Informe Técnico Detallado: Arquitectura RAG (Recuperación y Generación Aumentada)

Este documento sirve como guía para desarrolladores, detallando la función de cada archivo, la estructura de la base de datos vectorial Chroma y los principios de LangChain Expression Language (LCEL) utilizados.

I. Estructura de Directorios para Carga de Datos

El sistema RAG asume la siguiente estructura de directorios, crucial para la indexación y la gestión de temas/filtros:

```
.
├── libros/                # Directorio de Datos Fuente (DATA_PATH)
│   ├── tema_historia/     # Subcarpeta = FILTRO DE TEMA ('tema_historia')
│   │   └── documento_1.pdf
│   └── tema_ciencia/      # Subcarpeta = FILTRO DE TEMA ('tema_ciencia')
│       └── articulo_a.pdf
├── chroma_db_rag_recursive/ # Base de Datos Vectorial para estrategia Recursiva
├── chroma_db_rag_fija/    # Base de Datos Vectorial para estrategia Fija
├── processed_files.txt    # Log de archivos ya indexados
├── index_data.py          # Script de Indexación (Setup/Offline)
├── rag_utils.py           # Módulos de Lógica (LCEL/Modelos)
├── rag_streamlit_tts.py   # Frontend (Streamlit) y Control de Sesión
└── tts_component.html     # Componente TTS (JavaScript/HTML)
```

II. Base de Datos Vectorial Chroma

Chroma es la base de datos utilizada para almacenar los *embeddings* (vectores numéricos) de los fragmentos de texto (*chunks*).

¿Cómo funciona Chroma en este sistema?

1. **Persistencia:** Cada estrategia de segmentación (ej., Recursiva, Fija) tiene su propia carpeta de persistencia (ej., chroma_db_rag_recursive). Esto permite al desarrollador comparar las estrategias de RAG sin mezclar los datos.
2. **Metadatos (metadata):** Al indexar, a cada *chunk* se le adjunta un diccionario de metadatos. El campo crucial es **source**, que contiene la ruta completa del archivo de origen (ej., libros/tema_historia/documento_1.pdf).

3. **Filtros de Búsqueda:** Cuando el usuario selecciona un "Tema" en el frontend (ej., tema_historia), el sistema RAG aplica un filtro en la llamada a la DB de Chroma (db.as_retriever(search_kwargs={"filter": ...})). Este filtro asegura que la búsqueda solo se realice en los documentos cuyo metadato source contenga la cadena del tema seleccionado.
    - **Consulta sin Filtro:** select * from embeddings where vector_distance < threshold
    - **Consulta con Filtro:** select * from embeddings where vector_distance < threshold AND source LIKE 'libros/tema_historia/%'

III. Desglose de Archivos y Funciones

A. index_data.py (Procesamiento y Setup de la DB)

Este script se ejecuta *offline* para preparar los datos.

Función	Propósito Técnico
initialize_embeddings()	Carga el modelo HuggingFaceEmbeddings (paraphrase-multilingual-mpnet-base-v2). Esta es una función crítica que traduce el texto a un espacio vectorial de alta dimensionalidad.
_get_processed_files(log_path)	Función de utilidad para leer el log de archivos ya indexados. Esto previene el reprocesamiento, ahorrando tiempo y evitando duplicados en la DB.
get_db_path_for_strategy(strategy)	Mapea la clave de la estrategia (ej. "Recursive (Recomendada)") a la ruta de la carpeta de la DB (ej. ./chroma_db_rag_recursive/).
load_and_index_data(data_path, embeddings)	**Pipeline Principal de Indexación.** Itera sobre los archivos PDF: carga, divide el texto con la estrategia de *chunking* elegida, extrae metadatos (incluyendo la ruta completa como source), y finalmente llama a Chroma.from_documents para vectorizar y persistir los datos.

B. rag_utils.py (Lógica Central y Cadenas LCEL)

Contiene las funciones que definen la interacción entre los modelos y los datos, utilizando LangChain Expression Language (LCEL) para la orquestación.

Función	**Propósito Técnico**
initialize_embeddings()	(Compartida con index_data.py) Centraliza la definición del modelo de *embeddings* para asegurar la consistencia.
format_docs(docs: List[Document])	Función RunnableLambda utilizada en la cadena. Convierte la lista de objetos Document recuperados por Chroma en un solo *string* concatenado, listo para ser inyectado como contexto en el *prompt* del LLM.
format_sources(docs: List[Document])	Función RunnableLambda utilizada en la cadena. Extrae la propiedad source (ruta del archivo) de los metadatos de los documentos, generando una lista de fuentes citadas.
create_rag_chain(db, llm_model_name, theme_filter)	**Ensamblaje LCEL del RAG.** Define la tubería:
	1. Retriever: Configura el objeto db.as_retriever() con el filtro de metadatos (si se proporciona theme_filter).
	2. RunnableParallel: Divide la ejecución en dos ramas paralelas: **documents** (usa el *retriever*) y **question** (usa RunnablePassthrough para mantener la pregunta original).
	3. Segunda RunnableParallel: Utiliza el resultado de la recuperación para: a) Generar la **answer** (pasando los documentos al Prompt → LLM). b) Generar las **sources** (usando format_sources sobre los documentos).
create_summary_chain(llm_model_name)	Cadena LCEL simple para el modo

	Resumen: `PromptTemplate

C. rag_streamlit_tts.py (Frontend y Gestión de Sesión)

El controlador principal de Streamlit, gestiona la interfaz, la inicialización bajo demanda y la comunicación con el componente TTS.

Función	**Propósito Técnico**
load_tts_html_template(file_path)	Carga y cachea el contenido del archivo tts_component.html, previniendo múltiples lecturas del disco por recarga de Streamlit.
generate_tts_button(text_to_speak, key)	Inyecta el botón TTS. **Paso Crítico:** Antes de inyectar el HTML, usa json.dumps(cleaned_text) para asegurar que el texto de la respuesta del LLM (que puede contener saltos de línea y caracteres especiales) se serialice como un *string* válido de JavaScript, evitando errores de sintaxis en el HTML inyectado.
load_and_initialize_rag(selected_strategy, selected_theme)	**Función Cacheada (@st.cache_resource):** Carga la instancia específica de **Chroma DB** y las cadenas LCEL. El *hash* de la caché depende de la estrategia y el tema, asegurando que solo se recargue si el usuario cambia de DB/Filtro.
handle_user_input(prompt, mode)	Controla la ejecución de la cadena. Llama a st.session_state.rag_chain.invoke(prompt) o st.session_state.summary_chain.invoke(prompt) y actualiza el estado de st.session_state.messages.
main_app()	Configura la página, maneja el estado de sesión de Streamlit y define la estructura de la barra lateral y el chat principal.

D. tts_component.html (Módulo JavaScript del TTS)

Componente HTML/JS/CSS inyectado por Streamlit para manejar la interacción de audio de alta fidelidad.

Función	Propósito Técnico
fetchAudioFromAPI(text)	Maneja la llamada asíncrona (fetch) al *endpoint* de la API de **Gemini TTS**. El payload incluye el texto limpio, el modelo (gemini-2.5-flash-preview-tts) y la configuración de voz.
base64ToArrayBuffer(base64)	Utilidad JS crucial para decodificar los datos de audio Base64 que retorna la API, convirtiéndolos en un ArrayBuffer binario.
pcmToWav(pcm16, sampleRate)	**Función de Conversión de Audio.** El API retorna audio PCM16. Esta función usa los datos binarios PCM para construir un encabezado WAV válido, permitiendo que el navegador lo interprete como un archivo de audio estándar.
startPlayback()	Utiliza la AudioContext de la Web Audio API para decodificar el ArrayBuffer (que ahora tiene formato WAV) y crear un AudioBufferSourceNode para la reproducción directa y de baja latencia.
stopAudio()	Gestiona el control de reproducción, asegurando que cualquier fuente de audio anterior se detenga antes de iniciar una nueva.