Informe de Arquitectura y Componentes del Sistema RAG

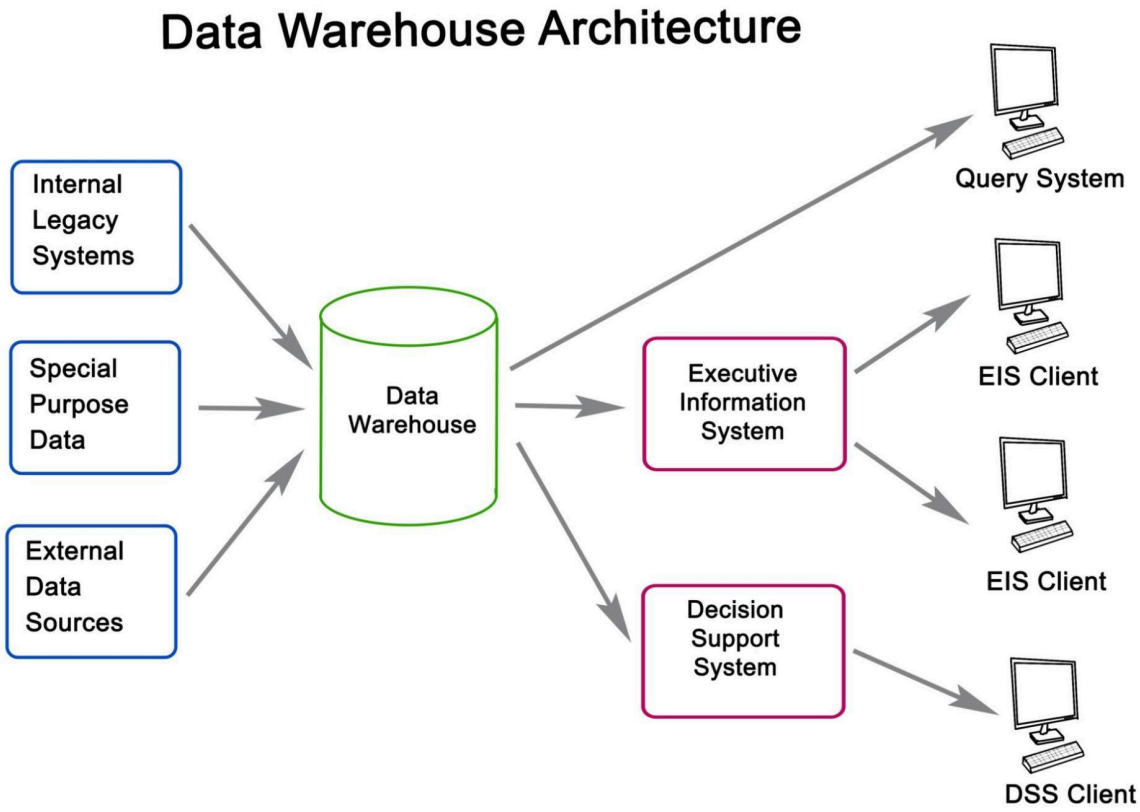
Este documento describe la arquitectura tecnológica (Stack) de la aplicación de Generación Aumentada por Recuperación (RAG) y detalla la función de cada archivo de código que compone el sistema.

I. Arquitectura Tecnológica (Technology Stack)

El sistema opera bajo el paradigma RAG, combinando una base de datos de conocimiento local con un Gran Modelo de Lenguaje (LLM) para generar respuestas precisas y referenciadas.

Componente	Tecnología Específica	Rol en el Sistema
Framework RAG (Orquestación)	**LangChain** (mediante **LCEL**)	Proporciona la estructura declarativa para construir las "cadenas" de ejecución: **Pregunta → Recuperación → Generación**. Permite la ejecución paralela y el manejo eficiente del flujo de datos.
LLM (Generación)	**Mistral** (Modelo de Ollama)	Es el cerebro del sistema. Genera las respuestas finales basándose en la pregunta del usuario y el contexto (chunks) recuperado de la base de datos.
Embeddings (Traducción)	**HuggingFace Embeddings** (paraphrase-multilingual-mpnet-base-v2)	Convierte el texto de los documentos y las preguntas del usuario en vectores numéricos de alta dimensión, permitiendo la búsqueda semántica eficiente.

Base de Datos Vectorial	**Chroma DB**	Almacena y gestiona los vectores de los documentos. Es la base de datos que permite la "memoria" del sistema.
Frontend/Interfaz	**Streamlit**	Proporciona la interfaz web interactiva (chat, selectores de estrategia, botones) en un entorno Python.
Text-to-Speech (TTS)	**Gemini TTS API** (gemini-2.5-flash-preview-tts)	Transforma el texto de las respuestas del LLM en audio de alta calidad, mejorando la accesibilidad de la aplicación.
Datos Fuente	Archivos **PDF** locales (en la carpeta ./libros/)	Los documentos de conocimiento sobre los que se entrena indirectamente el sistema RAG.



Getty Images

II. Informe de Archivos de Código

El sistema está dividido en cuatro archivos clave, cada uno con una responsabilidad clara dentro de la tubería RAG y la interfaz de usuario:

1. index_data.py (Pipeline de Datos)

Este archivo es el **motor de pre-procesamiento**. Se encarga de preparar los documentos de la carpeta ./libros/ para que puedan ser usados por el sistema RAG.

Función	Descripción
Carga de Datos	Utiliza PyPDFLoader para leer y extraer el texto de todos los archivos PDF nuevos.
Estrategias de Segmentación	Define y aplica las diferentes estrategias de *chunking* (ej: RecursiveCharacterTextSplitter) para dividir los documentos largos en trozos de

	tamaño manejable, preservando el contexto.
Creación/Actualización de DB	Inicializa los *embeddings*, convierte los *chunks* en vectores y los persiste en las carpetas de **Chroma DB** (una por estrategia).
Log de Procesamiento	Mantiene un registro (processed_files.txt) de los documentos ya indexados para evitar el reprocesamiento innecesario, optimizando el tiempo de ejecución.

2. rag_utils.py (Lógica Central y Cadenas LangChain)

Este es el **corazón lógico del sistema**. Contiene las funciones esenciales para inicializar los modelos y construir las cadenas de ejecución que definen el comportamiento del RAG y el Resumen. Su característica principal es el uso intensivo de **LCEL**.

Función	**Descripción**
initialize_embeddings()	Carga el modelo HuggingFaceEmbeddings para la traducción de texto a vectores.
format_docs() / format_sources()	Funciones de ayuda que se integran en las cadenas LCEL para dar formato al contexto y extraer las fuentes.
Uso de LCEL (Clave)	**LangChain Expression Language** (LCEL) se usa para construir las cadenas de manera modular y asíncrona:
	- **Operador Pipe (`
	- **RunnableParallel:** Se utiliza para la ejecución paralela. Esto es clave en el RAG para recuperar los documentos (**documents**) y pasar la pregunta (**question**) al mismo tiempo, sin esperar a que el otro termine.
	- **RunnablePassthrough:** Garantiza que la

	entrada inicial del usuario (la pregunta) pase sin modificaciones a través de partes de la cadena que solo necesitan ese valor.
create_rag_chain()	Construye la cadena RAG completa usando LCEL, conectando el retriever (Chroma) y el LLM (Mistral) con una estructura paralela para obtener la respuesta y las fuentes simultáneamente.
create_summary_chain()	Construye la cadena de Resumen más simple, donde el prompt y el LLM están conectados secuencialmente a través del operador pipe.

3. rag_streamlit_tts.py (Interfaz de Usuario y Control de Flujo)

Este es el **controlador de la aplicación web**. Inicializa la interfaz de Streamlit, gestiona la lógica de la sesión y coordina todas las demás funcionalidades.

Función	**Descripción**
UI/Sidebar	Define los selectores para la **Estrategia de Segmentación** y el **Tema** del documento, permitiendo al usuario cambiar entre las bases de datos vectoriales.
load_and_initialize_rag()	Función cacheada que carga dinámicamente la base de datos de Chroma y las cadenas RAG/Summary basadas en las selecciones del usuario.
handle_user_input()	Recibe la pregunta, determina el modo de operación (RAG o Resumen), invoca la cadena LangChain correspondiente y guarda la respuesta en el historial del chat.
generate_tts_button()	**Función crucial para el TTS**. Limpia el texto de la respuesta (solucionando errores de sintaxis JSON), serializa el texto, e inyecta el componente HTML (tts_component.html) en la interfaz.

4. tts_component.html (Componente de Text-to-Speech)

Este archivo es un **componente frontend aislado** (HTML, CSS y JavaScript) que se inyecta en la aplicación Streamlit.

Función	Descripción
Lógica de API	Contiene el código JavaScript para llamar al *endpoint* de la API de **Gemini TTS** de Google, enviando el texto a vocalizar.
Procesamiento de Audio	La API retorna datos de audio PCM sin procesar. El JS utiliza la **Web Audio API** para decodificar estos datos, convertirlos en un archivo WAV (*Blob*) y gestionar la reproducción de audio.
Interfaz de Botón	Muestra el botón de "Reproducir Audio" y gestiona su estado (cargando, reproduciendo, inactivo), con la lógica para detener la reproducción si ya está en curso.

Este conjunto de archivos trabaja en conjunto para ofrecer una experiencia completa: el archivo index_data.py se ejecuta primero para construir el conocimiento, rag_utils.py define cómo usar ese conocimiento, y rag_streamlit_tts.py lo presenta en una interfaz fácil de usar, complementada por la funcionalidad de tts_component.html.