

***Facultad
de
Ciencias***

**ASISTENTE CONVERSACIONAL PARA LA
CONSTRUCCIÓN DE WORKFLOWS DE IA
EN LA HERRAMIENTA LAREDO**
(Conversational Assistant for Building AI
Workflows in the LAREDO Tool)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Gabriel Gómez García

Directora: Marta Elena Zorrilla Pantaleón
Codirector: Ricardo Dinten Herrero

Junio - 2025

Resumen

Los avances recientes en procesamiento de lenguaje natural (NLP) han permitido el desarrollo de asistentes conversacionales cada vez más eficaces para comprender preguntas, mantener diálogos y generar respuestas útiles. Esto ha transformado la manera en la que los usuarios acceden y consultan información, al ofrecer una interacción más humana y accesible. Sin embargo, estos modelos presentan limitaciones a la hora de adaptarse a dominios privados o necesidades específicas, lo que ha motivado la aparición de enfoques como la generación aumentada por recuperación (RAG), capaces de combinar los asistentes conversacionales con mecanismos de búsqueda y recuperación de información relevante.

Este trabajo de fin de grado tiene como objetivo la integración de un asistente conversacional en la aplicación LAREDO, una herramienta desarrollada en el seno del grupo de investigación ISTR para facilitar a usuarios no expertos la programación de workflows destinados a la construcción y despliegue de predictores de inteligencia artificial (IA). Con ello, se pretende mejorar la accesibilidad y la usabilidad de LAREDO, ofreciendo una interfaz interactiva e intuitiva que ofrezca asistencia inmediata, personalizada y continua durante el uso del sistema, lo que redundará en una experiencia del usuario más satisfactoria.

El proyecto abarca desde el análisis de tecnologías y el diseño del asistente hasta su implementación, optimización y validación automática y humana, integrando modelos de lenguaje, técnicas RAG y una interfaz accesible para el usuario. Los resultados muestran que la incorporación del asistente mejora notablemente la accesibilidad y usabilidad de LAREDO, permitiendo a los usuarios consultar recursos y resolver dudas de manera más eficiente y natural.

Palabras clave: asistente conversacional, generación aumentada por recuperación, modelos de lenguaje de gran escala.

Abstract

Recent advances in Natural Language Processing (NLP) have led to the development of increasingly effective conversational assistants, capable of understanding questions, maintaining dialogues, and generating useful responses. This has transformed the way users access and consult information by enabling more human and accessible interactions. However, these models still present limitations when adapting to private domains or specific needs, which has driven the emergence of approaches such as Retrieval-Augmented Generation (RAG). RAG allows conversational assistants to be combined with search and retrieval mechanisms for relevant information.

This project aims to integrate a conversational assistant into the LAREDO application, a tool developed within the ISTR research group to help non-expert users create workflows for building and deploying artificial intelligence (AI) predictors. The objective is to improve the accessibility and usability of LAREDO by providing an interactive and intuitive interface that enhances the user experience.

The project includes the analysis of relevant technologies and the design of the assistant, as well as its implementation, optimization, and practical validation within LAREDO, integrating language models, RAG techniques, and an accessible user interface. The results show that incorporating the assistant significantly improves the accessibility and usability of LAREDO, enabling users to consult resources and resolve queries more efficiently and naturally.

Keywords: conversational assistant, retrieval-augmented generation, large language models.

Agradecimientos

Quisiera empezar agradeciendo a mi madre, por estar siempre a mi lado en cada momento, dándome fuerzas, ánimos y cariño incondicional. Ella es la persona que siempre me ha apoyado sin importar las circunstancias, celebrando conmigo cada logro y acompañándome en los días difíciles, aunque a veces no lo haya sabido valorar.

También quiero dar las gracias a mis amigos, que han tenido la infinita paciencia de escuchar todas mis quejas durante estos años, compartir risas y ayudarme a escapar de los problemas cuando más lo necesitaba.

Por último, agradecer a Marta, directora de este trabajo, por toda la ayuda, la atención y el tiempo que me ha dedicado a lo largo de este proyecto, así como por su orientación constante y su apoyo en cada etapa del desarrollo.

También quiero dar las gracias a Isaac y Ricardo por su implicación directa en la integración y despliegue del chatbot en LAREDO.

Índice general

1. Introducción	1
2. Fundamentos y evolución de los agentes conversacionales	3
2.1. Orígenes en la traducción automática	3
2.2. Nuevos enfoques para el procesado de lenguaje natural	4
3. Estudio de modelos y herramientas de desarrollo	6
3.1. Comparación funcional entre un LLM y un sistema RAG	6
3.2. Tipos de modelos de lenguaje	7
3.2.1. Clasificación por propósito	7
3.2.2. Clasificación por capacidad de entrada	8
3.2.3. Clasificación por modo de ejecución	8
3.2.4. Clasificación por restricciones de acceso	9
3.3. Evaluación de modelos	10
3.3.1. Análisis de los modelos locales	11
3.3.2. Análisis de modelos online	13
3.3.3. Elección de modelo	14
3.4. Frameworks de Desarrollo	14
3.4.1. LangChain	14
3.4.2. Rasa	14
3.4.3. Elección del framework de desarrollo	15
4. Asistente conversacional	16
4.1. Requisitos del sistema	16
4.2. Diseño	17
4.2.1. Fase de indexación de documentos	17
4.2.2. Fase de recuperación	19
4.2.3. Fase de generación	21
4.2.4. Mantenimiento del historial y resúmenes periódicos	22
4.3. Integración modular mediante LangGraph	23
4.4. Ejecución	25
4.5. Evaluación	25
4.5.1. Evaluación interna y trazabilidad	26
4.5.2. Evaluación automática	27
4.5.3. Evaluación humana	29
5. Diseño de <i>prompts</i>	31
5.1. Buenas prácticas en la redacción de <i>prompts</i>	31
5.2. Aplicación de las buenas prácticas en el diseño del <i>chatbot</i>	32
5.2.1. <i>Prompt</i> principal de respuesta	32
5.2.2. <i>Prompt</i> para historial de conversación	33
5.2.3. <i>Prompt</i> de resumen conversacional	34
5.2.4. <i>Prompt</i> de actualización de resumen	35
5.2.5. <i>Prompt</i> de traducción y normalización de preguntas	36
5.3. Evolución iterativa y evaluación de los <i>prompts</i>	37

6. Interfaz del <i>chatbot</i>	38
6.1. Tecnologías utilizadas	38
6.2. Diseño de la interfaz	39
6.2.1. El botón flotante del asistente	39
6.2.2. Ventana de chat emergente	40
6.2.3. Estructura de la ventana de chat	40
6.3. Publicación e integración del componente	42
6.4. Evaluación	43
7. Comunicación <i>Backend-Frontend</i>	45
7.1. Comunicación entre componentes	45
7.1.1. <i>Backend</i> y uso de Flask	45
7.1.2. <i>Frontend</i> y comunicación con el <i>backend</i>	46
7.1.3. Comunicación en tiempo real mediante SSE	46
7.2. Pruebas locales de los componentes	47
7.3. Automatización del despliegue mediante scripts	48
7.4. Despliegue final en la plataforma	48
7.5. Herramientas y proceso de despliegue final	48
7.5.1. Empaquetado y aislamiento de componentes	48
7.5.2. Orquestación y gestión de servicios	49
7.5.3. Automatización del despliegue y configuración	49
8. Conclusiones y trabajos futuros	50
8.1. Conclusiones	50
8.2. Trabajos futuros	50
Bibliografía	51
A. Anexos	54
A.1. Captura de pantalla del formulario utilizado para la evaluación.	54

CAPÍTULO 1

Introducción

Los avances en procesamiento de lenguaje natural (NLP, por sus siglas en inglés) han dado lugar a asistentes conversacionales cada vez más capaces de comprender preguntas, mantener diálogos y generar respuestas útiles en una gran variedad de contextos. La aparición de los modelos de lenguaje de gran tamaño (LLM, por sus siglas en inglés), capaces de comprender y generar texto en lenguaje natural, ha permitido automatizar tareas cognitivas complejas de manera muy eficiente y ha transformado radicalmente la forma en la que los usuarios interactúan con sistemas informáticos y acceden a la información.

Sin embargo, a pesar de su capacidad para generar texto coherente y responder a preguntas generales, los modelos de lenguaje actuales presentan limitaciones importantes cuando deben tratar con información especializada, reciente o privada, que no está disponible públicamente ni se incluye en los datos de entrenamiento. Este problema resulta especialmente relevante en entornos profesionales o corporativos, donde la precisión y la contextualización de las respuestas es fundamental para la toma de decisiones y la eficiencia operativa. Los asistentes tradicionales, al depender únicamente de conocimiento público y genérico, no pueden dar soluciones satisfactorias en escenarios donde se requiere dominio sobre documentación técnica interna o sobre casos de uso particulares.

Para superar estas limitaciones, han surgido recientemente los sistemas de generación aumentada por recuperación (RAG), que combinan modelos de lenguaje avanzados con técnicas de recuperación de información. Este enfoque permite a los asistentes consultar bases de datos privadas y extraer en tiempo real fragmentos relevantes que se integran en el proceso de generación de respuestas. Así, se mejora notablemente la precisión, actualidad y utilidad de las respuestas proporcionadas, y se optimiza el acceso a la información relevante en grandes volúmenes de datos.

En particular, este tipo de soluciones cobra aún mayor sentido en plataformas complejas como LAREDO, una herramienta surgida en el seno del grupo de investigación ISTR concebida para facilitar a usuarios no expertos, la construcción y despliegue de *workflows* escalables y distribuidos, dirigidos inicialmente a problemas de mantenimiento predictivo en el entorno industrial.

Sin embargo, la amplitud y complejidad de la documentación de LAREDO suponen una barrera para muchos usuarios, que a menudo encuentran dificultades a la hora de localizar información concreta o resolver dudas puntuales de manera ágil. Ante este reto, el objetivo de este trabajo es diseñar e implementar un asistente conversacional inteligente basado en un sistema RAG, plenamente integrado en la plataforma LAREDO. El sistema está pensado para que cualquier usuario pueda consultar en lenguaje natural cualquier aspecto relacionado con LAREDO y obtener respuestas claras, contextualizadas y adaptadas a su necesidad, sin requerir una lectura exhaustiva de la documentación técnica.

Para lograrlo, se abordan los siguientes objetivos específicos:

- Analizar los requisitos funcionales y técnicos del asistente conversacional.
- Construir un sistema RAG que combine recuperación de información de distintas fuentes (tanto documentación interna como recursos web).
- Diseñar y ajustar *prompts* efectivos que guíen el comportamiento del modelo de lenguaje.
- Implementar mecanismos de evaluación tanto automáticos como humanos para validar la calidad de las respuestas.

- Crear una interfaz accesible y atractiva que permita a los usuarios interactuar fácilmente con el *chatbot*.

De este modo, se consigue no solo mejorar la accesibilidad y la experiencia de usuario, sino también optimizar los tiempos de aprendizaje y fomentar la autonomía en el uso de la herramienta.

Después de esta introducción, la memoria se organiza en siete capítulos. Estos describen de manera sistemática el proceso completo que se siguió para el desarrollo, integración y validación del asistente conversacional.

En el capítulo 2 se revisa el estado del arte y los antecedentes relevantes el proceso del lenguaje natural y en asistentes conversacionales. El capítulo 3 analiza los componentes y sistemas actuales necesarios para la construcción del asistente, identificando las herramientas, infraestructuras y soluciones tecnológicas que permiten sentar las bases del desarrollo posterior. El capítulo 4 describe en detalle el diseño y la implementación del *backend* del *chatbot*. En el capítulo 5 se aborda el diseño específico de los *prompts*, presentando las estrategias y buenas prácticas seguidas para guiar el comportamiento del modelo de lenguaje, así como la evolución y validación de los distintos tipos de instrucciones textuales empleadas en el sistema. El capítulo 6 está dedicado al desarrollo del *frontend*, la interfaz de usuario que facilita la interacción natural con el asistente y permite acceder de manera intuitiva a todas las funcionalidades integradas en la plataforma. En el capítulo 7 se detalla la comunicación entre los distintos módulos, así como la integración completa del sistema en el ecosistema de LAREDO. Se explican los mecanismos de orquestación, las conexiones entre *backend* y *frontend* y los retos superados para lograr una solución cohesionada y eficiente. Finalmente, el capítulo 8 recoge las conclusiones alcanzadas a lo largo del trabajo y plantea posibles líneas de mejora y trabajo futuro.

Fundamentos y evolución de los agentes conversacionales

Este capítulo tiene por objeto trazar el desarrollo y la evolución de los *chatbots*, desde su concepción inicial hasta los sofisticados modelos actuales impulsados por inteligencia artificial que conocemos hoy en día. Se destacarán los principales hitos y los cambios de paradigma que han marcado los avances en este campo.

2.1 ORÍGENES EN LA TRADUCCIÓN AUTOMÁTICA

En la década de 1930 se desarrollaron los primeros dispositivos mecánicos para la traducción, como el *Cerebro mecánico* de Artsrouni [1], creado en 1933, considerado un antecedente de los diccionarios mecánicos. Sin embargo, no fue hasta la década de 1940, con la aparición de los primeros ordenadores e influenciados por los éxitos en el descifrado de códigos durante la Segunda Guerra Mundial, cuando se sugirió que la traducción podría abordarse de manera similar al descifrado y recodificación de mensajes. En este contexto, Warren Weaver propuso utilizar computadores para traducir, lo cual inspiró investigaciones como el experimento *Georgetown-IBM* de 1954 [1].

Este experimento, realizado el 7 de enero de 1954, fue la primera demostración pública de traducción automática entre idiomas mediante un ordenador. Desarrollado conjuntamente por la Universidad de Georgetown e IBM, el sistema tradujo automáticamente más de 60 frases del ruso al inglés utilizando un vocabulario de aproximadamente 250 palabras y seis reglas gramaticales básicas [2].

Aunque el sistema era limitado y las frases estaban cuidadosamente seleccionadas, el experimento tuvo un gran impacto mediático y generó expectativas optimistas sobre el futuro de la traducción automática. Los investigadores afirmaron que en pocos años se podría lograr una traducción automática de alta calidad [2]. Sin embargo, el progreso real fue mucho más lento, y en 1966, el informe ALPAC (*Automatic Language Processing Advisory Committee*) [1] concluyó que, tras una década de investigación, las expectativas no se habían cumplido, lo que llevó a una reducción drástica de los fondos para la investigación en traducción automática.

Aún así, aunque no se avanzó significativamente en la investigación, se crearon los primeros *chatbots* basados en reglas. En 1966, Joseph Weizenbaum, del MIT, desarrolló ELIZA, uno de los primeros programas diseñados específicamente para simular una conversación humana. ELIZA imitaba a un psicoterapeuta mediante la coincidencia de patrones y sustituciones en el texto de entrada del usuario, generando así respuestas predefinidas [3]. Más tarde, en 1972, Kenneth Colby desarrolló PARRY, otro *chatbot* temprano que simulaba a una persona con paranoia. Este era más avanzado que ELIZA en cuanto al uso de reglas, e incluso superó una versión modificada de la prueba de Turing, en la que psiquiatras reales no pudieron distinguirlo de un paciente auténtico [3].

No obstante, estos modelos compartían las mismas limitaciones ya mencionadas. Al estar basados únicamente en reglas, carecían de una comprensión real del lenguaje y dependían completamente de estructuras predefinidas [1].

Para superar las limitaciones de los modelos basados en reglas, surgió el campo del procesamiento de lenguaje natural NLP, una disciplina de la inteligencia artificial centrada en la interacción entre computadoras y lenguaje humano. El NLP combina técnicas de lingüística computacional, aprendizaje automático y aprendizaje profundo para permitir que las máquinas comprendan, interpreten y generen

texto de manera eficaz. Su objetivo es facilitar la comunicación entre humanos y sistemas mediante el lenguaje natural [4].

2.2 NUEVOS ENFOQUES PARA EL PROCESADO DE LENGUAJE NATURAL

No fue hasta la década de 1990 cuando comenzaron a desarrollarse enfoques más avanzados, que se acercaban a los sistemas de procesamiento del lenguaje natural. La creciente disponibilidad de grandes corpus de texto y el aumento de la potencia computacional fueron factores clave que impulsaron el auge del NLP estadístico [5].

En la figura 2.1 se ilustra esta evolución, mostrando cómo ha contribuido cada avance al desarrollo de modelos más sofisticados. A continuación, se analiza cada etapa, comenzando con la llegada de los modelos estadísticos.

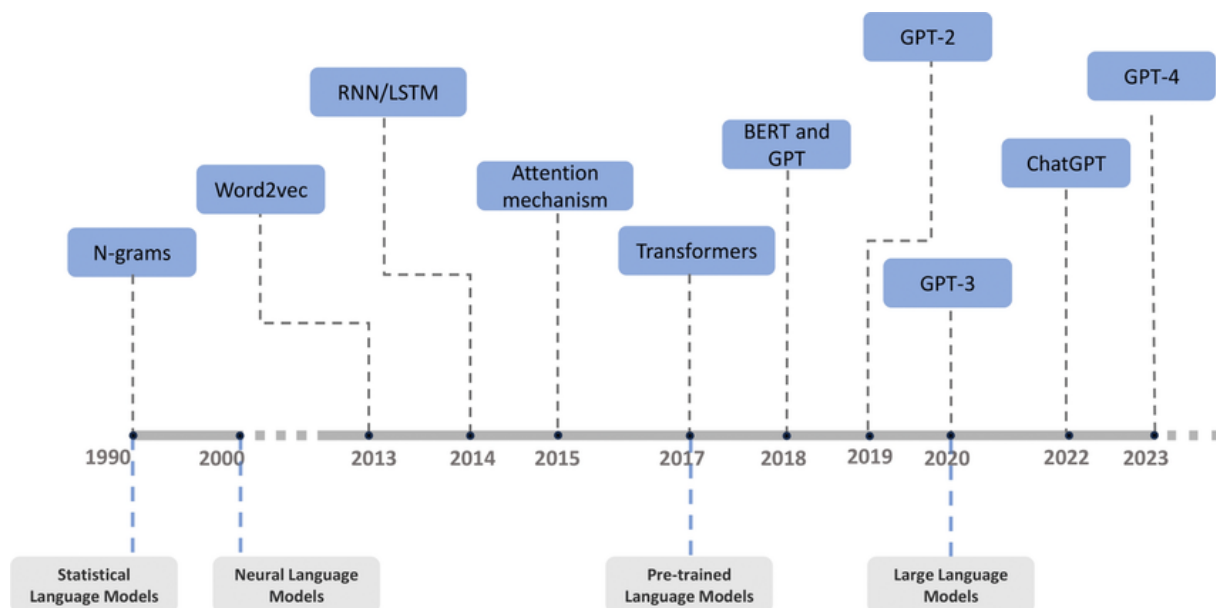


Figura 2.1: Avances del NLP hasta los LLM actuales [5]

Estos modelos superaban a los basados en reglas, ya que podían procesar grandes cantidades de datos con mayor facilidad y precisión. Esto los hizo útiles en aplicaciones como la predicción de texto, el reconocimiento de voz y la detección de spam. Un enfoque común dentro de los modelos estadísticos es el uso de *n-grams*, que consiste en representar secuencias de N palabras consecutivas para modelar la probabilidad de aparición de una palabra en función de su contexto previo. Sin embargo, estos modelos aún no lograban comprender completamente las relaciones semánticas entre conceptos ni interpretar las dependencias a largo plazo dentro de un texto [4].

Para superar estas limitaciones, en la primera década del siglo XXI surgieron los modelos de lenguaje basados en redes neuronales, que evolucionaron hacia las redes neuronales recurrentes (RNN). Esto permitió a los modelos comprender mejor el contexto de las palabras. No obstante, las RNN presentaban problemas para retener información, eran propensas al sobreajuste y requerían largos tiempos de entrenamiento. Para mitigar estos inconvenientes, se desarrollaron las *Long Short-Term Memory* (LSTM), una variante de las RNN que mejoraba la retención de información mediante mecanismos de control del flujo de datos. Esto supuso avances importantes en el modelado del lenguaje y la traducción automática [4].

A pesar de estos progresos, los modelos aún no representaban adecuadamente las relaciones semánticas entre palabras. Para solventar este problema, se introdujeron los *word embeddings* o representaciones vectoriales de palabras, como Word2Vec [6] y GloVe [7]. Estos modelos representaban las palabras en un espacio vectorial continuo, lo que permitía capturar relaciones semánticas entre ellas.

Sin embargo, no identificaban las palabras homónimas y homógrafas y de no lograban captar el contexto completo en el que se utilizaban estas palabras dentro de una frase.

El avance más significativo llegó en 2017 con la creación del modelo *Transformer*. Este modelo introdujo el mecanismo de *self-attention*, que permite capturar relaciones complejas entre palabras sin necesidad de procesarlas de forma secuencial. Esto posibilitó una mayor paralelización del entrenamiento y un procesamiento más eficiente de grandes volúmenes de datos, solucionando muchas de las limitaciones previas. Basándose en esta arquitectura, surgieron los modelos de lenguaje a gran escala, con miles de millones de parámetros, como BERT [8] y GPT [9], que han establecido los estándares actuales en el procesamiento del lenguaje [4]. No obstante, a pesar de su potencia, los LLM requieren enormes cantidades de datos y recursos computacionales. Por otra parte, plantean desafíos éticos importantes, como la generación de desinformación, los sesgos presentes en los datos de entrenamiento y la dificultad para explicar su razonamiento, por lo que no todos los problemas están aún resueltos [10, 11].

La evolución de los LLM, desde los modelos probabilísticos hasta los actuales basados en *Transformers*, ha impulsado un rápido progreso en el ámbito de la IA y el NLP. Sin embargo, estos modelos aún están en pleno desarrollo. Por ello, es fundamental mantenerse actualizado sobre las soluciones más innovadoras y de mayor rendimiento.

Estudio de modelos y herramientas de desarrollo

Para la construcción de un *chatbot* basado en RAG es fundamental realizar una selección adecuada tanto del modelo de lenguaje, es decir, el LLM, como del *framework* de desarrollo. La combinación de la capacidad generativa de los LLMs con la recuperación precisa de información proveniente de fuentes externas permite desarrollar sistemas capaces de responder preguntas fundamentadas en un corpus documental específico. Esto garantiza respuestas más contextualizadas, fiables y precisas que las generadas por un LLM.

En este capítulo se lleva a cabo un análisis de las distintas opciones de modelos de lenguaje y *frameworks* de desarrollo relevantes para la implementación de un *chatbot* basado en RAG. En primer lugar, se profundiza en la diferencia esencial entre el funcionamiento de un LLM y un sistema basado en RAG. A continuación, se presentan las distintas clasificaciones de los LLMs, tanto por su propósito como por sus restricciones de acceso. Posteriormente, se comparan dos *frameworks* de desarrollo prominentes en este ámbito: LangChain y Rasa. Finalmente, se selecciona tanto el modelo como las herramientas para construir el asistente conversacional.

3.1 COMPARACIÓN FUNCIONAL ENTRE UN LLM Y UN SISTEMA RAG

Los LLMs se caracterizan por generar respuestas coherentes a partir de los datos conseguidos durante su fase de entrenamiento. Al recibir una pregunta en un LLM, el modelo utiliza únicamente la información almacenada en sus parámetros, es decir, su conocimiento interno. En cambio, en un sistema RAG integra el LLM con un módulo de recuperación de información, permitiendo consultar fuentes externas y aportar así información más específica y actualizada [12-15].

Ambos enfoques comparten las etapas generales de entrada, procesamiento y generación, pero difieren en la forma de gestionar el conocimiento y el contexto. La figura 3.1 ilustra de forma esquemática la estructura y diferencias principales entre ambos sistemas.

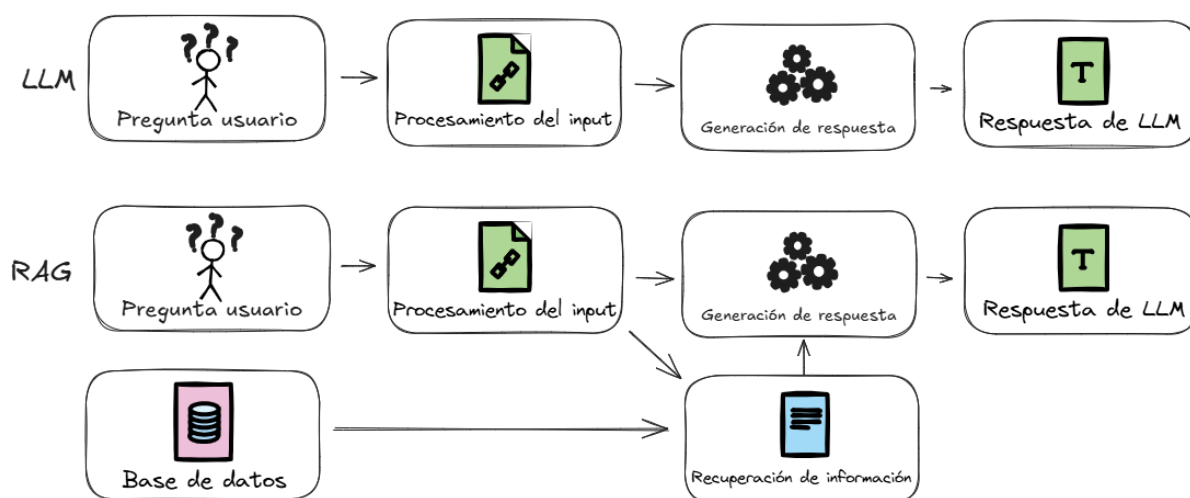


Figura 3.1: Comparación entre LLM y RAG

Entrada del usuario. En ambos sistemas, el proceso comienza cuando el usuario plantea una pregunta. En un LLM, esta entrada se utiliza directamente como contexto para la generación de la respuesta. En un sistema RAG, la consulta también sirve como contexto, pero además activa un mecanismo de búsqueda en una base de datos de documentos relevantes.

Gestión del contexto y del conocimiento. En el LLM, el contexto disponible para la respuesta se limita a la entrada del usuario y al conocimiento almacenado en los parámetros del modelo. En RAG, los fragmentos recuperados externamente se integran a la información que recibe el modelo, enriqueciendo el contexto con datos actualizados y específicos.

Generación de la respuesta. En ambos casos, la respuesta es generada por el modelo de lenguaje. Sin embargo, en el LLM depende únicamente de su conocimiento interno, mientras que en el RAG se combina este conocimiento general con los datos concretos recuperados, incrementando la precisión y la actualidad de las respuestas.

Aunque ambos sistemas comparten una estructura básica, la diferencia principal radica en la gestión del conocimiento: el LLM funciona con información estática, mientras que el RAG incorpora mecanismos de recuperación y actualización de información externa, ampliando considerablemente el alcance y la precisión de las respuestas [14, 16].

3.2 TIPOS DE MODELOS DE LENGUAJE

El modelo de lenguaje representa el componente central del *chatbot*, ya que es responsable tanto de la comprensión de las entradas del usuario como de la generación de respuestas coherentes y relevantes. La elección del LLM más adecuado depende de múltiples factores. Para orientar esta selección, se propone una clasificación basada en tres dimensiones fundamentales: el propósito funcional del modelo, sus capacidades de entrada y salida, y finalmente, aspectos prácticos como el modo de ejecución (local o en la nube) y las restricciones de acceso (abierto o cerrado). Cada una de estas dimensiones ofrece una perspectiva diferente, permitiendo evaluar de forma más precisa qué modelo se adapta mejor a los requisitos específicos de un *chatbot* basado en RAG.

3.2.1 Clasificación por propósito

La primera clasificación se centra en el enfoque principal para el que ha sido diseñado el modelo. Se pueden identificar dos categorías principales:

Modelos generales

Los modelos de propósito general están entrenados con enormes volúmenes de datos extraídos de la web. Esta característica les permite abordar una amplia variedad de tareas sin requerir un entrenamiento adicional específico. Entre sus capacidades más destacadas se encuentran la generación de texto, el resumen automático, la traducción, la respuesta a preguntas o la escritura de código, entre otras. Algunos ejemplos representativos de esta categoría son LLaMA [17], Gemini [18] o DeepSeek [19].

Sin embargo, aunque estos modelos sobresalen por su habilidad para comprender y generar lenguaje de forma coherente y fluida, diversos estudios han demostrado que su rendimiento puede ser inferior al de modelos entrenados específicamente para tareas especializadas [2, 13, 20]. Esta limitación resalta la necesidad de incorporar mecanismos de recuperación de información, como los que ofrece la arquitectura RAG, que permiten enriquecer el conocimiento general de estos modelos con datos actualizados y específicos del dominio objetivo, como documentos internos y manuales de uso.

Modelos de *embeddings*

A diferencia de los modelos generativos, los modelos de *embeddings* están diseñados para convertir entradas de texto en vectores numéricos de alta dimensión que capturan su significado semántico. Esta representación vectorial permite medir la similitud entre textos de forma eficiente y precisa,

lo que los convierte en un componente esencial en la etapa de recuperación de información de un sistema RAG.

Gracias a esta capacidad, estos modelos permiten comparar la consulta del usuario con el corpus documental para identificar los fragmentos más relevantes. Existen diferentes tipos de modelos de *embeddings*, como Gecko [21] y Gemini Embedding [22] donde cada uno de ellos está optimizado para un contexto de uso diferente.

Es importante destacar que muchos LLMs ya incluyen mecanismos internos para generar *embeddings* optimizados para su arquitectura, lo que facilita su integración en sistemas RAG sin necesidad de herramientas adicionales. La elección del modelo de *embeddings* impacta directamente en la calidad de los resultados recuperados y, por tanto, en la relevancia y precisión de las respuestas generadas [22].

3.2.2 Clasificación por capacidad de entrada

Además del propósito, es relevante considerar las capacidades de entrada que ofrece cada modelo. En este sentido, se puede distinguir entre:

Modelos unimodales

Los modelos unimodales han sido diseñados para procesar únicamente un tipo de dato, siendo el texto el formato más habitual en el caso de los modelos de lenguaje actuales. Este enfoque implica que el modelo es capaz de recibir, analizar y generar información exclusivamente en ese tipo de entrada, sin tener en cuenta otros formatos como imágenes, audio o vídeo.

Modelos multimodales

Los modelos multimodales amplían las capacidades de los modelos generales al permitir el procesamiento conjunto de distintos tipos de datos, como texto, imágenes, audio o vídeo [23]. Gracias a esta integración, son capaces de comprender un contexto más amplio y generar respuestas más completas y enriquecidas. Entre los ejemplos más destacados se encuentran GPT-4 [24], Gemini [18] y Copilot [25], que incorporan esta capacidad multimodal con distintos niveles de integración y rendimiento. En el contexto de un *chatbot* RAG, la incorporación de modelos multimodales resulta especialmente relevante cuando el corpus de documentos incluye formatos heterogéneos. En estos casos, un LLM multimodal puede aprovechar de manera más eficaz la información contenida en imágenes o diagramas para ofrecer respuestas más precisas y útiles.

Sin embargo, esta versatilidad viene acompañada de un coste computacional considerablemente mayor. Por ello, la decisión de integrar modelos multimodales debe considerar cuidadosamente los recursos disponibles, tanto en términos de *hardware* como de tiempo de inferencia.

3.2.3 Clasificación por modo de ejecución

La elección del modelo de lenguaje para un sistema RAG no depende únicamente de su rendimiento técnico, sino también del modo en que se accede y ejecuta dicho modelo. El entorno de ejecución (local u online) condiciona aspectos fundamentales como el grado de control, la privacidad de los datos, las posibilidades de personalización y los costes asociados [26].

Ejecución local

La ejecución local implica descargar y ejecutar el modelo directamente en la máquina del usuario, ya sea en estaciones de trabajo o servidores equipados con GPUs de alto rendimiento. Esta modalidad ofrece ventajas notables: garantiza una mayor privacidad, elimina costes recurrentes por uso de una API, permite un control absoluto sobre el modelo y abre la puerta a múltiples posibilidades de personalización.

Una interfaz de programación de aplicaciones (API, por sus siglas en inglés) es un mecanismo que permite que diferentes aplicaciones se comuniquen entre sí, normalmente a través de Internet. En el contexto de los modelos de lenguaje, muchas plataformas ofrecen acceso a modelos avanzados

mediante APIs, de modo que el usuario envía una consulta a un servidor externo y recibe la respuesta generada por el modelo.

Con el auge de herramientas como Huggingface¹ y Ollama², esta opción resulta cada vez más accesible, al facilitar la implementación y optimización de modelos de código abierto en entornos locales. Sin embargo, optar por la ejecución local también supone una inversión inicial considerable en *hardware*, así como conocimientos técnicos avanzados para la correcta configuración, actualización y mantenimiento del sistema. Además, la escalabilidad queda limitada a los recursos físicos disponibles, y pueden surgir problemas de compatibilidad con arquitecturas no estándar o de difícil optimización. Por otro lado, la actualización a nuevas versiones de los modelos o del *software* base suele ser más compleja y lenta, ya que requiere procedimientos manuales y puede implicar la reinstalación o reconfiguración del sistema entero [26].

Ejecución online

Por el contrario, la ejecución online se basa en el acceso a modelos alojados por terceros mediante APIs, lo que evita la necesidad de gestionar *hardware* propio. Este enfoque resulta especialmente atractivo cuando se desea acceder de forma inmediata a modelos de última generación, garantizar versiones siempre actualizadas o escalar el uso según la demanda, sin afrontar una inversión inicial elevada. Sin embargo, este tipo de ejecución también presenta limitaciones importantes como la dependencia del proveedor y sus políticas puede restringir el nivel de personalización, mientras que los costes pueden aumentar significativamente en función del volumen de uso. Además, la transmisión de datos a servidores externos introduce riesgos potenciales en materia de privacidad y puede generar latencias que afectan a la experiencia de uso al depender de la red [26].

3.2.4 Clasificación por restricciones de acceso

El tercer aspecto crucial a la hora de seleccionar un modelo de lenguaje es el tipo de acceso permitido, que se puede clasificar en dos grandes categorías: modelos propietarios (cerrados) y modelos de código abierto (*open source*).

Modelos propietarios

Los modelos propietarios, como GPT-4³, Claude 3⁴ o Gemini 2.0⁵, son desarrollados y mantenidos por grandes empresas tecnológicas como OpenAI, Anthropic o Google. Estos modelos destacan por ofrecer un alto rendimiento y acceso a las capacidades más avanzadas del mercado. El acceso se realiza generalmente a través de APIs, bajo modelos de suscripción o pago por uso, lo que facilita la integración en aplicaciones pero introduce costes recurrentes y dependencia de las condiciones comerciales del proveedor [26].

Para comparar estos modelos en la tabla 3.1 se han seleccionado aquellos parámetros técnicos y prácticos que resultan más relevantes a la hora de optar por una alternativa u otra en el contexto de aplicaciones reales.

Modelo	Ventana de contexto	Precio	Comentario Precio
GPT-4o	128k <i>tokens</i>	OpenAI Pricing	Cualquier uso de la API requiere tarjeta de crédito.
Gemini 2.0	1M <i>tokens</i>	Gemini Pricing	Acceso reducido sin coste.
Claude 3	200k <i>tokens</i>	Anthropic Pricing	Cualquier uso de la API requiere tarjeta de crédito.
Grok	128k <i>tokens</i>	xAI API Models	Inicialmente gratuito, ahora bajo suscripción.

Tabla 3.1: Comparativa de modelos propietarios relevantes (junio 2025)

¹<https://huggingface.co/models>

²<https://ollama.com/>

³<https://openai.com/es-ES/index/gpt-4/>

⁴<https://claude.ai>

⁵<https://gemini.google.com>

En primer lugar, la ventana de contexto refleja la cantidad máxima de información que el modelo es capaz de procesar de manera simultánea, medida en *tokens*. Este parámetro determina la longitud máxima de los documentos o la extensión de las conversaciones que pueden ser gestionadas sin pérdida de coherencia contextual. La importancia de este valor es especialmente notable en aplicaciones complejas, como el análisis de grandes volúmenes de texto, la recuperación de información a partir de múltiples documentos, o la gestión de diálogos prolongados, donde la limitación de contexto puede afectar significativamente a la calidad y precisión de las respuestas generadas por el modelo [27, 28].

Por otra parte, el precio, medido en dólares estadounidenses por millón de *tokens* procesados, representa un criterio clave en la viabilidad y escalabilidad de cualquier proyecto que haga uso intensivo de LLM propietarios, ya que los costes pueden crecer rápidamente en función del volumen de interacción o procesamiento requerido. Dado que estos precios varían con el tiempo y según el modelo seleccionado, en cada caso se proporciona un enlace a la página oficial donde se pueden consultar las tarifas actualizadas.

Finalmente, conviene destacar que todos los modelos incluidos en la comparativa ofrecen soporte multimodal, es decir, la capacidad de procesar entradas en formato texto como en imágenes o audio.

Se puede observar que todos los modelos analizados disponen actualmente de una ventana de contexto considerablemente amplia para el uso habitual en aplicaciones prácticas. Por ejemplo, una ventana de contexto de 1 millón de *tokens* (como la ofrecida por Gemini 2.0) permite procesar aproximadamente entre 700.000 y 800.000 palabras en inglés o español, lo que equivale a más de 1.000 páginas de texto estándar. Esta conversión varía según el idioma y la longitud media de palabra, pero se estima una media de 1,3 a 1,5 *tokens* por palabra en inglés y español, y unas 500 palabras por página [27]. Del mismo modo, los modelos con ventanas de 128.000 o 200.000 *tokens* (como GPT-4o o Claude 3) pueden gestionar textos que superan las 100 páginas, siendo más que suficiente para este trabajo, tanto en recuperación de información como en diálogos extensos o procesamiento documental.

Actualmente, Gemini 2.0 es el único modelo que ofrece acceso reducido sin coste y sin necesidad de registrar una tarjeta de crédito, lo que facilita la experimentación y el desarrollo de prototipos. Por el contrario, el uso de la API de modelos como GPT-4o, Claude 3 o Grok requiere obligatoriamente el registro de una tarjeta de crédito, incluso para realizar pruebas.

Modelos de código abierto (*open source*)

En contraposición, los modelos de código abierto, como *LLaMA 3*, *Mistral* o *Qwen*, se distribuyen bajo licencias que permiten su uso, modificación y despliegue sin restricciones comerciales⁶. Este enfoque permite una libertad considerable: control total sobre los datos, capacidad de personalización del modelo y ejecución sin costes asociados al uso por API.

Aunque su implementación requiere una inversión inicial en *hardware* adecuado y conocimientos técnicos para su configuración, los modelos *open source* han progresado notablemente en los últimos años. Si bien históricamente su rendimiento era inferior al de sus equivalentes propietarios, la evolución de su arquitectura, la mejora en las capacidades de razonamiento y la ampliación de ventanas de contexto los posicionan como alternativas competitivas y cada vez más viables para la implementación de sistemas RAG [29].

3.3 EVALUACIÓN DE MODELOS

Una vez analizadas las principales alternativas disponibles para el desarrollo del *chatbot*, es necesario seleccionar las herramientas técnicas que conformarán la solución final. Dado que la arquitectura es modular, los distintos componentes pueden combinarse de forma flexible, lo que permite seleccionar

⁶<https://huggingface.co/models>

cada uno de forma independiente. En primer lugar, se abordará la decisión entre utilizar un modelo local o un modelo alojado en una API online.

En el caso de los modelos locales, se pueden emplear plataformas como Ollama o Hugging Face, que permiten ejecutar modelos de lenguaje de código abierto sin coste adicional, siendo la única limitación relevante la capacidad de procesamiento del equipo. Por otro lado, para los modelos online se busca una solución accesible para cualquier usuario sin coste económico. En este contexto, se opta por utilizar Gemini, ya que representa una de las opciones gratuitas más potentes disponibles actualmente.

3.3.1 Análisis de los modelos locales

Para el análisis de modelos locales, se ha seleccionado la plataforma Ollama, que destaca por su simplicidad de uso y por facilitar la ejecución de modelos *open source* como Mistral o LLaMA 2, especialmente en versiones de tamaño medio (entre 7B y 13B parámetros). Esta elección permite agilizar las pruebas sin requerir conocimientos técnicos avanzados en despliegue de modelos.

En el contexto de los modelos de lenguaje, el término parámetro hace referencia a los valores numéricos internos (pesos y sesgos) que el modelo aprende durante su entrenamiento. Estos parámetros determinan cómo procesa el texto y genera respuestas. El tamaño de un modelo, expresado en miles de millones de parámetros (por ejemplo, 7B significa 7 mil millones de parámetros), influye directamente en su capacidad para captar patrones complejos del lenguaje, aunque también implica mayores requerimientos computacionales. El número de parámetros tiene un impacto directo sobre la memoria RAM requerida: a mayor tamaño del modelo, mayor será la cantidad de memoria necesaria para cargar todos los parámetros en la ejecución. Por ejemplo, un modelo de 7B parámetros suele requerir el doble de RAM. Esto implica que la elección del modelo debe considerar tanto la capacidad de procesamiento del equipo como el nivel de sofisticación lingüística que se desea alcanzar [28].

Las pruebas se han realizado sobre un equipo con las siguientes características de *hardware*⁷: 16 GB de RAM, tarjeta gráfica Intel Arc con 8 GB de VRAM, y procesador Intel Core Ultra 7 155H. A partir de esta configuración, se analizaron tres posibles modos de ejecución de modelos: mediante CPU, GPU dedicada y NPU (*Neural Processing Unit*, por sus siglas en inglés).

Es necesario entender cómo se calcula la relación entre el número de parámetros de un modelo de lenguaje y la memoria de vídeo (VRAM) necesaria para ejecutarlo de manera eficiente en *hardware* local. El tamaño en memoria de un modelo depende directamente del número total de parámetros y la precisión con la que se almacenan. Para la mayoría de implementaciones actuales, se emplea una precisión de 16 bits (*FP16*), equivalente a 2 bytes por parámetro. De este modo, una estimación sencilla de la VRAM necesaria se expresa mediante la siguiente fórmula:

$$\text{VRAM (GB)} \approx \text{Parámetros (miles de millones)} \times \text{Bytes por parámetro} \quad (3.1)$$

Por ejemplo, para un modelo de 7.000 millones de parámetros (7B) con 16 bits de precisión, el cálculo sería: $7 \times 2 = 14$ GB de VRAM. En el caso de utilizar cuantización (por ejemplo, 8 bits en lugar de 16), la memoria necesaria se reduce a la mitad. En la tabla 3.2 se resumen las necesidades estimadas de VRAM para diferentes tamaños de modelos LLM, con una precisión de 16 bits.

Tamaño del modelo	Parámetros	VRAM requerida (aprox.)
Pequeño	0.1B – 2B	0.2 – 4 GB
Mediano	2B – 10B	4 – 20 GB
Grande	10B – 20B	20 – 40 GB
Muy grande	20B – 70B	40 – 140 GB

Tabla 3.2: Requisitos estimados de VRAM calculados con la fórmula 3.1 en precisión de 16 bits.

⁷<https://www.samsung.com/es/computers/galaxy-book/galaxy-book4-pro360/buy/>

Tras analizar las opciones según tamaño y requisitos de *hardware*, se eligió la versión de 7B parámetros de LLaMA 2⁸ por su buen equilibrio entre rendimiento y facilidad de uso en equipos convencionales. Esta decisión se tomó después de experimentar con modelos más pequeños, con los que se realizaron pruebas informales para comprobar cómo respondían a preguntas variadas tanto en inglés como en español. Estas pruebas pusieron de manifiesto que, especialmente en español, los modelos pequeños presentan limitaciones importantes en cuanto a fluidez y precisión.

Una vez seleccionado el modelo, el siguiente paso consiste en realizar pruebas de inferencia para evaluar su comportamiento práctico en el entorno real. En este apartado, se analiza no solo la velocidad de generación de *tokens*, sino también la latencia o el tiempo que transcurre desde que se envía la consulta hasta que el modelo comienza a responder (*Time To First exitToken*, TTFT). La latencia depende tanto de la capacidad del modelo como del tamaño de la entrada proporcionada, y resulta un factor clave en aplicaciones interactivas, donde se espera una respuesta casi instantánea. Además, esta métrica es esencial para valorar la viabilidad real del modelo en el contexto de uso previsto [30].

Inferencia con CPU

La primera prueba consistió en ejecutar el modelo exclusivamente mediante CPU, una situación común para usuarios sin tarjeta gráfica dedicada. Aunque Ollama permite este modo de ejecución, el rendimiento es considerablemente inferior, con velocidades de inferencia que oscilan entre 5 y 10 *tokens* por segundo. Además, las operaciones solo con CPU exigen una cantidad sustancial de RAM del sistema para compensar la falta de memoria VRAM de la GPU; por ejemplo, un modelo de 7B requiere al menos 16 GB de RAM para funcionar de manera estable únicamente con CPU.⁹ Por otra parte, la latencia observada en cada respuesta generada es notablemente alta, ya que la CPU no está optimizada para los cálculos de los LLMs.

Inferencia con GPU (Intel Arc)

En una segunda prueba, se intentó ejecutar el modelo utilizando la GPU dedicada Intel Arc. Sin embargo, la mayoría de motores de ejecución de LLM actuales no ofrecen soporte oficial para GPUs Intel, limitando su compatibilidad a GPUs NVIDIA mediante CUDA¹⁰. Este hecho impidió realizar pruebas significativas en esta configuración, lo que pone de manifiesto una importante barrera técnica: el rendimiento óptimo con LLMs locales depende en gran medida de disponer de *hardware* específico compatible, lo cual no siempre está al alcance del usuario promedio. Por tanto, la latencia en esta configuración no pudo ser evaluada, pero la expectativa es que, de habilitarse el soporte completo, la latencia se reduciría drásticamente en comparación con la CPU, debido a la capacidad de cómputo paralelo de las GPUs [31].

Inferencia con NPU

Por último, se investigó la viabilidad de utilizar la NPU integrada en el procesador para acelerar la inferencia. Aunque esta tecnología es reciente y no está ampliamente disponible, su estudio resultó de gran interés. Las NPUs permiten un bajo consumo energético y buena eficiencia para tareas específicas de IA, y pueden ser utilizadas mediante formatos como *ONNX*¹¹. Sin embargo, las pruebas confirman que la NPU, si bien ofrece cierta aceleración para modelos más pequeños y operaciones específicas, no logra ser un sustituto eficaz de una GPU dedicada para LLMs de tamaño medio o grande [32-34]. Así, la latencia, aunque algo inferior a la observada con CPU, siguió siendo demasiado elevada para aplicaciones que exigen tiempos de respuesta bajos e interacción con el usuario [32-34]. Las causas principales de esta limitación son las siguientes:

- **Optimización para tareas específicas como el reconocimiento de voz y la clasificación de audio.** Estas tareas suelen implicar entradas de longitud fija y operaciones con enteros, lo

⁸<https://ollama.com/library/llama2>

⁹<https://collabnix.com/does-ollama-need-a-gpu/>

¹⁰<https://developer.nvidia.com/cuda-toolkit>

¹¹<https://onnx.ai/>

que las hace adecuadas para la arquitectura de las NPUs. Sin embargo, los LLMs presentan desafíos para las NPUs debido a sus entradas de longitud variable y la necesidad de operaciones en punto flotante, como la normalización de capas (*LayerNorm*) y los mecanismos de atención (*Attention*) [32, 33].

- **Limitaciones técnicas en NPUs.** Aunque existen técnicas modernas como la cuantificación por grupo, que permiten reducir el tamaño de los modelos sin sacrificar precisión [28], su compatibilidad y eficiencia en arquitecturas NPU no está siempre garantizada. Además, al tratarse de dispositivos integrados, las NPUs comparten el ancho de banda de memoria con el resto del sistema, lo que puede provocar cuellos de botella cuando se procesan modelos con pesos y activaciones de gran tamaño. Esto contrasta con las GPUs dedicadas, que cuentan con VRAM de alta velocidad optimizada específicamente para estas tareas.

En resumen, aunque las NPUs pueden representar una mejora marginal en eficiencia energética para modelos pequeños, actualmente no constituyen una solución viable para la inferencia de LLMs complejos. Por tanto, el despliegue efectivo de modelos locales requiere, en general, una GPU compatible, de lo que no todo usuario dispone [32].

En la tabla 3.3 se resumen los resultados de las pruebas realizadas. En todos los casos se ha utilizado el modelo Llama-2-7B, mencionado anteriormente, sin aplicar técnicas de cuantificación (16 bits por parámetro).

Configuración	Tokens/s	Latencia (TTFT)	Uso de RAM/VRAM	Observaciones
CPU	5–10	5 - 7 s	14 GB RAM	Rendimiento limitado; alta carga en CPU y RAM.
GPU (Intel Arc)	5–10	5 - 7 s	14 GB RAM	El modelo usa la CPU porque no es compatible con la gráfica Intel.
NPU	15 - 20	3 - 5 s	14 GB RAM	Mejora respecto a CPU pero no todos los ordenadores disponen de esta tecnología.

Tabla 3.3: Resultados de inferencia con LLaMA 2 7B en diferentes configuraciones de *hardware*

3.3.2 Análisis de modelos online

Para la evaluación de modelos LLM online, se consideraron las diversas alternativas mostradas, entre ellas GPT-4 de OpenAI y Claude Anthropic. Sin embargo, la API de Google Gemini se posicionó como la opción más adecuada, principalmente por ofrecer un acceso gratuito y sencillo a modelos avanzados. Esta elección nos permitió realizar las pruebas sin incurrir en costes iniciales.

A diferencia de los modelos ejecutados localmente, en los modelos online la capacidad de procesamiento y la velocidad de respuesta no dependen del *hardware* del usuario, sino de los recursos y límites definidos por el proveedor de la API. En la tabla 3.4 se muestran los valores máximos para los principales modelos gratuitos de Gemini, así como su equivalencia en *tokens* por segundo y la latencia típica observada en cada caso.

Modelo	TPM (nivel gratuito)	Tokens/segundo	Latencia (TTFT)
Gemini 2.5 Pro	2 000 000	33,33	2-6 s
Gemini 2.0 Flash	4 000 000	66,66	0,5 s

Tabla 3.4: Límites de *tokens* por minuto (TPM), *tokens* por segundo y latencia para los modelos gratuitos de Gemini (Junio 2025)¹²

La tabla muestra diferencias notables entre ambos modelos: Gemini 2.0 Flash duplica la capacidad de procesamiento por minuto respecto a Gemini 2.5 Pro y reduce drásticamente la latencia de respuesta inicial (TTFT), pasando de varios segundos en el modelo Pro (2 - 6 s) a solo medio segundo en Flash. Esta rapidez en la generación y respuesta es fundamental para aplicaciones interactivas. Por el contrario, Gemini 2.5 Pro, a pesar de ser más avanzado, presenta tiempos de espera mayores y, además, está sujeto a restricciones de uso adicionales y una disponibilidad inferior en el plan gratuito.

Estas cifras representan la capacidad máxima teórica permitida por la API; no obstante, factores como la latencia de red, el tamaño de las entradas y la disponibilidad del servicio pueden afectar el rendimiento real. Además, aunque el uso gratuito permite operar sin costes directos, está limitado a fines de prueba y supone el procesamiento de los datos con fines de mejora del producto¹³.

3.3.3 Elección de modelo

Una vez analizadas las distintas alternativas y evaluadas sus limitaciones y capacidades, se optó por Gemini 2.0 Flash como modelo principal. Optar por un modelo online no solo evita la necesidad de disponer de *hardware* dedicado o configuraciones complejas, sino que además permite el acceso a modelos avanzados desde cualquier dispositivo, independientemente de su capacidad local. Adicionalmente, la potencia y calidad de generación de los modelos de Gemini superan ampliamente las capacidades de los sistemas locales.

3.4 FRAMEWORKS DE DESARROLLO

El *framework* de desarrollo es esencial para la construcción de asistentes conversacionales, ya que proporciona las herramientas y abstracciones necesarias para integrar y orquestar los distintos módulos del sistema, incluido el modelo de lenguaje. En el contexto de *chatbots*, especialmente aquellos que implementan arquitecturas RAG, destacan dos *frameworks* de código abierto con amplio reconocimiento y comunidades activas: LangChain¹⁴ y Rasa¹⁵. Aunque ambos pueden usarse para crear asistentes conversacionales, cada uno sigue una filosofía y una forma de funcionamiento diferente. Esto hace que sean más o menos adecuados para sistemas que necesitan recuperación aumentada.

3.4.1 LangChain

LangChain es un *framework* diseñado específicamente para facilitar el desarrollo de aplicaciones basadas en modelos de lenguaje, con especial énfasis en arquitecturas como RAG. Su diseño modular permite combinar de manera flexible distintos componentes, como LLMs, bases de datos vectoriales, agentes, herramientas externas o fuentes de conocimiento estructurado. Esta capacidad es especialmente útil para la fase de recuperación en sistemas RAG, donde se requiere integrar datos provenientes de documentos, APIs o bases de datos.

Además, proporciona utilidades avanzadas para la construcción de flujos de procesamiento complejos mediante cadenas (*chains*) o nodos modulares con LangGraph¹⁶ [35]. A su vez, a través de su plataforma de observabilidad LangSmith¹⁷, el *framework* permite monitorizar, depurar y evaluar conversaciones, lo que facilita el desarrollo iterativo, la optimización del sistema y la mejora continua de los resultados.

Cabe destacar que LangChain es una herramienta relativamente reciente y está en continuo desarrollo¹⁸, por lo que tanto su funcionalidad como su documentación y manuales evolucionan rápidamente. Esto implica que el uso de la librería y las mejores prácticas pueden cambiar en cortos periodos de tiempo, requiriendo atención a las actualizaciones.

3.4.2 Rasa

Rasa es un *framework* enfocado en la construcción de asistentes conversacionales con gestión estructurada del diálogo. Ofrece herramientas para definir flujos conversacionales complejos mediante el uso de historias, reglas y políticas personalizadas. Asimismo, permite un control detallado del procesamiento del lenguaje natural y del seguimiento del estado de la conversación, lo que lo hace especialmente útil para *chatbots* orientados a tareas específicas.

¹³<https://support.google.com/gemini/answer/13594961>

¹⁴<https://www.langchain.com/>

¹⁵<https://rasa.com/>

¹⁶<https://www.langchain.com/langgraph>

¹⁷<https://www.langchain.com/langsmith>

¹⁸<https://python.langchain.com/docs/introduction/>

No obstante, su arquitectura está más orientada a sistemas de diálogo basados en reglas y entrenamiento supervisado, lo que puede dificultar la implementación directa de arquitecturas RAG. Aunque es posible integrar componentes RAG en Rasa, esta integración requiere un mayor esfuerzo de desarrollo e implica una curva de aprendizaje más grande, especialmente en comparación con *frameworks* diseñados desde el inicio para trabajar con LLMs, como LangChain.

A diferencia de LangChain, Rasa es una tecnología mucho más madura y estable. Cuenta con una comunidad consolidada y una documentación muy extensa y depurada¹⁹, lo que facilita su adopción y reduce la aparición de problemas inesperados [36].

3.4.3 Elección del framework de desarrollo

La elección del *framework* de desarrollo condiciona directamente la arquitectura y la implementación de la aplicación. En este proyecto, se ha optado por utilizar LangChain debido a su carácter modular y a su compatibilidad nativa con múltiples proveedores de LLMs, herramientas de almacenamiento vectorial y mecanismos de orquestación de procesos. Su diseño facilita la integración de fuentes de datos externas y permite implementar de forma estructurada todo el flujo de trabajo de un sistema RAG, incluyendo la carga de documentos, la generación y almacenamiento de *embeddings*, la recuperación semántica y la generación final de respuestas.

¹⁹<https://rasa.com/docs/>

Asistente conversacional

En este capítulo se recogen los requisitos necesarios para el desarrollo del *backend* y se establece el contexto técnico previo a su construcción. Después, se describe el diseño general del sistema, detallando las partes principales que lo componen y su funcionamiento. Para la implementación del *chatbot*, se ha llevado a cabo una evaluación continua de cada módulo, con el objetivo de garantizar la calidad de las respuestas, junto con una valoración global del sistema en su conjunto. Por este motivo, en cada sección se explican los parámetros más relevantes y los criterios empleados para seleccionar la mejor alternativa en cada caso. Finalmente, se aplican métricas específicas para evaluar la calidad global del asistente conversacional desarrollado.

Toda la estructura del *backend*, junto con el código fuente y la documentación detallada, está disponible en el repositorio del proyecto¹. En el archivo *README* se describen los distintos archivos y componentes del sistema, facilitando la consulta y comprensión de la arquitectura general.

4.1 REQUISITOS DEL SISTEMA

Antes de abordar el diseño y la implementación del sistema, es necesario establecer con claridad los requisitos que debe cumplir el *chatbot* para asegurar que su desarrollo responda a las necesidades reales de los usuarios y permita orientar correctamente el diseño y la evaluación del proyecto.

El principal objetivo del *chatbot* es ser una herramienta útil para cualquier persona que utilice LAREDO. El sistema debe ser capaz de responder con precisión y claridad a las preguntas sobre la aplicación, combinando la información contenida en los documentos disponibles, documentación en la web y el conocimiento propio del modelo de lenguaje.

Cualquier usuario, tenga o no conocimientos técnicos, debe poder aprovechar el asistente sin dificultad. Para ello el *chatbot* debe entender preguntas formuladas de distintas maneras, y ofrecer siempre respuestas comprensibles y útiles.

El público abarca a todos los usuarios de la aplicación de LAREDO, lo que obliga a diseñar una solución lo más accesible posible, sin barreras técnicas ni requisitos especiales de *hardware* o configuración y además se busca que el acceso sea gratuito para todos los usuarios. A nivel de desarrollo, no existen restricciones particulares respecto a los *frameworks* o herramientas empleados para el *backend*, lo que permite centrarse en la calidad final del producto.

Un aspecto importante para las respuestas del *chatbot* es que solo debe responder a cuestiones relacionadas con la aplicación, evitando temas ajenos que puedan confundir o desviar la atención del usuario. Además, la respuesta debe ser precisa y generarse en tiempo real. Es imprescindible también que el sistema permita incorporar fácilmente nuevos documentos y recursos, ya que la aplicación está en continua evolución y es necesario que el asistente se mantenga siempre actualizado.

El éxito del asistente se evaluará empleando tanto métricas objetivas como evaluaciones humanas, con el fin de medir su utilidad para los usuarios, así como la cobertura y la calidad de la información proporcionada.

¹<https://github.com/Gabiz053/LaredocMind>

4.2 DISEÑO

El *backend* sigue el modelo RAG, que se organiza en tres fases modulares: indexación del corpus documental en una base de datos semántica, recuperación de los fragmentos más relevantes ante cada consulta y generación de la respuesta final apoyándose en los fragmentos. Esta estructura permite actualizar el conocimiento del asistente sin reentrenar el modelo base². Cada fase se conecta siguiendo un diseño basado en grafos de estado facilitando el mantenimiento, las pruebas unitarias y la evolución independiente de cada componente.

4.2.1 Fase de indexación de documentos

En esta fase se construye la base de conocimiento que el *chatbot* utilizará para recuperar información. Incluye: (1) la generación o recopilación de la documentación, (2) el preprocesamiento en formato adecuado, la división en fragmentos (*chunking*) y (3) el almacenamiento de *embeddings* en una base vectorial. En la imagen 4.1 se puede ver una guía visual para cada paso identificando este por un número secuencial.

Fase de indexación

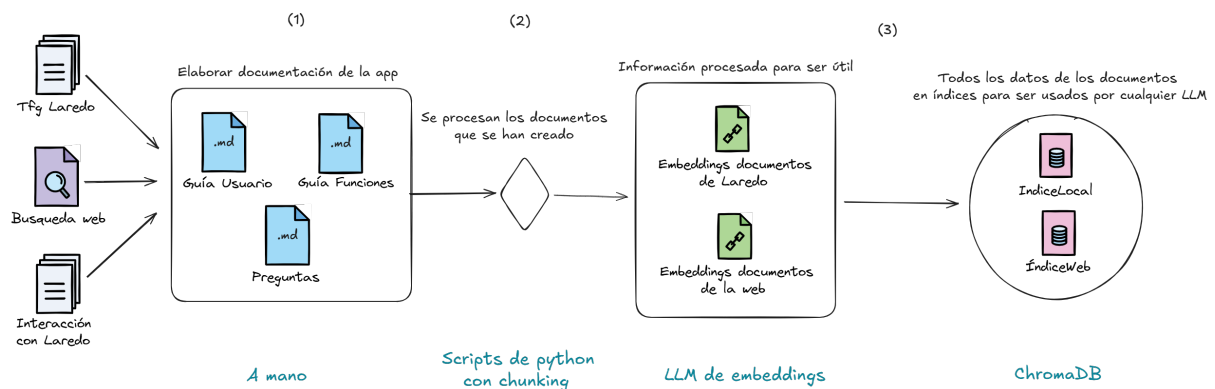


Figura 4.1: Diagrama de la fase de indexación para el sistema RAG

Documentación de la aplicación

Como primer paso, la aplicación LAREDO no disponía de ningún tipo de documentación. Por este motivo, fue necesario crear el manual de la aplicación (ver paso 1 figura). Para ello, se elaboró la guía de usuario, la guía de funciones y un recopilatorio de preguntas frecuentes, basados en la interacción directa con la aplicación y el TFG en el que se desarrolló LAREDO. Toda esta documentación se encuentra en el git del proyecto y además, se escribió en inglés ya que en este idioma se consigue una mayor calidad de respuestas al recuperar información [37].

Para facilitar el procesamiento y posterior indexación, se decidió convertir todos los documentos al formato Markdown, aprovechando sus ventajas para el manejo textual. Este es un formato basado en texto plano con convenciones estructurales (encabezados, listas, código, etc) que permiten preservar la estructura lógica del texto de forma simple sin introducir etiquetas HTML u otro tipo de artefactos sin significado semántico. Esto resulta en una mejor fragmentación del contenido, facilitando la búsqueda y recuperación de unidades de información coherentes y con sentido completo³.

Por otra parte, se recopiló documentación de terceros procedente de fuentes externas, como la documentación oficial de librerías que usa LAREDO como *scikit-learn*. Dado que estas páginas están escritas en HTML, se ha recurrido a técnicas de scraping y limpieza con scripts en Python (empleando librerías como *Requests* y *BeautifulSoup*) para extraer el HTML relevante de las páginas web [38].

²<https://outshift.cisco.com/blog/retrieval-augmented-generation-knowledge-intensive-nlp>

³<https://www.pinecone.io/learn/chunking-strategies/>

Posteriormente, se aplicó la herramienta `markdownify`⁴ para transformar estos documentos HTML a Markdown de forma automatizada, lo que eliminó la mayoría de etiquetas innecesarias y conservó el contenido estructurado. Sólo se descargó la información necesaria para el proyecto, seleccionando la versión específica de la biblioteca utilizada en el desarrollo, ya que la aplicación se apoya en una versión concreta.

El Markdown resultante aún podía contener cierto ruido (menús, pies de página, símbolos, etc.), por lo que se llevó a cabo una limpieza adicional mediante expresiones regulares para eliminar fragmentos no informativos (barras de navegación, anuncios, fragmentos de código irrelevante, enlaces a redes sociales, etc.) y para normalizar caracteres a ASCII. Así, el resultado final son documentos en Markdown limpios, consistentes y preparados para ser segmentados y convertidos en embeddings en la siguiente etapa del pipeline conversacional.

División en fragmentos del contenido

Como segundo paso, dado que los documentos pueden ser extensos y superar con facilidad varios miles de caracteres, es necesario particionarlos en fragmentos manejables antes de generar sus embeddings. Esta técnica, conocida como *chunking*, consiste en dividir cada documento en porciones más pequeñas que conserven un significado autónomo. Hay dos motivos principales para aplicar este proceso: por un lado, las limitaciones de longitud de los modelos de embedding, que sólo permiten procesar un número máximo de *tokens* por fragmento. Por otro lado, la relevancia durante la búsqueda, ya que fragmentos más focalizados ayudan a que el sistema recupere respuestas completas y precisas, en lugar de frases fuera de contexto [39].

Para ello, se utilizó la clase `MarkdownTextSplitter` de `LangChain`, que aprovecha la estructura Markdown para realizar cortes inteligentes: separa el texto en secciones (encabezados), párrafos o bloques de código, evitando dividir frases o listas a la mitad. De esta manera, los fragmentos generados se alinean con los apartados del documento (por ejemplo, un apartado entero o una lista con todos sus elementos), manteniendo el contenido íntegro dentro de cada *chunk*⁵.

Los parámetros más importantes del proceso de fragmentación de texto (*chunking*) son el tamaño máximo de cada fragmento (*chunk size*) y el solapamiento entre fragmentos (*chunk overlap*). El fragmento de código 4.1 muestra estos parámetros:

```
MARKDOWNIFY_CONFIG: Dict[str, Any] = {
    "split": ["dl", "dt", "dd"], # Tags to split the content
    "heading_style": "ATX",      # Heading style (ATX for # headers)
}

MARKDOWN_SPLITTER_CONFIG: Dict[str, Any] = {
    "chunk_size": 1000,          # Maximum size of each chunk
    "chunk_overlap": 200,        # Overlap between chunks to maintain context
    "keep_separator": True,      # Include separators in the resulting chunks
}
```

Código 4.1: Configuración de los parámetros de fragmentación de texto.

El *chunk size* determina la longitud máxima permitida para cada fragmento, normalmente medida en *tokens*. Por ejemplo, un *chunk size* de 1000 *tokens* implica que cada fragmento generado contendrá hasta 1000 *tokens* como máximo.

El *chunk overlap*⁶ define cuántos *tokens* del final de un fragmento se repiten al inicio del siguiente. El objetivo de este solapamiento es evitar la pérdida de contexto en los puntos de corte, asegurando

⁴<https://github.com/matthewwithanm/python-markdownify>

⁵<https://medium.com/@kenzic/getting-started-chunking-strategy-ebd4ab81f745>

⁶<https://chunkviz.up.railway.app/>

que la información relevante situada cerca del límite entre dos fragmentos esté presente en ambos. El valor del solapamiento puede configurarse como un número absoluto de *tokens* (por ejemplo, 100 *tokens*) o como un porcentaje del tamaño del fragmento (por ejemplo, un 10 % de 1000 *tokens* serían 100 *tokens* solapados entre fragmentos consecutivos) [39].

Es importante comprender cómo se aplica este mecanismo en la práctica y sus posibles efectos. Cuando el texto original es menor que el *chunk size* establecido, el resultado es un único fragmento que contiene todo el texto, ya que no es necesario realizar cortes ni aplicar solapamiento. En estos casos, el valor de solapamiento no tiene ningún efecto, puesto que no se generan fragmentos adicionales ni duplicados. Sin embargo, cuando el texto supera el tamaño máximo de fragmento, se generan varios fragmentos, cada uno con la longitud definida y con la cantidad de solapamiento especificada. Por ejemplo, si se emplea un *chunk size* de 1000 y un solapamiento de 100, el primer fragmento abarcará los *tokens* 1 a 1000, el segundo los *tokens* 901 a 1900, el tercero de 1801 a 2800, y así sucesivamente. Así, el contenido cercano a los límites de cada fragmento queda preservado en ambos fragmentos adyacentes, lo que permite que el modelo disponga de suficiente contexto a la hora de recuperar la información relevante.

El ajuste de estos parámetros se ha realizado de forma experimental comparando las respuestas del modelo con diferentes combinaciones, como se explica posteriormente en la sección 4.5.

Almacenamiento de *embeddings*

Como tercer paso, una vez obtenidos los fragmentos de texto, se debe generar un *embedding* vectorial para cada uno y almacenarlo en una base de datos de vectores, con el objeto de facilitar búsquedas semánticas eficientes. Para este proyecto, se optó por ChromaDB⁷, una base de datos de vectores de código abierto, diseñada específicamente para aplicaciones de IA y que ofrece funcionalidades avanzadas de búsqueda por similitud. Además, se integra nativamente con frameworks como LangChain. Esta integración permitió conectar los fragmentos de documento (generados y procesados previamente) directamente al almacenamiento sin necesidad de desarrollar conectores personalizados⁸.

Para gestionar las distintas fuentes de conocimiento, se crearon dos colecciones separadas de *embeddings* en ChromaDB: una para la documentación local (que recoge los manuales y materiales internos creados específicamente para LAREDO) y otra para el contenido web (como la documentación de librerías externas). Esta separación según la procedencia facilita la aplicación de estrategias diferenciadas de recuperación: por ejemplo, priorizar las respuestas de la colección local, considerada de mayor confianza y más específica, o limitar la cantidad de fragmentos web a consultar para evitar información menos relevante o potencialmente ruidosa.

Una vez almacenados los fragmentos de los documentos en la base de datos ya se encuentran disponibles para cualquier consulta que se realice y no hace falta repetir este proceso de nuevo, a menos que se modifiquen los documentos o se quiera añadir nuevos.

4.2.2 Fase de recuperación

Con la base de conocimiento ya indexada, el siguiente paso consiste en recuperar los fragmentos más relevantes para responder a la consulta del usuario. La fase de recuperación abarca desde que el usuario introduce su pregunta hasta que el sistema compone un conjunto de fragmentos de documentos relevantes, que servirán de contexto al modelo generativo. En la figura 4.2 se puede ver una guía visual para cada paso identificando este con un número secuencial.

Preprocesamiento de la consulta del usuario

El usuario puede realizar su pregunta en distintos idiomas, con posibles errores ortográficos, ambigüedades o formulaciones poco claras. Para mejorar la eficacia de la recuperación, se implementó un paso de preprocesamiento de la consulta antes de acceder a las fuentes de información.

⁷<https://www.trychroma.com/>

⁸<https://www.datacamp.com/tutorial/chromadb-tutorial-step-by-step-guide>

Fase de recuperación

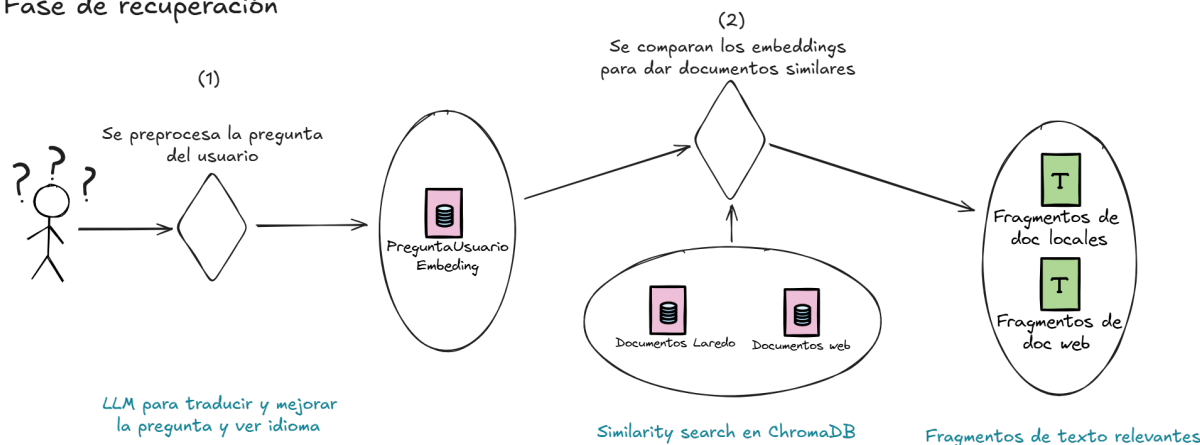


Figura 4.2: Diagrama que recoge los pasos de fase de recuperación para el sistema RAG

Como primer paso, si la pregunta está en un idioma diferente al inglés, el sistema realiza una traducción automática utilizando un modelo LLM ligero para no sobrecargar el sistema. De esta forma, se garantiza que tanto la consulta como los documentos estén en el mismo idioma⁹, lo que favorece la coincidencia semántica entre los *embeddings* [37].

Además, el preprocesamiento realiza correcciones ortográficas y gramaticales en la medida de lo posible. Si la consulta es ambigua, corta o poco clara, se intenta inferir la intención y reformular la pregunta de manera más normalizada. Por ejemplo, una consulta como *¿Qué es random forest?* se transforma automáticamente en *What is the Random Forest algorithm?*, asegurando una redacción clara antes de la búsqueda. Esta traducción se devuelve en formato JSON por el LLM. A su vez se incluye el idioma original de la pregunta para que la respuesta final se realice en ese mismo idioma.

Búsqueda en fuentes locales y web

Como segundo paso, se procede a la búsqueda en la base de datos. Para cada búsqueda, se genera el *embedding* de la consulta del usuario y se calcula la similitud semántica con los fragmentos disponibles en ambas colecciones. Para ello se usan las funciones de similitud proporcionadas por ChromaDB.

Esta doble búsqueda permite recuperar contexto relevante tanto de la documentación específica de la aplicación como de fuentes externas por separado. Un aspecto fundamental es decidir cuántos fragmentos recuperar de cada colección, ya que este parámetro impacta directamente en la calidad de las respuestas: elegir un número insuficiente puede limitar la información disponible, mientras que seleccionar demasiados puede introducir ruido o información irrelevante. Además, los modelos generativos suelen aprovechar sobre todo la información situada al principio y al final del contexto, por lo que un exceso de fragmentos puede resultar contraproducente [40]. La configuración utilizada puede consultarse en el código 4.2.

```
K_WEB_SEARCH = 1 # Default number of web documents to retrieve
K_LOCAL_SEARCH = 3 # Default number of local documents to retrieve
```

Código 4.2: Parámetros para el número de fragmentos recuperados en cada colección.

Por estas razones, se ha optado por partir de un número reducido de fragmentos para evitar sobrecargar el *prompt* (el mensaje enviado al modelo, que incluye la consulta del usuario, los fragmentos

⁹<https://raghunaathan.medium.com/query-translation-for-rag-retrieval-augmented-generation-applications-46d74bff8f07>

seleccionados y las instrucciones para el LLM) con información secundaria. En este sistema, existen dos fuentes principales: la documentación interna de la aplicación y la documentación de sklearn. Dado que se espera que la mayoría de las preguntas de los usuarios estén relacionadas con la propia aplicación, se recuperan más fragmentos de la colección local que de la externa, priorizando así la información más relevante para el usuario final.

Para saber cuántos fragmentos conviene recuperar, es necesario probar el sistema completo con preguntas reales. Así se puede ajustar este parámetro según lo que necesita la aplicación y cómo responde el modelo en la práctica.

4.2.3 Fase de generación

Una vez recuperados los fragmentos relevantes, el sistema pasa a la última fase, donde se produce la respuesta final en lenguaje natural. En la figura 4.3 se puede ver una guía visual con los pasos que conlleva.

Fase de generación

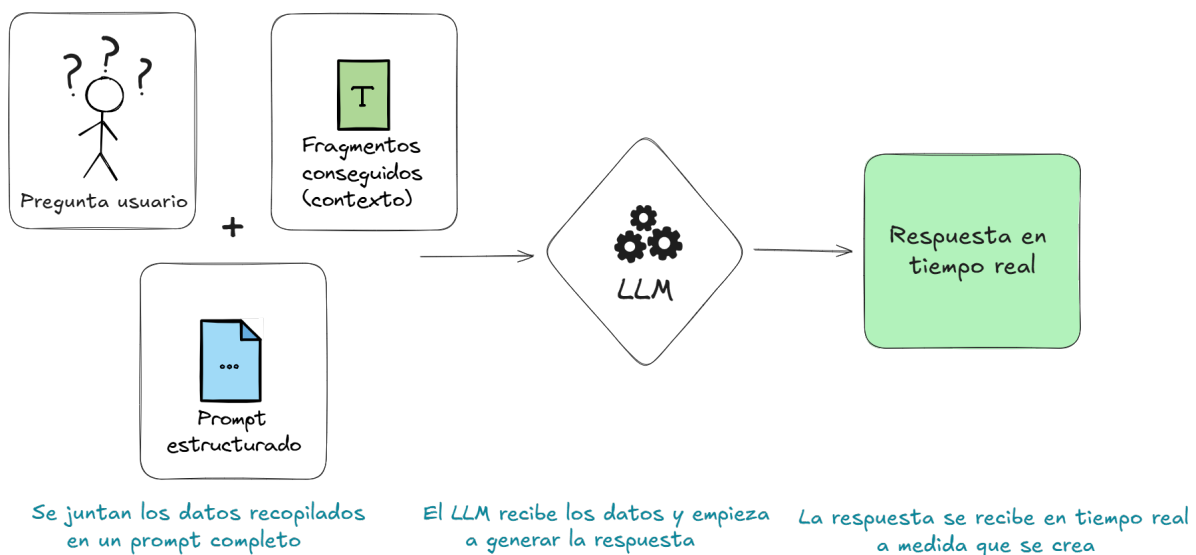


Figura 4.3: Diagrama de la fase de generación para el sistema RAG

En esta etapa se utiliza el LLM, al que se le envía un *prompt* que está formado por tres elementos principales:

- Un mensaje de sistema que define las órdenes de comportamiento del asistente (idioma, tono, uso del contexto, etc.).
- El historial reciente de la conversación entre el usuario y el asistente, lo que aporta continuidad y memoria al diálogo.
- Los fragmentos de contexto seleccionados, presentados de forma estructurada, seguidos de la pregunta actual del usuario.

El diseño detallado del *prompt*, así como sus variantes y consideraciones específicas, se analiza en profundidad en el capítulo 5.

Parámetros del LLM

Para generar las respuestas, los modelos LLM emplean un proceso de decodificación, es decir, la selección progresiva de cada *token* que forma la respuesta final. Los parámetros de decodificación regulan cómo se realiza esta selección, influyendo en el estilo, la coherencia y la diversidad de las salidas generadas.

Entre los más relevantes se encuentran la temperatura, el top-p (*nucleus sampling*) y el top-k, ya que permiten ajustar el equilibrio entre respuestas precisas y respuestas creativas o variadas [41, 42]. Un ejemplo de configuración utilizada puede verse en el código 4.3.

```
LLM_CONFIG: Dict[str, Any] = {  
    "model": "gemini-2.0-flash", # Google Gemini model  
    "temperature": 0.0, # Lower creativity for more factual responses  
    "top_p": 0.9, # Controls response diversity  
    "top_k": 40, # Filters top candidate tokens at each step  
}
```

Código 4.3: Configuración de los parámetros principales para un modelo de lenguaje.

La temperatura controla la aleatoriedad en la elección de las siguientes palabras en la respuesta¹⁰. Valores bajos (próximos a 0) hacen que el modelo elija siempre las opciones más probables, produciendo respuestas más deterministas y fieles a la documentación, lo que resulta adecuado en asistentes que requieren respuestas claras y fiables. Por el contrario, valores altos (0.7 hasta 1.0) incrementan la variedad y creatividad de las respuestas, pero también el riesgo de generar información menos fiable o inventada (alucinación). Por este motivo, en este proyecto se utiliza como inicio una temperatura muy baja (0) [41].

El top-p define un umbral de probabilidad acumulada en lugar de un número fijo de *tokens*. Es decir, el modelo considerará el conjunto más pequeño de *tokens* cuya suma de probabilidades alcanza el umbral definido. Por ejemplo, con top-p = 0.9, se tomará el grupo de palabras que conjuntamente tienen 90 % de probabilidad, ignorando las demás menos probables. Este método se adapta al contexto: si la distribución es muy difusa, tomará más *tokens*; si está muy concentrada en unos pocos, tomará solo esos. En el asistente, se usa inicialmente top-p con valor 0.9. Un valor de 1.0 equivale a no recortar por probabilidad (considerar toda la distribución), mientras que 0.8 sería bastante estricto. Un valor intermedio como 0.9 es una buena elección para mantener algo de variabilidad sin dejar que se recuperen *tokens* muy inverosímiles [42].

El top-k define cuántos *tokens* candidatos considera el modelo en cada paso de generación. Por ejemplo, con $k = 1$, el modelo selecciona siempre el *extittoken* más probable, mientras que valores superiores permiten elegir aleatoriamente entre los k *tokens* con mayor probabilidad, introduciendo cierta diversidad en las respuestas [41]. Sin embargo, top-k es una técnica menos flexible que top-p, ya que fija un número de opciones independientemente de cómo se distribuyan sus probabilidades. Por este motivo, aunque sigue siendo útil para limitar explícitamente el tamaño del espacio de muestreo, en asistentes conversacionales suele fijarse en un valor suficientemente grande (por ejemplo, $k = 40$), priorizando el uso de top-p para adaptar dinámicamente la selección de *tokens* a cada contexto¹¹.

4.2.4 Mantenimiento del historial y resúmenes periódicos

Para que el *chatbot* pueda mantener una conversación coherente, es necesario mantener un historial de mensajes entre usuario y sistema. Este historial permite al asistente recordar el contexto de la conversación, retomar temas previos y dar respuestas que tengan en cuenta la información ya compartida. Sin embargo, si el historial crece de manera indefinida, pueden surgir varios problemas previamente mencionados: el modelo puede superar su capacidad máxima de entrada, la generación de respuestas se vuelve más lenta y se corre el riesgo de que detalles importantes pierdan entre mensajes antiguos [40].

Para gestionar esta situación, se ha implementado una estrategia basada en resúmenes periódicos del historial [43]. El sistema utiliza un contador de mensajes que se incrementa con cada nuevo

¹⁰<https://www.ibm.com/think/topics/llm-temperature>

¹¹<https://www.phdata.io/blog/how-to-tune-llm-parameters-for-top-performance-understanding-temperature-top-k-and-top-p>

mensaje del usuario. Cuando el contador alcanza un umbral determinado se activa automáticamente un proceso que genera un resumen breve de la conversación acumulada hasta ese momento. En este caso se ha escogido como umbral 6 mensajes. Este valor responde a la necesidad de encontrar un equilibrio: esperar más mensajes antes de resumir puede saturar el *prompt* y ralentizar la respuesta, mientras que resumir con demasiada frecuencia podría llevar a perder información relevante.

La generación del resumen se realiza utilizando un LLM que recibe como entrada el historial completo de la conversación hasta ese punto y produce un texto breve y estructurado que recoge los aspectos esenciales tratados. Finalmente, el resumen se guarda como parte del *prompt* y se reinicia el contador de mensajes del usuario. Con esta estrategia conseguimos que el usuario pueda hacer referencia a partes anteriores de la conversación, a la vez que conservamos intactos los mensajes más recientes [43].

4.3 INTEGRACIÓN MODULAR MEDIANTE LANGGRAPH

Tras haber desarrollado cada uno de los componentes fundamentales del asistente conversacional, el siguiente paso consiste en integrarlos de forma ordenada y eficiente. Para lograrlo, se ha utilizado LangGraph, un *framework* que permite organizar la lógica del *backend* como un grafo de estados, en el que cada nodo corresponde a una tarea específica del proceso conversacional.

Esta estructura, representada en la figura 4.4, permite visualizar y comprender fácilmente cómo circula la información y cómo se enlazan las distintas etapas del sistema. Además, el uso de subgrafos (cajas amarillas en el diagrama) facilita la modularización, lo que permite modificar o ampliar cualquier parte del flujo sin afectar al resto.

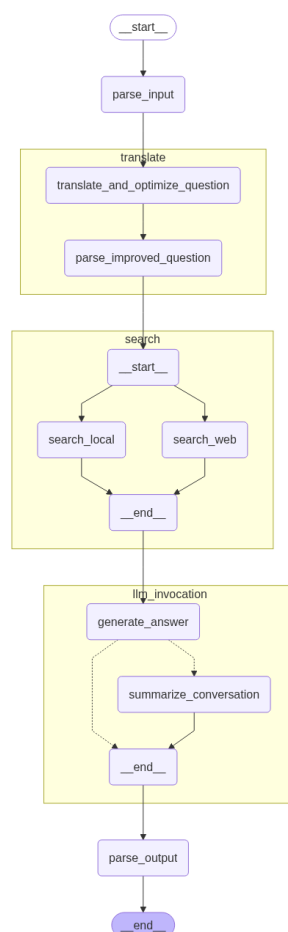


Figura 4.4: Grafo modular del *chatbot* construido con LangGraph.

Nodos del grafo conversacional

Antes de describir el flujo del grafo, es importante entender cómo funcionan los nodos que lo componen. En esta arquitectura, distinguimos dos tipos principales:

- **Nodos funcionales.** Se encargan de tareas específicas y observables, como la traducción de la pregunta, la búsqueda de información relevante o la generación de respuestas a partir del contexto.
- **Nodos de control o decisión.** Regulan el flujo interno del sistema. Aunque no siempre aparecen reflejados como nodos independientes en el grafo, resultan fundamentales para tomar decisiones dinámicas durante la conversación, por ejemplo, activando el resumen automático del historial cuando este supera una longitud determinada.

A continuación, se describe cada paso del grafo, siguiendo el recorrido que hace cada mensaje desde que llega al sistema hasta que se genera la respuesta final:

1. **parse_input.** El sistema recibe la pregunta original del usuario y la transforma en un formato estructurado que sirve de base para las siguientes etapas. Así se garantiza que el flujo siempre parte de datos homogéneos y controlados.
2. **Subgrafo translate.** Aquí el sistema prepara la consulta para maximizar la calidad de las búsquedas posteriores:
 - **translate_and_optimize_question.** Traduce la consulta al inglés y la reformula para que sea más precisa y comprensible.
 - **parse_improved_question.** Extrae y añade al *prompt* la información de la pregunta optimizada, dejando clara la lengua y el contenido de la consulta mejorada.

Este subgrafo puede ampliarse fácilmente en el futuro, por ejemplo, para añadir nuevas técnicas de mejora de consultas.

3. **Subgrafo search.** El asistente recupera información relevante de distintas fuentes:
 - **search_local.** Busca en la base de datos local para extraer los fragmentos más relevantes.
 - **search_web.** De forma paralela, busca en la base de datos web para complementar la respuesta con información externa y actualizada.

Para acabar, los resultados se añaden al *prompt* que recibirá el LLM.

4. **Subgrafo llm_invocation.** Este bloque se encarga tanto de generar la respuesta como de gestionar el historial:
 - **generate_answer.** Construye el *prompt* final a partir de la consulta del usuario, los fragmentos de contexto seleccionados y, si está disponible, un resumen previo de la conversación. En una implementación sencilla, el sistema acumularía toda la respuesta del modelo y la mostraría únicamente cuando está completa. Sin embargo, en este proyecto se optó por realizar la respuesta en *streaming*, de modo que la respuesta se transmite al usuario de forma progresiva a medida que se va generando.
 - **summarize_conversation.** Este nodo funciona como un nodo de decisión, ya que solo se activa cuando el historial de mensajes supera el umbral definido. Su objetivo es generar el resumen de los mensajes anteriores y añadirlo al *prompt* para las siguientes preguntas.
5. **parse_output.** Finalmente, la respuesta generada se formatea para ser presentada al usuario, dejando preparado el sistema para la siguiente pregunta.

4.4 EJECUCIÓN

Para poner en funcionamiento el *chatbot* se ha implementado un *script* principal que automatiza el proceso de inicialización y permite interactuar con el sistema directamente desde la consola. Este *script* garantiza que todos los recursos necesarios se encuentren correctamente configurados antes de comenzar la conversación.

En primer lugar, el *script* verifica que se hayan establecido las claves de acceso para los servicios externos, como los modelos de lenguaje de Gemini, mediante variables de entorno. También gestiona la instalación automática de las dependencias requeridas y la inicialización de la base de datos con los documentos y los *embeddings* correspondientes.

Una vez completada la configuración, el sistema inicializa el grafo conversacional del *chatbot* y queda listo para su uso. El usuario puede entonces introducir preguntas directamente desde la consola y recibir respuestas en tiempo real, manteniéndose el historial de la conversación durante toda la sesión. Para finalizar la interacción, basta con escribir el comando `exit` (o `salir`), y el sistema se cierra de forma segura y controlada. De este modo, el asistente puede utilizarse directamente desde la línea de comandos, sin necesidad de una interfaz gráfica ni configuraciones manuales adicionales.

En la figura 4.5 se muestra una captura de pantalla de la consola durante una sesión de prueba, donde se aprecia el flujo de preguntas y respuestas generadas por el sistema.

```
You: que es la regresion?
Bot: La regresión es un tipo de aprendizaje supervisado donde un modelo aprende a predecir un valor numérico en lugar de una categoría. A diferencia de la clasificación, donde el objetivo es asignar etiquetas como "gato" o "perro", la regresión tiene como objetivo predecir valores continuos, como el precio de una casa, la temperatura de mañana o el ingreso mensual de una persona.

El modelo de regresión aprende patrones de datos pasados y los utiliza para hacer predicciones sobre nuevos datos. Esto se hace ajustando una función matemática que mejor representa la relación entre las variables de entrada (factores que influyen en la predicción) y la variable de salida (el valor que se va a predecir).

Por ejemplo, si queremos predecir el precio de una casa en función de su tamaño, el modelo de regresión identificará cómo el tamaño influye en el precio y creará una ecuación que nos permita hacer predicciones para nuevas casas.

You: que es la clasificacion?
Bot: La clasificación es una tarea de aprendizaje automático supervisado en la que un modelo aprende a asignar elementos a diferentes categorías o clases en función de sus características.

Para ello, el modelo se entrena con un conjunto de datos etiquetado, es decir, ejemplos con sus respectivas categorías. Una vez entrenado, el modelo puede predecir la clase de una nueva muestra basándose en los patrones aprendidos.

La clasificación se aplica en múltiples áreas, tales como:

* Detección de spam (spam/no spam).
* Reconocimiento de imágenes (identificación de objetos o caras).
* Diagnóstico médico (clasificación de enfermedades en función de los síntomas).
You: exit
```

Figura 4.5: Ejemplo de interacción con el asistente conversacional desde la terminal.

4.5 EVALUACIÓN

La evaluación es una parte fundamental en cualquier sistema ya que permite no solo comprobar si el sistema funciona correctamente, sino también entender en detalle cómo se comporta ante distintos escenarios y cómo se pueden mejorar sus resultados. En este caso, la evaluación se va a enfocar en ajustar los parámetros para conseguir que el *chatbot* responda de una forma precisa y relevante para el usuario. Para abordar esta tarea, se ha planteado un proceso de evaluación iterativo durante el desarrollo del *chatbot* y apoyado en diferentes herramientas.

En primer lugar, se han utilizado mecanismos de monitorización interna, como un servicio de *log* que registra en tiempo real cada una de las etapas por las que pasa la consulta del usuario, así como la plataforma LangSmith, que permite visualizar de forma gráfica y detallada cómo se procesa

internamente cada pregunta dentro del flujo conversacional. Esto permite comprender cómo el sistema selecciona y utiliza la información para generar sus respuestas.

En segundo lugar, se han aplicado métricas automáticas que permiten medir objetivamente la calidad de las respuestas generadas. Estas métricas son esenciales para comparar distintas configuraciones del modelo y decidir, con datos, cuáles son los valores óptimos para parámetros como la longitud del historial, el número de fragmentos recuperados o las opciones de generación del modelo de lenguaje.

Por último, y dado que la utilidad real de un *chatbot* depende en gran parte de la percepción del usuario, se ha incluido una fase de evaluación humana. En ella, personas reales interactúan con el asistente y valoran la calidad, claridad y utilidad de las respuestas recibidas en situaciones reales.

4.5.1 Evaluación interna y trazabilidad

En este proyecto, la principal herramienta empleada para este fin ha sido LangSmith¹², una plataforma parte de Langchain diseñada para registrar y visualizar el recorrido de cada consulta a través de los diferentes nodos del flujo conversacional. Con ella es posible inspeccionar el estado de cada paso, los tiempos de ejecución, las entradas y salidas generadas y cualquier posible error.

En la figura 4.6 se observa una ejecución real registrada en LangSmith. En este caso, se muestra todo el flujo de procesamiento correspondiente a la consulta “¿qué es la regresión?”. A la izquierda, el gráfico de tipo cascada refleja cada uno de los nodos del grafo conversacional, mostrando tanto los pasos principales (traducción, búsqueda local y web, generación de respuesta) como los subprocessos internos, junto con el tiempo empleado en cada fase. A la derecha, se visualizan la entrada recibida, la pregunta traducida al inglés, la salida generada por el modelo y la respuesta final entregada al usuario.

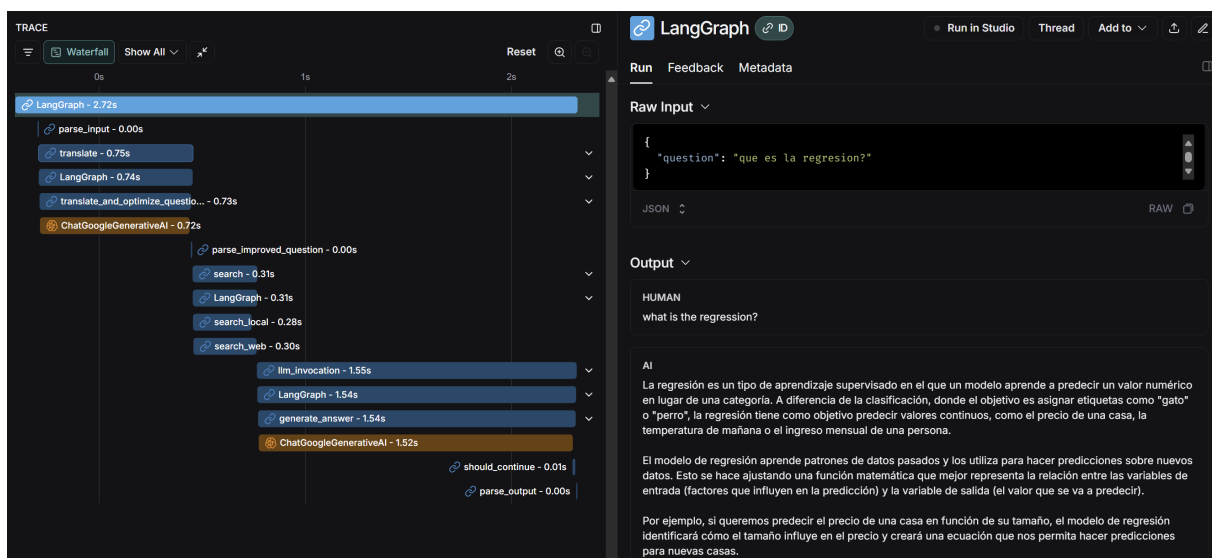


Figura 4.6: Ejemplo de ejecución en Langsmith

Por otro lado, se implementó un *logger* en consola, encargado de informar en tiempo real sobre el avance y el estado del sistema durante el procesamiento de cada consulta. El *logger* describe, paso a paso, cómo se inicia el sistema, qué módulos se van activando, cuándo se realiza cada operación (como la recuperación de contexto o la generación de la respuesta) y si se producen errores en algún punto de la ejecución.

¹²<https://www.langchain.com/langsmith>

En la figura 4.7 se muestra un ejemplo de ejecución del *logger*.

```
2025-06-08 17:19:57.095 | INFO | key_manager:__init__:29 - Initializing KeyManager...
2025-06-08 17:19:57.095 | INFO | key_manager:_load_keys:38 - Loading API keys from .env file...
2025-06-08 17:19:57.096 | INFO | key_manager:_load_keys:46 - API keys successfully loaded.
2025-06-08 17:19:57.096 | INFO | gemini_model_manager:__init__:26 - Initializing ModelManager...
2025-06-08 17:19:57.097 | INFO | gemini_model_manager:_initialize_llm:50 - Initializing LLM model (Gemini)...
```

Figura 4.7: Ejemplo de ejecución del *logger*

4.5.2 Evaluación automática

El primer paso para optimizar el comportamiento del *chatbot* consiste en realizar una evaluación automática que permita comparar de forma objetiva distintas configuraciones y parámetros del sistema. El objetivo de esta fase es identificar qué ajustes contribuyen a que el asistente genere respuestas más precisas, relevantes y coherentes, basándose en indicadores cuantitativos y replicables.

Para ello, se ha construido un conjunto de referencia (*gold set*), compuesto por un conjunto de preguntas reales y sus correspondientes respuestas ideales, validadas previamente. Este conjunto se utiliza como base para evaluar el rendimiento del *chatbot*: en cada experimento, se le plantea la misma batería de preguntas y se analizan sus respuestas comparándolas con las soluciones de referencia [44].

Métricas de evaluación

Para analizar la calidad de las respuestas generadas, se emplean diferentes métricas automáticas, cada una con su propio enfoque. Las principales utilizadas en este trabajo son:

ROUGE (*Recall-oriented understudy for gisting evaluation*) [45]: mide la coincidencia de subsecuencias de palabras entre la respuesta generada y la referencia. En particular, ROUGE-L se centra en la longitud de la subsecuencia común más larga (*LCS*):

$$\text{ROUGE-L} = \frac{\text{Longitud de la subsecuencia común más larga (LCS)}}{\text{Longitud de la respuesta de referencia}}$$

BLEU (*Bilingual evaluation understudy*) [46]: evalúa la precisión de *n*-gramas, es decir, fragmentos de *n* palabras, en la respuesta generada respecto a la referencia. Su fórmula general es:

$$\text{BLEU} = \text{BP} \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

donde:

- *BP* es el *brevity penalty*, un factor que penaliza respuestas demasiado cortas respecto a la referencia.
- w_n es el peso asignado a los *n*-gramas de longitud *n* (por ejemplo, unigramas, bigramas, etc.).
- p_n es la precisión de los *n*-gramas de orden *n*, es decir, el porcentaje de *n*-gramas generados que aparecen también en la respuesta ideal.

Las métricas como ROUGE y BLEU permiten una comparación cuantitativa y replicable, pero a menudo no capturan matices semánticos importantes en las respuestas. Por ejemplo, dos respuestas pueden transmitir el mismo significado con palabras distintas, o aportar información relevante de manera diferente. Para abordar estos casos, se recurre a un enfoque complementario basado en el uso de modelos de lenguaje como jueces (*LLM-as-a-judge*) [47].

Evaluación semántica con LLM-as-a-judge

Este método consiste en emplear un LLM para valorar la calidad de las respuestas generadas, funcionando como un evaluador imparcial [47]. Esto resulta especialmente útil en métricas donde la

valoración es más subjetiva o dependiente del contexto, como la relevancia del contexto o la fidelidad de la respuesta. En vez de aplicar una fórmula estricta, el modelo analiza la pregunta, el contexto recuperado y la respuesta del *chatbot*, y asigna una puntuación siguiendo unos criterios definidos (por ejemplo, en una escala del 1 al 5). Esta estrategia permite capturar aspectos más alineados con la comprensión humana y la utilidad práctica, aunque introduce un componente subjetivo que depende de la interpretación del propio modelo [48].

- Relevancia del Contexto (CR): Esta métrica evalúa hasta qué punto los fragmentos de contexto recuperados por el sistema son pertinentes y útiles para responder a la pregunta del usuario. Una puntuación alta indica que el sistema ha seleccionado fragmentos de contexto relevantes.
- Fidelidad (F): Esta métrica mide si la respuesta generada por el *chatbot* se apoya de manera fiel al contenido proporcionado en el contexto y no se inventa datos. Una puntuación alta refleja que la respuesta es coherente con el contexto disponible y no aporta afirmaciones que no estén justificadas.
- Relevancia de la Respuesta (AR): Evalúa en qué medida la respuesta del *chatbot* es pertinente, clara y completa respecto a la pregunta planteada. Una puntuación alta se asigna cuando la respuesta responde al objetivo de la consulta y es útil desde el punto de vista práctico.

Optimización de parámetros

Una vez definidos los criterios y métricas de evaluación, se procedió a analizar los distintos parámetros a tener en cuenta sobre la calidad de las respuestas generadas por el *chatbot*. En concreto, se evaluaron los tres parámetros principales de los que ya se ha hablado anteriormente: la temperatura, el valor de top-p y la estrategia de *chunking* de documentos. Para cada uno de ellos, se realizaron experimentos independientes en los que solo se modificaba el parámetro en cuestión, manteniendo constantes los demás y aplicando el mismo conjunto de métricas descrito previamente. Se han evaluado valores cercanos a los propuestos inicialmente ya que son los recomendados para un *chatbot* de este estilo.

Cabe destacar que la optimización del *prompt* principal, dada su importancia y naturaleza específica, se aborda de forma separada en el capítulo 5.

Temperatura

Aunque la tabla 4.1 muestra pequeñas irregularidades, como un valor más alto de BLEU en 0,08 o de AR en 0,20, el análisis global revela que la temperatura 0,10 proporciona el mejor equilibrio entre todas las métricas relevantes. Otros valores pueden superar de forma aislada alguna métrica concreta, pero sólo 0,10 mantiene un rendimiento elevado y constante en precisión (ROUGE-L), fidelidad (F) y relevancia de contexto (CR), sin sacrificar coherencia general.

Temperatura	ROUGE-L	BLEU	CR	F	AR
0,00	0,39	0,32	4,0	3,9	4,1
0,05	0,42	0,34	4,0	4,4	3,9
0,08	0,43	0,38	4,3	4,6	4,2
0,10	0,46	0,37	4,4	4,7	4,5
0,20	0,44	0,35	4,3	4,5	4,6
0,40	0,38	0,32	4,1	4,2	4,0
0,60	0,39	0,28	3,5	3,6	3,6

Tabla 4.1: Resultados de las métricas de evaluación para distintos valores de temperatura.

Top-p

Al igual que en la temperatura, en la tabla 4.2 se aprecia que los resultados no siguen una tendencia estrictamente lineal. Por ejemplo, hay casos puntuales en los que alguna métrica concreta, como BLEU en 0.82 o AR en 0.90, presenta valores ligeramente superiores al óptimo seleccionado. Sin embargo, el valor 0.85 destaca por ofrecer el mejor equilibrio general.

Top-p	ROUGE-L	BLEU	CR	F	AR
0,70	0,39	0,32	3,9	4,0	4,1
0,78	0,43	0,34	4,0	4,3	4,0
0,82	0,44	0,37	4,3	4,4	4,3
0,85	0,46	0,36	4,4	4,7	4,5
0,90	0,45	0,36	4,3	4,6	4,6
0,95	0,41	0,35	4,1	4,2	4,3
1,00	0,39	0,31	3,8	3,9	4,0

Tabla 4.2: Resultados de las métricas de evaluación para distintos valores de top-p.

Chunking de documentos

En la tabla 4.3 se observa que el tamaño de *chunk* 1000 *tokens* ofrece el mejor rendimiento global, maximizando ROUGE-L, CR y F. Sin embargo, para *chunk* 1200 el BLEU y AR son ligeramente superiores. Al igual que en las pruebas anteriores, ciertas métricas pueden verse favorecidas por la aleatoriedad o el tipo de consulta. A pesar de estas oscilaciones, *chunk* 1000 mantiene valores altos y estables en todas las métricas principales, lo que lo convierte en la opción más robusta y equilibrada para la configuración final.

Tamaño de chunk	ROUGE-L	BLEU	CR	F	AR
200	0,35	0,28	3,8	3,8	3,7
350	0,41	0,32	4,1	4,0	4,0
500	0,43	0,34	4,0	4,3	4,2
700	0,46	0,36	4,2	4,5	4,2
1000	0,47	0,37	4,5	4,8	4,6
1200	0,44	0,38	4,4	4,7	4,7
1500	0,38	0,34	4,2	4,5	4,5

Tabla 4.3: Resultados de las métricas de evaluación para distintos tamaños de chunk.

4.5.3 Evaluación humana

Aunque las métricas automáticas aportan información objetiva, la evaluación humana es imprescindible para valorar la calidad y utilidad real de las respuestas del *chatbot*, ya que muchos aspectos dependen de la percepción y expectativas del usuario final.

Para este propósito, se empleó un formulario online¹³ en el que los participantes interactuaban libremente con dos versiones del *chatbot*: una configuración “literal” (temperatura baja, T=0) y otra “creativa” (temperatura alta, T=1). En cada caso, se pedía al usuario que formulase una pregunta realista y, tras recibir la respuesta, la evaluase en distintas dimensiones mediante escalas numéricas (1 a 5) y comentarios abiertos.

¹³<https://forms.gle/R1FGrMkcxEhwn5e9>

En la prueba participaron dos personas, y cada una realizó veinte preguntas distintas a ambas versiones del *chatbot*. Esto permitió recopilar un total de 40 evaluaciones para cada configuración, obteniendo así una visión más amplia del comportamiento del sistema en diferentes situaciones.

El formulario utilizado para cada pregunta, que se encuentra en la figura del anexo A.1 incluía:

- Pregunta que se le ha hecho al *chatbot* y respuesta conseguida.
- Calificación de relevancia (si la respuesta ayudaba a resolver la duda planteada).
- Calificación de calidad y fluidez (si la respuesta era comprensible y estaba bien redactada).
- Espacio para comentarios libres sobre aciertos, errores o sugerencias de mejora.
- Opcionalmente, indicadores sobre problemas detectados (respuestas vacías, alucinaciones, etc.).

La tabla 4.4 resume los resultados promedio de las calificaciones humanas para cada versión evaluada. Como se observa, la versión creativa supera a la literal en calidad y fluidez, ya que sus respuestas tienden a ser más extensas, amables y fáciles de leer, aportando matices o detalles adicionales valorados por algunos usuarios. Sin embargo, la versión literal es percibida como más útil y relevante en ciertas preguntas que requieren precisión o donde se prefiere una respuesta breve y fiel al contexto original.

Dimensión evaluada	Literal (T=0)	Creativo (T=1)
Relevancia	3,8	3,7
Calidad	4,1	4,5

Tabla 4.4: Puntuaciones medias de la evaluación humana para las versiones literal y creativa del *chatbot*.

El análisis de los comentarios libres permitió identificar aciertos, errores recurrentes y posibles áreas de mejora en el comportamiento del asistente. Además, se registraron algunos problemas puntuales, como respuestas vacías o alucinaciones, lo que resulta útil para ajustar la configuración del modelo y priorizar futuras mejoras.

Este proceso ha servido, sobre todo, para perfeccionar el diseño del *prompt* y optimizar la estrategia de recuperación de documentos, ya que en la mayoría de los casos en los que el asistente no conseguía dar una respuesta adecuada era debido a que el modelo consideraba la consulta demasiado alejada del tema principal, o bien a que no se había recuperado la información relevante necesaria para construir una respuesta completa.

Estos resultados confirman que el ajuste óptimo del asistente no depende únicamente de maximizar métricas automáticas, sino de encontrar un equilibrio que responda a las expectativas y preferencias reales de los usuarios. La evaluación humana demuestra que la calidez y claridad aportadas por la creatividad pueden mejorar notablemente la experiencia, aunque un exceso puede afectar a la confianza. Por ello, se opta por emplear una temperatura baja, pero superior a la que sugieren solo las métricas automáticas ($T=0.2$), integrando así parte de la riqueza expresiva valorada por los usuarios sin perder precisión ni fiabilidad.

Diseño de *prompts*

El diseño de *prompts* es uno de los factores más importantes en el comportamiento de los sistemas basados en LLM. La capacidad de un modelo para ofrecer respuestas útiles y precisas depende en gran medida de cómo se formula el *prompt* que recibe como entrada. Por este motivo, el diseño y la optimización de los *prompts* constituyen un área crítica en el desarrollo de asistentes conversacionales y sistemas RAG. De hecho, el *prompt engineering* ha demostrado ser capaz de potenciar significativamente el rendimiento de los LLM sin necesidad de re-entrenamiento o ajuste fino de los modelos [49].

La importancia de un diseño cuidadoso también radica en la sensibilidad de los LLM a pequeños cambios en la entrada. Se ha observado que incluso modificaciones mínimas como agregar un espacio extra al final de la instrucción puede alterar la respuesta del modelo [50].

A lo largo de este capítulo se explica, en primer lugar, el proceso de elaboración y ajuste de un *prompt* efectivo, basándose tanto en recomendaciones como en la experiencia práctica. En segundo lugar, se describe cómo se ha implementado el diseño de *prompts* en el *chatbot*. Por último, se presenta una evaluación iterativa del proceso, analizando el impacto de las distintas variantes y ajustes en la calidad de las respuestas.

5.1 BUENAS PRÁCTICAS EN LA REDACCIÓN DE *PROMPTS*

Existen varias recomendaciones para construir un *prompt* efectivo^{1 2 3} :

- Claridad y especificidad. Los modelos responden mejor ante peticiones concretas y bien definidas. Es fundamental evitar instrucciones vagas o ambiguas, así como términos imprecisos que puedan interpretarse de formas distintas.
- Estructura organizada. Un *prompt* debe estar bien organizado, separando claramente las tareas, el contexto y los ejemplos mediante títulos, etiquetas o delimitadores (por ejemplo, <question></question>). Esta organización facilita la interpretación por parte del modelo y reduce errores de comprensión.
- Instrucciones explícitas y paso a paso. Desglosar tareas complejas en subtareas numeradas o instrucciones paso a paso ayuda al modelo a seguir el razonamiento deseado.
- Tono y rol. Es importante definir explícitamente el rol o tono que debe adoptar el modelo. Instrucciones como “Eres un experto en...” ayudan a contextualizar la respuesta y a mantener coherencia en el estilo, ya sea formal, empático, técnico, etc.
- Formato de salida. Es recomendable especificar el formato de la respuesta esperado, especialmente si se requiere una estructura concreta (por ejemplo, lista, JSON, Markdown, etc.). Esto incrementa la utilidad del resultado y facilita su procesamiento posterior.
- Iteración y experimentación. La redacción de *prompts* es un proceso iterativo. Cambios mínimos en el orden, la redacción o la selección de palabras pueden tener un impacto grande en la

¹<https://blog.stackademic.com/openai-new-prompting-guide-how-to-get-the-best-results-from-reasoning-models-354a6adf76c2>

²<https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-design-strategies>

³<https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>

salida del modelo. Por ello, se debe probar diferentes versiones, analizar los resultados y ajustar progresivamente el *prompt* hasta encontrar la mejor formulación [50].

5.2 APLICACIÓN DE LAS BUENAS PRÁCTICAS EN EL DISEÑO DEL *CHATBOT*

A partir de las buenas prácticas, se diseñaron y ajustaron diferentes *prompts* especializados, cada uno con una función que corresponde a una parte del flujo del *chatbot*. El proceso de creación de estos *prompts* fue iterativo y experimental [30].

5.2.1 *Prompt* principal de respuesta

El *prompt* principal de respuesta constituye el núcleo del sistema conversacional, ya que es responsable de generar la respuesta final mostrada al usuario. Su diseño integra de manera estructurada tanto el contexto recuperado por el sistema RAG como la pregunta del usuario y una serie de directrices explícitas orientadas a optimizar el comportamiento del modelo. La configuración detallada puede consultarse en el código 5.1.

```
ANSWER_PROMPT = PromptTemplate(
    input_variables=["context", "question", "language"],
    template="""
You are LaredoMind, an expert chatbot specialized in the LAREDO application.
You have full access to the official documentation and knowledge base of LAREDO.

Guidelines:
- Always format your response using Markdown (headings, bullet points, numbered lists, code
  ↪ blocks, etc. where appropriate).
- Respond in the specified language: {language}.
- Maintain a warm, professional, respectful, and empathetic tone at all times.

User question:
{question}

Instructions:
1. Respond directly to the user's question without mentioning the context, the question itself,
  ↪ or that you are answering.
3. Use your own knowledge only if it is directly and clearly related to the topic.

Relevant context:
{context}
""",
)
```

Código 5.1: *Prompt* utilizado para la generación de respuestas personalizadas por el asistente conversacional.

Características principales del diseño:

- Establecimiento del rol experto. El *prompt* inicia especificando que el asistente es un *chatbot* experto en la aplicación LAREDO y que dispone de acceso completo a la documentación oficial. Este enfoque de *role prompting* ayuda a delimitar el ámbito de conocimiento del modelo y refuerza la confianza y precisión de las respuestas.
- Guía de formato y tono. Se incluyen instrucciones para que la respuesta esté siempre en formato Markdown, permitiendo una presentación clara y estructurada (listas, títulos, bloques de código, etc.). Además, se exige que el asistente responda en el idioma especificado por el usuario ({language}) y mantenga un tono cálido, profesional y empático, incrementando así la accesibilidad y la percepción positiva por parte del usuario.

- Instrucciones operativas precisas. El prompt pide explícitamente que se responda de forma directa a la pregunta, evitando referencias al propio contexto o a la pregunta original, y limita el uso de conocimiento general únicamente a aquellos casos en los que esté claramente relacionado con el tema consultado. Esto contribuye a que las respuestas sean naturales, relevantes y centradas en el usuario, sin divagaciones ni revelación de detalles internos del sistema.
- Integración ordenada del contexto. El contexto recuperado por el sistema RAG se incluye al final del prompt, precedido por la etiqueta "Relevant context:". Esta delimitación clara indica al modelo cuál es la información de referencia, ayudando a centrar la generación y reducir el riesgo de alucinaciones o invenciones.

5.2.2 *Prompt* para historial de conversación

El *prompt* para el historial de conversación se diseñó para permitir al asistente gestionar adecuadamente los mensajes recientes de la conversación y responder de forma coherente y sin repeticiones. La estructura concreta utilizada puede verse en el código 5.2.

```
RECENT_MESSAGES_PROMPT = PromptTemplate(  
    input_variables=["messages"],  
    template="""  
Recent conversation history:  
<messages>{messages}</messages>  
  
Instructions for responding:  
1. If the user asks for clarification, more details, or expresses confusion (e.g., "Can you  
↪ explain more?", "I don't understand", "Please clarify"):  
    - Use the recent conversation history to infer the context.  
    - Refer specifically to the last relevant message if unsure.  
    - Provide a clear and coherent response related to the identified topic.  
  
2. If the user introduces a new topic:  
    - Address the new topic independently without relying on the conversation history.  
  
3. Maintain the flow, tone, and style consistent with the previous conversation.  
""",  
)
```

Código 5.2: *Prompt* diseñado para gestionar el historial de mensajes recientes y mantener la coherencia conversacional.

Características principales del diseño:

- Delimitación clara del historial. Los mensajes recientes se encapsulan dentro de la etiqueta `<messages>{messages}</messages>`. Esta separación explícita ayuda al modelo a distinguir el historial del resto de instrucciones y contenido, facilitando una interpretación correcta del contexto conversacional.
- Instrucciones condicionales precisas. El *prompt* incluye directrices que indican cómo debe utilizarse el historial en función de la consulta del usuario. Si la pregunta hace referencia a un tema anterior, a una aclaración o a una posible confusión, el modelo debe recurrir a los mensajes previos para responder de manera coherente. En cambio, si el usuario plantea una cuestión completamente nueva, el historial debe ignorarse para evitar contaminación de contexto.
- Consistencia de tono y estilo. Se refuerza la necesidad de mantener la continuidad en el tono y la redacción, incluso cuando cambie el tema tratado. Esto mejora la naturalidad y la experiencia de usuario, generando diálogos más fluidos y coherentes.

5.2.3 *Prompt* de resumen conversacional

El *prompt* de resumen conversacional se ha diseñado para abordar las limitaciones de contexto en conversaciones largas, permitiendo al asistente mantener una visión general del diálogo sin necesidad de incluir todo el historial, lo que sería inviable por restricciones de *tokens*. La estructura empleada puede consultarse en el código 5.3.

```
SUMMARY_PROMPT = PromptTemplate(
    input_variables=["summary"],
    template="""
You are assisting in an ongoing conversation.

Conversation summary:
<summary>{summary}</summary>

Instructions:
1. Use the summary to understand the general context if helpful, but it is not mandatory.
2. Respond consistently with the prior conversation flow, maintaining logical continuity
   ↳ without repeating unnecessary details.
3. If the user introduces a new topic, address it clearly and independently, without relying
   ↳ solely on the summary.
4. Prioritize clarity, structure, and progression in your responses.
""",
)
```

Código 5.3: *Prompt* utilizado para la gestión de resúmenes conversacionales en diálogos extensos.

Características principales del diseño:

- Delimitación y encapsulado del resumen. Para introducir el resumen se utiliza la etiqueta `<summary>{summary}</summary>`, utilizando delimitadores tipo XML. Esta estructura distingue claramente el bloque de resumen del resto del contenido, lo que ayuda al modelo a identificarlo correctamente y evitar confusiones con instrucciones u otros datos contextuales.
- Contextualización del rol. El *prompt* sitúa explícitamente al modelo en el rol de asistente en una conversación en curso, reforzando el objetivo de mantener coherencia y continuidad en el flujo del diálogo.
- Instrucciones flexibles y adaptativas. Se indica que el resumen debe emplearse para comprender el contexto general únicamente si es útil, permitiendo que el modelo lo ignore si el usuario introduce un tema completamente nuevo. Esta flexibilidad evita que el resumen interfiera negativamente en situaciones donde podría ser irrelevante o confuso.
- Evitar repeticiones y priorizar la progresión. Se refuerza la necesidad de no repetir información innecesaria ya presente en el resumen y de mantener la lógica y progresión del diálogo. Esto fomenta respuestas más concisas, naturales y alineadas con el desarrollo real de la conversación.

Estos tres *prompts* descritos se combinan en cada interacción para construir el *prompt* final que se envía al modelo de lenguaje principal, asegurando que la respuesta generada tenga en cuenta tanto el contexto, como el historial y las necesidades específicas del usuario en cada momento.

A continuación, se presentan los dos *prompts* auxiliares empleados en el sistema: el *prompt* encargado de generar los resúmenes conversacionales y el *prompt* de traducción de preguntas. Cada uno se usa por un LLM diferente e independiente.

5.2.4 *Prompt* de actualización de resumen

El *prompt* de actualización de resumen se utiliza para mantener actualizado el resumen de la conversación a medida que avanza el diálogo. Es el que usa el LLM condicional cada 6 mensajes. La estructura concreta empleada se muestra en el código 5.4.

```
SUMMARIZATION_PROMPT = PromptTemplate(
    input_variables=["summary", "messages"],
    template="""
You are an expert assistant tasked with maintaining an up-to-date conversation summary for a
↪ language model.

Given:
- The current conversation summary:
<summary>{summary}</summary>

- New incoming messages:
<messages>{messages}</messages>

Update Instructions:
1. Carefully read both the existing summary and the new messages.
2. Identify new important topics, details, or shifts in the conversation.
3. Integrate these new points clearly and logically into the updated summary.
4. Eliminate redundancy; keep the summary concise yet comprehensive.
5. Ensure the updated summary accurately reflects the full conversation progression.
6. Prioritize clarity, structure, and explicitness | optimize for LLM processing over human
↪ naturalness.

Only provide the updated summary text without preambles, explanations, or additional
↪ formatting.
""",
)
```

Código 5.4: *Prompt* utilizado para la actualización periódica del resumen conversacional.

Características principales del diseño:

- Definición explícita del rol. A diferencia de los *prompts* dirigidos a la interacción con el usuario, aquí el modelo adopta el rol de asistente experto cuya tarea es mantener y actualizar de forma autónoma el resumen conversacional. Este enfoque orienta la generación hacia la síntesis y no hacia la respuesta directa a consultas.
- Delimitación clara de bloques. El *prompt* estructura la entrada en dos bloques bien diferenciados: el resumen existente, encapsulado entre `<summary>{summary}</summary>`, y los mensajes recientes, delimitados con `<messages>{messages}</messages>`. Esta organización facilita que el modelo identifique lo que ya ha sido resumido y lo que debe incorporar.
- Instrucciones detalladas y secuenciales. Se incluyen seis instrucciones numeradas que guían el proceso de actualización: leer cuidadosamente ambos bloques, identificar novedades, integrarlas en el resumen de forma clara, eliminar redundancias, asegurar que el resumen resultante recoja toda la conversación (no solo los últimos mensajes) y priorizar la claridad para facilitar el procesamiento por parte de los otros LLM.
- Salida controlada. Se ordena que el modelo devuelva únicamente el nuevo resumen, sin introducciones, explicaciones ni formato adicional. Esto permite reutilizar la salida directamente como entrada para futuras actualizaciones, manteniendo la eficiencia y coherencia del sistema.

5.2.5 Prompt de traducción y normalización de preguntas

Este es el prompt utilizado por el LLM de traducción para detectar el idioma de la pregunta del usuario, traducirla al inglés si es necesario y optimizarla antes de realizar la búsqueda documental. La estructura de este prompt puede consultarse en el código 5.5.

```
TRANSLATE_OPTIMIZE_QUESTION_PROMPT = PromptTemplate(
    input_variables=["question"],
    template="""
You are an expert assistant specializing in technical documentation search optimization.

Your tasks:
1. Detect the language of the user's question. If uncertain, assume English ('en').
2. If the question is not in English, translate it into English directly without interpreting
   ↳ or adding explanations.
6. If 'LAREDO' is mentioned, clarify that it refers to a software application, not a city.
7. Return only a valid JSON object with two fields:
   - 'language': Detected language code (e.g., 'en', 'es', 'ja', etc.).
   - 'question': The translated and optimized English question.

Important rules:
- Translate exactly as written. Do not ask for clarification.
- Do not add any interpretation, examples, or questions.
- Strictly output only the JSON object. No extra text, markdown, or explanations.

User question:
</question>{question}</question>
""",
)
```

Código 5.5: *Prompt* utilizado para la detección de idioma, traducción y normalización de preguntas de usuario antes de la búsqueda documental.

Características principales del diseño:

- Definición explícita del rol. El modelo es presentado como un experto en optimización de búsquedas técnicas. Este enfoque refuerza que el objetivo es preparar la pregunta para maximizar la calidad de la recuperación, y no simplemente traducirla de manera literal.
- Secuencia clara de tareas. Se establecen tres pasos principales: (1) detección del idioma de la pregunta, asumiendo inglés por defecto en caso de ambigüedad; (2) traducción al inglés si es necesario, sin explicaciones adicionales; y (3) desambiguación del término "LAREDO", dejando claro que se refiere a la aplicación de *software* y no a otras entidades homónimas.
- Formato de salida estrictamente controlado. Se exige que la salida sea un objeto JSON con dos campos: `language` (código de idioma detectado) y `question` (pregunta traducida y optimizada). Se prohíbe cualquier explicación, formato extra o texto fuera del JSON, asegurando que la respuesta sea directa y máquina-legible.
- Restricciones y validación experimental. Durante el desarrollo se identificó la tendencia del modelo a añadir explicaciones o mensajes adicionales. Para solucionarlo, se reforzaron las instrucciones y se realizaron iteraciones hasta que el formato de salida fue consistente. La desambiguación de "LAREDO" también se integró tras observar que el sistema devolvía resultados irrelevantes al confundirlo con la ciudad.

5.3 EVOLUCIÓN ITERATIVA Y EVALUACIÓN DE LOS *PROMPTS*

La evolución de los *prompts* se llevó a cabo de forma manual y cualitativa. El proceso consistió en ajustar el *prompt* progresivamente, probando cada versión y comprobando el efecto de cada cambio puntual sobre la calidad de las respuestas. Como se ha mencionado anteriormente, incluso una modificación mínima en la redacción o el orden de las instrucciones puede alterar completamente la salida del modelo, por lo que fue necesario ir puliendo el diseño paso a paso hasta alcanzar el resultado deseado.

Aunque existen métricas automáticas y objetivas que se han utilizado en otras fases del desarrollo, en el caso del diseño de *prompts* no se han empleado. El motivo principal es que lo que más varía en cada iteración es la manera de responder del asistente, un aspecto altamente subjetivo y difícil de cuantificar con métricas automáticas. Aspectos como el tono, la naturalidad, la claridad o la utilidad real de la respuesta sólo pueden valorarse adecuadamente mediante juicio humano y observación directa.

Para obtener resultados comparables y consistentes, la evaluación fue llevada a cabo por las mismas dos personas en ambas fases del proceso. Estas personas analizaron las respuestas generadas por el asistente tanto con el *prompt* inicial como con el *prompt* mejorado, utilizando el mismo formulario empleado en la evaluación humana de respuestas. En este caso se puso especial énfasis en los comentarios cualitativos que acompañaban a cada valoración ya que permitieron identificar de forma precisa los aciertos, carencias y posibles mejoras en el comportamiento del asistente, aportando una visión mucho más rica que la que puede ofrecer una valoración cuantitativa.

Los resultados mostraron una mejora clara en la versión final del *prompt*: el sistema respondía con mayor coherencia, era capaz de adaptarse mejor a diferentes preguntas y las valoraciones, tanto numéricas como descriptivas, reflejaron una experiencia de usuario notablemente más satisfactoria. La tabla 5.1 muestra la comparación de puntuaciones medias obtenidas en ambas versiones.

Dimensión evaluada	<i>Prompt</i> básico	<i>Prompt</i> mejorado
Relevancia	3,8	4,6
Calidad	4,1	4,8

Tabla 5.1: Puntuaciones medias de la evaluación humana para el *chatbot* con el *prompt* básico y con el *prompt* mejorado.

Interfaz del *chatbot*

Una vez conseguido que el asistente conversacional funcione correctamente a nivel de *backend*, el siguiente paso es crear una interfaz para que cualquier usuario pueda interactuar con este de manera sencilla y agradable desde la propia herramienta de LAREDO.

En este capítulo se explica cómo se ha diseñado e implementado la interfaz, qué tecnologías se han utilizado, cómo se ha integrado en la aplicación y cómo se ha evaluado su funcionamiento y diseño.

6.1 TECNOLOGÍAS UTILIZADAS

Para el desarrollo del *frontend* del *chatbot* se seleccionaron las tecnologías que ya formaban parte de LAREDO. De este modo, se garantiza una integración sencilla evitando incompatibilidades y reduciendo el esfuerzo de despliegue y mantenimiento. Las tecnologías elegidas han sido React, Node.js, JavaScript, Tailwind CSS y la técnica de Shadow DOM en React. A continuación, se describe brevemente el papel de cada una:

- React¹: Biblioteca de JavaScript orientada a la construcción de interfaces de usuario interactivas mediante componentes reutilizables y una arquitectura declarativa. React permite desarrollar la interfaz del *chatbot* de manera modular y flexible, facilitando su integración en la plataforma LAREDO. Una de sus principales ventajas es el uso del Virtual DOM, una representación ligera de la estructura de la interfaz que se mantiene en memoria. El Virtual DOM permite que React detecte qué partes de la interfaz han cambiado y actualice únicamente esos elementos en el navegador, en lugar de volver a renderizar toda la página.
- Node.js²: Entorno de ejecución para JavaScript. En este proyecto, Node.js se emplea fundamentalmente para tareas relacionadas con el desarrollo y despliegue del *frontend*, tales como la gestión de dependencias y el empaquetado de la aplicación. El *backend* puede desplegarse en cualquier servidor compatible, manteniendo la independencia entre cada parte de la herramienta.
- JavaScript: Lenguaje de programación empleado para el desarrollo de aplicaciones React y Node.js. Permite implementar la lógica de la interfaz y gestionar la comunicación con el *backend*.
- Tailwind CSS³: Framework de utilidades CSS que facilita el diseño visual de interfaces web de manera rápida y coherente. Sus elementos reutilizables y la capacidad de crear estilos personalizados que no interfieran con los estilos globales de la aplicación, han permitido mantener un diseño alineado con la identidad visual de LAREDO.
- Shadow DOM (con React): Técnica empleada para encapsular el componente del *chatbot* usando la librería *react-shadow*. Esto asegura el aislamiento total de estilos y scripts, evitando interferencias con el CSS o JavaScript global de LAREDO y permitiendo insertar el *chatbot* como widget independiente.

¹<https://es.react.dev/>

²<https://nodejs.org/es>

³<https://tailwindcss.com/>

6.2 DISEÑO DE LA INTERFAZ

La interfaz del *chatbot* se ha diseñado tomando como referencia las últimas tendencias en experiencia de usuario, inspirándose en los sistemas actuales de Android.

En concreto, se ha seguido la línea de diseño propuesta por One UI 7⁴ de Samsung, que destaca por el uso de *glassmorphism*⁵: menús translúcidos, efectos de desenfoque y esquinas suavizadas. Este enfoque busca transmitir ligereza y profundidad visual mediante capas semitransparentes, consiguiendo una apariencia moderna y minimalista. Así, el *glassmorphism* se adopta como base para la interfaz, alineándola con las tendencias actuales en diseño de experiencias de usuario.

No obstante, se ha prestado atención a la integración visual con la plataforma LAREDO, adaptando la paleta cromática y los estilos gráficos propios de la aplicación. De este modo, se consigue que el asistente forme parte de la interfaz de LAREDO, manteniendo coherencia y continuidad en la experiencia del usuario.

A nivel conceptual, el diseño de la interfaz también sigue los principios de la teoría de Gestalt [51], que ayudan a organizar los elementos visuales para que el usuario los perciba como un conjunto coherente. Gracias a esto, la estructura y las funciones del sistema se entienden de forma rápida y natural.

Dado que ciertas características como las animaciones o los efectos de transparencia resultan difíciles de apreciar en capturas de pantalla, se ha incluido en el repositorio del proyecto además del código, un vídeo⁶ donde se muestra la interacción real del asistente dentro de la aplicación LAREDO.

6.2.1 El botón flotante del asistente

Tras definir el diseño general, el primer elemento que se necesita es el botón flotante que da acceso al *chatbot*. Este botón está situado de forma fija en la esquina inferior derecha de la pantalla, lo que garantiza su disponibilidad en todo momento sin interferir con el contenido de la aplicación. Tiene forma circular, un tamaño fácilmente localizable en cualquier dispositivo y utiliza un color azul translúcido para integrarse con los colores de LAREDO. El icono empleado es una burbuja de chat, lo que permite identificar su función de un solo vistazo (véase figura 6.1).

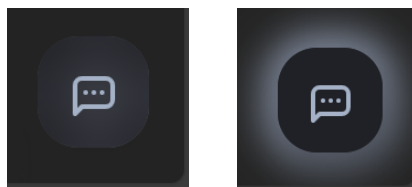


Figura 6.1: Botón flotante del asistente: (a) estado normal y (b) estado al pasar el ratón.

El botón incorpora una animación sutil al interactuar con él. Cuando el usuario pasa el cursor sobre el botón, aparece una sombra azul a su alrededor, lo que refuerza la percepción de interactividad y ayuda a destacar el elemento sobre el fondo. Además, al posar el cursor, se muestra un pequeño mensaje emergente (*tooltip*) con el texto “Abrir chat”. Esto refuerza el significado del icono y facilita el reconocimiento de la funcionalidad, permitiendo que cualquier usuario entienda para qué sirve el botón, incluso si no está familiarizado con la iconografía.

Desde el punto de vista cognitivo, se ha buscado que el botón sea accesible y fácil de entender. Su localización, forma e icono facilitan que el usuario lo identifique rápidamente y sepa para qué sirve, sin añadir distracciones ni complicaciones al flujo de uso.

⁴<https://www.samsung.com/es/one-ui/>

⁵<https://css.glass/>

⁶<https://github.com/Gabiz053/LaredoMind>

6.2.2 Ventana de chat emergente

Al pulsar el botón flotante, se abre una ventana de chat que se superpone a la interfaz de LAREDO. La ventana tiene un diseño moderno y minimalista, con bordes redondeados y una sombra sutil que la separa visualmente del resto de la aplicación. Ocupa una parte moderada de la pantalla, aproximadamente un 40 % de ancho y 50 % de alto, aunque el tamaño puede ser ajustado por el usuario.

El fondo es gris oscuro y semitransparente, inspirado en el estilo *glassmorphism*, lo que aporta profundidad sin distraer del contenido. Las transiciones y animaciones al abrir y cerrar la ventana refuerzan la sensación de fluidez.

Nada más abrirse, la ventana muestra un mensaje de bienvenida en la parte superior del área de conversación, bajo el nombre **LaredocMind**, acrónimo de “LAREDO”, “documents” y “mind”, que refuerza la idea de asistente inteligente y próximo a la documentación de la plataforma. Este mensaje facilita que la conversación comience de forma natural y evita que la ventana se perciba vacía o poco intuitiva (véase figura 6.2).

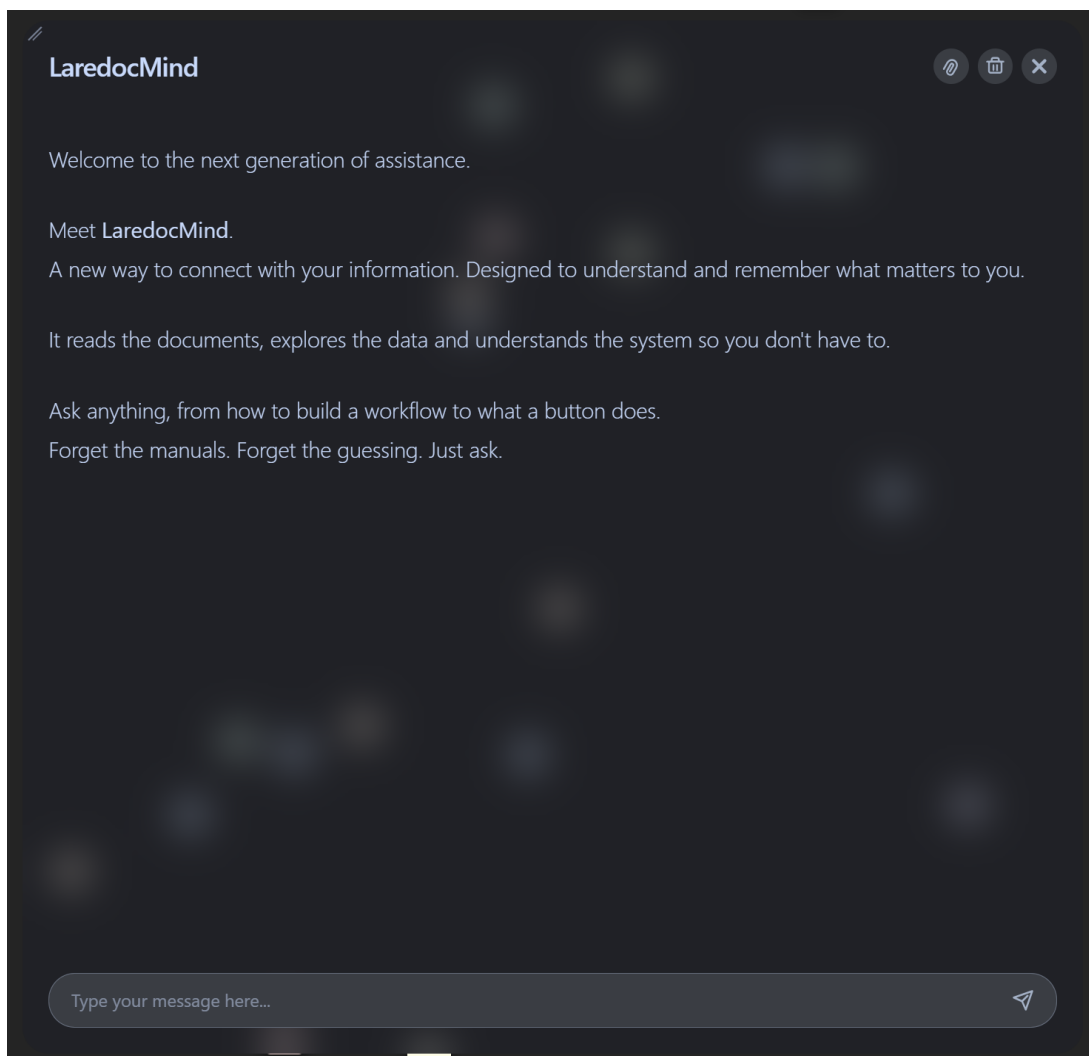


Figura 6.2: Ventana emergente del asistente, con diseño translúcido, bordes redondeados y sombreado sutil.

6.2.3 Estructura de la ventana de chat

Para facilitar la explicación de la interfaz, en la figura 6.3 se muestran las principales zonas que componen la ventana emergente: (1) barra superior, (2) área de conversación y (3) barra de entrada.

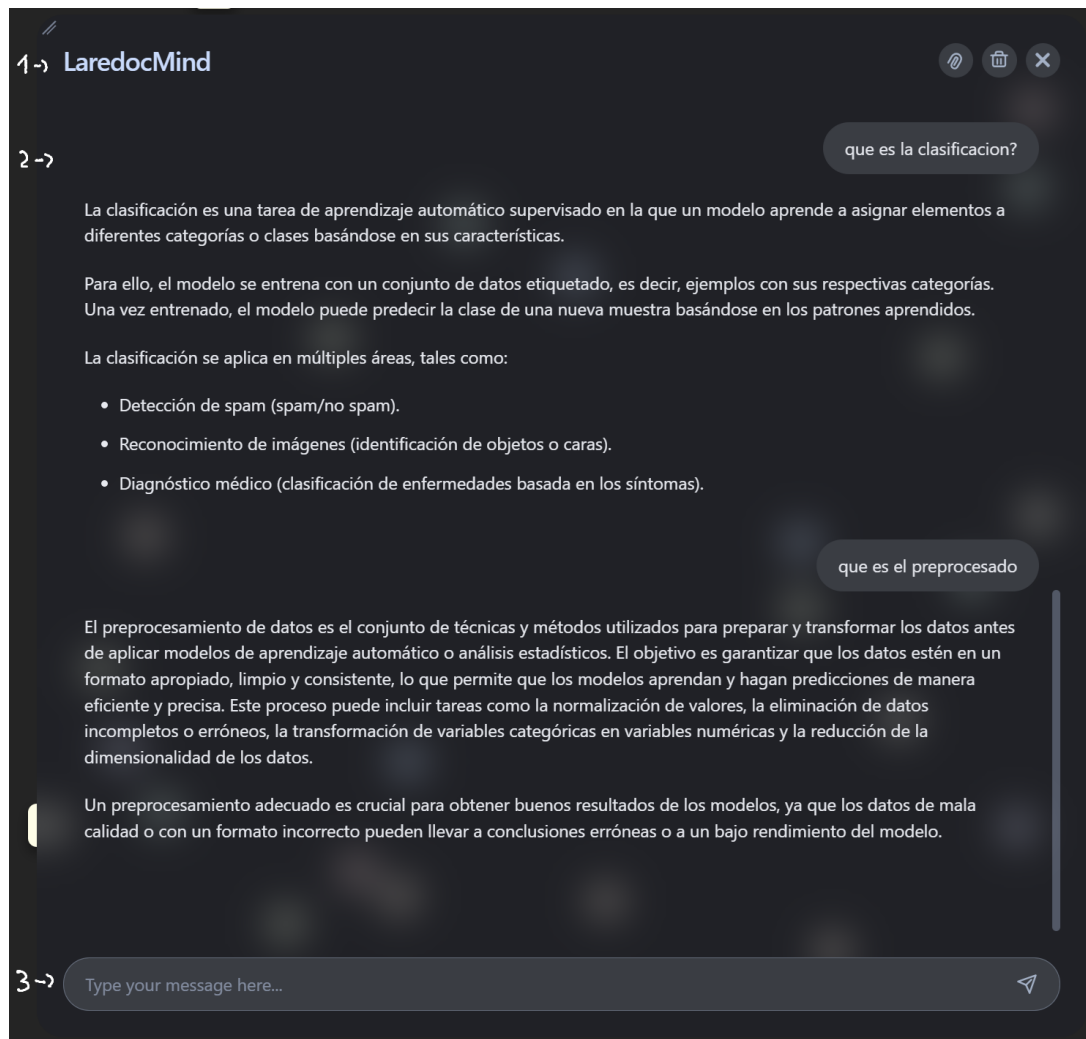


Figura 6.3: Estructura de la ventana de chat: (1) barra superior, (2) área de conversación, (3) barra de entrada.

Barra superior de título y acciones

La barra superior (1 en la figura 6.3) agrupa los elementos de identificación y control del chat. A la izquierda se muestra el nombre del asistente (*LaredocMind*) en una tipografía clara y ligeramente mayor, lo que refuerza su identidad y ayuda a orientar al usuario dentro de la interfaz.

En el lado derecho, la barra incluye tres iconos de acción: fijar la ventana, limpiar la conversación y cerrar el chat. Todos utilizan símbolos universales (clip, papelera y "X") para que el usuario comprenda su función de forma inmediata, y al pasar el cursor aparece un pequeño *tooltip* descriptivo con la función de cada icono. Además, al interactuar con estos iconos, se producen cambios sutiles de color y sombra al igual que con el botón del *chatbot* para ofrecer retroalimentación visual.

Desde el punto de vista cognitivo, la barra permanece siempre visible, incluso al hacer *scroll* en la conversación, garantizando acceso constante a todas las funciones y mejorando la eficiencia y accesibilidad del sistema.

Área de conversación

El área de conversación (2 en la figura 6.3) muestra los mensajes en formato de diálogo, siguiendo el estilo de una aplicación de mensajería. Los mensajes del usuario aparecen como burbujas alineadas a la derecha, con un color de fondo claro, mientras que las respuestas del asistente se integran en

el fondo de la ventana y se alinean a la izquierda, usando tonos gris oscuro y texto blanco para mantener la coherencia visual.

Los mensajes nuevos se presentan con animaciones suaves, como *fade-in*, y la ventana hace *scroll* automático para mantener siempre a la vista el último mensaje. Mientras el asistente está generando una respuesta, se muestra un mensaje de estado (“pensando...”) acompañado de una animación minimalista que indica al usuario que el sistema está procesando la consulta, sin recargar la interfaz.

Las respuestas del asistente se escriben en formato Markdown, lo que permite mostrar listas, títulos o bloques de código de manera clara y estructurada según se van generando. Este formato es habitual en foros y plataformas como GitHub o ChatGPT, por lo gran parte de los potenciales usuarios estarán acostumbrados a leer la información así.

Barra de entrada de mensaje

La barra de entrada (3 en la figura 6.3) se sitúa en la parte inferior de la ventana y ocupa todo el ancho de la ventana. Permite al usuario escribir sus consultas. Está formada por un campo de texto expandible y un botón de envío.

El campo de texto incluye un *placeholder* instructivo (“Escribe tu pregunta aquí...”) en gris suave, que ayuda a identificar su función. El contraste del fondo distingue el área de entrada del resto de la conversación sin romper la armonía visual.

El botón de envío muestra un icono de avión de papel en el color de acento de LAREDO y cambia de estado según el contenido del campo: se muestra deshabilitado y semitransparente cuando está vacío, y se activa cuando hay texto. Además, cuando el usuario comienza a escribir, aparece una animación de sombra azul alrededor del campo de entrada, lo que refuerza visualmente que está activo y aporta dinamismo a la interfaz. El envío del mensaje se puede realizar tanto con el botón como con la tecla Enter, mientras que Shift + Enter permite añadir saltos de línea, igual que en aplicaciones de mensajería habituales.

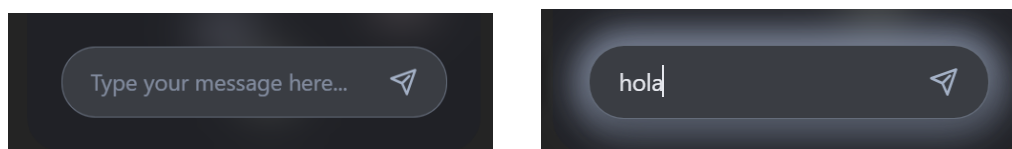


Figura 6.4: Barra de entrada de mensajes: (a) estado vacío con botón deshabilitado y (b) estado activo con texto y animación de sombra.

Desde el punto de vista cognitivo, la barra permanece siempre visible para facilitar el acceso y el envío rápido de mensajes. El diseño minimalista y el alto contraste en el texto mejoran la legibilidad, ayudando al usuario a centrarse en el contenido y evitando distracciones.

6.3 PUBLICACIÓN E INTEGRACIÓN DEL COMPONENTE

Una vez finalizado el diseño y desarrollo de la interfaz, el siguiente paso es incorporar el *chatbot* en LAREDO de la forma más sencilla posible, sin tener que modificar el código base de la aplicación. Para lograrlo, se optó por publicar el *chatbot* como un componente independiente en npm⁷, el repositorio estándar de paquetes para aplicaciones JavaScript y React. De este modo, el asistente puede ser descargado y reutilizado fácilmente en LAREDO o en cualquier otra aplicación compatible.

El proceso de publicación consiste en varios pasos. Primero, se empaqueta el *chatbot* como un módulo reutilizable. A continuación, se sube este paquete a npm, lo que permite que cualquier desarrollador pueda instalarlo simplemente añadiendo la dependencia correspondiente en su proyecto.

⁷<https://docs.npmjs.com/cli/v8/commands/npm-publish>

El componente queda así disponible online, listo para ser integrado en distintas aplicaciones sin complicaciones.

Para asegurar que la integración sea limpia y no haya conflictos de estilos, el *chatbot* utiliza encapsulación mediante Shadow DOM, como se explicó anteriormente. Esto aísla por completo los estilos y la estructura interna del asistente respecto al resto de la aplicación.

Por último, una vez que el paquete está publicado en npm, solo hay que instalarlo para tener el *chatbot* funcionando en cualquier aplicación, ya que no depende de la interfaz de LAREDO. Si en algún momento se quiere actualizar el asistente, basta con cambiar la versión del componente, sin necesidad de tocar el resto de la aplicación.

En la figura 6.5 se muestra cómo se ve la aplicación una vez integrado el *chatbot* en LAREDO.

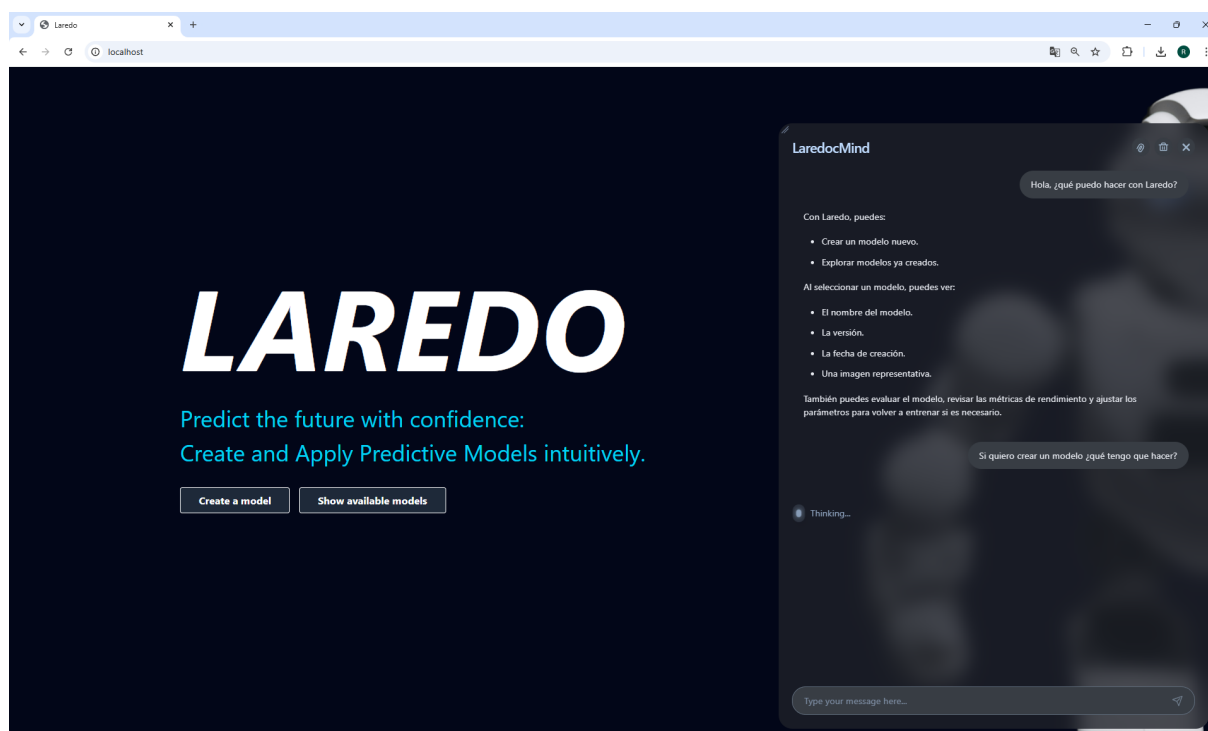


Figura 6.5: Aspecto de la aplicación LAREDO con el *chatbot* integrado.

6.4 EVALUACIÓN

La evaluación de la interfaz del *chatbot* se centró en asegurar que su uso resultara claro, accesible y cómodo en el contexto real de la aplicación. Dado que no existían requisitos externos estrictos ni se contó con un grupo de usuarios finales definido, el proceso de evaluación se basó principalmente en la experiencia propia durante el desarrollo, combinada con observaciones y comentarios tanto por parte de los tutores del TFG como compañeros.

El método principal consistió en pruebas iterativas personales: tras implementar cada funcionalidad, se revisó cuidadosamente su funcionamiento desde el punto de vista del usuario final, identificando posibles mejoras en usabilidad, visualización y comportamiento general. Este proceso permitió ir refinando los aspectos fundamentales de la interfaz, como la disposición de los elementos, la claridad de los iconos y el flujo de interacción.

Además, en diferentes momentos del desarrollo, se mostró la interfaz a varios compañeros para recoger impresiones directas sobre la comprensión de las funciones, la facilidad de uso y la estética general. Estas aportaciones, aunque informales, ayudaron a detectar pequeños detalles que podían generar confusión o dificultar el uso en determinadas situaciones.

A lo largo de varias iteraciones, se realizaron ajustes y pequeñas mejoras en base a la experiencia directa y el *feedback* recibido, hasta alcanzar una versión estable que cumplía con los objetivos definidos al inicio: ofrecer una interfaz moderna, intuitiva y coherente con la aplicación LAREDO.

Comunicación *Backend-Frontend*

Una vez desarrollados tanto el *backend* como el *frontend* del *chatbot*, el siguiente paso es conectarlos para que puedan funcionar juntos y tener el *chatbot* completo.

Este capítulo describe la conexión entre el *backend* y el *frontend* del *chatbot*, el flujo de comunicación entre ambos, los retos técnicos encontrados y las soluciones implementadas. También se detalla el proceso de pruebas y el despliegue final empleando las mismas herramientas LAREDO.

7.1 COMUNICACIÓN ENTRE COMPONENTES

El primer paso es analizar cómo se comunican el *backend* y el *frontend* del *chatbot*. La figura 7.1 muestra un esquema visual que resume el recorrido de la información, desde que el usuario envía una consulta hasta que recibe la respuesta.

Comunicación en la herramienta

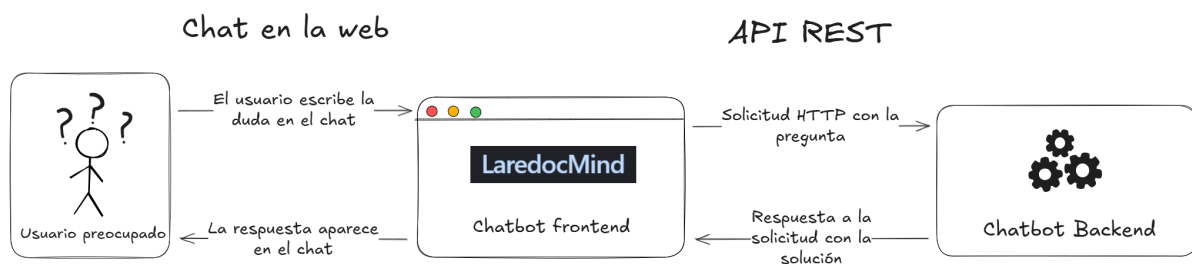


Figura 7.1: Esquema general del flujo de comunicación entre el *frontend* y el *backend* del *chatbot*.

7.1.1 *Backend* y uso de Flask

El *backend* del *chatbot* se encarga de recibir las preguntas de los usuarios y generar una respuesta adecuada. Para facilitar la comunicación entre el *frontend* y el *backend*, se ha utilizado Flask como herramienta principal.

Flask es un framework escrito en Python que permite crear APIs de manera sencilla. Su funcionamiento se basa en definir “endpoints”, es decir, rutas a las que otros programas pueden enviar peticiones y asociarlos a funciones. Cuando el sistema recibe una petición en una de estas rutas, Flask ejecuta la función asociada que procesa la información recibida y devuelve una respuesta.

En el caso de este proyecto, Flask actúa como intermediario exponiendo el *backend* como una API REST que recibe las preguntas enviadas desde la interfaz web, realiza el procesamiento necesario en el *backend* (como invocar al modelo de lenguaje y consultar fuentes de información), y finalmente devuelve la respuesta al usuario.

Endpoints principales del *backend*

El *backend* del *chatbot* define dos endpoints principales para la comunicación con el *frontend* y para tareas de comprobación.

Por un lado, se encuentra el endpoint encargado de procesar las preguntas de los usuarios. Este endpoint está disponible en la ruta `/chatbot/stream` y espera recibir peticiones HTTP de tipo

POST. El *frontend* envía la pregunta en formato JSON, y el *backend* responde transmitiendo la respuesta de forma progresiva, fragmento a fragmento, mediante el protocolo SSE. Así, el usuario puede ver cómo la respuesta se va generando en tiempo real.

Por otro lado, existe un endpoint de prueba en la ruta `/hello`. Este endpoint responde a peticiones HTTP de tipo GET y simplemente devuelve un mensaje fijo en formato JSON que confirma que el *backend* está funcionando correctamente.

A continuación se muestra un ejemplo de cómo utilizar el endpoint desde la terminal. En el código 7.1 se ilustra cómo realizar una petición POST al endpoint de streaming del chatbot empleando la herramienta `curl`.

```
curl -X POST http://localhost:5000/chatbot/stream \
-H "Content-Type: application/json" \
-d '{"question": "¿Qué es la regresión?"}'
```

Código 7.1: Ejemplo de petición POST al endpoint de streaming del chatbot utilizando `curl`.

Un aspecto importante es que para permitir que la interfaz web pueda comunicarse correctamente con el *backend*, aunque ambos se ejecuten en servidores o puertos diferentes, es necesario habilitar el intercambio de recursos entre orígenes distintos (CORS, por sus siglas en inglés). Por defecto, los navegadores bloquean las peticiones realizadas a un servidor externo por motivos de seguridad. Para resolver este problema, el *backend* utiliza la extensión `flask-cors`, que permite configurar Flask para aceptar peticiones desde el *frontend*, aunque provenga de un origen diferente.

7.1.2 Frontend y comunicación con el backend

El *frontend* del *chatbot* es el encargado de recoger la pregunta del usuario y mostrar las respuestas que va generando el *chatbot*.

La comunicación entre el *frontend* y el *backend* se realiza utilizando la función estándar `fetch` de JavaScript, que permite enviar y recibir datos mediante peticiones HTTP. Cuando el usuario introduce una pregunta, el *frontend* construye una petición de tipo POST, incluyendo el texto de la consulta en formato JSON, y la envía al endpoint correspondiente del *backend*.

Tal como se ha descrito en la sección dedicada al *backend*, la respuesta del sistema se transmite en tiempo real: en lugar de recibir todo el contenido de una vez, el *frontend* va recibiendo fragmentos (*chunks*) de la respuesta a medida que el *backend* los genera. Cada fragmento se muestra en pantalla inmediatamente, permitiendo que el usuario lea la respuesta progresivamente, incluso antes de que se haya completado la generación. Una vez finalizada la transmisión, la aplicación está preparada para procesar la siguiente pregunta.

7.1.3 Comunicación en tiempo real mediante SSE

Server-Sent Events (SSE) es una tecnología basada en el protocolo HTTP que permite que un servidor envíe datos al navegador de forma continua, manteniendo una única conexión abierta. Es decir, en lugar de que el navegador tenga que estar preguntando todo el tiempo si hay algo nuevo, con SSE el servidor puede mandar información al navegador en cuanto la tiene disponible.

Esta comunicación es siempre en una sola dirección: del servidor hacia el navegador. Es especialmente útil en aplicaciones donde el usuario necesita ver actualizaciones en tiempo real, como en este *chatbot*, donde la respuesta se va mostrando poco a poco conforme se genera.

Funcionamiento de la comunicación mediante SSE

La comunicación mediante SSE se inicia cuando el *frontend* envía una petición al endpoint específico del *backend* (`/chatbot/stream`). El *backend* responde manteniendo la conexión abierta y enviando fragmentos de la respuesta a medida que se van generando.

Cada fragmento se transmite utilizando el formato estándar de SSE, en el que los datos se envían como mensajes de texto precedidos por el prefijo `data:` y separados por dobles saltos de línea. La comunicación se considera finalizada cuando el navegador recibe un mensaje que termina con un doble salto de línea, lo que indica que ya no se enviarán más fragmentos y la respuesta está completa. El *frontend*, por su parte, escucha estos mensajes y los va mostrando en la interfaz conforme llegan.

Problemas encontrados con el uso de SSE

Las respuestas generadas por el *chatbot* están formateadas en Markdown, un lenguaje de marcado que utiliza dobles saltos de línea (`\n\n`) para separar párrafos o bloques de texto. El problema aparece porque el protocolo SSE también utiliza este patrón de dobles saltos de línea para delimitar cada fragmento de datos enviado al cliente.

Al coincidir ambos usos, puede ocurrir que el *frontend* confunda un salto de línea propio del contenido Markdown con el final real de un fragmento SSE. Esto puede provocar que las respuestas se muestren cortadas o desordenadas, ya que el sistema no es capaz de diferenciar claramente entre ambos casos.

Solución adoptada: delimitador explícito

Para resolver este problema, se ha optado por sustituir el delimitador estándar de SSE (`\n\n`) por un delimitador explícito personalizado que no vaya a aparecer en la respuesta de Markdown, por ejemplo, `<END_OF_CHUNK>`. Ahora, al finalizar cada fragmento de respuesta, el *backend* utiliza este marcador especial en lugar del doble salto de línea habitual.

De esta manera, el *frontend* puede identificar de forma precisa dónde termina cada fragmento de información, sin importar cuántos saltos de línea internos aparezcan en el contenido Markdown. Cada vez que recibe datos, el *frontend* busca el delimitador personalizado y ensambla los fragmentos correctamente, garantizando que la respuesta se muestre de forma completa en cualquier caso.

A continuación se muestra un ejemplo de cómo se transmitía un fragmento de respuesta antes y después de aplicar la solución propuesta. En el código 7.2 se presenta un chunk utilizando el delimitador estándar de SSE, mientras que en el código 7.3 se muestra el uso de un delimitador explícito personalizado.

```
data: #Título de la respuesta.\n\n Esta es la respuesta en Markdown.\n\n Más texto aquí.\n\n
```

Código 7.2: Chunk transmitido utilizando el delimitador estándar de SSE (`\n\n`).

```
data: #Título de la respuesta.\n\n Esta es la respuesta en Markdown.\n\n Más texto  
↪ aquí.<END_OF_CHUNK>
```

Código 7.3: Chunk transmitido utilizando un delimitador explícito personalizado (`<END_OF_CHUNK>`).

7.2 PRUEBAS LOCALES DE LOS COMPONENTES

Antes de integrar todos los componentes en la plataforma final, se realizaron pruebas básicas para asegurarse de que tanto el *backend* como el *frontend* funcionaban correctamente de forma independiente.

El primer paso consistió en probar el *backend* por separado. Para ello, se utilizó la herramienta `curl`¹, enviando peticiones directamente a los endpoints definidos en la API. Esto permitió comprobar que el

¹<https://curl.se/>

backend era capaz de recibir una pregunta, procesarla correctamente y devolver la respuesta esperada en formato adecuado.

Una vez verificado el *backend*, se pasó a probar el *frontend*. En esta fase, el *frontend* se conectó directamente al *backend* ya operativo y se comprobó que las preguntas introducidas a través de la interfaz web llegaban correctamente a la API y que las respuestas se mostraban en pantalla sin problemas. Estas pruebas fueron suficientes para detectar posibles errores de comunicación y de formato como los problemas con SSE.

7.3 AUTOMATIZACIÓN DEL DESPLIEGUE MEDIANTE SCRIPTS

Para que cualquier usuario pueda instalar y ejecutar el sistema de forma sencilla, se ha creado un script general de automatización. Este script está pensado para que, con un solo comando, se realicen todos los pasos necesarios para dejar tanto el *backend* como el *frontend* funcionando en el entorno adecuado, sin necesidad de instalar manualmente cada herramienta o dependencia.

El proceso comienza comprobando que el equipo tenga instalados Python, Node.js y npm, e instala cualquier elemento que falte.

A continuación, se prepara el entorno del *backend*. Para la gestión de dependencias en Python se utiliza Poetry², una herramienta que simplifica la instalación y organización de todas las librerías necesarias. El script instala Poetry si aún no está disponible, crea el entorno virtual correspondiente y ejecuta los comandos necesarios para dejar el *backend* listo para funcionar. Por último, inicia el servidor Flask, que quedará a la espera de recibir peticiones.

En paralelo, el script se encarga de instalar las dependencias del *frontend* utilizando npm. Una vez descargadas todas las librerías, se pone en marcha el servidor de desarrollo, que servirá la interfaz web del *chatbot*.

7.4 DESPLIEGUE FINAL EN LA PLATAFORMA

El último paso, una vez desarrollados y probados todos los componentes del sistema, consiste en integrarlos y desplegarlos de forma conjunta en la plataforma principal.

7.5 HERRAMIENTAS Y PROCESO DE DESPLIEGUE FINAL

Para llevar a cabo el despliegue se han utilizado las mismas herramientas que emplea la plataforma LAREDO, asegurando así la coherencia, facilidad de gestión y escalabilidad del sistema. A continuación, se describen paso a paso las herramientas utilizadas y el orden en el que intervienen en el proceso de despliegue:

7.5.1 Empaquetado y aislamiento de componentes

Docker³ es una herramienta de virtualización que permite empaquetar todo lo necesario para ejecutar una aplicación (como el código, las librerías y las configuraciones) en una unidad estandarizada llamada contenedor. Esto garantiza que la aplicación se ejecuta de forma correcta en cualquier sistema con Docker instalado, independientemente del entorno *software* y *hardware* en el que se encuentre.

Como LAREDO ya utiliza Docker encapsular sus componentes, se ha decidido empaquetar también el *chatbot* usando la misma tecnología. Para ello, se ha actualizado la imagen del *frontend* de LAREDO para incluir el paquete npm del *chatbot* y se ha creado una nueva imagen con el *backend* del *chatbot*. De este modo, tanto el *backend* como el *frontend* del *chatbot* se pueden integrar directamente en el entorno ya existente.

²<https://python-poetry.org/>

³<https://www.docker.com/>

7.5.2 Orquestación y gestión de servicios

Kubernetes⁴ es un sistema de orquestación de contenedores que permite gestionar el despliegue, el escalado, la comunicación y la supervisión de aplicaciones distribuidas. Además, incluye mecanismos para asegurar que los contenedores desplegados se mantienen en funcionamiento, recuperándose ante errores de forma automática. En el caso de LAREDO, cada parte principal del sistema se despliega como un servicio independiente dentro de su propio clúster de Kubernetes. Así, el *backend* del *chatbot* cuenta con su propio despliegue, separado tanto del *backend* principal como del *frontend* de LAREDO, lo que facilita la gestión y el escalado de cada componente por separado.

En cuanto al *frontend* del *chatbot*, se sigue una estrategia ligeramente diferente. Como el componente se ha instalado como un paquete npm, basta con recompilar el *frontend* de LAREDO para que el *chatbot* quede integrado en la interfaz de usuario. De este modo, cualquier actualización futura se puede aplicar fácilmente simplemente actualizando el paquete y recompilando la aplicación principal.

7.5.3 Automatización del despliegue y configuración

Finalmente, la configuración y creación de todos los elementos desplegables de Kubernetes se realiza mediante Helm⁵. Esta herramienta permite definir archivos llamados "charts". Estos archivos contienen plantillas parametrizadas para la creación de los objetos kubernetes necesarios para desplegar todos los componentes de la aplicación. De esta forma, el usuario solamente tiene que rellenar un fichero de configuración `values.yaml` con parámetros como: variables de entorno, rutas, número de réplicas, etc. y ejecutar el comando de instalación del *chart* para poner en funcionamiento el sistema.

⁴<https://kubernetes.io/es/>

⁵<https://helm.sh/>

Conclusiones y trabajos futuros

Como último capítulo, se recogen las principales conclusiones alcanzadas tras el desarrollo del asistente conversacional y su integración en la plataforma LAREDO. También se presentan posibles mejoras y líneas de trabajo futuro que permitirían seguir evolucionando el sistema.

8.1 CONCLUSIONES

El principal objetivo de este trabajo era crear un asistente conversacional que pudiera integrarse de forma sencilla y eficaz dentro de la plataforma LAREDO. Para ello, se han desarrollado tanto el *backend*, encargado de procesar las preguntas y generar las respuestas, como el *frontend*, que muestra la información de manera clara al usuario.

Una lección aprendida con la realización del proyecto ha sido la importancia de analizar y planificar bien cada fase antes de empezar a desarrollar. Dedicar tiempo a entender los requisitos, a estudiar el funcionamiento de la plataforma y a definir bien la arquitectura ha resultado fundamental para evitar problemas posteriores y asegurar que el *chatbot* encaja correctamente en el entorno de LAREDO.

Desde el principio, se ha buscado que el sistema fuera fácil de mantener, escalable y compatible con el resto de los servicios de LAREDO. Por este motivo, se han elegido tecnologías como Flask para la API, React para la interfaz y herramientas como Docker, Kubernetes y Helm para el despliegue. Esta selección ha permitido que el *chatbot* se integre sin problemas en la infraestructura ya existente. También se ha procurado que la instalación del sistema sea fácil usando scripts y buena documentación.

El proyecto me ha supuesto un reto en múltiples dimensiones, ya que nunca antes me había enfrentado a un proyecto de esta envergadura. En la parte del *backend*, he tenido que estudiar y profundizar en técnicas para el procesamiento del lenguaje natural, el uso de modelos de lenguaje y métodos de recuperación aumentada, así como aprender a integrar todas estas tecnologías para conseguir un sistema funcional. Además, me he visto en la necesidad de familiarizarme con nuevas herramientas específicas para el despliegue y la gestión de modelos, lo que ha supuesto un proceso de aprendizaje intenso. En cuanto al *frontend*, partía completamente desde cero, sin experiencia previa en el desarrollo de interfaces web. Esto me ha implicado aprender desde la base tecnologías como React y los principios modernos de diseño orientado al usuario. Más allá del desarrollo de cada parte por separado, también era la primera vez que abordaba la conexión efectiva entre *backend* y *frontend* mediante APIs, así como la modularización de las aplicaciones usando contenedores con Docker.

8.2 TRABAJOS FUTUROS

Aunque el asistente conversacional desarrollado cumple con los objetivos planteados, existen varias actividades interesantes que se pueden abordar en el futuro. Una de ellas es la implementación de un agente que permita al *chatbot* comunicarse directamente con la plataforma LAREDO, aprovechando la integración con los nodos de LangChain. Otra posible acción sería crear una versión local del *chatbot*, para aprovechar tarjetas gráficas potentes, si estuvieran disponibles, y poder usar modelos avanzados sin depender de la nube. Por último, incorporar una funcionalidad de chat de voz permitiría que la interacción con el asistente fuera todavía más natural y cómoda para los usuarios.

Bibliografía

- [1] W. J. Hutchins, *Machine translation: past, present, future*. Ellis Horwood Chichester, 1986 (vid. pág. 3).
- [2] R. Tamrakar y N. Wani, *Design and Development of CHATBOT: A Review*, abr. de 2021 (vid. págs. 3, 7).
- [3] M. Al-Amin et al., *History of generative Artificial Intelligence (AI) chatbots: past, present, and future development*, 2024. arXiv: 2402.05122 [cs.GL]. dirección: <https://arxiv.org/abs/2402.05122> (vid. pág. 3).
- [4] D. Jurafsky y J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*, 3rd. Daniel Jurafsky y James H. Martin, 2025, Online manuscript released January 12, 2025. dirección: <https://web.stanford.edu/~jurafsky/slp3/> (vid. págs. 4, 5).
- [5] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang y W. Zhang, «History, development, and principles of large language models: an introductory survey,» *AI and Ethics*, oct. de 2024, ISSN: 2730-5961. DOI: 10.1007/s43681-024-00583-7. dirección: <https://doi.org/10.1007/s43681-024-00583-7> (vid. pág. 4).
- [6] T. Mikolov, K. Chen, G. Corrado y J. Dean, *Efficient Estimation of Word Representations in Vector Space*, 2013. arXiv: 1301.3781 [cs.CL]. dirección: <https://arxiv.org/abs/1301.3781> (vid. pág. 4).
- [7] J. Pennington, R. Socher y C. Manning, *Glove: Global Vectors for Word Representation*, ene. de 2014. DOI: 10.3115/v1/D14-1162 (vid. pág. 4).
- [8] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019. arXiv: 1810.04805 [cs.CL]. dirección: <https://arxiv.org/abs/1810.04805> (vid. pág. 5).
- [9] A. Radford, «Improving language understanding by generative pre-training,» *OpenAI*, 2018 (vid. pág. 5).
- [10] M. A. K. Raiaan et al., «A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges,» *IEEE Access*, vol. 12, págs. 26 839-26 874, 2024. DOI: 10.1109/ACCESS.2024.3365742 (vid. pág. 5).
- [11] H. Naveed et al., *A Comprehensive Overview of Large Language Models*, 2024. arXiv: 2307.06435 [cs.CL]. dirección: <https://arxiv.org/abs/2307.06435> (vid. pág. 5).
- [12] S. Gupta, R. Ranjan y S. N. Singh, *A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions*, 2024. arXiv: 2410.12837 [cs.CL]. dirección: <https://arxiv.org/abs/2410.12837> (vid. pág. 6).
- [13] Y. Gao et al., *Retrieval-Augmented Generation for Large Language Models: A Survey*, 2024. arXiv: 2312.10997 [cs.CL]. dirección: <https://arxiv.org/abs/2312.10997> (vid. págs. 6, 7).
- [14] K. Inc, *RAG vs LLM? Understanding the Unique Capabilities and Limitations of Each Approach* — kanerika, <https://medium.com/@kanerika/rag-vs-llm-understanding-the-unique-capabilities-and-limitations-of-each-approach-f922486a4225>, [Accessed 20-05-2025] (vid. págs. 6, 7).
- [15] T. Şakar y H. Emekci, «Maximizing RAG efficiency: A comparative analysis of RAG methods,» *Natural Language Processing*, vol. 31, n.º 1, págs. 1-25, 2025. DOI: 10.1017/nlp.2024.53 (vid. pág. 6).

- [16] S. K. Dam, C. S. Hong, Y. Qiao y C. Zhang, *A Complete Survey on LLM-based AI Chatbots*, 2024. arXiv: 2406.16937 [cs.CL]. dirección: <https://arxiv.org/abs/2406.16937> (vid. pág. 7).
- [17] H. Touvron et al., *Llama 2: Open Foundation and Fine-Tuned Chat Models*, 2023. arXiv: 2307.09288 [cs.CL]. dirección: <https://arxiv.org/abs/2307.09288> (vid. pág. 7).
- [18] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu y R. S. et al., *Gemini: A Family of Highly Capable Multimodal Models*, 2024. arXiv: 2312.11805 [cs.CL]. dirección: <https://arxiv.org/abs/2312.11805> (vid. págs. 7, 8).
- [19] DeepSeek-AI et al., *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, 2025. arXiv: 2501.12948 [cs.CL]. dirección: <https://arxiv.org/abs/2501.12948> (vid. pág. 7).
- [20] *RAG makes LLMs better and equal — Pinecone — pinecone.io*, <https://www.pinecone.io/blog/rag-study/>, [Accessed 19-05-2025] (vid. pág. 7).
- [21] J. Lee et al., *Gecko: Versatile Text Embeddings Distilled from Large Language Models*, 2024. arXiv: 2403.20327 [cs.CL]. dirección: <https://arxiv.org/abs/2403.20327> (vid. pág. 8).
- [22] *State-of-the-art text embedding via the Gemini API — developers.googleblog.com*, <https://developers.googleblog.com/en/gemini-embedding-text-model-now-available-gemini-api/>, [Accessed 19-05-2025] (vid. pág. 8).
- [23] L. Mei, S. Mo, Z. Yang y C. Chen, *A Survey of Multimodal Retrieval-Augmented Generation*, 2025. arXiv: 2504.08748 [cs.IR]. dirección: <https://arxiv.org/abs/2504.08748> (vid. pág. 8).
- [24] J. Achiam et al., «Gpt-4 technical report,» *arXiv preprint arXiv:2303.08774*, 2023 (vid. pág. 8).
- [25] J. Stratton, «An introduction to microsoft copilot,» en *Copilot for Microsoft 365: Harness the Power of Generative AI in the Microsoft Apps You Use Every Day*, Springer, 2024, págs. 19-35 (vid. pág. 8).
- [26] A. Novikau, «Online vs. offline LLM inference: Unlocking the best of both worlds in Mobile applications,» *IJSEA— International Journal of Science and Engineering Applications*, vol. 1, pág. 5, 2024 (vid. págs. 8, 9).
- [27] Z. Dong et al., *Exploring Context Window of Large Language Models via Decomposed Positional Vectors*, 2024. arXiv: 2405.18009 [cs.CL]. dirección: <https://arxiv.org/abs/2405.18009> (vid. pág. 10).
- [28] W. Kwon et al., *Efficient Memory Management for Large Language Model Serving with PagedAttention*, 2023. arXiv: 2309.06180 [cs.LG]. dirección: <https://arxiv.org/abs/2309.06180> (vid. págs. 10, 11, 13).
- [29] J. Manchanda, L. Boettcher, M. Westphalen y J. Jasser, *The Open Source Advantage in Large Language Models (LLMs)*, 2025. arXiv: 2412.12004 [cs.CL]. dirección: <https://arxiv.org/abs/2412.12004> (vid. pág. 10).
- [30] C. Brousseau y M. Sharp, *LLMs in production: From language models to successful products*. Manning Publications, 2025 (vid. págs. 12, 32).
- [31] X. Zhuge et al., *SpecOffload: Unlocking Latent GPU Capacity for LLM Inference on Resource-Constrained Devices*, 2025. arXiv: 2505.10259 [cs.LG]. dirección: <https://arxiv.org/abs/2505.10259> (vid. pág. 12).
- [32] A. Fei y M. S. Abdelfattah, *NITRO: LLM Inference on Intel Laptop NPUs*, 2024. arXiv: 2412.11053 [cs.CL]. dirección: <https://arxiv.org/abs/2412.11053> (vid. págs. 12, 13).
- [33] *NITRO: LLM Inference on Intel Laptop NPUs — PromptLayer — promptlayer.com*, <https://www.promptlayer.com/research-papers/unlocking-laptop-ai-running-llms-on-your-intel-npu>, [Accessed 21-05-2025] (vid. págs. 12, 13).

- [34] *What is a Neural Processing Unit (NPU)?* — IBM — *ibm.com*, <https://www.ibm.com/think/topics/neural-processing-unit>, [Accessed 21-05-2025] (vid. pág. 12).
- [35] J. Wang y Z. Duan, *Agent AI with LangGraph: A Modular Framework for Enhancing Machine Translation Using Large Language Models*, 2024. arXiv: 2412.03801 [cs.CL]. dirección: <https://arxiv.org/abs/2412.03801> (vid. pág. 14).
- [36] T. Bocklisch, J. Faulker, N. Pawlowski y A. Nichol, «Rasa: Open Source Language Understanding and Dialogue Management,» dic. de 2017. DOI: 10.48550/arXiv.1712.05181 (vid. pág. 15).
- [37] S. R. Dipta y S. Vallurupalli, *UMBCLU at SemEval-2024 Task 1A and 1C: Semantic Textual Relatedness with and without machine translation*, 2024. arXiv: 2402.12730 [cs.CL]. dirección: <https://arxiv.org/abs/2402.12730> (vid. págs. 17, 20).
- [38] R. Mitchell, *Web scraping with python*. " O'Reilly Media, Inc.", 2024 (vid. pág. 17).
- [39] S. R. Bhat, M. Rudat, J. Spiekermann y N. Flores-Herr, *Rethinking Chunk Size For Long-Document Retrieval: A Multi-Dataset Analysis*, 2025. arXiv: 2505.21700 [cs.IR]. dirección: <https://arxiv.org/abs/2505.21700> (vid. págs. 18, 19).
- [40] N. F. Liu et al., *Lost in the Middle: How Language Models Use Long Contexts*, 2023. arXiv: 2307.03172 [cs.CL]. dirección: <https://arxiv.org/abs/2307.03172> (vid. págs. 20, 22).
- [41] B. Donato, L. Mariani, D. Micucci y O. Riganelli, *Studying How Configurations Impact Code Generation in LLMs: the Case of ChatGPT*, 2025. arXiv: 2502.17450 [cs.SE]. dirección: <https://arxiv.org/abs/2502.17450> (vid. pág. 22).
- [42] M. N. Nguyen, A. Baker, C. Neo, A. Roush, A. Kirsch y R. Shwartz-Ziv, *Turning Up the Heat: Min-p Sampling for Creative and Coherent LLM Outputs*, 2025. arXiv: 2407.01082 [cs.CL]. dirección: <https://arxiv.org/abs/2407.01082> (vid. pág. 22).
- [43] Q. Wang, Y. Fu, Y. Cao, S. Wang, Z. Tian y L. Ding, *Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models*, 2025. arXiv: 2308.15022 [cs.CL]. dirección: <https://arxiv.org/abs/2308.15022> (vid. págs. 22, 23).
- [44] A. Wolters, A. Arz von Straussenburg y D. M. Riehle, «Evaluation Framework for Large Language Model-based Conversational Agents,» jul. de 2024 (vid. pág. 27).
- [45] C.-Y. Lin, «ROUGE: A Package for Automatic Evaluation of summaries,» ene. de 2004, pág. 10 (vid. pág. 27).
- [46] K. Papineni, S. Roukos, T. Ward y W. J. Zhu, «BLEU: a Method for Automatic Evaluation of Machine Translation,» oct. de 2002. DOI: 10.3115/1073083.1073135 (vid. pág. 27).
- [47] J. Gu et al., *A Survey on LLM-as-a-Judge*, 2025. arXiv: 2411.15594 [cs.CL]. dirección: <https://arxiv.org/abs/2411.15594> (vid. pág. 27).
- [48] H. Li et al., *LLMs-as-Judges: A Comprehensive Survey on LLM-based Evaluation Methods*, 2024. arXiv: 2412.05579 [cs.CL]. dirección: <https://arxiv.org/abs/2412.05579> (vid. pág. 28).
- [49] S. Vatsal y H. Dubey, *A Survey of Prompt Engineering Methods in Large Language Models for Different NLP Tasks*, 2024. arXiv: 2407.12994 [cs.CL]. dirección: <https://arxiv.org/abs/2407.12994> (vid. pág. 31).
- [50] A. Salinas y F. Morstatter, *The Butterfly Effect of Altering Prompts: How Small Changes and Jailbreaks Affect Large Language Model Performance*, 2024. arXiv: 2401.03729 [cs.CL]. dirección: <https://arxiv.org/abs/2401.03729> (vid. págs. 31, 32).
- [51] Y. Li y K. Fu, «Research on Minimalism in Interface Design Based on Gestalt Psychology,» dic. de 2022, págs. 825-837, ISBN: 978-2-494069-04-6. DOI: 10.2991/978-2-494069-05-3.101 (vid. pág. 39).

APÉNDICE A

Anexos

A.1 CAPTURA DE PANTALLA DEL FORMULARIO UTILIZADO PARA LA EVALUACIÓN.

PREGUNTA CHATBOT 1

Pregunta realizada al chatbot

Tu respuesta

Respuesta chatbot

Tu respuesta

Relevancia de la respuesta

1

2

3

4

5

☆

☆

☆

☆

☆

Calidad y fluidez

1

2

3

4

5

☆

☆

☆

☆

☆

Evaluación adicional

☐ Alucinación detectada

☐ Fallo de compresión

☐ Respuesta vacía

☐ Otro:

Comentarios sobre respuesta

Tu respuesta

Figura A.1: Fragmento ilustrativo del formulario de evaluación utilizado.