CARRERA DE ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL

MEMORIA DEL TRABAJO FINAL

Desarrollo de un chatbot especializado para optimizar la búsqueda de información en documentos propietarios

Autor:
Ing. Fabián Alejandro Massotto

Director:
Mg. Ing. Ezequiel Guinsburg (FIUBA)

Jurados:
Dr. Lic. Rodrigo Cárdenas (FIUBA)
Esp. Ing. Abraham Rodriguez (FIUBA)
Esp. Ing. Ariadna Garmendia (FIUBA)

Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires, entre abril de 2024 y abril de 2025.

Resumen

El presente trabajo aborda el desarrollo de un chatbot que interpreta consultas realizadas en lenguaje natural y ofrece respuestas precisas basadas en documentos empresariales previamente procesados. Su valor radica en la eficiencia operativa que se obtiene al optimizar el acceso a información crítica de una organización.

En esta memoria se detallan todas las etapas del desarrollo, desde la preparación de los datos hasta la evaluación del rendimiento del chatbot. Para lograrlo, se aplicaron conocimientos de procesamiento de lenguaje natural, modelos grandes de lenguaje e inteligencia artificial generativa.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción general

En este capítulo se introduce la problemática que motivó el presente trabajo, seguida de una breve descripción de la solución propuesta. A continuación, se expone el estado del arte de las tecnologías aplicadas. Finalmente, se detallan el alcance y los requerimientos necesarios para su implementación.

1.1. Marco de la propuesta

En un entorno empresarial, la eficiencia en la búsqueda de información es crucial para la productividad y el rendimiento de los empleados. Sin embargo, con la creciente cantidad de datos y documentos disponibles, encontrar información específica de manera rápida y precisa puede convertirse en un desafío.

La abundancia de fuentes de información, lejos de agilizar el trabajo, a menudo lo complica. En una empresa, es común que existan múltiples repositorios de documentos, políticas y datos históricos, pero la falta de centralización y la dificultad para identificar la fuente correcta suelen traducirse en pérdidas de tiempo significativas. En muchas ocasiones, se dedica más tiempo a la búsqueda de información que a la ejecución de las tareas en sí, lo que afecta tanto la productividad como la efectividad en la toma de decisiones.

El presente trabajo surgió como un desarrollo personal para abordar esta problemática. Su propósito es proponer una solución que facilite el acceso ágil y preciso a la información necesaria, de manera tal que optimice el uso del tiempo en un entorno con gran cantidad de fuentes de información.

Un chatbot especializado ofrece una solución prometedora al permitir a los usuarios realizar consultas en lenguaje natural y obtener respuestas de manera instantánea. Mientras que otros sistemas de inteligencia artificial ampliamente conocidos y utilizados destacan en su capacidad para generar respuestas generales basadas en un amplio conocimiento del lenguaje, el presente trabajo se distingue por su capacidad para trabajar con documentos altamente específicos (y potencialmente privados). Esto le permite ofrecer respuestas adaptadas al contexto interno de la organización, que no podrían obtenerse mediante el uso de los chatbots de propósito general disponibles en el mercado [1] [2] [3].

En la figura 1.1 se presenta un diagrama de alto nivel de la solución. En primer lugar, los usuarios interactúan con el chatbot a través de una interfaz gráfica desde donde pueden realizar consultas sobre la información deseada. Estas consultas, procesadas mediante técnicas de lenguaje natural, permiten extraer la información más relevante de la fuente de documentos. Luego, un modelo de inteligencia

artificial interpreta las consultas y genera respuestas que proporcionan al usuario la información solicitada de manera precisa y contextualizada.

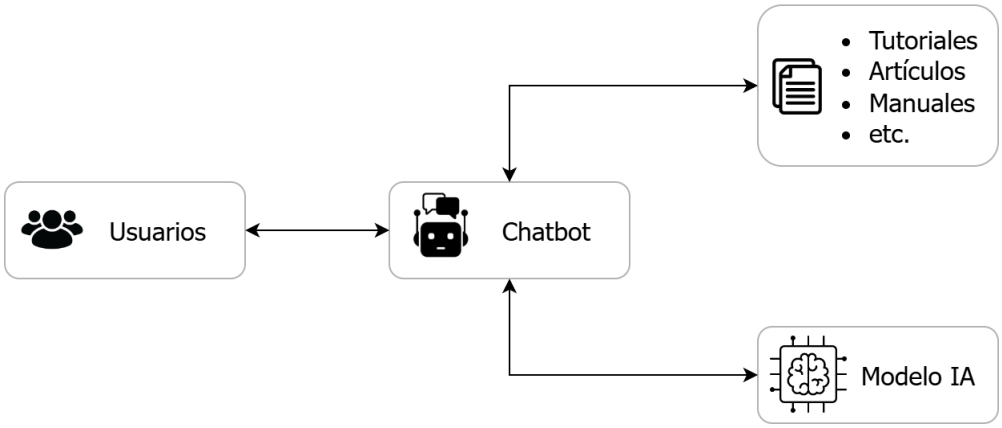

FIGURA 1.1. Diagrama de alto nivel de la solución.

1.2. Estado del arte

El desarrollo de chatbots y sistemas de recuperación de información ha avanzado considerablemente en los últimos años, impulsado por mejoras en el procesamiento de lenguaje natural (NLP, por su sigla en inglés) y el acceso a grandes volúmenes de datos. En este contexto, los chatbots especializados han surgido como soluciones destacadas para el acceso eficiente a información específica en distintos entornos, incluyendo el empresarial. A continuación, se presenta una revisión de las principales tecnologías y enfoques actuales que sustentan el desarrollo del presente trabajo.

Los chatbots modernos han evolucionado desde sistemas de reglas simples hasta modelos sofisticados capaces de mantener conversaciones complejas. Entre los primeros desarrollos, como ELIZA [4] en la década de 1960, se empleaban reglas predefinidas que limitaban la interacción a una cantidad pequeña de respuestas posibles. Sin embargo, el uso de redes neuronales y el aprendizaje profundo en las últimas décadas han revolucionado el campo, al permitir la aparición de sistemas como Siri de Apple, Alexa de Amazon y Google Assistant [5]. Estos asistentes virtuales han popularizado el uso de interfaces de conversación en la vida cotidiana, ya que son capaces de responder a preguntas comunes, realizar tareas administrativas y ofrecer asistencia en tiempo real.

Una tendencia reciente en el desarrollo de chatbots es la aplicación de modelos generativos de lenguaje, como GPT-3 y GPT-4 de OpenAI [6], BERT de Google [7], y LLAMA de Meta [8]. Estos modelos, basados en arquitecturas de *transformers* [9], permiten una comprensión profunda del contexto y del significado en secuencias de palabras. Su capacidad de generar respuestas coherentes y bien estructuradas ha llevado al desarrollo de los tan populares chatbots modernos como ChatGPT [1], Microsoft Copilot [2] o Google Gemini [3], cuyas interfaces se observan en la figura 1.2.

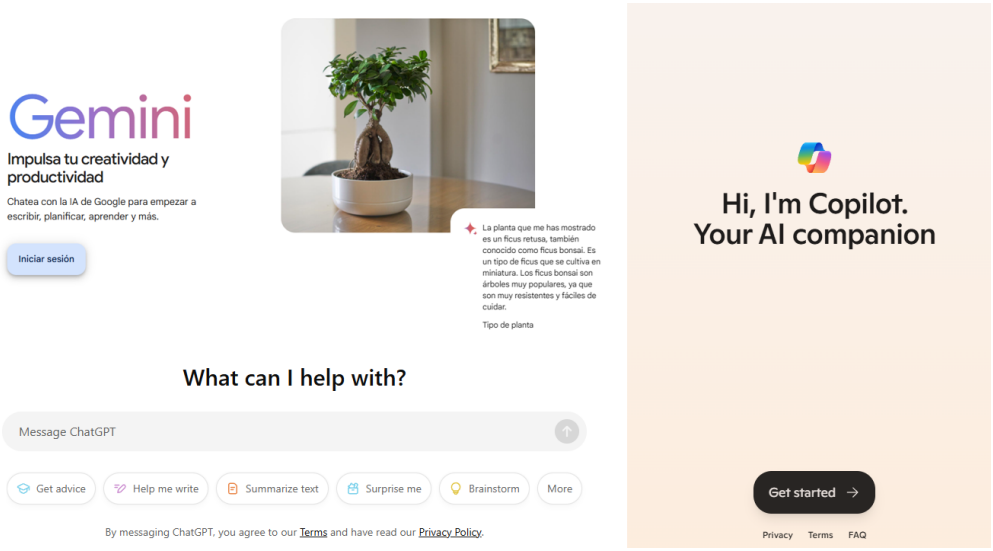

FIGURA 1.2. ChatGPT, Gemini y Copilot, los chatbots más populares actualmente.

Si bien los modelos generativos han alcanzado un alto grado de sofisticación, presentan algunas limitaciones importantes. En primer lugar, su conocimiento es en gran medida de propósito general, dado que han sido entrenados con grandes volúmenes de datos públicos y no específicos, lo que limita su precisión cuando se requiere información particular de una organización. En segundo lugar, estos modelos tienden a "inventar" respuestas cuando no encuentran información relevante, fenómeno conocido como *hallucinations* [10]. En un contexto empresarial, esto puede provocar confusión o incluso proporcionar información errónea.

En la búsqueda de soluciones que combinen la capacidad de los modelos generativos con la precisión de la información propietaria, ha surgido el enfoque de generación aumentada por recuperación (RAG, por su sigla en inglés). Este enfoque combina sistemas de recuperación de información con modelos de generación de texto, lo que permite que las respuestas no solo se basen en la capacidad generativa del modelo, sino también en una búsqueda previa en bases de datos o documentos específicos [11] [12].

El presente trabajo se apoya en el estado del arte de los modelos de lenguaje y la técnica de RAG para crear una solución innovadora.

1.3. Objetivos y alcance

El propósito de este trabajo fue optimizar el proceso de búsqueda de información por parte de los empleados. Se buscó proporcionar una herramienta eficaz que permita acceder rápidamente a los datos relevantes, y que mejore la eficiencia y productividad en el entorno laboral.

Para ello, se realizaron las siguientes tareas:

- Procesamiento de los documentos y posterior almacenamiento en una base de datos.
- Integración con un modelo lingüístico grande (LLM) que pueda entender las consultas de los usuarios y proporcionar respuestas precisas basadas en el contenido de los documentos ingestados.
- Diseño e implementación de una interfaz de usuario intuitiva y fácil de utilizar que permita a los empleados interactuar con el chatbot de manera eficiente.
- Desarrollo de un *pipeline* de despliegue continuo que facilite la ingesta de nuevos documentos y la actualización de la aplicación.
- Evaluación del rendimiento del chatbot mediante pruebas exhaustivas con diferentes tipos de consultas.

Las siguientes actividades no formaron parte del alcance:

- Despliegue del chatbot en un ambiente productivo.
- Entrenamiento continuo del chatbot en base a las consultas realizadas por los usuarios.
- Desarrollo de funcionalidades avanzadas de seguridad, tales como autenticación de usuarios o cifrado de datos.

1.4. Requerimientos

A continuación se describen los principales requerimientos establecidos durante la etapa de planificación:

1. Requerimientos funcionales:
   a) El sistema debe permitir a los usuarios consultar por información a través de una interfaz gráfica.
   b) El sistema debe ser capaz de entender consultas escritas en lenguaje natural.
   c) El sistema debe proporcionar respuestas precisas basadas en el contenido de los documentos procesados.
2. Requerimientos de la interfaz:
   a) La interfaz gráfica debe ser intuitiva y fácil de usar para los usuarios.
   b) Se debe proporcionar retroalimentación instantánea al usuario luego de realizar una consulta.
3. Requerimiento de testing:
   a) Se deben realizar pruebas exhaustivas para garantizar la precisión y la robustez del sistema.

4. Requerimientos de documentación:
    a) Se deben documentar las pruebas realizadas y los resultados obtenidos.
    b) Se debe elaborar un informe de avance del proyecto.
    c) Se debe confeccionar una memoria técnica del proyecto.
5. Requerimientos de cumplimiento normativo:
    a) El sistema debe cumplir con las regulaciones de privacidad de datos vigentes.

Capítulo 2

Introducción específica

Este capítulo presenta las técnicas y herramientas clave utilizadas en el desarrollo de este trabajo. Se analizan los enfoques fundamentales para el procesamiento de lenguaje natural, junto con los modelos, frameworks e infraestructura necesarios para construir un sistema de recuperación de información eficiente y escalable. Este recorrido técnico permite comprender los fundamentos sobre los que se apoya la solución implementada.

2.1. Técnicas de procesamiento de lenguaje natural

El procesamiento de lenguaje natural (NLP) es un área de la inteligencia artificial que les permite a las máquinas comprender e interpretar el lenguaje humano [13]. Esto juega un papel esencial para que un chatbot interprete las consultas de los usuarios y localice la información relevante en los documentos procesados. Las técnicas de NLP aplicadas contribuyen a que el sistema entienda el significado de las solicitudes, independientemente de variaciones lingüísticas o de sintaxis. Entre los métodos de mayor importancia para este trabajo se encuentran los *word embeddings* y los *transformers*.

2.1.1. *Word embeddings*

Los *embeddings* son una poderosa técnica para transformar datos complejos en formas numéricas que son fácilmente procesadas y analizadas por algoritmos de aprendizaje automático [14]. Esta técnica permite representar virtualmente cualquier tipo de dato como vectores, lo que posibilita su manipulación en tareas de procesamiento de lenguaje natural.

Sin embargo, no se trata solo de convertir las palabras en vectores. Es crucial preservar el significado original de los datos para que las tareas realizadas en este espacio transformado mantengan la coherencia semántica. Por ejemplo, al comparar dos frases, no solo se desea analizar las palabras que contienen, sino también evaluar si ambas expresan un significado similar.

Para conservar el significado, es necesario generar vectores donde las relaciones entre ellos sean representativas del contenido. Para ello, se emplea un modelo de *embeddings* pre-entrenado, que produce una representación compacta de los datos mientras mantiene sus características semánticas. El objetivo es capturar el significado o las relaciones semánticas entre los puntos de datos, de modo que los elementos similares se encuentren cercanos en el espacio vectorial y los disímiles estén alejados. Por ejemplo, si se consideran las palabras “rey” y “reina”, un *embedding* podría mapear estas palabras en vectores de modo que la diferencia entre

"rey' y "reina" sea similar a la diferencia entre "hombre" y "mujer", y reflejar así las relaciones semánticas subyacentes.

En este trabajo, se ensayaron diferentes modelos de *embeddings* de OpenAI, Google y Hugging Face, cuya evaluación y selección se detallan en el capítulo 4.

2.1.2. *Transformers*

La arquitectura de *transformers* ha revolucionado el procesamiento de lenguaje natural al introducir un mecanismo de *self-attention*, que permite a un modelo evaluar la relevancia de cada palabra en una secuencia en relación con las demás [9]. Este enfoque supera las limitaciones de modelos secuenciales tradicionales, ya que es capaz de procesar palabras en paralelo y captar dependencias de largo alcance en el texto. En lugar de analizar cada palabra en un orden específico, el mecanismo de *self-attention* permite que el modelo "preste atención" a las palabras más relevantes en el contexto de la frase, al asignar pesos a cada una según su importancia relativa en la oración. Gracias a esta capacidad, los *transformers* pueden capturar relaciones contextuales complejas y matices semánticos entre las palabras, lo que resulta fundamental para tareas como la generación de texto. Esta arquitectura ha hecho posible que los modelos comprendan y generen lenguaje natural de una forma mucho más cercana a la comprensión humana. Esta técnica ha sido fundamental en el desarrollo de los modelos grandes de lenguaje, que se introducen a continuación.

2.2. Modelos grandes de lenguaje

Los modelos grandes de lenguaje (LLM, por su sigla en inglés) son redes neuronales de gran escala, entrenadas para comprender y generar texto en lenguaje natural. Estos modelos, basados en arquitecturas de *transformers*, están compuestos por millones o incluso billones de parámetros, lo que les permite capturar patrones complejos y relaciones contextuales en vastos conjuntos de datos de texto. Los LLM son capaces de realizar múltiples tareas de procesamiento de lenguaje natural, como la generación de texto, la traducción automática y el resumen de documentos. En la tabla 2.1 se presentan algunos de los modelos más relevantes en la actualidad [15].

TABLA 2.1. Modelos LLM disponibles en el mercado.

Modelo	**Creador**	**Año de publicación**	**Cant. de parámetros (aprox.)**
GPT-4o	OpenAI	2024	200 mil millones
Gemini 1.5	Google	2024	1,5 billones
LLaMa 3	Meta	2024	70 mil millones
GPT-4	OpenAI	2023	1,76 billones
LLaMa 2	Meta	2023	70 mil millones
Mistral-7B	Mistral AI	2023	7 mil millones
BLOOM	BigScience	2022	176 mil millones
GPT-3.5	OpenAI	2022	20 mil millones

En este trabajo, el modelo LLM desempeña un papel central ya que es el encargado de entender las consultas de los usuarios y generar respuestas coherentes. Para seleccionar el modelo más adecuado, se ensayaron diferentes variantes, cuyos detalles se describen en el capítulo 4.

2.2.1. Modelos LLM versus modelos tradicionales

A continuación se mencionan las principales diferencias entre los modelos grandes de lenguaje y los modelos tradicionales:

- Los LLM pueden adaptarse a nuevas tareas simplemente con el empleo de ejemplos en el *prompt*, sin necesidad de modificar sus parámetros o re-entrenar. Esto contrasta con los modelos tradicionales, que requieren grandes conjuntos de datos etiquetados y re-entrenarse cada vez que se incorporan nuevas tareas.
- Comprenden instrucciones en lenguaje natural, lo que les permite seguir indicaciones complejas e incluso inferir reglas implícitas a partir de ejemplos. Los modelos tradicionales, en cambio, suelen necesitar formatos de entrada específicos y siguen reglas programadas explícitamente, sin capacidad para inferir patrones nuevos en tiempo real.
- Mantienen la coherencia en conversaciones largas y son capaces de recordar información dada en partes previas del diálogo. Los modelos tradicionales procesan cada entrada de manera independiente y no pueden conservar el estado conversacional.
- Permiten una interacción bidireccional, donde pueden solicitar aclaraciones y ajustar sus respuestas en función del *feedback* recibido. Los modelos tradicionales, en cambio, ofrecen una interacción unidireccional y no manejan *feedback* en tiempo real.
- Un solo LLM es capaz de realizar múltiples tareas sin configuración adicional y de adaptar su salida según el contexto. Por el contrario, los modelos tradicionales están diseñados para una tarea específica, por lo que se requiere un modelo separado para cada tarea.

2.3. Generación aumentada por recuperación

Una limitación importante que presentan los LLM es que, debido a su naturaleza generalista, pueden carecer de precisión cuando se aplican a dominios específicos, ya que su conocimiento está limitado a la información con la que fueron entrenados. Si bien esta información es sumamente vasta, no alcanza a cubrir la infinidad de temas posibles, y no contempla contenido no disponible en fuentes públicas.

Para abordar esta limitación, los LLM pueden adaptarse mediante la técnica conocida como recuperación aumentada por generación (RAG) [11], que integra fuentes de datos especializadas que mejoran su precisión en contextos concretos, como el entorno empresarial. De esta manera, los LLM pueden aprovechar su capacidad de comprensión profunda del lenguaje, pero también acceder a información precisa y actualizada de documentos específicos.

Básicamente, esta técnica consta de dos fases principales: primero, se realiza una búsqueda entre los documentos provistos y se seleccionan fragmentos que sean relevantes para la consulta del usuario. Luego, un modelo LLM utiliza esa información para construir una respuesta contextualizada. En la figura 2.1 se ilustra un diagrama básico de su funcionamiento.

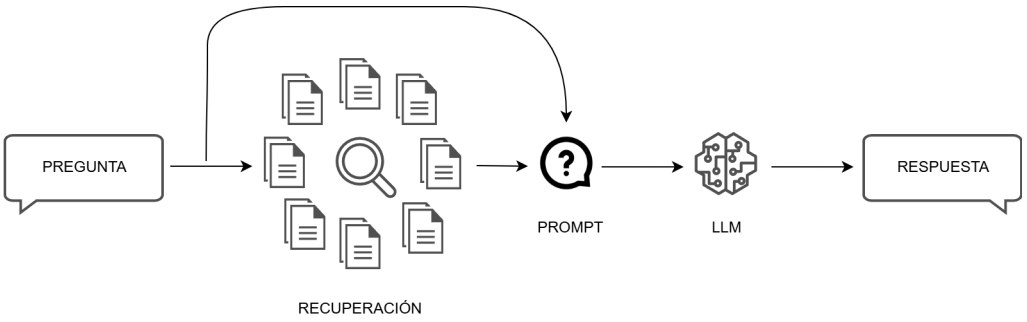

FIGURA 2.1. Diagrama de un sistema RAG.

La combinación de recuperación y generación permite un flujo de información enriquecido, al lograr un equilibrio entre las capacidades generales del modelo y la especificidad requerida en el contexto empresarial.

2.4. Bases de datos vectoriales

Una base de datos de vectorial [16] es un sistema diseñado para almacenar y gestionar *embeddings*. Esta cumple un papel fundamental en la etapa de recuperación de información.

A diferencia de las bases de datos tradicionales, en las que las consultas suelen coincidir exactamente con los valores almacenados, las bases de datos vectoriales aplican métricas de similitud para identificar el vector más cercano a la consulta. El proceso de recuperación se realiza mediante el cálculo de la distancia entre el vector de la consulta y los vectores almacenados en la base de datos. Los fragmentos con menor distancia al vector de la consulta son seleccionados como los más relevantes y luego utilizados como insumo en la fase de generación de la respuesta.

Para seleccionar la base de datos vectorial más adecuada para este trabajo, se exploraron varias opciones disponibles en el mercado y se evaluaron aspectos como el rendimiento y la facilidad de integración con el chatbot. Los resultados de estos ensayos se describen en el capítulo 4.

2.5. Frameworks utilizados

En el desarrollo de este trabajo se utilizaron una serie frameworks que facilitaron la implementación de cada parte del sistema. A continuación, se detallan sus principales características y su rol específico en el producto final.

- LangChain: este framework es fundamental para la construcción del sistema RAG, ya que permite una gestión eficiente de flujos de trabajo con modelos de lenguaje. LangChain facilita la integración de modelos LLM con

fuentes de datos externas, y además ofrece herramientas para diseñar flujos complejos que manejan las consultas del usuario, el proceso de recuperación y la generación de respuestas [17].

- FastAPI: se utiliza para construir la API que conecta los componentes del sistema y gestiona las solicitudes del usuario. Este framework, reconocido por su alto rendimiento y facilidad de uso, permitió desarrollar una API eficiente y escalable que procesa las consultas en tiempo real. FastAPI facilita el manejo de solicitudes asíncronas y permite definir de manera clara los puntos de conexión de la API, asegurando que las interacciones entre el frontend y el backend se realicen de forma fluida y rápida [18].
- NextUI: es un framework de diseño de interfaces de usuario basado en React que permite construir interfaces modernas y responsivas. NextUI facilitó la creación de una interfaz intuitiva y fácil de usar, que permite a los usuarios interactuar con el chatbot de manera fluida. Este framework ofrece una variedad de componentes pre-construidos y altamente personalizables, lo que ayudó a reducir el tiempo de desarrollo de la interfaz y asegurar una experiencia de usuario satisfactoria [19].

2.6. Servicios en la nube utilizados

En este trabajo se empleó Microsoft Azure [20] como la plataforma principal de infraestructura en la nube, seleccionada por su robustez y variedad de servicios orientados a aplicaciones de inteligencia artificial. Azure ofrece soluciones escalables y seguras para gestionar datos y modelos, lo que facilita la integración de distintos recursos en un entorno controlado y eficiente.

Los recursos específicos de Azure utilizados incluyen:

- Azure OpenAI: este recurso se utilizó para deployar el modelo de lenguaje LLM y el modelo de *embeddings*, lo que provee la capacidad de procesamiento de lenguaje natural avanzada requerida por el sistema [21].
- Azure AI Search: utilizado como base de datos vectorial, este servicio permite almacenar y recuperar *embeddings* mediante búsquedas de similitud [22].
- Azure App Service: empleado para hospedar el backend del chatbot, este servicio garantiza un entorno seguro y escalable para la ejecución de la API y la gestión de consultas de los usuarios [23].
- Azure Static Web Apps: utilizado para hospedar el frontend del chatbot, proporciona un despliegue rápido y eficiente de la interfaz de usuario y puede ser accedido por los empleados desde cualquier ubicación [24].

Además, se implementó GitHub Actions [25] como *pipeline* de integración y despliegue continuo, no solo para publicar actualizaciones del backend y el frontend, sino también para automatizar la actualización del contenido de la base de datos.

Capítulo 3

Diseño e implementación

Este capítulo describe el diseño y la implementación de cada uno de los componentes que conforman el sistema. Se explican las decisiones de diseño, los flujos de trabajo y los aspectos técnicos involucrados en la construcción de cada módulo principal.

3.1. Arquitectura del sistema

El sistema está diseñado para permitir la implementación y operación de un chatbot mediante un enfoque de recuperación aumentada por generación. Se implementó una arquitectura modular, basada en servicios en la nube, que facilita la escalabilidad y el mantenimiento del chatbot, y permite una actualización ágil de los componentes y una experiencia optimizada para los usuarios finales. En la figura 3.1 se ilustra el diagrama de arquitectura, donde se observan la totalidad de los componentes que conforman el sistema.

En primer lugar, los usuarios interactúan con el chatbot a través de una interfaz gráfica (desarrollada con NextUI y deployada en Azure Static Web App), donde consultan por la información deseada. Estas consultas son luego transferidas al servidor (Azure App Service) a través de una API desarrollada con la librería FastAPI. Una vez allí, se realiza un proceso de búsqueda por similitud que toma la solicitud del usuario y la compara con la información disponible en la base de datos (Azure AI Search), con el objetivo de identificar aquellos fragmentos más relevantes. Previamente, un administrador debe haber cargado aquellos documentos que conforman la base de conocimiento del chatbot en un repositorio de GitHub. Luego, se ejecuta una automatización que los procesa y los envía a la base de datos.

Finalmente, la consulta del usuario y los fragmentos relevantes se envían al modelo LLM (deployado en el servicio Azure OpenAI), que genera una respuesta contextualizada, que es devuelta a la interfaz gráfica para ser presentada al usuario.

Además, se incluye una pequeña base de datos adicional (Azure Table Storage) que se utiliza para guardar el feedback de los usuarios, para así obtener métricas del desempeño del chatbot.

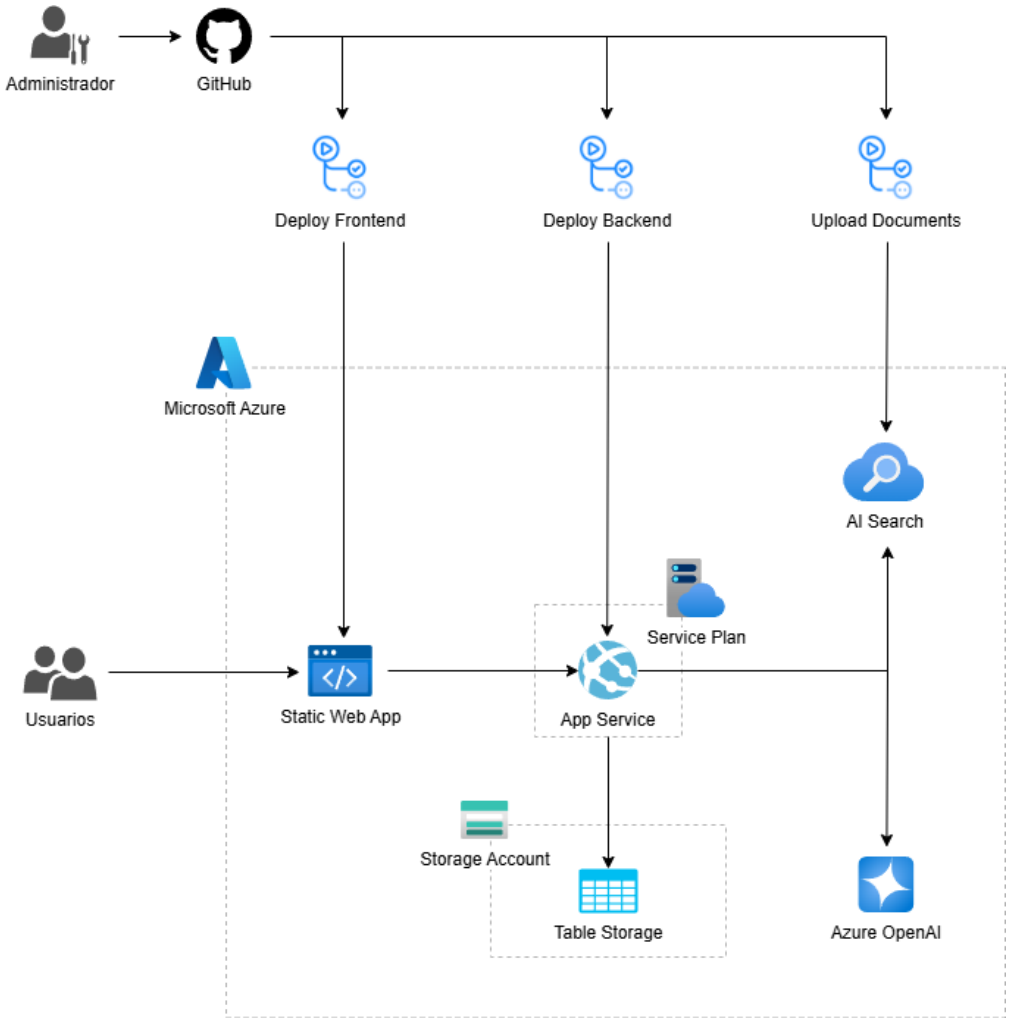

FIGURA 3.1. Diagrama de arquitectura del sistema.

3.2. Configuración de la infraestructura en la nube

Para garantizar el despliegue, disponibilidad y escalabilidad del sistema, se configuró una infraestructura en la nube basada en Microsoft Azure. A continuación, se describen los pasos principales para la configuración de cada recurso empleado en el proyecto.

3.2.1. OpenAI Service

El servicio de Azure OpenAI proporciona acceso a los potentes modelos de lenguaje de OpenAI, incluidos los más recientes. Estos modelos pueden adaptarse fácilmente a tareas específicas, como en este caso es la generación de contenido.

En la tabla 3.1 se presenta un resumen de las configuraciones realizadas. Se seleccionó *East US* como región, junto con el modelo de precios *Standard S0*, adecuado para balancear el costo y el rendimiento del sistema.

Como medida de seguridad, se configuraron reglas de red para restringir el acceso únicamente al rango de direcciones IP del backend alojado en Azure App Service. Esto asegura que solo las solicitudes provenientes de la aplicación puedan acceder al modelo de lenguaje, lo que protege el servicio de accesos no autorizados.

Una vez deployado el recurso, fue necesario seleccionar y deployar los modelos específicos requeridos por el chatbot. En este caso, se optó por los siguientes modelos:

- *GPT-4o* como modelo de lenguaje para generación de respuestas. Este modelo está diseñado para brindar velocidad y eficiencia, iguala la inteligencia de su antecesor *GPT-4 Turbo*, y es notablemente más eficiente al entregar texto al doble de velocidad y a la mitad del costo. Además, exhibe el rendimiento más alto en idiomas distintos del inglés en comparación con los modelos de OpenAI anteriores.
- *Ada-002* para el cálculo de *embeddings* de texto. Este modelo supera a todos los modelos de *embeddings* anteriores en tareas de búsqueda de texto y similitud de oraciones.

Adicionalmente, es fundamental obtener el *endpoint* y la *key* del servicio, valores que se utilizan luego para la comunicación programática con Azure OpenAI. Estos datos se almacenan en el backend como variables de entorno para que el sistema pueda acceder al servicio de manera segura.

TABLA 3.1. Configuración de Azure OpenAI.

Configuración	**Detalles**
Región	East US
Nivel de precios	Standard S0
Reglas de firewall	Rango IP de App Service
Modelos deployados	- gpt-4o - text-embeddings-ada-002
Credenciales	Endpoint y key

3.2.2. AI Search

Azure AI Search es un servicio de búsqueda en la nube con capacidades de inteligencia artificial integradas que enriquecen todo tipo de información con el fin de identificar y explorar fácilmente contenido relevante a escala.

La tabla 3.2 presenta un resumen de las configuraciones realizadas. Se seleccionó la región de *East US* y un nivel de precios *Standard*, que ofrece hasta 50 índices para el almacenamiento y la búsqueda de documentos. En cuanto a la escala, el servicio se configuró para utilizar una sola unidad de búsqueda, lo cual es adecuado en principio para el volumen de consultas esperado y los requisitos de rendimiento.

Al igual que con el servicio de Azure OpenAI, se configuraron reglas de red restrictivas para mejorar la seguridad del servicio, al permitir únicamente el acceso desde el rango de direcciones IP asociado al backend.

Aquí también es fundamental obtener la *key* y el *endpoint* del servicio para integrarlos en las variables de entorno del backend.

Es importante destacar que no es necesario crear un índice manualmente en esta etapa, ya que este será generado automáticamente como parte del pipeline de despliegue.

TABLA 3.2. Configuración de Azure AI Search.

Configuración	**Detalles**
Región	East US
Nivel de precios	Standard
Reglas de firewall	Rango IP de App Service
Credenciales	Endpoint y key

3.2.3. App Service

Este servicio ofrece una plataforma completamente administrada donde es posible alojar una aplicación avanzada en la nube sin necesidad de manejar la infraestructura asociada. En este caso se utilizó para deployar el backend del chatbot.

La tabla 3.4 resume la configuración principal del servicio. En primer lugar, se seleccionó un App Service Plan de categoría *Basic B3*, que proporciona un poder de procesamiento de 4 vCPU, 7 GB de memoria RAM, 10 GB de almacenamiento remoto, y permite escalar hasta 3 instancias en caso de ser necesario. Como entorno de ejecución, se configuró Python 3.10, mientras que como región se optó una vez más por *East US*. Con el fin de optimizar los costos, se deshabilita la opción de redundancia zonal.

Para que la aplicación funcione correctamente, es esencial configurar un conjunto variables de entorno, que se listan en la tabla 3.3. Estas variables incluyen las claves y los puntos de conexión de los servicios de Azure que interactúan con el backend.

TABLA 3.3. Variables de entorno en el backend.

Variable de entorno	**Descripción**
openai_api_key	Clave de acceso del servicio de Azure OpenAI
openai_endpoint	Endpoint del servicio de Azure OpenAI
search_key	Clave de acceso del servicio de Azure AI Search
search_endpoint	Endpoint del servicio de Azure AI Search
storage_connection_string	Connection string de la cuenta de almacenamiento
db_index	Índice de la base de datos a utilizar

Para asegurar un monitoreo efectivo de la aplicación, se habilitó la integración con Application Insights, lo que permite un seguimiento detallado de las métricas de rendimiento y de uso. Adicionalmente, se configuró la opción de *application logging*, de modo que los logs de la aplicación fueran visualizados directamente desde la terminal del App Service, lo que facilita la depuración y la supervisión continua.

TABLA 3.4. Configuración de Azure App Service.

Configuración	Detalles
App Service Plan	Basic B3
Runtime	Python 3.10
Región	East US
Zone redundancy	Deshabilitada
Application Insights	Habilitado
Application logging	Filesystem

3.2.4. Static Web App

Al utilizar este servicio, el contenido estático como HTML, CSS, JavaScript e imágenes, se distribuye globalmente desde diversos puntos alrededor del mundo a diferencia de un servidor web tradicional, por lo que los archivos se encuentran físicamente más cerca de los usuarios finales, sin importar su ubicación.

La tabla 3.5 resume los aspectos clave de la Static Web App. Este recurso es sencillo de configurar, ya que solo requiere seleccionar un modelo de precios. En este caso, se optó por la versión *Standard*. A diferencia de otros servicios en Azure, las Static Web Apps se despliegan globalmente, por lo que no es necesario especificar una región. Además, ya que todas las conexiones con otros servicios son manejadas exclusivamente por el backend, la única variable de entorno requerida es la URL de la API, lo que simplifica la configuración.

Es importante obtener el *deployment token* asociado al recurso, que será necesario más adelante al configurar el pipeline de despliegue automático para permitir la autenticación y el despliegue seguro desde GitHub.

TABLA 3.5. Configuración de Azure App Service.

Configuración	Detalles
Nivel de precios	Standard
Región	Global
Variables de entorno	URL del backend
Credenciales	Deployment token

3.2.5. Table Storage

Azure Table Storage es un servicio que almacena datos estructurados no relacionales (también conocidos como datos NoSQL) en la nube, que almacena claves/atributos con un diseño sin esquemas. En el contexto de este trabajo se utilizó para almacenar el feedback de los usuarios acerca de las respuestas provistas por el chatbot.

La tabla 3.6 resume la configuración principal de este recurso. Primeramente, fue necesario deployar una Storage Account en la región *East US*, para así mantener la consistencia con los demás recursos del sistema. Se seleccionó una performance de tipo *Standard*, adecuada para las necesidades de la aplicación.

En cuanto a la redundancia de los datos, se eligió la oferta de redundancia local, una alternativa rentable que asegura que los datos estén replicados dentro de un

único centro de datos en la misma región, que reduce costos sin comprometer la disponibilidad básica.

Una vez que la cuenta de almacenamiento fue deployada, se procedió a crear la tabla que almacenará los resultados recopilados a través del sistema. Es importante obtener la *connection string* asociada al recurso, ya que será necesaria para conectar la aplicación con la tabla y permitir el acceso programático.

TABLA 3.6. Configuración de Azure Table Storage.

Configuración	**Detalles**
Región	East US
Performance	Standard
Redundancia	Locally-redundant storage
Credenciales	Connection string
Nombre de la tabla	FeedbackTable

3.3. Procesamiento de los documentos

La finalidad del procesamiento de documentos es transformar archivos de texto y PDF en un formato que facilite su búsqueda y análisis mediante herramientas de inteligencia artificial. Este flujo de trabajo incluye el uso de modelos de *embeddings* para convertir el texto en representaciones vectoriales, y su posterior inserción en una base de datos vectorial. Para ello, se desarrolló un script en Python que asegura un manejo eficiente de grandes volúmenes de datos, al implementar mecanismos para distribuir las solicitudes a la API de Azure Search y evitar superar sus límites de tasa de uso, además de optimizar el almacenamiento en el índice de búsqueda.

3.3.1. Importación de bibliotecas y configuración inicial

El script comienza con la importación de las bibliotecas necesarias para realizar el procesamiento de documentos. Se utilizan paquetes específicos que integran funciones avanzadas para trabajar con *embeddings*, carga de documentos, y búsqueda por similitud. El módulo *nltk* es esencial para el procesamiento de texto en lenguaje natural, al proporcionar herramientas de tokenización y etiquetado de texto.

```
1  import os
2  import requests
3  from langchain_openai import AzureOpenAIEmbeddings
4  from langchain_google_genai import GoogleGenerativeAIEmbeddings
5  from langchain_community.vectorstores.azuresearch import AzureSearch
6  from langchain_text_splitters import RecursiveCharacterTextSplitter
7  from langchain_community.document_loaders import
     UnstructuredMarkdownLoader, PyPDFLoader
8  import nltk
9  import time
10
11 nltk.download('punkt_tab')
12 nltk.download('averaged_perceptron_tagger_eng')
```

CÓDIGO 3.1. Importación de bibliotecas y configuración inicial.

3.3.2. Configuración del índice de búsqueda

Antes de insertar documentos en la base de datos, es necesario realizar una limpieza para eliminar cualquier dato previo. La función *delete_index* se encarga de esta tarea, al enviar una solicitud de tipo *DELETE* a la API de Azure Search para borrar todos los documentos de un índice específico.

```
def delete_index(azure_search_endpoint, azure_search_key, index_name):
    url = f"{azure_search_endpoint}/indexes('{index_name}')?api-version
    =2023-11-01"
    headers = {
        "Content-Type": "application/json",
        "api-key": f"{azure_search_key}"
    }
    response = requests.delete(url, headers=headers)
    if response.status_code == 204:
        print("All documents deleted successfully.")
    else:
        print(f"Failed to delete documents. Status code: {response.
    status_code}, Response: {response.text}")
```

CÓDIGO 3.2. Configuración del índice de búsqueda.

Este paso es esencial para asegurar que los datos insertados sean consistentes con los documentos procesados más recientemente, al evitar duplicación de información.

3.3.3. Configuración del modelo de *embeddings*

La selección del modelo de *embeddings* depende de una variable de entorno. El script soporta tanto modelos de Azure OpenAI como de Google. Según el modelo especificado, se inicializa una instancia de *AzureOpenAIEmbeddings* o *GoogleGenerativeAIEmbeddings*.

```
if os.getenv("EMBEDDINGS_MODEL") == "openai":
    embeddings = AzureOpenAIEmbeddings(model="ada-002",
    openai_api_version="2024-06-01")
elif os.getenv("EMBEDDINGS_MODEL") == "google":
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding
    -001")
else:
    embeddings = AzureOpenAIEmbeddings(model="ada-002",
    openai_api_version="2024-06-01")
```

CÓDIGO 3.3. Configuración del modelo de *embeddings*.

3.3.4. Conexión con la base de datos

Una vez configurado el modelo de *embeddings*, el siguiente paso es instanciar una conexión con el índice de Azure Search. Esto permite la interacción directa con el índice y la inserción de documentos procesados.

```
azure_search = AzureSearch(
    azure_search_endpoint=os.getenv("AZURE_SEARCH_URI"),
    azure_search_key=os.getenv("AZURE_SEARCH_KEY"),
    index_name=index_name,
    embedding_function=embeddings.embed_query
)
```

CÓDIGO 3.4. Conexión con la base de datos.

Este objeto se utiliza posteriormente en el flujo para agregar documentos en el índice de búsqueda, una vez que hayan sido procesados y transformados en *embeddings*.

3.3.5. Divisor de texto

El script emplea un divisor de texto para procesar documentos extensos de manera eficiente. Este divisor separa el contenido en fragmentos manejables de hasta 512 caracteres. Además, aplica un solapamiento de 64 caracteres entre fragmentos consecutivos. Este solapamiento garantiza la continuidad del contexto en los fragmentos generados.

El script busca archivos en la carpeta *knowledge-base* y aplica distintos *loaders* según el tipo de archivo (PDF o Markdown). Luego, cada archivo se divide en fragmentos por medio del divisor de texto configurado previamente.

```
splitter = RecursiveCharacterTextSplitter(chunk_size=512,
  chunk_overlap=64)

for root, dirs, files in os.walk('knowledge-base'):
  for file in files:
      file_path = os.path.join(root, file)
      if file.endswith('.pdf'):
          data_loader = PyPDFLoader(file_path)
      elif file.endswith('.md'):
          data_loader = UnstructuredMarkdownLoader(file_path)
      file_chunks = data_loader.load_and_split(text_splitter=splitter)
      print(f"{file_path} splitted into {len(file_chunks)} chunks")
```

CÓDIGO 3.5. Divisor de texto.

3.3.6. Inserción en lotes

Para evitar alcanzar el límite de uso de la API, los fragmentos se insertan en el índice en lotes pequeños, con una demora entre cada de por medio. La función *batch_insert_chunks* controla esta inserción, al establecer un tamaño de lote y un tiempo de espera.

```
def batch_insert_chunks(chunks,batch_size=5,delay_between_batches=3):
  batch_count=0
  inserted_ids = []
  for i in range(0, len(chunks), batch_size):
      batch = chunks[i:i+batch_size]
      batch_count += 1
      inserted_ids_batch = azure_search.add_documents(batch)
      inserted_ids.extend(inserted_ids_batch)
      print(f"Inserted batch #{batch_count} ({len(inserted_ids_batch)}
   documents)")
      time.sleep(delay_between_batches)
  return inserted_ids
```

CÓDIGO 3.6. Inserción en lotes.

La inserción en lotes es un aspecto clave del diseño, ya que permite manejar eficientemente grandes volúmenes de datos sin interrumpir el flujo debido a límites de solicitud.

3.4. Lógica de generación aumentada por recuperación

El sistema RAG se compone de dos etapas principales:

- Fase de recuperación: se realiza una búsqueda semántica en la base de datos vectorial para identificar los fragmentos más relevantes a partir de una consulta.
- Fase de generación: los fragmentos recuperados se combinan con la consulta del usuario y se utilizan como contexto para un modelo de lenguaje, que genera una respuesta contextualizada.

Este enfoque aprovecha lo mejor de ambos mundos: la precisión en la recuperación de datos específicos y la capacidad generativa avanzada de los LLM.

3.4.1. Fase de recuperación

Esta fase se encarga de encontrar los fragmentos más relevantes de la base de datos vectorial. Para ello se desarrolló la clase *Retriever*, cuyo código se muestra a continuación.

```
class Retriever():

  def __init__(self):
    if os.getenv("EMBEDDINGS_MODEL") == "openai":
      self.embeddings = AzureOpenAIEmbeddings(model="ada-002",
openai_api_version="2024-06-01")
    elif os.getenv("EMBEDDINGS_MODEL") == "google":
      self.embeddings = GoogleGenerativeAIEmbeddings(model="models/
embedding-001")
    else:
      self.embeddings = AzureOpenAIEmbeddings(model="ada-002",
openai_api_version="2024-06-01")

    self.vstore = AzureSearch(
      azure_search_endpoint=os.getenv("AZURE_SEARCH_URI"),
      azure_search_key=os.getenv("AZURE_SEARCH_KEY"),
      index_name=os.getenv("DB_INDEX"),
      embedding_function=self.embeddings.embed_query
    )

  def invoke(self, query):
    docs = self.vstore.similarity_search(query['input'], k=3)
    print(docs)
    return self.format_docs(docs)

  def format_docs(self, docs):
    return "\n\n".join(doc.page_content for doc in docs)
```

CÓDIGO 3.7. Clase *Retriever*.

En el constructor se inicializa el modelo de *embeddings* y se establece la conexión con la base de datos vectorial en Azure Search.

El método *invoke* realiza una búsqueda semántica en la base de datos y devuelve los documentos más relevantes. El método utiliza la función *similarity_search* para recuperar los tres fragmentos más relevantes (k=3). Posteriormente, los fragmentos se formatean a través del método *format_docs* en un texto único que servirá como contexto para la fase de generación. Esta estructura garantiza que el modelo de generación reciba un contexto consolidado y relevante.

3.4.2. Fase de generación

La fase de generación utiliza los fragmentos recuperados para producir respuestas contextuales a las consultas del usuario.

Para ello se desarrolló la clase *Generator*, cuyo código se muestra a continuación.

```
class Generator():

  def __init__(self, retriever):
    self.llm = AzureChatOpenAI(
      deployment_name="gpt-4o",
      api_version="2023-06-01-preview"
    )

    self.system_prompt = (
      "You are an AI assistant for question-answering tasks."
      "You are able to answer questions related to Fabian's final project for his master degree in AI."
      "If someone asks, 'What can I ask you about?' or other similar questions, respond with the above topics."
      "If you're unsure, use the following pieces of retrieved context to answer the question."
      "If you don't know the answer, say that you don't know."
      "If a question does not relate to Fabian's project, respond with: 'This question falls outside of my knowledge base'."
      "Use three sentences maximum and keep the answer concise."
      "\n\n"
      "{context}"
    )

    self.prompt = ChatPromptTemplate.from_messages(
      [
        ("system", self.system_prompt),
        ("human", "{question}"),
      ]
    )

    self.parser = StrOutputParser()

    self.rag_chain = (
      {"context": retriever.invoke, "question": RunnablePassthrough()}
      | self.prompt
      | self.llm
      | self.parser
    )

  def invoke(self, user_question):
    answer = self.rag_chain.invoke({"input": user_question})
    return answer
```

CÓDIGO 3.8. Clase *Generator*.

En el constructor se inicializan los componentes clave: un modelo de lenguaje en Azure OpenAI, el *prompt* del sistema y una cadena que gestiona el flujo desde la recuperación hasta la respuesta.

El *prompt* establece las instrucciones que guían al modelo en la generación de respuestas, al definir tanto el tono como las reglas que debe seguir. Este está diseñado para:

1. Concientizar al chatbot sobre su dominio de conocimiento.

2. Incluir solo información del contexto recuperado.
3. Evitar responder a consultas fuera del dominio del chatbot.
4. Limitar las respuestas a tres oraciones.

El método *invoke* utiliza la cadena para procesar la consulta del usuario y generar la respuesta final.

3.5. Implementación de la API

La API desarrollada con FastAPI es el puente entre la interfaz de usuario y las funcionalidades del sistema. La tabla 3.7 detalla los *endpoints* implementados.

TABLA 3.7. Diseño de la API.

Endpoint	Método	Ruta
Generación de respuesta	POST	/api/ask
Registro de feedback	POST	/api/feedback
Consulta de feedback	GET	/api/feedback
Prueba	GET	/api/ping

3.5.1. *Endpoint* de generación de respuestas

Este *endpoint* recibe la consulta del usuario e invoca el proceso RAG para generar una respuesta en lenguaje natural.

```
@app.post("/api/ask")
def generate_answer(body: Prompt):
  try:
    answer = generator.invoke(body.prompt)
    return {"question": body.prompt, "answer": answer}
  except Exception as e:
    raise HTTPException(status_code=500, detail=f"Error: {e}")
```

CÓDIGO 3.9. *Endpoint* de generación de respuestas.

3.5.2. *Endpoint* de registro de opiniones

Este *endpoint* permite a los usuarios enviar su opinión sobre una interacción, es decir, si están o no satisfechos con la respuesta entregada por el chatbot. Los datos se almacenan en Azure Table Storage. El propósito es capturar la percepción del usuario sobre las respuestas generadas, con el objetivo de evaluar y mejorar el sistema.

```
@app.post("/api/feedback")
async def store_feedback(feedback: Feedback):
  entity = TableEntity()
  entity["PartitionKey"] = "likes" if feedback.like else "hates"
  entity["RowKey"] = str(uuid.uuid4())
  entity["Question"] = feedback.question
  entity["Answer"] = feedback.answer

  try:
    table_client.create_entity(entity=entity)
    return {"message": "Feedback stored successfully."}
```

```
    except Exception as e:
      raise HTTPException(status_code=500, detail=f"Error: {e}")

```

CÓDIGO 3.10. *Endpoint* de registro de opiniones.

3.5.3. *Endpoint* de consulta de opiniones

Este *endpoint* retorna estadísticas sobre el feedback registrado por los usuarios, al indicar la cantidad de interacciones positivas y negativas. El propósito es ofrecer estadísticas que permitan monitorear el rendimiento del sistema y la satisfacción del usuario.

```
@app.get("/api/feedback")
async def get_feedback_count():
  try:
    likes_count = len(list(table_client.query_entities(query_filter="
  PartitionKey eq 'likes'")))
    hates_count = len(list(table_client.query_entities(query_filter="
  PartitionKey eq 'hates'")))

    return {"likes": likes_count, "hates": hates_count}
  except Exception as e:
    raise HTTPException(status_code=500, detail=f"Error: {e}")
```

CÓDIGO 3.11. *Endpoint* de consulta de opiniones.

3.5.4. *Endpoint* de prueba

Este *endpoint* verifica que la API está operativa y devuelve una respuesta sencilla. Su propósito es validar la conectividad con la API y confirmar el correcto funcionamiento del sistema.

```
@app.get("/api/ping")
def ping():
  return "pong"
```

CÓDIGO 3.12. *Endpoint* de prueba.

3.6. Implementación de la interfaz de usuario

3.7. Pipelines de despliegue automático

Capítulo 4

Ensayos y resultados

Todos los capítulos deben comenzar con un breve párrafo introductorio que indique cuál es el contenido que se encontrará al leerlo. La redacción sobre el contenido de la memoria debe hacerse en presente y todo lo referido al proyecto en pasado, siempre de modo impersonal.

4.1. Ensayo de modelos

4.2. Ensayo de embeddings

4.3. Ensayo de bases de datos

4.4. Casos de uso

4.5. Validación de requerimientos

Capítulo 5

Conclusiones

Todos los capítulos deben comenzar con un breve párrafo introductorio que indique cuál es el contenido que se encontrará al leerlo. La redacción sobre el contenido de la memoria debe hacerse en presente y todo lo referido al proyecto en pasado, siempre de modo impersonal.

5.1. Resultados

5.2. Trabajo futuro

Bibliografía

[1] OpenAI. *ChatGPT*. URL: https://chatgpt.com/.
[2] Microsoft. *Copilot*. URL: https://copilot.microsoft.com/.
[3] Google. *Gemini*. URL: https://gemini.google.com/.
[4] Joseph Weizenbaum. *ELIZA — a computer program for the study of natural language communication between man and machine*. Ene. de 1966. DOI: 10.1145/365153.365168.
[5] Matthew B. Hoy. *Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants*. Ene. de 2018. DOI: 10.1080/02763869.2018.1404391.
[6] Alec Radford y col. *Improving Language Understanding by Generative Pre-Training*. Jun. de 2018. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.
[7] Jacob Devlin y col. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Oct. de 2018. DOI: 10.48550/arXiv.1810.04805.
[8] Hugo Touvron y col. *LLaMA: Open and Efficient Foundation Language Models*. Feb. de 2023. DOI: 10.48550/arXiv.2302.13971.
[9] Ashish Vaswani y col. *Attention Is All You Need*. Jun. de 2017. DOI: 10.48550/arXiv.1706.03762.
[10] Rahul Awati. *What are AI hallucinations and why are they a problem?* URL: https://www.techtarget.com/whatis/definition/AI-hallucination.
[11] Patrick Lewis y col. *Retrieval-augmented generation for knowledge-intensive NLP tasks*. Dic. de 2020. DOI: 10.48550/arXiv.2005.11401.
[12] Kurt Shuster y col. *Retrieval Augmentation Reduces Hallucination in Conversation*. Abr. de 2021. DOI: 10.48550/arXiv.2104.07567.
[13] Nitin Indurkhya y Fred J. Damerau. *Handbook of Natural Language Processing*. Chapman y Hall, 2010.
[14] Tomas Mikolov y col. *Efficient Estimation of Word Representations in Vector Space*. Ene. de 2013. DOI: 10.48550/arXiv.1301.3781.
[15] LifeArchitect.ai. *Models Table*. URL: https://lifearchitect.ai/models-table/.
[16] Roie Schwaber-Cohen. *What is a Vector Database and How Does it Work?* URL: https://www.pinecone.io/learn/vector-database/.
[17] LangChain. URL: https://python.langchain.com/docs/introduction/.
[18] FastAPI. URL: https://fastapi.tiangolo.com/.
[19] NextUI. URL: https://nextui.org/docs/guide/introduction/.
[20] Microsoft. *Microsoft Azure*. URL: https://azure.microsoft.com/.
[21] Microsoft. *Azure OpenAI Service*. URL: https://azure.microsoft.com/en-us/products/ai-services/openai-service.
[22] Microsoft. *Azure AI Search*. URL: https://azure.microsoft.com/en-us/products/ai-services/ai-search.
[23] Microsoft. *Azure App Service*. URL: https://azure.microsoft.com/en-us/products/app-service.
[24] Microsoft. *Azure Static Web Apps*. URL: https://azure.microsoft.com/en-us/products/app-service/static.
[25] GitHub. *GitHub Actions*. URL: https://github.com/features/actions.