UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

Implementación de modelo de lenguaje basado en RAG para asistencia en CityLab Biobío

POR

Kevin Alejandro Lagos Villagran

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero Civil en Telecomunicaciones.

Profesor Guía
Sebastián Godoy

Concepción,
Enero de 2025

© 2025 Kevin Lagos

© 2025 Kevin Alejandro Lagos Villagran
Ninguna parte de esta tesis puede reproducirse o transmitirse bajo ninguna forma o por ningún medio o procedimiento, sin permiso por escrito del autor.

Agradecimientos

Primero que nada quiero agradecer a mi familia y amigos por siempre estar ahí. A Citylab Biobio por la oportunidad de hacer esta memoria con ellos, especialmente al Diego por su buena disposición y buena onda. Y por ultimo, pero no menos importante, al resto de memoristas que trabajamos juntos en Citylab, al Francisco, el Javier, el Ricardo y todo el resto del laboratorio, gracias a ellos me daban ganas de seguir asistiendo y aprendiendo.

Gracias por todo

Resumen

City Lab Biobío es una iniciativa del MIT, auspiciada por la Gobernación del Biobío, que desarrolla y utiliza diversas herramientas para apoyar la toma de decisiones en el ámbito de la planificación urbana y el diseño espacial. La herramienta principal en este contexto es Cityscope, una maqueta de una porción de la ciudad sobre la cual se proyecta un mapa de colores que permite visualizar y comprender cómo se distribuyen distintos indicadores a lo largo del territorio urbano. Este proyecto despierta el interés de varias entidades, por lo que el laboratorio recibe visitas frecuentemente.

Para gestionar este flujo de visitas, se decide crear un asistente virtual capaz de responder de manera autónoma y eficaz a las preguntas de los visitantes. El asistente es una aplicación web en la cual un modelo de lenguaje (LLM) gestiona las preguntas y respuestas. A través de técnicas de Recuperación Aumentada con Generación (RAG), el modelo recibe documentos relevantes sobre City Lab, lo que le permite responder con información precisa y actualizada. Toda la lógica de las respuestas se maneja desde un servidor desarrollado en Python con la librería FastAPI, mientras que la interfaz del asistente está implementada en HTML, JavaScript y CSS.

Para verificar el correcto funcionamiento de la aplicación, se propone una prueba de estrés que simula un escenario donde múltiples dispositivos se conectan al mismo tiempo. Además, los integrantes de City Lab responsables de supervisar el proyecto validan las respuestas entregadas por el modelo. En el primer caso, se producen instancias donde no se generan respuestas, pero este problema se debe a fallos en la API de OpenAI y no a la lógica del servidor. Por ello, se implementan medidas para detectar cuándo ocurre este problema y notificar al usuario. Con respecto a la validación realizada por City Lab, se busca que las respuestas sean coherentes con la información existente y que no se revele información privada del laboratorio. Los casos en los que el modelo no responde de manera correcta ocurren por una falta de documentos o porque la recuperación de estos no es satisfactoria. Sin embargo, esto no se considera un error inherente del asistente, sino simplemente una falta de información. Ademas, el asistente no es capaz de revelar información privada, pues los documentos son seleccionados directamente por el laboratorio.

Abstract

City Lab Biobío is an initiative of MIT, sponsored by the Biobío Regional Government, that develops and utilizes various tools to support decision-making in urban planning and spatial design. The primary tool in this context is Cityscope, a model of a portion of the city on which a color-coded map is projected to visualize and understand how different indicators are distributed throughout the urban area. This project has garnered the interest of several entities, and the lab frequently receives visitors.

To manage this flow of visits, a virtual assistant is created to respond autonomously and effectively to visitors' questions. The assistant is a web application in which a language model (LLM) handles questions and answers. Through Retrieval-Augmented Generation (RAG) techniques, the model receives relevant documents about City Lab, enabling it to respond with accurate and up-to-date information. All response logic is handled by a server developed in Python with the FastAPI library, while the assistant's interface is implemented in HTML, JavaScript, and CSS.

To verify the correct functioning of the application, a stress test is proposed, simulating a scenario where multiple devices connect at the same time. Additionally, the City Lab team members responsible for overseeing the project validate the responses provided by the model. In both cases, the results are satisfactory.

Índice General

Índice de Figuras

Siglas

LLM Large Language Model

RAG Retrieval Augmented Generation

Se usaran los terminos LLM, modelo de lenguaje o modelo simplemente de manera intercambiable para evitar redundancia a la hora de desarrollar el informe.

1. Introducción

1.1. Introducción general

City Lab Biobío es una iniciativa nacida en el Instituto Tecnológico de Massachusetts o MIT[2], la cual busca generar herramientas que ayuden a mejorar la toma decisiones en ámbitos como la planificación urbana y el diseño espacial[1], la herramienta principal es el Cityscope, una maqueta de ciudad donde se proyectan distintos tipos de indicadores, como pueden ser la densidad de población, distancia a distintos tipos de servicios, etc. Debido a la innovación que presenta el laboratorio, tanto entidades gubernamentales como privadas se interesan por esta idea, por lo que es normal que vayan visitas varias veces a la semana. Durante estas visitas se da la oportunidad de que la gente pregunte y resuelva sus dudas acerca de los proyectos del laboratorio. Varias de estas preguntas se repiten entre visitas, lo que ha dado pie a crear una solución para poder responder todas estas preguntas de forma eficiente, masiva y autónoma.

1.2. Estado del arte

Con la popularidad de modelos de lenguaje como ChatGPT de OpenAI o Claude de Anthropic aumentando cada vez más [3], es normal que el campo de asistentes virtuales, que se nutre directamente de estos, esté siendo revolucionado de la misma forma. Además de los modelos ya mencionados, existen todo tipo de tecnologías nuevas, las cuales buscan ya sea una mejora en rendimiento, mejorar las capacidades de los modelos o reducir la cantidad de recursos necesarios para usar estos modelos. A continuación se presentarán los últimos avances en este campo al momento de escribir este informe.

Los modelos multimodales o MLLM [4] son modelos de lenguaje capaces de combinar distintos tipos de datos, como imágenes, texto, voz o audio, para así generar respuestas más completas a los usuarios. El modelo más notable de este tipo es el GPT-4 V de OpenAI, capaz de digitalizar notas desde una imagen o entender imágenes complejas.

Como contraparte de estos modelos más grandes y potentes están los modelos como Distil-

BERT, TinyBERT y ALBERT, los cuales buscan reducir su tamaño y así su demanda en la máquina que los esté corriendo, sin perder una cantidad significativa de poder. Para lograr esta compresión del número de parámetros se utilizan distintas técnicas [5], como el pruning, el cual se basa en eliminar parámetros redundantes o que aportan poca información, otra técnica que es bastante usada es la cuantización del modelo, donde se reduce la precisión del modelo para disminuir significativamente el tamaño de este. Estos modelos más pequeños pueden ser usados por dispositivos móviles o IoT.

El aprendizaje federado [6] es una técnica usada para aumentar la personalización de los modelos de lenguaje mediante el uso de datos locales de los dispositivos donde los modelos están recibiendo preguntas, de esta forma se pueden generar preguntas adaptadas a las preferencias y gustos del usuario.

Una forma de mejorar las respuestas de un asistente virtual es usar retroalimentación humana directa, esto permite que el modelo tenga información en tiempo real con la cual aprender, lo que le permite una mayor capacidad para alinearse con las expectativas humanas. Otra técnica que se está implementado para mejorar las respuestas es la de implementar modelos con memoria persistente o MANN [7], los cuales tienen la capacidad de almacenar y recuperar información de interacciones anteriores. Esto permite mantener la coherencia y precisión de las respuestas entre conversaciones con el mismo usuario, esta técnica en específico es usada en asistentes virtuales empresariales, donde es importante mantener la continuidad de la conversación durante varias sesiones de interacción.

A la hora de diseñar un asistente virtual hay que tomar en cuenta distintos parámetros para elegir el tipo de modelo de lenguaje que calza mejor con el problema a resolver, como puede ser si el modelo va a estar corriendo de manera local o si se usara una API para generar las respuestas, que tipo de datos se le podrá entregar al asistente como entrada y que tipo de respuestas se espera que produzca, etc. En este caso el asistente usará la API de OpenAI y tanto la entrada como la salida del LLM será solo texto, por lo menos la página principal del asistente. De este modo no será necesario utilizar modelos multimodales o modelos comprimidos y en vez de depender de un modelo con memoria persistente se usarán técnicas para mantener la memoria de la conversación en una sesión única para cada usuario, de este modo no hay que preocuparse de guardar cada conversación para que el asistente tenga acceso a estas, pues esto no es lo que se espera del LLM.

1.3. Definición del problema

La estructura del asistente debe ser tal que se pueda acceder a el estando en el laboratorio y en la misma red que el computador que donde estará desplegada la aplicación, además es necesario que el servidor que maneja las preguntas y respuestas sea capaz de manejar solicitudes de manera asincronica, pues en la mayoría de los casos serán varias personas utilizando el asistente al mismo tiempo. Por ultimo, hay que habilitar un método para actualizar el banco de documentos del cual sacara información el LLM.

1.4. Objetivos

1.4.1. Objetivo general

Desarrollar un asistente virtual para City Lab usando un LLM y RAG, el cual sea capaz responder las preguntas de los visitantes de manera autónoma y automática.

1.4.2. Objetivos específicos

- Definir requerimientos de software para el correcto funcionamiento del asistente.
- Implementar el asistente de formal local.
- Desplegar y validar el asistente en la red de City Lab Biobio.

1.5. Alcances y limitaciones

1.5.1. Alcances

El alcance de este proyecto será tener la aplicación web desplegada permanentemente en la red del laboratorio. Esta aplicación estará disponible para cualquier persona que esté conectada a la red y será capaz de responder sus preguntas.

1.5.2. Limitaciones

Las limitaciones de este proyecto se reducen a limitaciones técnicas y de información, las primeras hacen referencia al tipo de modelo de lenguaje que se usara para responder las preguntas, la elección de este modelo depende de factores como el costo computacional, la disponibilidad de los modelos, si es que estos son de pago, etc. Por otro lado, la calidad de las respuestas va a depender también de la cantidad de documentos que se puedan recopilar para alimentar al LLM, pues el modelo es capaz de inferir, pero a la hora de responder preguntas específicas depende en gran medida de estos documentos.

1.6. Metodología

El componente principal del asistente será un modelo de lenguaje, complementado con RAG para aumentar su conocimiento sobre City Lab. Para lograr esto, es necesario recopilar documentos relevantes sobre el laboratorio, como PDFs, documentos de Word, texto plano, PowerPoints, entre otros. Estos documentos formarán parte del Vectorstore o banco de datos que contendrá toda la información que el asistente necesitará.

Una vez que se tiene el banco de documentos es necesario inicializar al LLM y conectarlo a este, para esto se creara un script en Python donde el modelo de lenguaje y la función para llamar a este estarán dentro de una clase, de modo que cada vez que sé instancia un objeto de esta se creara una conexión con la API del LLM. Esta clase será instanciada en el Backend de la aplicación y será la encargada de responder las preguntas de los usuarios, en este Backend es donde se manejaran las peticiones HTTP y conexiones por websocket. El Frontend de la aplicación será la parte que ve el público, se usara HTML, JavaScript y CSS para diseñar la página y para conectarse por medio de Websocket al Backend.

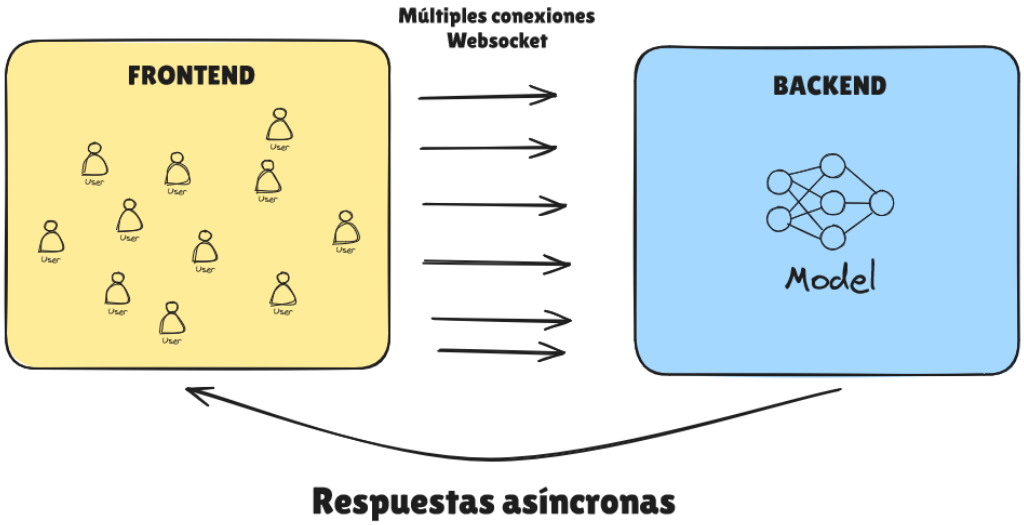

Fig. 1.1: Conexión por Websocket

Una vez que se tiene la aplicación funcionando localmente, es necesario preparar el directorio donde están alojados los archivos para ser transformados en un contenedor de Docker. Para esto se creará los archivos Dockerfile y docker-compose, el primero sirve para crear la imagen del servicio o contenedor y el segundo se usa para levantar contenedores con configuraciones predeterminadas, estableciendo variables de entorno, nombres de los servicios que estarán dentro del contenedor, las relaciones entre estos servicios, si uno tiene que levantarse antes que otro, etc. En este caso se hará para facilitar el levantar y bajar el contenedor, pues con este archivo el proceso se reduce a un solo comando en la terminal. Ya teniendo estos archivos, lo único que hay que hacer es levantar el contenedor en el computador principal de City Lab Biobio.

2. Marco Teórico

2.1. LLM y RAG

El componente principal y más importante del asistente serán los LLM[8], cuyas siglas significan modelos de lenguaje grandes. Un modelo de lenguaje es un modelo de deep learning que se entrena con una gran cantidad de datos de manera no supervisada, su tarea es predecir la siguiente palabra al entregarle un texto o palabra como entrada. Al contar con redes neuronales y transformadores en su arquitectura, el modelo es capaz de inferir la gramática, idiomas y la relación entre las palabras. Los modelos más poderosos pueden llegar a tener billones de parámetros y son capaces traducir y generar texto fácilmente. La forma en la que estos modelos reconocen la relación entre palabras es mediante los embeddings o incrustaciones de palabras, donde se toman palabras o frases de la entrada y se transforman en vectores numéricos multidimensionales, donde palabras que sean similares tendrán una distancia vectorial menor entre ellas.

El RAG[9] es una técnica que busca expandir la habilidad de los LLM de realizar tareas fuera del contexto en el que fueron entrenados. Esta técnica se basa en la creación de una base de datos supervisada a la cual el modelo pueda acceder para responder de manera precisa y eficiente. Para llevar a cabo esto es necesario tener la información que se le quiere alimentar al LLM en formato de texto, pueden ser documentos como PDFs, Word, PowerPoints, etc. Una vez que se tiene este banco de información se usan embeddings para transformarla en vectores numéricos, este banco se llamara vector store. De esta forma, cuando se le haga una petición al modelo, este transformará la entrada en un vector numérico y se comparará con el Vectorstore, donde finalmente se recuperará la información más relevante. De este modo se pueden actualizar los conocimientos de los LLM sin necesidad de reentrenarlos.

2.2. Lenguajes de programación

2.2.1. Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, conocido por su sintaxis clara y legible. Su diseño enfatiza la legibilidad del código, lo que facilita su aprendizaje y uso. Python soporta múltiples paradigmas de programación, incluidos el imperativo, orientado a objetos y funcional. Además de esto, cuenta con una gran cantidad de librerías y paquetes de terceros, los cuales permiten un desarrollo más rápido y eficiente. Langchain es uno de estos paquetes y es el pilar de esta aplicación web, nos da herramientas y abstracciones para crear aplicaciones con LLM como base.

El "Backend" de esta aplicación será creado con este lenguaje. Para manejar las preguntas y respuestas del asistente se usará la librería FastAPI para crear un servidor, FastAPI es un framework web de Python diseñado para la creación de API rápidas y eficientes, conocido por su alto rendimiento, comparable al de frameworks como Node.js y Go, gracias a su uso de Starlette para la gestión de conexiones asíncronas y Pydantic para la validación y serialización de datos. Aquí es donde llegaran las preguntas de los usuarios y se despacharan las respuestas generadas por el LLM.

2.2.2. Javascript, HTML y CSS

JavaScript[17] es un lenguaje de programación interpretado, orientado a objetos y basado en prototipos. Es esencial para el desarrollo web, ya que permite la creación de interfaces de usuario interactivas y dinámicas. JavaScript se ejecuta en el navegador del cliente, lo que reduce la carga del servidor y permite respuestas rápidas a las interacciones del usuario. Además, puede integrarse con HTML y CSS para manipular el DOM (Document Object Model) y actualizar el contenido visual en tiempo real.

HTML[18] (HyperText Markup Language) es el lenguaje estándar para la creación de documentos web. Define la estructura básica de una página web utilizando etiquetas que organizan el contenido en elementos como encabezados, párrafos, enlaces, imágenes, y más. HTML es un lenguaje de marcado, lo que significa que su función principal es describir la estructura y el significado del contenido en lugar de su apariencia.

CSS[19] (Cascading Style Sheets) es un lenguaje de estilos utilizado para describir la presentación visual de un documento HTML. CSS permite separar el contenido de la estructura (definida por HTML) de su presentación, facilitando la aplicación de estilos como colores, fuentes, márgenes, y disposición de elementos. CSS sigue un modelo de cascada, lo que significa que los estilos pueden ser heredados y sobreescritos por reglas más específicas, permitiendo un control preciso sobre la apariencia de una página web.

Estos tres lenguajes serán usados en unísono para crear el frontend del asistente virtual, el cual equivale a la parte visual con la que el usuario va a interactuar y donde se generarán las preguntas para luego ser enviadas al backend.

2.3. Otros

2.3.1. Websockets

Una conexión por websocket implica un canal bidireccional que persiste en el tiempo, a diferencia de una petición HTTP, la cual necesita hacer el handshake cada vez que se ocupa un método, el websocket mantiene la conexión abierta después del handshake. Estas características hacen que los websockets sean más adecuados que las peticiones HTTP a la hora de necesitar comunicación en tiempo real.

2.3.2. API

Una API (Application Programming Interface) es un conjunto de definiciones y reglas que permite que dos aplicaciones o programas interactúen entre sí y compartan información. Las API actúan como intermediarios, de modo que cada aplicación interactúa solo con la API y no directamente entre ellas. Esto significa que no necesitan conocer la estructura o implementación interna de la otra aplicación, ya que la API se encarga de gestionar y, si es necesario, transformar la información para asegurar la compatibilidad entre las partes.

2.3.3. Backend

El backend hace referencia a la parte del desarrollo de software que gestiona la lógica interna, el procesamiento de datos y la comunicación con la base de datos y otros servicios del sistema. Generalmente, está compuesto por servidores, bases de datos, y API que permiten que el sistema funcione de manera fluida y segura, manejando peticiones desde el frontend y respondiendo con los datos solicitados o realizando operaciones específicas.

2.3.4. Frontend

El frontend es la capa visible e interactiva de una aplicación web, que se presenta directamente al usuario. Su propósito es ofrecer una interfaz intuitiva que permita a los usuarios interactuar con la aplicación de manera eficiente. Los desarrolladores de frontend trabajan con tecnologías como HTML, CSS, y JavaScript (o frameworks como React, Angular, Vue) para construir interfaces que se adapten a diferentes dispositivos y que proporcionen una experiencia de usuario óptima. Es responsable de la comunicación con el backend, usualmente a través de API REST o WebSockets, para obtener y presentar datos en tiempo real.

2.3.5. Docker

Docker es una plataforma de contenedorización que permite a los desarrolladores empaquetar aplicaciones y sus dependencias en contenedores, asegurando que puedan ejecutarse de manera consistente en cualquier entorno. Los contenedores son entornos ligeros y aislados que incluyen todo lo necesario para ejecutar una aplicación, desde el sistema operativo hasta las bibliotecas y el código de la aplicación. Docker facilita el despliegue y la escalabilidad, reduciendo problemas relacionados con la configuración de entornos y la compatibilidad. En el contexto del desarrollo y despliegue de software, Docker es una herramienta fundamental para la infraestructura moderna, como microservicios y sistemas distribuidos.

3. Desarrollo

3.1. Construcción del asistente

3.1.1. Set-up para LLM y RAG

Para manejar tanto al LLM como la recuperación de documentos se usara la librería de Python Langchain[10]. Lo primero que hay que hacer es conseguir una API key para poder conectarnos a algún modelo de lenguaje, en este caso se usara la API de OpenAI[**?**] y el modelo GPT-3.5-turbo.

ChatOpenAI: Esta clase es la integración de la API de OpenAI en Langchain, al crear un objeto especificando el LLM que se quiere usar, la temperatura y la API key se pueden usar los métodos de este para acceder a los servidores de OpenAI. La temperatura hace referencia a que tan creativas son las respuestas del LLM, donde una temperatura más alta da luz a respuestas menos predecibles. Una vez teniendo el objeto instanciado podemos acceder al modelo especificado mediante métodos como **invoke** y **ainvoke**. Estas funciones mandan nuestra pregunta a los servidores de OpenAI y después retornan la respuesta del modelo como una string, ainvoke es simplemente la versión asincrónica de invoke.

Para crear nuestro Vectorstore, o en otras palabras, los documentos de interés en formato de vectores numéricos, se creó un script con funciones encargadas de añadir distintos tipos de documentos a este banco de documentos. El proceso que siguen las funciones es bastante similar, primero se usa alguna librería de Python para acceder al texto contenido en los documentos y después se toma el texto y se parte en chunks o pedazos de texto.

Si se quisiera agregar, por ejemplo, un PDF al Vectorstore primero se usaría la función **get-pdf-split** para leer el archivo y convertirlo en chunks o pedazos de texto, luego se llamaría a la función **text-to-vector** para añadir estos pedazos al vectorstore. A esta función se le entrega el nombre del Vectorstore para que pueda cargarlo dentro del script y actualizarlo, en el caso de no exista un Vectorstore con el nombre especificado se creara uno con este nombre y se añadirán los pedazos de texto a este. Implementaciones de este proceso se puede ver más adelante cuando se explique como funciona el Backend.

Ahora que tenemos nuestro LLM y vectorstore funcionando, es necesario crear la cadena que va a caracterizar el comportamiento de nuestro asistente. En langchain una cadena o chain hace referencia a los pasos que sigue el LLM cuando le llega una pregunta. En este caso, el asistente contará con recuperación de documentos (RAG) e historial de chat.

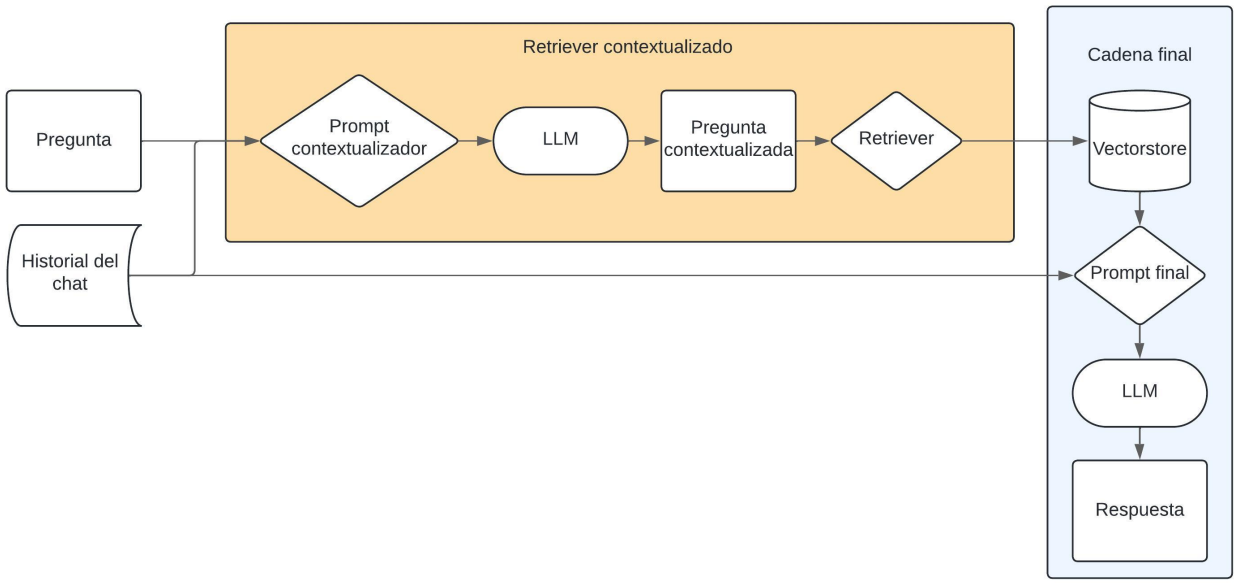

Fig. 3.1: Cadena final

Cuando llega una pregunta lo primero que pasa es que el LLM toma esta y la reformula usando el historial del chat como contexto, es esta nueva pregunta ya contextualiza la que es usada para hacer la recuperación de documentos, lo que permite mayor precisión y evita que se pierda el sentido de la conversación.

```
1    promptContextualizador= """Teniendo en cuenta el historial del chat y la
    ultima pregunta del usuario \
2    la cual puede referenciar contexto del historial del chat, formula una
    nueva pregunta que pueda ser \
3    entendida sin necesidad del historial del chat. NO respondas esta
    pregunta, simplemente \
4    reformulala si es necesario o de lo contrario devuelvela sin cambios.
5    """
```

Ahora se toman tanto los documentos recuperados como el historial del chat para crear un prompt final, el cual tendrá la siguiente forma.

```
1  qaSystemPrompt= """ Eres un asistente virtual para Citylab Biobio.\
```

```
    Responde las preguntas de la mejor manera posible basandote en el
    contexto proporcionado. \
    Si no sabes la respuesta, di que no la sabes simplemente.\
    limita las respuestas a un maximo de 40 palabras.\

    {context}"""
```

Finalmente, el LLM seguirá las instrucciones estipuladas en este prompt y responderá la pregunta del usuario basándose en el contexto proporcionado.

3.1.2. Backend

Para manejar las conexiones de usuarios al asistente virtual se usará la librería FastAPI de Python.

FastAPI[12] es la clase principal de la librería, una vez instanciado un objeto tenemos acceso a decoradores que nos permiten manejar peticiones HTTP y Websocket. Un decorador es una función que envuelve a otra, tomando a esta otra función entera como un argumento. La forma de manejar peticiones se basa en crear una función que se ejecutara cada vez que llegue un cierto tipo de método HTTP a un cierto endpoint, para después envolver a esta función en el decorador respectivo y usar el endpoint como argumento del decorador. Un posible ejemplo sería: **@app.get(“/home") ...**, este decorador iría antes de la función que se ejecutara cada vez que llegue una petición GET al endpoint **“/home"**.

Si se quieren servir documentos desde el servidor es necesario montar un directorio estático dentro del entorno de FastAPI, para esto se usará la clase **StaticFiles** usando la ruta de la carpeta que contiene los documentos estáticos que se le quieren servir al cliente, al crear este directorio se tiene que especificar una ruta donde va a estar alojado este dentro del servidor. Si una aplicación alojada en este servidor tiene que acceder a un documento estático, lo hará desde esta ruta.

En el caso de que se quiera entregar un archivo HTML como respuesta a una petición HTTP GET es necesario que la función que maneje esta respuesta retorne un objeto de la clase **FileResponse** tomando la ruta del archivo que se quiere servir como argumento, como este archivo está en el mismo directorio que el servidor y se está pidiendo desde la misma máquina se puede usar directamente la ruta local, pero cualquier proceso que sea de parte del cliente tendrá que usar la ruta del directorio estático de FastAPI.

Las conexiones entre el usuario y el servidor se harán mediante websockets, no peticiones HTTP, pues se necesita que la comunicación sea en tiempo real. Para manejar las conexiones por Websocket se usará el decorador **Websocket** de la misma forma que los decoradores de los métodos HTTP y dentro de la función que este envuelve se definirá la lógica para manejar las conexiones de clientes. Dentro de esta función se aceptaba la conexión al Websocket, se creara un código único para cada conexión y se entrara a un loop donde se estará escuchando y esperando de manera asíncrona que el cliente mande un mensaje, hasta el momento que llegue información la ejecución de código va a estar frenada y en el momento que llegue una pregunta se va a llamar a la función **respuesta-LLM** usando la pregunta que mando el usuario y el código único como argumento, el código será usado como **session-id** para mantener el historial de chat de cada cliente.

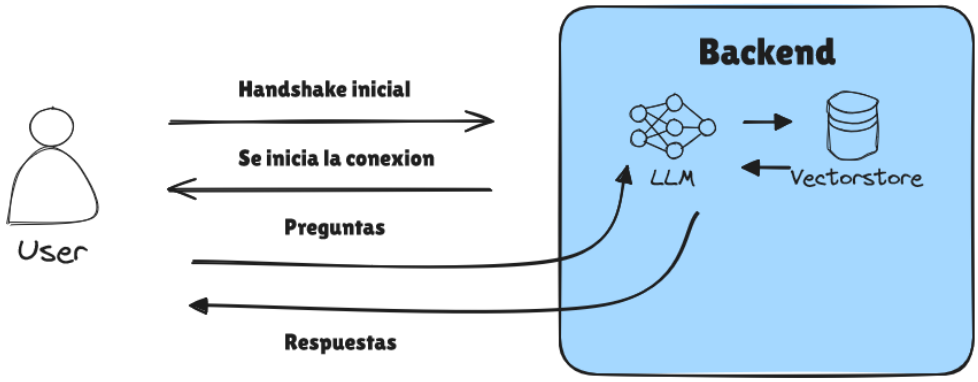

Fig. 3.2: Conexión por Websocket al Backend

Con estos endpoints ya es posible establecer la comunicación entre backend y frontend, pero también es necesario tener una forma de agregar documentos al Vectorstore, pues a medida que pase el tiempo va a ser necesario que el banco de documentos que suplemento al LLM se vaya actualizando. Para esto se habilitará un endpoint donde se pueden mandar archivos usando la petición HTTP POST. Estos archivos se procesarán usando las funciones presentadas en la sección anterior.

Una vez llegan archivos a "/upload" se extrae el texto de estos mediante funciones específicas para el tipo de documento. Hasta el momento se soportan PDFs, documentos de texto, PowerPoint y documento Word, pero es posible añadir más tipos si fuera necesario.

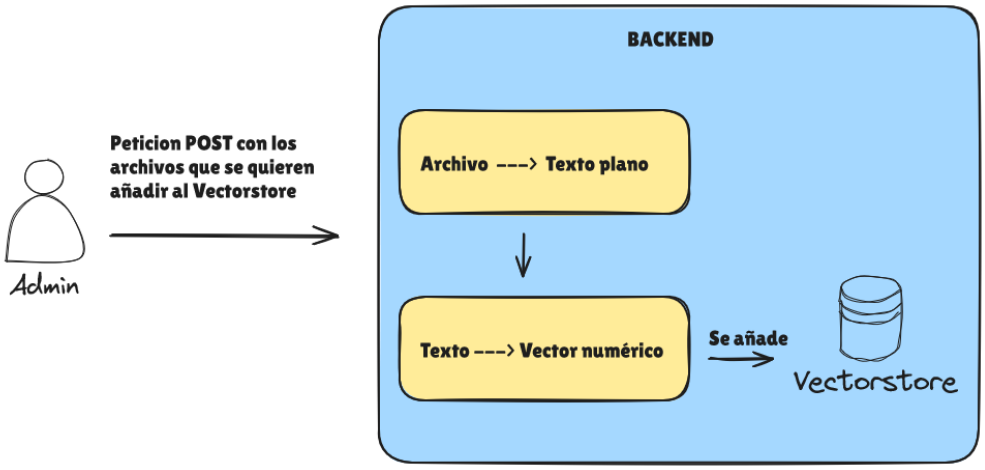

Fig. 3.3: Adición de archivos al Vectorstore

3.1.3. Frontend

La página principal contendrá el chat donde los usuarios pueden hacer sus preguntas, el historial de mensajes y una pestaña con preguntas frecuentes de los visitantes de Citylab.

Cuando los clientes entren al URL que contiene al asistente, se hará una petición HTTP GET al endpoint "/" y el servidor enviará el archivo HTML que contiene la estructura de la página. Este archivo importará un script JavaScript y una lista de estilos CSS desde el directorio estático que se creó en el Backend, el script contendrá la lógica de la conexión por Websocket al servidor y cualquier tipo de interactividad de la página.

Para levantar el Websocket cada vez que se conecta un cliente es necesario usar la API **Websocket** de JavaScript, la cual nos proporciona una interfaz con herramientas para manejar esta conexión. Al inicializar un objeto **Websocket** tenemos acceso a distintos métodos y manejadores de eventos, los primeros son funciones que se pueden llamar para realizar cambios al Websocket y los segundos son funciones que se ejecutan automáticamente cuando ocurre un evento, es necesario definir estas funciones para manejar de manera correcta la conexión desde el lado del cliente.

- **socket.onopen :** Cuando se abre la conexión. Se envía un mensaje a la consola denotando esta información.
- **socket.onmessage :** En el caso de que llegue una respuesta desde el servidor. Esta se guardará en una variable y se creará una nueva división en el HTML donde irá esta

respuesta.

- **socket.onclose :** Cuando se cierra la conexión. Se registra si él cierra fue abrupto o no en la consola.
- **socket.onerror :** En caso de algún error. Se registra la causa del error.

Una vez definidos los manejadores de eventos del Websocket es necesario definir una función que maneje el evento de enviar un mensaje, para esto se creó la función **onSubmit**. En esta función se envía el mensaje del usuario al Backend y se crea una división para almacenar este mensaje de la misma forma que el **onMessage** del Websocket.

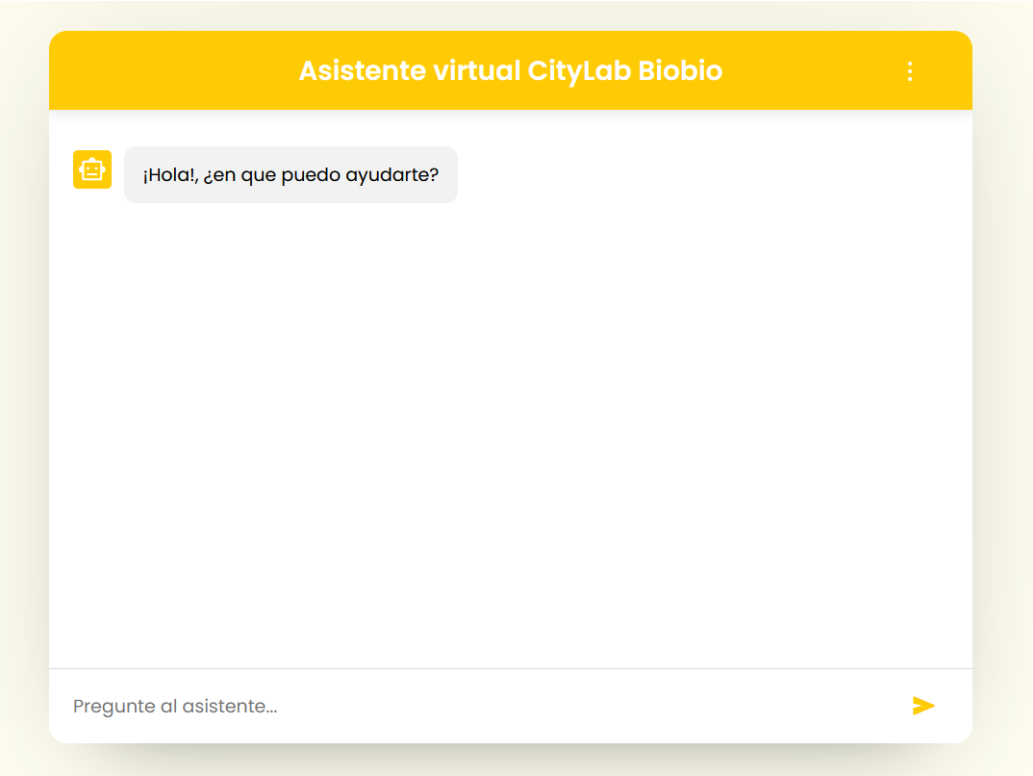

Fig. 3.4: Diseño de la pagina principal

3.1.4. Sección para administradores

Además de la función principal del asistente de responder las preguntas de los visitantes, se quiso explorar la posibilidad de que este interactuara con otras aplicaciones dentro de Citylab. Esta sección estará alojada en un endpoint diferente a la página principal y se necesitará de una contraseña para entrar. Para manejar las interacciones entre API se usará la librería Langgraph,

derivada de Langchain, la cual está especializada para trabajar con agentes o LLM capaces de ejecutar código mediante prompts. Langgraph permite crear aplicaciones con LLM donde la estructura de estas está representada por grafos, donde cada nodo representa un posible estado, pasar de un estado a otro es representado por una decisión del LLM y las condiciones de esta pueden ser definidas para amoldarse al problema que se quiere resolver.

En este caso solo necesitamos que el asistente use herramientas cuando se le pregunte algo que tenga que ver con los proyectos de Citylab, en específico el Cityscope. Se crearán dos herramientas, una para controlar el mapa que se está mostrando y otro para cambiar el estado de las placas de este.

Antes de entrar en detalle de como funcionan estas herramientas hay que entender como funciona el Cityscope junto con las aplicaciones y API que lo rodean. Empezando con la parte electrónica tenemos que el Cityscope está conformado por siete sensores RFID, los cuales son capaces de detectar ondas de radio, en conjunto con estos detectores la mesa contiene catorce placas que representan una parte de la ciudad, siete representan el estado actual y las otras representan posibles cambios que se puedan hacer en un futuro, en la parte inferior de estas placas se encuentra unas etiquetas RFID con un código único para diferenciar una placa de la otra. Todos los detectores RFID están conectados a un Arduino, el cual transforma el código que detectan en un número que representa a la placa y una vez que hay siete placas se manda un arreglo con la combinación de placas al Backend del Cityscope. Dentro de este se recupera la imagen correspondiente a las placas que se tiene que proyectar en el Cityscope. Cada vez que haya un cambio de placas se actualiza el arreglo y por ende la imagen proyectada. Además de cambiar la placa físicamente también se puede hacer este cambio mediante una petición HTTP al Backend especificando el arreglo de placas que se debe mostrar, de esta misma manera se puede cambiar el tipo de mapa que se está mostrando.

Langchain proporciona varias formas de crear herramientas, en este caso se eligió crear estas como sub-clases de **BaseTool**, de forma que esta nueva clase herede los métodos y atributos de **BaseTool**. Además de los parámetros de la superclase la herramienta debe definir un set de nuevos parámetros y métodos.

- **Name :** Es el nombre de la herramienta, debe comunicar claramente su propósito.
- **Description :** En este campo va la descripción del funcionamiento de la herramienta. El como, cuando y como se debe usar.
- **args-schema :** Representa la estructura de los argumentos de la herramienta, se deben

definir clases especificando el tipo de dato que se acepta y una descripción para que el LLM sea capaz de extraer estos argumentos de la pregunta.

- **-run :** Lógica que se ejecuta cuando el modelo de lenguaje elije usar la herramienta.

Teniendo esto en cuenta, lo primero que hay que hacer es definir los argumentos necesarios para el **args-schema** para las dos herramientas que se crearan. En el caso del cambio de mapas, el único argumento necesario será el mapa que pidió el usuario, por lo que se especificara esto en la descripción. En el caso del cambio de placas, el argumento que se pide es el número de la placa. Una vez teniendo nuestras clases especificando el esquema de los argumentos hay que crear las herramientas, los nombres de las herramientas serán "Cambio de mapas" y "Cambio de placas" respectivamente, las descripciones explicaran que hace cada herramienta y cuando debe usarse. Ahora es necesario definir la lógica que seguirá la herramienta cuando se llame, ambas herramientas tienen un funcionamiento parecido, pues la forma de cambiar ya sea mapa o placa en el Cityscope es mediante una petición HTTP, por lo que solo es necesario usar los argumentos definidos en el esquema y mandar una petición a la API que se encarga de los cambios.

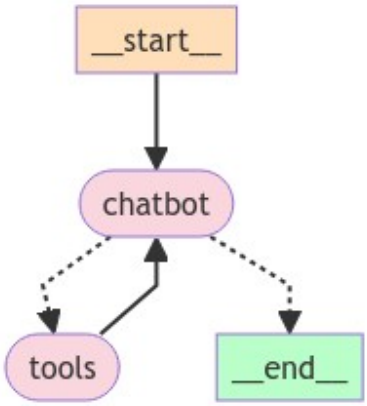

Fig. 3.5: Grafo que representa el flujo del agente

El siguiente paso es poder usar la voz para que el asistente ejecute los comandos que se requieran, para esto se usara un modelo capaz de transformar audio a texto para transformar el comando del usuario en un prompt para el LLM. Para esto se habilitará una sección dentro de la página de admins donde el usuario podrá enviar un comando por voz al Backend mediante una petición HTTP POST.

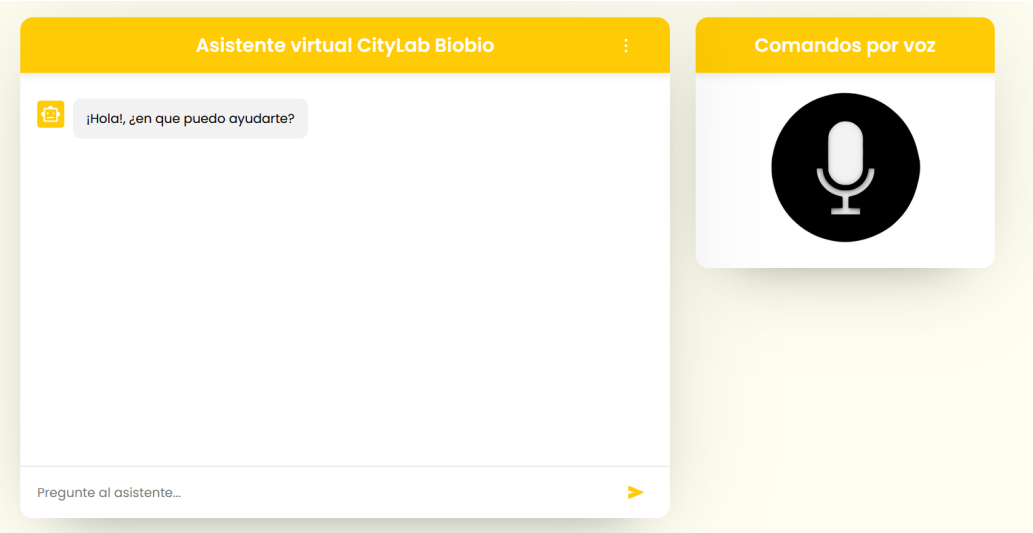

Fig. 3.6: Seccion de comandoz por voz

Una vez llega la información al backend esta se guarda como un archivo .wav y es transcrito a texto mediante el uso de un modelo de reconocimiento de voz. Ya teniendo el comando como un string se puede llamar al modelo de forma normal.

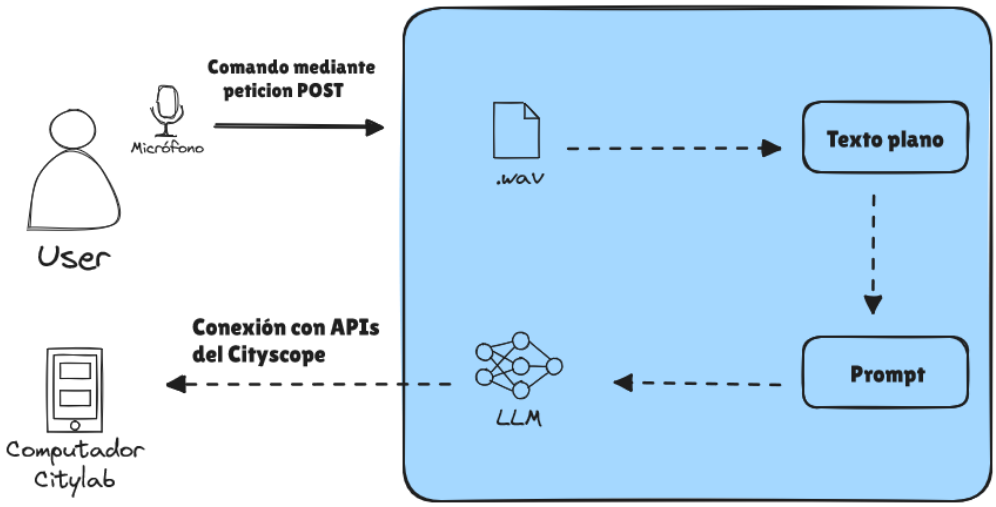

Fig. 3.7: Funcionamiento de comandos por voz

3.1.5. Migración a Docker

Una vez el proyecto está funcionando correctamente de forma local, es necesario organizar el directorio de este para migrar la aplicación a Docker. Esto es necesario, pues al tener el proyecto en formato de contenedor nos aseguramos de que este podrá funcionar en cualquier

tipo de máquina, pues Docker se encarga de estandarizar todos los parámetros necesarios para que nuestra imagen sea totalmente replicable.

Primero es necesario crear nuestro Dockerfile, aquí se establecen los parámetros de nuestra imagen mediante palabras clave de Docker:

- **FROM python:3.12.2 :** Aquí se establece la imagen base de nuestro proyecto, la cual es la versión 3.12.2 de Python.
- **WORKDIR /app :** Se establece el nombre del directorio donde estarán alojados los archivos del proyecto.
- **COPY requirements.txt /app/ :** Se copia el archivo requirements.txt del directorio original, aquí están todas las librerías de Python necesarias para que funcione el asistente, junto con sus respectivas versiones.
- **RUN pip install –no-cache-dir -r requirements.txt :** Este comando ejecuta el comando pip install en la terminal del contenedor para instalar las librerías de Python. Las flags –**no-cache-dir** y -**-r** hacen referencia, respectivamente, a que no se tomara él cuenta el caché del contenedor en el caso de que se vuelva a crear desde cero y que se está pasando un archivo requirements.txt como argumento.
- **COPY . /app/ :** Aquí se copian todos los archivos del directorio original al directorio de trabajo del contenedor, excepto los archivos especificados en el archivo .dockerignore, todos los archivos dentro de este serán totalmente ignorados por Docker.
- **EXPOSE 12500 :** Se expone el puerto 12500 para que pueda ser accedido desde otras máquinas
- **CMD [“uvicor”, "main:app", “–host”, "0.0.0.0", “–port”, "12500"] :** Este es el comando que levanta el servidor que entrega la información a los clientes.

Una vez hecho este Dockerfile es necesario crear un archivo **docker-compose.yml** para facilitar la acción de levantar y bajar contenedores desde la terminal. Dentro de este archivo se especifica la ruta al Dockerfile, también se mapea un puerto del contenedor a uno de la máquina que lo está corriendo, en este caso **12500:12500**. Por último se establece un volumen entre nuestro directorio original y la carpeta de trabajo **/app** del contenedor, el volumen permite que cambios que se hagan en el directorio original se copien en el contenedor y viceversa. Una vez

teniendo esto listo, el contenedor se puede levantar usando el comando **docker-compose up –build**.

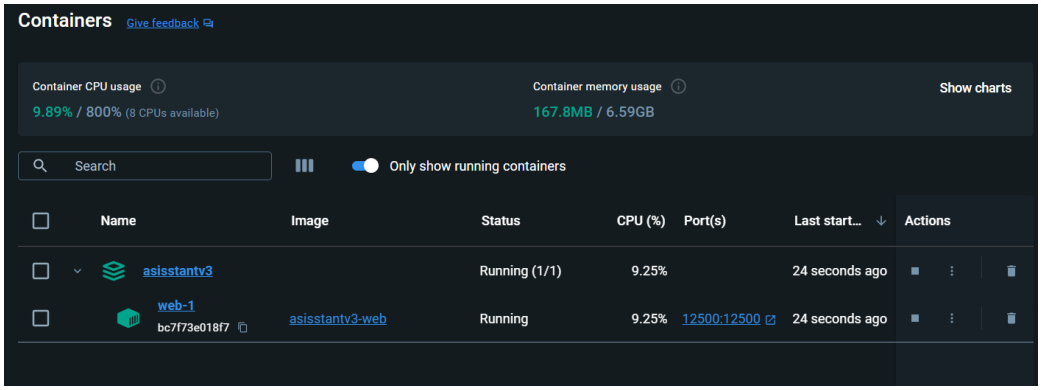

Fig. 3.8: Contenedor levantado en Docker

4. Resultados

4.1. Pruebas

Una vez que el contenedor está corriendo en el computador de City Lab, es necesario realizar las pruebas necesarias para asegurar el correcto funcionamiento del asistente.

Primero, se verifica que la aplicación sea accesible desde un dispositivo externo. Al intentar esto, surge el primer inconveniente: para permitir que un dispositivo externo establezca una conexión con el puerto donde se aloja el contenedor de Docker, es necesario habilitar la transmisión de datos desde y hacia este puerto en el firewall del dispositivo. Esto se configura mediante el siguiente comando:

“ New-NetFirewallRule -DisplayName “SERVICE-NAME” -Direction Inbound -LocalPort SERVICE-PORT -Protocol TCP -Action Allow ”, donde **SERVICE-NAME** y **SERVICE-PORT** son el nombre de la regla que estamos creando y el puerto que se está configurando, respectivamente.

A continuación, se realiza una prueba de estrés para asegurar que el servidor es capaz de manejar múltiples solicitudes de manera adecuada. Para ello, se solicita a varios miembros de City Lab que usen la aplicación de forma simultánea. Durante esta prueba, se mide el tiempo de respuesta y se monitorean posibles fallos en la lógica o en la gestión de errores del servidor. En total, siete dispositivos se conectan de forma simultánea. No se observa un aumento significativo en la latencia comparado con el uso individual, lo cual se debe al comportamiento asíncrono de las funciones que procesan las preguntas y respuestas. Las respuestas se generan con un tiempo promedio de entre dos y cuatro segundos. Los errores registrados se deben a la API de OpenAI, que en algunas ocasiones falla al generar una respuesta en un tiempo adecuado. Para mitigar este problema, se define un tiempo máximo de ejecución para la función encargada de realizar las solicitudes a OpenAI. Si se supera este tiempo, se genera un error que se notifica al usuario, indicando que intente enviar la pregunta nuevamente.

4.2. Validaciones

Una vez que la aplicación está operativa, es fundamental que los integrantes y responsables del laboratorio evalúen su desempeño para garantizar su eficacia. Para ello, se formulan al asistente las preguntas más frecuentes de los visitantes con el objetivo de evaluar tanto la precisión como la suficiencia de las respuestas proporcionadas. Además, se verifica que no se comparta información de carácter privado. Estas validaciones las realiza el personal a cargo de esta memoria de título.

Al verificar la calidad de las respuestas, hay ocasiones en las que estas no son del todo correctas o no son coherentes. Estos problemas se deben a una falta de información de parte del modelo, por lo que la solución es agregar más documentos al Vectorstore para contar con un banco de información más robusto. Por otro lado, el modelo no es capaz de revelar información privada, ya que los documentos recuperados con RAG son de carácter público o seleccionados específicamente por los integrantes del laboratorio.

5. Conclusiones

5.1. Conclusión

Basándose en las pruebas tanto del funcionamiento principal del asistente como de las herramientas para controlar el Cityscope se ha determinado que el asistente funciona de manera satisfactoria, capaz de responder a múltiples personas al mismo tiempo de manera consistente haciendo uso de los documentos en el Vectorstore. Las herramientas también funcionan de manera correcta al interactuar con el Cityscope, además, el hecho de poder ejecutar los prompts mediante voz le da una capa de inclusividad y facilidad a la hora de manipular estas herramientas.

Tanto la página principal del asistente como la de administradores han recibido una respuesta positiva por parte del equipo de City Lab Biobío, quienes han contribuido significativamente en el proceso de creación y prueba de las distintas funcionalidades del asistente. Además, se han asegurado de que estas funcionen correctamente y sean útiles a futuro.

La limitación principal del asistente es la cantidad de información a la cual tiene acceso, pues las respuestas dependen directamente de cuanta información relevante tenga el asistente a la hora de generar la respuesta. La solución para esto es ir actualizando el Vectorstore mediante el endpoint **/archivos**, de esta forma se podrán generar documentos para reforzar el conocimiento del asistente en algún campo en específico o agregar nuevos documentos nuevos que contengan información nueva, para mejorar la precisión de las respuestas y mantener al asistente al día con City Lab.

Además de permitir la actualización del Vectorstore también se dejará una extensa documentación explicando como funciona el código y como mantenerlo y/o refactorizarlo si fuera necesario.

5.2. Trabajo a Futuro

Si bien el asistente es totalmente funcional hay varios aspectos de este que se pueden mejorar, uno de ellos sería correr el LLM de forma local, de esta forma se reduciría el tiempo de respuesta de este, pues no tendría que hacer una petición a los servidores de OpenAI, además de esto sería costo 0 para el laboratorio. Otro punto de posible mejora seria darle más capacidades al modelo, por ejemplo que sea capaz de mostrar imágenes o devolver documentos en el chat, que pueda interactuar con más elementos del laboratorio, etc.

Bibliografía

[1] Página de CityLab Biobio. https://citylabbiobio.cl/nosotros/

[2] Pagina del MIT acerca del Cityscope. https://cityscope.media.mit.edu/

[3] De Angelis L, Baglivo F, Arzilli G, Privitera GP, Ferragina P, Tozzi AE, Rizzo C. ChatGPT and the rise of large language models: the new AI-driven infodemic threat in public health. Front Public Health. 2023 Apr 25;11:1166120. doi: 10.3389/fpubh.2023.1166120. PMID: 37181697; PMCID: PMC10166793.

[4] LLMs multimodales. https://medium.com/@cout.shubham/exploring-multimodal-large-language-models-a-step-forward-in-ai-626918c6a3ec

[5] Tecnicas para reducir el tamaño de LLMs. https://medium.com/@sasirekharameshkumar/understanding-compression-of-large-language-models-2ee3b8a350a

[6] LLMs con aprendizaje federado. https://www.techopedia.com/es/aprendizaje-federado

[7] LLM con memoria persistente. https://medium.com/@genebernardin/llm-persistent-memory-def236d0d63f

[8] Que es un LLM. https://aws.amazon.com/es/what-is/large-language-model/

[9] Retrieval Augmented Generation https://aws.amazon.com/es/what-is/retrieval-augmented-generation/

[10] Documentación de la librería de python “langchain” . https://python.langchain.com/docs/get_started/introduction, accedido: 5 de mayo de 2024.

[11] Documentacion para developers daiOpenAI. https://platform.openai.com/overview

[12] Documentacion de librería de Python FastAPI. https://fastapi.tiangolo.com/learn/

[13] Dibitonto, M., Leszczynska, K., Tazzi, F., Medaglia, C.M. (2018). Chatbot in a Campus Environment: Design of LiSA, a Virtual Assistant to Help Students in Their University Life. In: Kurosu, M. (eds) Human-Computer Interaction. Interaction Technologies. HCI

2018. Lecture Notes in Computer Science(), vol 10903. Springer, Cham. https://doi.org/10.1007/978-3-319-91250-9_9

[14] David N. Sousa, Miguel A. Brito, and Carlos Argainha. 2019. Virtual customer service: building your chatbot. In Proceedings of the 3rd International Conference on Business and Information Management (ICBIM '19). Association for Computing Machinery, New York, NY, USA, 174–179. https://doi.org/10.1145/3361785.3361805

[15] Tamayo, P. A., Herrero, A., Martin, J., Navarro, C. and Tranchez, J. M. (2020). Design of a chatbot as a distance learning assistant. Open Praxis, 12(1), 145–153. https://search.informit.org/doi/10.3316/informit.21938462222049

[16] Mekni, M. (2021) An Artificial Intelligence Based Virtual Assistant Using Conversational Agents. Journal of Software Engineering and Applications, 14, 455-473. https://www.scirp.org/journal/paperinformation?paperid=111666

[17] Documentacion de Javascript. https://developer.mozilla.org/es/docs/Web/JavaScript

[18] Documentacion de HTML. ttps://developer.mozilla.org/es/docs/Web/HTML

[19] Documentacion de CSS. https://developer.mozilla.org/es/docs/Web/CSS

[20] Documentacion de LangGraph. https://langchain-ai.github.io/langgraph/tutorials/