



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN



Inteligencia Artificial Generativa aplicada a la edición de lenguaje natural de un sistema de comunicaciones automatizadas para procesos de empresa

Estudiante: Juan Blas Araujo Fernández

Dirección: Carlos Vázquez Regueiro

Juan Antonio Martín Pernas

A Coruña, septiembre de 2024.

Dedicatoria

Agradecimientos

En primer lugar, quiero agradecer a la Universidad de Coruña por brindarme la oportunidad de llevar a cabo este proyecto, y por proporcionarme los recursos necesarios para realizarlo.

A Juan y a Carlos, por su ayuda durante todo el proceso, por sus consejos y propuestas, y por su paciencia. También quiero reconocer la ayuda de Diego y del resto del equipo de SREC Solutions, por su colaboración y sugerencias, que aportaron mucho a este proyecto.

Finalmente, quiero agradecer a mis padres y mi hermano, por estar siempre a mi lado en los momentos buenos y en los malos. Por apoyarme y por confiar en mí siempre, no solo en este proyecto, sino en todo lo que me he propuesto a lo largo de mi vida.

Resumen

Este proyecto tiene como objetivo diseñar un sistema de comprensión de lenguaje natural en base a un [LLM](#) (Large Language Model) libre, con el fin de agilizar y optimizar diversos procesos de empresa. Para ello, se evalúa primero el rendimiento de los diferentes modelos de lenguaje disponibles, utilizando librerías de procesamiento de lenguaje. El modelo seleccionado se integra en una API, a la que se podrán enviar solicitudes en lenguaje natural, y esta devolverá una respuesta, siendo capaz de comprender las instrucciones que recibe, evaluar el contenido de las peticiones y razonar respuestas coherentes. Además, se crean las diferentes entradas ([prompts](#)) que se le presentarán al [LLM](#), para guiar su comportamiento y asegurar que responda de la manera deseada para cada funcionalidad que ofrezca el sistema. El objetivo final del proyecto es integrar el sistema dentro del chatbot de [SINVAD](#), para que pueda ser probado en situaciones reales.

Abstract

This project aims to design a natural language understanding system, based on a open [LLM](#) (Large Language Model), with the goal of streamlining and optimizing various business processes. To achieve this, the performance of the different available language models is first be evaluated using language processing libraries. The selected model is integrated into an API, to which requests in natural language can be sent, and it will return a response, being capable of understanding the instructions it receives, evaluating the content of the requests, and reasoning coherent responses. Furthermore, different [prompts](#) are created for the [LLM](#) in order to guide its behaviour and ensure that it responds in the desired manner for each functionality offered by the system. The final goal of the project is to integrate the system into [SINVAD](#)'s chatbot, so it can be tested in real-world situations.

Palabras clave:

- LLM
- IA Generativa
- Generación de texto
- Lenguaje natural
- Python
- PyTorch
- ChatBot
- Transformers

Keywords:

- LLM
- Generative AI

- Text generation
- Natural language
- Python

- PyTorch
- ChatBot
- Transformers

Índice general

1	Introducción	1
1.1	Contexto y motivación del proyecto	1
1.2	Objetivos	2
1.3	Estructura de la memoria	2
1.4	Trabajo relacionado	3
2	Fundamentos tecnológicos	5
2.1	Hardware	5
2.2	Asistentes virtuales: SINVAD	6
2.3	Modelos de lenguaje	6
2.3.1	Mistral 7B	7
2.3.2	Phi-2	7
2.3.3	Llama 2 7B	7
2.3.4	GPT 3.5	7
2.3.5	Bert	7
2.3.6	Discusión	7
2.4	Herramientas específicas para manejar LLM	8
2.4.1	LM Studio	8
2.4.2	Python	9
2.5	Librerías	10
2.5.1	PyTorch	10
2.5.2	Transformers	10
2.5.3	Flask	11
2.5.4	Requests	11
2.6	Otras tecnologías	11
2.6.1	Git	11
2.6.2	GitHub Desktop	12

2.6.3	Visual Studio Code	12
2.6.4	Lucidchart	12
2.6.5	Docker Compose	12
2.6.6	Jmeter	12
3	Análisis y metodología	13
3.1	Metodología de desarrollo	13
3.1.1	Tipos de metodologías de desarrollo	13
3.1.2	Metodología empleada	14
3.2	Análisis de requisitos	14
3.2.1	Requisitos funcionales	15
3.2.2	Requisitos no funcionales	15
3.3	Casos de uso	15
3.3.1	<i>Comandos</i>	16
3.3.2	<i>Traducción</i>	17
3.3.3	<i>Lista</i>	17
3.3.4	<i>Extracción de datos</i>	17
3.3.5	<i>Gestión de diálogo</i>	18
4	Planificación y costes	21
4.1	Planificación	21
4.1.1	Elección del modelo	21
4.1.2	Estudio de las librerías de procesamiento de lenguaje	22
4.1.3	Creación del primer prototipo	22
4.1.4	Implementación de la API	22
4.1.5	Conexión con SINVAD	22
4.2	Riesgos y planes de contingencias	22
4.2.1	Riesgos	23
4.2.2	Planes de contingencia	23
4.3	Seguimiento de la planificación	24
4.3.1	Elección del modelo	24
4.3.2	Estudio de las librerías de procesamiento de lenguaje	24
4.3.3	Creación del primer prototipo	25
4.3.4	Implementación de la API	25
4.3.5	Conexión con SINVAD	26
4.4	Recursos	27
4.4.1	Recursos humanos	27
4.4.2	Recursos materiales	28

4.4.3	Recursos software	28
4.5	Estimación de costes	29
4.5.1	Costes de los recursos humanos	29
4.5.2	Costes de los recursos materiales	29
5	Arquitectura y diseño	31
5.1	Arquitectura general de la aplicación	31
5.2	Servidor LLM	33
5.3	Procesador de texto	34
5.4	Vistas de la aplicación	34
5.4.1	Estructura general	35
5.4.2	Prototipos de vistas	35
5.4.3	Navegación en las vistas	38
6	Implementación	39
6.1	Generación de texto	39
6.1.1	Carga del modelo y el <i>tokenizer</i>	39
6.1.2	Creación de conversaciones	40
6.1.3	Generación de texto a partir de una conversación inicial	40
6.2	Implementación de los casos de uso	41
6.2.1	Prompts	42
6.2.2	Composición de respuestas	42
6.2.3	<i>Comandos</i>	44
6.2.4	<i>Traducción</i>	45
6.2.5	<i>Lista</i>	46
6.2.6	<i>Extracción de datos</i>	47
6.2.7	<i>Gestión de diálogo</i>	47
6.3	Implementación de Docker	49
6.4	Implementación en LM Studio	50
6.5	Interfaz de usuario	50
7	Pruebas y Resultados	55
7.1	Comparativa de resultados de Mistral 7B y Phi-2	55
7.2	Tiempos de generación	56
7.3	Pruebas de carga al Servidor LLM	56
7.4	Pruebas de <i>Comandos</i>	57
7.5	Pruebas de <i>Traducción</i>	58
7.6	Pruebas de <i>Lista</i>	59

7.6.1	Errores leves	59
7.6.2	Errores graves	60
7.7	Pruebas de <i>Extracción de datos</i>	61
7.8	Pruebas de <i>Gestión de diálogo</i>	62
8	Conclusiones y trabajo futuro	65
8.1	Conclusiones	65
8.2	Trabajo futuro	66
A	Documentos de texto para las pruebas	68
A.1	Ejemplos para las pruebas de <i>Comandos</i>	68
A.2	Ejemplos para las pruebas de <i>Traducción</i>	70
A.3	Ejemplos para las pruebas de <i>Lista</i>	71
A.4	Ejemplos para las pruebas de <i>Extracción de datos</i>	73
A.5	Conversaciones obtenidas en las pruebas de <i>Gestión de diálogo</i>	75
A.5.1	Mistral 7B	75
A.5.2	ChatGPT 4o	77
A.5.3	Claude 3	78
B	Especificación API del sistema	81
B.1	Servidor LLM	81
B.2	Procesador de texto	82
B.2.1	Comandos	82
B.2.2	Traducción	84
B.2.3	Lista	85
B.2.4	Extracción de datos	85
B.2.5	Gestión de diálogo	86
	Glosario	88
	Bibliografía	90

Índice de figuras

2.1	Interfaz de LM Studio para buscar modelos	9
2.2	Pantalla de chat de LM Studio	10
3.1	Diagrama de flujo de SINVAD	18
4.1	Diagrama de Gantt con la planificación del proyecto	21
5.1	Esquema general de nuestro sistema	32
5.2	Prototipo general de interfaz de la aplicación	35
5.3	Prototipo para la definición de <i>Comandos</i>	36
5.4	Prototipo para el caso de uso <i>Comandos</i>	36
5.5	Prototipo para el caso de uso <i>Gestión de diálogo</i>	37
6.1	Pruebas con LM Studio	51

Lista de Tablas

2.1	Equipos utilizados en el proyecto	5
2.2	Tabla comparativa de diferentes modelos LLM	8
4.1	Costes de los recursos humanos	29
4.2	Costes de los recursos materiales	29
4.3	Coste total del proyecto	30
7.1	Comparativa de resultados según los modelos Mistral y Phi-2	55
7.2	Tiempos de generación del modelo Mistral 7B	56
7.3	Pruebas de carga al servidor LLM (tiempos en milisegundos)	56
7.4	Pruebas de carga al servidor LLM (peticiones simultáneas, tiempos en milise- gundos)	56

Índice de Códigos

6.1	Función de carga del LLM y el tokenizer	39
6.2	Formato para la creación de conversaciones	40
6.3	Función de generación de texto	41
6.4	Función para definir la conversación para <i>Comandos</i>	43
6.5	Función de obtención de comandos	44
6.6	Función de obtención de la traducción	45
6.7	Función de obtención de la lista	46
6.8	Función de <i>extacción de datos</i>	47
6.9	Función de <i>Gestión de diálogo</i>	48
6.10	archivo .yaml del contenedor de la API del servidor LLM	49
6.11	Función para obtener el estado del modelo para la interfaz de usuario	52
6.12	Función para realizar consultas al modelo para la interfaz de usuario	53
B.1	Formato del <i>userInput</i> en el cuerpo de una petición /genResponse	82
B.2	Cuerpo de la solicitud en la petición /commands/loadChat	83

Introducción

EN este primer capítulo se presentará una primera visión del proyecto, incluyendo el contexto y las motivaciones para la realización del trabajo. También se expondrán los principales objetivos del proyecto y se detallará la estructura de esta memoria. Además, se incluirá un resumen de varios trabajos relacionados que fueron útiles como punto de partida y a lo largo del desarrollo del proyecto.

1.1 Contexto y motivación del proyecto

El personal de cualquier empresa pierde una parte importante de su tiempo para ser capaz de acceder a la información que necesita para llevar a cabo su trabajo o en registrar nuevos datos en un sistema de información de la empresa (hojas de cálculo, CRM, ERP, formularios web, etc.)

SINVAD es un sistema de Comunicaciones Automatizadas basado en Comprensión de Lenguaje Natural desarrollado por la empresa SREC Solutions. Su objetivo es asistir de manera autónoma al usuario para consultar información, registrar datos y supervisar actividades desde un dispositivo iOS, Android o un navegador web con sus propias palabras, en múltiples idiomas y adaptado a los procesos de la empresa.

El sistema tiene limitaciones relacionadas con su capacidad de usar comprensión de lenguaje natural. Por ejemplo, el cliente debe aprender qué posibles expresiones puede utilizar y configurar paso a paso el flujo de comunicaciones con todas sus posibles ramas y bifurcaciones.

En este proyecto, se propone la realización de un prototipo de sistema basado en un modelo de lenguaje (**LLM**) libre que facilite su instalación en la plataforma cloud y su integración con la API REST de **SINVAD** para gestionar la interacción a través de la API REST de **SINVAD**. Esto facilitaría las tareas de gestión y consultas, como por ejemplo:

- Realizar traducción automática en tiempo real.

- Conectarlo a una base de datos de registros y realizar consultas en base a texto desestructurado.
- Convertir texto desestructurado en un conjunto estructurado de acciones y datos configurados en el flujo de comunicación.

1.2 Objetivos

El objetivo del proyecto es diseñar un sistema de gestión de la comprensión de lenguaje natural en base a un [LLM](#) libre.

- Se realizará un prototipo de aplicación web.
- Se integrará con el sistema de Comunicaciones Automatizadas [SINVAD](#) a través de llamadas tipo REST.
- Se testeará el sistema final en condiciones reales de operación.

1.3 Estructura de la memoria

La memoria se compone de los siguientes capítulos:

- **Introducción:** Primer capítulo de la memoria, en el que se exponen los principales objetivos y motivaciones para la realización del proyecto. También se incluye una sección de trabajo relacionado.
- **Fundamentos tecnológicos:** En este capítulo se detallan las tecnologías, herramientas y plataformas utilizadas en el desarrollo del proyecto.
- **Análisis y metodologías:** Se realiza un análisis exhaustivo de los objetivos del proyecto y se presentan las metodologías empleadas para abordar dichos objetivos. También se mostrarán los requisitos funcionales y no funcionales, así como una descripción de los principales casos de uso implementados.
- **Planificación y costes:** Este capítulo aborda la planificación del proyecto, incluyendo las fases que lo componen, los riesgos que se presentarán durante el desarrollo y el seguimiento de dicha planificación. Además, se proporciona un análisis de los costes asociados al proyecto, incluyendo recursos humanos, materiales y software.
- **Arquitectura y diseño:** Este capítulo está dedicado a la arquitectura del sistema, analizando cada uno de sus componentes. Además, se detallan todas las decisiones de diseño tomadas. Se incluye también la arquitectura de la interfaz de la aplicación y sus diferentes vistas.

- **Implementación:** Este capítulo incluye todos los detalles sobre la implementación del sistema de generación de texto y de los casos de uso, junto con varios ejemplos de resultados obtenidos. También se muestran los detalles de la implementación de la interfaz de usuario y en LM Studio.
- **Pruebas y resultados:** Documentación de las pruebas realizadas al sistema, incluyendo comparativas entre modelos, tiempos de respuesta y pruebas específicas de cada funcionalidad. También se presentan los resultados finales del proyecto, demostrando cómo se han cumplido los objetivos planteados.
- **Conclusiones y trabajo futuro:** Capítulo final de la memoria donde se hará un resumen de los objetivos iniciales del proyecto y las conclusiones que se pueden sacar de los mismos después de todo el proceso. También se analizarán las posibles mejoras y avances futuros.

1.4 Trabajo relacionado

Existen varios trabajos relacionados con el uso de los modelos de lenguaje y sus múltiples funcionalidades y posibles aplicaciones. Muchos de estos trabajos sirvieron como guía en el desarrollo de este proyecto. A continuación detallaremos los más importantes.

Un artículo que nos ayudó al principio del proyecto es *Understanding the Capabilities, Limitations, and Societal Impact of Large Language Models*[1], de Alex Tamkin, Miles Brundage, Jack Clark y Deep Ganguli, que da una visión general de las capacidades reales de los modelos de lenguaje en la actualidad, así como el impacto que pueden llegar a tener en nuestra sociedad.

En relación con la configuración de los modelos de lenguaje, el artículo *Conversational AI: An Explication of Few-Shot Learning Problem in Transformers-Based Chatbot Systems*[2] ofrece una configuración para obtener respuestas más precisas al utilizar los LLMs, el modelo HIST. El estudio demuestra que este modelo es efectivo en resolver los problemas de aprendizaje con pocos ejemplos (*few shot learning*) y gestión del contexto en los chatbots, mejorando significativamente la precisión y la coherencia en las respuestas. Se consiguió un enfoque innovador para mejorar el desempeño de los chatbots en aplicaciones prácticas.

El trabajo publicado por los integrantes de MistralAI llamado *Mistral 7B*[3], fue crucial a la hora de seleccionar el LLM adecuado para nuestro proyecto. Este trabajo ofrece un análisis detallado de las capacidades del modelo *Mistral 7B* frente a otros modelos más conocidos en el mercado. Los resultados presentados en este artículo fueron verificados con métodos robustos y consistentes, confirmando su validez y exactitud.

El proyecto de Simran Arora *Ask Me Anything: A simple strategy for prompting language*

models[4] propone una técnica innovadora para mejorar el rendimiento de los modelos de lenguaje al combinar respuestas de múltiples *prompts*, en lugar de depender de un solo *prompt* que contenga toda la información necesaria. Este trabajo fue de mucha utilidad, sobre todo al complementarlo con la información de la página de *Prompting Mistral*[5], que ofrece ejemplos y técnicas de *prompting* en diversas funcionalidades, pero más centrado en el modelo de Mistral, que es el modelo principal utilizado en este proyecto.

Olga Zem publicó un estudio llamado *Exploring LLM Leaderboards*[6] en el que describe de forma sencilla las categorías que existen en el *Open LLM Leaderboard*[7] para evaluar los *LLMs*. Esta *leaderboard* contiene un ranking de *LLMs*, en el que a cada modelo se le asigna una puntuación para cada categoría. Este estudio resultó muy útil para comprender los métodos y técnicas que se realizan para evaluar el rendimiento de los modelos de lenguaje en diferentes tareas. Estas técnicas de evaluación fueron muy útiles para comparar modelos aparentemente equivalentes por tamaño, pero muy diferentes en cuanto a funcionalidades.

Fundamentos tecnológicos

ESTE capítulo describe los elementos tecnológicos empleados durante el desarrollo de este proyecto, tanto a nivel hardware como a nivel software. Además, se incluye una breve explicación sobre los modelos de lenguaje de gran tamaño, así como algunos de los considerados para la ejecución del trabajo.

2.1 Hardware

Con relación al hardware, se usaron tres equipos, un portátil y dos equipos de sobremesa. Las especificaciones de cada uno de ellos se pueden ver en la tabla 2.1.

El portátil MSI se utilizó para las pruebas iniciales y para todo el proceso de investigación. Sin embargo, sus limitaciones de VRAM imposibilitaron la carga en memoria del modelo utilizado en todos los casos de uso del proyecto.

El primer equipo de sobremesa es un servidor de la UDC, que permitió realizar las primeras pruebas reales con el modelo de Mistral, gracias a sus 24GB de VRAM.

El segundo equipo es otro servidor, que pertenece a SREC Solutions [8]. Fue adquirido para realizar todas las tareas relacionadas con la IA generativa y los modelos de lenguaje. Este equipo también cuenta con 24GB de VRAM, suficiente para cargar el LLM.

Equipo	Sist. operativo	Procesador	Tarjeta gráfica NVIDIA
<i>MSI GF65</i>	Windows 10	Intel Core I7-9750H	GeForce RTX 2060 (6GB)
<i>Servidor UDC</i>	Ubuntu 22.4	Ryzen Threadripper 3970X	GeForce RTX 3090 (24GB)
<i>Servidor SREC</i>	Ubuntu 22.4	Intel Core i9-14900K	GeForce RTX 4090 (24GB)

Tabla 2.1: Equipos utilizados en el proyecto

2.2 Asistentes virtuales: SINVAD

Un asistente virtual es un programa que utiliza inteligencia artificial para realizar tareas y brindar asistencia a los usuarios de manera automatizada.

SINVAD es un chatbot inteligente de la empresa SREC Solutions [8], que ofrece una plataforma de configuración de comandos, adaptándose a diferentes procesos de empresa. Facilita procesos como consultar y registrar datos o automatizar tareas, entre otros. Este proceso se lleva a cabo mediante un flujo de trabajo donde se van configurando una serie de pasos que definen el comportamiento del sistema. Su principal inconveniente radica en las limitaciones a la hora de comprender el lenguaje natural. El usuario debe conocer los comandos específicos o las formas exactas en las que se deben proporcionar los datos para que el sistema los comprenda.

2.3 Modelos de lenguaje

Los "large language models" (LLMs) [9] son modelos de aprendizaje profundo basados en redes neuronales muy grandes. Estas redes pueden estar formadas por miles de millones de parámetros. El modelo más conocido y utilizado actualmente es el de OpenAI. La versión GPT-3 tiene más de 175.000 millones de parámetros. Actualmente, se desconoce el número de parámetros de GPT-3.5 y GPT-4, algunos expertos estiman que GPT-4 tiene una capacidad 600 veces mayor que GPT-3, lo que serían unos 100 billones de parámetros.

Los LLM funcionan como un modelo matemático probabilístico, en el que se calcula la probabilidad de la siguiente palabra a generar, dada una secuencia de entrada (**prompt**). Esto se consigue gracias a un previo entrenamiento del modelo utilizando grandes cantidades de datos, normalmente recopilados de Internet (documentos académicos, páginas web, artículos de noticias, libros, redes sociales, etc.).

Estos modelos están diseñados para procesar y generar texto en lenguaje natural, por lo que pueden realizar una amplia gama de tareas. Entre las más utilizadas encontramos:

- Extraer características de un texto.
- Responder preguntas.
- Resumir textos.
- Traducir textos.
- Generar código.
- Crear nuevas historias.

A continuación se describen los principales modelos considerados para este proyecto.

2.3.1 Mistral 7B

Mistral 7B [3] es uno de los modelos de Mistral AI, una startup enfocada en inteligencia artificial. Es de software libre y su arquitectura se compone de alrededor de 7.300 millones de parámetros. Tiene mucha versatilidad a la hora de realizar diferentes tareas gracias a la gran cantidad de datos con los que fue entrenado. Esto también le permite la generación de texto en múltiples idiomas.

2.3.2 Phi-2

Este modelo fue desarrollado por la empresa Microsoft con el objetivo de obtener resultados similares o superiores a otros modelos del mercado, a pesar de ser significativamente más pequeño, con 2.700 millones de parámetros [10]. Ofrece buenos resultados en una amplia variedad de tareas. En 2024 se liberó el modelo para su uso comercial gratuito.

2.3.3 Llama 2 7B

LLaMA 2 [11] es un modelo abierto desarrollado por Meta. Está compuesto de alrededor de 6.740 millones de parámetros y se centra en las tareas de comprensión de texto. Fue entrenado con datos diversos y permite la generación de texto en varios idiomas diferentes.

2.3.4 GPT 3.5

Este modelo [12], desarrollado por OpenAI, cuenta con aproximadamente 175.000 millones de parámetros. Gracias a su gran tamaño, ofrece excelentes resultados en una amplia variedad de aplicaciones. También ofrece soporte para muchos idiomas, aunque los mejores resultados se obtienen en inglés. No es un modelo abierto.

2.3.5 Bert

BERT (Bidirectional Encoder Representations from Transformers) [13] fue desarrollado por Google AI Language. Publicado en 2018, es uno de los modelos de lenguaje más influyentes y ampliamente utilizados en el campo de la IA. Es un modelo abierto de 110 millones de parámetros, enfocado principalmente a tareas como la clasificación de texto, extracción de información y respuesta a preguntas.

2.3.6 Discusión

Una muy buena forma de comparar los modelos de lenguaje es utilizando el *LLM Benchmark* [14]. Esta herramienta emplea una serie de pruebas y métricas para evaluar y comparar

Modelo	Núm. parámetros (Aprox.)	VRAM	Software Libre	Objetivo principal
<i>Mistral 7B</i>	7300 mill.	24GB	Si	Gen. texto
<i>Phi-2</i>	2700 mill.	12GB	Si	Gen. texto
<i>Llama 2 7B</i>	6740 mill.	24GB	Si	Gen. texto
<i>GPT-3.5</i>	175.000 mill.	>128GB	Si	Gen. texto
<i>Bert (base)</i>	110 mill.	6GB	Si	Comp. de lenguaje ¹

Tabla 2.2: Tabla comparativa de diferentes modelos LLM

el rendimiento de los modelos de lenguaje. Se centra en diversas capacidades y aplicaciones, como la comprensión de texto, la eficiencia y la resolución de tareas específicas, entre otras.

Para nuestro proyecto nos interesa el *Open LLM Leaderboard* [7], que se centra en los modelos libres. Esta página se actualiza constantemente con nuevos modelos y avances, por lo que es probable que los modelos considerados para este proyecto ya estén superados por los actuales en cuanto a rendimiento general.

En la tabla 2.2 encontramos un resumen de las características de los modelos presentados. *GPT 3.5* es un modelo muy superior al resto, pero no se contempló para este proyecto debido a que no es un modelo libre y es demasiado grande para ser ejecutado en un servidor local. Se decidió optar por *Mistral 7B* y *Phi-2*, porque tienen una mayor versatilidad a la hora de realizar tareas diferentes, a diferencia de *Bert*, que está enfocado en tareas más específicas. *Llama 2* es un modelo similar a *Mistral 7B* en tamaño, pero diversos estudios (artículo *Mistral 7B* [3]) han demostrado que el modelo de Mistral es superior. Ya sea en razonamiento y sentido común, conocimiento del mundo, comprensión lectora o tareas relacionadas con las matemáticas, Mistral siempre está por encima.

2.4 Herramientas específicas para manejar LLM

2.4.1 LM Studio

Esta herramienta permite descargar y utilizar modelos libres [15]. Es muy útil para probar y comparar los diferentes modelos que están disponibles gracias a su gran facilidad de uso (ver figura 2.1). Muestra una interfaz de chat para poder interactuar con los LLM de forma local (ver figura 2.2). Sus principales ventajas son:

- **GPU offload:** Permite una gran flexibilidad en el uso de la GPU para la inferencia, permitiendo indicar el número de capas del modelo que se procesarán con GPU, mientras

¹ Tareas como la clasificación de texto, extracción de información y respuesta a preguntas

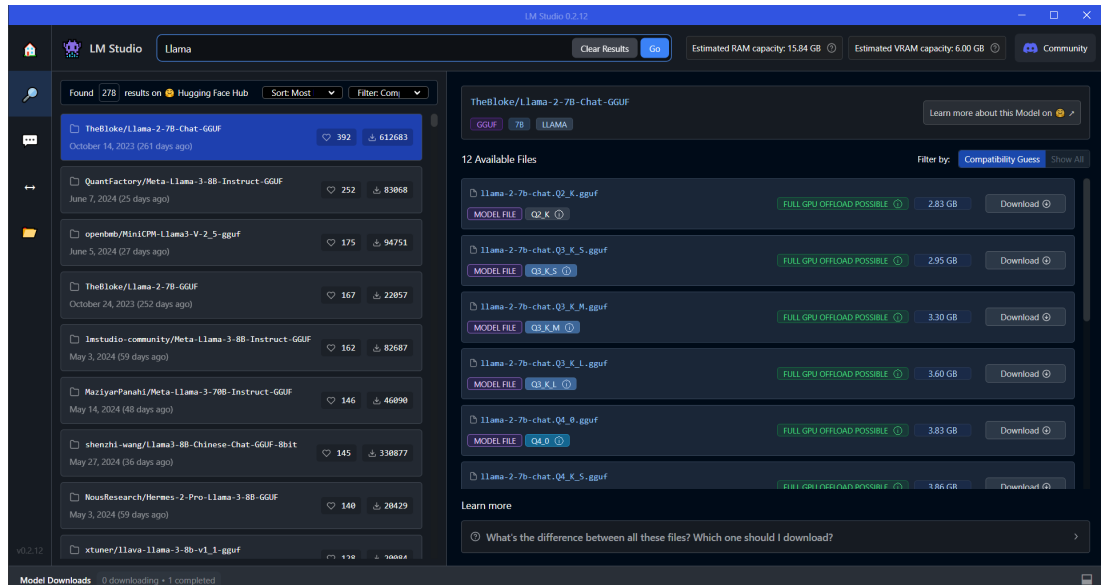


Figura 2.1: Interfaz de LM Studio para buscar modelos

que el resto se deja para la CPU. Esto resulta muy útil para casos en los que no se tiene suficiente VRAM, como en el caso del portátil empleado en las etapas iniciales del proyecto (ver tabla 2.1).

- **Modelos cuantizados²:** Gracias a esto, se pueden utilizar modelos grandes en equipos que, de otro modo, no podrían soportarlos.

2.4.2 Python

Python [16] es un lenguaje de programación de alto nivel, diseñado para ser fácil de leer y escribir. Fue creado por Guido van Rossum y lanzado por primera vez en 1991. Python es muy utilizado en el campo de la inteligencia artificial y los modelos de lenguaje (LLM) gracias a su gran flexibilidad y su amplia variedad de librerías diseñadas para ello. También cuenta con una comunidad muy activa que contribuye con nuevas herramientas, mejoras y soporte técnico.

²La cuantización es el cambio en el tipo de dato de los parámetros que contiene un modelo. Por ejemplo, los parámetros podrían pasar de un float32, a su valor equivalente en float16, lo que reduciría a la mitad el tamaño total del modelo. Una cuantización demasiado agresiva (float64 a int8) podría cambiar excesivamente los resultados del modelo y reducir la calidad de sus respuestas

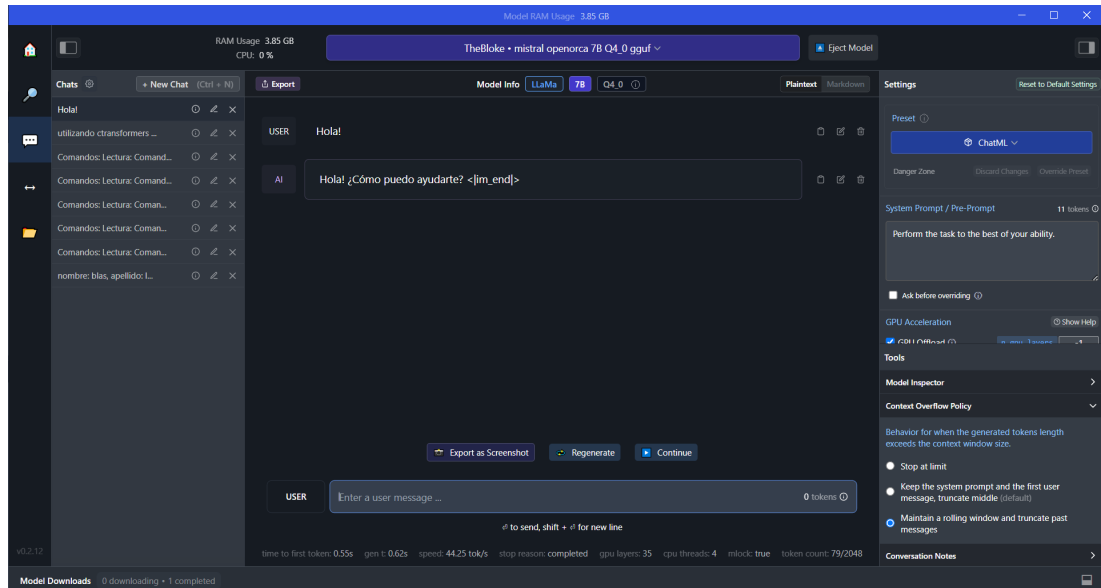


Figura 2.2: Pantalla de chat de LM Studio

2.5 Librerías

2.5.1 PyTorch

PyTorch [17] es un marco de aprendizaje profundo de código abierto basado en software que permite construir redes neuronales profundas de una manera flexible y dinámica, combinando la biblioteca de aprendizaje automático back-end de Torch con una API de alto nivel basada en Python.

2.5.2 Transformers

Esta librería proporcionada por Huggingface [18], ofrece APIs y herramientas para descargar y entrenar fácilmente modelos de lenguaje preentrenados. Entre las funcionalidades que ofrece podemos encontrar:

- Procesamiento del Lenguaje Natural: clasificación de texto, respuesta a preguntas, resumen, traducción y generación de texto.
- Visión por Computador: clasificación de imágenes, detección de objetos y segmentación.
- Audio: reconocimiento automático de voz y clasificación de audio
- Multimodal: respuesta a preguntas sobre tablas, reconocimiento óptico de caracteres,

extracción de información de documentos escaneados, clasificación de vídeo y respuesta a preguntas visuales.

Esta librería está diseñada para funcionar con los tres principales frameworks de aprendizaje profundo: PyTorch [17], TensorFlow y JAX. Esto aporta mucha flexibilidad a la hora de utilizar un marco de trabajo diferente en diferentes etapas de la vida de un modelo. Por ejemplo, entrenando un modelo utilizando PyTorch, pero después hacer la inferencia con TensorFlow.

Transformers también cuenta con una gran comunidad de desarrolladores que constantemente contribuyen a su crecimiento con nuevos proyectos y avances. En la misma página de Huggingface [18] se pueden encontrar más de 600.000 modelos preentrenados proporcionados por la comunidad. Cada uno tiene una aplicación diferente, algunos son más generales, y otros funcionan muy bien en tareas específicas, pero son mediocres en el resto de casos.

2.5.3 Flask

Microframework para Python que se utiliza principalmente para el desarrollo de aplicaciones web y servicios API [19]. Su diseño minimalista facilita y acelera el proceso de creación de APIs en comparación a otras librerías como Django, que es la más conocida para Python.

2.5.4 Requests

Librería de Python que simplifica el envío de solicitudes HTTP, permitiendo interactuar con servicios web de manera sencilla [20]. Además, maneja automáticamente la codificación de URL y la decodificación de respuestas JSON, lo que la convierte en una herramienta potente y fácil de usar para trabajar con APIs

2.6 Otras tecnologías

2.6.1 Git

Git [21] es un software de control de versiones distribuido que permite a los desarrolladores rastrear los cambios en el código fuente durante el desarrollo de software. Cada copia de un repositorio Git es completa y contiene toda la historia del proyecto, facilitando el trabajo tanto online como offline. Destaca por su capacidad para crear y fusionar ramas, permitiendo el desarrollo paralelo y la colaboración entre múltiples desarrolladores sin conflictos. Git fue creado por Linus Torvalds en 2005.

2.6.2 GitHub Desktop

GitHub Desktop [22] es una aplicación que permite interactuar con GitHub a través de una interfaz gráfica en lugar de usar la línea de comandos.

2.6.3 Visual Studio Code

Entorno de trabajo utilizado en las fases de pruebas del proyecto [23]. Su facilidad de uso y extensiones de Python agilizaron el proceso de trabajo en gran medida.

2.6.4 Lucidchart

Aplicación de diagramación inteligente basada en la web, que permite a los usuarios crear diagramas y gráficos de flujo [24]. Fue utilizada en este proyecto para realizar varias de las figuras que podemos encontrar en esta memoria.

2.6.5 Docker Compose

Herramienta que permite definir y ejecutar aplicaciones multi-contenedor en Docker [25]. Utiliza un archivo de configuración, generalmente en formato YAML, denominado docker-compose.yml, donde se especifican los servicios, redes y volúmenes necesarios para la aplicación. Esta herramienta se utilizó en el proyecto para poder desplegar el servidor LLM y configurar la exposición de su API al exterior.

2.6.6 Jmeter

Apache JMeter [26] es una herramienta de software de código abierto diseñada para realizar pruebas de rendimiento y carga en aplicaciones web y otros servicios. Permite simular múltiples usuarios concurrentes que interactúan con una aplicación, midiendo su rendimiento bajo diferentes condiciones. Utilizamos esta herramienta para realizar diferentes pruebas de carga sobre la APIs.

Análisis y metodología

ESTE capítulo contiene un análisis de las metodologías empleadas en la creación del software. También incluye una breve explicación de los tipos de metodologías que se contemplaron durante la planificación del proyecto. Además, se proporciona una explicación detallada de los casos de uso que componen el proyecto.

3.1 Metodología de desarrollo

En esta sección analizaremos el concepto de metodologías de desarrollo y sus diferentes tipos. Posteriormente, veremos la metodología escogida para este trabajo.

3.1.1 Tipos de metodologías de desarrollo

Una metodología de desarrollo es un conjunto de técnicas y métodos utilizados en el proceso de diseño de un software. Proporcionan un marco estructurado que ayuda a los equipos de desarrollo a planificar, diseñar, implementar y probar software de manera organizada y eficiente.

Podemos dividir estas metodologías en dos grandes grupos: Ágiles y tradicionales:

- Las metodologías tradicionales, también conocidas como metodologías "predictivas" o "en cascada", se basan en un enfoque secuencial y estructurado para el desarrollo de software. Cada fase del proyecto debe completarse antes de que comience la siguiente.
- Las metodologías ágiles son un conjunto de principios y prácticas que se enfocan en la flexibilidad, la colaboración y la entrega incremental. En lugar de un enfoque secuencial, las metodologías ágiles promueven iteraciones cortas y frecuentes, donde en cada iteración podremos obtener una versión funcional del software.

Finalmente, se decidió ejecutar una metodología ágil, ya que se adecuaba más al tipo de proyecto que se quería conseguir, donde cada iteración podría ser un producto funcional.

3.1.2 Metodología empleada

SCRUM es una metodología ágil desarrollada en los años 90 y es ampliamente utilizada en el desarrollo de software.

La metodología adoptada para este proyecto se fundamenta en los principios de SCRUM:

- Se realizarán reuniones diarias al principio del proyecto, para coordinar los avances del equipo y analizar los objetivos futuros. En estas reuniones participa Juan, que actúa como consultor del proyecto, y también otros integrantes de SREC Solutions, así como el propio estudiante. En estas reuniones se utilizaba Trello, una herramienta muy útil para tener organizadas las tareas.

Las reuniones seguían siempre la misma estructura, en la que cada participante tenía su turno de palabra:

1. El participante expone los avances hechos desde la última reunión.
 2. Si se tuvo algún problema a la hora de conseguir los objetivos, explicarlo y compartir las diferentes soluciones al problema
 3. El resto de participantes podrán exponer sus opiniones y consejos, y también ayudar con soluciones a los problemas que se presentan.
- Se tendrán reuniones con menor frecuencia con el cotutor del proyecto, Carlos Vázquez, con la función de hacer un seguimiento del proyecto y asegurar el cumplimiento de los plazos establecidos. También se centrarán más adelante en la correcta documentación del trabajo.
 - En el *backlog* de Trello se pondrán todas las tareas realizadas y por hacer. Se dividirán en los siguientes grupos:
 - Tareas a realizar en los próximos 3 días
 - Tareas a realizar en la próxima semana
 - Tareas a largo plazo (1-3 meses)
 - Tareas terminadas

Dentro de las tareas no terminadas también se pueden categorizar como: No empezada, en proceso o en pruebas.

3.2 Análisis de requisitos

Especificaremos los requisitos funcionales y no funcionales del sistema, que deben ser cumplidos para satisfacer las expectativas de nuestro proyecto.

3.2.1 Requisitos funcionales

Los requisitos funcionales describen lo que el sistema debe hacer. Estos requisitos detallan las funciones y características específicas que el sistema debe proporcionar. Nuestro sistema debe cumplir:

- RF-1 La aplicación será capaz de obtener una respuesta válida a partir de la salida del LLM, utilizando un procesamiento del texto. Es decir, la aplicación podrá detectar y separar las partes relevantes del texto generado por el modelo.
- RF-2 La aplicación dará la opción de cargar y descargar el LLM del sistema.
- RF-3 La aplicación se integrará con el sistema de Comunicaciones Automatizadas SINVAD a través de llamadas tipo REST.
- RF-4 El modelo será capaz de comprender el lenguaje natural, ofreciendo resultados correctos en las diferentes funcionalidades propuestas, que se detallarán en la siguiente sección (casos de uso 3.3).

3.2.2 Requisitos no funcionales

Los requisitos no funcionales describen cómo el sistema debe comportarse y abarcan aspectos de calidad y restricciones del sistema. Los principales son:

- RNF-1 Se debe utilizar un modelo de lenguaje libre.
- RNF-2 La aplicación deberá correr en hardware abierto
- RNF-3 El LLM deberá generar respuestas de forma suficientemente rápida como para ofrecer una buena experiencia al usuario.
- RNF-4 La aplicación ha de ser intuitiva, para que pueda ser utilizada por un trabajador sin ningún tipo de formación sobre el funcionamiento interno del sistema.
- RNF-5 El modelo será capaz de comprender el lenguaje natural, proporcionando fluidez en la interacción con el usuario y simulando una conversación con una persona real.

3.3 Casos de uso

Los modelos de lenguaje de gran tamaño pueden ser utilizados para una amplia gama de aplicaciones en diversos campos. Para este sistema se propusieron una serie de funcionalidades que fueran útiles en los objetivos de la empresa SREC Solutions.

Estos casos de uso se centran, en su mayoría, en reconocer y entender información dentro de un texto en lenguaje natural. De esta forma, se dará al usuario una sensación de conversación con un asistente real. Procesos en los que fuera necesario que el usuario conociera palabras clave, o que entendiera cómo funciona el sistema, ahora se hacen de forma natural, como si se estuviera hablando con otra persona.

En este trabajo se han estudiado cinco casos de usos diferentes (aunque se han evaluado varios más): *Comandos*, *Traducción*, *Lista*, *Extracción de datos* y *Gestión de diálogo*.

3.3.1 Comandos

La funcionalidad del modelo en este caso es que, a partir de una serie de elementos seguidos de una pequeña descripción, debe ser capaz de deducir a cuál de esos elementos está haciendo referencia un input, que sería una palabra o un texto corto en lenguaje natural. Para entender mejor esta funcionalidad, pongamos un ejemplo:

Tenemos una serie de comandos (con su descripción) que se podrían utilizar en cualquier empresa:

- Salida: Comando para finalizar la jornada
- Acceso: Comando para el inicio de la jornada
- Datos: Comando para añadir datos en una base de datos
- Reunión: Comando para solicitar una reunión de grupo
- Vacaciones: Comando para solicitar días de vacaciones
- Default: No se asemeja a ninguno de los comandos anteriores

Nuestro sistema debería ser capaz de, a partir de un texto como "Me voy a casa", deducir que el comando indicado es "Salida". O a partir de "Quiero unos días libres", deducir que el comando al que se refiere es "Vacaciones".

Anteriormente, este proceso de reconocimiento de comandos se realizaba mediante técnicas de extracción de información más "tradicionales". Esto obligaba al usuario a conocer las palabras clave específicas que activaban cada comando. Aunque el sistema permitía diferentes maneras de activar los comandos, admitiendo algunos sinónimos y textos alternativos, estaba limitado a las opciones que se habían codificado previamente. Al utilizar un LLM, el usuario no necesita ningún conocimiento previo para utilizar el sistema. Además, el modelo es capaz de reconocer comandos, incluso si el usuario comete errores ortográficos.

3.3.2 Traducción

La funcionalidad en este caso sería más simple: la traducción automática de textos cortos utilizando un LLM. La ventaja de esta traducción sobre la que ofrecen traductores "tradicionales", como el traductor de Google o el de Cambridge, es que no realiza traducciones literales, tiene en cuenta el contexto. Además, este nuevo traductor también tiene en cuenta posibles errores gramaticales que pudiera tener el usuario. Por ejemplo, al traducir al inglés una frase como "Él ir a la tienda ayer", un traductor normal devolvería "He go to the store yesterday", ya que traduce palabra a palabra sin tener en cuenta el contexto. En cambio, el nuevo traductor devuelve "He went to the store yesterday", que es la forma correcta.

Esta traducción también puede ser utilizada, en conjunto con el resto de casos de uso, para internacionalizar nuestro sistema. Las peticiones que estén en otro idioma, serían traducidas antes de ser procesadas, permitiendo tratarlas de igual forma que las peticiones en español, sin necesidad de hacer cambios en la implementación.

3.3.3 Lista

A partir de un texto de entrada, detectar los elementos separables del mismo, devolviendo una lista de los elementos encontrados. Como ejemplo, podemos poner el siguiente caso:

El texto de entrada sería: *"Las herramientas que vamos a necesitar son una llave inglesa, un martillo, un metro y tres placas de madera"*. El modelo devolvería la lista de *strings*: ["llave inglesa", "martillo", "metro", "tres placas de madera"]

3.3.4 Extracción de datos

A partir de un texto de entrada, extraer una serie de datos preestablecidos, devolviéndolos con un formato específico. Por ejemplo, queremos extraer datos en un proceso de una empresa constructora. Los datos podrían ser: Medida de altura del edificio, número de ventanas, material de las paredes, estado de la construcción (no empezada, en proceso, finalizada) y número de constructores asignados. La entrada del usuario podría ser: *"Aún no empezamos a construir, pero ya hay 23 constructores. El edificio será de unos 30 metros."*

El modelo devolvería algo como:

- *Altura: 30m*
- *Num. ventanas: N/A*
- *Material: N/A*
- *Estado: no empezada*
- *Num. constructores: 23*

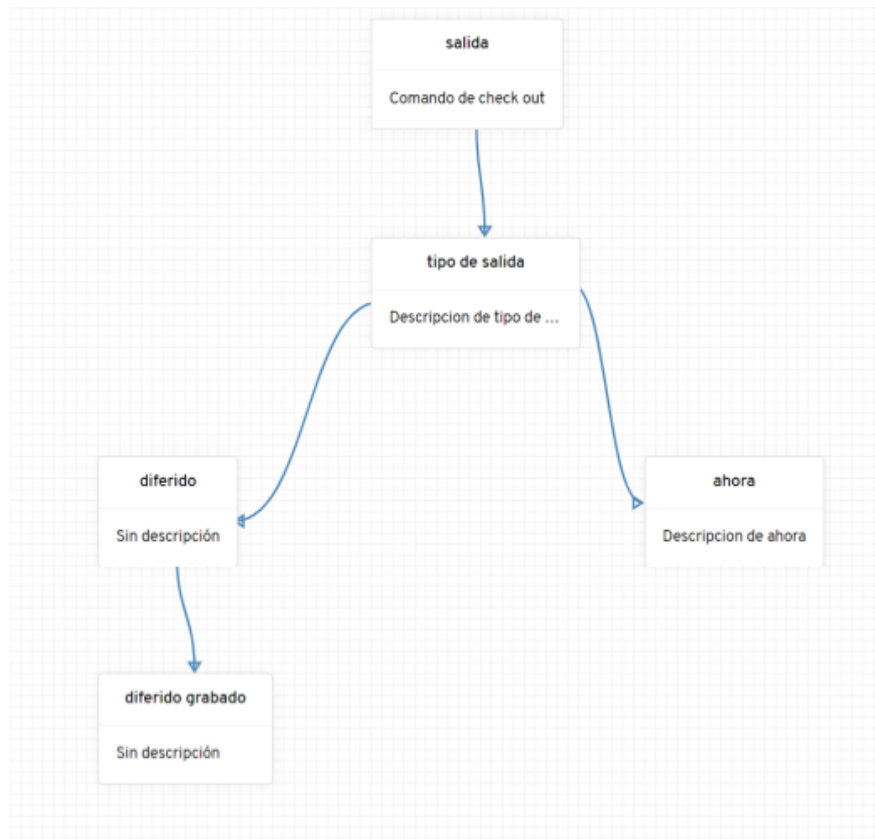


Figura 3.1: Diagrama de flujo de SIN VAD

3.3.5 Gestión de diálogo

El chatbot SIN VAD, de la empresa SREC Solutions, es un asistente inteligente que tiene la función de recabar información del usuario. Este proceso se lleva a cabo mediante un flujo de trabajo donde se van configurando una serie de pasos que definen el comportamiento del sistema. En cada iteración del flujo, el asistente pregunta al usuario por un dato. Dependiendo de la respuesta del usuario, el bot dirigirá la conversación hacia nodos diferentes y almacenará la información proporcionada. El sistema no entiende el lenguaje natural, por lo que el usuario debe responder a las preguntas con un formato específico. En la figura 3.1 podemos ver una representación de este proceso.

Para que este proceso se haga de una manera más ágil e intuitiva, se propusieron varias modificaciones:

1. El chatbot será capaz de reconocer datos que no se corresponden al paso actual. Entonces, el usuario podrá introducir más de un dato en cada iteración, y también podrá modificar datos que se indicaron en iteraciones anteriores.

2. Si el usuario añade un dato de una iteración posterior a la actual, cuando se llegue en el flujo de datos a ese paso, se pasará al siguiente, sin pedir al usuario los datos que ya había indicado anteriormente.
3. El chatbot devolverá respuestas más naturales, simulando una conversación natural.
4. Si el usuario no indica los datos que se le piden en un paso, el chatbot le devolverá una respuesta explicando cuál es el dato que se estaba pidiendo y la forma de expresarlo correctamente.

Pongamos un ejemplo de una posible conversación usuario/asistente, en la que se tienen que obtener los datos del punto anterior (3.3.4):

ASISTENTE: *¿Cuál es la altura del edificio?*

USUARIO: *39 metros, tiene también 12 ventanas*

ASISTENTE: *Vale, he registrado también el número de ventanas. ¿Cuál es el material de las paredes?*

USUARIO: *No me acuerdo, déjame pensar.*

ASISTENTE: *Claro, tómate tu tiempo. ¿Cuál es el material de las paredes?*

USUARIO: *Mármol. La altura era de 41 metros, no 39.*

ASISTENTE: *Gracias por la información. Ahora, dígame el estado de la construcción. ¿Está empezada, en proceso o ya finalizada?*

USUARIO: *Somos 8 trabajando y aún estamos en ello.*

ASISTENTE: *Aquí están los datos recopilados:*

- *Altura: 41*
- *Num. ventanas: 12*
- *Material: Mármol*
- *Estado: en proceso*
- *Num. constructores: 8*

Este caso de uso es más completo que el anterior, pero también más complejo, ya que se tienen que realizar varias funciones al mismo tiempo. Primero, es necesario identificar los datos que se deben obtener para hacer las preguntas adecuadas. También se debe llevar un registro de los datos que el usuario va proporcionando para mostrarlos al final de la conversación. Además, se debe poder comprender las respuestas del usuario para completar los datos correctamente y hacer las preguntas que corresponda en cada paso de la conversación.

Planificación y costes

ESTE capítulo aborda todos los aspectos relacionados con la planificación del proyecto. Una buena planificación es esencial para evitar imprevistos que puedan afectar al éxito del proyecto. Se mostrarán también en esta sección la identificación de riesgos, la creación de planes de contingencia y el seguimiento del progreso.

4.1 Planificación

La planificación de un proyecto es el proceso de definir los objetivos, estrategias y pasos necesarios para alcanzar esos objetivos dentro de un marco de tiempo específico. Al inicio del proyecto, no fue posible realizar una planificación completa de los objetivos debido al desconocimiento del rendimiento y resultados que se obtendrían en las primeras etapas. Por lo tanto, dividiremos la planificación en diferentes fases, que fueron naciendo a partir de los resultados obtenidos en fases anteriores. Podemos observar esta planificación final en el diagrama de la figura 4.1.

4.1.1 Elección del modelo

En esta parte del proyecto se hará un estudio de las capacidades de los diferentes modelos libres que se pueden encontrar en la página de huggingface [18]. También se probarán los modelos elegidos en algunos ejemplos de uso, para escoger el que tenga mejores resultados.

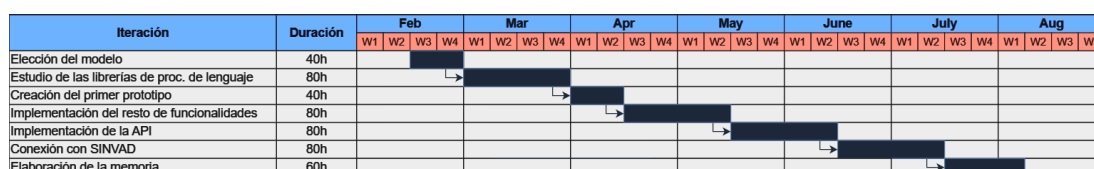


Figura 4.1: Diagrama de Gantt con la planificación del proyecto

Esta parte se realizará con LM Studio (ver apartado 2.4.1).

4.1.2 Estudio de las librerías de procesamiento de lenguaje

Estudio y familiarización con las librerías de transformers y PyTorch. En esta fase se instalarán las librerías y todas sus dependencias. También se investigará cómo interactuar con el modelo para obtener respuestas ante un input de entrada.

4.1.3 Creación del primer prototipo

Con los conocimientos obtenidos en la iteración anterior, se creará el primer prototipo de asistente inteligente. Realizará en primera instancia las tareas descritas en el caso de uso *Comandos* (ver sección 3.3.1), y más adelante se añadirán los demás casos de uso en el siguiente orden:

- Traducción
- Lista
- Extracción de datos
- Gestión de diálogo

4.1.4 Implementación de la API

Se implementará con Flask [19] un primer prototipo de API unificada con todas las funcionalidades propuestas, así como la API con las funciones del modelo de lenguaje.

4.1.5 Conexión con SINVAD

El servidor que alberga el LLM será expuesto al exterior para permitir su acceso desde SINVAD. Para lograr esto, se empleará Docker-Compose para la orquestación y gestión de los contenedores necesarios. Además, se implementará un mecanismo de control de acceso con autenticación para garantizar la seguridad y proteger el servidor contra accesos no autorizados.

4.2 Riesgos y planes de contingencias

En este apartado estudiaremos los posibles riesgos del proyecto, analizando tanto sus probabilidades de ocurrencia como su posible impacto. Posteriormente, se especificarán los planes de contingencia ante esos riesgos.

4.2.1 Riesgos

- **Modificación de requisitos:** Dado que se trata de un proyecto en colaboración con una empresa y que la planificación no estaba definida en un primer momento, es muy probable que surjan muchas modificaciones en el camino a seguir durante el proceso.
- **Modelos demasiado pesados:** Los [LLMs](#), en general, requieren una gran cantidad de VRAM, y cuanto mejor sea el modelo, más probabilidades habrá de que la memoria no sea suficiente. Un modelo considerado "pequeño" puede llegar a necesitar hasta 20GB de VRAM, lo que requeriría de tarjetas gráficas muy potentes y costosas.
- **Uso de librerías sin experiencia previa:** Para la implementación del sistema es necesario utilizar librerías con las cuales el programador no tenía experiencia previa.
- **Dedicación del equipo de desarrollo:** El analista y programador del proyecto es un estudiante que durante las fases de desarrollo del proyecto tendrá clases, trabajos y exámenes. Esto puede quitar tiempo de trabajo y provocar que la dedicación no sea óptima.
- **Implementación de una API:** El programador no tiene conocimientos previos sobre la creación de [APIs](#), ni sobre librerías para la creación de [APIs](#) en Python.
- **Exposición de la API al exterior:** El estudiante no conoce la herramienta Docker Compose, y no tiene experiencia en la configuración de contenedores para manejar la exposición segura de [APIs](#). Esto puede llevar a una configuración inadecuada, que retrase el avance del proyecto o que exponga la [API](#) a riesgos de seguridad.

4.2.2 Planes de contingencia

- **Modificación de requisitos:** Realizar varias reuniones semanalmente para informar a los consultores de los cambios realizados. De esta manera, se podrán definir las modificaciones necesarias lo antes posible.
- **Modelos demasiado pesados:** Proporcionar al programador equipos con tarjetas gráficas de tamaño y rendimiento suficientes para poder utilizar los modelos.
- **Uso de librerías sin experiencia previa:** Hacer una buena investigación previa y un estudio exhaustivo de dichas librerías. De esta manera se evitarán problemas futuros al utilizar sus funciones.
- **Dedicación del equipo de desarrollo:** En el caso de que surja un fallo en la planificación, será necesario reajustar las fechas del proyecto. Esto se llevará a cabo durante una de las reuniones mencionadas anteriormente.

- **Implementación de una API:** El estudiante contará con el apoyo de algunos de los consultores, que le ayudarán a realizar esta parte del trabajo. Además, tendrá que realizar un estudio previo sobre las librerías de Python necesarias.
- **Exposición de la API al exterior:** El estudiante realizará un estudio previo del funcionamiento de Docker Compose y sus medidas de seguridad. Además, contará con la ayuda de algunos consultores con mayor experiencia utilizando esta herramienta.

4.3 Seguimiento de la planificación

En esta sección podemos encontrar todo el seguimiento del proyecto, incluyendo los problemas que fueron surgiendo durante el proceso.

4.3.1 Elección del modelo

El modelo a utilizar era un paso primordial para la ejecución del proyecto, ya que todo el funcionamiento del sistema dependía de la eficacia de dicho modelo.

Primero, se utilizó el *Open LLM Leaderboard* [7] para tener algunas nociones generales del estado del arte en cuanto al rendimiento de los LLMs, antes de pasar a la fase de pruebas.

Para facilitar y agilizar las pruebas, se empleó la aplicación *LM Studio*, que permite descargar y usar modelos libres. Su interfaz integra un chatbot que requiere muy poca configuración, permitiendo interactuar con el LLM de forma inmediata.

Los modelos que se tuvieron en consideración para las pruebas fueron: Mistral 7B, Microsoft Phi-2, Meta Llama 2 y Google Bert Base (ver tabla 2.2). Finalmente, se escogió Mistral 7B.

4.3.2 Estudio de las librerías de procesamiento de lenguaje

Para esta fase hubo un período de investigación sobre las librerías de manejo de LLMs. La librería Transformers (ver sección 2.5.2) es la que más libertad permite a la hora de utilizar cualquier tipo de modelo libre, ya que es una librería proporcionada por huggingface [18], donde están actualmente la mayoría de modelos libres disponibles. Transformers emplea una librería de desarrollo e implementación de aprendizaje profundo para realizar la inferencia. Permite utilizar PyTorch [17] o TensorFlow. Se decidió utilizar PyTorch, ya que, aunque la eficacia de ambas librerías resulta similar en la mayoría de casos, PyTorch tiene una API más clara y bien diseñada que es fácil de aprender. Además, tiene una comunidad muy activa, proporcionando un soporte robusto a través de foros, repositorios de GitHub, y otros canales comunitarios.

Tras un estudio de los métodos y funciones de Transformers, se obtuvo un primer asistente que era capaz de devolver una respuesta estructurada y coherente ante un input de texto.

4.3.3 Creación del primer prototipo

Con los conocimientos obtenidos en el paso anterior, se realizó un primer prototipo de sistema, para que realizara la función detallada en el primer caso de uso (ver sección 3.3.1). Las primeras pruebas se realizaron con el modelo *Phi-2*, que es más pequeño que *Mistral 7B*, debido a que el equipo de la universidad aún no estaba disponible y el portátil carecía de suficiente capacidad de VRAM. Utilizar otro modelo de menor tamaño provocó que se obtuvieran resultados poco alentadores, lo que condujo a empezar a considerar realizar un entrenamiento supervisado al modelo.

Entrenar el modelo no sería adecuado para nuestro sistema. Por ejemplo, para el caso de uso de *Comandos* (ver sección 3.3.1), la necesidad de utilizar un modelo entrenado requeriría obtener una gran cantidad de datos sobre posibles inputs de los empleados, y generar un dataset con el que se pudiera entrenar el LLM. Dado que cada empresa utiliza comandos diferentes, sería necesario entrenar un modelo nuevo para cada cliente, algo que sería inviable a gran escala.

Cuando finalmente se tuvo acceso al equipo de la universidad, con sus 24GB de VRAM, se hicieron las mismas pruebas, pero con *Mistral 7B*. El modelo dio un resultado mucho mejor, descartándose la posibilidad de realizar un entrenamiento y continuando con la planificación inicial.

Algo muy importante también para mejorar los resultados obtenidos es la creación de buenos *prompts*. Para que el modelo entienda correctamente la función que debe realizar, es importante especificar todos los detalles sobre la forma y el contenido del texto que va a generar. Por ello, la creación de un buen *prompt* fue una de las tareas más importantes de este proyecto. Se realizaron múltiples pruebas para cada caso de uso, y lo más importante en estas pruebas fue ser capaz de interpretar las razones por las cuales los resultados no eran los esperados, y actuar en consecuencia. De esta forma, se pudo ir comprendiendo cómo interactuar con el LLM para obtener los resultados deseados.

Una página que resultó ser de gran ayuda a la hora de crear buenos *prompts* fue la propia de Mistral [5], que describe muy bien cómo obtener el mayor beneficio del modelo para diferentes funcionalidades.

4.3.4 Implementación de la API

Esta fase de la planificación comenzó con un estudio de las opciones disponibles para la creación de aplicaciones web utilizando Python, ya que el estudiante no tenía conocimientos previos en el tema. Se utilizó finalmente la librería Flask.

En primera instancia, se realizó un primer prototipo de la API con los diferentes servicios de cada caso de uso propuesto. Esta aplicación combinaba la creación del **prompt** a partir del input, la generación de texto y el procesamiento del texto generado. La aplicación recibía un input y devolvía la respuesta sin hacer ninguna llamada adicional.

Más adelante, la empresa propuso otro enfoque para la aplicación: La generación de texto iría separada de la creación del **prompt** y del procesamiento de la salida del **LLM**. Por lo tanto, el servidor que contiene el modelo pasó a encargarse únicamente de recibir una entrada, proporcionársela al modelo para que genere el texto, y devolver dicho texto. El resto de procesamiento ahora es responsabilidad de la aplicación integrada en **SINVAD**, que, dependiendo de la funcionalidad que se esté ejecutando, envía una entrada diferente al servidor. También es la que recibe y procesa la salida del modelo, de forma diferente en cada caso de uso.

Este cambio de enfoque facilita la introducción de nuevas funcionalidades al sistema, al no tener que cambiar la implementación del servidor con cada modificación que se quiera hacer. Solo sería necesario cambiar la parte de la aplicación que procesa el texto.

4.3.5 Conexión con **SINVAD**

Para integrar el sistema con **SINVAD**, primero fue necesario instalar el modelo e introducir la API en el nuevo servidor de SREC Solutions.

El alumno visitó a las oficinas de la empresa para hacer la instalación y puesta en marcha de todos los requisitos necesarios para lanzar la **API** en el servidor de SREC Solutions [8]. La visita incluyó las siguientes tareas:

1. Instalación del sistema operativo en el servidor. Se optó por instalar Ubuntu, versión 22.4, tras un breve estudio de opciones se concluyó que esta era la mejor alternativa. Las versiones de Ubuntu LTS (Long term support) proporcionan un mayor soporte para el uso de las GPU de NVIDIA [27].
2. Instalación de las librerías necesarias en el servidor (PyTorch, Transformers, Flask, Requests).
3. Instalación de *Mistral 7B* en el equipo.
4. Configuración del acceso SSH a través de Cloudflare para permitir el trabajo remoto del estudiante.

En los días posteriores a la visita a SREC Solutions, el estudiante pudo realizar las siguientes tareas tanto de forma presencial como remota, lo que proporcionó mucha comodidad y agilizó el resto del proyecto.

Posteriormente, se integró la API del servidor en un contenedor utilizando la herramienta Docker Compose [25]. Para ello, se implementaron los archivos *docker-compose.yml* y *Dockerfile* correspondientes.

Finalmente, se expuso la API al exterior utilizando *Cloudflare Tunnel*, creando una conexión segura sin necesidad de exponer directamente la IP del servidor. Para garantizar la seguridad del servicio, se añadió un control de autenticación en el código de la API, que comprueba las credenciales del usuario (incluidas en los *headers* de la solicitud) antes de devolver una respuesta. Además, se realizaron varias pruebas de carga al servidor, para comprobar los tiempos de respuesta del mismo ante varias solicitudes simultáneas.

4.4 Recursos

En esta sección describiremos los recursos utilizados durante el transcurso del proyecto. Los podemos dividir en recursos humanos, materiales y software.

4.4.1 Recursos humanos

Describiremos todos los roles desempeñados por los participantes del proyecto.

- **Jefe del proyecto:** Desempeñado por el propio estudiante. Se encarga de la dirección del proyecto y es el último responsable en las decisiones a tomar.
- **Consultores:** Desempeñado principalmente por Juan Antonio Martín Pernas, uno de los codirectores del proyecto. Perteneció a la empresa SREC Solutions, por lo que pudo aportar muchas facilidades y conocimientos, funcionando como nexo de unión de la empresa con el proyecto. También lideraba las reuniones periódicas que podemos encontrar en la sección 3.1.2 de metodología. También participaron en este rol otros integrantes de SREC Solutions.
- **Tutor académico:** Desempeñado por Carlos Vázquez Regueiro. Forma parte del cuerpo docente de la universidad, y se encarga de orientar al estudiante para evitar desviaciones entre los requisitos de la universidad y la empresa. Tuvo un papel fundamental en la revisión de la documentación del proyecto.
- **Analista:** Desempeñado por el estudiante, encargado de realizar las tareas de análisis y diseño. Establece las necesidades del proyecto junto con los consultores para definir las tareas a realizar.
- **Programador:** Desempeñado también por el estudiante. Se encarga de las partes de implementación y test del proyecto.

4.4.2 Recursos materiales

Con respecto a los recursos materiales, encontramos los siguientes:

- **Ordenador portátil:** Elemento principal del proyecto, donde se realizaron las primeras pruebas de implementación. También se utilizó para la conexión a través de SSH con los equipos más potentes en pruebas posteriores. El modelo es MSI GF65 con sistema operativo Windows, con una tarjeta gráfica NVIDIA GeForce RTX 2060.
- **Servidor UDC:** Equipo donde se pudieron realizar las primeras pruebas con modelos de mayor tamaño. El modelo cuenta con un procesador *Ryzen Threadripper 3970X* y una gráfica NVIDIA GeForce RTX 3090, con 24GB de VRAM.
- **Servidor SREC:** Equipo utilizado para integrar el [LLM](#) en la [API](#) de SREC Solutions. Cuenta con una gráfica algo superior a la del equipo de la universidad, la NVIDIA GeForce RTX 4090.

4.4.3 Recursos software

Los recursos software empleados (más detalles en el capítulo 2, a partir de la sección 2.2) fueron los siguientes:

- LM Studio
- Visual Studio Code
- Python 3.10
- Librerías de uso de [LLM](#) (PyTorch, Transformers...)
- Git
- Lucidchart
- Docker Compose
- Apache Jmeter
- Modelos de lenguaje

Estos recursos no implicaron ningún coste, ya que se utilizó software de código abierto en todos los casos.

Recurso	Salario (€/h)	Trabajo (h)	Coste (€)
Jefe del proyecto	50€	60h	3.000€
Tutor / Consultor	50€	20h	1.000€
Analista	40€	120h	4.800€
Programador	30€	260h	7.800€
Total	-	460h	16.600€

Tabla 4.1: Costes de los recursos humanos

Recurso	Coste imputable (€)
Portátil MSI	15€
Servidor UDC	18€
Servidor SREC	68€
Total	95€

Tabla 4.2: Costes de los recursos materiales

4.5 Estimación de costes

En la estimación de costes se encuentran tanto los recursos humanos como los materiales (ver tabla 4.3). No se incluyeron los gastos por recursos de software, ya que se utilizaron exclusivamente herramientas *open source*, sin ningún coste adicional.

4.5.1 Costes de los recursos humanos

Como se puede ver en la tabla 4.1, en la estimación de costes por recursos humanos se tuvo en cuenta cada participante del proyecto. Para realizar esta estimación, se calculó el tiempo de trabajo en el proyecto de los participantes. También se le asignó un coste por hora trabajada a cada uno.

4.5.2 Costes de los recursos materiales

En la tabla 4.2 encontramos la estimación de los costes de recursos materiales, en la que se tendrá en cuenta el coste inicial de cada elemento, así como su amortización. El portátil tuvo un coste inicial de alrededor de 1.000€; el servidor UDC, de alrededor 4.000€, y el servidor SREC, de alrededor de 8.000€. Teniendo en cuenta el tiempo de uso de cada uno durante el proyecto, junto con el tiempo de amortización de cada uno, obtenemos su coste final imputable.

Recurso	Coste (€)
Humanos	16.600€
Materiales	95€
Total	16.695€

Tabla 4.3: Coste total del proyecto

Arquitectura y diseño

EN este capítulo se realiza un análisis de la arquitectura de nuestro sistema. Se ofrece una visión general de la misma y se describe cada una de sus partes. Además, se presentan las vistas que conforman la interfaz de nuestra aplicación

5.1 Arquitectura general de la aplicación

Nuestro sistema está diseñado para ser un intermediario entre el usuario final y un asistente virtual o sistema de comunicaciones automatizadas (SINVAD) empleando las enormes capacidades de un modelo de lenguaje natural (LLM).

En primer lugar, debe recolectar los datos del usuario (la entrada) y realizar los cambios necesarios para poder presentar dicha entrada al modelo de comprensión de lenguaje. A continuación, debe recoger la respuesta del modelo y realizar los cambios pertinentes para poder enviársela al usuario. Es SINVAD quien, en función de cada caso de uso, define cómo presentar la entrada al modelo y cómo procesar su respuesta. Gracias a nuestro sistema, el usuario puede llegar a tener una interacción mucho más flexible y general con SINVAD, ya que nuestro sistema le va guiando en el proceso utilizando un LLM.

Pongamos un ejemplo de cómo nuestro sistema facilita la interacción entre SINVAD y el usuario. Una de las principales funciones de SINVAD es recopilar una serie de datos del usuario. Nuestro sistema ofrece un método para realizar esta tarea de forma mucho más flexible e intuitiva, simulando una interacción con un humano, mediante el LLM. El usuario podrá interactuar con nuestro sistema, indicando a través de texto los datos necesarios, y recibiendo con las respuestas del modelo la información que necesite durante el proceso. Una vez que se hayan reunido todos los datos necesarios, nuestro sistema los enviará a SINVAD.

En la figura 5.1 podemos encontrar un diagrama con la arquitectura general del sistema, los principales módulos que lo componen y cómo interactúan entre ellos. Nuestra aplicación se basa en la comunicación entre dos módulos. Primero, tenemos el procesador de texto, que

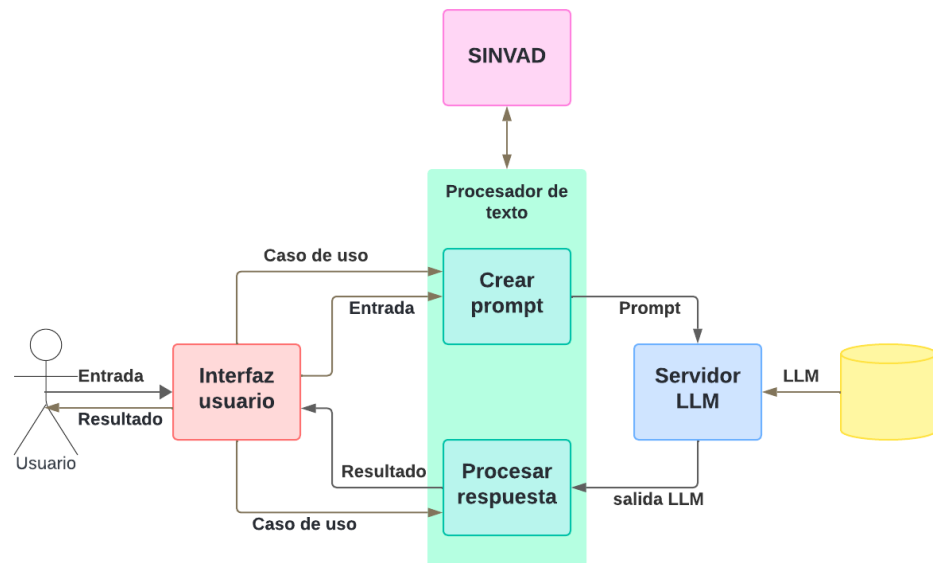


Figura 5.1: Esquema general de nuestro sistema

se encargará de construir un **prompt** a partir de la entrada del usuario. Este **prompt** se enviará a un servidor que contiene el **LLM**, para que genere nuevo texto. El servidor devolverá este nuevo texto de vuelta al procesador, que lo transformará en la salida final del sistema. Nuestro sistema consta de los siguientes elementos (figura 5.1):

- **SINVAD:** Sistema de comprensión de lenguaje desarrollado por la empresa SREC Solutions [8]. **SINVAD** ofrece una **API** con múltiples llamadas disponibles. En este punto se añadirán llamadas correspondientes a los diferentes casos de uso que tendrá nuestro sistema.
- **LLM:** El modelo obtenido desde la página de *huggingface*. Su función es generar nuevos **tokens** (texto codificado) a partir de una serie de **tokens** de contexto. Nuestro sistema puede emplear diferentes modelos, tal y como se verá en el capítulo de pruebas.
- **Servidor LLM:** Expone una API REST con la única función de utilizar el modelo de lenguaje para procesar una entrada y devolver la respuesta obtenida tras la generación. La entrada que recibe la transforma en los **tokens** que recibirá el **LLM**. Posteriormente, decodifica la salida del modelo para transformarla en texto.
- **Procesador de texto:** Este módulo se compone de dos elementos que se encargan de transformar las entradas del usuario y las salidas del Servidor LLM. A partir de la entrada del usuario, genera un **prompt** con toda la información sobre el comportamiento

que deberá tener el modelo, diferente en cada caso de uso. Después, con la respuesta recibida, genera una salida con el formato correspondiente según el caso de uso elegido.

- **Interfaz usuario:** Se encarga de la comunicación entre el usuario y el procesador de texto. Permite al usuario introducir un texto, que se enviará al procesador de texto, para posteriormente mostrar el resultado al usuario.

Actualmente, toda la interacción del usuario con nuestro sistema se realiza a través de texto, aunque en un futuro próximo se podría ampliar e incluir un módulo de traducción de voz a texto para incluir interacción por voz.

En un diseño preliminar, la arquitectura del sistema integraba las funciones del procesador y del servidor LLM en una sola aplicación, que haría el procesamiento y la generación de texto. Finalmente, se decidió sustituir este enfoque por uno más modular. Añadir más módulos tiene el inconveniente de que aumenta la complejidad y el tiempo de procesamiento, porque añade una llamada adicional al servidor LLM. Sin embargo, tiene la ventaja de que proporciona una mayor libertad a la hora de crear nuevos casos de uso o modificar los ya existentes.

La implementación del servidor LLM es siempre la misma, mientras que la aplicación que procesa el texto es la encargada de generar las diferentes entradas (**prompts**) y salidas para cada caso de uso considerado. De esta forma, no se tendrá que modificar la implementación del servidor LLM si se quiere introducir una nueva funcionalidad, o si se quiere trabajar con datos de un cliente nuevo.

Este sistema es capaz de mantener una conversación y comprender las peticiones del usuario, y al finalizar el proceso, devolver un resultado coherente con toda la información obtenida. Este es un proceso bastante complejo y difícil de abordar directamente, por este motivo se propusieron varios casos de uso que guían el desarrollo y ejecución del sistema, conduciendo finalmente al resultado deseado. Cada caso de uso nos puede ayudar en alguna parte de nuestro objetivo final. A partir de un texto de entrada del usuario, *Comandos* (apartado 3.3.1), permite deducir a que comando se puede referir este texto; *Traducción* (apartado 3.3.2), nos sirve para traducir las peticiones del usuario e internacionalizar nuestro sistema; *Lista* (apartado 3.3.3), permite obtener una lista de los elementos encontrados en el texto, y *Extracción de datos* (apartado 3.3.4), consigue obtener los datos específicos que necesitemos del texto.

A continuación comentaremos cada módulo con mayor grado de detalle.

5.2 Servidor LLM

El servidor donde se encuentra el modelo de lenguaje ofrece varios servicios relacionados con el estado y la ejecución del LLM:

- **Carga del modelo en memoria:** Si no se pudo cargar, por falta de espacio o por alguna otra razón, se indicará devolviendo un mensaje de error
- **Descarga del modelo en la memoria:** Se devolverá un mensaje de confirmación al finalizar la operación
- **Generación de respuesta:** El servidor recibirá una conversación/[prompt](#) inicial. A partir de este texto tendrá que generar nuevo texto. También se podrá especificar el número máximo de [tokens](#) nuevos a generar. Al finalizar la operación, se devolverá la respuesta generada. Si hay algún error en la petición que se recibe o algún error en la operación de generación, se devolverá un mensaje indicando el motivo de dicho error.
- **Visualización del estado del modelo:** Se devolverá un valor booleano que indica si el modelo está listo para ser usado o no (si está cargado).

Este servidor expone una API a través de la cual se pueden realizar diversas solicitudes [HTTP](#) para interactuar con el [LLM](#).

5.3 Procesador de texto

El procesador de texto tiene la función de traductor entre el usuario y el [LLM](#). Compone los [prompts](#) a partir de la entrada del usuario y compone las respuestas a partir de la salida del [LLM](#). Ofrece estos servicios:

- **Definición de los datos necesarios:** Para la mayoría de casos de uso, es necesario definir una serie de datos antes de hacer una petición. Por ejemplo, en el caso de uso *Comandos* (apartado [3.3.1](#)), es necesario indicar los comandos que se quieren utilizar, o en el caso de uso *Traducción* (apartado [3.3.2](#)), en el que se deben definir los idiomas de entrada y de salida.
- **Generación de respuesta:** Se recibe un texto (entrada del usuario) y se combina con los datos definidos en el paso anterior, para formar un [prompt](#) que se envía al Servidor LLM. A partir de la salida del servidor, se forma una respuesta. Esta respuesta se devuelve con un formato distinto según el caso de uso. Por ejemplo, para el caso de *Lista* (apartado [3.3.3](#)), se devuelve una lista de *Strings* con los elementos encontrados, mientras que para *Traducción* (apartado [3.3.2](#)), se devuelve el texto traducido.

5.4 Vistas de la aplicación

En esta sección encontramos las diferentes vistas que componen la interfaz de nuestra aplicación.

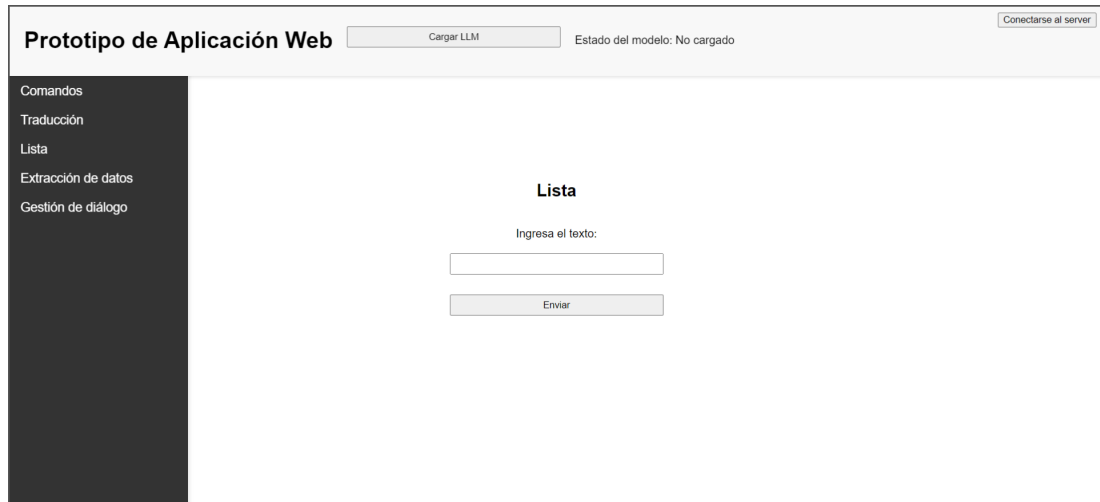


Figura 5.2: Prototipo general de interfaz de la aplicación

5.4.1 Estructura general

La interfaz de nuestra aplicación para el uso del **LLM** se compone de tres partes principales (ver figura 5.2): Una barra superior, una barra lateral y el cuerpo. A continuación se muestran los detalles de cada una de ellas.

- **Barra superior:** Se encuentra en la parte superior de la pantalla, y ofrece al usuario las opciones de conectarse al servidor y cargar el **LLM** en el mismo.
- **Barra lateral:** Se encuentra en el lateral izquierdo de la pantalla. Contiene varios elementos seleccionables para elegir entre los distintos casos de uso implementados en el proyecto.
- **Cuerpo:** Se encuentra en el centro de la pantalla e incluye todos los elementos para interactuar con el **LLM**. Alberga todas las vistas de la aplicación.

5.4.2 Prototipos de vistas

En nuestra aplicación, podemos identificar dos tipos de vistas: algunas están destinadas a definir los datos que se utilizarán en las consultas, mientras que otras sirven para realizar consultas al modelo.

- **Hacer consultas al LLM:** Dentro de este tipo de pantallas encontramos cinco vistas diferentes, una para cada caso de uso:
 - **Comandos:** El modelo de vista que vemos en la figura 5.4 permite hacer una consulta al **LLM** y ver el resultado devuelto. También se puede acceder a la definición de comandos, que se muestra a la derecha del formulario para las consultas.

Prototipo de Aplicación Web

Cargar LLM Estado del modelo: No cargado Conectarse al server

Comandos Traducción Lista Extracción de datos Gestión de diálogo

Definir comandos

Comando1 -

Comando2 -

Comando3 -

+

Definir ejemplos

Ejemplo -

Ejemplo -

+

Aceptar

Figura 5.3: Prototipo para la definición de *Comandos*

Prototipo de Aplicación Web

Cargar LLM Modelo no cargado Conectarse al server

Comandos Traducción Lista Extracción de datos Gestión de diálogo

Comandos

Definir comandos

Ingresa el texto: Comandos no definidos

Enviar

Figura 5.4: Prototipo para el caso de uso *Comandos*

Figura 5.5: Prototipo para el caso de uso *Gestión de diálogo*

- **Traducción:** Esta vista es similar al prototipo anterior, y permite hacer consultas al LLM para realizar traducciones. También se podrán seleccionar los idiomas de la traducción y ver la respuesta obtenida.
 - **Lista:** Esta vista que se muestra en la figura 5.2 permite realizar una consulta al modelo para mostrar la lista de elementos obtenida.
 - **Extracción de datos:** Este modelo es similar al presentado en la figura 5.3 pero con los datos y elementos específicos para esta funcionalidad. Permite al usuario realizar consultas al modelo para recibir los datos correspondientes a las mismas. Además, proporciona acceso a la pantalla de definición de los datos que se utilizarán en dichas consultas, los cuales se muestran a la derecha del formulario para las consultas.
 - **Gestión de diálogo:** En este caso encontramos un modelo diferente (ver figura 5.5). Se optó por un formato de chat en el que se puede enviar mensajes al bot y ver sus respuestas. También se puede acceder a la pantalla de definición de los datos que se van a utilizar en las consultas.
- **Definir datos:** Pantallas utilizadas para definir los datos que se necesitarán para cada consulta. Por ejemplo, en el caso de *Comandos* (ver sección 3.3.1), esta pantalla sirve para definir los comandos que se utilizarán para las consultas al modelo. Los casos de uso que necesitan estas vistas son *Comandos*, *Extracción de datos* y *Gestión de diálogo*. En la figura 5.3 podemos ver un prototipo para el caso de uso *Comandos*, donde se pueden ir añadiendo y eliminando los elementos necesarios.

5.4.3 Navegación en las vistas

La navegación por todas las vistas de la interfaz se realiza mediante la barra lateral, desde la que se puede acceder a las pantallas correspondientes a todos los casos de uso implementados. Desde algunas de estas pantallas, se podrá acceder a cada una de las vistas correspondientes con la definición de datos (ver *Definir datos* 5.4.2).

La barra superior está incluida en todas las vistas de la aplicación, para permitir que sus funcionalidades sean de fácil acceso.

Implementación

Este capítulo incluye todos los detalles sobre la implementación del sistema de generación de textos y de los casos de uso, junto con varios ejemplos de resultados obtenidos. También se presenta la implementación en LM Studio y de la interfaz de usuario.

6.1 Generación de texto

El proceso de generación de texto requiere seguir una serie de pasos que se detallarán a continuación.

6.1.1 Carga del modelo y el *tokenizer*

Primero, necesitamos un *tokenizer*, que se encargará de [parsear](#) el texto de entrada a *tokens*, que es con lo que trabajará el modelo. Utilizaremos el *tokenizer* de Mistral.

Para cargar el modelo concreto de LLM en memoria, se utiliza la función **AutoModelForCausalLM.from_pretrained()** (ver código 6.1). Si el modelo no está en los archivos de nuestro sistema, esta función se encargará de descargarlo desde la página de huggingface [18]. Después indicamos que el modelo esté en GPU, ya que por defecto va a CPU, con la función **model.to("cuda")**.

```
1 tokenizer = AutoTokenizer.from_pretrained(  
    "mistralai/Mistral-7B-Instruct-v0.2")  
2 model = AutoModelForCausalLM.from_pretrained(  
    "mistralai/Mistral-7B-Instruct-v0.2",  
    torch_dtype=torch.bfloat16).eval()  
3 model.to("cuda")
```

Código 6.1: Función de carga del LLM y el tokenizer

```

1 [
2   {"role": "user", "content": "Hola, ¿puedes ayudarme a
3   planificar mi día?"},
4   {"role": "assistant", "content": "¡Claro! ¿Tienes alguna tarea
5   o evento importante que ya tengas planeado?"},
6   {"role": "user", "content": "Sí, tengo una reunión a las 10 AM
7   y una cita médica a las 3 PM."},
8   {"role": "assistant", "content": "Perfecto. Entonces te sugiero
9   usar el tiempo entre la reunión y la cita médica para trabajar
10  en tus tareas pendientes o tomar un descanso. ¿Te gustaría que
11  te ayudara a organizar el resto del día?"}
12 ]

```

Código 6.2: Formato para la creación de conversaciones

6.1.2 Creación de conversaciones

Es necesario crear una conversación inicial usuario/asistente como input inicial para tener un punto de partida sobre el que continuar la generación del texto. La librería *transformers* ofrece varios métodos para ello, pero el que mejor se adapta a las necesidades de nuestro sistema es el siguiente:

Una lista de diccionarios, donde cada elemento representa una contribución a la conversación por parte del usuario o del asistente. Cada diccionario tiene el siguiente formato: `{"role": "user"/"assistant", "content": *contenido de la conversación*`

La lista deberá intercalar los roles *user* y *assistant*. Una lista que contenga dos o más elementos consecutivos con el mismo "role" no sería válida. Toda la aportación de un agente (user o assistant) debe estar en la parte de "content".

En el código 6.2 tenemos un ejemplo de una conversación simple, con el formato descrito.

6.1.3 Generación de texto a partir de una conversación inicial

Una vez cargado el modelo y el *tokenizer* y definida la conversación inicial, es hora de generar texto nuevo. Para ello, tenemos la función **gen_response()** (ver código 6.3), que recibe como entrada el input inicial del usuario (conversación) y el número máximo de *tokens* adicionales que se generarán.

Primero, se tokenizará el input del usuario con el *tokenizer* previamente cargado. Para el formato de conversación empleado, se debe usar la función **apply_chat_template()**. La opción *add_generation_prompt* indica que en la conversación se añadirá la parte del asistente. Si se marca como *False*, el texto generado sería una continuación de la aportación del usuario. Por ejemplo, si el usuario hace la consulta: "Dime tres ciudades de España". Al generar el texto se seguiría la frase del usuario, añadiendo algo como: "Dime tres ciudades de España que te

```

1 def gen_response(userInput , maxTokens):
2     with torch.no_grad():
3         tokenized_chat = tokenizer.apply_chat_template(
4             userInput ,
5             tokenize=True ,
6             add_generation_prompt=True ,
7             return_tensors='pt'
8         ).to(device)
9         outputs = model.generate(
10             tokenized_chat ,
11             pad_token_id=tokenizer.pad_token_id ,
12             max_new_tokens=maxTokens
13         )
14         res = tokenizer.decode(outputs[0])
15         ind1 = res.rfind("INST")+5
16         ind2 = res.rfind("</s>")
17         if ind2<ind1:
18             ind2 = -1
19         res = res[ind1:ind2]
20         return res

```

Código 6.3: Función de generación de texto

gustaría visitar ...". En `return_tensors` indicamos que estamos utilizando PyTorch.

La función `generate()` genera nuevos `tokens` a partir de la conversación tokenizada. El número máximo de `tokens` a generar se establece en `max_new_tokens`, pero la generación podría detenerse antes.

Para obtener el resultado de la generación como un String, hay que decodificar los `tokens` obtenidos. El método `decode()` convierte una lista de `tokens` en una lista de cadenas de caracteres.

Finalmente, se corta la cadena para obtener solo la parte generada, y para quitar también los `tokens` sobrantes ("`/INST`", que indica el final del texto de "user", y "`</s>`", que indica el final del texto de "assistant").

6.2 Implementación de los casos de uso

Para la creación de la API se utilizó la librería Flask [19]. A continuación, se describirán todos los aspectos de su desarrollo, incluyendo la implementación de los `prompts` y de cada caso de uso del proyecto. Los `prompts` específicos de cada funcionalidad no se mostrarán debido a que son información confidencial, considerada valiosa por la empresa, y se ha decidido mantenerlos en reserva por intereses comerciales.

6.2.1 Prompts

Hay 2 formas de indicar un **prompt** en nuestro sistema:

- Una conversación completa entre usuario/asistente, como una lista de JSON, con el formato:

```
[
  {"role": "user", "content": *entrada1*}
  {"role": "assistant", "content": *respuesta 1*}
  {"role": "user", "content": *entrada 2*}
  ...
]
```

- Una String que incluya toda la información necesaria para la generación. Después se convierte al formato de lista de JSON, y quedaría de la forma:

```
[{"role": "user", "content": *prompt completo*}]
```

6.2.2 Composición de respuestas

Para generar los resultados finales a partir de la salida del **LLM**, el procesador de texto tendrá que extraer las partes relevantes de esta salida.

En primer lugar, se debe separar el nuevo texto generado del resto de la salida, ya que el modelo siempre devolverá la conversación completa. Mistral tiene una serie de **tokens** especiales para separar las diferentes partes de una conversación, nos interesan los siguientes:

- `"[/INST]":` Inicio de una contribución del asistente.
- `"</s>":` Final de una contribución del asistente.

Para obtener el texto generado será suficiente con encontrar las últimas ocurrencias de estos dos **tokens** y extraer el texto que se encuentra entre ellas.

Después se debe realizar un segundo procesamiento, que será diferente para cada caso de uso. Para poner un ejemplo, en el caso de uso *Comandos* (apartado 3.3.1), las salidas deben tener el formato `"Comando: *resultado*"` y nada más. Pero la salida generada por el modelo podría añadir posteriormente más texto que no tiene relevancia para este caso, como explicaciones adicionales sobre por qué se decidió ese comando y no otro. Todo este texto adicional debe ser suprimido del resultado final. Para obtener el resultado en este ejemplo, será necesario encontrar la parte del texto `"Comando: "` y añadir la siguiente palabra.

En la implementación específica de cada caso de uso, que se mostrará a continuación, se puede ver cómo se hace esta composición de las respuestas, después de recibir la salida del servidor LLM.

```
1 def Commands_LoadChat():
2     global chatCmd
3     data = request.json
4
5     #comprobación de los datos de entrada
6     if "commands" not in data:
7         return jsonify({"status": "400", "Error": "No commands
8         provided"})
9     if "examples" not in data:
10        return jsonify({"status": "400", "Error": "No examples
11        provided"})
12    elif len(data["examples"]) % 2 != 0:
13        return jsonify({"status": "400", "Error": "Examples do not
14        have the correct format"})
15
16    #definición de la estructura de la conversación
17    chatCmd = *prompt confidencial*
18
19    #añadir los comandos y los ejemplos al prompt
20    for string in data["commands"]:
21        chatCmd[2]["content"] += '\n' + string
22    user = True
23    for string in data["examples"]:
24        if user:
25            chatCmd += [{"role": "user", "content": string}]
26        else:
27            chatCmd += [{"role": "assistant", "content": "Comando:
28            " + string}]
29        user = not user
30    return jsonify({"status": "200"})
```

Código 6.4: Función para definir la conversación para *Comandos*

```

1 def Commands():
2     data = request.json
3
4     #obtención de las entradas del usuario
5     userInput = data.get('userInput')
6     numTokens = 40
7
8     #generación del prompt con las entradas del usuario
9     prompt = *prompt confidencial*
10
11    #obtención de la respuesta del servidor
12    response = requests.post(URL+ "/genResponse",
13    json={"userInput": prompt, "numTokens": numTokens},
14    headers=headers).json()
15    if response.get("response"):
16        cmd = response["response"]
17    else:
18        return jsonify({"status": "400", "data": "error"})
19
20    #composición de la respuesta
21    ind2 = cmd.find('\n')
22    if ind2 != -1:
23        cmd = cmd[:ind2]
24    while cmd.count(' ') > 1:
25        cmd = cmd[:cmd.rfind(' ')]
26    while cmd.count(':') > 1:
27        cmd = cmd[:cmd.rfind(':')]
28    while cmd[-1] == '\n':
29        cmd = cmd[:-1]
30    return jsonify({'output': cmd})

```

Código 6.5: Función de obtención de comandos

6.2.3 Comandos

Identificación de comandos a partir de una entrada del usuario. *Commands_LoadChat()* (ver código 6.4) se encarga de definir el **prompt** (los comandos) que se utilizará en la inferencia junto con la entrada del usuario. Está compuesta por los siguientes puntos:

1. Comprobación de la validez de los datos de entrada (línea 6).
2. Definición de la estructura de la conversación (línea 14).
3. Composición de la conversación con los datos de la petición (*request*) (líneas 17).

La parte de la obtención del comando (ver código 6.5), sigue los siguientes pasos:

1. Obtención de las entradas del usuario (línea 5).
2. Composición del **prompt** con las entradas del usuario y los datos obtenidos en *Commands_LoadChat()* (línea 9).


```

1 def Translate():
2     data = request.json
3
4     #obtención de las entradas del usuario
5     userInput = data.get('userInput')
6     numTokens = 20
7
8     #generación del prompt con las entradas del usuario
9     prompt = *prompt confidencial*
10
11    #obtención de la respuesta del servidor
12    response = requests.post(URL + "/genResponse",
13                             json={"userInput": prompt, "numTokens": numTokens},
14                             headers=headers).json()
15    if response.get("response"):
16        res = response["response"]
17    else:
18        return jsonify({"status": "200", "output": "error"})
19
20    #composición de la respuesta
21    ind2 = res.find('\n')
22    if ind2 != -1:
23        res = res[:ind2]
24    ind2 = res.find('\n')
25    if ind2 != -1:
26        res = res[:ind2]
27    return jsonify({'output': res})

```

Código 6.6: Función de obtención de la traducción

3. Obtención de la respuesta del servidor, a partir del `prompt` generado (línea 12).
4. Composición de la respuesta (línea 19).

6.2.4 Traducción

Traducción automática de textos cortos a diferentes idiomas. La función *Translate_SetLanguage* define los idiomas con los que se trabajará en las consultas. Sigue una estructura similar a *Commands_LoadChat* (ver código 6.4):

1. Comprobación de la validez de los datos de entrada
2. Composición de la conversación con los datos de la *request*.

No es necesaria la definición de una estructura de la conversación debido a que esta cambia totalmente al modificar los idiomas de entrada y salida. No hay ninguna parte de la conversación que se repita para todos los idiomas.

Para la generación del texto (ver código 6.6) encontramos los siguientes pasos:

1. Obtención de las entradas del usuario (línea 5).

```

1 def List():
2     data = request.json
3
4     #obtención de las entradas del usuario
5     userInput = data.get('userInput')
6     numTokens = 500
7
8     #generación del prompt con las entradas del usuario
9     prompt = *prompt confidencial*
10
11    #obtención de la respuesta del servidor
12    response = requests.post(URL+ "/genResponse",
13    json={"userInput": prompt, "numTokens": numTokens},
14    headers=headers).json()
15    if response.get("response"):
16        res = response.get("response")
17    else:
18        return jsonify({"status": "400", "data": "error"})
19
20    #composición de la respuesta
21    if res.find("\n\n") != -1:
22        res = res[:res.find("\n\n")]
23    res = res.replace('-', '')
24    res = res.split('\n')
25    res = res[1:]
26    return jsonify({'output': res})

```

Código 6.7: Función de obtención de la lista

2. Composición del `prompt` con las entradas del usuario y los datos obtenidos en `Commands_LoadChat()` (línea 8).
3. Obtención de la respuesta del servidor, a partir del `prompt` generado (línea 12).
4. Composición de la respuesta (línea 19).

6.2.5 Lista

Extracción de elementos separables dentro de un texto de entrada, devolviendo una lista de los mismos. Para este caso de uso no es necesario definir una conversación ni un `prompt` previamente, ya que se utilizará el mismo en todos los casos. Para la obtención del resultado se siguen los siguientes pasos (ver código 6.7):

1. Obtención de las entradas del usuario (línea 5).
2. Generación del `prompt` con las entradas del usuario (línea 9).
3. Obtención de la respuesta del servidor (línea 12).
4. Composición de la respuesta a partir de la salida del servidor (línea 19).

```

1 def Extract():
2     if prompt_ex == "":
3         return jsonify({"status": "500", "message": "prompt not
4         defined"})
5         data = request.json
6
7     #obtención de las entradas del usuario
8     userInput = data.get('userInput')
9     numTokens = 150
10
11    #generación del prompt con las entradas del usuario
12    prompt = *prompt confidencial*
13
14    #obtención de la respuesta del servidor
15    response = requests.post(URL+ "/genResponse",
16    json={"userInput": prompt, "numTokens": numTokens},
17    headers=headers).json()
18    res = response.get("response")
19    if res:
20        return jsonify({"status": "200", "output": res})
21    else:
22        return jsonify({'output': "error"})

```

Código 6.8: Función de *extracción de datos*

6.2.6 Extracción de datos

Extracción de datos específicos y preestablecidos de un texto de entrada, organizándolos en un formato definido. Como podemos ver en el código 6.8, no es necesario procesar la salida del LLM, ya que, para esta funcionalidad, el modelo genera la salida ya en el formato correcto desde el inicio. El `prompt` se crea previamente con la función `Extract_SetData()`, que tiene una estructura similar a `Commands_LoadChat` (ver código 6.4).

6.2.7 Gestión de diálogo

Extracción de datos preestablecidos manteniendo una conversación estructurada y coherente con el usuario. Como se muestra en el código 6.9, la implementación para este caso de uso es diferente a la de los anteriores. Se siguen los siguientes pasos:

1. Obtención de las entradas del usuario, que se añadirá a la conversación con el LLM (línea 8).
2. Generación del `prompt` con la conversación completa (línea 15).
3. Obtención de la respuesta del servidor (línea 18).
4. Añadir la salida del LLM a la conversación (línea 22).

```
1 def Dialogue():
2     global chatDlg
3
4     if chatDlg == []:
5         return jsonify({"status": "500", "message": "prompt not
6         defined"})
7     if len(chatDlg) > 1:
8         #obtención de las entradas del usuario
9         data = request.json
10        userInput = data.get('userInput')
11        #añadir las entradas a la conversación
12        chatDlg += [{"role": "user", "content": userInput}]
13        numTokens = 150
14
15        #generación del prompt con la conversación completa
16        prompt = *prompt confidencial*
17
18        #obtención de la respuesta del servidor
19        response = requests.post(URL+ "/genResponse",
20        json={"userInput": prompt, "numTokens": numTokens},
21        headers=headers).json()
22        res = response.get("response")
23
24        #añadir el texto nuevo a la conversación
25        if res:
26            chatDlg += [{"role": "assistant", "content":
27            res.replace(" ", " ")}]
28            return jsonify({"status": "200", "output": res})
29        else:
30            return jsonify({'output': "error"})
```

Código 6.9: Función de *Gestión de diálogo*

```

1 services:
2   api:
3     image: nvidia/cuda:12.2-base-ubuntu22.04
4     build: .
5     ports:
6       - "5000:5000"
7     deploy:
8       resources:
9         reservations:
10          devices:
11            - driver: nvidia
12              count: 1
13              capabilities: [gpu]
14     healthcheck:
15       test: ["CMD", "curl", "-f", "http://localhost:5000/health"]
16       interval: 30s
17       timeout: 5s
18       retries: 3
19     restart: always
20     volumes:
21       - *archivos del LLM*
22     environment:
23       - CUDA_VISIBLE_DEVICES=0
24   autoheal:
25     restart: always
26     image: willfarrell/autoheal
27     environment:
28       - AUTOHEAL_CONTAINER_LABEL=all
29     volumes:
30       - /var/run/docker.sock:/var/run/docker.sock

```

Código 6.10: archivo .yaml del contenedor de la API del servidor LLM

El `prompt` se crea previamente en la función `Dialogue_SetData()`, que recibe los datos que se utilizarán en la conversación, y genera el `prompt` correspondiente.

6.3 Implementación de Docker

Para exponer la API del servidor LLM al exterior, es imprescindible configurar correctamente el contenedor. Si empleamos Docker Compose para este propósito, es necesario ajustar tanto el archivo `docker-compose.yml` como el `Dockerfile`.

En el código 6.10 se muestra la configuración del archivo YAML, que define los servicios necesarios para desplegar la aplicación con soporte para GPU mediante CUDA. En la línea 14, tenemos el `healthcheck`, que se utiliza para monitorear el estado del servidor. Cada 30 segundos, se enviará al servidor una petición al endpoint `/health`, que devolverá siempre un mensaje de `OK`. Si la respuesta tarda más de 5 segundos en llegar, se considerará la prueba como fallida. Si esto ocurre 3 veces consecutivas, el servidor será marcado como `unhealthy`.

Si el servidor está marcado como *unhealthy*, se reiniciará automáticamente, empleando la imagen *willfarrell/autoheal*.

Estos tiempos de espera se establecieron porque el servidor no debería tardar más de 5 segundos en responder. Sin embargo, es posible que alguna de estas llamadas coincida con una petición muy larga, por lo tanto, se debe repetir el fallo 3 veces para que el servidor se reinicie.

6.4 Implementación en LM Studio

Se realizaron algunas pruebas en LM Studio para garantizar que el modelo de Mistral daría buenos resultados en las pruebas reales. Se utilizó el modelo *TheBloke/Mistral-7B-OpenOrca-GGUF*, que es un modelo creado a partir de Mistral-7B [3], y reentrenado con el dataset de OpenOrca. También está cuantizado, para que pueda ser cargado en la GPU del portátil (6GB de capacidad).

Se le proporcionaron varios comandos en el chat. Después, debería deducir, a partir de un texto de entrada, el comando correspondiente. Como input/prompt inicial se utilizó el siguiente texto: *"Recibirás una serie de comandos con una descripción de cada uno. Después recibirás un texto y tienes que deducir a cuál de los comandos se refiere"*.

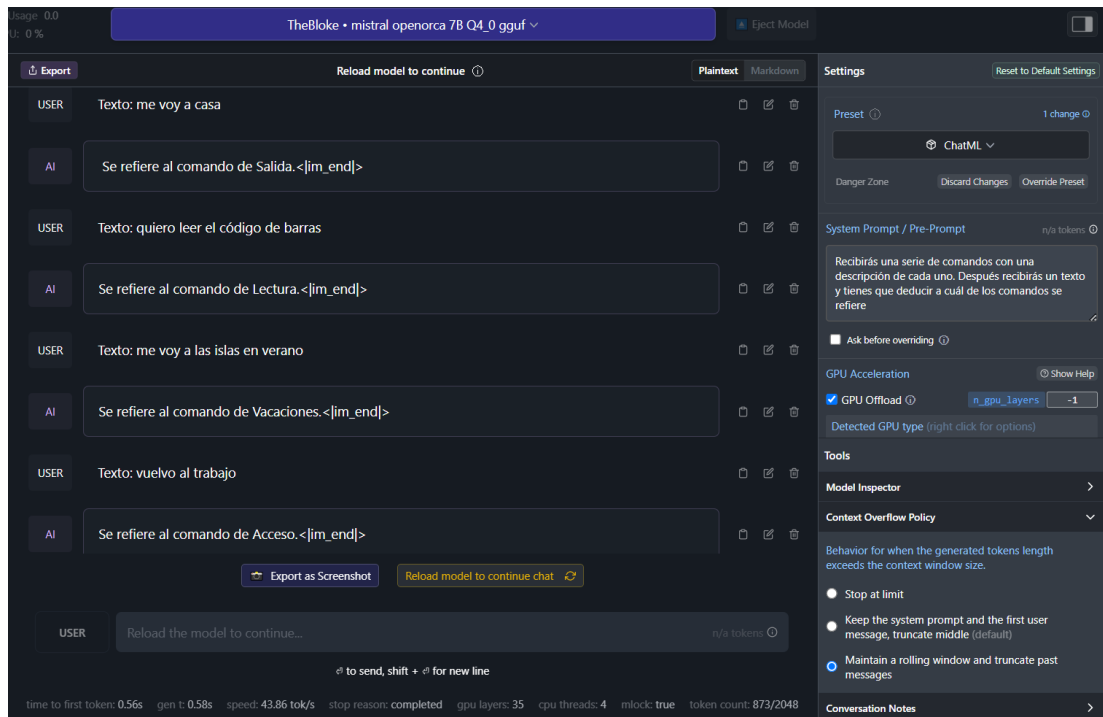
Las pruebas tuvieron buenos resultados. Podemos destacar la que se muestra en la figura 6.1b. El resultado obtenido fue de 0 errores en 15 ejemplos. Entre las frases de prueba se incluyeron algunas en inglés, para comprobar si el modelo modificaría la forma de responder. Sin embargo, no tuvo fallos en esos casos tampoco.

Podemos comprobar que en cada prueba el formato de las respuestas es diferente (en el primer caso responde "Se refiere a *comando*", mientras que en el segundo responde "Comando: *comando*"). Esto se debe a que no se le indicó al modelo en ningún momento de qué forma debería responder. Para estas primeras comprobaciones no era necesario que se devolviera un formato específico, ya que el objetivo era únicamente evaluar el modelo en este caso de uso.

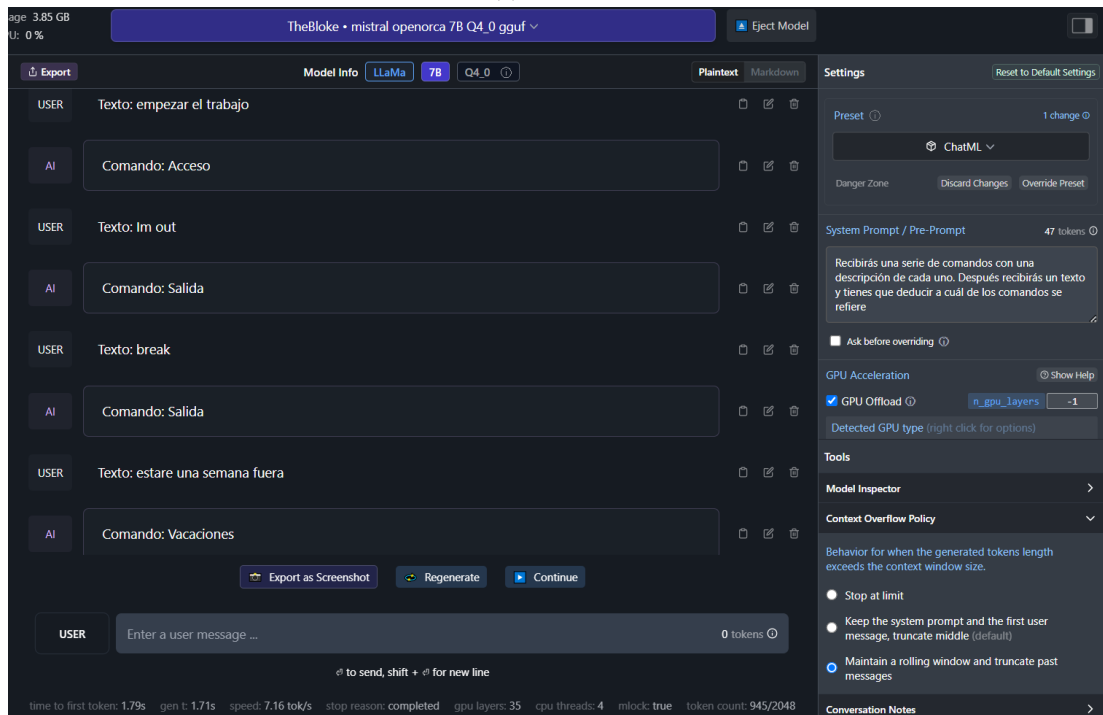
En definitiva, el resultado de las pruebas en LM Studio fue satisfactorio y demostró la eficacia del modelo de Mistral.

6.5 Interfaz de usuario

La interfaz de nuestro sistema está construida en HTML y su diseño se organiza en tres secciones principales: una barra lateral, una barra superior y el cuerpo. A través de cajas de texto y botones ubicados dentro de un formulario, el usuario puede interactuar con el LLM. Los datos obtenidos del usuario se envían a la API del procesador de texto mediante



(a) Prueba 1



(b) Prueba 2

Figura 6.1: Pruebas con LM Studio

```
1 document.addEventListener('DOMContentLoaded', async function() {  
2     const response = await fetch('/api/check-model-status', {  
3         method: 'POST',  
4         headers: {  
5             'Content-Type': 'application/json',  
6         }  
7     });  
8     const result = await response.json();  
9     if (result.running) {  
10         modelLoaded = true;  
11         document.getElementById('modelStatus').textContent =  
12         'Modelo cargado';  
13     } else {  
14         modelLoaded = false;  
15         document.getElementById('modelStatus').textContent =  
16         'Modelo no cargado';  
17     }  
18 });
```

Código 6.11: Función para obtener el estado del modelo para la interfaz de usuario

llamadas [HTTP](#), y los resultados obtenidos se muestran en la interfaz utilizando párrafos de texto (etiqueta "`<p>`").

Las dos principales funciones de esta interfaz son:

- **Visualización del estado del modelo:** Como podemos observar en el código 6.11, se comprueba el estado del [LLM](#), si está o no disponible, y se indica con una etiqueta que se muestra en la barra superior. Utilizando el evento *DOMContentLoaded*, esta función se ejecuta cada vez que se carga la página, después que el contenido de la misma se ha cargado completamente. *Fetch* se utiliza para hacer la solicitud [HTTP](#) al endpoint */check-model-status*. Finalmente, se comprueba la respuesta de la solicitud y se indica el estado del modelo en la etiqueta *"modelStatus"*.
- **Interacción con el [LLM](#):** Utilizando un formulario, el usuario podrá realizar consultas al [LLM](#). En el código 6.12 (para el caso *Comandos*) la función se ejecuta cuando el usuario selecciona la opción de *"enviar"* y salta el evento *"submit"* del formulario. Con *fetch* se realiza la solicitud [HTTP](#), con el *"userInput"* en el cuerpo de la solicitud. Finalmente, se muestra la respuesta de esta solicitud en la etiqueta *"outputText"*.


```
1 document.getElementById('inputForm').addEventListener('submit',  
2   async function(event) {  
3     event.preventDefault();  
4     const userInput = document.getElementById('userInput').value;  
5  
6     response = await fetch('/api/commands', {  
7       method: 'POST',  
8       headers: {  
9         'Content-Type': 'application/json',  
10      },  
11      body: JSON.stringify({userInput: userInput}),  
12    });  
13    result = await response.json();  
14    document.getElementById('outputText').textContent =  
15    result.output;  
16  });
```

Código 6.12: Función para realizar consultas al modelo para la interfaz de usuario

Pruebas y Resultados

ESTE capítulo se centra en la evaluación y análisis del LLM en diferentes aspectos. Se presentan y examinan los resultados obtenidos durante las pruebas de las distintas funcionalidades de nuestro sistema. Además, se comparan los resultados de los modelos Mistral (el modelo finalmente utilizado en el proyecto) y Phi-2. Se muestran también los tiempos obtenidos en la generación de texto, y se realiza un pequeño análisis de estos resultados.

7.1 Comparativa de resultados de Mistral 7B y Phi-2

Los modelos Mistral 7B y Phi-2 tienen una clara diferencia en tamaño, siendo el segundo considerablemente más pequeño. Si ambos producen resultados similares, Phi-2 sería mucho más conveniente que Mistral, ya que su menor tamaño permite la instalación y uso en equipos menos potentes, reduciendo así los costos.

En la tabla 7.1 se pueden observar los resultados obtenidos en las pruebas. Se compararon los resultados, en el caso de uso de *Comandos*, con unos comandos de prueba. Se utilizó un documento formado por 47 ejemplos de textos para deducir el comando correspondiente (ver anexo A.1). Para el caso de *Lista*, se realizó la comparativa con 30 ejemplos diferentes (ver anexo A.3).

La tabla muestra claras diferencias entre los dos modelos. Mistral 7B produce resultados significativamente mejores que Phi-2, y no vale la pena sacrificar tanta calidad en las respuestas por el posible ahorro en costos que ofrecería el modelo más pequeño.

Porcentaje de acierto	<i>Comandos</i>	<i>Lista</i>
Mistral 7B	93,62%	70,00%
Phi-2	80,85%	56,67%

Tabla 7.1: Comparativa de resultados según los modelos Mistral y Phi-2

Núm. tokens	Tiempo (ms)	Desv. típica (ms)
10	184,100	0,031
25	459,700	0,076
100	1837,40	0,46
500	9323,40	0,47

Tabla 7.2: Tiempos de generación del modelo Mistral 7B

	Tiempo min.	Tiempo max.	Tiempo medio
5 peticiones	560	1327	771
10 peticiones	564	2595	1076
15 peticiones	803	8276	3773

Tabla 7.3: Pruebas de carga al servidor LLM (tiempos en milisegundos)

7.2 Tiempos de generación

En la tabla 7.2 se pueden observar los tiempos de generación de texto con Mistral 7B, calculados utilizando la gráfica NVIDIA GeForce RTX 4090 para distintos números de **tokens**. Las pruebas se llevaron a cabo con 10, 25, 100 y 500 **tokens**, midiendo el tiempo en cada caso 10 veces para luego calcular la media y la desviación estándar.

En la tabla se observa que el tiempo de generación aumenta linealmente con el número de **tokens**, con una media de 0,01845 segundos por **token**. 1 **token** representa aproximadamente 0,75 palabras, lo que resulta en un tiempo de aproximadamente 0,0246 segundos por palabra.

7.3 Pruebas de carga al Servidor LLM

Se realizaron varias pruebas de carga sobre la API del servidor LLM, una vez expuesto al exterior. Estas pruebas se realizaron con la herramienta JMeter, que permite realizar varias peticiones **HTTP** simultáneas. En la tabla 7.3 se presentan los tiempos mínimo, máximo y medio de las peticiones realizadas al servidor LLM en estas pruebas. En cada solicitud se genera una respuesta, que podrá ser de 5, 10 o 20 **tokens**, seleccionada aleatoriamente entre

	Tiempo min.	Tiempo max.	Tiempo medio
5 peticiones	561	3203	821
10 peticiones	562	5738	1138
15 peticiones	908	9319	3780

Tabla 7.4: Pruebas de carga al servidor LLM (peticiones simultáneas, tiempos en milisegundos)

estas 3 opciones. Cada 5 segundos, se va enviando un número de peticiones, distribuidas a lo largo de ese intervalo de tiempo. Por ejemplo, si se envían 10 peticiones, una solicitud llega al servidor cada 0,5 segundos.

El servidor tuvo unos resultados aceptables, pero con 15 peticiones ya se empezaban a acumular las peticiones, y no le daba tiempo al servidor a finalizarlas antes de que se acabaran los 5 segundos. Con este número de peticiones, el tiempo máximo que se obtuvo es de 8,28 segundos, que es excesivamente alto.

La tabla 7.4 muestra los resultados obtenidos para las mismas pruebas que las anteriores, pero enviando todas las solicitudes simultáneamente. Cada 5 segundos se envía un número de peticiones (5, 10 y 15) a la vez. Con este cambio, los tiempos mínimo y medio prácticamente no cambian, pero el tiempo máximo aumenta.

Debido a que cada solicitud debe procesarse de manera secuencial, cuando se alcanzan aproximadamente 2,5 peticiones por segundo, estas pueden comenzar a acumularse, lo que incrementaría significativamente el tiempo de respuesta. Aunque actualmente no se espera recibir tal cantidad de solicitudes, si en un futuro fuera necesario, se debería considerar la adquisición de un servidor más potente y con mayor capacidad.

7.4 Pruebas de Comandos

El objetivo de estas pruebas es comprobar si el LLM es capaz de deducir el comando que está describiendo (o al que se refiere) la entrada del usuario.

En estas pruebas se utilizaron comandos de ejemplo que simulan los que una empresa podría utilizar. Los comandos son los siguientes:

- **Lectura:** Comando para leer un código de barras
- **Micrófono:** Añadir datos del micrófono
- **Vacaciones:** Comando para pedir vacaciones
- **Acceso:** Comando para el inicio de jornada o para volver a trabajar después de una pausa
- **Salida:** Comando para finalizar la jornada o para hacer una pausa
- **Default:** Si no se asemeja a ninguno de estos comandos anteriores

Para analizar los resultados de este caso, se creó un documento con 47 ejemplos de posibles entradas del usuario (ver anexos A.1). El resultado obtenido mostró que el modelo alcanzó un porcentaje de acierto del 93,62%, tan solo fallando en 3 casos (concretamente los números 1, 14 y 41).

Se incluyeron textos muy variados, intentando simular el comportamiento de un usuario cualquiera, que podría tener errores ortográficos, acortar palabras, utilizar lenguaje coloquial, etc. El LLM proporcionó unos resultados correctos también en estos casos. Por ejemplo, con el texto "inicio", el modelo dedujo correctamente el comando *Acceso*. Con el texto en inglés "bar code", también dedujo correctamente el comando *Lectura*.

7.5 Pruebas de Traducción

El objetivo de estas pruebas es evaluar la capacidad de traducción del modelo. La traducción a otro idioma debe conservar el sentido y la estructura del texto original.

Las pruebas se hicieron con los idiomas: español (es), inglés (en), italiano (it), francés (fr) y alemán (de). Esto se debe a que, en otros idiomas, el modelo Mistral no fue entrenado con suficientes datos como para dar buenos resultados. Los ejemplos utilizados para estas pruebas podemos encontrarlos en el anexo A.2.

A continuación se mostrará una comparativa de la traducción con diferentes idiomas. Se utilizarán los ejemplos 2, 3, 4, 8 y 11 del anexo A.2:

2. *El café humeante en la fría mañana.*
3. *Los niños ríen en el parque soleado.*
4. *Un libro viejo guarda muchas historias.*
8. *Un gato duerme en el sillón cómodo.*
11. *La ciudad despierta con el amanecer.*

Primero, de español a italiano:

2. *Il caffè caldicchio nella mattina fredda.*
3. *I bambini ridono nel parco soleggiato.*
4. *Un vecchio libro conserva molte storie.*
8. *Un gatto dorme sul divano confortevole.*
11. *La città si sveglia all'alba.*

Después, de español a inglés:

2. *The steaming coffee on a cold morning.*
3. *Children laugh in the sunny park.*

4. *An old book holds many stories.*
8. *Music sets a festive mood for the environment.*
11. *Colorful flowers decorate the garden.*

Finalmente, hacemos la traducción de inglés a italiano a partir de los resultados obtenidos:

2. *Il caffè caldamente servito su un mattino freddo.*
3. *I bambini ridono nel parco soleggiato.*
4. *Un vecchio libro contiene tante storie.*
8. *Un gatto dorme sulla poltrona confortevole.*
11. *La città si sveglia all'alba.*

Ahora comparamos los resultados obtenidos en este último paso con la traducción del español al italiano. Podemos observar que no siempre se obtuvieron frases iguales, pero el significado de las mismas es equivalente.

También se incluyeron en algunos de los textos fallos ortográficos y gramaticales, para evaluar el comportamiento del modelo en estos casos. Por ejemplo, en el texto *"La hojas crujen bajo sus pies"*, tenemos un error en *"La"*, que debería ser *"Las"*, pero al traducirlo al inglés, se obtiene *"The leaves crunch under his feet"*, que es una traducción correcta.

7.6 Pruebas de Lista

El objetivo de las pruebas de este caso de uso es comprobar que el LLM sea capaz de extraer una lista de elementos a partir de un texto. Para que el resultado sea satisfactorio, el LLM deberá poder realizar esta tarea con una amplia variedad de textos diferentes.

Se utilizaron 30 ejemplos de textos con elementos separables (ver anexo A.3). Se obtuvo un resultado de 21 aciertos y 9 errores, logrando un porcentaje de acierto del 70%. De los 9 errores, algunos son menos graves que otros, y solamente 2 de ellos se consideran realmente "graves". Esto es muy positivo, ya que los errores más leves podrían llegar a considerarse como aciertos, lo que nos daría un porcentaje de acierto del 93,33%.

A continuación veremos una descripción más detallada de estos errores.

7.6.1 Errores leves

Dentro de los errores leves podemos destacar 2 categorías:

- **Se identifican erróneamente partes del texto como elementos separables cuando no lo son:** Esta clase de error es la menos grave, ya que en un caso real, el usuario podría simplemente eliminar el elemento encontrado erróneamente y quedarse con el resto. Los casos donde se encontraron elementos de más son (errores en negrita):
 - Ejemplo 4: [Pedro, Marcos, Antonio, **tres, amigos**]
 - Ejemplo 18: [**Para planificar un viaje**, El destino, Reservar alojamiento, Comprar boletos de avión, Hacer una lista de lugares a visitar, Preparar el equipaje]
 - Ejemplo 19: [**Tigre**, Pelaje anaranjado, Rayas negras, Garras afiladas, Gran agilidad, Dieta carnívora, **Esperanza de vida de aproximadamente**, 20 años]
 - Ejemplo 24: [**Developers of software**, **Diversas herramientas**, Editores de código (Visual Studio Code, Atom), Sistemas de control de versiones (Git), Plataformas de colaboración (GitHub, GitLab)]
 - Ejemplo 25: [**La producción de automóviles**, **El proceso meticuloso**, El diseño del vehículo, La fabricación de piezas, El ensamblaje, Las pruebas de calidad, La distribución]
 - Ejemplo 30: [**Sistema de gestión empresarial**, Contabilidad, Gestión de inventario, Recursos humanos, Compras, Ventas, Producción]
- **Existe ambigüedad en la solución:** Estos casos podrían no ser considerados como errores, ya que su validez depende de la interpretación que le pueda dar el usuario.
 - Ejemplo 22: [Objetos, Suelo, Superficies, Trapeador, Espejos, Ventanas, Muebles, Sábanas, Basura]
Podríamos considerar que los elementos separables del texto son las acciones enumeradas en él (Barrer el suelo, limpiar superficies, pasar el trapeador...). Sin embargo, también se podría considerar que los elementos separables son los objetos mencionados en el texto (Suelo, Superficies, Trapeador...). Ambas interpretaciones son válidas, y textos de este tipo podrían llevar a errores sin una solución aparente.

7.6.2 Errores graves

Dentro de los errores más graves, podemos encontrar 2 tipos:

- **No se encuentran todos los elementos del texto:** Esta clase de error es la que se debería disminuir en mayor medida, ya que implica que el usuario tenga que introducir manualmente los elementos que el LLM no pudo encontrar.
 - Ejemplo 13: [Proteínas magras, Carbohidratos complejos, Grasas saludables, Vitaminas, Minerales, Fibra dietética]

Falta "Una adecuada hidratación"

- **Se identifica erróneamente y no se encuentran todos los elementos:** Es el tipo de error más grave, y lo encontramos en el caso:

- Ejemplo 28: [HTML, CSS, JavaScript, imágenes, **ventanas atractivas, historias cautivadoras**]

Falta "contenido textual", y además, se detectaron incorrectamente "ventanas atractivas" e "historias cautivadoras".

En algunos de los ejemplos utilizados se incluyeron intencionalmente algunos textos con errores ortográficos. Esto se realizó para comprobar si el LLM devolvería los elementos con errores tal y como se presentan en el texto, o los modificaría para que estuvieran en su forma correcta.

Los resultados no fueron positivos en este aspecto, en algunos casos se obtuvo un resultado correcto y en otros no. Por ejemplo, en el texto *"Me gustan las hamburguesas y las pizzas, pero también la ensalada y el cocido. Mi favorito es el helado"*, el LLM devolvió correctamente ["hamburguesas", "pizzas", "ensalada", "cocido", "helado"], sin el error "helado". Pero en el ejemplo *"La rutina matutina incluye flecciones, abdominales, planchas, sentadillas y burpees"*, el resultado fue ["Flecciones", "Abdominales", "Planchas", "Sentadillas", "Burpees"], que mantiene el error ortográfico "Flecciones".

7.7 Pruebas de Extracción de datos

La evaluación de las pruebas para esta funcionalidad se centra en la capacidad del LLM para comprender el significado del texto y separar una serie de datos (definidos previamente) del mismo. Por ejemplo, si se tiene que obtener el dato "estado del coche" y tenemos el texto *"las ventanas del coche están rotas"*, el modelo debería ser capaz de obtener el dato "estado del coche: **defectuoso**".

Para estas pruebas se utilizaron unos datos proporcionados por SREC Solutions, que pertenecen a un caso real para obtener los datos de un destornillador. Los datos que se tienen que obtener son:

1. Estado de la punta
2. Longitud de la punta
3. Estado del vástago
4. Longitud del vástago

5. Estado del mango
6. Diámetro de una cara del hexágono de la punta
7. Diámetro de la segunda cara
8. Diámetro de la tercera cara

Se utilizó un documento de 12 ejemplos de textos que podría introducir un usuario para estos datos (ver anexo A.4).

El resultado obtenido fue de un 66,67% de acierto, fallando en 4 de los 12 ejemplos. Los errores fueron los siguientes:

- Ejemplo 1: Falta un dato (Defectos mango: N/A).
- Ejemplo 5: Hay un dato erróneo (Defectos punta: no) y falta un dato (Defectos vástago: N/A).
- Ejemplo 7: Hay un dato correcto puesto en un lugar erróneo (Longitud punta puesta en Diámetro 3).
- Ejemplo 8: Faltan 2 datos (Defectos vástago: N/A, Defectos mango: N/A).

7.8 Pruebas de Gestión de diálogo

En estas pruebas se pretende obtener una conversación entre usuario/asistente, en la que el asistente recaba una serie de datos del usuario. Esta conversación debe dar al usuario una sensación de estar hablando con otra persona, dando respuestas naturales y comprendiendo las peticiones del usuario.

Se utilizaron para estas pruebas los mismos datos que en el caso de uso anterior (ver sección 7.7). Las primeras pruebas se realizaron con el modelo *Mistral 7B*.

Se crearon 2 conversaciones diferentes, para probar diversas situaciones, observando cómo reaccionaría el modelo. La primera conversación es más sencilla, proporcionando datos de iteraciones posteriores y varios datos simultáneamente. Se muestra a continuación un ejemplo de cómo se debería comportar el asistente ante estas entradas del usuario:

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: El mango tiene buen aspecto

ASSISTANT: Gracias por la información sobre el mango. Ahora, continuemos: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Ah sí, la punta también

ASSISTANT: Perfecto, he registrado que la punta está bien. Ahora, ¿puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: 12mm

ASSISTANT: He registrado el primer diámetro de la punta. Ahora, ¿puedes darme la medida del diámetro de la segunda cara del hexágono de la punta en milímetros?

USER: 7 y la otra 14

ASSISTANT: He registrado las medidas de las tres caras del hexágono de la punta. Ahora, ¿puedes decirme cuál es la longitud de la punta en milímetros?

USER: El vástago mide 45, pero está roto

ASSISTANT: Gracias por la información. He registrado la longitud y el estado del vástago. Ahora, para completar: ¿Cuál es la longitud de la punta en milímetros?

USER: 5

ASSISTANT: Aquí está la información recopilada:

- Diámetro 1: 12
- Diámetro 2: 7
- Diámetro 3: 14
- Longitud punta: 5
- Longitud vástago: 45
- Defectos punta: no
- Defectos vástago: sí
- Defectos mango: no

Como podemos observar, el usuario proporciona datos que no corresponden a la pregunta formulada, o responde con información adicional que no se le pidió. El LLM debería ser capaz de interpretar estas respuestas y reflejarlo correctamente en el resultado final.

En la segunda conversación, se incluyen respuestas algo más complicadas de comprender. Además, se modifica un dato de una iteración anterior para verificar si las respuestas se actualizan. En algunas respuestas, no se responde a la pregunta planteada, para evaluar el comportamiento del modelo en esos casos.

Los resultados para la primera conversación fueron buenos, ya que solo fue necesario repetir un dato, y los datos que se expusieron al final fueron correctos. En cambio, en la segunda conversación, que era más compleja, el rendimiento fue deficiente. Muchos de los datos proporcionados no fueron detectados correctamente por el LLM. También se recibieron malas respuestas que no aclaraban adecuadamente lo que el usuario debía responder. Al final se mostraron los datos correctamente, pero al necesitar repetir los datos en repetidas ocasiones

e intentar que el modelo comprenda la información que se le proporciona, el proceso se hace muy lento y tedioso. El usuario tardaría menos introduciendo los datos manualmente uno por uno (ver conversaciones completas en el anexo [A.5.1](#)).

Ante estos malos resultados, se probó a realizar las mismas pruebas, pero con modelos más grandes, que estuviesen disponibles al público como *ChatGPT* y *Claude 3* (ver resultados en anexo [A.5](#)).

Los resultados fueron muy positivos en estos casos, demostrando que el mayor tamaño de estos modelos produce una gran diferencia en cuanto a rendimiento. Los tres modelos devolvieron respuestas distintas, pero todos llegaron al resultado final, comprendiendo todas las entradas del usuario. En un futuro, podría considerarse la opción de adquirir un plan de suscripción para poder utilizar alguno de estos modelos, especialmente para funcionalidades más complejas, como *Gestión de diálogo*.

Conclusiones y trabajo futuro

DURANTE esta memoria hemos visto en detalle todo el proceso por el cual se consiguió un sistema de comprensión de lenguaje natural. Desde la planificación inicial hasta los resultados finales. En este capítulo veremos las conclusiones finales del proyecto, así como posibles mejoras y avances futuros.

8.1 Conclusiones

El principal objetivo que se propuso al principio de este proyecto era diseñar un sistema de gestión de la comprensión de lenguaje natural, en base a un [LLM](#) libre. También se crearía un prototipo de aplicación web con ese sistema y se integraría el sistema con [SINVAD](#). A lo largo de esta memoria se fue detallando todo el proceso por el cual se fueron cumpliendo los objetivos y obteniendo los siguientes resultados:

- Un sistema que, utilizando un modelo de lenguaje libre, puede comprender el lenguaje natural.
- Una aplicación web que permite interactuar con un modelo de lenguaje de forma sencilla e intuitiva, e implementa cinco funcionalidades diferentes.
- Un sistema que permite hacer peticiones al servidor que alberga el [LLM](#), y recibir respuestas del mismo.
- Un servidor que está expuesto al exterior para que pueda ser accedido desde el sistema de comunicaciones avanzadas [SINVAD](#) a través de llamadas tipo REST.

Como podemos comprobar, se cumplieron todos los objetivos propuestos al inicio del proyecto.

Además de alcanzar los objetivos, este proyecto ha sentado las bases para futuras mejoras y adaptaciones. La arquitectura modular del sistema facilita la incorporación de nuevas

funcionalidades, lo que permitirá expandir el sistema a nuevos clientes. Además, el diseño del servidor que contiene el modelo de lenguaje, permite cambiar el LLM que se esté utilizando con mínimas modificaciones en el código. Esto facilita mantenerse actualizado en todo momento con los nuevos modelos que vayan surgiendo.

8.2 Trabajo futuro

Nuestro sistema final puede recibir aún muchos cambios que se considerarán para un futuro. Estos cambios mejorarán considerablemente los resultados obtenidos en las diferentes pruebas que se hicieron en el proyecto.

Desde los inicios de este proyecto, cuando se comenzó a buscar modelos de lenguaje libres, el mundo de los LLM ha cambiado significativamente. Fueron surgiendo nuevos modelos que ofrecen resultados superiores al modelo que se utilizó para las pruebas, *Mistral 7B*. Estos nuevos modelos han demostrado ser mejores en la mayoría de las métricas que se pueden encontrar en el *Open LLM Leaderboard* [7], siendo algunos de ellos de menor tamaño. Incluso el modelo de Mistral recibió algunas actualizaciones y mejoras.

A medida que vayan surgiendo nuevos clientes, será necesario adaptar nuestro sistema para satisfacer las necesidades particulares de cada uno de ellos. Esto implicará implementar nuevas funcionalidades en nuestro sistema, con el objetivo de garantizar que nuestras soluciones se cumplan con los requisitos únicos de cada cliente.

Cabe destacar que este sistema no es muy escalable debido a la capacidad de la GPU, que tiene "solo" 24GB de VRAM. Este tamaño no permite realizar ningún tipo de concurrencia en el procesamiento, y limita el servidor a poder procesar tan solo una petición a la vez. Esto significa que cada solicitud necesita usar la GPU en su totalidad para generar la respuesta, y no puede atender varias solicitudes simultáneamente. Si varias solicitudes llegan al servidor al mismo tiempo, las peticiones tendrán que esperar en una cola hasta que la GPU esté disponible. En otras palabras, las solicitudes adicionales se procesarán secuencialmente, lo que generará un aumento en el tiempo de respuesta para cada usuario. Actualmente, el número de solicitudes que le llegarán al servidor no es suficiente para que este se llegue a sobrecargar y aumente significativamente el tiempo de respuesta. En un futuro, si aumenta el número de peticiones, será necesario adquirir un servidor más potente y con más capacidad.

Apéndices

Documentos de texto para las pruebas

CONJUNTO de datos utilizados en las pruebas de cada caso de uso. Contiene ejemplos de textos de entrada que se utilizaron para obtener los porcentajes de acierto en la sección de resultados del proyecto.

A.1 Ejemplos para las pruebas de *Comandos*

Como se describe en la sección de pruebas (ver sección 7.4), se utilizó un documento con 47 textos cortos para los siguientes comandos de ejemplo:

- **Lectura:** Comando para leer un código de barras
- **Micrófono:** Añadir datos del micrófono
- **Vacaciones:** Comando para pedir vacaciones
- **Acceso:** Comando para el inicio de jornada o para volver a trabajar después de una pausa
- **Salida:** Comando para finalizar la jornada o para hacer una pausa
- **Default:** Si no se asemeja a ninguno de estos comandos anteriores

El documento contiene los siguientes ejemplos:

1. "hola" (Comando: Acceso)
2. "he llegado" (Comando: Acceso)
3. "acceso" (Comando: Acceso)

4. "entrar" (Comando: Acceso)
5. "llegue al trabajo" (Comando: Acceso)
6. "vuelvo" (Comando: Acceso)
7. "vuelvo al trabajo" (Comando: Acceso)
8. "iniicio" (Comando: Acceso)
9. "inicio jornada" (Comando: Acceso)
10. "trabajar" (Comando: Acceso)
11. "empiezo el trabajo" (Comando: Acceso)
12. "me voy" (Comando: Salida)
13. "salgo" (Comando: Salida)
14. "adios" (Comando: Salida)
15. "voy a hacer una pausa" (Comando: Salida)
16. "salgo un momento" (Comando: Salida)
17. "me voy" (Comando: Salida)
18. "salida" (Comando: Salida)
19. "fin de jornada" (Comando: Salida)
20. "pausa" (Comando: Salida)
21. "codigo" (Comando: Lectura)
22. "codigo de barras" (Comando: Lectura)
23. "leer" (Comando: Lectura)
24. "leer codigo" (Comando: Lectura)
25. "bar code" (Comando: Lectura)
26. "lectura" (Comando: Lectura)
27. "lectura codigo" (Comando: Lectura)
28. "ver codigo" (Comando: Lectura)
29. "microfono" (Comando: Microfono)
30. "datos micro" (Comando: Microfono)

31. "añadir datos" (Comando: Microfono)
32. "data del microfono" (Comando: Microfono)
33. "ver datos del micro" (Comando: Microfono)
34. "introducir datos microfono" (Comando: Microfono)
35. "añadir datos" (Comando: Microfono)
36. "nuevos datos del micro" (Comando: Microfono)
37. "microfono datos" (Comando: Microfono)
38. "Me voy de vacaciones" (Comando: Vacaciones)
39. "dias de descanso" (Comando: Vacaciones)
40. "dias libres" (Comando: Vacaciones)
41. "me voy a la playa" (Comando: Vacaciones)
42. "pido unos dias" (Comando: Vacaciones)
43. "vacaciones" (Comando: Vacaciones)
44. "quiero 5 dias de descanso" (Comando: Vacaciones)
45. "que dia puedo irme de vacaciones" (Comando: Vacaciones)
46. "2 semanas de vacaciones" (Comando: Vacaciones)
47. "solicito 8 dias libres" (Comando: Vacaciones)

Algunos de estos textos contienen fallos ortográficos, como en el ejemplo 8, y también encontramos ejemplos en inglés, como en el ejemplo 25. Esto se hizo intencionalmente para verificar si el LLM proporcionaría resultados correctos frente a textos de este tipo.

A.2 Ejemplos para las pruebas de *Traducción*

En este caso, se utilizaron 20 ejemplos de textos cortos para traducir a diferentes idiomas.

1. La luna brilla sobre el tranquilo lago.
2. El café humeante en la fría mañana.
3. Los niños ríen en el parque soleado.
4. Un libro viejo guarda muchas historias.

5. La música envuelve el ambiente festivo.
6. Flores de colores adornan el jardín.
7. Las olas rompen contra las rocas.
8. Un gato duerme en el sillón cómodo.
9. Las estrellas titilan en el cielo nocturno.
10. El viento susurra entre los árboles.
11. La ciudad despierta con el amanecer.
12. Un perro corre feliz por la playa.
13. El tren llega puntual a la estación.
14. La lluvia cae suavemente sobre el techo.
15. Una sonrisa ilumina su rostro cansado.
16. La montaña se alza majestuosa y firme.
17. Los pájaros cantan al inicio del día.
18. El aroma a pan recién horneado.
19. Un arcoíris aparece tras la tormenta.
20. La hojas crujen bajo sus pies.

De la misma forma que en el caso de uso *Comandos*, se incluyeron textos con algunos errores ortográficos y gramaticales, para comprobar el comportamiento del modelo en estos casos.

A.3 Ejemplos para las pruebas de *Lista*

En el caso de uso *Lista* se utilizó un documento con 30 ejemplos de textos con elementos separables.

1. Mis deportes favoritos son el baloncesto y el futbol, pero también me gusta el tenis o el ajedrez
2. Voy a comer un filete de ternera con patatas fritas con sal y un huevo frito
3. Use un martillo, dos tornillos, una tuerca, una llave Allen y dos latiguillos
4. Tengo tres amigos: Pedro, Marcos y Antonio
5. La tortilla lleva huevos, cebolla, patatas y aceite

6. Entre los países implicados están Francia, Suecia y España. Mas tarde llegaron Alemania y Rusia
7. Messi era el delantero izquierdo, Neymar el derecho, Suarez en el centro, y por el medio estaban Xabi e Iniesta
8. Tengo cinco camisetas, tres pantalones, dos faldas, cuatro pares de calcetines y seis prendas interiores
9. En la fiesta de cumpleaños habrá pastel de chocolate, cupcakes de vainilla, galletas decoradas, helado y dulces variados
10. Para sobrevivir necesitarás un machete, una cantimplora con filtro de agua, una hamaca, repelente de mosquitos y un manual de supervivencia
11. Los componentes que se necesitarán son: Intel Core i9-14900K 3.2/6GHz Box (679€), ASUS TUF GAMING Z790 PLUS WIFI (258,99€), Corsair Vengeance RGB DDR5 6000MHz 64GB 2x32GB CL30 Memoria Dual AMD EXPO e Intel XMP (267,99€), Forgeon Arcanite ARGB Mesh Torre ATX Negra (159,99€), Corsair RMx Series RM1000x 1000W 80 Plus Gold Modular (202,99€), MSI GeForce RTX 4090 GAMING X SLIM 24GB GDDR6X DLSS3 (1999,90€)
12. Costos de consumibles: Papel deslizante de corte, Plástico de corte, Plantilla de cartón, Collarín de cartón, Palomita de cuello de plástico, Clic sujeción metálico, Bolsa plástico marca, Caja de transporte marca, Precinto marca.
13. Una dieta equilibrada se compone de varios elementos clave, incluyendo proteínas magras, carbohidratos complejos, grasas saludables, vitaminas y minerales, fibra dietética y una adecuada hidratación.
14. Me gustan las hamburguesas y las pizzas, pero tambien la ensalada y el cocido. Mi favorito es el helado.
15. Tejido base, 1.10uds; 2º Tejido, 0.15uds; Entretela de cuello, 0.15uds; Entretela de puño, 0.19uds; Botones delantero, 10uds; Botones cuello, 2uds; Botones puño, 2uds; Ballenitas, 2uds; Hilo, 4uds.
16. Para desayunar tengo galletas, cereales y tostadas. También tengo leche o café.
17. La ensalada Caprese lleva tomates frescos, mozzarella de búfala, hojas de albahaca fresca, aceite de oliva virgen extra, vinagre balsámico, sal y pimienta al gusto.
18. Para planificar un viaje, es necesario elegir el destino, reservar alojamiento, comprar boletos de avión, hacer una lista de lugares a visitar y preparar el equipaje.
19. El tigre tiene pelaje anaranjado con rayas negras, garras afiladas, gran agilidad, una dieta carnívora y una esperanza de vida de aproximadamente 20 años.
20. La rutina matutina incluye fliciones, abdominales, planchas, sentadillas y burpees.

21. La Industria 4.0 está impulsada por tecnologías como el Internet de las cosas (IoT), la inteligencia artificial (IA), la realidad aumentada (RA), la impresión 3D, el blockchain y el análisis de datos. Estas tecnologías están transformando los procesos industriales y mejorando la eficiencia y la calidad del producto.
22. Organizar objetos, barrer el suelo, limpiar superficies, pasar el trapeador, limpiar espejos y ventanas, aspirar muebles, cambiar sábanas, vaciar la basura.
23. Para realizar un proyecto escolar, asegúrate de tener cartulinas, lápices de colores, pegamento, tijeras, revistas, marcadores y papel de construcción.
24. Los desarrolladores de software utilizan diversas herramientas, como editores de código como Visual Studio Code o Atom, sistemas de control de versiones como Git, y plataformas de colaboración como GitHub o GitLab.
25. La producción de automóviles sigue un proceso meticuloso. Comienza con el diseño del vehículo, seguido de la fabricación de piezas, el ensamblaje, las pruebas de calidad y, finalmente, la distribución.
26. Los sistemas operativos populares incluyen Windows, macOS, Linux, Android e iOS.
27. HTTP es como el carril rápido, HTTPS el túnel seguro, TCP el correo confiable, IP la dirección del hogar y FTP el servicio de mensajería.
28. HTML, cimientos sólidos; CSS, pintura elegante; JavaScript, toques interactivos; imágenes, ventanas atractivas; contenido textual, historias cautivadoras.
29. El proceso de desarrollo de software implica una serie de pasos clave: análisis de requisitos, diseño, codificación, pruebas, implementación y mantenimiento.
30. Un sistema de gestión empresarial integra varios módulos: contabilidad, gestión de inventario, recursos humanos, compras, ventas y producción, para optimizarlos procesos internos de una organización.

Como en los casos de uso anteriores, podemos ver varios errores ortográficos en algunos ejemplos, pero estos también son intencionales.

A.4 Ejemplos para las pruebas de *Extracción de datos*

Para el caso de uso de *Extracción de datos* se utilizó un documento compuesto de 12 ejemplos de posibles textos de un cliente. Los datos a recabar son los siguientes (ver sección 7.7):

1. Estado de la punta
2. Longitud de la punta

3. Estado del vástago
4. Longitud del vástago
5. Estado del mango
6. Diámetro de una cara del hexágono de la punta
7. Diámetro de la segunda cara
8. Diámetro de la tercera cara

El documento contiene los siguientes ejemplos:

1. El diámetro de la primera cara es de 4mm y el de la segunda de 9mm, se ve todo correcto, menos el vástago que tiene varios defectos.
2. Se han encontrado defectos en la punta y el mango.
3. La punta tiene algún problemilla, los diámetros de las caras son: 3mm, 6mm y 15mm, el resto todo bien.
4. El mango está roto. La punta mide 5mm, medidas de las caras: primera 5mm, segunda 9mm, tercera 3mm. El vástago mide 34mm, pero tiene varios defectos.
5. Hay que arreglar el mango y el vástago. La punta mide 12. Diámetros son 3 y 4.
6. Cara 1 5mm, cara 2 5mm, cara 3 9mm. Punta 34mm vástago fastidiado.
7. La punta 34 el vástago 23.
8. Los diámetros son 12, 33 y 5. Vástago y mango están bien, pero la punta no.
9. La punta mide 13mm y está defectuosa, el mango esta defectuoso. El vástago mide 5.
10. 5mm el diámetro 1, 33mm el 2, y 10 el 3.
11. El vástago unos 20mm, el mango está chungo.
12. Vastago mide 32mm. medidas diametro punta: 3mm, 4mm, 334mm. La punta no tiene defectos.

Algunos de estos textos están en lenguaje coloquial. Esto es debido a que se está intentando simular posibles entradas que podría hacer cualquier usuario. También se buscó introducir algún error ortográfico, y términos como *problemilla*, *chungo* o *fastidiado* para comprobar cómo se comportaría el modelo ante entradas de este tipo.

A.5 Conversaciones obtenidas en las pruebas de *Gestión de diálogo*

En las pruebas de *Gestión de diálogo* se utilizaron 2 plantillas de conversaciones para comprobar los resultados de diferentes modelos ante diversas entradas del usuario. A continuación se muestran los resultados que se recibieron de cada uno de los modelos en estas conversaciones.

A.5.1 Mistral 7B

Primera conversación

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: El mango tiene buen aspecto

ASSISTANT: Gracias por la información sobre el mango. Ahora, ¿puedes decirme cuál es el aspecto visual de la punta y si está bien o tiene defectos?

USER: Ah si, la punta también

ASSISTANT: Perfecto, gracias por la actualización. Ahora, ¿puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros? (Diámetro 1)

USER: 12mm

ASSISTANT: Gracias por la información. Ahora, ¿puedes proporcionarme la medida del diámetro de la segunda cara del hexágono de la punta en milímetros? (Diámetro 2)

USER: 7 y la otra 14

ASSISTANT: Perfecto, he registrado las medidas del diámetro de las dos caras del hexágono de la punta. Ahora, ¿puedes decirme cuál es la medida del diámetro de la tercera cara del hexágono de la punta en milímetros? (Diámetro 3)

USER: 14

ASSISTANT: Gracias por la información. Ahora, ¿puedes decirme cuál es la longitud de la punta en milímetros? (Longitud punta)

USER: El vástago mide 45, pero está roto

ASSISTANT: Gracias por la información. Ahora, para continuar: ¿Cuál es la longitud de la punta en milímetros? (Longitud punta)

USER: 5

ASSISTANT: Gracias por la información. La longitud de la punta es de 5 milímetros. Datos:

- Diámetro 1: 12
- Diámetro 2: 7
- Diámetro 3: 14
- Longitud punta: 5
- Longitud vástago: 45
- Defectos punta: no
- Defectos vástago: si
- Defectos mango: no

Segunda conversación

En este caso, con el modelo *Mistral 7B*, algunas de las respuestas del LLM fueron en inglés, en este documento se muestran traducidas.

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Correcto

ASSISTANT: Gracias por la información. Ahora, ¿puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: Mide 13 el vástago

ASSISTANT: Lo siento, pedí el diámetro de una cara del hexágono de la punta, no el largo del eje. ¿Podría proporcionarnos la medida de una cara del hexágono de la punta?

USER: 4, 5 y 12

ASSISTANT: Gracias por la información. He anotado las medidas de los tres lados del hexágono de la punta. Ahora, ¿podrías decirme el largo de la punta en milímetros?

USER: Acabo de ver un defecto en la punta

ASSISTANT: Ya veo. En ese caso, la respuesta a la pregunta anterior sobre el estado de la punta fue incorrecta. ¿Podría proporcionarnos ahora la longitud de la punta en milímetros?
Además, como mencionaste un defecto en la punta, actualizaré el formato de respuesta para incluir un campo "Defectos punta".

USER: no

ASSISTANT: Lo siento, no entendí bien eso. ¿Podría repetir la longitud de la punta en milímetros y también informarme si hay algún defecto en la punta o no?

USER: Un momento

ASSISTANT: Por supuesto, tómate tu tiempo. Si necesita alguna aclaración o información adicional, no dude en preguntar.

USER: Vale, 15mm

ASSISTANT: Gracias por la longitud de la punta. Como mencionaste un defecto, actualizaré el formato de respuesta para incluir un campo "Defectos punta".

USER: continúa

ASSISTANT: Ahora, ¿podría proporcionarnos el diámetro de una cara del hexágono de la punta en milímetros?

USER: 4, 5 y 12

ASSISTANT: Gracias por la información. He anotado las medidas de las tres caras del hexágono de la punta. Ahora, ¿podría decirme la longitud del vástago en milímetros?

USER: 13

ASSISTANT: Gracias por la longitud del vástago

USER: continúa

ASSISTANT: Ahora, ¿podría describir el estado del mango? ¿Está en buenas condiciones o tiene algún defecto?

USER: Está bien

ASSISTANT: Gracias por la información. Según los datos proporcionados, las medidas y condiciones de la punta, vástago y mango son las siguientes:

- Diametro 1: 4
- Diametro 2: 5

- Diametro 3: 12
- Longitud punta: 15
- Longitud vastago: 13
- Defectos punta: si
- Defectos vastago: no
- Defectos mango: no

A.5.2 ChatGPT 4o

Primera conversación

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: El mango tiene buen aspecto

ASSISTANT: Gracias por la información sobre el mango. Ahora, continuemos: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Ah si, la punta también

ASSISTANT: Perfecto, he registrado que la punta está bien. Ahora, ¿puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: 12mm

ASSISTANT: He registrado el primer diámetro de la punta. Ahora, ¿puedes darme la medida del diámetro de la segunda cara del hexágono de la punta en milímetros?

USER: 7 y la otra 14

ASSISTANT: He registrado las medidas de las tres caras del hexágono de la punta. Ahora, ¿puedes decirme cuál es la longitud de la punta en milímetros?

USER: El vástago mide 45, pero está roto

ASSISTANT: Gracias por la información. He registrado la longitud y el estado del vástago. Ahora, para completar: ¿Cuál es la longitud de la punta en milímetros?

USER: 5

ASSISTANT: Perfecto, he registrado la longitud de la punta. Aquí está la información recopilada:

- Diámetro 1: 12
- Diámetro 2: 7
- Diámetro 3: 14
- Longitud punta: 5
- Longitud vástago: 45
- Defectos punta: no
- Defectos vástago: sí
- Defectos mango: no

¿Hay algo más en lo que pueda asistirte?

Segunda conversación

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Correcto

ASSISTANT: Gracias por la información. Ahora, ¿puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: Mide 13 el vástago

ASSISTANT: He registrado la longitud del vástago como 13. Continuando, ¿puedes darme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: 4, 5 y 12

ASSISTANT: Perfecto, he registrado las medidas de los diámetros del hexágono de la punta. Continuando, ¿cuál es la longitud de la punta en milímetros?

USER: Acabo de ver un defecto en la punta

ASSISTANT: Gracias por la actualización. ¿Podrías decirme cuál es la longitud de la punta en milímetros?

USER: no

ASSISTANT: Entiendo, parece que hubo un malentendido en la pregunta anterior. ¿Podrías indicarme la longitud de la punta en milímetros?

USER: Un momento

ASSISTANT: Claro, tómate tu tiempo. Estoy aquí cuando estés listo para continuar.

USER: Vale, 15mm

ASSISTANT: Gracias por la información. Ahora, ¿cómo describirías el aspecto visual del vástago? ¿Está bien o tiene defectos?

USER: Está bien, y también el mango

ASSISTANT: Gracias por la actualización. Aquí están los datos recopilados:

- Diametro 1: 4
- Diametro 2: 5
- Diametro 3: 12
- Longitud punta: 15
- Longitud vastago: 13
- Defectos punta: si
- Defectos vastago: no
- Defectos mango: no

¿Hay algo más en lo que pueda ayudarte?

A.5.3 Claude 3

Primera conversación

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: El mango tiene buen aspecto

ASSISTANT: Gracias por la información sobre el mango. Ahora, volvamos a la pregunta sobre la punta: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Ah si, la punta también

ASSISTANT: Entiendo. Gracias por la información sobre la punta y el mango. Ahora, pasemos a las medidas:

¿Puedes proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: 12mm

ASSISTANT: Gracias. Ahora necesito la medida del diámetro de la segunda cara del hexágono de la punta en milímetros. ¿Puedes proporcionármela?

USER: 7 y la otra 14

ASSISTANT: Gracias por proporcionar esa información. Ahora tenemos las medidas de los tres diámetros del hexágono de la punta.

La siguiente pregunta es: ¿Cuál es la longitud de la punta en milímetros?

USER: El vástago mide 45, pero está roto

ASSISTANT: Entiendo, gracias por la información sobre el vástago. Ahora, volvamos a la pregunta sobre la punta:

¿Cuál es la longitud de la punta en milímetros?

USER: 5

ASSISTANT: Gracias por proporcionar toda esa información. Ahora tengo todos los datos necesarios. Aquí está el resumen de la información recopilada:

- Diametro 1: 12
- Diametro 2: 7
- Diametro 3: 14
- Longitud punta: 5
- Longitud vástago: 45
- Defectos punta: no
- Defectos vástago: si
- Defectos mango: no

Segunda conversación

ASSISTANT: ¿Cuál es el aspecto visual de la punta? ¿Está bien o tiene defectos?

USER: Correcto

ASSISTANT: Gracias por su respuesta, pero necesito una aclaración. Cuando dice "Correcto", ¿se refiere a que la punta está bien y no tiene defectos? Por favor, especifique si la punta está en buen estado o si tiene algún defecto visible.

USER: bien

ASSISTANT: Entendido, la punta está en buen estado.

Ahora, ¿puede proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: Mide 13 el vástago

ASSISTANT: Gracias por la información sobre la longitud del vástago. La anotaré para más adelante.

Ahora, volviendo a la pregunta sobre la punta: ¿Puede proporcionarme la medida del diámetro de una cara del hexágono de la punta en milímetros?

USER: 4, 5 y 12

ASSISTANT: Gracias por proporcionar las medidas de los diámetros de las tres caras del hexágono de la punta. He registrado esa información.

Ahora, ¿puede decirme cuál es la longitud de la punta en milímetros?

USER: Acabo de ver un defecto en la punta

ASSISTANT: Entiendo, gracias por la actualización sobre el estado de la punta. He tomado nota de que tiene un defecto.

Ahora, volviendo a la pregunta anterior: ¿Puede decirme cuál es la longitud de la punta en milímetros?

USER: no

ASSISTANT: Entiendo que no puede proporcionar la longitud de la punta en este momento. No hay problema, podemos volver a esta pregunta más tarde si es necesario.

Pasemos a la siguiente pregunta: ¿Cuál es el aspecto visual del vástago? ¿Está bien o tiene defectos?

USER: Está bien, y también el mango

ASSISTANT: Gracias por la información sobre el vástago y el mango. He registrado que ambos están en buen estado.

Ahora, volviendo a una pregunta pendiente: ¿Podría proporcionarme la longitud de la punta en milímetros? Esta información es importante para completar los datos requeridos.

USER: Un momento

ASSISTANT: Entiendo. Tomaré un momento mientras usted verifica la longitud de la punta. Cuando esté listo, por favor proporcióneme la medida en milímetros.

USER: Vale, 15mm

ASSISTANT: Gracias por proporcionar la longitud de la punta. Ahora tengo toda la información necesaria.

Aquí está el resumen de los datos recopilados en el formato solicitado:

- Diametro 1: 4
- Diametro 2: 5
- Diametro 3: 12
- Longitud punta: 15
- Longitud vástago: 13
- Defectos punta: si
- Defectos vástago: no
- Defectos mango: no

Especificación API del sistema

DOCUMENTO de especificación enviado a la empresa SREC Solutions para hacer un seguimiento de los avances del proyecto. Se incluyen los *endpoints* en el servido LLM y en el procesador de textos.

B.1 Servidor LLM

El servidor expone una API con las siguientes llamadas:

- **(POST) /loadModel:** Carga el modelo en memoria

Respuestas:

- 200: El LLM se ha cargado correctamente.
{“status”: “200”, “message”: “Model loaded”}
- 500: Ocurrió algún error en la operación
{“status”: “500”, “message”: “Model could not load”}

- **(POST) /unloadModel:** Descarga el modelo en memoria

Respuestas:

- 200: El LLM se ha descargado correctamente.
{“status”: “200”, “message”: “Model unloaded”}

- **(POST) /genResponse:** Recibe una conversación user/assistant y un número de tokens máximos a generar. A partir de esos datos devolverá una respuesta.

Esquema cuerpo de la solicitud: application/json

- *userInput*: String con el **prompt** de entrada o una conversación completa, que tendrá el formato que se muestra en el código B.1.

```

1 [
2   { "role": "user", "content": ... },
3   { "role": "assistant", "content": ... },
4   { "role": "user", "content": ... },
5   ...
6 ]

```

Código B.1: Formato del *userInput* en el cuerpo de una petición /genResponse

- *numTokens*: Número máximo de tokens a generar, en el caso de que no se indique, se utilizará el valor por defecto, 100.

Respuestas:

- 200: Operación completada con éxito
{"status": "200", "response": *respuesta*}
- 400: No se indicó ningún input
{"status": "200", "message": "No input provided"}
- 412: El modelo no está cargado
{"status": "412", "message": "Model is not loaded"}
- 500: Ocurrió algún error en la operación.
{"status": "500", "message": "Server error"}
- **(GET) /statusModel**: Indica el estado del modelo, si está o no listo para usarse.

Respuestas:

- {"status": "200", "message": *respuesta*}
La respuesta es un JSON con el campo "running" que indica si el modelo está cargado en memoria (True/False).

B.2 Procesador de texto

La API del procesador de texto tiene las siguientes peticiones en cada caso de uso:

B.2.1 Comandos

- **(POST) /commands/loadChat**: Define los comandos que se utilizarán.

Esquema cuerpo de la solicitud: application/json (ver figura B.2)

- *commands*: lista de strings (serializada como JSON). Cada string debe contener el siguiente formato: "*nombre del comando*: *pequeña descripción del comando*"

```
1 {  
2   "commands": [  
3     "comando1: *descripción*",  
4     "comando2: *descripción*",  
5     ...  
6   ]  
7   "examples": [  
8     "ejemplo 1",  
9     "comando 1",  
10    "ejemplo 2",  
11    "comando 2",  
12    ...  
13  ]  
14 }
```

Código B.2: Cuerpo de la solicitud en la petición `/commands/loadChat`

- *examples*: lista de strings (serializada como JSON). Deben ser parejas de strings con un texto (*userInput*) de ejemplo y su comando correspondiente. Cuantos más ejemplos mejor, pero con dos es suficiente:

Respuestas:

- 200: Comandos definidos correctamente.
{`"status": "200", "message": "Commands defined"`}
- 400: No se envió como parámetro ningún comando.
{`"status": "400", "message": "No commands provided"`}

- **(POST) /commands**: Devuelve el comando correspondiente a partir del input que se indique.

Precondiciones:

- Se debe haber indicado antes los comandos con `/loadChat`

Esquema cuerpo de la solicitud: `application/json`

- *userInput*: input del usuario

Respuestas:

- 200: Operación completada con éxito.
{`"status": "200", "command": *comando obtenido*`}
- 400: No se indicó ningún input.
{`"status": "400", "message": "No input provided"`}

- 412: No se indicaron los comandos.
{“status”: “412”, “message”: “The chat was not loaded”}

B.2.2 Traducción

- **(POST) /translate/setLanguage:** Define el idioma de entrada y el idioma al que se quiere traducir.

Esquema cuerpo de la solicitud: application/json

- *inLang*: idioma de entrada
- *outLang*: idioma de salida

Respuestas:

- 200: Operación completada con éxito.
{“status”: “200”, “command”: ”Language selected successfully”}
- 406: El idioma escogido no está disponible para la traducción.
{“status”: “406”, “message”: “Language not supported”}
- 400: No se indicaron los idiomas de entrada o salida.
{“status”: “400”, “message”: “Input language not provided”}
{“status”: “400”, “message”: “Output language not provided”}

- **(POST) /translate:** Devuelve el texto traducido.

Precondiciones:

- Se debe haber establecido antes los idiomas de la traducción con /setLanguage

Esquema cuerpo de la solicitud: application/json

- *userInput*: texto a traducir.

Respuestas:

- 200: Operación completada con éxito.
{“status”: “200”, “text”: *texto traducido*}
- 400: No se indicó ningún input.
{“status”: “400”, “message”: “No input provided”}
- 412: No se establecieron los idiomas de la traducción.
{“status”: “412”, “message”: “Languages were not set”}

- 500: Ocurrió algún error en la operación.
{“status”: “500”, “message”: “Server error”}

B.2.3 Lista

- **(POST) /list:** Devuelve una lista con los elementos encontrados en la búsqueda.
Esquema cuerpo de la solicitud: application/json

- *userInput*: texto de entrada

Respuestas:

- 200: Operación completada con éxito.
{“status”: “200”, “list”: *lista de strings*}
- 400: No se indicó ningún input.
{“status”: “400”, “message”: “No input provided”}

B.2.4 Extracción de datos

- **(POST) /extract/setData:** Define los datos que se utilizarán en las consultas.
Esquema cuerpo de la solicitud: application/json

- *data*: lista de strings (serializada como JSON). Cada string debe indicar un dato.

Respuestas:

- 200: Datos definidos correctamente.
{“status”: “200”, “message”: ”Data defined”}
- 400: No se envió como parámetro ningún dato.
{“status”: “400”, “message”: “No data provided”}

- **(POST) /extract:** Devuelve un texto ordenado que indica los datos obtenidos en la búsqueda.

Esquema cuerpo de la solicitud: application/json

- *userInput*: texto de entrada

Respuestas:

- 200: Operación completada con éxito.
{“status”: “200”, “data”: *datos obtenidos*}
- 400: No se indicó ningún input.
{“status”: “400”, “message”: “No input provided”}

- 412: No se establecieron los datos a utilizar en las consultas.
{“status”: “412”, “message”: “Data not defined”}

B.2.5 Gestión de diálogo

- **(POST) /dialogue/setData:** Define los datos que se utilizarán en la conversación.

Esquema cuerpo de la solicitud: application/json

- *data*: lista de strings (serializada como JSON). Cada string debe indicar un dato.

Respuestas:

- 200: Datos definidos correctamente.
{“status”: “200”, “message”: “Data defined”}
- 400: No se envió como parámetro ningún dato.
{“status”: “400”, “message”: “No data provided”}

- **(POST) /dialogue:** Devuelve la respuesta del LLM en la conversación.

Esquema cuerpo de la solicitud: application/json

- *userInput*: texto de entrada

Respuestas:

- 200: Operación completada con éxito.
{“status”: “200”, “output”: “*respuesta del modelo*”}
- 400: No se indicó ningún input.
{“status”: “400”, “message”: “No input provided”}
- 412: No se establecieron los datos a utilizar en las consultas.
{“status”: “412”, “message”: “Data not defined”}

Glosario

API (Interfaz de Programación de Aplicaciones) Conjunto de reglas y protocolos que permite a diferentes programas y aplicaciones comunicarse entre sí. Las APIs definen cómo deben interactuar los componentes de software, permitiendo que una aplicación acceda a funciones o datos de otra sin necesidad de conocer su código interno.. [10–12](#), [22](#), [23](#), [26–28](#), [32](#)

headers Campos clave-valor incluidos en una petición HTTP que proporcionan información adicional sobre la solicitud o la respuesta. Pueden incluir datos como el tipo de contenido, la longitud del contenido, el idioma preferido, y más.. [27](#)

HTTP (Hypertext Transfer Protocol) Protocolo de comunicación utilizado en la web para transferir datos entre un cliente (como un navegador web) y un servidor. Permite la transmisión de documentos hipermedia, como páginas web, imágenes y videos, a través de internet. HTTP funciona a través de solicitudes y respuestas: el cliente envía una solicitud al servidor, y el servidor responde con el contenido solicitado o un mensaje de error. Es la base de la navegación web y funciona principalmente sobre el protocolo TCP/IP.. [11](#), [34](#), [52](#), [56](#)

LLM (large language model) Modelo de aprendizaje profundo diseñado para comprender y generar texto en lenguaje natural. Estos modelos se entrenan utilizando enormes cantidades de datos textuales, lo que les permite aprender patrones, contextos, y asociaciones entre palabras y frases, y, como resultado, generar texto coherente y relevante en respuesta a una amplia variedad de entradas. Está formado por un conjunto de redes neuronales de miles, millones o hasta billones de parámetros.. [i](#), [1–6](#), [8](#), [9](#), [12](#), [15–17](#), [22–26](#), [28](#), [31–35](#), [37](#), [39](#), [42](#), [47](#), [50](#), [52](#), [55](#), [57–61](#), [63](#), [65](#), [66](#)

parsear Proceso de analizar una secuencia de datos o un conjunto de información estructurada, como un texto, un archivo, o una cadena de caracteres, para extraer y convertirla en una estructura más comprensible y manipulable para un programa.. [39](#)

prompt Entrada inicial o estímulo que se proporciona al modelo de lenguaje para guiar la generación de texto. Es crucial porque define el tono, el estilo y el contexto del texto generado. La calidad y relevancia del prompt pueden influir significativamente en la calidad y coherencia de las respuestas del modelo. 1, 4, 6, 25, 26, 32–34, 41, 42, 44–47, 49, 50, 81

SINVAD Sistema de Comunicaciones Automatizadas basado en Comprensión de Lenguaje Natural desarrollado por la empresa SREC Solutions. Su propósito principal consiste en brindar ayuda a los usuarios al momento de buscar información, ingresar datos y supervisar diferentes actividades.. ii, 1, 2, 6, 22, 26, 31, 32, 65

token En el contexto de los modelos de lenguaje, un token es una representación numérica de una parte de texto. Cada token corresponde a una palabra, parte de una palabra, o símbolo, y su representación numérica es lo que permite al modelo trabajar con el texto de manera computacional.. 32, 34, 39–42, 56

Bibliografía

- [1] A. Tamkin, M. Brundage, J. Clark, and D. Ganguli, “Understanding the capabilities, limitations, and societal impact of large language models,” 2021. [En línea]. Disponible en: <https://arxiv.org/abs/2102.02503>
- [2] M. Ahmed, H. U. Khan, and E. U. Munir, “Conversational ai: An explication of few-shot learning problem in transformers-based chatbot systems,” *IEEE Transactions on Computational Social Systems*, vol. 11, no. 2, pp. 1888–1906, 2024.
- [3] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mistral 7b,” 2023. [En línea]. Disponible en: <https://arxiv.org/abs/2310.06825>
- [4] S. Arora, A. Narayan, M. F. Chen, L. Orr, N. Guha, K. Bhatia, I. Chami, and C. Re, “Ask me anything: A simple strategy for prompting language models,” in *The Eleventh International Conference on Learning Representations*, 2023. [En línea]. Disponible en: <https://openreview.net/forum?id=bhUPJnS2g0X>
- [5] “Mistral prompting capabilities,” 2024, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: https://docs.mistral.ai/guides/prompting_capabilities/
- [6] O. Zem, “Exploring llm leaderboards,” 2024. [En línea]. Disponible en: <https://medium.com/@olga.zem/exploring-llm-leaderboards-8527eac97431>
- [7] “Open llm leaderboard 2,” 2024, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard
- [8] “Srec solutions,” 2021, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: <https://www.s-recsolutions.com/>

- [9] T. Amaratunga, *Understanding Large Language Models: Learning Their Underlying Concepts and Technologies*. Apress, 2023. [En línea]. Disponible en: <https://books.google.es/books?id=cyQo0AEACAAJ>
- [10] S. B. Mojan Javaheripi, “Phi-2: The surprising power of small language models,” 2023. [En línea]. Disponible en: <https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/>
- [11] T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” 2023. [En línea]. Disponible en: <https://arxiv.org/abs/2307.09288>
- [12] X. Chen, J. Ye, C. Zu, N. Xu, R. Zheng, M. Peng, J. Zhou, T. Gui, Q. Zhang, and X. Huang, “How robust is gpt-3.5 to predecessors? a comprehensive study on language understanding tasks,” 2023. [En línea]. Disponible en: <https://arxiv.org/abs/2303.00293>
- [13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019. [En línea]. Disponible en: <https://arxiv.org/abs/1810.04805>
- [14] J. Wang, F. Mo, W. Ma, P. Sun, M. Zhang, and J.-Y. Nie, “A user-centric benchmark for evaluating large language models,” 2024. [En línea]. Disponible en: <https://arxiv.org/abs/2404.13940>
- [15] “Lm studio,” 2024, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: <https://lmstudio.ai/>
- [16] Y. B.-G. y R. A. Challenger-Pérez, Ivet Díaz-Ricardo, “El lenguaje de programación python,” *Ciencias Holguín*, 2014. [En línea]. Disponible en: <https://www.redalyc.org/articulo.oa?id=181531232001>
- [17] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” 2019. [En línea]. Disponible en: <https://arxiv.org/abs/1912.01703>
- [18] “Huggingface,” 2013, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: <https://huggingface.co/>
- [19] “Flask,” 2010, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: https://flask.palletsprojects.com/en/3.0.x/?source=post_page-----fbc6ec9aa5f7-----

- [20] K. Reitz, “Requests: Http for humans,” 2024. [En línea]. Disponible en: <https://requests.readthedocs.io/en/latest/>
- [21] “Git.” [En línea]. Disponible en: <https://git-scm.com/>
- [22] “Getting started with github desktop.” [En línea]. Disponible en: <https://docs.github.com/en/desktop/overview/getting-started-with-github-desktop>
- [23] J. Tan, Y. Chen, and S. Jiao, “Visual studio code in introductory computer science course: An experience report,” 2023. [En línea]. Disponible en: <https://arxiv.org/abs/2303.10174>
- [24] “Lucidchart,” 2024, consultado el 11 de septiembre de 2024. [En línea]. Disponible en: <https://www.lucidchart.com/pages/es>
- [25] “Docker compose overview.” [En línea]. Disponible en: <https://docs.docker.com/compose/>
- [26] “Apache jmeter.” [En línea]. Disponible en: <https://jmeter.apache.org/>
- [27] E. Perkins, “Local llms part 3 – linux,” 2024. [En línea]. Disponible en: <https://blogs.iuvotech.com/local-llms-part-3-linux#:~:text=Linux%20Distribution&text=If%20this%20is%20an%20organization's,area%20we%20can%20assist%20in.>