictly on the content of a given website. The system uses a Retrieval-Augmented Generation (RAG) architecture, combining web scraping, vector embeddings, similarity search, and large language model response generation.

The assistant:

- **Extracts textual data from a website**
- **Converts content into embeddings**
- **Stores embeddings in a vector database**
- **Retrieves relevant information for user queries**
- **Generates context-aware answers**
- **Provides a web-based chat interface**

The entire system is implemented in Python.

2. Objective

The primary objective of this project is to build a functional AI customer-support assistant that:

1. **Ingests and processes website content.**
2. **Stores semantic representations in a vector database.**
3. **Retrieves relevant information based on user queries.**
4. **Generates accurate responses using a Large Language Model.**
5. **Provides an interactive user interface.**

Students will gain hands-on experience with:

- **API development**
- **Vector databases**
- **Embeddings**
- **Retrieval pipelines**
- **LLM integration**
- **UI integration**

3. Technology Stack Used

The project uses the following technologies:

Backend Framework

- **FastAPI**
  Used to build REST APIs for ingestion and question answering.

Vector Database

- **Qdrant**
  Used to store and search high-dimensional vector embeddings.

LLM & Embeddings

- **OpenAI**
  Used for:
  - Generating embeddings from text
  - Producing final contextual answers

Web Scraping

- Playwright (for dynamic website content rendering)
- BeautifulSoup (for text extraction and cleaning)

UI Layer

- **Gradio**
  Used to build a simple conversational interface.

4. System Architecture

The system follows a Retrieval-Augmented Generation architecture.

High-Level Flow:

User → UI → FastAPI → Vector Search → LLM → Response → UI

Detailed Flow:

1. Website content is scraped.
2. Text is split into manageable chunks.
3. Each chunk is converted into an embedding.
4. Embeddings are stored in Qdrant.
5. User query is converted into an embedding.
6. Similar chunks are retrieved from Qdrant.
7. Retrieved chunks are passed to the LLM as context.
8. LLM generates a final answer grounded in retrieved data.
9. Answer is returned to the UI.

5. Backend Implementation

5.1 API Endpoints

1. /init

- Scrapes website content.
- Splits content into chunks.
- Generates embeddings.
- Stores embeddings in Qdrant.

This endpoint initializes the knowledge base.

2. /ask

- Accepts a query parameter q.
- Generates embedding for the query.
- Performs similarity search in Qdrant.
- Retrieves top relevant chunks.
- Sends retrieved context to LLM.
- Returns generated answer as JSON.

6. Data Processing Pipeline

6.1 Web Scraping

- Playwright launches a headless browser.
- Website HTML content is loaded.
- Script/style elements are removed.
- Clean text is extracted.

6.2 Text Chunking

Text is divided into smaller segments using:

- Fixed chunk size
- Overlapping window

This ensures:

- Better semantic embedding
- Improved retrieval accuracy

6.3 Embedding Generation

Each chunk is converted into a high-dimensional vector representation.

These embeddings capture semantic meaning rather than just keywords.

7. Vector Database Storage

Qdrant stores:

- Vector embeddings
- Associated text payload

This allows similarity search using cosine distance or dot product.

When a query is asked:

- Query embedding is generated.
- Vector search retrieves most similar stored chunks.

8. LLM Response Generation

The retrieved chunks are passed to the language model as context.

A structured system prompt ensures:

- The assistant answers only using retrieved context.
- It avoids hallucinations.
- It remains concise and professional.

9. User Interface

A basic chat interface was implemented using Gradio.

Features:

- Chat-based interaction
- Query input
- Real-time response display
- Backend API integration via HTTP request

The UI communicates with the FastAPI backend using GET requests to /ask.

10. What Has Been Successfully Achieved

The following components are fully functional:

✓ Website scraping pipeline
✓ Text chunking mechanism
✓ Embedding generation
✓ Vector storage in Qdrant
✓ Similarity-based retrieval
✓ LLM-based answer generation
✓ FastAPI backend with endpoints
✓ Gradio-based interactive UI
✓ End-to-end working RAG system

11. Learning Outcomes

By completing this project, students understand:

- **How RAG systems work in production**
- **How vector databases differ from traditional databases**
- **How embeddings enable semantic search**
- **How LLMs can be grounded using retrieved context**
- **How to build and connect backend APIs**
- **How to integrate a UI with an AI backend**

12. Scope Limitation (Important)

This project strictly includes:

- **Single-domain website ingestion**
- **Stateless question answering**
- **Basic vector search**
- **Basic UI interface**

The project does NOT include:

- **Conversation memory**
- **Multi-tenant architecture**
- **SaaS deployment**
- **Analytics dashboard**
- **Advanced crawling**
- **Authentication**
- **Production-grade security**

13. Expected Deliverables from Students

Students should submit:

1. **Fully working FastAPI backend**
2. **Functional /init and /ask endpoints**
3. **Qdrant integration**
4. **Embedding and retrieval pipeline**
5. **Gradio UI connected to backend**
6. **Demonstration of successful contextual question answering**