



GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Diseño y creación de un LLM-ChatBot genérico-especializado

Autor

Juan Carlos Vecino de Haro

Dirigido por

Mario Castro Ponce

Madrid
May 2024



GRADO EN INGENIERÍA EN TECNOLOGÍAS DE
TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

**Diseño y creación de un LLM-ChatBot
genérico-especializado**

Autor

Juan Carlos Vecino de Haro

Dirigido por

Mario Castro Ponce

Madrid
May 2024

Resumen

0.1. Resumen ejecutivo

En este Trabajo Fin de Grado titulado 'Diseño y creación de un LLM-Chatbot genérico-especializado', se ha desarrollado la arquitectura de un Chatbot genérico con el objetivo de minimizar el coste de desarrollo de nuevas funcionalidades: determinar la necesidad del cliente, ejecutar acciones, como la compra de billetes de avión, todo ello con sin la necesidad de programar. Los resultados de las pruebas indican que el Chatbot puede resolver preguntas y ejecutar acciones con un desempeño comparable al de un ser humano.

Para saber la calidad de desempeño del chatbot, se han desarrollado una serie de pruebas automáticas para analizar la latencia y la precisión de las acciones. Además, en base a este análisis, se han desarrollado posibles extensiones avanzadas de este trabajo, para mejorar el Chatbot.

0.2. Introducción

La inteligencia artificial está experimentando un crecimiento exponencial, especialmente en el ámbito de los modelos de lenguaje de gran escala (LLM) como GPT-4, Llama3 y Claude3. Las empresas están realizando inversiones significativas en estos modelos debido a su enorme potencial. Un ejemplo destacado es la inversión de 10.000 millones de euros de Meta en OpenAI. Se prevé que el mercado de los LLM alcance los 4,35 billones de dólares para el año 2030. Además, se estima que el 15 % de la fuerza laboral en Estados Unidos podría realizar entre el 47 % y el 56 % de su trabajo de manera más rápida y eficiente gracias a estos modelos.

Diversos casos de uso, como los asistentes de correo electrónico y la automatización del servicio al cliente, demuestran la utilidad de los LLM en la vida cotidiana. Estos avances abren nuevas oportunidades para la automatización de la atención al cliente, lo que permite reducir costes y mejorar la eficiencia.

0.3. Definición del proyecto

El objetivo de este proyecto es desarrollar un ChatBot autónomo basado en modelos LLM para realizar diversas tareas de atención al cliente. El ChatBot se enfocará en la personalización sin necesidad de reentrenamiento continuo, integrará diversos prompts y utilizará múltiples LLM. Además, se implementará un sistema de agentes especializados para validar y asegurar la calidad de cada modelo. La meta es automatizar la atención al cliente de una empresa, optimizando sus costes operativos y mejorando la experiencia del usuario en sus interacciones con los servicios de atención al cliente.

0.4. Chatbot

Para desarrollar un Chatbot autónomo basado en modelos LLM se ha seguido un top down approach. Empezando por un LLM básico, al que se la han ido añadiendo herramientas y estructura para desarrollar un Chatbot autónomo.

0.4.1. Chatbot sin acciones

En primer lugar, se ha intentado automatizar tareas de atención al cliente con un LLM. Los resultados nos han demostrado que un LLM puede simular que realiza las tareas, pero realmente tiene una limitación muy grande al poder generar solamente texto sin ser procesado por una herramienta externa.

0.4.2. Chatbot con acciones

Para erradicar la flaqueza que tiene el LLM por si solo, se decide añadir agentes especializados y herramientas para que el Chatbot pueda realizar tareas de atención al cliente. Los resultados indica que el Chatbot puede resolver preguntas y ejecutar acciones con un desempeño comparable al de un ser humano. Sin embargo, el coste temporal y el conocimiento técnico necesario para modificar el prompt, las herramientas o el modelo LLM son muy elevados, lo que llevaría a que solamente gente experta podría modificar este Chatbot.

0.4.3. Chatbot general

En este último caso, se analiza la arquitectura que se debería de desarrollar para minimizar el coste temporal y no ser necesario tener conocimientos previos de programación. Se ha desarrollado una arquitectura de base de datos modular, con la cual podemos modificar o crear herramientas, LLM o prompts de forma simple y con un coste temporal mínimo. Los resultados demuestran cómo el diseño de arquitecturas de atención al cliente son más fáciles de crear.

Como prueba se ha desarrollado la siguiente arquitectura:

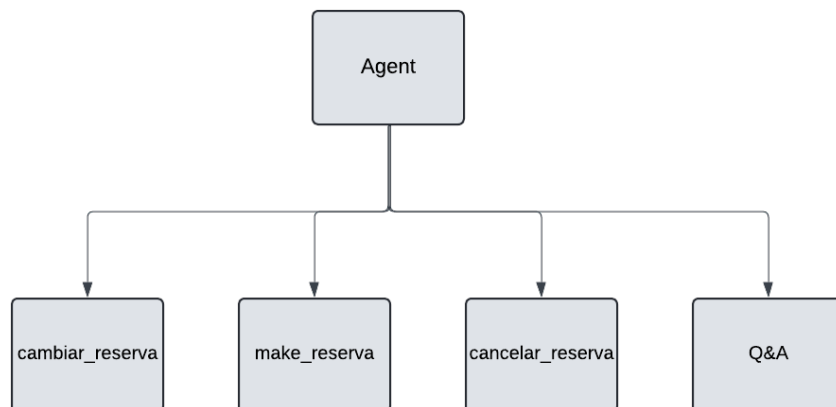


Figura 1: Arquitectura atención al cliente de un restaurante

Los resultados han seguido demostrando un desempeño comparable al de un ser humano, pero con un coste temporal mínimo y sin la necesidad de saber programación. A continuación se muestran los resultados de las pruebas:

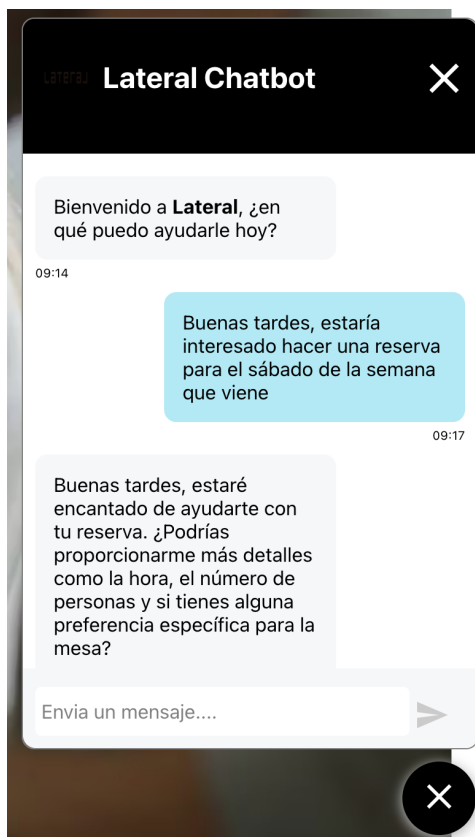


Figura 2: Captura de Pantalla 1

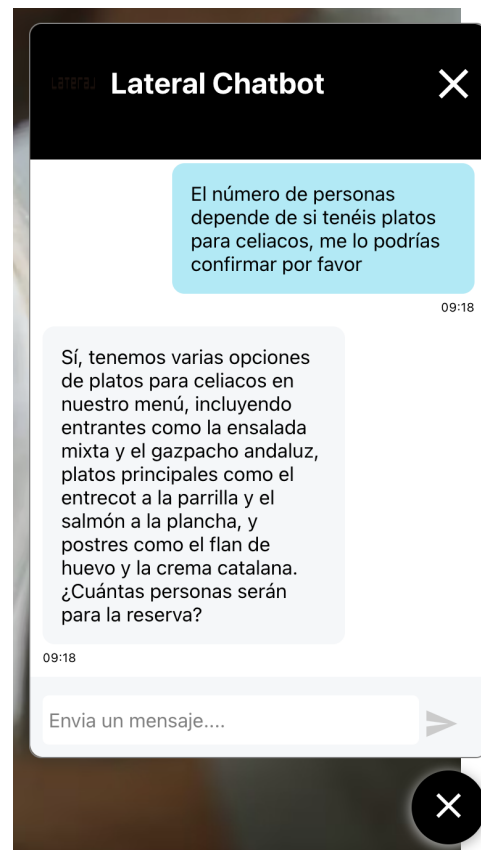


Figura 3: Captura de Pantalla 2

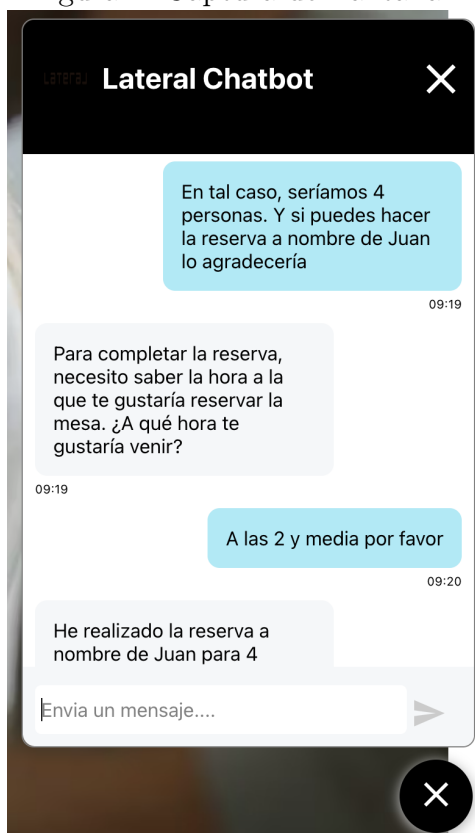


Figura 4: Captura de Pantalla 3

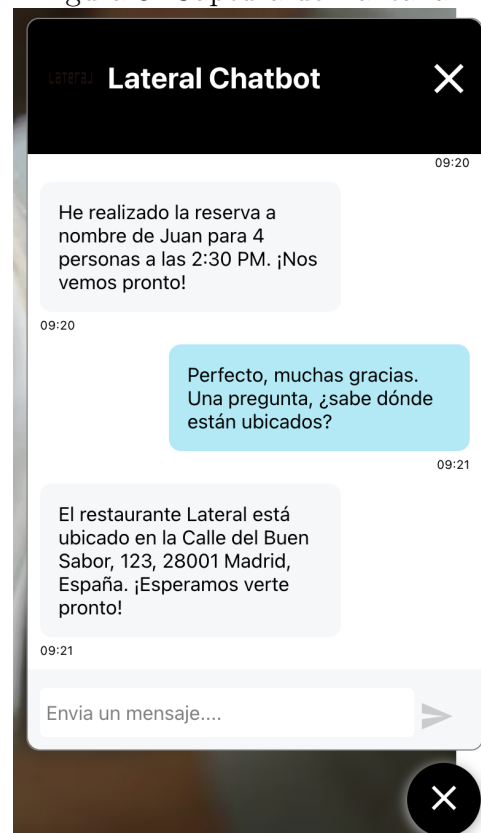


Figura 5: Captura de Pantalla 4

Figura 6: Prueba atención al cliente de un restaurante

0.5. Evaluador

Con el objetivo de saber la calidad del desempeño del Chatbot, se ha desarrollado un proceso de evaluación automático. Para el evaluador se han usado varios LLM, uno para crear la acción que se quiere simular, como comprar un billete de avión, otro emulando la conversación con nuestro Chatbot y por último un LLM que evalúe si nuestro Chatbot ha realizado con éxito la tarea.

0.6. Resultados y Conclusiones

En conclusión, los resultados han demostrado que el Chatbot puede resolver preguntas y ejecutar acciones con un desempeño comparable al de un ser humano. También se ha observado que el coste temporal para desarrollar una arquitectura y los costes operativos se ven minimizados, manteniendo un nivel de calidad al cliente similar o superior con los servicios de atención al cliente tradicionales.

No obstante, para mejorar la solución de automatización de atención al cliente, se recomienda implementar mecanismos de supervisión humana para monitorear y corregir respuestas, minimizar las alucinaciones de los LLM, desarrollar capacidades multilingües, integrar el chatbot con otras herramientas y plataformas de software, establecer un sistema de retroalimentación continuo para el aprendizaje y mejora constante, aplicar estrictas medidas de seguridad y privacidad para proteger la información del usuario, y realizar evaluaciones y pruebas regulares del rendimiento del chatbot para asegurar la calidad e identificar áreas de mejora.

0.7. Referencias

- [1] Grand View Research, *Large Language Model Market Size, Share & Trends Analysis Report By Application (Customer Service, Content Generation), By Deployment, By Industry Vertical, By Region, And Segment Forecasts, 2024 - 2030*, Grand View Research, Region-Specific, Market Research Report GVR-4-68040-186-2, 2024, Number of Pages: 100 Format: Electronic (PDF). Available: <http://www.grandviewresearch.com>.
- [2] J. Novet, *Microsoft's \$13 Billion Bet on OpenAI Carries Huge Potential Along with Plenty of Uncertainty*, CNBC, Published on April 8, 2023, Updated on April 9, 2023.
- [3] T. Eloundou, S. Manning, P. Mishkin and D. Rock, *GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models*, 2023. arXiv:2303.10130 [econ.GN].

Project Summary

0.8. Abstract

In this Final Degree Project entitled 'Design and creation of a generic-specialized LLM-Chatbot', the architecture of a generic Chatbot has been developed with the objective of minimizing the cost of developing new functionalities, such as purchasing airline tickets. Test results indicate that the Chatbot can solve questions and execute actions with performance comparable to that of a human being.

0.9. Introduction

Artificial intelligence is experiencing exponential growth, especially in the area of large-scale language models (LLMs) such as GPT-4, Llama3 and Claude3. Companies are making significant investments in these models due to their enormous potential. A prominent example is Meta's €10 billion investment in OpenAI. The LLM market is expected to reach \$4.35 trillion by 2030. In addition, it is estimated that 15 percent of the U.S. workforce could perform 47-56 percent of their work faster and more efficiently thanks to these models.

Several use cases, such as email assistants and customer service automation, demonstrate the usefulness of LLMs in everyday life. These advances open up new opportunities for customer service automation, leading to lower costs and improved efficiency.

0.10. Project definition

The objective of this project is to develop an autonomous ChatBot based on LLM models to perform various customer service tasks. The ChatBot will focus on personalization without the need for continuous retraining, integrate various prompts and use multiple LLMs. In addition, a system of specialized agents will be implemented to validate and ensure the quality of each model. The goal is to automate a company's customer service, optimizing its operational costs and improving the user experience in their interactions with customer services.

0.11. Chatbot

To develop a standalone Chatbot based on LLM models a top down approach has been followed. Starting with a bare LLM and adding tools to develop a standalone Chatbot.

0.11.1. Chatbot without actions

First, an attempt has been made to automate customer service tasks with an LLM. The results have shown us that an LLM can simulate that it performs the tasks, but it really has a huge limitation in that it can only generate text without being processed by an external tool.

0.11.2. Chatbot with actions

To eradicate the weakness that LLM alone has, it is decided to add specialized agents and tools for the Chatbot to perform customer service tasks. The results indicate that the Chatbot can solve questions and execute actions with performance comparable to that of a human being. However, the time cost and technical knowledge required to modify the prompt, tools or LLM model are very high. This would lead to the fact that only expert people would be able to modify this Chatbot.

0.11.3. General chatbot

In this last case, we analyze the architecture that should be developed to minimize the time cost and not be necessary to have previous programming knowledge. A modular database architecture has been developed, with which we can modify or create tools, LLM or prompts in a simple way and with a minimum time cost. The results demonstrate how the design of customer service architectures are easier to create.

As a test, the following architecture has been developed:

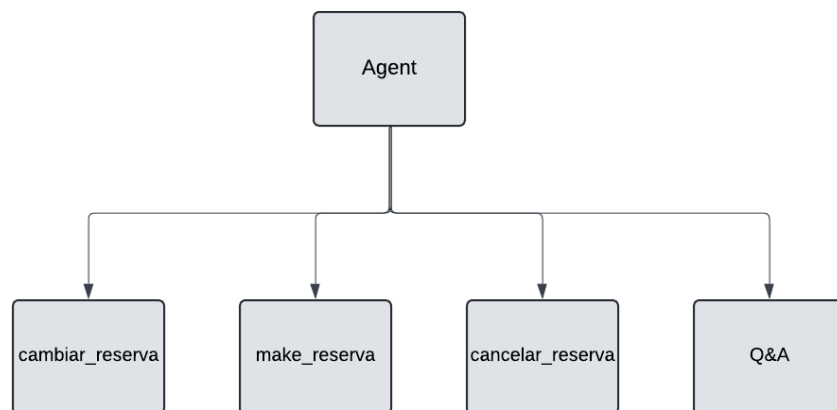


Figura 7: Arquitectura atención al cliente de un restaurante

The results have continued to demonstrate performance comparable to that of a human, but with minimal time cost and without the need for programming knowledge. The test results are shown below:

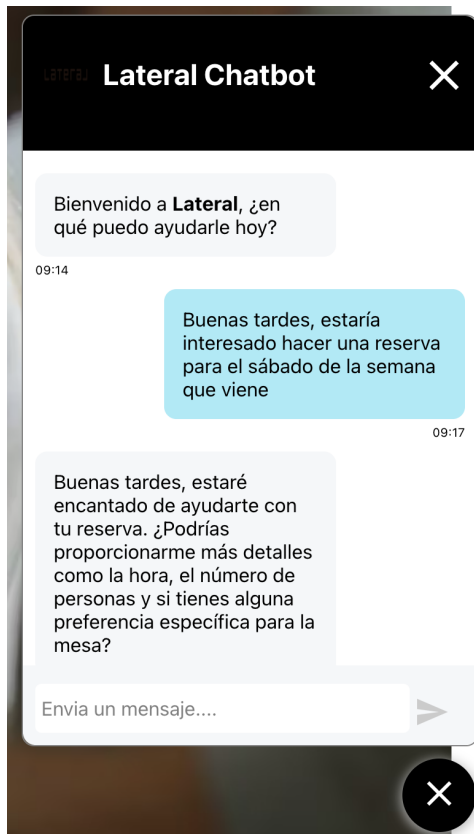


Figura 8: Captura de Pantalla 1

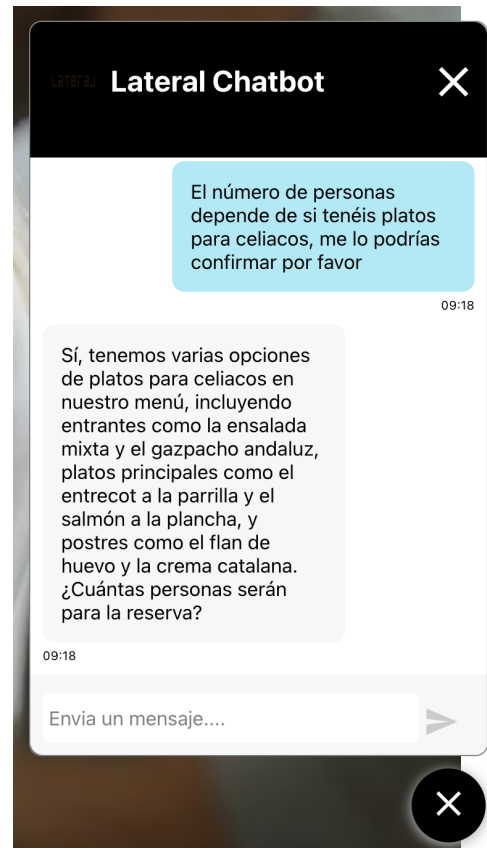


Figura 9: Captura de Pantalla 2

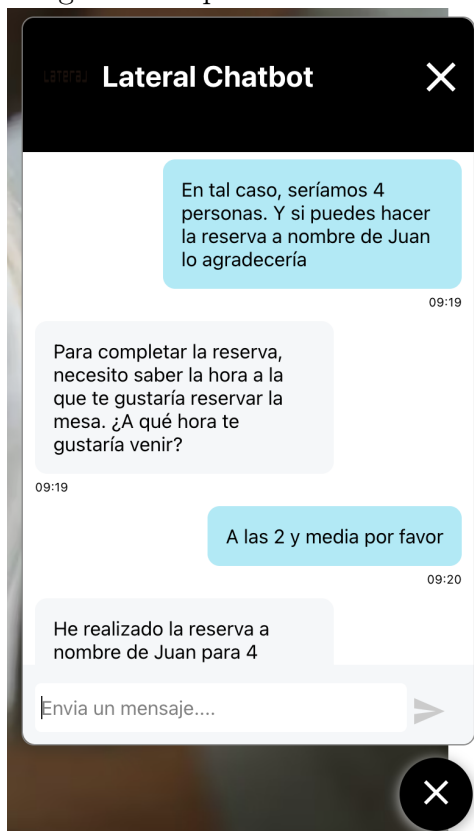


Figura 10: Captura de Pantalla 3

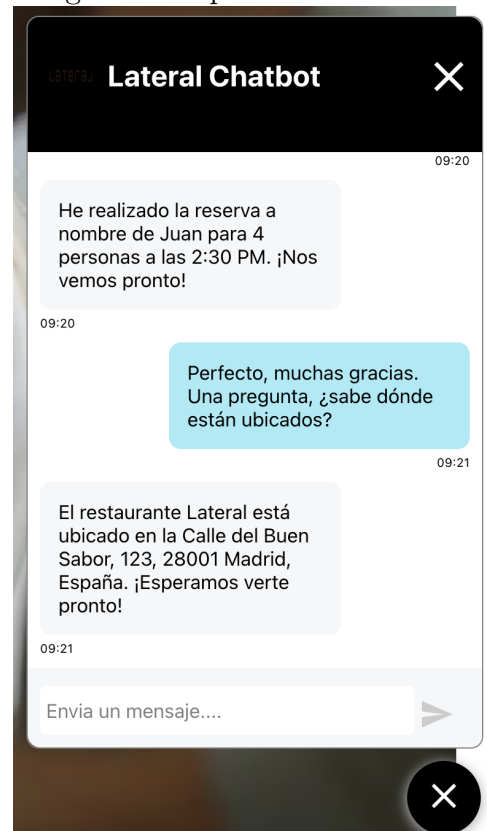


Figura 11: Captura de Pantalla 4

Figura 12: Prueba atención al cliente de un restaurante

0.12. Evaluator

In order to know the quality of the Chatbot performance, an automatic evaluation process has been developed. For the evaluator, several LLMs have been used, one to create the action to be simulated, such as buying a plane ticket, another one emulating the conversation with our Chatbot and finally an LLM that evaluates if our Chatbot has successfully performed the task.

0.13. Results and Conclusions

In conclusion, the results have shown that the Chatbot can solve questions and execute actions with performance comparable to that of a human being. It has also been observed that the time cost to develop an architecture and operational costs are minimized, maintaining a similar or higher level of customer quality with traditional customer services.

However, to improve the customer care automation solution, it is recommended to implement human supervision mechanisms to monitor and correct responses, minimize LLM hallucinations, develop multilingual capabilities, integrate the chatbot with other tools and software platforms, establish a continuous feedback system for constant learning and improvement, apply strict security and privacy measures to protect user information, and perform regular evaluations and testing of chatbot performance to ensure quality and identify area

0.14. References

- [1] Grand View Research, *Large Language Model Market Size, Share & Trends Analysis Report By Application (Customer Service, Content Generation), By Deployment, By Industry Vertical, By Region, And Segment Forecasts, 2024 - 2030*, Grand View Research, Region-Specific, Market Research Report GVR-4-68040-186-2, 2024, Number of Pages: 100 Format: Electronic (PDF). Available: <http://www.grandviewresearch.com>.
- [2] J. Novet, *Microsoft's \$13 Billion Bet on OpenAI Carries Huge Potential Along with Plenty of Uncertainty*, CNBC, Published on April 8, 2023, Updated on April 9, 2023.
- [3] T. Eloundou, S. Manning, P. Mishkin and D. Rock, *GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models*, 2023. arXiv:2303.10130 [econ.GN].

Agradecimientos

Muchas gracias tanto a mi padre como a mi tío Carlos por apoyarme siempre con mis ideas.

Índice general

Resumen	III
0.1. Resumen ejecutivo	III
0.2. Introducción	III
0.3. Definición del proyecto	III
0.4. Chatbot	IV
0.4.1. Chatbot sin acciones	IV
0.4.2. Chatbot con acciones	IV
0.4.3. Chatbot general	IV
0.5. Evaluador	VII
0.6. Resultados y Conclusiones	VII
0.7. Referencias	VII
Project Summary	IX
0.8. Abstract	IX
0.9. Introduction	IX
0.10. Project definition	IX
0.11. Chatbot	IX
0.11.1. Chatbot without actions	X
0.11.2. Chatbot with actions	X
0.11.3. General chatbot	X
0.12. Evaluator	XII
0.13. Results and Conclusions	XII
0.14. References	XII
1. Introducción	1
1.1. Motivación	1
1.2. Objetivo	2
2. Marco Teórico y estado del arte	3
2.1. Modelo Generativo	3
2.1.1. Embedding	4
2.1.2. Atención y Codificación Posicional	6
2.2. Transformers	8
2.2.1. Generative Pre-trained Transforme (GPT)	10
2.2.2. Large Language Models (LLM)	11
2.3. Agents and Tools	11
2.4. Retrieval-Augmented Generation (RAG)	12
2.4.1. Funcionamiento de RAG	12
2.4.2. Ejemplo Práctico	13

2.4.3.	Bases de Datos Vectoriales	14
2.5.	AutoGPT: Automatización de Agentes y Herramientas	14
2.5.1.	Funcionamiento de AutoGPT	14
2.6.	Estado del Arte	15
3.	Metodología	17
3.1.	Cuestiones técnicas	17
3.2.	Evaluador	17
3.2.1.	Evaluación Objetiva	17
4.	Chatbot	19
4.1.	Creación Chatbot sin acciones	19
4.2.	Creación Chatbot con acciones	20
4.3.	Chatbot general	23
4.3.1.	Parametrización	23
4.3.2.	Modificación del código	25
4.3.3.	Ejemplo	27
5.	Evaluador	31
5.1.	Evaluador del Chatbot	31
5.1.1.	Extract Information	32
5.1.2.	Create Action	33
5.1.3.	AutoGPT	33
5.1.4.	Evaluate	34
5.2.	Ejemplo	35
5.2.1.	Extract Information	35
5.2.2.	Create Action	35
5.2.3.	AutoGPT	36
5.2.4.	Evaluate	36
6.	Análisis de Resultados	37
6.1.	Precisión	37
6.1.1.	Valoración de los Resultados	37
6.2.	Facilidad de Crear un Chatbot	37
6.2.1.	Alucinaciones	37
6.2.2.	Robustez cuando hay muchas <i>Tools</i>	38
6.2.3.	Generalización	38
6.2.4.	Latencia	38
6.2.5.	Tareas sencillas	39
7.	Conclusiones	41
7.1.	Posibles extensiones avanzadas de este trabajo	41
	Appendix	42
	A. Ockam.py	43
	B. Tools.py	47

C. Api.py	51
D. Información Langchain Smith	55
E. Alineación con los Objetivos de Desarrollo Sostenible (ODS)	57
E.1. ODS 8: Trabajo Decente y Crecimiento Económico	57
E.2. ODS 9: Industria, Innovación e Infraestructura	57
Bibliografía	59
Bibliografía	59

Índice de figuras

1.	Arquitectura atención al cliente de un restaurante	IV
2.	Captura de Pantalla 1	VI
3.	Captura de Pantalla 2	VI
4.	Captura de Pantalla 3	VI
5.	Captura de Pantalla 4	VI
6.	Prueba atención al cliente de un restaurante	VI
7.	Arquitectura atención al cliente de un restaurante	X
8.	Captura de Pantalla 1	XI
9.	Captura de Pantalla 2	XI
10.	Captura de Pantalla 3	XI
11.	Captura de Pantalla 4	XI
12.	Prueba atención al cliente de un restaurante	XI
1.1.	Email Assitant	1
2.1.	Ejemplo de un modelo generativo capaz de generar imágenes nuevas de caballos	3
2.2.	Arquitectura de embedding Skip-gram	5
2.3.	Ejemplo de Word2Vec con la palabra 'Spain'	6
2.4.	Ejemplo de una capa de Multihead Attention	8
2.5.	Bloque de Codificador Transformer	9
2.6.	Modelo Transformer Codificador-Decodificador	9
2.7.	Arquitectura GPT	10
2.8.	Ejemplo de un agente utilizando herramientas para responder a una entrada	12
2.9.	Flow de RAG	13
2.10.	Proceso de Auto-GPT	14
4.1.	Ejemplo Agente con Tools con Langchain	22
4.2.	Estructura árbol chatbot restaurante	22
4.3.	Conversación probando el chatbot de un restaurante	23
4.4.	Payload conversación	23
4.5.	Diagrama Base de Datos	25
4.6.	Estructura árbol chatbot restaurante v2.0	28
4.7.	Captura de Pantalla 1	30
4.8.	Captura de Pantalla 2	30
4.9.	Captura de Pantalla 3	30
4.10.	Captura de Pantalla 4	30
4.11.	Prueba atención al cliente de un restaurante	30

5.1.	Diagrama de flujo del Evaluador	32
5.2.	Conversación AutoGPT con Chatbot	36
6.1.	Histograma Latencia gpt-3.5-turbo-1106	38
6.2.	Histograma Latencia gpt-4-turbo	39
D.1.	Captura de Pantalla de LangChain Smith con los proyectos de los 2 Chatbots	55
D.2.	Captura de Pantalla 2 de LangChain Smith con los proyectos de los 2 Chatbots	55

Índice de cuadros

2.1. Tipos de arquitecturas de Transformers	10
3.1. Criterios de evaluación para la simulación de conversaciones	18
4.1. Descripción de los parámetros de la tabla Chatbot	24
4.2. Descripción de los parámetros de la tabla Tool	24
4.3. Descripción de los parámetros de la tabla Params	24
4.4. Descripción tabla Historial de la conversación	25
6.1. Precisión de los modelos LLM	37
6.2. Latencia de los modelos LLM	38

Listings

A.1. Ockam.py	43
B.1. Tools.py	47
C.1. Api.py	51

Acrónimos

NLP Natural language processing
LLM Large Language Model

Capítulo 1

Introducción

1.1. Motivación

La inteligencia artificial ha experimentado un crecimiento exponencial en los últimos años. Con el reciente lanzamiento de modelos de lenguaje de gran escala (LLM) como GPT-4, Llama3 y Claude3, las empresas han reconocido su gran potencial. Están invirtiendo millones de euros en la creación de estos modelos; por ejemplo, Meta ha invertido 10.000 millones de euros en OpenAI para desarrollar sus modelos.[2].

Diversos artículos predicen que el mercado de los LLM llegará a los 4,35 billones de dólares[1] en 2030. Respecto a lo que se refiere la fuerza laboral, se cree que el 15 % del workforce de USA podría realizar entorno al 47 %-56 % de su trabajo más rápido y manteniendo la calidad con el uso de un LLM[3].

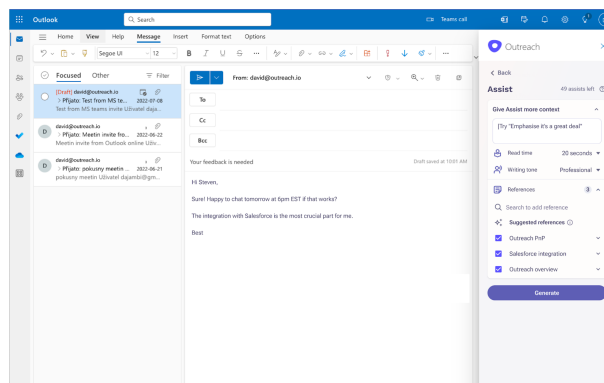


Figura 1.1: Email Assitant

Diferentes casos de uso están saliendo para usar los modelos LLM en nuestro día a día, como 1.1 el cual, con la información del email más referencias se pueden generar mails de forma automática. Otros casos de uso ,cómo el de Keivalya Pandya, hablan de como automatizar el customer service de una empresa con Langchain y una Base de Datos vectorial como Chroma, respondiendo a las preguntas del cliente con la información de la empresa.

1.2. Objetivo

En la era digital actual, la interacción con servicios de atención al cliente es una parte integral de nuestra vida cotidiana. Añadir una maleta extra a nuestro vuelo con Iberia, reservar una mesa en nuestro restaurante favorito, hasta consultar sobre las políticas de viaje con mascotas en Ouigo, nos encontramos constantemente en la necesidad de comunicarnos con las empresas para obtener información o realizar acciones específicas. Tradicionalmente, estas interacciones han requerido de equipos de atención al cliente compuestos por personas entrenadas para responder y ejecutar solicitudes de manera eficaz. Sin embargo, esta solución conlleva costes significativos para las empresas, no solo en términos salariales, sino también en formación y mantenimiento de dichos equipos.

El 12 de junio de 2017, Google dio a conocer al mundo la arquitectura Transformer a través del paper 'Attention Is All You Need' [6], sentando las bases para una revolución en el campo del procesamiento del lenguaje natural (NLP). Esta innovación ha permitido el desarrollo de modelos de lenguaje de gran escala (LLM), como GPT-3 de OpenAI, que son capaces de entender y generar texto con un nivel de coherencia y relevancia hasta entonces inimaginable. Estos avances abren un abanico de posibilidades para la automatización de la atención al cliente, ofreciendo una alternativa más rápida y económica a los métodos tradicionales.

El objetivo de este proyecto es diseñar y desarrollar un ChatBot autónomo basado en modelos LLM, que sea capaz de llevar a cabo una amplia gama de acciones de atención al cliente. Este ChatBot buscará combinar la eficiencia y la capacidad de personalización, con el fin de adaptarse a diversas tareas sin incurrir en costes elevados de re-entrenamiento. Al hacer uso de la tecnología más avanzada en el campo de la inteligencia artificial, nuestro proyecto no solo busca optimizar los costes operativos para las empresas, sino también mejorar la experiencia de los usuarios al interactuar con servicios de atención al cliente.

Para la creación de un ChatBot robusto y eficiente, este proyecto se centrará en tres pilares fundamentales: **la personalización** sin la necesidad de re-entrenamiento continuo, la integración de diversos prompts y el **uso de múltiples LLM** (Modelos de Lenguaje de Gran Escala) y, por último, la implementación de un **sistema de agentes especializados encargados de validar** y asegurar la calidad de cada modelo.

Capítulo 2

Marco Teórico y estado del arte

2.1. Modelo Generativo

Un modelo generativo es un tipo de modelo estadístico diseñado para generar nuevas instancias de datos basadas en la distribución de probabilidad aprendida de un conjunto de datos de entrenamiento. Estos modelos se entrenan para generar nuevas instancias que no se han visto durante el entrenamiento, a partir de las características aprendidas.

Por ejemplo, si entrenamos un modelo generativo con un conjunto de datos compuesto por fotos de caballos, el modelo aprenderá características inherentes a estas imágenes sin tener un conocimiento explícito de lo que es un caballo. Esto incluye detalles como la presencia de cuatro patas, dos ojos y dos orejas, y el contexto típico, como un campo. A pesar de no tener un entendimiento conceptual de estos elementos, el modelo captura estas complejidades a partir de los píxeles en las imágenes y es capaz de generar nuevas imágenes realistas de caballos que no existían en el conjunto de datos original. Un ejemplo visual se puede ver en la siguiente figura:

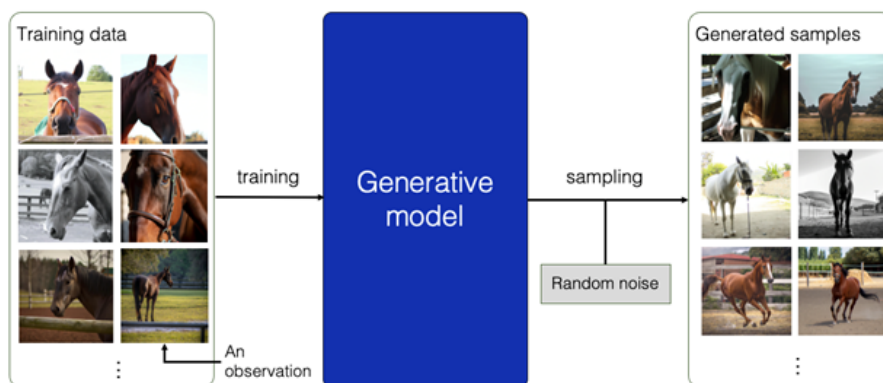


Figura 2.1: Ejemplo de un modelo generativo capaz de generar imágenes nuevas de caballos

Es importante destacar que los modelos generativos son *probabilísticos* y no *determinísticos*. Esto significa que para una misma entrada, el modelo puede generar múltiples salidas posibles, cada una representando una variación plausible basada en la distribución aprendida. En contraste, un modelo determinista generaría siempre la misma salida para una entrada dada.

Para saber como funciona un modelo generativo por debajo primero debemos saber los siguientes conceptos:

2.1.1. Embedding

Al igual que una red neuronal no interpreta directamente un píxel, sino que trabaja con sus componentes RGB, tampoco entiende texto o caracteres directamente. Para hacer que el texto sea procesable por una red neuronal, necesitamos convertirlo en una forma numérica mediante un proceso llamado tokenización.

Una técnica inicial para la tokenización es el *One-hot encoding*, donde cada palabra en el vocabulario se representa como un vector en un espacio de alta dimensión. En este espacio, cada dimensión corresponde a una palabra del vocabulario, de modo que cada palabra se representa con un vector donde un único elemento es 1 y todos los demás son 0. Sin embargo, este método tiene limitaciones significativas, ya que no captura relaciones semánticas y es computacionalmente ineficiente para vocabularios grandes. Por ejemplo, si consideramos un vocabulario que consiste solo en las palabras 'hogar', 'casa' y 'cohete', los vectores de one-hot serían:

$$\text{hogar} = [1, 0, 0]$$

$$\text{casa} = [0, 1, 0]$$

$$\text{cohete} = [0, 0, 1]$$

Como se observa, la distancia euclidiana entre cada par de vectores es la misma, indicando que están igualmente distanciados sin importar sus posibles relaciones semánticas.

$$d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2} \quad (2.1)$$

Para abordar estas limitaciones, Google desarrolló el modelo *Word2Vec*[4], un enfoque de procesamiento de lenguaje natural (NLP) que mapea directamente cada palabra a un vector en un espacio de baja dimensionalidad (típicamente entre 100 y 300 dimensiones), capturando relaciones semánticas y sintácticas entre las palabras.

Word2Vec utiliza arquitecturas como Skip-gram o CBOW (Continuous Bag of Words). En el caso de Skip-gram, el modelo intenta predecir el contexto a partir de una palabra central. Por ejemplo, a partir de la palabra 'river', el modelo podría predecir palabras como 'water' y 'flow'.

Este modelo se entrena utilizando el método de Softmax Jerárquico, que utiliza un árbol binario de Huffman para organizar el vocabulario, donde cada palabra es una hoja del árbol. El modelo comienza desde la raíz del árbol y busca la probabilidad de llegar a una palabra específica.

$$P(w) = \prod_{n \in \text{path to } w} P(n) \quad (2.2)$$

donde $P(n)$ es la probabilidad de tomar la decisión correcta en el nodo n para seguir el camino hacia la palabra w .

En el caso de tener un vocabulario con las palabras 'manzana', 'banana', 'ciudad', 'coche' y 'árbol', en lugar de comparar 'manzana' con todas las demás palabras para ver cuán probable es que aparezca, simplemente seguimos un camino en el árbol:

Primero decidimos entre 'cosas comestibles' y 'no comestibles', eligiendo 'cosas comestibles'. Luego, entre 'frutas' y 'verduras', eligiendo 'frutas'. Y finalmente, llegamos a

'manzana'. Este método simplificado nos permite calcular rápidamente la probabilidad de 'manzana' sin tener que hacer comparaciones con cada palabra del vocabulario.

El entrenamiento se realiza usando técnicas de optimización como el descenso de gradiente estocástico. En cada iteración, se selecciona una palabra central junto con varias palabras de contexto, y se ajustan los vectores de palabras para maximizar la probabilidad condicional de las palabras de contexto dadas las palabras centrales usando la siguiente fórmula en el modelo Skip-gram:

$$p(w_{\text{context}} | w_{\text{central}}) = \frac{\exp(\text{vec}(w_{\text{central}}) \cdot \text{vec}(w_{\text{context}}))}{\sum_{w \in \text{vocab}} \exp(\text{vec}(w_{\text{central}}) \cdot \text{vec}(w))} \quad (2.3)$$

Donde $\text{vec}(w)$ representa el vector de la palabra w , y el producto escalar entre los vectores indica la similitud. La pérdida se calcula comparando esta probabilidad con la distribución ideal (1 para las palabras de contexto real, 0 para las demás), promoviendo que palabras semánticamente similares tengan productos escalares altos, reflejando una alta similitud en el espacio de embedding.

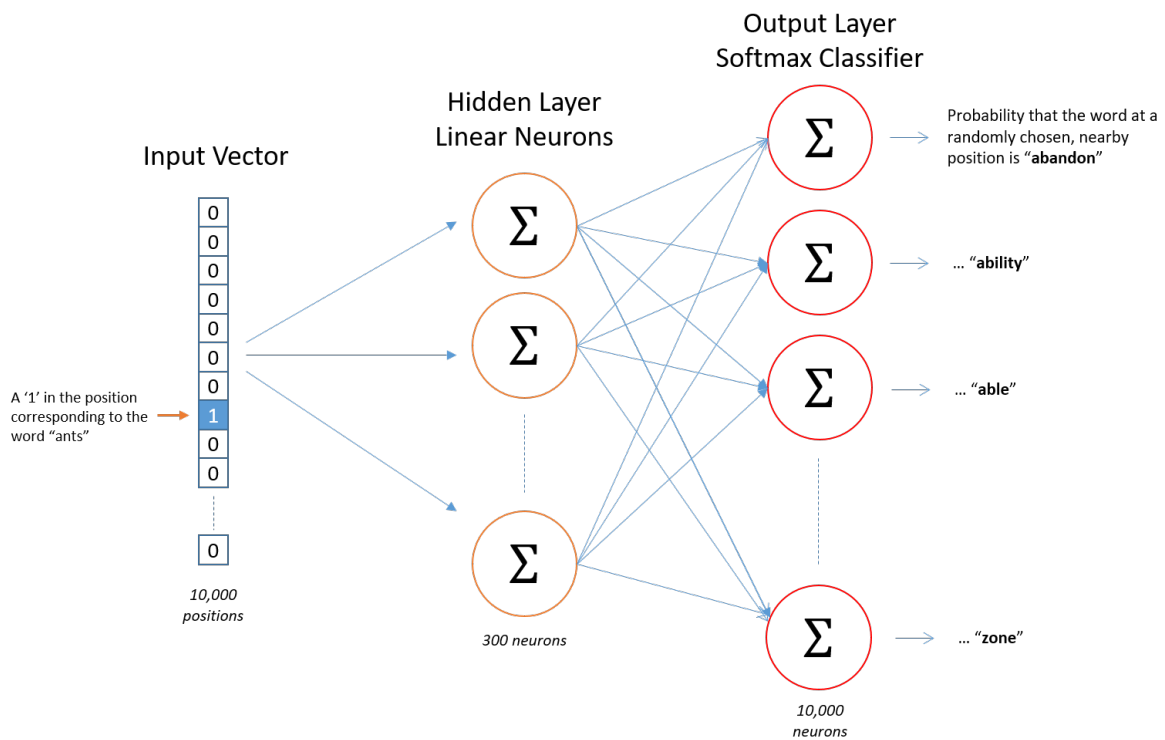


Figura 2.2: Arquitectura de embedding Skip-gram

De tal forma que al final del entrenamiento, palabras con relaciones sintácticas y semánticas similares estén más cerca en el espacio dimensional. Cada dimensión del vector deja de significar una palabra, sino que puede representar conceptos abstractos como 'felicidad' o 'tiempo'. Ahora, la distancia euclidiana entre vectores no es la misma, teniendo aquellas palabras similares una menor distancia entre ellas. A continuación se muestra un ejemplo de vectores similares a la palabra 'Spain' en Word2Vec 10K:

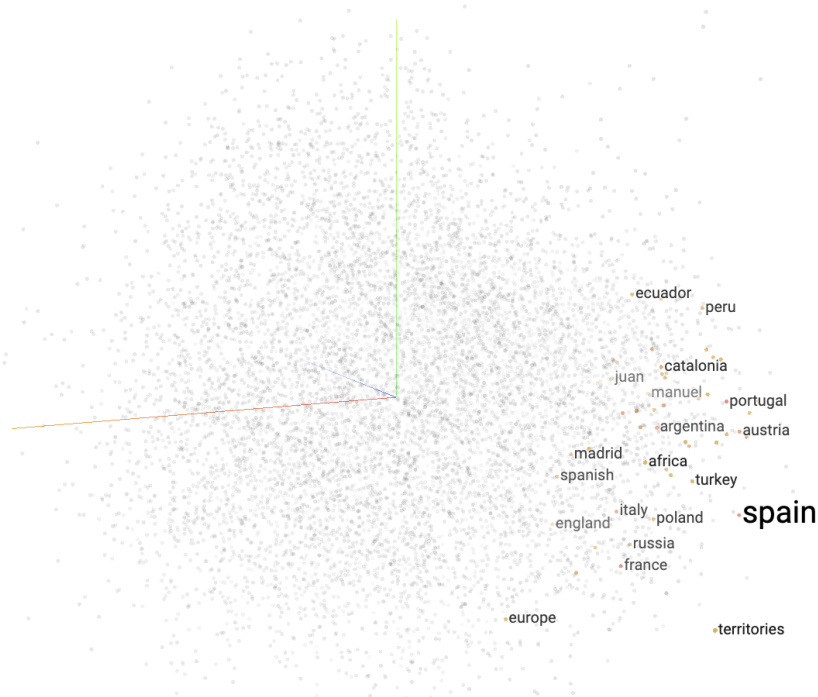


Figura 2.3: Ejemplo de Word2Vec con la palabra 'Spain'

Como podemos observar, palabras como 'portugal', 'francia', y 'spanish' aparecen dentro de los 50 vectores más cercanos a 'Spain'.

Por lo tanto, los embeddings son una manera de reducir la dimensionalidad y, al mismo tiempo, aumentar la capacidad del modelo para interpretar y procesar el lenguaje de manera que refleje mejor la complejidad y la riqueza semántica.

2.1.2. Atención y Codificación Posicional

Los embeddings convencionales de palabras no consideran dos aspectos cruciales: la posición de las palabras dentro de la secuencia y el contexto proporcionado por las demás palabras en la oración. Para ilustrar cómo se abordan estas limitaciones, consideremos la frase:

“Los bomberos llegaron porque la casa verde estaba en”

Intuitivamente, esperamos que la siguiente palabra sea *“llamas”*. Esta predicción se facilita al otorgar relevancia a palabras clave como *“bomberos”*, *“llegaron”*, *“casa”* y *“estaba”*, mientras se minimiza la atención a palabras menos críticas como *“los”*, *“verde”* y *“porque”*. El mecanismo de atención está diseñado para ponderar la relevancia de cada palabra en el contexto de la entrada, mejorando así la predicción de palabras futuras.

El proceso comienza con la selección de una palabra, por ejemplo, *“en”*. Esta palabra se convierte en la *query* (Q), que busca información relevante en el contexto de la frase. La query se calcula de la siguiente manera:

$$Q = X_{\text{en}} W^Q \quad (2.4)$$

Donde Q representa la query, X_{en} es el embedding de la palabra *“en”*, y W^Q es una matriz de transformación aprendida durante el entrenamiento, diseñada para extraer información específica del embedding que responde a la query.

Simultáneamente, las demás palabras en la oración se procesan para obtener los *key vectors* y los *value vectors*. Los *key vectors* indican en qué circunstancias puede una palabra ser relevante para el *query*, mientras que los *value vectors* representan la información real contenida en cada palabra.

Estos se calculan de la siguiente manera:

$$K = X_{\text{word}}W^K, \quad V = X_{\text{word}}W^V \quad (2.5)$$

Donde K y V representan los vectores de key y value respectivamente, X_{word} es el embedding de una palabra específica en la oración, y W^K y W^V son matrices de transformación aprendidas durante el entrenamiento. Estas matrices están diseñadas para transformar el embedding inicial de cada palabra en vectores que facilitan la determinación de relevancia y la extracción de información contextual en el proceso de atención.

A continuación cada *key vector* se compara con la *query* mediante un producto escalar. Un valor alto en este producto QK^T indica una mayor relevancia de la palabra correspondiente con la *query*. Este resultado se normaliza aplicando la función softmax para que la suma de todos los productos escalares sea 1, obteniendo así los *attention weights* o pesos de atención, que cuantifican la importancia relativa de cada palabra.

$$\text{Attention Weights} = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) \quad (2.6)$$

Finalmente, estos pesos se multiplican por los *value vectors* correspondientes y se suman para obtener un vector de salida:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2.7)$$

El resultado de este cálculo de atención es una matriz donde cada fila es una combinación ponderada de los vectores de value, con los pesos dados por los puntajes de atención. Esto significa que cada salida de un attention head es un vector que es una suma ponderada de los vectores de value, basada en cuánta "atención" o relevancia se le da a cada posición de entrada.

Así como en una red convolucional donde cada capa se centra en una parte diferente de la imagen, cada *Attention Head* se enfoca en diferentes aspectos del texto. Por ello, es común emplear un *Multihead Attention*, donde cada cabeza aporta un mecanismo de atención diferente, permitiendo extraer relaciones más complejas entre las palabras.

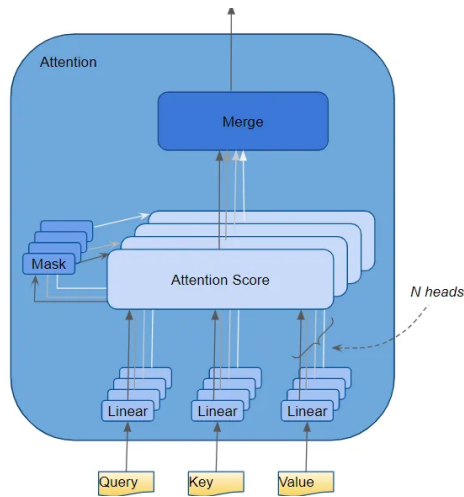


Figura 2.4: Ejemplo de una capa de Multihead Attention

Es importante recalcar que el embedding procesado en el *Attention Layer* incluye codificación posicional. No solo se realiza un embedding de la palabra, sino que también se añade información sobre la posición de la palabra en el texto, proporcionando así más contexto al modelo.

La salida de este *Attention Head* es información relevante y contextualizada, la cual nos va a servir para predecir la siguiente palabra.

2.2. Transformers

El modelo Transformer, introducido por Vaswani et al. en 2017[6], ha revolucionado el campo del procesamiento del lenguaje natural por su eficacia y eficiencia en la manipulación de secuencias largas de datos. Este modelo se basa completamente en mecanismos de atención para capturar dependencias a largo plazo sin la necesidad de elementos recurrentes, permitiendo así un procesamiento en paralelo y reduciendo significativamente los tiempos de entrenamiento. Algo que no conseguían otros modelos como *RNN*, *LSTM* o *GRU*.

La arquitectura del Transformer consta de dos componentes principales: el codificador y el decodificador.

Cada codificador en la pila del Transformer incluye una serie de capas, cada una con dos sub-capas principales. La primera es una capa de *Multihead Attention*, que permite al modelo atender a diferentes partes de la entrada simultáneamente. La segunda sub-capa es una red de feed-forward completamente conectada que opera de manera independiente en cada posición y en paralelo. Ambas sub-capas están seguidas por una conexión residual y una normalización de capa, contribuyendo así a estabilizar el aprendizaje y mejorar la convergencia del modelo.

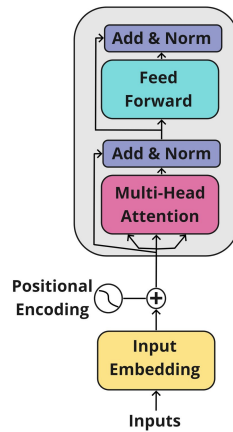


Figura 2.5: Bloque de Codificador Transformer

El decodificador está compuesto igualmente por una pila de capas idénticas. Además de las dos sub-capas que se encuentran en el codificador, incorpora una tercera sub-capas que realiza una *Multihead Attention Layer* sobre la salida del codificador. Esta sub-capas permite al decodificador enfocarse en partes relevantes de la entrada mientras genera la secuencia de salida. Al igual que en el codificador, cada sub-capas está equipada con una conexión residual y una normalización de capa.

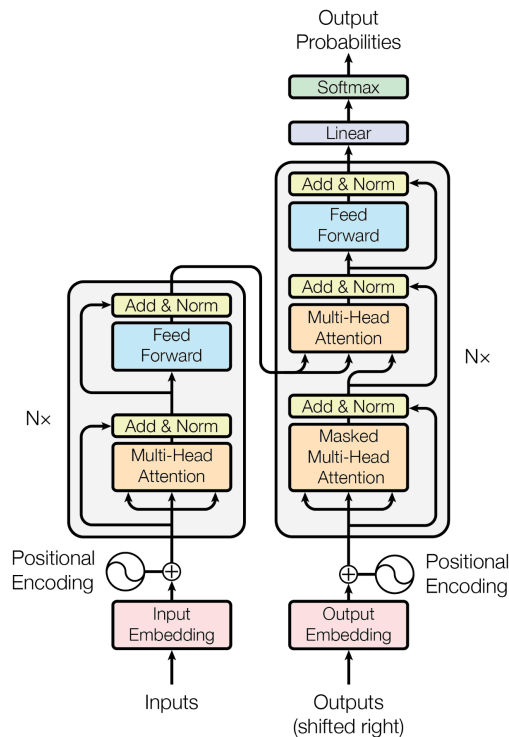


Figura 2.6: Modelo Transformer Codificador-Decodificador

En la parte izquierda del diagrama está el *codificador*, que transforma la secuencia de entrada en información contextual. Este componente no está diseñado para predecir texto por sí mismo.

En el lado derecho, el *decodificador* utiliza la información del *codificador* y la salida generada en la etapa anterior para predecir la siguiente palabra posible en la secuencia.

En el lado derecho, el *decodificador* utiliza la información del *codificador* y la salida generada en la etapa anterior para predecir la siguiente palabra posible en la secuencia. La salida del último bloque del decodificador se proyecta a través de una capa lineal para ajustar las dimensiones al tamaño del vocabulario inicial y se pasa a través de una función softmax, convirtiendo los logits¹ en probabilidades de palabras. Esto permite que el modelo prediga la siguiente palabra en la secuencia basándose en el contexto y la información codificada.

Este es un modelo de Transformer de tipo codificador-decodificador. Sin embargo, existen varios modelos adaptados a diferentes casos de uso:

Tipo	Ejemplo	Caso de Uso
Codificador	BERT	Clasificación
Codificador-Decodificador	T5	Traducción de frases
Decodificador	GPT-3	Generación de texto

Cuadro 2.1: Tipos de arquitecturas de Transformers

2.2.1. Generative Pre-trained Transformer (GPT)

Como se ha comentado en la Tabla 2.1 este modelo utiliza exclusivamente un decodificador para la generación de texto. A continuación se presenta una representación simplificada de su arquitectura:

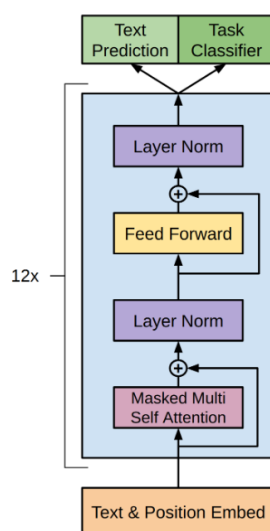


Figura 2.7: Arquitectura GPT

En el caso de GPT-3 el input es una secuencia de hasta 2048 tokens (Aunque en el caso de gpt-3 OpenAI Byte-level BPE tokenizer, podemos asumir 2048 palabras), luego GPT-3 nos devuelve una predicción de las siguientes 2048 palabras subsiguientes. En nuestro caso

¹En modelos de aprendizaje automático como los Transformers, los *logits* son básicamente los valores de salida de la última capa del modelo antes de que cualquier transformación final (como la función softmax) los convierta en probabilidades. Estos valores representan las puntuaciones brutas que cada posible próxima palabra recibe, indicando cuán probable es cada una antes de convertir estas puntuaciones en una distribución de probabilidad normalizada.

si queremos predecir la siguiente palabra solamente haríamos caso a la última posición de este vector.

El proceso comienza introduciendo una frase como entrada, luego esta frase es pasada a un hot-encoding de OpenAI de 50.257 palabras. Como la dimensión de esta matriz es demasiado grande 2048×50257 reducimos la dimensión del vocabulario a través del mecanismo de embedding, convirtiendo la matriz en una de tamaño 2048×12.288 . Posteriormente la información pasa por las 96 *Masked Attention-Head* y luego se conecta con la capa residual y la normalización. Por último, como se ha explicado anteriormente la salida del último bloque del decodificador se proyecta a través de una capa lineal para ajustar las dimensiones al tamaño del vocabulario inicial 2048×5027 y se pasa a través de una función softmax, obteniendo las posibilidades de cada palabra.

2.2.2. Large Language Models (LLM)

A lo largo de este documento se van a usar diversos modelos LLM:

- GPT-3.5 de OpenAI
- Claude3

2.3. Agents and Tools

En esta sección, vamos a explorar cómo los modelos de Lenguaje de Aprendizaje Profundo (LLM) no solo generan texto, sino que también pueden ejecutar acciones específicas a través de lo que denominamos agentes y herramientas.

Un modelo LLM estándar por sí solo está limitado a generar texto basado en el input recibido. No puede realizar tareas más complejas, como buscar información en Internet. Para superar esta limitación, se han desarrollado los *agentes*, que emplean una técnica conocida como *prompt engineering* para manipular la salida del modelo de manera que pueda interactuar con diversas herramientas (tools). Este concepto fue introducido en profundidad por Schick et al. en 2023[8].

En la Figura 2.8, se muestra cómo un agente utiliza una herramienta de QA (Preguntas y Respuestas) para buscar información necesaria y responder adecuadamente a una pregunta. A continuación, se detalla el funcionamiento de este proceso:

1. El agente recibe un *prompt* específico que le indica cómo responder para activar una acción. Por ejemplo: *Action : QA; Query : Information you want to know.*
2. El *prompt* incluye instrucciones sobre cuándo y cómo usar la herramienta, especificando los parámetros necesarios.
3. Cuando el LLM produce un texto como salida, se verifica si sigue el patrón *Action : X; Query : X*. Si es así, se ejecuta la herramienta con la consulta especificada.
4. La respuesta de la herramienta se utiliza como nueva entrada para el modelo LLM, continuando el proceso hasta obtener el resultado final.

Veamos un ejemplo práctico de este mecanismo:

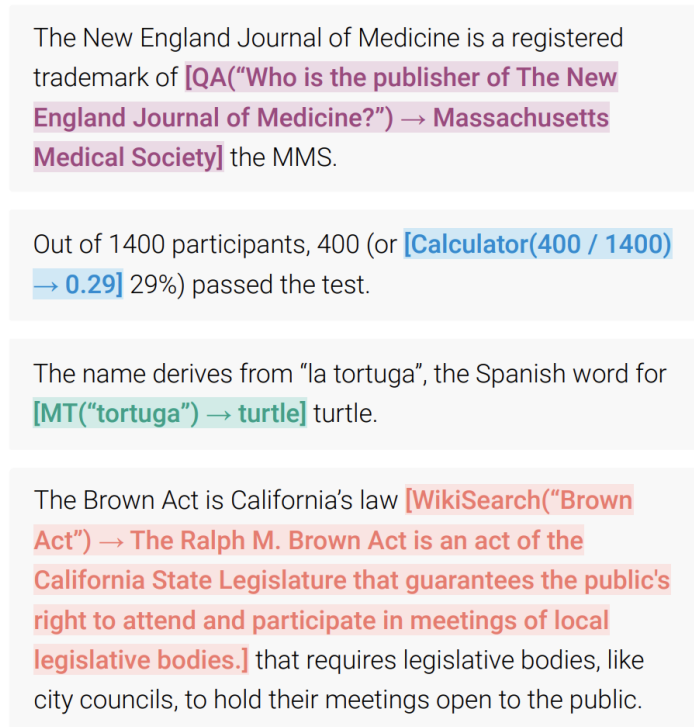


Figura 2.8: Ejemplo de un agente utilizando herramientas para responder a una entrada

Usuario: ¿Qué temperatura hace en Nueva York?
 LLM: Action: QA; Query: Weather in New York
 Agente: Acción detectada, ejecutando Tool QA("Weather in New York")
 Output Tool: The weather is 23°C.
 LLM: La temperatura actual en Nueva York es de 23 grados Celsius.

Esta estructura permite que el modelo LLM realice tareas que de otro modo estarían fuera de su alcance directo, utilizando herramientas especializadas para interactuar con el mundo externo y proporcionar respuestas informativas y relevantes.

2.4. Retrieval-Augmented Generation (RAG)

Una de las Tools más importantes que se utilizará en los próximos capítulos es Q&A. Esta herramienta se basa en el concepto de Retrieval-Augmented Generation (RAG), una técnica avanzada que combina una base de datos de conocimientos externa con un modelo de lenguaje grande (LLM) para mejorar la precisión y relevancia de las respuestas generadas.

2.4.1. Funcionamiento de RAG

El proceso de RAG se desarrolla de la siguiente manera:

1. **Consulta del Usuario:** Cuando se realiza una consulta al modelo de LLM, el sistema primero convierte esta consulta en una representación vectorial usando un modelo de embeddings.

2. **Recuperación de Información:** A continuación, el sistema busca información relevante en una base de datos vectorial. Esta base de datos contiene documentos que han sido previamente transformados en vectores a través de embeddings. Los documentos se dividen en fragmentos (*chunks*) que se vectorizan individualmente.
3. **Similitud de Vectores:** La similitud entre el vector de la consulta y los vectores de los documentos en la base de datos se calcula utilizando productos escalares u otras métricas de similitud. Los documentos más relevantes se seleccionan en base a esta similitud.
4. **Generación de Respuesta:** Los documentos recuperados se añaden como contexto al prompt original del modelo de LLM. Con esta información adicional, el modelo puede generar una respuesta más precisa y relevante, incluso sobre temas no presentes en su entrenamiento inicial.

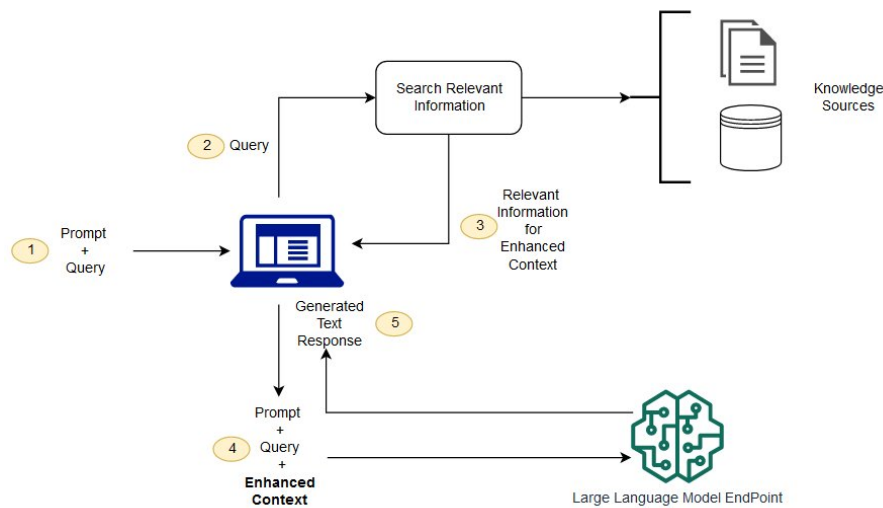


Figura 2.9: Flow de RAG

2.4.2. Ejemplo Práctico

En el caso de desarrollar un Chatbot cuya tarea es resolver preguntas de la atención un restaurante, los *knowledge sources* relevantes podrían ser el menú, información de la página web o ofertas especiales del restaurante.

Ahora en este Chatbot funcional cuando entre una consulta, como 'De postre, ¿tienen tarta de queso?', se va vectorizar la consulta obteniendo el siguiente embedding:

```
[31282, 51, 1994, 24315, 259, 26017, 409, 934, 29930, 409, 1772, 265, 30]
```

Este vector será comparado con todos los de la base de datos y se sacarán los documentos más relacionados con este vector, en este caso el menú. La información considerada como relevante se añadirá en el prompt del LLM, dando lugar a una respuesta a la pregunta con información precisa de la empresa. Esto no hubiese sido posible sin reentrenar el modelo específicamente con información del restaurante, con sus respectivos costes asociados y en caso de cambiar la información el coste sería enorme.

2.4.3. Bases de Datos Vectoriales

Las bases de datos vectoriales, como Pinecone o aquellas manejadas por Amazon Bedrock, son esenciales en RAG. Estas bases permiten almacenar y gestionar grandes volúmenes de datos en forma de vectores, facilitando la rápida recuperación de información relevante para las consultas del usuario.

2.5. AutoGPT: Automatización de Agentes y Herramientas

AutoGPT es una metodología avanzada diseñada para optimizar la interacción entre los Modelos de Lenguaje de Gran Escala (LLM) y herramientas externas, automatizando el proceso de *prompt engineering*, que es crucial para la operación efectiva de los agentes. Esta técnica facilita que los modelos sean más autónomos y eficientes al integrar y utilizar herramientas externas.

La eficacia de AutoGPT se basa en su capacidad para aprender de interacciones pasadas, optimizando secuencias de *prompts* y respuestas para maximizar la efectividad de las herramientas empleadas. Además, AutoGPT puede adaptar su estrategia basándose en el contexto y los resultados obtenidos, mejorando de manera continua su rendimiento.

2.5.1. Funcionamiento de AutoGPT

El proceso operativo de AutoGPT se puede esquematizar en los siguientes pasos:

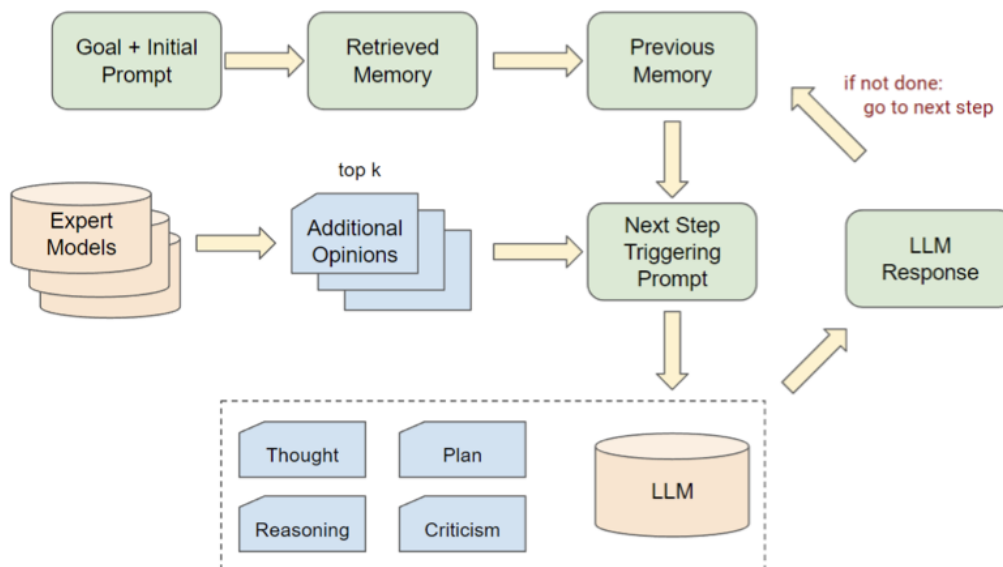


Figura 2.10: Proceso de Auto-GPT

1. **Inicialización:** El modelo comienza con una base de conocimientos sobre cómo interactuar con ciertas herramientas y qué tipo de acciones puede requerir según

el contexto del usuario, pidiéndole que defina el objetivo a cumplir (ej. Buscar la temperatura en Boston y registrar la respuesta en un archivo file.txt).

2. El modelo considera los pasos necesarios para completar la acción. Siguiendo el ejemplo anterior, los pasos serían: 1) Buscar en Internet la temperatura de Boston. 2) Escribir la temperatura en un archivo llamado file.txt.
3. Por cada paso, AutoGPT ejecuta una acción. El modelo LLM evalúa si se ha completado la acción; si no, ajusta la guía para completarla. Si la respuesta es afirmativa, pasa al siguiente paso.
4. El proceso termina cuando no quedan más acciones por realizar.

En pseudocódigo, el algoritmo de Auto-GPT se describiría de la siguiente manera:

Algorithm 1: Algoritmo de Opinión Adicional Auto-GPT

Require:

expert_opinion: opinión adicional muestreada de un modelo experto.

prompt_template(x): una plantilla con las top k opiniones más recomendadas.

regular_prompt: prompt regular.

Add(x): Añade x al contexto de Auto-GPT.

```
1: Initialize Auto-GPT
2: for each Auto-GPT step do
3:   Add(Initial Goal and Instruction Prompt)
4:   if sampled expert_opinion from expert models exists then
5:     Add(prompt_template(expert_opinion)) for i < k
6:   else
7:     Add(regular_prompt)
8:   end if
9:   Auto-GPT runs with the prompt added.
10: end for
11: return result
```

Este pseudocódigo ilustra cómo Auto-GPT gestiona la integración de opiniones expertas para guiar las respuestas del LLM, facilitando una interacción más eficiente y autónoma.

2.6. Estado del Arte

El paper de Keivalya Pandya [19], titulado *Automatic Customer Service using Langchain*, discute cómo se puede automatizar el servicio al cliente de una empresa utilizando Langchain y Retrieval-Augmented Generation (RAG). Este enfoque permite responder automáticamente a las preguntas de los clientes, mejorando la eficiencia y reduciendo la necesidad de intervención humana. Sin embargo, esta solución está limitada a la utilización exclusiva de RAG, sin incluir *Tools* que puedan ejecutar acciones adicionales ni ofrecer una solución verdaderamente genérica.

Además, existen muchos ejemplos de personas que han creado chatbots específicos para resolver problemas concretos o para proyectos personales. Estos chatbots están diseñados

para cumplir tareas específicas, pero si es necesario cambiar el modelo o crear una nueva *Tool*, se requiere modificar el código directamente, lo que limita su flexibilidad y facilidad de adaptación a nuevas necesidades.

Empresas como Bouygues Telecom han implementado inteligencia artificial generativa para resumir automáticamente las llamadas y extraer temas, reduciendo las operaciones pre y post-llamada en un 30% y ahorrando más de 5 millones de dólares anuales. Esta tecnología permite analizar y extraer información accionable de las conversaciones entre clientes y agentes, mejorando significativamente la eficiencia operativa. IBM Consulting desarrolló modelos base que ayudan en la generación automática de resúmenes de llamadas y la extracción de temas relevantes, lo que permite actualizar rápidamente los sistemas de gestión de relaciones con clientes (CRM) con información valiosa.

Otro ejemplo destacado es el asistente virtual de Lloyds Banking Group, que mediante el uso de modelos de lenguaje grande (LLMs), mejoró la precisión de las respuestas en un 85% y redujo el esfuerzo manual en un 80%. Esta implementación permitió mejorar la experiencia del asistente virtual al reducir las búsquedas fallidas y personalizar las interacciones según las necesidades específicas de los clientes. El uso de LLMs permitió clasificar de manera más precisa las conversaciones mal categorizadas, lo que resultó en una significativa mejora en la eficiencia y la satisfacción del cliente.[21][22]

Capítulo 3

Metodología

A lo largo de este capítulo vamos a explicar la metodología que vamos a seguir para nuestro Chatbot.

3.1. Cuestiones técnicas

Todo el código de este proyecto se ha escrito en **Python** 3.10. y el IDE utilizado ha sido **PyCharm** Professional Edition 2023.3.5. Se ha usado *poetry* para la gestión de las librerías en la dockerización. Tanto para el caso del Chatbot como el Evaluador se ha creado un proyecto **Django** permitiendo de esta forma comunicarnos con el servidor a través de Postman.

3.2. Evaluador

El objetivo del evaluador son dos funciones: Evaluar Tools creadas manualmente generando conversaciones y evaluar si se han ejecutado las Tools correctas con conversaciones pasadas.

Es necesario definir correctamente las métricas que se van a medir a la hora de evaluar el modelo, permitiendo saber como mejoras las Tools correspondientes.

3.2.1. Evaluación Objetiva

Durante la simulación de una conversación u obtención de una conversación pasada, es crucial evaluar la precisión de las Tools. Para ello, extraemos el *payload* deseado por el usuario, que representa la información específica que el usuario espera recibir de la herramienta. Por ejemplo, si la Tool es de preguntas y respuestas (QA) y se pregunta: “¿Quién ha sido el último premio nobel de física?”, extraeremos este *payload* y lo compararemos con el *payload* recibido por la herramienta.

La precisión se calcula de la siguiente manera:

$$\text{Precisión} = \begin{cases} 1 & \text{si el payload del cliente es igual al payload de la herramienta} \\ 0 & \text{en caso contrario} \end{cases} \quad (3.1)$$

Además de la precisión, evaluaremos la latencia. Estos indicadores nos permiten comparar la eficiencia de diferentes modelos en términos de respuesta y manejo de las interacciones.

Criterio de Evaluación	Descripción
Precisión	Coincidencia entre el payload del cliente y de la herramienta
Latencia	Tiempo de respuesta del chatbot

Cuadro 3.1: Criterios de evaluación para la simulación de conversaciones

Estos criterios proporcionan una base sólida para analizar y comparar el desempeño de diferentes modelos de chatbot en un entorno controlado.

Capítulo 4

Chatbot

A lo largo de este capítulo vamos a explicar cómo hacer la arquitectura de un Chatbot genérico para luego aplicarlo a los diferentes casos de uso. Posteriormente, haciendo uso de nuestra arquitectura crearemos un chatbot capaz de simular la atención al cliente en un restaurante, pudiendo buscar información del restaurante en su base de datos y simular hacer una reserva.

4.1. Creación Chatbot sin acciones

En el caso de un Chatbot sin acciones lo podemos hacer muy fácil con `prompt_engineering` que es capaz de indicar a nuestro modelo LLM cual va a ser su función o cómo debe de actuar.

Por ejemplo, si queremos que actúe como la atención al cliente de un hotel podemos añadirle el siguiente prompt:

```
Eres el recepcionista principal del icónico Hotel Ritz de Madrid, un establecimiento de lujo ubicado en el corazón de la ciudad, frente al Museo del Prado. Con más de 110 años de historia, el Ritz es sinónimo de elegancia, sofisticación y servicio impecable. Tu trabajo es encarnar la esencia del hotel, ofreciendo a cada huésped una experiencia verdaderamente excepcional.
```

Si este texto lo agregamos al `system_messages` ya podemos tener conversaciones con el modelo como la siguiente:

```
User: Hola, buenas tardes. Mi nombre es Harry Walker
```

```
AI: Buenas tardes, Sr. Walker. Será un placer asistirle. ¿En qué puedo serle de ayuda hoy?
```

```
User: ¿A qué hora es el desayuno?
```

```
AI: Nuestro desayuno se sirve en el elegante restaurante del hotel de 7:00 a 11:00 de la mañana. Si prefiere disfrutar de su desayuno en la comodidad de su habitación, también podemos organizar el servicio de habitaciones. ¿Hay algo más en lo que pueda asistirle?
```

```
User: No, gracias. Eso es todo que tenga un buen día.
```

```
AI: El placer ha sido mío, Sr. Walker. Le deseo un excelente día. No dude en contactarnos si necesita cualquier otra cosa.
```

Como podemos comprobar, el LLM es capaz de simular el recepcionista del Hotel Ritz de Madrid. Sin embargo, este no es capaz de realizar acciones cómo hacer el *check-in*, *check-out*, *hacer una reserva*, o muchas otras.

4.2. Creación Chatbot con acciones

Como hemos podido observar, un modelo de LLM no es capaz de ejecutar acciones por si solo, solamente tiene como output texto. Con este propósito se han diseñado los agentes[8], los cuales, a través de `prompt_engeniering`, son capaces de formatear el output, permitiendo interpretar la acción necesaria, las llamadas *Tools*.

Para que nuestro modelo identifique cuando utilizar esta *Tool* será necesario definir qué hace esa *Tool*, los parámetros necesarios y por último programar la acción a ejecutar al llamar a la función (i.e: buscar en internet la query y devolver el resultado). El resultado es un chatbot capaz de ejecutar acciones.

Debido a la complejidad de diseñar los agentes y el código necesario para saber cuando se ha llamado a una *Tool*, usaremos LangChain. Una librería de código abierto que nos permite utilizar agentes como *React (Reason + Act)*, modelos LLM y *Tools* a través de clases, permitiéndonos posteriormente a crear un chatbot genérico de forma sencilla.

Como se ha mencionado anteriormente, se va a crear un chatbot capaz de simular la atención al cliente en un restaurante. En primer lugar, se definirá el modelo de LLM a usar, en LangChain lo hacemos de la siguiente forma:

```
1 llm = ChatOpenAI(model="gpt-3.5-turbo-0125")
```

Como se puede observar en el código, para este chatbot se ha decidido usar el modelo: *gpt-3.5-turbo-0125*.

Una vez definido el LLM se redactará el *prompt*, definiendo el modo en el que se comportará el chatbot.

```
1 sys_msg = "Eres un empleado del restaurante Lateral. Se agradable y amable."
```

Definamos los mensajes a introducir al modelo de lenguaje (LLM). Incluiremos el *prompt*, los mensajes anteriores que se guardarán en una variable denominada `chat_history`, el *input* (el texto de entrada proporcionado por el cliente) y el `agent_scratchpad`, una variable donde se almacenará la salida del agente. Esta última se utiliza cuando se llama a la *tool* y se obtiene un output. Quedando el *prompt* final de la siguiente forma:

```
1 prompt = ChatPromptTemplate.from_messages(  
2     [  
3         ("system", sys_msg),  
4         MessagesPlaceholder(variable_name="chat_history"),  
5         ("human", "{input}"),  
6         MessagesPlaceholder(variable_name="agent_scratchpad")  
7     ]  
8 )
```

Por último, se debe crear la *tool* para realizar una reserva es crear la *Tool* de hacer una reserva. Será necesario tenemos que definir cuando el modelo tiene que usar esta *Tool*, qué parámetros de entrada se tienen que pasar, y qué acción ha de ejecutar:

```
1 class make_reserva(BaseModel):  
2     description= "Utilizar esta Tool cuando el cliente quiera hacer una reserva, es \  
3         muy importante que tengas todos los datos antes de llamar a la Tool."
```



```

3
4     client_name: str = Field(description="Nombre del cliente que quiere hacer la \
reserva")
5     num_people: int = Field(description="Nmero de comensales")
6     date: str = Field(description="Fecha de la reserva")
7     hour: str = Field(description="Hora a la que se va a hacer la reserva")
8
9     def _run(self, *args, **kwargs):
10         # Validar que todos los parmetros estn presentes y no vacos
11         if not self.client_name or not self.num_people or not self.date or not \
self.hour:
12             return "Error: Todos los parmetros son obligatorios."
13
14         # Construir el payload
15         payload = {
16             "client_name": self.client_name,
17             "num_people": self.num_people,
18             "date": self.date,
19             "hour": self.hour
20         }
21
22         # Realizar la peticin POST al webhook
23         try:
24             response = \
requests.post("https://webhook.site/e5d5ba9e-7853-404a-9f3f-483945782d6c", \
json=payload)
25             if response.status_code == 200:
26                 return "La reserva ha sido realizada correctamente"
27             else:
28                 return f"Error: La reserva no pudo ser realizada. Cdigo de estado: \
{response.status_code}"
29             except requests.exceptions.RequestException as e:
30                 return f"Error: Ocurri un problema al realizar la peticin. {str(e)}"

```

Esta sería la *Tool make_reserva* donde se define cuando se debe usar, qué parámetros de entrada necesita y qué acción ha de ejecutar. En este caso se ha definido que cuando se ejecute la acción haga una petición a un webhook y si la respuesta es 200 devuelva que la reserva se ha realizado correctamente, con el objetivo de simular un restaurante con mesas y horas infinitas en un primer momento.

```

1     #Lista de Tools
2     tools = [make_reserva]

```

Quedaría pendiente unir todo lo anterior en un agente, para lo que usaremos el agente especializado para los modelos OpenAI de Langchain *create_openai_tools_agent*:

```

1     agent = create_openai_tools_agent(llm, tools, prompt)

```

Quedando la estructura algo similar a esto:

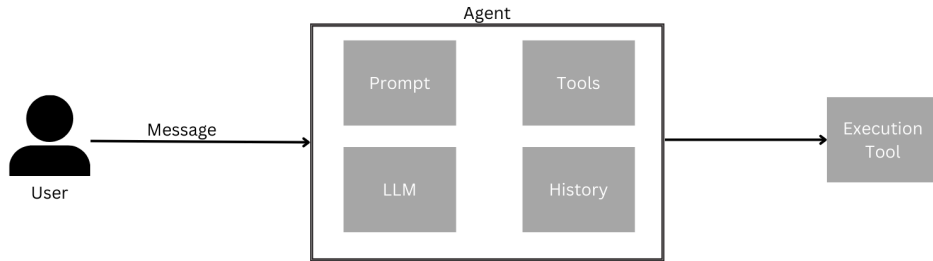


Figura 4.1: Ejemplo Agente con Tools con Langchain

Y el chatbot con las tools de esta forma:

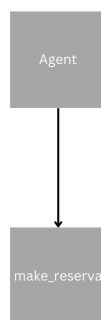


Figura 4.2: Estructura árbol chatbot restaurante

Creado el agente con la *tool* probaremos nuestro modelo haciendo una reserva:

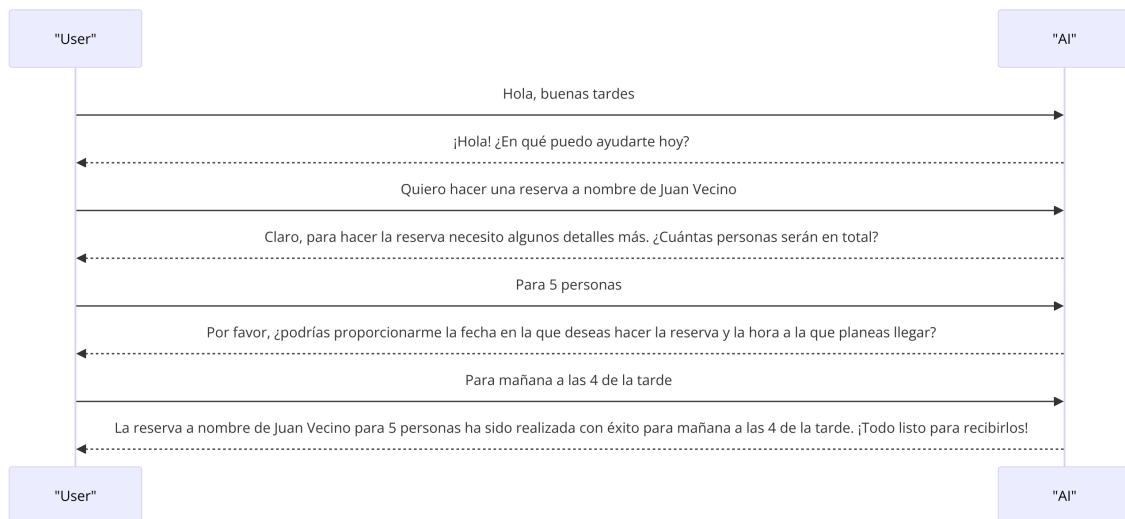


Figura 4.3: Conversación probando el chatbot de un restaurante

Y siendo el payload recibido por *Tool Execution* el siguiente:

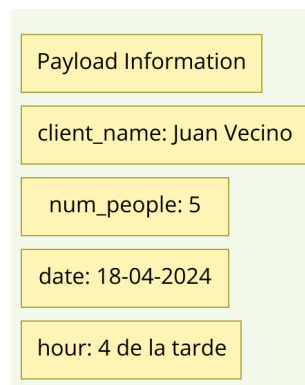


Figura 4.4: Payload conversación

Podemos observar que se ha creado un chatbot capaz de ejecutar una acción que es hacer una reserva.

Una vez logrado implementar un chatbot utilizando una *Tool* específica, el siguiente paso será generalizar este proceso para cualquier tipo de *Tool* y chatbot.

4.3. Chatbot general

Para la creación de un chatbot genérico será necesario hacer es un código capaz de crear toda la estructura detallada anteriormente de forma automática. Para ello lo que se ha hecho es parametrizar y modificar el código para que se cree automáticamente.

4.3.1. Parametrización

En el caso de la definición de los parámetros como *LLM*, *agent_type*, *Tools*, *descripción*, *parámetros de las Tools*, *etcétera* se ha definido una estructura en una base de datos Postgres SQL. Esta permite identificar a cada chatbot con ID único y este va a tener un

modelo, *agent_type*, lista de *Tools*, etcétera asociado a él. La estructura que se ha seguido son 3 tablas: *Chatbot*, *Tools*, *Params*.

En la base de datos *Chatbot*, se definen todos los parámetros necesarios para crear un chatbot (LLM y *Agent_Type*). En la base de datos *Tool*, se almacena la descripción de la herramienta y en la tabla de *Params* la descripción de cada uno de los parámetros. También se debe definir una relación entre las tablas en función del ID.

A continuación se muestra la arquitectura de cada una de las tablas:

Tabla 1: Chatbot

Parámetro	Descripción
ID	ID único del Chatbot. Nos va a servir para identificar que chatbot queremos usar.
LLM	Especificación del Modelo de Lenguaje de Gran Escala a utilizar (p.ej.: gpt-3.5-turbo).
Agent_type	Tipo de agente que se va a implementar (p.ej.: React).
Tools	Lista de los <i>Tools_ID</i> que se utilizarán.
Descripción	Descripción de cómo queremos que actúe nuestro chatbot.

Cuadro 4.1: Descripción de los parámetros de la tabla Chatbot

Tabla 2: Tool

Parámetro	Descripción
ID	ID único de la Tool.
Nombre	Nombre de la Tool (i.e: <i>make_reserva</i>).
Descripción	Breve explicación de cuándo debe utilizar el LLM esta <i>Tool</i> .
params_ID	Listado de los ID de los parámetros de entrada para llamar a la función.
Execution	Definición de la operación que realizará la <i>Tool</i> , que podría ser la ejecución de un archivo Python, realizar una petición POST a un URL, efectuar un cálculo matemático, entre otros.

Cuadro 4.2: Descripción de los parámetros de la tabla Tool

Tabla 3: Params

Parámetro	Descripción
ID	ID único de la Tool.
Nombre	Nombre del parámetro (i.e: <i>client_name</i>).
Descripción	Descripción de lo que significa este parámetro.
type	Tipo de parámetro. Str, int, ...

Cuadro 4.3: Descripción de los parámetros de la tabla Params

Con estas tablas relacionadas de la siguiente forma:

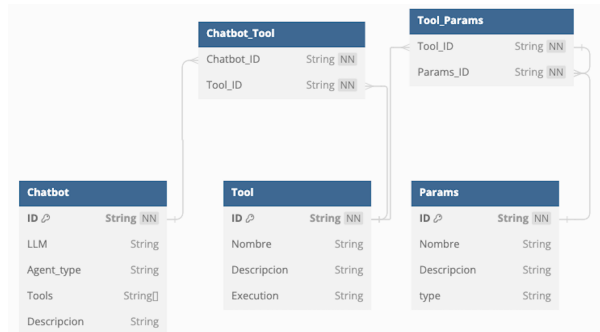


Figura 4.5: Diagrama Base de Datos

Se pueden crear clases de *Chatbot's* e ir añadiendo *Tool's* según sean necesarias, permitiéndonos crear Chatbots especializados en los que ir agregando Tools en función de las necesidades.

4.3.2. Modificación del código

Creado la arquitectura de la base de datos, se modificará nuestro proyecto para que realice este proceso de manera automática.

En un primer lugar, se creará un proyecto Django que actúe como un servidor, con una base de datos Postgres SQL con todas las tablas definidas anteriormente. Para poder popular estas tablas, no se puede hacer a mano, por lo que vamos a crear todas las *apis* pertinentes para que cualquier persona con ellas pueda ir definiendo su chatbot, sus Tools, etcétera.

Por ejemplo, una api que se llame */api/create_chatbot*, en donde se pasen como parámetros la descripción, lista de Tools, etcétera y se creará en nuestra base de datos un chatbot.

Al ser un servidor y poder tener múltiples conversaciones a la vez, no va a ser posible tener una variable de *conversation_history* con el historial de las interacciones pasadas, esta variable también va a tener que ser una base de datos definida de la siguiente forma:

Tabla 4: Historial de la conversación

Parámetro	Descripción
ID	ID único de la conversación
request_text	Lista de todas las interacciones pasadas

Cuadro 4.4: Descripción tabla Historial de la conversación

Una vez ya hemos creado las tablas ahora vamos a tener que modificar el código que hemos hecho anteriormente. El proceso que vamos a seguir es el siguiente: en vez de tener un único archivo en donde este definido un chatbot para un solo caso de uso, tener un archivo capaz de crear la arquitectura de un chatbot en función de los parámetros pasados.

Para ello se ha creado dentro del servidor el archivo llamado *Ockam.py*, ver Anexo A, el cual es una clase en donde con un atributo llamado *request*, este va a tener definidas todas las variables necesarias para crear el chatbot. Por ejemplo, ahora se define el LLM de la siguiente forma:

```

1 self.llm = ChatOpenAI(
2     openai_api_key=OPENAI_API_KEY,

```

```

3     temperature=self.temperature,
4     openai_api_base=os.getenv('OPENAI_API_BASE'),
5     model = self.model,
6 )

```

El system_message:

```

1 sys_msg = request.service.service_description

```

En el caso de los Tools se pasa también toda la información y se ejecuta en este caso el archivo tools.py , el cual consigue definir la clase que hemos hecho anteriormente de la siguiente forma:

```

1 class_code = f"""class {self.class_name}(BaseTool):
2     name = "{self.function_name}"
3     description = "{self.description}"
4     # args_schema: Type[BaseModel] = tool_input
5     request: models.Model = None
6
7     def _run(self, *args, **kwargs):
8         return _run(self, *args, **kwargs)
9
10    def _arun(self, **kwargs):
11        return _arun(self, **kwargs)
12
13    def set_request(self, request):
14        self.request = request
15    """
16
17    # create the class
18    locals_dict = {}
19    globals_dict = {
20        "BaseTool": BaseTool,
21        "_run": _run,
22        "_arun": _arun,
23        "BaseModel": BaseModel,
24        'Type': Type,
25        'tool_input': self.output_class,
26        'models': models,
27        'return_direct': self.return_direct
28    }
29    exec(class_code, globals_dict, locals_dict)
30
31    s = locals_dict[self.class_name]()
32
33    return s

```

El código es capaz de crear una clase y devolverla con la descripción, parámetros, función especificada.

Con todos los parámetros creados, LLM, tools, agent_type y tools podemos ya invocar a nuestro chatbot. Aún tenemos que definir como le vamos a pasar esta información a Ockam.py

Para ello ahora vamos a crear una api en nuestro servidor en donde especifiquemos: 1) El input, es decir, lo que queremos decirle al modelo 2) El ID del chatbot con el que estamos hablando, 3) ID del conversation_id. Quedando el body de la petición POST de la siguiente forma:

```

1 {

```

```

2  "service_id": "1541c6a1-dd60-4176-a47d-c06d82375bc5",
3  "conversation_id": "111234156745564323545641",
4  "text": "Hola buenas tardes me llamo Gonzalo quiero hacer una reserva"
5  }

```

De esta manera, podemos añadir una *api* a nuestro proyecto de Django, en donde le pasemos como payload el ID de nuestro chatbot y con toda esta información ir sacando los valores de `agent_type`, descripción del chatbot, ID de las *tools*, etcétera. Para luego pasárselo a `Ockam.py`. Esta *api* se puede encontrar en el anexo C.

Al realizar un POST con el ID específico de un chatbot, se crearán automáticamente el Agente y todas las *Tools* necesarias. Así obtenemos un chatbot genérico capaz de adaptarse a cualquier estructura y *Tool*, incluyendo la capacidad de crear *Tools* especiales para distintos casos de uso, como la búsqueda de información en una base de datos vectorizada, conexión con servicios de Google para *Tools* con acceso a internet, y muchas otras funcionalidades que se pueden agregar a demanda del cliente.

4.3.3. Ejemplo

En este caso vamos a hacer el mismo ejemplo anterior, pero ahora como el coste de hacer n *Tools* es muy bajo vamos a agregar más *Tools* a nuestro restaurante. Para ello vamos a crear las siguientes *Tools*: *cancelar_reserva*, *cambiar_reserva* y *QA*.

Lo único que vamos a tener que definir es en la base de datos que descripción, qué parámetros y qué función va a ejecutar.

El caso de *cancelar_reserva* y *cambiar_reserva* es muy similar al de *make_reserva*. Por esta razón, no vamos a detenernos en poner otra vez el código. Sin embargo, para el caso de *QA* vamos a hacer uso de un código opensource para crear un Retrieval-Augmented Generation (RAG).

El objetivo de esta *tool* es que pueda buscar cualquier información del restaurante que pregunte el cliente. Se ha tenido que crear una base de datos con toda la información del restaurante, para que posteriormente nuestro chatbot pueda hacer preguntas a esta base de datos y obtener información de vuelta.

Se ha usado la base de datos vectorizada Pinecone, la cual se ha dividido la información del restaurante en chunks de información. Posteriormente, cuando la *Tool* se ejecute se hará una petición a esta base de datos vectorizada y con la solución, el modelo podrá responder a la pregunta del cliente.

El código de la *Tool* de QA sería el siguiente:

```

1  class Q&A(BaseTool):
2  name= "Search"
3  description = """Utiliza esta funcion cuanto el cliente tenga una pregunta de \
    cualquier ndole. Tu objetivo es ayudar al cliente a resolver de la mejor forma \
    posible."""
4  return_direct = False
5  pinecone_namespace: str
6  sticky_text: str = ''
7  args_schema: Type[BaseModel] = PineconeToolInput
8
9  def _run(self, **kwargs) -> str:
10     try:
11         input_data = PineconeToolInput(**kwargs)
12
13         pinecone.init(api_key=PINECONE_API_KEY,
14                       environment=PINECONE_ENV)

```

```

15     embeddings = OpenAIEmbeddings()
16
17     docsearch = Pinecone.from_existing_index(index_name,
18                                             embeddings,
19                                             namespace=self.pinecone_namespace)
20
21     docs = docsearch.similarity_search(input_data.query, k=2)
22
23     for doc in docs:
24         response += "Source: " + doc.metadata['source'] + '\n' + \
25 doc.page_content + '\n\n'
26
27         response += f'{self.sticky_text}\n'
28
29     return response
30
31 except Exception as e:
32     print(e)
33     return "Intenta de nuevo."

```

Al final la arquitectura del modelo quedaría de la siguiente forma:

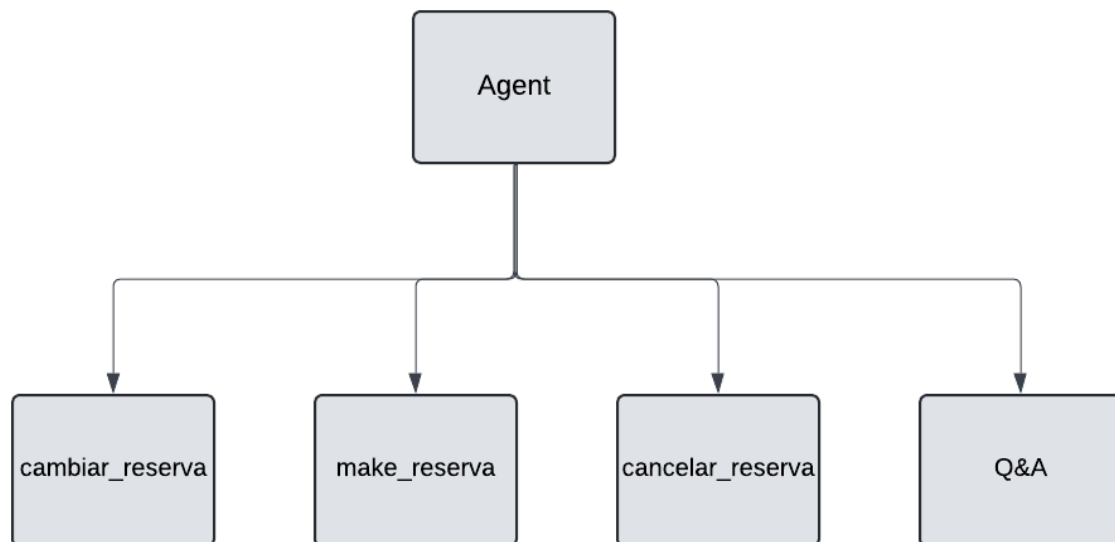


Figura 4.6: Estructura árbol chatbot restaurante v2.0

Haciendo ejemplos como el siguiente:

```

1 User: Hola
2 AI: Hola, buenas tardes bienvenido al restaurante Lateral
3 User: Me gustara hacer una reserva para maana a las 5 de la tarde
4 AI: Claro! Para cuntas personas sera la reserva?
5 User: Pues depende tenis tarta de queso?
6 AI: Tool: Q&A {'query': 'tarta de queso'}
7 Output Q&A: ### Platos para Celacos
8
9 - Heineken (Sin gluten): 3,50

```



```
10 - Ensalada Mixta: 7,00
11 - Gazpacho Andaluz: 6,00
12 - Sopa de Verduras: 6,50
13 - Entrecot a la Parrilla: 18,00
14 - Salmn a la Plancha: 16,00
15 - Risotto de Setas: 13,00
16 - Flan de Huevo: 4,50
17 - Crema Catalana: 4,50
18 - Macedonia de Frutas: 4,00
19 - Pia Natural: 4,50
20 AI: S, tenemos tarta de queso en nuestro men. Te gustara hacer la \
    reserva para maana a las 5 de la tarde?
21 User: Pues en tal caso sera para 3 personas y a nombre de Juan por favor.
22 AI: Tool: make_reserva {'client_name': {'value': 'Juan'}, 'num_people': \
    {'value': 3}, 'Date': {'value': 'maana'}, 'hour': {'value': '5 de la \
    tarde'}}
23 Output Tool: Se ha realizado la reserva
24 AI: La reserva para maana a las 5 de la tarde para 3 personas a nombre \
    de Juan ha sido realizada con xito. Esperamos verte maana en nuestro \
    restaurante!
25 User: Perdn, me acaba de enviar un mensaje mi madre y al final no voy a \
    poder ir. Me la puedes cancelar, por favor?
26 AI Tool: cancelar reserva {'client_name': {'value': 'Juan'}, 'Date': \
    {'value': 'maana'}, 'hour': {'value': '5 de la tarde'}}
27 Output Tool: Se ha cancelado la reserva
28 AI: La reserva a nombre de Juan ha sido cancelada con xito. Si en el \
    futuro deseas hacer otra reserva, no dudes en contactarnos. \
    Esperamos verte pronto!
29 User: Perfecto, muchas gracias por tu ayuda
```

Los resultados han seguido demostrando un desempeño comparable al de un ser humano, pero con un coste temporal mínimo y sin la necesidad de saber programación. A continuación se muestran los resultados de las pruebas a través del frontend que verá el cliente:

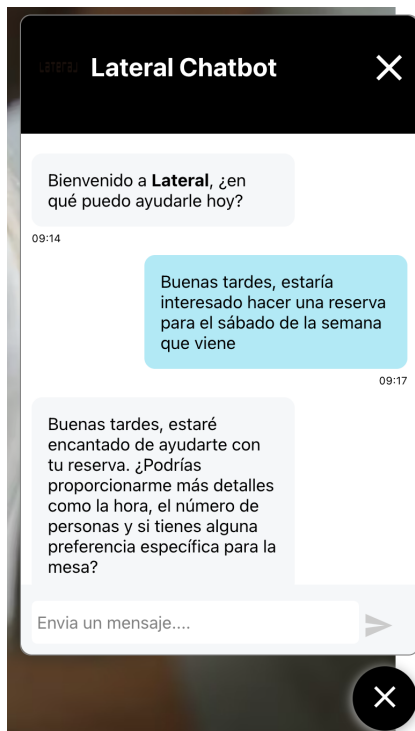


Figura 4.7: Captura de Pantalla 1

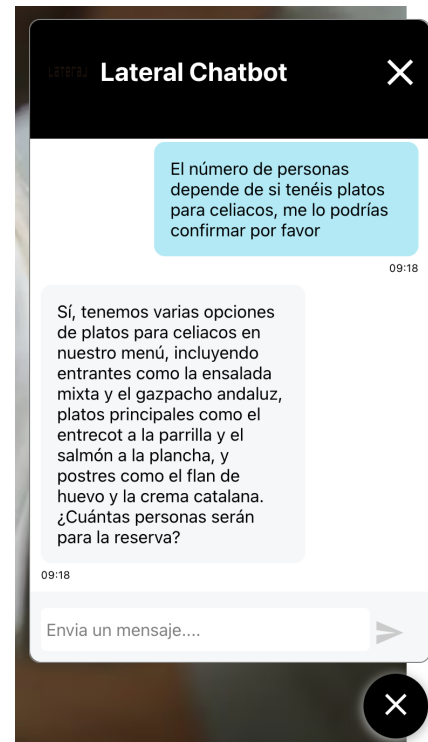


Figura 4.8: Captura de Pantalla 2

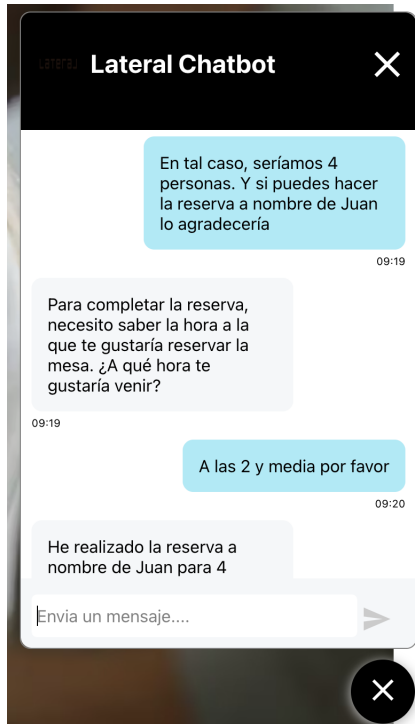


Figura 4.9: Captura de Pantalla 3

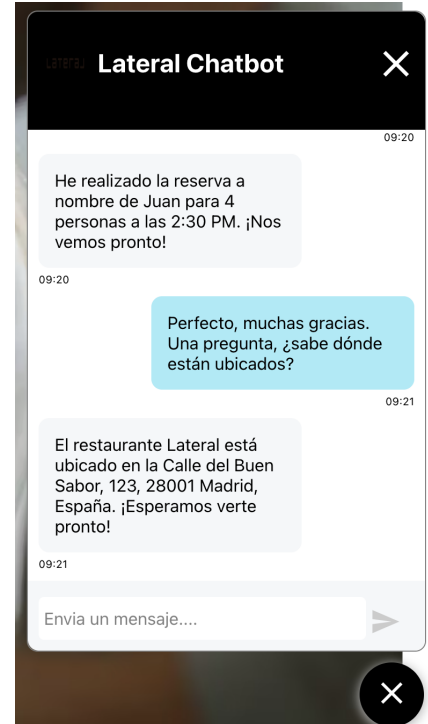


Figura 4.10: Captura de Pantalla 4

Figura 4.11: Prueba atención al cliente de un restaurante

Capítulo 5

Evaluador

5.1. Evaluador del Chatbot

Para garantizar que nuestro chatbot funcione correctamente, es necesario contar con un evaluador. Utilizaremos AutoGPT para simular conversaciones con nuestro chatbot y verificar si las *Tools* funcionan correctamente.

El flujo del proceso es el siguiente:

1. Seleccionar el *ID* del chatbot y el *ID* de la *Tool* que se va a evaluar.
2. Extraer automáticamente la descripción del chatbot, la herramienta y sus parámetros.
3. Utilizar un LLM para generar una acción ficticia que use esta herramienta, basándose en la descripción obtenida (por ejemplo: "Haz una reserva a nombre de").
4. Pasar esta acción como input a AutoGPT, que tendrá una *Tool* para comunicarse con nuestro chatbot.
5. Evaluar los parámetros obtenidos por nuestro chatbot y la acción deseada, determinando si la acción tomada es correcta (1) o incorrecta (0).

A continuación, se presenta un diagrama de flujo que ilustra estos pasos:

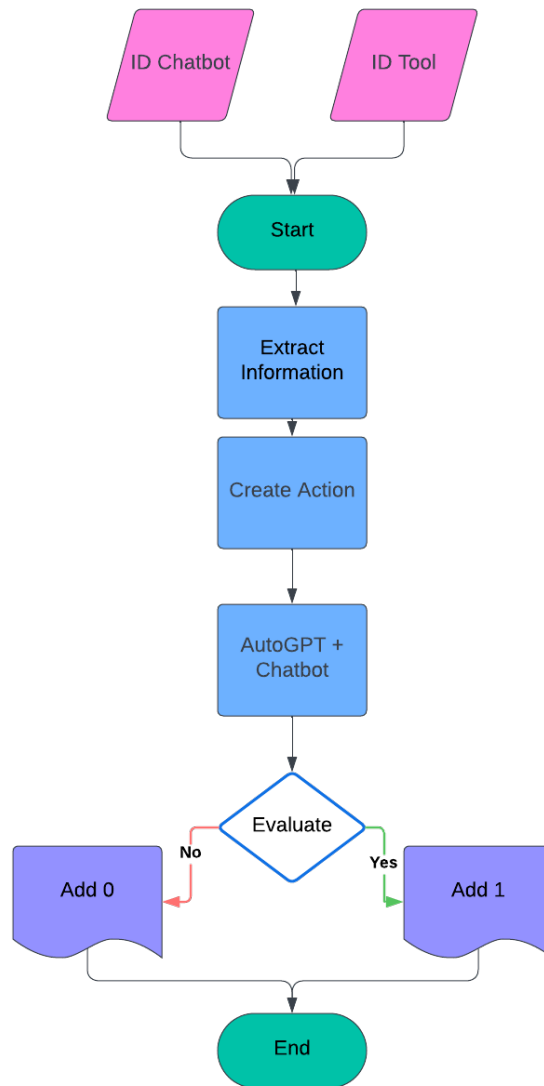


Figura 5.1: Diagrama de flujo del Evaluador

Se detallan las acciones realizadas en cada uno de los procesos marcados en azul en el diagrama.

5.1.1. Extract Information

Se ha creado una API dentro de nuestro proyecto de Django en el endpoint `api/-get_info`, donde se pasan los siguientes parámetros:

```

json:
{
"chatid": "string",
"toolID": "string"
}
  
```

La respuesta de la API es:

```
{
```

```
"chatid": { "dictionary con toda la información del chatbot" },
"toolID": { "dictionary con toda la información de la herramienta y sus parámetros"
}
}
```

Con esta información, podemos proceder al siguiente paso.

5.1.2. Create Action

Con la información obtenida del paso anterior, la pasamos a un LLM que generará la acción que le pasaremos a AutoGPT. Para ello, usamos el siguiente prompt:

```
example = [
{'name': 'name_client', 'type': 'str', 'description': 'Name of the client'},
{'name': 'num_people', 'type': 'int', 'description': 'Number of people for
the table.'},
{'name': 'date', 'type': 'str', 'description': 'Date of the reservation'}
]
```

```
message = f""EXAMPLE:
```

```
INPUT:
```

```
description: Useful for making a hotel reservation
params: {example},
```

```
OUTPUT:
```

```
You need to make a hotel reservation for Alfonso, for tomorrow for 3 people.
```

```
#####
```

```
DESCRIPTION: Based on the following input create an output just like in the
example
```

```
INPUT:
```

```
description: {description}
params: {params}
```

```
OUTPUT:
```

```
"""
```

Este prompt generará una acción similar a la siguiente:

```
"You need to make a hotel reservation for Alfonso, for tomorrow for 3 people."
```

Esta acción será la que se pase a AutoGPT en el siguiente paso.

5.1.3. AutoGPT

En este paso, se utiliza AutoGPT, aunque con modificaciones para incluir una *Tool* específica que se comunica con nuestro chatbot. Cuando se llama a esta herramienta, el parámetro de entrada será la consulta generada en el paso anterior.

AutoGPT ejecutará la acción generada, simulando una interacción con el chatbot.

5.1.4. Evaluate

Una vez que AutoGPT ha completado la tarea, necesitamos evaluar si el modelo la realizó correctamente. Obtenemos el payload de la conversación (mediante otra API en nuestro modelo de Django) y la acción generada en el paso *Create Action*.

Con esta información y un LLM, comparamos el payload con la acción. El prompt utilizado para esta comparación es el siguiente:

```
message = f"""EXAMPLE_1:

INPUT:
Information_given: You need to make a hotel reservation for Alfonso, for tomorrow
for 5 people.
payload_extracted_from_information: {example}

OUTPUT:
information_matches_payload(True/False): False
Reason: The number of people is incorrect. It should be 3 instead of 5.

EXMAPLE_2:

INPUT:
Information_given: You need tell you have a problem with a bill, give this
information 5654335Y and David
payload_extracted_from_information: {example2}

OUTPUT:
information_matches_payload(True/False): True
Reason: The information given matches the payload extracted.

#####

DESCRIPTION: Based on the following input create an output just like in the
example. If the information given matches the payload extracted the CORRECT
should be True otherwise False.
CURRENT_DATE: {current_date}

INPUT:

Information_given: {Information}
payload_extracted_from_information: {payload}

OUTPUT:
"""
```

Si el resultado de *information_matches_payload* es True, se responde a una API de nuestro proyecto Django con el resultado y se actualiza en nuestra base de datos indicando si la acción fue correcta o incorrecta.

5.2. Ejemplo

Haciendo uso del chatbot creado en el capítulo, vamos a evaluar la *Tool* de *make_reserva*. Para ello, vamos a obtener el ID de nuestro Chatbot y Tool.

Con estos, vamos a ejecutar el evaluador mostrando los pasos intermedios y todos los outputs de cada paso definido anteriormente.

5.2.1. Extract Information

Se ha obtenido:

```
{
  "chatid": { "description": "Responde a las preguntas como si fueses un empleado
de una empresa en el sector hostelero. Eres amigable y servicial." },
  "toolID": {
    "description": "Utiliza esta tool cuando el cliente quiera hacer una reserva,
es muy importante tener todos los datos antes de utilizar esta tool.",
    "parameters": [
      {"name": "client_name", "type": "str", "required": True, "description":
"Nombre de la persona que quiere hacer la reserva", "order_position": -1, "validator_type":
None, "validator_value": None},
      {"name": "num_people", "type": "int", "required": True, "description":
"Numero de comensales para la reserva", "order_position": -1, "validator_type":
None, "validator_value": None},
      {"name": "Date", "type": "str", "required": True, "description": "Date
de la reserva", "order_position": -1, "validator_type": None, "validator_value":
None},
      {"name": "hour", "type": "str", "required": True, "description": "Hora
de la reserva", "order_position": -1, "validator_type": None, "validator_value":
None}
    ]
  }
}
```

5.2.2. Create Action

En base a la información, esta ha sido la acción que ha dado como output el LLM: *Necesitas hacer una reserva para Carlos, para 4 comensales, el 15 de marzo a las 20:00 horas.*

5.2.3. AutoGPT

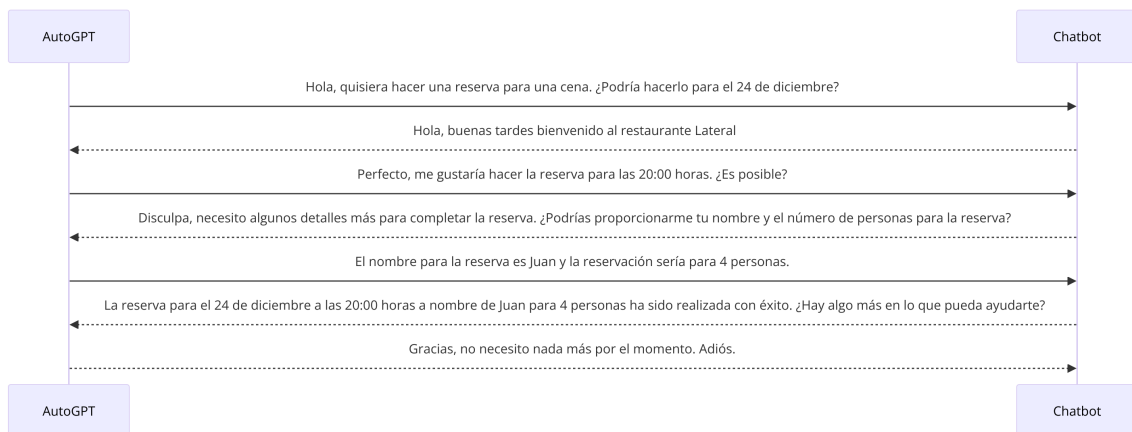


Figura 5.2: Conversación AutoGPT con Chatbot

5.2.4. Evaluate

El proceso de evaluación se ha basado en el payload extraído y en la acción que se ha pasado a AutoGPT y determina que es correcto:

```
{'Correct': True, 'Reason': 'The information given matches the payload extracted.'}
```


Capítulo 6

Análisis de Resultados

Este capítulo se centra en el análisis de los chatbots creados y la comparación entre distintos modelos de Lenguaje de Aprendizaje Profundo (Large Language Models - LLM) utilizados. Se evaluarán criterios específicos para determinar la precisión y capacidad de respuesta de cada uno de los modelos en diversas situaciones de uso.

6.1. Precisión

Como se ha comentado anteriormente, se han evaluado diferentes ChatBots con diferentes *Tools* para posteriormente comprobar si el payload recibido con el esperado es el mismo. Se ha realizado de forma automática y de forma manual al hacer las pruebas. Resultados:

Modelo LLM	Precisión
gpt-3.5-turbo-1106	87.5 %
gpt-4-turbo	95.3 %

Cuadro 6.1: Precisión de los modelos LLM

6.1.1. Valoración de los Resultados

Los resultados obtenidos muestran que los modelos más actuales, como gpt-4-turbo, presentan una mayor precisión en comparación con sus predecesores y otros modelos como gpt-3.5-turbo-1106. Esto sugiere que los avances en la tecnología de LLM están logrando reducir las alucinaciones, aunque no eliminarlas por completo.

6.2. Facilidad de Crear un Chatbot

Se ha creado la infraestructura necesaria para crear un chatbot de cualquier tipo, con un número de *Tools* infinitos y para cualquier caso de uso. En este proceso, hemos identificado ciertos aspectos a tener en cuenta durante las pruebas de nuestros modelos:

6.2.1. Alucinaciones

En muchos casos, el modelo, al no tener información suficiente, decide llamar igualmente al *Tool* y si le faltan datos, se los inventa. La solución implementada es incluir

dentro de la *Tool* otro LLM capaz de verificar si los datos que han entrado son alucinaciones o no, pasando toda la conversación y el input realizado a la *Tool*. En el caso de detectar datos inventados, la respuesta de la *Tool* será: "Por favor, pregunta por los parámetros *xxx*".

También se han observado casos con modelos más antiguos como *gpt-3.5-turbo*, donde el modelo indicaba que había realizado la reserva, pero luego no la había hecho, es decir, se inventaba la ejecución de la *Tool*. Sin embargo, esto no se ha observado con los modelos más recientes como *gpt-4*.

6.2.2. Robustez cuando hay muchas *Tools*

Aunque nuestra arquitectura permite crear infinitas *Tools* para un mismo chatbot, se ha visto a lo largo de las pruebas que la creación de muchas *Tools* dificulta al agente la selección adecuada de una *Tool*, provocando errores frecuentes.

6.2.3. Generalización

El modelo tiene una capacidad muy grande para adaptarse con la arquitectura diseñada, siendo capaz de crear cualquier tipo de chatbot y para cualquier caso de uso sin la necesidad de programar. Esto otorga a cualquier persona la capacidad de crear un chatbot para su caso de uso específico.

6.2.4. Latencia

En el caso de la latencia, se observa que los modelos más recientes pueden tardar más, pero la mejora en calidad es tan sustancial que se considera más importante que la latencia.

La forma que hemos tenido de medirlo es a través de Langchain Smith. Se ha calculado una media de la latencia por interacción con el chatbot de forma automática .

Modelo LLM	Latencia (segundos)
gpt-3.5-turbo-1106	3,75
gpt-4-turbo	6,50

Cuadro 6.2: Latencia de los modelos LLM

Y una distribución de la siguiente forma:

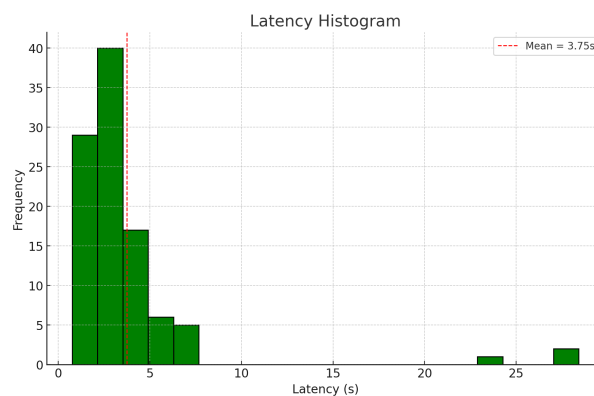


Figura 6.1: Histograma Latencia gpt-3.5-turbo-1106

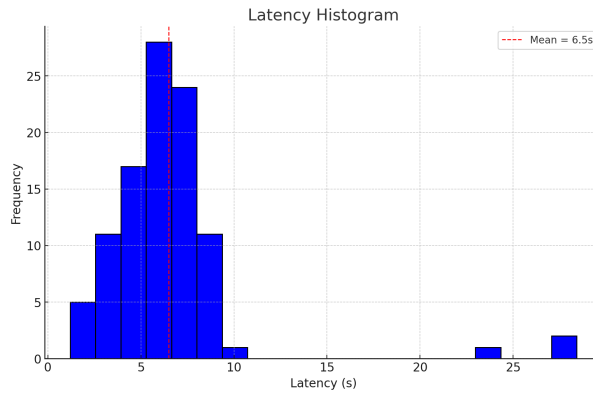


Figura 6.2: Histograma Latencia gpt-4-turbo

En el análisis de los histogramas de latencia presentados, se pueden identificar claramente outliers que corresponden a momentos en los que los servidores de OpenAI experimentan una sobrecarga (overload). Estos valores atípicos influyen en la media, pero si se excluyen, se puede observar que la mayoría de las interacciones se realizan en tiempos de latencia significativamente menores.

6.2.5. Tareas sencillas

No solamente existe un límite debido a las alucinaciones, sino también hay un límite relacionado con la dificultad de las tareas. Hacer una reserva es una tarea relativamente sencilla, pero en situaciones donde se requiere comprobar primero la disponibilidad de una mesa para un día específico o seguir una secuencia de pasos fija antes de realizar una reserva u otra acción, el modelo LLM se enfrenta a mayores desafíos. Los modelos de LLM tienen dificultades para seguir estos pasos a menos que el flujo esté codificado de manera rígida, lo que contravendría la ventaja principal de un chatbot: la capacidad de ser creado de forma genérica y flexible usando solo descripciones.

Capítulo 7

Conclusiones

A lo largo de este trabajo, se ha diseñado y desarrollado una arquitectura capaz de crear Chatbots con herramientas específicas de manera sencilla y rápida. De esta experiencia, se han extraído las siguientes conclusiones:

- **Capacidad de Suplantación:** Es posible crear un chatbot que pueda reemplazar la atención al cliente de un negocio. Sin embargo, otorgar al chatbot la capacidad total para tomar decisiones puede ser muy peligroso. Persisten problemas como las alucinaciones, donde el modelo puede afirmar que ha ejecutado acciones que en realidad no ha realizado, lo que puede generar desconfianza y serios problemas operativos en la gestión de reservas y consultas.
- **Precisión y Latencia:** Los modelos actuales, aunque tienen una mayor latencia, presentan una mayor precisión y se ajustan mejor al prompt en comparación con sus predecesores. Esto se ha evidenciado en nuestro análisis, donde modelos como GPT-4 han mostrado una reducción significativa en errores de interpretación y ejecución, mejorando la experiencia general del usuario.
- **Limitación en el Uso de Herramientas:** Existe una limitación en el número de herramientas (Tools) que se pueden usar simultáneamente. El modelo comienza a fallar más cuando el número de Tools es muy alto, debido a la complejidad adicional en la toma de decisiones y la selección de la herramienta adecuada. Esto indica que, aunque la arquitectura es flexible, se requiere una gestión eficiente del número y tipo de herramientas disponibles para mantener la fiabilidad.
- **Versatilidad y Adaptabilidad:** La arquitectura desarrollada es capaz de crear cualquier tipo de chatbot especializado para cada caso de uso con mucha facilidad. Además, permite cambiar el modelo de LLM de forma muy sencilla, ofreciendo una solución versátil y adaptable que puede ajustarse rápidamente a nuevas necesidades y avances tecnológicos.

7.1. Posibles extensiones avanzadas de este trabajo

Analizando los resultados, se identifican los siguientes pasos para solucionar algunos de los problemas detectados y mejorar aún más la arquitectura desarrollada:

- **Integración de Guardrails:** Existe un framework llamado Guardrails que permite extraer información de manera más fiable al proporcionarle un contexto específico.

Este framework puede integrarse dentro de las herramientas (Tools) para realizar comprobaciones de los parámetros de entrada, mejorando la robustez y reduciendo la incidencia de alucinaciones del modelo. Guardrails ofrece mecanismos avanzados de validación y corrección que pueden complementar significativamente nuestro sistema actual.

- **Arquitectura Multinivel:** Para superar la limitación en el número de Tools, se podría diseñar una arquitectura multinivel. Esto implicaría crear una herramienta principal que, al ejecutarse, dirija al usuario a un agente secundario con una selección de Tools más específica y especializada. Esta estructura jerárquica permitiría manejar un número mucho mayor de Tools sin comprometer la eficiencia y precisión del sistema.
- **Agentes Colaborativos:** Se podría probar el mismo funcionamiento con nuevas arquitecturas como Agentes Colaborativos. Estos agentes, especializados en tareas específicas, pueden trabajar en conjunto para manejar solicitudes complejas. Implementar Agentes Colaborativos puede añadir una capa adicional de complejidad y eficiencia a nuestra estructura de chatbot, permitiendo la integración de Tools más avanzadas y complejas. Estos agentes se especializan en diferentes aspectos del proceso, mejorando así la capacidad del chatbot para manejar una variedad más amplia de situaciones y consultas.[17]

Finalmente, como conclusión de este trabajo, cabe destacar que, aunque la arquitectura desarrollada ya muestra una gran capacidad y flexibilidad, existen oportunidades claras para mejorar su robustez y eficiencia. Implementar frameworks de validación como Guardrails, diseñar una arquitectura multinivel y explorar el uso de Agentes Colaborativos son pasos clave para avanzar en la creación de Chatbots más fiables y efectivos.

Apéndice A

Ockam.py

```
1 from langchain_openai import ChatOpenAI
2 from langchain.agents import AgentExecutor, create_openai_tools_agent
3 from .request import OkamRequest
4 import os
5 from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
6 import threading
7 import requests
8 import time
9 from .utils import convenios_search
10
11 OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
12
13
14 class Okam:
15     """
16     This is the AI system that powers VotConnect.
17     Attributes:
18         instance (Okam): The singleton instance of the Okam class.
19     """
20     instance = None
21
22     def __init__(self, temperature=None, always_context=False, model_version=None):
23
24         if temperature is None:
25             temperature = 0.0
26         if model_version is None:
27             model_version = "gpt-3.5-turbo-1106"
28
29         self.temperature = temperature
30         self.model = model_version
31         print(f"OKAM is using model: {self.model}")
32         self.llm = ChatOpenAI(
33             openai_api_key=OPENAI_API_KEY,
34             temperature=self.temperature,
35             openai_api_base=os.getenv('OPENAI_API_BASE'),
36             model=self.model,
37         )
38         self.always_context = always_context
39
40     def __call__(self, request: OkamRequest):
41         """
42         Runs the Okam AI system on the given request.
```

```

43     Args:
44         request (OkamRequest): The request to process.
45     Returns:
46         str: The response generated by the Okam AI system.
47     """
48     # STEP 1: Get the available tools
49     tools = request.get_tools()
50
51     sys_msg = request.system_message
52     prompt = ChatPromptTemplate.from_messages(
53         [
54             ("system", sys_msg),
55             MessagesPlaceholder(variable_name="chat_history"),
56             ("human", "{input}"),
57             MessagesPlaceholder(variable_name="agent_scratchpad")
58         ]
59     )
60
61     # STEP 3 : Construct the agent
62
63
64     agent_tools = tools
65     print("AGENT TOOLS")
66     print(agent_tools)
67     # STEP 3.2 : Append flow note if the agent was in a tool flow
68     text = request.text
69
70     # STEP 3.3 : Create the agent
71     agent = create_openai_tools_agent(self.llm, agent_tools, prompt)
72     self.agent_chain = AgentExecutor(agent=agent, tools=agent_tools, \
handle_parsing_errors=True)
73
74     # Run the agent
75     class TimeoutException(Exception):
76         pass
77
78     def run_agent(chain, request):
79         print("Running agent...")
80
81         current_retry = 0
82         max_retries = 3
83         timeout = 20
84         while current_retry < max_retries:
85             timeout_flag = [False] # Flag to know if timeout time is surpassed
86
87             def timeout_handler():
88                 timeout_flag[0] = True
89
90             timer = threading.Timer(timeout, timeout_handler)
91             timer.start()
92             try:
93                 if request.service.answer_briefly:
94                     query = str("Answer briefly and mantain the language of the \
query. " + text)
95                 else:
96                     query = text
97                 response = chain.invoke(
98                     {

```



```

99         "input": query,
100         "chat_history": request.get_memory()
101     }
102    )["output"]
103     print("TEXTO: " + query)
104     timer.cancel() # Cancel timer
105     if timeout_flag[0]:
106         raise TimeoutException("Waiting time of " + str(2) + " in \
OpenAI API call exceeded")
107
108     # WORKAROUND FOR CONVENIOS SEARCH
109     if response == 'EJECUTAR CONVENIOS SEARCH':
110         filters = request.get_current_payload("convenio_filters")
111         print("FILTERS: " + str(filters))
112         if filters:
113             return convenios_search(chain, request, filters)
114         else:
115             return "No se han encontrado filtros para la bsqueda de \
convenios."
116
117         return response
118     except TimeoutException or requests.RequestException as e:
119         timer.cancel() # Cancel timer
120         print(f"Error running agent: {e}")
121         time.sleep(2)
122         current_retry += 1
123         print(f"Retrying... ({current_retry}/{max_retries}")
124
125     response = run_agent(self.agent_chain,
126                          request=request)
127
128     return response
129
130 @classmethod
131 def chat(cls, request, temperature, always_context, model_version):
132     """
133     Runs the Okam AI system on the given request.
134     Args:
135     request (OkamRequest): The request to process.
136     Returns:
137     str: The response generated by the Okam AI system.
138     """
139     if cls.instance is None:
140         cls.instance = cls(temperature=temperature, \
always_context=always_context, model_version=model_version)
141     else:
142         cls.instance.always_context = always_context
143         cls.instance.temperature = temperature # also update the temperature
144         cls.instance.model = model_version # also update the temperature
145         # Update LLM if necessary
146         cls.instance.llm.temperature = temperature
147     return cls.instance(request) # now passing only the request

```

Listing A.1: Ockam.py

Apéndice B

Tools.py

```
1 import requests
2 from typing import Type, Any, Dict, Optional
3 from pydantic import Field, BaseModel, create_model
4 from langchain.tools import BaseTool
5 import json
6 from requests.exceptions import RequestException
7
8 class SkillCreator:
9     url_orchestrator = "https://api.uncall-dev.hi-iberia.es/availability/check"
10
11     def __init__(self,
12                 class_name,
13                 function_name,
14                 description,
15                 fields,
16                 callback=None,
17                 response = None):
18         self.class_name = class_name
19         self.function_name = function_name
20         self.description = description
21         self.fields = fields
22         self.callback = callback
23         self.response = response
24         self.skill_input_model = self.create_pydantic_model(self.class_name + 'Input',
25                                                         self.fields)
26
27     def get_tool(self):
28
29         def _run(self2, **kwargs):
30             try:
31                 input_data = self.skill_input_model(**kwargs)
32                 self._callback(input_data)
33                 headers = {
34                     "Content-Type": "application/json"
35                 }
36                 payload = {}
37                 payload['skill_name'] = self.function_name
38                 payload['chat_service'] = 11231123
39                 payload['payload'] = input_data.json()
40             try:
41                 response = requests.post(self.url_orchestrator, headers=headers, \
42                                         data=json.dumps(payload))
```

```

42         data = response.text
43
44         except RequestException as error:
45             data = error
46             return data
47
48         except Exception as e:
49             return str(e)
50
51     # define the _arun method
52     def _arun(self2, **kwargs):
53         raise NotImplementedError("This tool does not support async")
54
55     # define the class code
56     class_code = f"""
57 class {self.class_name}(BaseTool):
58     name = "{self.function_name}"
59     description = "{self.description}"
60     args_schema: Type[BaseModel] = tool_input
61
62     def _run(self, **kwargs):
63         return _run(self, **kwargs)
64
65     def _arun(self, **kwargs):
66         return _arun(self, **kwargs)
67 """
68
69     # create the class
70     locals_dict = {}
71     globals_dict = {
72         "BaseTool": BaseTool,
73         "_run": _run,
74         "_arun": _arun,
75         "BaseModel": BaseModel,
76         'Type': Type,
77         'tool_input': self.skill_input_model
78     }
79     exec(class_code, globals_dict, locals_dict)
80
81     s = locals_dict[self.class_name]
82     return s
83
84     def _callback(self, input_data):
85         if self.callback is not None:
86             return self.callback(input_data)
87
88     @staticmethod
89     def create_pydantic_model(name: str, fields: Dict[str, Dict[str, Any]]) -> \
Type[BaseModel]:
90
91         attributes = {}
92
93         for field_name, field_info in fields.items():
94             field_type = field_info.get("type")
95             field_description = field_info.get("description")
96             field_default = field_info.get("default")
97             field_required = field_info.get("required")
98

```

```
99         if not field_required:
100             field_type = Optional[field_type]
101
102             attributes[field_name] = (field_type, Field(field_default, \
description=field_description))
103
104         model = create_model(name, **attributes)
105
106     return model
```

Listing B.1: Tools.py

Apéndice C

Api.py

```
1 import json
2 import time
3
4 from django.http import HttpResponse, HttpResponseBadRequest, JsonResponse, \
    HttpRequest
5 from drf_yasg import openapi
6 from drf_yasg.utils import swagger_auto_schema
7 from rest_framework import status, viewsets
8 from rest_framework.decorators import action, api_view
9 from rest_framework.response import Response
10
11 import okam
12 from data.models import ChatConversation, ChatDataSource, ChatRequest, ChatService, \
    ChatSkill, Client, \
13     DataSourceDocument, Subscription
14 from data.serializers import ChatDataSourceSerializer, ChatServiceSerializer, \
    ChatSkillSerializer, ClientSerializer, \
15     DataSourceDocumentSerializer, SubscriptionSerializer
16 from .serializers import ConversationInputSerializer, QuickstartRequestSerializer, \
    ResponseSerializer, \
17     ClientInitiatorSerializer, PymeInitiatorSerializer
18
19
20 @swagger_auto_schema(method='post',
21                     request_body=ConversationInputSerializer,
22                     responses={200: ResponseSerializer},
23                     )
24 @api_view(['POST'])
25 def query(request):
26     """
27     This is the main endpoint of the API. It allows to send text to the AI BlackBox \
    and obtain and response.
28
29     The request body should be a json that contains the following properties:
30     - text: text of the question being asked
31     - conversationId: id of the conversation in char format
32     - subscriptionId: id of the subscription in UUID format
33
34     """
35
36     time_start = time.time()
37
```

```

38 # Get the json file
39 data = json.loads(request.body.decode('utf-8'))
40
41 if 'conversationId' in data:
42     # Check if conversationId is registered
43     chat_conversation = \
44 ChatConversation.objects.filter(id=data['conversationId']).first()
45
46     if not chat_conversation and 'subscriptionId' in data:
47         subscription = \
48 Subscription.objects.filter(id=data['subscriptionId']).first()
49         if subscription:
50             chat_service = subscription.chat_service.all()[0]
51             chat_conversation = ChatConversation(id=data['conversationId'], \
52 chat_service=chat_service)
53             chat_conversation.save()
54         else:
55             return HttpResponseBadRequest('SubscriptionId not found')
56     else:
57         return HttpResponseBadRequest('conversationId is required')
58
59 # Create request
60 chat_request = ChatRequest(chat_conversation=chat_conversation,
61                             request_text=data['text'])
62 chat_request.save()
63
64 # Obtain the data sources
65 data_sources = list(map(lambda x: x.to_okam(), \
66 chat_conversation.chat_service.chatdatasource_set.all()))
67
68 # Obtain skills
69 skills = list(
70     map(lambda x: x.to_okam(chat_request=chat_request), \
71 chat_conversation.chat_service.chatskill_set.all()))
72
73 # TODO: Daniel Bicand; 28-07-2023; Investigate different types of history \
74 langchain.
75 # Obtain the response
76 okam_request = okam.OkamRequest(data['text'],
77                                 \
78 chat_history=chat_conversation.get_full_conversation(),
79                                 \
80 description=chat_conversation.chat_service.description_for_llm,
81                                 data_sources=data_sources,
82                                 skills=skills,
83                                 \
84 company=chat_conversation.chat_service.client.name)
85
86 # print(chat_conversation)
87 response_text = okam.Okam.chat(okam_request)
88 # print(response_text)
89
90 # Save the information
91 chat_request.response_text = response_text
92 chat_request.seconds_taken = time.time() - time_start
93 chat_request.save()
94
95 response_dict = {'conversationId': chat_conversation.id,

```



```

87         'text': response_text,
88         'id': chat_request.id,
89         'intent': 'text',
90         'hang_up': False}
91
92     for skill_call in chat_request.skillcall_set.all():
93         if skill_call.chat_skill.passthrough == True:
94             \
95             chat_conversation.change_chat_service(skill_call.chat_skill.next_chat_service)
96
97             # Call with the new chat service
98             response_passthrough = {'conversationId': str(chat_conversation.id),
99                                     'text': 'En funcin de la conversacin que te \
100             estoy pasando escoge que funcin utilizar, si necesitas ms infromacin pregunta \
101             por ella. Haz como si este mensaje no hubiese aparecido en la conversacin con el \
102             cliente.'}
103             request_passthrough = HttpRequest()
104             request_passthrough.method = 'POST'
105             request_passthrough._body = \
106             json.dumps(response_passthrough).encode('utf-8')
107             response_passthrough = query(request_passthrough)
108             response_passthrough = \
109             json.loads(response_passthrough.content.decode('utf-8'))
110
111             # Call completed now get the response of the call, for the next request \
112             it will work as a normal call
113             # but with a different chat service
114             response_dict['text'] = response_passthrough['text']
115             response_dict['id'] = response_passthrough['id']
116             response_dict['intent'] = response_passthrough['intent']
117
118             if response_passthrough.get('payload', None) is not None:
119                 response_dict['payload'] = response_passthrough['payload']
120
121             continue
122         else:
123             response_dict['intent'] = skill_call.chat_skill.name
124             response_dict['payload'] = skill_call.content
125
126     return JsonResponse(response_dict,
127                         json_dumps_params={'ensure_ascii': False})

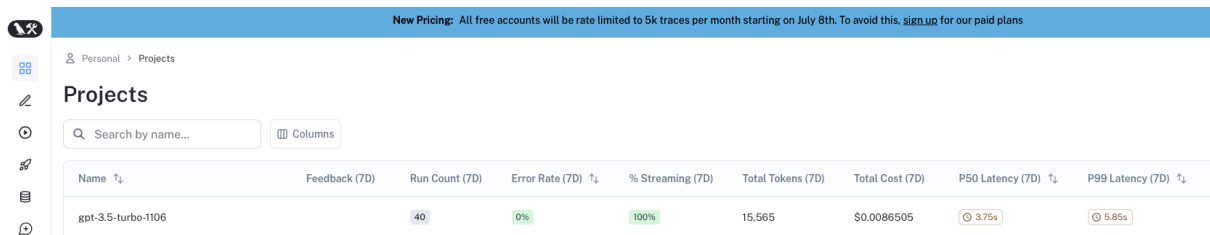
```

Listing C.1: Api.py

Apéndice D

Información Langchain Smith

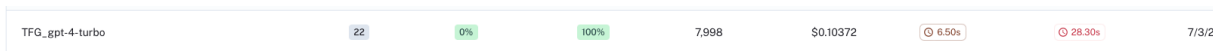
A continuación se muestra una captura de pantalla de Langchain Smith en donde se pueden ver los 2 proyectos creados y la latencia promedio en cada uno de ellos.



The screenshot shows the LangChain Smith interface. At the top, there is a blue banner with the text: "New Pricing: All free accounts will be rate limited to 5k traces per month starting on July 8th. To avoid this, sign up for our paid plans". Below the banner, the breadcrumb "Personal > Projects" is visible. The main heading is "Projects". There is a search bar "Search by name..." and a "Columns" button. The table below has the following columns: Name, Feedback (7D), Run Count (7D), Error Rate (7D), % Streaming (7D), Total Tokens (7D), Total Cost (7D), P50 Latency (7D), and P99 Latency (7D). The table contains one row for the project "gpt-3.5-turbo-1106" with the following values: Feedback (40), Run Count (40), Error Rate (0%), % Streaming (100%), Total Tokens (15,565), Total Cost (\$0.0086505), P50 Latency (3.75s), and P99 Latency (5.85s).

Name	Feedback (7D)	Run Count (7D)	Error Rate (7D)	% Streaming (7D)	Total Tokens (7D)	Total Cost (7D)	P50 Latency (7D)	P99 Latency (7D)
gpt-3.5-turbo-1106	40	40	0%	100%	15,565	\$0.0086505	3.75s	5.85s

Figura D.1: Captura de Pantalla de LangChain Smith con los proyectos de los 2 Chatbots



The screenshot shows a single row from the LangChain Smith Projects table for the project "TFG_gpt-4-turbo". The values are: Feedback (22), Run Count (22), Error Rate (0%), % Streaming (100%), Total Tokens (7,998), Total Cost (\$0.10372), P50 Latency (6.50s), P99 Latency (28.30s), and a status indicator "7/3/2".

TFG_gpt-4-turbo	22	22	0%	100%	7,998	\$0.10372	6.50s	28.30s	7/3/2
-----------------	----	----	----	------	-------	-----------	-------	--------	-------

Figura D.2: Captura de Pantalla 2 de LangChain Smith con los proyectos de los 2 Chatbots

Apéndice E

Alineación con los Objetivos de Desarrollo Sostenible (ODS)

El desarrollo y la implementación de tecnologías avanzadas en el ámbito del servicio al cliente no solo representan un avance tecnológico significativo, sino que también se alinean estratégicamente con los Objetivos de Desarrollo Sostenible (ODS), promoviendo un impacto positivo en el ámbito socioeconómico global. A continuación, se detalla la contribución específica de este proyecto a dos ODS clave:

E.1. ODS 8: Trabajo Decente y Crecimiento Económico

Este proyecto contribuye al ODS 8 mediante la optimización de procesos en las empresas, lo que permite una reducción de costes operativos y facilita la reinversión en áreas críticas como investigación y desarrollo. Al automatizar la atención al cliente con tecnologías basadas en Modelos de Lenguaje de Gran Escala (LLMs), se promueve la eficiencia y se abre el camino hacia un crecimiento económico sostenible. Además, la implementación de este tipo de tecnología no supone la eliminación de puestos de trabajo, sino la transformación de los mismos, creando nuevas oportunidades laborales en áreas como el mantenimiento de sistemas, la gestión de la experiencia del cliente y la supervisión de la inteligencia artificial, promoviendo así el trabajo decente en el sector tecnológico.

E.2. ODS 9: Industria, Innovación e Infraestructura

La integración de soluciones tecnológicas avanzadas en el servicio al cliente se alinea con el espíritu del ODS 9, fomentando la innovación y la construcción de infraestructuras resilientes. Este proyecto representa un paso adelante en la adopción de la inteligencia artificial para automatizar y mejorar los servicios de atención al cliente en diversas industrias. Al hacerlo, no solo se mejora la calidad y la accesibilidad de estos servicios, sino que también se promueve la innovación tecnológica y se contribuye al desarrollo de una infraestructura digital más robusta y sostenible, capaz de adaptarse y responder a las necesidades cambiantes de la sociedad y la economía global.

En conclusión, mediante la promoción del crecimiento económico sostenible y la innovación tecnológica, este proyecto no solo busca optimizar los servicios de atención al cliente, sino que también se compromete con la visión de un futuro más justo, sostenible e innovador, alineado con los principios y metas de los Objetivos de Desarrollo Sostenible.

Bibliografía

- [1] Grand View Research, «Large Language Model Market Size, Share & Trends Analysis Report By Application (Customer Service, Content Generation), By Deployment, By Industry Vertical, By Region, And Segment Forecasts, 2024 - 2030,» Grand View Research, Region-Specific, Market Research Report GVR-4-68040-186-2, 2024, Number of Pages: 100 Format: Electronic (PDF). dirección: <http://www.grandviewresearch.com>.
- [2] J. Novet, *Microsoft's \$13 Billion Bet on OpenAI Carries Huge Potential Along with Plenty of Uncertainty*, CNBC, Publicado el 8 de abril de 2023, actualizado el 9 de abril de 2023, abr. de 2023. dirección: <https://www.cnbc.com/2023/04/08/microsofts-complex-bet-on-openai-brings-potential-and-uncertainty.html>.
- [3] T. Eloundou, S. Manning, P. Mishkin y D. Rock, *GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models*, 2023. arXiv: 2303.10130 [econ.GN].
- [4] T. Mikolov, I. Sutskever, K. Chen, G. Corrado y J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, 2013. arXiv: 1310.4546 [cs.CL].
- [5] H. Yang, S. Yue e Y. He, *Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions*, 2023. arXiv: 2306.02224 [cs.AI].
- [6] A. Vaswani, N. Shazeer, N. Parmar et al., *Attention Is All You Need*, 2023. arXiv: 1706.03762 [cs.CL].
- [7] S. Yao, J. Zhao, D. Yu et al., *ReAct: Synergizing Reasoning and Acting in Language Models*, 2023. arXiv: 2210.03629 [cs.CL].
- [8] T. Schick, J. Dwivedi-Yu, R. Dessì et al., *Toolformer: Language Models Can Teach Themselves to Use Tools*, 2023. arXiv: 2302.04761 [cs.CL].
- [9] D. Dugas, *The GPT-3 Architecture, on a Napkin*, Accessed: 2024-04-30. dirección: https://dugas.ch/artificial_curiosity/GPT_architecture.html.
- [10] A. Radford, K. Narasimhan, T. Salimans e I. Sutskever, «Improving Language Understanding by Generative Pre-Training,» 2018.
- [11] A. Radford y K. Narasimhan, «Improving Language Understanding by Generative Pre-Training,» 2018. dirección: <https://api.semanticscholar.org/CorpusID:49313245>.
- [12] D. Foster, *Generative Deep Learning*, 2.^a ed. O'Reilly Media, Inc., 2023, ISBN: 9781098134181.

- [13] I. Amazon Web Services, *What is Retrieval-Augmented Generation?* <https://aws.amazon.com/es/what-is/retrieval-augmented-generation/>, 2024.
- [14] Pinecone, *Vector Similarity Search*, <https://www.pinecone.io/learn/vector-similarity/>, 2024.
- [15] Pinecone, *What is a Vector Database?* <https://www.pinecone.io/learn/vector-database/>, 2024.
- [16] LangChain, *LangChain Documentation*, <https://python.langchain.com/v0.2/docs/introduction/>, 2024.
- [17] W. Liu, C. Wang, Y. Wang et al., *Autonomous Agents for Collaborative Task under Information Asymmetry*, 2024. arXiv: 2406.14928 [cs.AI]. dirección: <https://arxiv.org/abs/2406.14928>.
- [18] P. Lewis, E. Perez, A. Piktus et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, 2021. arXiv: 2005.11401 [cs.CL]. dirección: <https://arxiv.org/abs/2005.11401>.
- [19] K. Pandya, *Automating Customer Service using LangChain*, 2023. dirección: <https://arxiv.org/pdf/2310.05421>.
- [20] J. Briggs, *Create Custom Tools for Chatbots in LangChain — LangChain 8*, 2023. dirección: <https://www.youtube.com/watch?v=q-HNpWrWsDE>.
- [21] IBM, «Transforming customer service: How generative AI is changing the game,» 2023. dirección: <https://www.ibm.com/blog/transforming-customer-service-how-generative-ai-is-changing-the-game/>.
- [22] ICMI, «The Promise of Generative AI: New, Flexible Customer Experiences,» 2023. dirección: <https://www.icmi.com/resources/2023/the-promise-of-generative-ai-new-flexible-customer-experiences>.