# Smart AI Assistant — Project Documentation

## Overview

The **Smart AI Assistant** is an agentic application built using **LangGraph, LangChain,** and **Streamlit** that enables users to:

* Interact with **uploaded documents** (PDF, DOCX, TXT)
* Query **structured data (databases)** using natural language
* Ask **general knowledge questions** through an integrated LLM
* View **reasoning transparency —** see how and why the assistant chose its response path

## Core Features

| **Feature** | **Description** |
| --- | --- |
| **Document Upload** | Users can upload multiple files (PDF, Word, Text) which are automatically extracted, chunked, embedded, and stored in FAISS for semantic retrieval. |
| **Semantic Search (RAG)** | Supports natural-language queries across documents using embeddings and vector similarity search. |
| **Database Chat** | Converts user questions to SQL queries via LangChain’s SQL Agent to retrieve structured data. |
| **General LLM Chat** | Handles open-ended questions using Gemini (or Hugging Face fallback model). |
| **Reasoning Transparency** | Each answer includes a “reasoning trace” explaining how the model reached that conclusion. |
| **Contextual Memory** | Maintains session memory so the assistant remembers prior user questions and context. |

**System Components Overview**

The architecture is designed using a modular agentic framework, combining LangGraph, LangChain, and Streamlit, with clear separation of concerns between document ingestion, semantic retrieval, structured data querying, and conversational reasoning.

Each module performs a dedicated role but integrates seamlessly through a shared LangGraph State (ChatbotState), enabling memory and reasoning transparency.

**Layered Architecture**

| Layer | Description |
| --- | --- |
| 1. User Interface (UI Layer) | Built with Streamlit, providing a conversational chat interface and file uploader. It enables document upload, chat input, and displays reasoning traces, responses, and document context. |
| 2. Agentic Orchestration Layer | Implemented using LangGraph, which coordinates all nodes (Classifier → RAG/DB/LLM). It maintains the shared ChatbotState and handles conditional routing dynamically based on the query type. |
| 3. Intelligence Layer (LLM Layer) | Uses Google Gemini (for embeddings and generation) or Hugging Face models as fallback. This layer powers query understanding, semantic reasoning, and tool (node) selection. |
| 4. Knowledge Access Layer | Handles data retrieval via two submodules:  - RAG (Unstructured): Retrieves text chunks from documents stored in a FAISS Vector Store using cosine similarity.  - DB (Structured): Uses LangChain’s SQL Agent to query structured data sources like SQLite or PostgreSQL. |
| 5. Persistence Layer | Stores document embeddings (FAISS), database connections, and conversation history if extended to persistent memory. |

**Data Flow**

┌────────────────────────────────────────────────────┐

│ Streamlit UI │

│ - Document Upload │

│ - Chat Interface │

└───────────────┬────────────────────────────────────┘

│

▼

┌──────────────────────┐

│ LangGraph Engine │

│ (Agentic Workflow) │

└─────────┬────────────┘

│

┌──────────┼────────────┐

│ │ │

▼ ▼ ▼

┌────────┐ ┌────────┐ ┌────────┐

│ RAG │ │ DB │ │ LLM │

│ Node │ │ Node │ │ Node │

└────────┘ └────────┘ └────────┘

│ │ │

▼ ▼ ▼

┌──────────────────────────────────┐

│ Response + Reasoning │

│ (via shared ChatbotState) │

└──────────────────────────────────┘

Node Responsibilities (LangGraph)

| Node | Function | Tools Used |
| --- | --- | --- |
| Classify Node | Uses LLM prompt to classify query into one of: RAG / DB / LLM | Gemini / HF Model |
| Router | Decides next node based on classification result | LangGraph conditional edges |
| RAG Node | Retrieves semantically similar chunks from FAISS, summarizes or answers based on them | LangChain + FAISS |
| DB Node | Converts question into SQL query and executes against structured data | LangChain SQL Agent |
| LLM Node | Handles general questions not related to uploaded docs or DB | Gemini / HF Model |

### Context and Memory Management

* Each conversation turn is represented as a ChatbotState object containing:
  + query (user input)
  + query\_type (classification result)
  + context (retrieved documents or DB rows)
  + answer (final response)
  + reasoning\_trace (explanation of process)
* The memory mechanism ensures that follow-up questions can reuse previous context (for example, “What about the next quarter?” after a financial summary).