



COMILLAS
UNIVERSIDAD PONTIFICIA

ICAI

ICADE

CIHS

AGENTE LLM PARA LA RECOLECCIÓN DE DATOS

Miguel Ángel Dávila Romero

Supervisado por: Carlos Morrás Ruiz-Falcó

UNIVERSIDAD PONTIFICIA DE COMILLAS

ICAI

Máster en Big Data: Tecnología y Analítica Avanzada

Enero de 2025

Miguel Ángel Dávila Romero, declara bajo su responsabilidad, que el Proyecto con título **Agente LLM para la recolección de datos** presentado en la ETS de Ingeniería (ICAI) de la Universidad Pontificia Comillas en el curso académico 2023/24 es de su autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Fecha: ..07.. / ..01.. / ..2025..

Autoriza la entrega:

EL DIRECTOR DEL PROYECTO

Carlos Morrás Ruiz-Falcó

Fdo.: Fecha: / /

V. B. DEL COORDINADOR DE PROYECTOS

Emilio Martín Gallardo

Fdo.: Fecha: / /

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESIS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Miguel Ángel Dávila Romero **DECLARA** ser el titular de los derechos de propiedad intelectual de la obra: Agente LLM para la recolección de datos, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, los derechos de digitalización, de archivo, de reproducción, de distribución y de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

- (a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar “marcas de agua” o cualquier otro sistema de seguridad o de protección.
- (b) Reproducir la en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
- (c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.

- (d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
- (e) Asignar por defecto a estos trabajos una licencia Creative Commons.
- (f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente*).

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

- (a) Que la Universidad identifique claramente su nombre como autor de la misma
- (b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
- (c) Solicitar la retirada de la obra del repositorio por causa justificada.
- (d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

El autor se compromete a:

- (a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
- (b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
- (c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

- (d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusiva del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a ...7...de ...enero.....de .2025.

ACEPTA

Fdo.:

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:



COMILLAS
UNIVERSIDAD PONTIFICIA

ICAI

ICADE

CIHS

AGENTE LLM PARA LA RECOLECCIÓN DE DATOS

Miguel Ángel Dávila Romero

Supervisado por: Carlos Morrás Ruiz-Falcó

UNIVERSIDAD PONTIFICIA DE COMILLAS

ICAI

Máster en Big Data: Tecnología y Analítica Avanzada

Enero de 2025

Resumen

Este trabajo de fin de máster presenta el desarrollo de un sistema modular para la automatización de interacciones cliente-empresa mediante grandes modelos de lenguaje (LLM). El sistema, implementado en la empresa **Votconnect**, se centra en la creación de *skills* configurables que permiten a los agentes autónomos recopilar, validar y operar sobre parámetros de manera robusta y adaptable a diversos casos de uso.

Se revisa el estado del arte en Procesamiento del Lenguaje Natural (PLN) y LLM, destacando los avances recientes y los desafíos asociados, como la falta de robustez y las alucinaciones de los modelos. La arquitectura propuesta utiliza componentes modulares y técnicas de validación avanzadas para asegurar la precisión y fiabilidad de las interacciones, superando las limitaciones de los agentes autónomos actuales.

El sistema es evaluado en contextos prácticos como la gestión de reservas en restaurantes y la consulta de información empresarial, demostrando su efectividad y adaptabilidad. A pesar de los avances, se identifican áreas de mejora en términos de latencia y costes, y se propone la exploración de nuevas herramientas como LangGraph para futuras iteraciones.

Este trabajo contribuye al desarrollo de soluciones más robustas y eficientes en la automatización de interacciones mediante LLM, con el propósito de conseguir que los agentes autónomos sean una herramienta viable en un entorno de producción.

Abstract

This master’s thesis presents the development of a modular system for automating client-company interactions using large language models (LLMs). The system, implemented at **Votconnect**, focuses on creating configurable *skills* that allow autonomous agents to robustly and adaptively collect, validate, and operate on parameters for various use cases.

The state of the art in Natural Language Processing (NLP) and LLMs is reviewed, highlighting recent advances and associated challenges, such as the lack of robustness and model hallucinations. The proposed architecture employs modular components and advanced validation techniques to ensure the precision and reliability of interactions, overcoming the limitations of current autonomous agents.

The system is evaluated in practical contexts such as restaurant reservation management and business information queries, demonstrating its effectiveness and adaptability. Despite the advances, areas for improvement in terms of latency and costs are identified, and the exploration of new tools like LangGraph is proposed for future iterations.

This work contributes to the development of more robust and efficient solutions for automating interactions using LLMs, with the goal of making autonomous agents a viable tool in a production environment.

Agradecimientos

Me gustaría comenzar esta memoria agradeciendo a Carlos Morrás Ruiz-Falcó, a quien he tenido la suerte de conocer primero como profesor y, poco después, como jefe en el ámbito profesional y tutor de este trabajo. Desde noviembre del año pasado, hasta hoy, he estado en las mejores manos.

Como en todo lo que haga, también debo agradecer enormemente a mi familia: a mis padres, a mis hermanos y a mi novia. Ellos me han apoyado y se han interesado por mi trabajo, en todas esas demos que les hacía cuando volvía a casa y que nunca salían bien a la primera... No faltaban las bromas al respecto, pero tampoco el interés y el apoyo.

De igual manera, quiero agradecer a mis amigos, tanto a los que ya tenía como a los que he hecho este año en clase. Este máster no solo me ha aportado conocimiento y preparación; también he podido conocer a muchas personas que, estoy seguro, seguirán en mi vida.

En particular, agradezco a mis compañeros y amigos de Votconnect por haber hecho del trabajo, que uno supone aburrido y tedioso, una experiencia divertida que me impulsaba a levantarme de la cama cada mañana.

Por último, quiero agradecer su labor a todos los profesores que me han impartido clase a en este máster. Todos ellos me han guiado hasta aquí, ayudándome cuando lo necesitaba, y les doy las gracias por ello.

Índice general

1. Introducción	1
1.1. Objetivos	2
1.2. Estructura del trabajo	2
2. Antecedentes y estado del arte	5
2.1. Procesamiento del Lenguaje Natural	5
2.1.1. Historia del Procesamiento del Lenguaje Natural	6
2.1.2. Fases del Procesamiento del Lenguaje Natural	7
2.1.3. Tareas del Procesamiento del Lenguaje Natural	8
2.2. Large Language Models	9
2.2.1. Primeros modelos de generación de texto	9
2.2.2. Transformers	10
2.2.3. Principales LLM Actuales	12
2.3. Agentes autónomos	13
2.3.1. Auto-GPT	13
2.3.2. CrewAI	14
2.3.3. LangChain	15
3. Arquitectura propuesta	17
3.1. Componentes de una skill	17
3.1.1. Obtención de parámetros	18
3.1.2. Validación de parámetros	19
3.2. Actualización del payload	22
3.2.1. Salida al usuario	23

3.2.2. Finish	24
3.3. Creación de skills	25
4. Casos de uso	27
4.1. Gestión de reservas en un restaurante	27
4.1.1. Componentes de la skill	28
4.1.2. Ejemplos de funcionamiento	29
4.2. Consulta de información empresarial en distintos convenios	29
4.2.1. Componentes de la skill	30
4.2.2. Ejemplos de funcionamiento	31
5. Desafíos y soluciones implementadas	35
5.1. Escalabilidad	35
5.2. Robustez	36
5.3. Latencia y costes	38
6. Conclusiones y líneas de trabajo futuro	40
6.1. Conclusiones	40
6.2. Líneas de trabajo futuro	41
Referencias	45
A. Payload para creación de skill	49

Capítulo 1

Introducción

Este trabajo ha sido desarrollado como parte del periodo de prácticas que llevé a cabo desde noviembre en la empresa **Votconnect**: una *start-up* cuyo objetivo es la automatización de los canales de comunicación cliente-empresa mediante el uso de grandes modelos de lenguaje o LLM.

Cuando llegué a la empresa, el producto más desarrollado era el RAG [27]: un sistema capaz de agregar información externa a un LLM, el cual no forma parte de este trabajo. Sin embargo, había otro caso de uso contemplado: las llamadas **Skills**: procesos llevados a cabo por el LLM en interacción con el usuario, en el que se le pedía una serie de parámetros para, posteriormente, operar con ellos. El ejemplo más usado en mi estancia en la empresa, y que se recuperará más adelante, es la **gestión de una reserva** en un restaurante.

Este trabajo se centra en el diseño y desarrollo del sistema de *Skills* de *Votconnect*, del cual he formado parte durante los **nueve meses** de prácticas que he estado vinculado a la empresa.

En este primer capítulo se presenta el contenido de este trabajo de fin de máster, en el cual se propone una arquitectura software basada en modelos de lenguaje que tiene como objetivo la automatización de un proceso de recopilación, validación y operación sobre parámetros a partir de la interacción con el usuario.

Se explicarán aspectos relativos al trabajo, de modo que, una vez finalizado el capítulo, se puedan presentar los antecedentes que preceden al mismo.

1.1. Objetivos

El **objetivo principal** de este trabajo es presentar el concepto de *Skill* desarrollado en *Votconnect*: una arquitectura modular que permite a los clientes definir la interacción entre usuario y **agente autónomo** (bot) para tareas en las que se recopilan parámetros y se opera sobre ellos.

De este objetivo, cuya definición es más amplia, se derivan múltiples **objetivos específicos**:

- Estudiar el **estado del arte** en cuanto a agentes autónomos, capaces de llevar a cabo las tareas que se desean solucionar, para tomar inspiración y conocer posibles alternativas.
- Proponer una **arquitectura modular** que se adapte a distintos casos de uso. El cliente debe ser capaz de especificar los componentes de la misma para adaptarla a sus necesidades.
- Conseguir la **robustez** necesaria en el proceso para que sea factible la **puesta en producción** del producto, un aspecto que los agentes autónomos del estado del arte aún no han conseguido solventar.
- Explicar **dos casos de uso** concretos que se han trabajado durante la elaboración del trabajo: cómo se adapta la arquitectura a los mismos y cómo afectarían al cliente final.
- Por último, valorar la arquitectura propuesta: sus posibles **problemas** y las **líneas de trabajo** futuras que permitirían solventarlos.

1.2. Estructura del trabajo

La memoria de este trabajo de fin de máster, a partir de este punto, se organiza de la siguiente manera:

- En el Capítulo 2, **Antecedentes y estado del arte**, se realiza una revisión exhaustiva del campo del Procesamiento del Lenguaje Natural (PLN). Se abordan su historia, fases y tareas fundamentales. Además, se profundiza en los Modelos de Lenguaje (LLM), con un especial énfasis en los agentes autónomos.

- En el Capítulo 3, **Arquitectura propuesta**, se describe detalladamente la arquitectura de la solución planteada. Se desglosan los componentes de la misma y se explica cómo se crean y configuran las *skills* en el sistema propuesto.
- En el Capítulo 4, **Casos de uso**, se ilustran aplicaciones prácticas de la arquitectura propuesta mediante ejemplos específicos. Se presentan dos casos de uso: la gestión de reservas en un restaurante y la consulta de información empresarial en distintos convenios.
- En el Capítulo 5, **Desafíos y soluciones implementadas**, se analizan los principales retos encontrados durante el desarrollo del proyecto, como la escalabilidad y la latencia, así como los costes asociados. Se describen las soluciones implementadas para superar estos desafíos.
- En el Capítulo 6, **Conclusiones y líneas de trabajo futuro**, se resumen las principales conclusiones obtenidas a lo largo del trabajo. Se reflexiona sobre el impacto y las contribuciones del proyecto y se sugieren posibles líneas de investigación y desarrollo futuras que podrían expandir o mejorar los resultados obtenidos.

Capítulo 2

Antecedentes y estado del arte

Una vez introducido el trabajo y definidos los objetivos del mismo, el siguiente paso es llevar a cabo un repaso de aquellos **conceptos clave** que han permitido llevarlo a cabo, así como el **estado del arte** actual.

En este capítulo se comenzará introduciendo el concepto de **procesamiento de lenguaje natural** (PLN), que es el amplio campo en el que se incluye el trabajo.

Se hará especial hincapié en los **grandes modelos de lenguaje** (LLM), que tanto impacto han tenido en los últimos tiempos: se repasará la línea temporal que culminó con la arquitectura **transformer**, en la que se basan los mencionados modelos y que será descrita a grandes rasgos.

Por último, se explicará el concepto de **agentes autónomos**: una idea revolucionaria que permite construir aplicaciones complejas basadas en LLM que interactúen entre sí y con el usuario para llevar a cabo una tarea, lo cual encaja perfectamente con el objetivo principal de este trabajo.

2.1. Procesamiento del Lenguaje Natural

El **procesamiento del lenguaje natural** (en inglés, *Natural Language Processing*) o PLN [33] es una disciplina dentro del ámbito de la **Inteligencia Artificial** (IA) dedicada al estudio y desarrollo de algoritmos y modelos capaces de entender, interpretar y generar el lenguaje humano de manera que resulta natural para las personas.

Este área interdisciplinaria combina conocimientos de lingüística, ciencias de la computación y aprendizaje automático para abordar una amplia gama de problemas relacionados

con el lenguaje, desde la traducción automática y el análisis de sentimientos, de los que datan los primeros estudios, hasta la generación de texto [31], línea en la cual se lleva trabajando desde hace años, pero que ha tenido un gran crecimiento en el último lustro gracias a los **grandes modelos de lenguaje**, que se explicarán en la siguiente sección.

2.1.1. Historia del Procesamiento del Lenguaje Natural

El PLN surge en la década de 1950, con un enfoque influenciado por la lingüística estructural y la teoría de la información. Uno de los primeros hitos fue el desarrollo del **Modelo de Markov** [5] y el **Algoritmo de Viterbi** [16], que se utilizaron para el reconocimiento de patrones en el lenguaje. Al mismo tiempo, se iniciaron los primeros esfuerzos en la traducción automática, destacando el proyecto de **Georgetown-IBM** [22], que logró traducir automáticamente unas sesenta oraciones del ruso al inglés.

Durante los años 1970 y 1980, el campo del PLN comenzó a evolucionar hacia enfoques más **basados en reglas**, con el desarrollo de gramáticas generativas y el análisis sintáctico. Este periodo vio también el surgimiento de los primeros sistemas de diálogo y chatbots, como ELIZA [36], un programa que simulaba una conversación con un psicoterapeuta. Sin embargo, el rendimiento de los sistemas de la época, como cabría esperar, distaba mucho del panorama actual, y no daba pie en ningún caso a vislumbrar un futuro tan prometedor como el que hoy vivimos.

En la década de 1990, con el auge del **aprendizaje automático**, el enfoque del PLN cambió drásticamente. Se empezó a utilizar el aprendizaje supervisado [13] y no supervisado [6] para el procesamiento de grandes cantidades de datos textuales. Los **métodos estadísticos** se convirtieron en herramientas esenciales para tareas como el etiquetado de partes del discurso (*POS tagging*) [30] y el reconocimiento de entidades nombradas (*named entity recognition*) [29].

En los años 2000, la atención se centró en la clasificación de textos con enfoques probabilísticos, con tareas como la detección de spam [24]. Además, con la **expansión de la web** (que ya empezó en la década anterior), el corpus de datos disponible para el entrenamiento de los modelos aumentó en proporciones inconcebibles unas décadas atrás.

Por su parte, la década de 2010 marcó un punto de inflexión con la introducción del **aprendizaje profundo** y las redes neuronales artificiales. Modelos como Word2Vec [11]

permitieron representar palabras en **espacios vectoriales** de alta dimensión, capturando relaciones semánticas de manera más efectiva: se trata de los famosos *embeddings* [4].

El lanzamiento de los *transformers* [28] revolucionó el campo al ofrecer capacidades sin precedentes en comprensión y **generación de lenguaje**, aunque profundizaremos en ello más adelante.

2.1.2. Fases del Procesamiento del Lenguaje Natural

El PLN se compone de cinco fases principales [9], que se describen a continuación.

- **Análisis Morfológico y Léxico:** implica descubrir y describir la estructura de las palabras. Se descomponen las palabras en sus morfemas, que son las unidades más pequeñas de significado, identificando y clasificando cada palabra de acuerdo con su categoría gramatical y significado léxico.
- **Análisis Sintáctico:** esta fase, también conocida como *parsing*, se centra en analizar las palabras dentro de una oración para representar su estructura gramatical. Las palabras se transforman en una estructura que demuestra cómo están relacionadas entre sí.
- **Análisis Semántico:** se encarga de extraer el significado del contexto, intentando asignar un sentido a la estructura de las palabras que fue formada previamente.
- **Integración del Discurso:** implica la comprensión del contexto más amplio y la relación entre las oraciones para interpretar el significado coherente de un texto completo. Esta fase es esencial para mantener la coherencia y la cohesión en la interpretación del lenguaje, y es una de las más complejas.
- **Análisis Pragmático:** se refiere al proceso de abstraer o derivar el uso deliberado del lenguaje en situaciones específicas, especialmente aquellas en las que se requiere conocimiento del mundo. Se consideran aspectos como la intención del hablante, el contexto social y cultural, y cómo estos factores influyen en la interpretación del lenguaje.

Tradicionalmente, estas fases han sido llevadas a cabo por **modelos distintos**. Sin embargo, con la irrupción de los ya nombrados *transformers*, todo el proceso puede llevarse a cabo por **un único modelo** de dicha arquitectura.

2.1.3. Tareas del Procesamiento del Lenguaje Natural

Como se ha podido apreciar en el apartado de historia del PLN, son muchas las tareas que se abarcan en este campo. Algunas de ellas se listan a continuación.

- **POS Tagging:** consiste en asignar una etiqueta gramatical a cada palabra de una oración. Estas etiquetas pueden ser sustantivos, verbos, adjetivos, adverbios, etc, y nos dan información sobre la función de cada palabra dentro de la oración.
- **Named Entity Recognition (NER):** implica identificar y clasificar las entidades mencionadas en un texto, como nombres de personas, organizaciones, lugares, etc.
- **Análisis de sentimientos:** se centra en determinar la opinión o el sentimiento expresado en un texto. Es una tarea muy útil en redes sociales o encuestas, por ejemplo.
- **Traducción automática:** consiste en convertir texto de un idioma a otro de manera automatizada, manteniendo la fluidez y precisión en la medida de lo posible.
- **Generación de resúmenes:** se trata de una tarea compleja, que en un principio comenzó abordándose como una concatenación de frases importantes del texto. Sin embargo, con el tiempo se introdujeron técnicas más sofisticadas.
- **Clasificación de texto:** consiste en categorizar documentos o fragmentos de texto en diferentes clases o categorías predefinidas. El análisis de sentimientos no es más que un caso de clasificación, como también lo es la detección de spam.
- **Generación de texto:** el siguiente paso lógico, una vez los algoritmos de PLN son capaces de comprender el texto, es que sean capaces de generarlo. Esta tarea es en la que se concentra la mayoría del esfuerzo a día de hoy.

La generación de texto, de forma indirecta, **soluciona todas las demás tareas:** un algoritmo capaz de generar texto también está capacitado para resolver el resto de tareas de PLN. Por ese mismo motivo, actualmente nos encontramos la época de la irrupción de los *Large Language Models* (LLM).

2.2. Large Language Models

Los *Large Language Models* (LLM) son modelos de IA diseñados para **comprender y generar** lenguaje natural. Utilizan técnicas de aprendizaje profundo para analizar y producir texto, lo que les permite realizar una variedad de tareas lingüísticas, como traducción automática, resumen de texto, generación de contenido y más. Los LLM se entrenan con vastas cantidades de datos textuales para capturar patrones y relaciones en el lenguaje, lo que les permite generar respuestas coherentes y contextualmente relevantes a las entradas de texto.

2.2.1. Primeros modelos de generación de texto

Antes la llegada de los LLM modernos, los primeros intentos de generar texto automáticamente se basaron en arquitecturas de redes neuronales recurrentes (RNN) y sus variantes, como las Long Short-Term Memory (LSTM) y las Gated Recurrent Units (GRU).

Redes Neuronales Recurrentes

Las redes neuronales recurrentes (RNN) [17] fueron una de las primeras arquitecturas utilizadas para la generación de texto. Estas redes son adecuadas para procesar **secuencias de datos**, como el lenguaje natural, debido a su capacidad para mantener información en un “estado oculto” que se actualiza a medida que se procesa cada elemento de la secuencia. Sin embargo, las RNN enfrentan problemas significativos, como la dificultad para capturar dependencias a largo plazo debido a problemas de **desvanecimiento de gradiente** [20].

Long Short-Term Memory

Para abordar las limitaciones de las RNN, se desarrollaron las Long Short-Term Memory (LSTM) [21], una variante que introduce nuevos componentes que permiten a la red mantener y actualizar información a lo largo de secuencias más largas. Las LSTM son capaces de recordar información relevante durante **periodos extendidos de tiempo** y son menos susceptibles a los problemas de desvanecimiento de gradientes. Esto las hace más efectivas para la captura de dependencias a largo plazo.

Gated Recurrent Units

Las Gated Recurrent Units (GRU) [10] son otra variante de las RNN que **simplifica** la estructura de las LSTM manteniendo capacidades similares. Introducen un mecanismo de puertas que controla el flujo de información dentro de la unidad, permitiendo que la red aprenda dependencias a largo plazo con **menos parámetros** y menor complejidad computacional.

2.2.2. Transformers

El paper *Attention is All You Need* [38] presentó una **nueva arquitectura** que reemplazó las RNN y LSTM en muchas aplicaciones de procesamiento de lenguaje natural. Debido a su complejidad, esta arquitectura fue conocida como “Transformers”. Los Transformers se basan en mecanismos de atención, que permiten al modelo enfocarse en diferentes partes de la secuencia de entrada al procesar cada elemento, sin necesidad de procesar el texto en orden secuencial. Esto se traduce en la posibilidad de paralelizar el procesamiento y, por tanto, en un mayor rendimiento.

El Transformer se compone de dos partes principales: el codificador y el decodificador. Ambos están formados por bloques de atención y capas de *feed-forward* [7].

- **Mecanismo de Atención:** La atención permite que el modelo evalúe la importancia relativa de diferentes partes de la secuencia de entrada. El mecanismo de atención más común es la atención “multi-cabeza” (*multi-head attention*), que permite al modelo aprender diferentes aspectos de las relaciones en la secuencia de manera paralela.
- **Codificador:** El codificador o *encoder* transforma la secuencia de entrada en una representación interna a través de múltiples capas de atención y *feed-forward*. Cada capa aplica el mecanismo de atención para reponderar la importancia de las diferentes partes de la entrada y luego utiliza una red neuronal *feed-forward* para procesar esa información.
- **Decodificador:** El decodificador o *decoder* utiliza la representación interna generada por el codificador para producir la secuencia de salida, aplicando un proceso similar pero con atención adicional sobre la secuencia de salida previamente generada para asegurar la coherencia en la generación del texto.

En la Figura 2.1 se plasma la arquitectura de un Transformer, en la que se pueden distinguir los componentes ya explicados. Como se puede apreciar, destaca por su **complejidad**, y es por ello que, al no ser el tema principal de este trabajo, no se va a entrar en mayor profundidad respecto a la arquitectura.

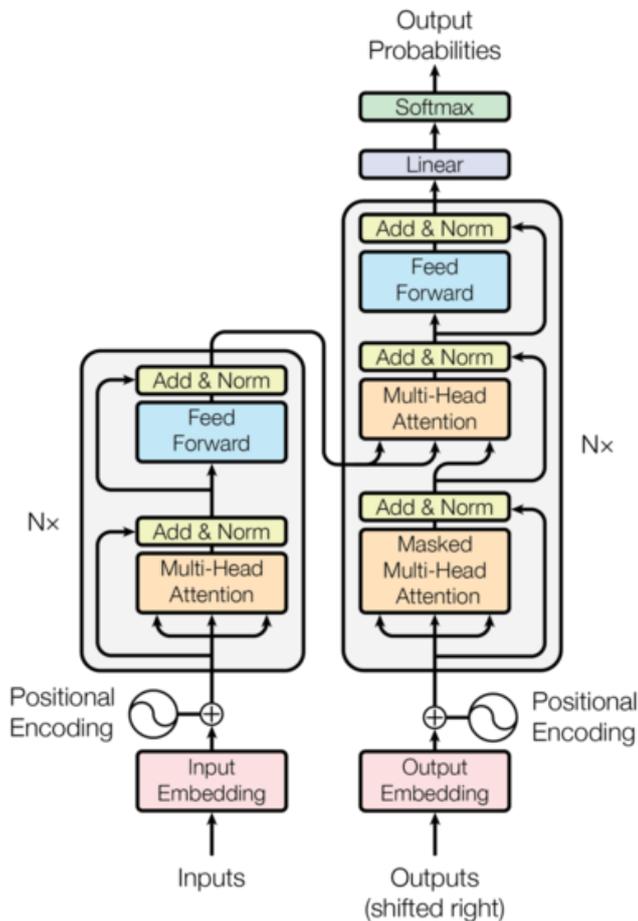


Figura 2.1: Esquema de la arquitectura Transformer. Fuente: [38]

El **entrenamiento** de un modelo transformer para dar lugar a un LLM implica un proceso intensivo de **recopilación y procesamiento** de grandes volúmenes de datos textuales extraídos de **internet**, abarcando múltiples idiomas y contextos culturales. Esta vasta diversidad de datos permite que los modelos sean **multilingües**, capaces de entender y generar texto en varios idiomas. Durante el entrenamiento, el modelo se expone a secuencias de palabras y aprende a predecir la siguiente palabra en la secuencia, ajustando sus parámetros para **minimizar el error** en estas predicciones. Este proceso iterativo y masivo de ajuste permite al modelo captar patrones lingüísticos complejos y contextuales, dotándolo de una comprensión profunda y versátil del lenguaje humano.

2.2.3. Principales LLM Actuales

Desde el surgimiento de la arquitectura *transformer*, han sido muchas las empresas que han entrenado sus propios modelos que, posteriormente, han liberado ya sea como producto de pago o como modelo de código abierto (*open source*). Algunos de estos modelos son:

- **GPT:** propiedad de la empresa *OpenAI* [1], se trata del LLM más famoso. Su versión 4o es la más moderna en el momento en el que se escribe esta memoria. Se trata de un modelo de pago, aunque existe una versión gratuita que hoy en día es una de las herramientas de IA más empleadas: *ChatGPT*.
- **Gemini:** desarrollado por *Google* [37], este modelo destaca por su enorme ventana de contexto (un concepto que representa la “memoria” del modelo). Al igual que GPT, es de pago, y de hecho es su principal competidor.
- **Claude:** creado por *Anthropic* [3], es un LLM diseñado con un enfoque en la alineación ética y la seguridad. Claude busca minimizar los sesgos y proporcionar respuestas que sean útiles y seguras para los usuarios, abordando algunas de las críticas y desafíos éticos asociados con los LLM. Se trata de un modelo de pago, y en el momento de la escritura de este trabajo, su versión más avanzada es la 3.
- **Llama:** diseñado por *Meta* [2], se trata de la alternativa *open source* más famosa, pero no por ello es menos potente que el resto. Existen múltiples modelos, cada uno con un número diferente de parámetros, sobre los que se puede hacer *fine-tuning* para mejor rendimiento en tareas específicas.

A pesar de llamarse grandes modelos de **lenguaje**, algunos de los modelos actuales, como GPT-4o y Gemini, han evolucionado para ser multimodales [25]. Esto significa que, además de procesar y generar texto, estos modelos pueden comprender y generar **múltiples tipos de datos**, como imágenes, audio y video. La capacidad multimodal permite que estos modelos integren información de diversas fuentes y contextos, lo que enriquece su comprensión y respuesta a entradas complejas. Por ejemplo, un modelo multimodal puede analizar una imagen y generar una descripción textual precisa, o interpretar una pregunta hablada y responder con texto, lo que amplía significativamente su aplicabilidad en campos como la asistencia virtual, la educación y el entretenimiento.

2.3. Agentes autónomos

Los **agentes autónomos** son sistemas de IA diseñados para llevar a cabo tareas de manera **independiente**, sin necesidad de supervisión humana constante. Estos agentes aprovechan las capacidades avanzadas de los LLM y el método conocido como *Chain of Thought* [39] para ejecutar una variedad de tareas de manera eficiente y efectiva. La combinación de estos dos elementos permite a los agentes autónomos no solo comprender y generar texto, sino también razonar y **tomar decisiones** en base a la información disponible.

El concepto de *Chain of Thought* se refiere a la capacidad de los modelos para descomponer problemas complejos en pasos más manejables y resolverlos de manera **secuencial**. Este enfoque es crucial para los agentes autónomos, ya que les permite estructurar sus procesos de pensamiento de manera lógica y sistemática, facilitando la resolución de tareas que requieren múltiples etapas de razonamiento. Por lo general, el funcionamiento consiste en forzar al agente a identificar su **situación** y generar una **acción** frente a ella.

A pesar de ser un concepto novedoso, en el momento de la escritura de este trabajo existen ya múltiples proyectos de agentes autónomos, que se recogen a continuación.

2.3.1. Auto-GPT

Auto-GPT [19] fue uno de los primeros agentes autónomos en utilizar LLM para llevar a cabo tareas complejas. Desarrollado como una extensión de GPT-3, AutoGPT se diseñó para ejecutar comandos en cadena, utilizando el método de *Chain of Thought* para abordar problemas y generar soluciones de manera autónoma. Algunas de las tareas que puede llevar a cabo son la búsqueda de información, generación de contenido o programación, todo mientras interactúa con múltiples fuentes de datos y ajusta sus acciones en función de los resultados obtenidos.

Auto-GPT opera mediante la descomposición de tareas en **subtareas más pequeñas**, resolviendo cada una de manera secuencial. Utiliza un LLM para comprender el contexto de la tarea, generar posibles soluciones y evaluar la mejor ruta de acción. A medida que avanza en la resolución de la tarea, **ajusta su enfoque** en función de los resultados intermedios, asegurando una alta precisión y eficiencia.

Algunos proyectos desarrollados con Auto-GPT permiten desarrollar aplicaciones, apro-



Figura 2.2: Logo de Auto-GPT

vechando la capacidad de programar que tienen los LLM (y más concretamente GPT), hacer análisis de mercado de ciertos productos, gestionar correos y calendarios... En definitiva, tareas sencillas pero que consumen tiempo y **pueden ser automatizadas**.

2.3.2. CrewAI

CrewAI [18] es un ejemplo innovador de agentes autónomos que colaboran entre sí para resolver tareas complejas. En lugar de operar de manera individual, coordina **múltiples agentes**, cada uno especializado en una parte específica de la tarea. Esta colaboración permite abordar problemas más grandes y complejos que un solo agente podría no ser capaz de resolver por sí mismo.



Figura 2.3: Logo de CrewAI

CrewAI se utiliza en áreas donde la colaboración y la integración de diversas habilidades son esenciales. Por ejemplo, en la **gestión de proyectos de software**, diferentes agentes pueden encargarse de la planificación, codificación, pruebas y documentación, trabajando juntos para completar el proyecto de manera eficiente.

Los **sistemas multiagente** [26] son la novedad en el estado del arte, y han demostrado un gran potencial en tareas autónomas, con **poca interacción** del usuario. Como se hablará al final de este trabajo, se trata de una línea de investigación futura. Sin embargo, en el

momento en el que se desarrolló este trabajo aún no se había introducido este concepto, por lo que se optó por la alternativa más potente en ese momento: LangChain.

2.3.3. LangChain

LangChain [32] no es como tal un agente, sino una plataforma diseñada para facilitar la creación de flujos de trabajo complejos utilizando LLM. Permite a los desarrolladores **encadenar** múltiples modelos y herramientas para ejecutar procesos detallados. Es muy común en proyectos que tratan con LLM, pues ayuda a controlarlos y conducirlos dentro de una aplicación, como puede ser un sistema de RAG [27].

Sin embargo, dentro de LangChain sí que se encuentra el concepto de agente, implementado en una clase llamada *AgentExecutor*. Esta no es más que un LLM de nuestra elección que cuenta también con una serie de *tools*: funciones que toman una serie de parámetros y operan con ellos.



Figura 2.4: Logo de LangChain

El agente de LangChain, ante cada nuevo mensaje del usuario, lleva a cabo un proceso de *Chain of Thought* que le permite decidir qué *tool* usar (si es que tiene que hacerlo). Este proceso se conoce como *intent detection* o detección de la intención. Se trata de una tarea vital para los agentes autónomos. Para tomar la decisión, se basa en la **descripción** de la *tool*, que debe ser del tipo “Usa esta *tool* cuando el cliente quiera....”

Una vez el agente sabe qué *tool* invocar, su misión pasa a ser **recopilar los parámetros** necesarios. Para ello, puede preguntar al usuario o... **inventárselos**. Y es que ese es el problema del *AgentExecutor* de LangChain y, en general, de todos los agentes autónomos: **no son lo suficientemente robustos** para un entorno de producción. Frente a este inconveniente, se desarrolló la arquitectura que protagoniza este trabajo, la cual se explica a continuación.

Capítulo 3

Arquitectura propuesta

Una vez hemos descrito el problema al que nos enfrentamos, así como el estado del arte en cuanto a los agentes autónomos, es momento de presentar la **arquitectura propuesta** para hacer frente a dicho problema.

A esta arquitectura nos referiremos a partir de ahora como **Skills**: una abstracción que hace referencia al proceso en el que el agente LLM interactúa con el cliente para recoger una serie de parámetros y hacer algo con ellos.

El objetivo fue, en todo momento, dar lugar a un **flujo muy controlado**, en el que en cada mensaje el agente esté guiado por el código para seguir una serie de pasos: pedir parámetros, informar de errores en los mismos, etc.

El motivo de forzar un flujo fue conseguir **la mayor robustez posible**, dado que, como ya se ha explicado anteriormente, los LLM tienden a alucinar. Este escenario no es nada deseable en un entorno de producción, por lo que el control sobre el flujo es indispensable.

De igual manera, se busca un **diseño modular**, en el que cada etapa esté diferenciada y modelada por una entidad distinta, de modo que un usuario pueda crear una *Skill* especificando una serie de componentes.

3.1. Componentes de una skill

En la Figura 3.1 se ilustra la arquitectura de *Skill*. Como se puede apreciar, se trata de una arquitectura compleja, a la que se llegó tras múltiples iteraciones y que contempla una gran variedad de escenarios distintos, como se comprueba más adelante, manteniendo el foco siempre en la modularización del proceso.

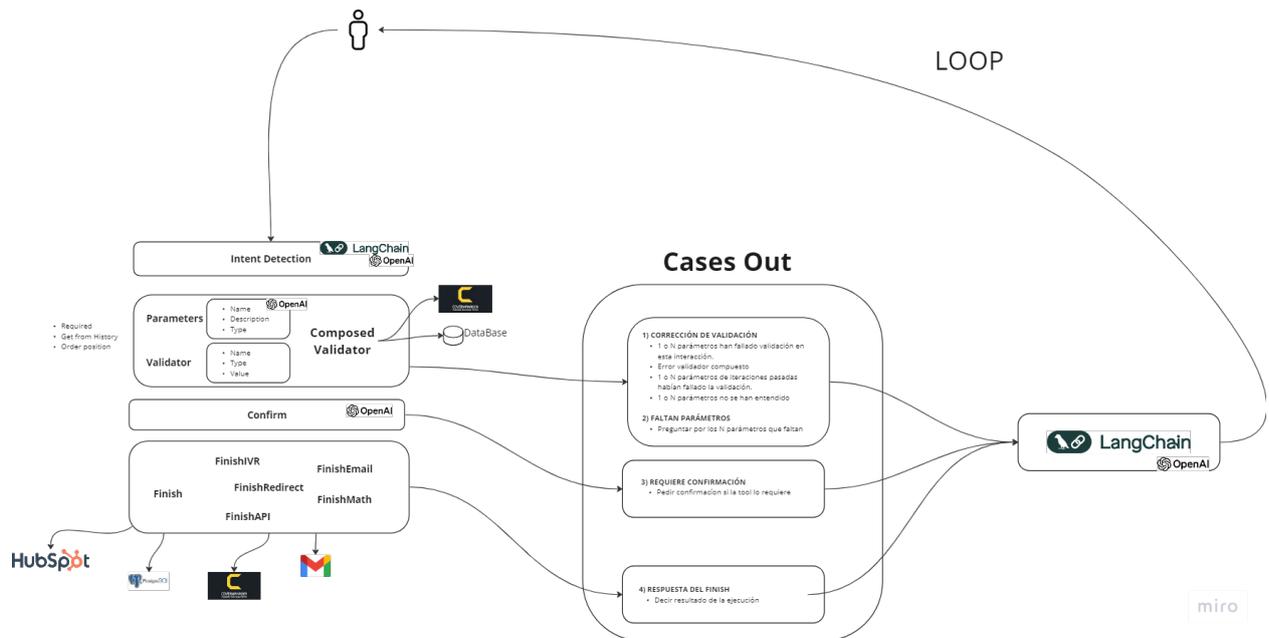


Figura 3.1: Arquitectura de Skills propuesta

A grandes rasgos, podemos distinguir cinco etapas en la ejecución de una *skill*:

1. Recopilación de parámetros.
2. Validación de parámetros.
3. Actualización de valores.
4. Salida al usuario (puede requerir confirmación de parámetros).
5. Ejecución de la acción final o *finish*.

A continuación se explica cada una de estas etapas.

3.1.1. Obtención de parámetros

La primera fase, como es de esperar, es la **obtención de los parámetros** que componen la *skill*.

Para ello, se usa *Guardrails* [34], una librería que permite, entre otras tareas, obtener un diccionario con una serie de parámetros dado un mensaje de entrada.

En este caso, el mensaje de entrada puede ser:

- En la primera iteración, se trata de **todo el historial de la conversación**, para que se puedan obtener parámetros que se hayan ido mencionando “de pasada“, como puede ser el nombre del cliente, que haya dicho antes de entrar a la *skill*.
- En el resto de iteraciones, se incluyen **los dos últimos mensajes**: la pregunta que ha hecho el bot (por ejemplo: “¿qué día quieres reservar?”) y la respuesta del usuario: “mañana”. Adicionalmente, se envía también toda la conversación anterior en la que no se estaba ejecutando ninguna *skill*. De esa forma, si el agente no detectara la *skill*, los mensajes intercambiados seguirían teniéndose en cuenta en la recogida de parámetros.

Una vez se ha construido ese mensaje de entrada, junto con un *prompt* específico de *Guardrails*, se lleva a cabo una llamada a LLM que devuelve el diccionario en cuestión.

3.1.2. Validación de parámetros

Una vez se tienen los parámetros obtenidos por *Guardrails*, estos deben ser validados. La **validación** consiste en tomar los valores tal cual se han recopilado (por ejemplo, la fecha puede ser igual a “mañana“) y convertirlos a un formato deseado, haciendo algunas comprobaciones por el camino. Por ejemplo, “mañana“ se transformaría en una fecha con un formato específico, y se podría comprobar si se encuentra en un rango de fechas permitido.

Los validadores son un componente fundamental del sistema de *Skills* de Votconnect. Permiten hacer comprobaciones y cambios sobre los datos que lee *Guardrails* en el flujo de la *Skill*, para que los parámetros se ajusten al formato que desea el cliente.

Podemos distinguir **dos tipos de validadores**: los simples (que se aplican de manera individual sobre un único parámetro) y los compuestos (que validan de forma compuesta los valores introducidos para distintos parámetros).

Validadores simples

El funcionamiento es el siguiente: si el parámetro es correcto (o, tras aplicarle ciertos cambios, como puede ser cambiar el formato de fecha, pasa a serlo), entonces se devolverá el valor (cambiado, si hubiera sido necesario) y un indicador de que no ha habido fallo. La *skill* entonces se encargará de asignar el nuevo valor. Sin embargo, si hubiera fallo, se devolvería un mensaje de error, que se pasaría al agente para que lo comunicara al usuario.

Se contemplan los siguientes tipos de validadores:

- **Validador de rango numérico:** sirve para comprobar que un valor numérico esté en un rango específico, definido por un valor máximo y un mínimo. De no especificar alguno de ellos, no se impondrá restricción. Por ejemplo, si solo se especifica el mínimo, el máximo será infinito, y viceversa.
- **Validador de email:** este validador, para valores de tipo string, permite comprobar que un correo electrónico tenga un formato correcto: una serie de caracteres sin espacios, un arroba, otra serie de caracteres, un punto y una extensión. Para ello, compara la cadena de entrada con una expresión regular.
- **Validador de teléfono:** este validador, para valores de tipo string, permite ajustar un teléfono al formato deseado, pudiendo comprobar su extensión o añadir una por defecto si se desea.
- **Validador de opciones:** permite comprobar si un valor está dentro de una lista de valores permitidos. Es muy útil, por ejemplo, para locales de restaurantes: “el Grosso de Argüelles”, “el Grosso de calle de Ventura Rodríguez“. . . se refieren al mismo local, pero son nombres distintos.

El funcionamiento es el siguiente: se comprueba si el valor introducido se encuentra, de manera aproximada (se permiten ciertas diferencias, no es necesaria una coincidencia exacta caracter a caracter) en la lista de posibles valores de una opción. De ser así, se pone el nombre principal de la opción (como veremos a continuación, la clave del diccionario). De lo contrario, se devuelve un mensaje de error en el que se indican los posibles valores que puede tomar el parámetro.

Un ejemplo: si la opción es “Grosso Arguelles“, entonces si ponemos “Grosso de Ventura Rodrigues“, el validador devolverá “Grosso Arguelles“.

- **Validador de fecha:** es el que permite expresar fechas en el formato que nos dé la gana (con restricciones, tampoco nos pasemos) a un formato fecha por defecto.

El funcionamiento es muy básico: se comprueban una serie de palabras clave (“hoy”, “mañana”) y patrones (“dentro de x días”, “x del x“) y se llevan a cabo las operaciones necesarias para obtener la fecha en el formato deseado por el usuario.

En caso de que la fecha recuperada no se ajuste a ninguna de las expresiones contempladas, se lleva a cabo una llamada a LLM para obtener la fecha en el formato deseado. Por ejemplo, si la fecha es “el día de la madre”, aunque no es un caso contemplado (al ser demasiado específico), un LLM sí que sería capaz de obtener la fecha.

- **Validador de hora:** este validador, complementario al de fecha, tiene un funcionamiento muy similar: permite detectar expresiones del tipo “dentro de una hora“, “al medio día“, “las cinco menos 10“... Y unificarlas en formato “hh:mm“ (en este caso, no se permite al usuario decidir el formato, ya que es mucho más estandarizado).

Es muy útil para, por ejemplo, un restaurante que gestiona cada día pedidos y puede poner una hora máxima, una hora mínima, no permitir horas anteriores a la actual, no permitir que se pida con menos de X tiempo de antelación, etc.

- **Validador de regex:** este sencillo validador tiene una función básica pero efectiva en muchos casos: permite comprobar si nuestro valor cumple o no una expresión regular. De lo contrario, se devuelve un mensaje de error indicándolo, que el agente recibe y transmite al usuario de manera más sencilla.

Una vez se han validado los parámetros de manera individual, es momento de llevar a cabo las validaciones conjuntas correspondientes, con los validadores compuestos.

Validadores compuestos

Para complementar a los validadores simples, que son capaces de validar cada parámetro por separado, están los **validadores compuestos**. Estos permiten validar varios valores a la vez para comprobar si la combinación de los mismos es correcta.

El funcionamiento es el siguiente: cuando todos los parámetros que componen al validador han sido recopilados, se ejecuta este validador. Una vez ejecutado, no volverá a hacerlo a no ser que alguno de los valores involucrados cambie.

Los validadores compuestos tienen capacidad de borrar los valores involucrados si no cumplen las condiciones, o de simplemente mandar un mensaje con información (muy útil para, por ejemplo, consultar en una base de datos e informar al usuario).

Los tipos de validadores compuestos contemplados son los siguientes:

- **Validador de fecha y hora:** este validador sirve para que, una vez comprobados los valores de fecha y hora por separado para ver que tienen un formato correcto y demás restricciones, llevar a cabo una validación conjunta. Es muy útil para, por ejemplo, comprobar que, para una reserva en un restaurante, si es para el día de hoy, la hora no sea anterior a la actual, o demasiado cercana.
- **Validador de autorización:** Este validador sirve para autenticar a un usuario antes de continuar el flujo de la *skill*. Para ello, toma dos parámetros: uno que actúa como nombre de usuario y otro que lo hace como contraseña. El valor del validador es un diccionario con los nombres de usuarios como clave y las contraseñas cifradas con el algoritmo de hasheo *argon2* [8] (a fin de no almacenar contraseñas en texto plano).

Estos son, de momento, los dos validadores compuestos contemplados. Se trata de un concepto introducido poco tiempo antes de la redacción de este trabajo, por lo que aún no se cuenta con mucha variedad.

En ambos tipos de validadores, el resultado de la validación puede ser uno de los siguientes:

- Parámetro corregido (si fuera necesario) y listo para usarse.
- Error en la validación y mensaje a devolver al usuario o al agente, para informar del motivo y pedir un valor válido.
- Valor no entendido y mensaje a devolver al usuario o al agente, para pedir que repita el valor al no haberlo entendido bien.
- Parámetro corregido y listo para usarse y, además, un mensaje que se desea comunicar al usuario.

3.2. Actualización del payload

Cuando se han validado los datos, tanto de forma simple como compuesta, procedemos a actualizar el *payload* de la *skill* que almacenamos en base de datos como parte de la conversación. Dicho *payload* representa el estado de la ejecución y tiene la siguiente estructura:

- **value:** valor del parámetro,

- **asked:** booleano que indica si se ha preguntado el parámetro,
- **validated:** booleano que indica si se ha validado el parámetro,
- **understood:** booleano que indica si se ha entendido el parámetro,
- **composed-validated:** diccionario que, para cada validador compuesto del que forma parte el parámetro (de haberlos), indica si se ha validado o no,
- **validator-info:** información devuelta por el validador para devolver al usuario o al agente,

En función de ese diccionario, se construirá la salida al usuario.

3.2.1. Salida al usuario

Se contemplan varios casos de cara a la posible respuesta al usuario. Por orden de prioridad, son los siguientes:

- **Caso 1: Fallo de validador.** Aunque varios parámetros fallen en una misma iteración, se tratarán los fallos de uno en uno. Se puede desglosar en varios subcasos, ordenados por prioridad.
 - **Caso 1.1:** Un parámetro de esta iteración ha fallado la validación simple. Se explica el motivo al usuario y se pide un valor correcto.
 - **Caso 1.2:** Un conjunto de parámetros de esta iteración han fallado una validación compuesta. Se explica el motivo al usuario y se piden nuevos valores.
 - **Caso 1.4:** Un parámetro de una iteración pasada ha fallado la validación simple. Se explica el motivo al usuario y se pide un valor correcto.
- **Caso 2:** Aún no se han obtenido todos los parámetros. El agente pregunta por los mismos.
- **Caso 3:** Un parámetro de esta u otra iteración no se ha entendido. Se pregunta de nuevo, pidiendo al usuario que sea más claro. Se deja este caso para el final a fin de no pedir al usuario que repita un valor varias veces seguidas (denota poca inteligencia).

- **Caso 4:** Se han recopilado todos los parámetros necesarios pero el usuario aún tiene que confirmarlos. Tenemos, por tanto, dos subcasos:
 - **Caso 4.1:** El usuario ha confirmado y, por tanto, podemos proceder con el *finish*.
 - **Caso 4.2:** El usuario no ha confirmado y, por tanto, debemos seguir iterando, cambiando parámetros y pidiendo confirmación hasta que esté conforme con todos.
- **Caso 5:** Se han recopilado todos los parámetros y, además, no se requiere confirmación. Por tanto, directamente se llama al *finish*.

De este modo, podemos **guiar la conversación** con el usuario de manera natural hasta recopilar, validar y, de ser necesario, confirmar todos los parámetros antes de llevar a cabo el *finish*.

3.2.2. Finish

Una vez se han recuperado todos los parámetros de la *skill* en el formato deseado y cumpliendo las restricciones pertinentes, podemos usarlos para ejecutar un *finish* que contiene varias **acciones a ejecutar** con dichos parámetros. Estas acciones incluyen mandar un correo electrónico, hacer una llamada a una API, redireccionar a un servicio... incluso funciones ad hoc para ciertos clientes como la gestión de reservas con *covermanager* [12].

La estructura de los finishes es la siguiente: cada *skill* tiene asociado un *Finish*, que es una especie de contenedor de tipos de *finish*, como *FinishApi* o *FinishEmail*. Por tanto, si queremos enviar un correo, debemos crear un *FinishEmail* con los campos correspondientes, un *Finish* que contenga ese *FinishEmail*, y asociar el *Finish* a la *skill*.

Al igual que con los validadores, tenemos varios tipos de *Finish* distintos, pensados para situaciones variadas. Son los siguientes:

- **FinishEmail:** Este *finish* se usa cuando se quiere mandar un correo electrónico. De momento, el destinatario debe ser fijo, pero en el futuro se permitirá que el destinatario también se obtenga como un parámetro más de los introducidos por el usuario (que será su propio correo).

- **FinishDatasource:** Este *finish* es un valor booleano, y es sencillo ya que lo único que tenemos que hacer es marcarlo si deseamos que se usen los parámetros recogidos por la *skill* como filtros en una query de RAG, que será la respuesta.
- **FinishMath:** Este *finish* tan solo lleva a cabo una operación matemática usando como operandos los parámetros recogidos. Esta operación está escrita como código python.
- **FinishIVR:** Este *finish* permite redirigir a un servicio distinto del actual. Así, se simula la redirección entre departamentos que puede tener lugar en un *call center*.
- **FinishApi:** Este *finish* tiene como fin llevar a cabo llamadas a API. En una versión genérica, se llevará a cabo una llamada a API y se le devolverá al agente el resultado de la misma.

Sin embargo, dado que el funcionamiento de las APIs puede llegar a ser muy complejo y las lógicas asociadas pueden ser muy dependientes de ellas, también se contempla la posibilidad de asociar una función específica a ejecutar.

- **FinishAuth:** Este *finish* tiene como fin autenticar a un usuario dado un campo de usuario y una contraseña. Esto también puede hacerse durante el proceso de la *skill* usando un validador compuesto, como ya se ha explicado. **FinishHumanAgent:** Este *finish* permite, en una llamada telefónica, redirigir la llamada a un agente humano, cuyo número de teléfono se especifica como parte del *finish*.

3.3. Creación de skills

La creación de una *skill* puede llevarse a cabo de múltiples maneras. A falta de una interfaz de usuario, en la cual aún se está trabajando, el método más sencillo es hacer una llamada a API, pues como parte del trabajo se crearon múltiples *endpoints* que permitían crear la *skill* al completo, así como añadir, actualizar o eliminar componentes concretos.

En el apéndice [A](#) se muestra el ejemplo de *payload* que se debe incluir en la llamada al *endpoint* de creación para obtener una *skill* para un restaurante de pizzas, que permita hacer un pedido para recoger.

Como se puede apreciar en él, se definen de forma independiente cada uno de los componentes, y de manera muy sencilla, a partir de ese momento, puede usarse la *skill*.

Capítulo 4

Casos de uso

Una vez explicada la arquitectura diseñada para solucionar el problema planteado inicialmente, es momento de exponer algunos de los **casos de uso** contemplados, y que se resuelven con variaciones distintas del concepto de skill.

4.1. Gestión de reservas en un restaurante

Uno de los casos de uso que se concibieron desde un inicio fue el **proceso de reserva en un restaurante**. Se trata de un proceso **fácilmente automatizable** (partiendo, claro, de la arquitectura diseñada, pues realmente automatizar un proceso que requiere una interacción tan directa en lenguaje natural no es para nada sencillo).

Las ventajas de esta automatización para un restaurante es, principalmente, el mantenimiento del **canal telefónico**, bien por mensajería o llamada, para la gestión de reservas (al que aún sigue recurriendo mucha gente a pesar de la posibilidad de reservar por internet que ofrecen muchos restaurantes) **a un precio muy barato**, liberando así al trabajador encargado de coger el teléfono, que ahora podría dedicarse en tiempo completo a sus labores culinarias.

Tras múltiples demos mostradas a diversos restaurantes, este caso de uso tuvo bastante éxito entre los posibles clientes, dado su buen funcionamiento y el *efecto wow* que generaba.

Veamos, por tanto, qué elementos, de los descritos en el capítulo anterior, componen la *skill* de reservas.

4.1.1. Componentes de la skill

Los parámetros que el sistema debería pedir son, como mínimo, los siguientes:

- Fecha y hora de la reserva.
- Número de comensales para la reserva.
- Nombre al que se hace la reserva.

Cualquier otro parámetro, como algún requisito adicional para alérgenos, o la elección de un restaurante específico en el caso de una franquicia, puede añadirse de manera sencilla especificando una descripción.

Las validaciones que deben llevarse a cabo sobre dichos parámetros son las siguientes:

- Que la fecha esté en el formato necesario para la llamada a API que implica la reserva.
- Que la fecha no sea anterior al día de hoy y que no supere una fecha máxima.
- Que la hora se encuentre dentro del horario de apertura del local.
- Que el número de comensales no supere un límite establecido.
- Que haya disponibilidad para la fecha, hora y número de comensales especificado.

Se trata de un validador compuesto que implementa una funcionalidad concreta: la consulta en *covermanager*.

El resultado de las validaciones serán mensajes al usuario que le informarán de aspectos como la disponibilidad a otras horas (en caso de no haberla en el tramo seleccionado) y le permitirán modificar los valores. Una vez asegurado que todos los valores son válidos, el usuario debe confirmarlos, pues ha podido haber algún error al entender los valores que el usuario debería corregir.

Por último, el *finish* de esta *skill* sería una llamada a la API de *covermanager* que solicitaría la reserva. Como en este momento ya se ha comprobado la disponibilidad, el resultado de esta llamada, salvo error del servidor, debería ser exitoso. Igualmente, la respuesta del servidor se proporcionará al agente, que será capaz de interpretarla y comunicársela al usuario de forma natural. Por último, si se ha podido llevar a cabo la reserva, se enviará un SMS al cliente (en caso de que la conversación haya tenido lugar por teléfono).

4.1.2. Ejemplos de funcionamiento

En la Figura 4.1, a la izquierda, se recoge una conversación por chat con un agente de gestión de reservas para un restaurante de ejemplo. Como se puede apreciar, la interacción es similar a la que se tendría con una persona, usándose expresiones temporales como “mañana”, que el LLM debe interpretar y transformar a un formato correcto. Esto, aunque parezca sencillo, fue uno de los problemas con los que nos encontramos.

Una vez la reserva ha sido llevada a cabo, puesto que almacenamos los datos en una base de datos, la siguiente interacción comenzará con el agente informándonos de la reserva que ya tenemos hecha, y preguntando si queremos modificarla o cancelarla (en esta primera versión, no se contempla la posibilidad de volver a hacer otra reserva). Este comportamiento se observa en la imagen de la derecha de la Figura 4.1.

4.2. Consulta de información empresarial en distintos convenios

Otro caso de uso que se ha tratado empleando las *skills* es la **búsqueda de información empresarial en distintos convenios**.

Un convenio colectivo es un acuerdo negociado entre los representantes de los trabajadores y los empleadores para establecer las condiciones laborales en un determinado sector o empresa. Este acuerdo regula aspectos como salarios, horarios de trabajo, y otros derechos y deberes laborales. Por tanto, cualquier pregunta sobre dichos aspectos puede tener una respuesta distinta en función del convenio al que pertenezca la empresa.

De igual manera, una empresa no pertenece a un único convenio, sino que existe una **jerarquía** en la que, dependiendo del sector, la provincia y la comunidad, correspondan varios convenios. El convenio más específico prevalece sobre los superiores, pero si cierta información no está disponible en uno de los convenios, la respuesta está en los más generales.

Como se puede deducir, se trata de un proceso tedioso para los trabajadores de una empresa o los gestores de las mismas, que deben consultar múltiples documentos en orden para dar con la respuesta a una pregunta. Gracias a la arquitectura de *skills*, este proceso **puede ser automatizado**, de modo que, especificando los datos de la empresa, se pueden hacer preguntas y obtener la respuesta ofrecida por cada convenio (especificándose la correcta, que es la del convenio más específico).

4.2.1. Componentes de la skill

Este caso de uso es completamente distinto al descrito anteriormente, lo que demuestra la **flexibilidad** de la arquitectura propuesta, capaz de modelar *skills* tan variadas.

En este caso, los parámetros que se deben pedir al usuario no son siempre obligatorios. Es decir, si el usuario directamente especifica el convenio sobre el que quiere preguntar, no es necesario que le pidamos nada más. De igual manera, si especificamos un NIF o el nombre de la empresa, podemos obtener los convenios, que están almacenados en una base de datos. Sin embargo, si el usuario proporciona un CNAE, es necesario que especifique también la provincia.

Por tanto, no siempre se piden todos los parámetros, pero se debe recopilar alguno de los siguientes:

- El convenio colectivo o sector sobre el que se quiere preguntar.
- El código CNAE sobre el que se quiere preguntar, así como la provincia de España donde se encuentra la sede de la empresa.
- El NIF de la empresa sobre la que se quieren hacer consultas.
- El nombre de la empresa. En caso de que haya varias con el mismo nombre, se preguntará por la provincia para desambiguar.

Las validaciones que se llevan a cabo son:

- Que la provincia sea válida, y se pasa a un formato normalizado para consultar en la base de datos.
- Que el CNAE tenga formato correcto. Es decir, que cumpla una expresión regular.
- Lo mismo con el NIF.
- Por último, una validación consistente en consultar la base de datos para obtener el convenio e informar al usuario al respecto.

Tras confirmar el usuario los datos de su empresa y los convenios, se lleva a cabo un *finish* de tipo IVR, el cual nos permite cambiar de servicio a uno que cuenta con un sistema

de RAG [27] y que busca en una base de datos vectorial aplicando filtros correspondiente con los convenios. Una vez recopilada la información, prevalece aquella que proviene de un convenio **más específico**.

Dentro del servicio de información, en cualquier momento **podemos volver al servicio anterior** a través de una skill sin parámetros que se activa cuando el agente detecta que queremos cambiar de empresa. Dicha skill cuenta con un *finish* de tipo IVR que cambia de nuevo el servicio.

4.2.2. Ejemplos de funcionamiento

En la Figura 4.2, a la izquierda, se muestra una interacción en la que el usuario especifica el nombre de su empresa al bot, que es capaz de obtener el resto de parámetros y no pregunta ninguno más. Tras confirmar, se lleva a cabo el cambio de servicio (inapreciable para el usuario) y a partir de ese momento se pueden hacer preguntas. A pesar de que no es el tema de este trabajo, destaca el buen funcionamiento de la arquitectura RAG, y la posibilidad de ir directamente a la fuente de la información (una *feature* en la que también trabajé).

De igual manera, a la derecha de la Figura 4.2 se muestra un ejemplo similar, solo que en este caso el usuario especifica el CNAE de la empresa sobre la que quiere llevar a cabo las consultas. Puesto que el CNAE no aporta la suficiente información, el bot solicita al usuario la provincia en la que opera la empresa. Una vez el usuario se la ha dado, el funcionamiento es similar. Nótese que la pregunta del usuario es respondida en base a información recogida en el Estatuto de Trabajadores [15], que es el nivel más alto de la jerarquía, puesto que no se ha podido encontrar información al respecto en otros convenios más específicos.

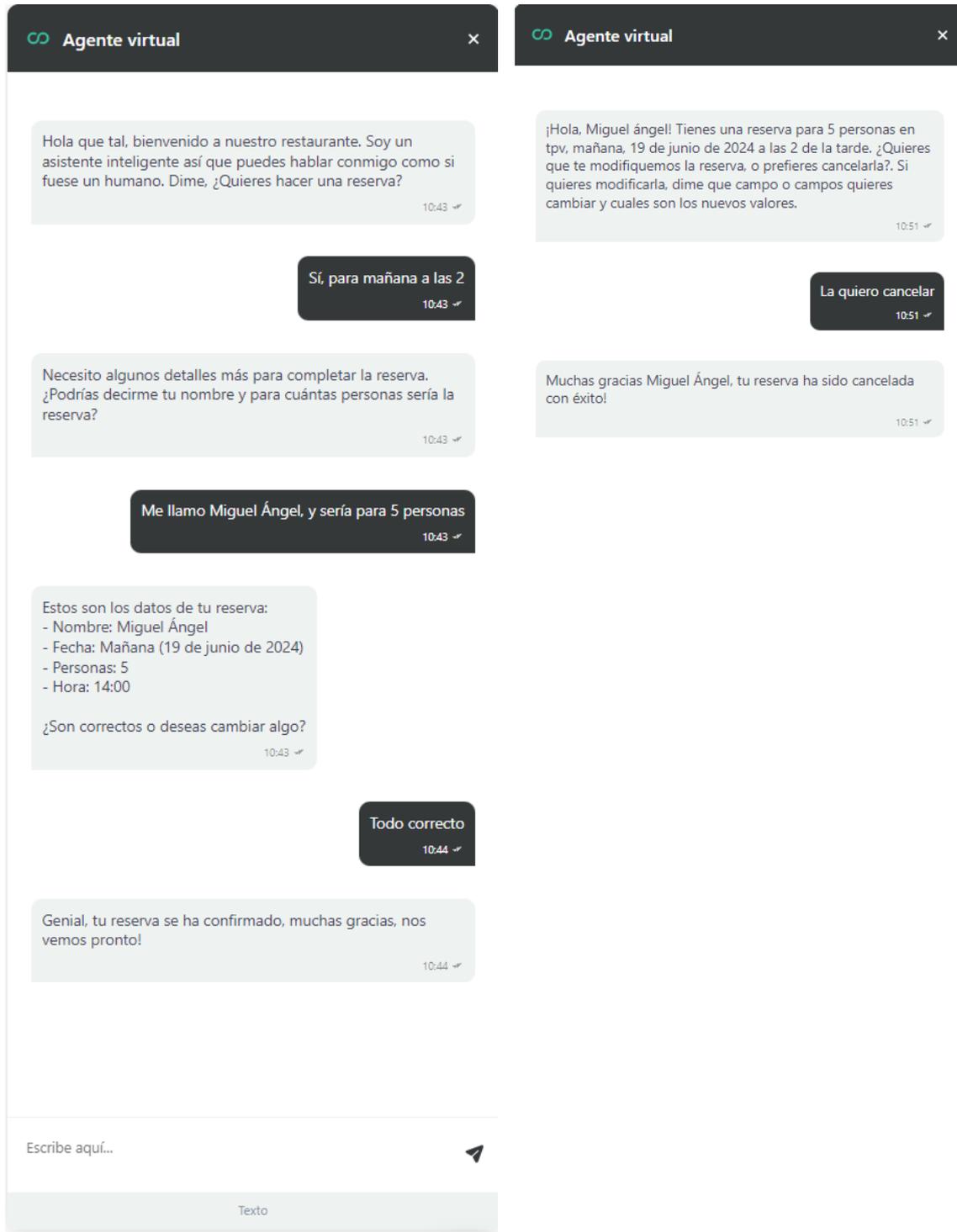


Figura 4.1: Ejemplos de reserva en restaurante

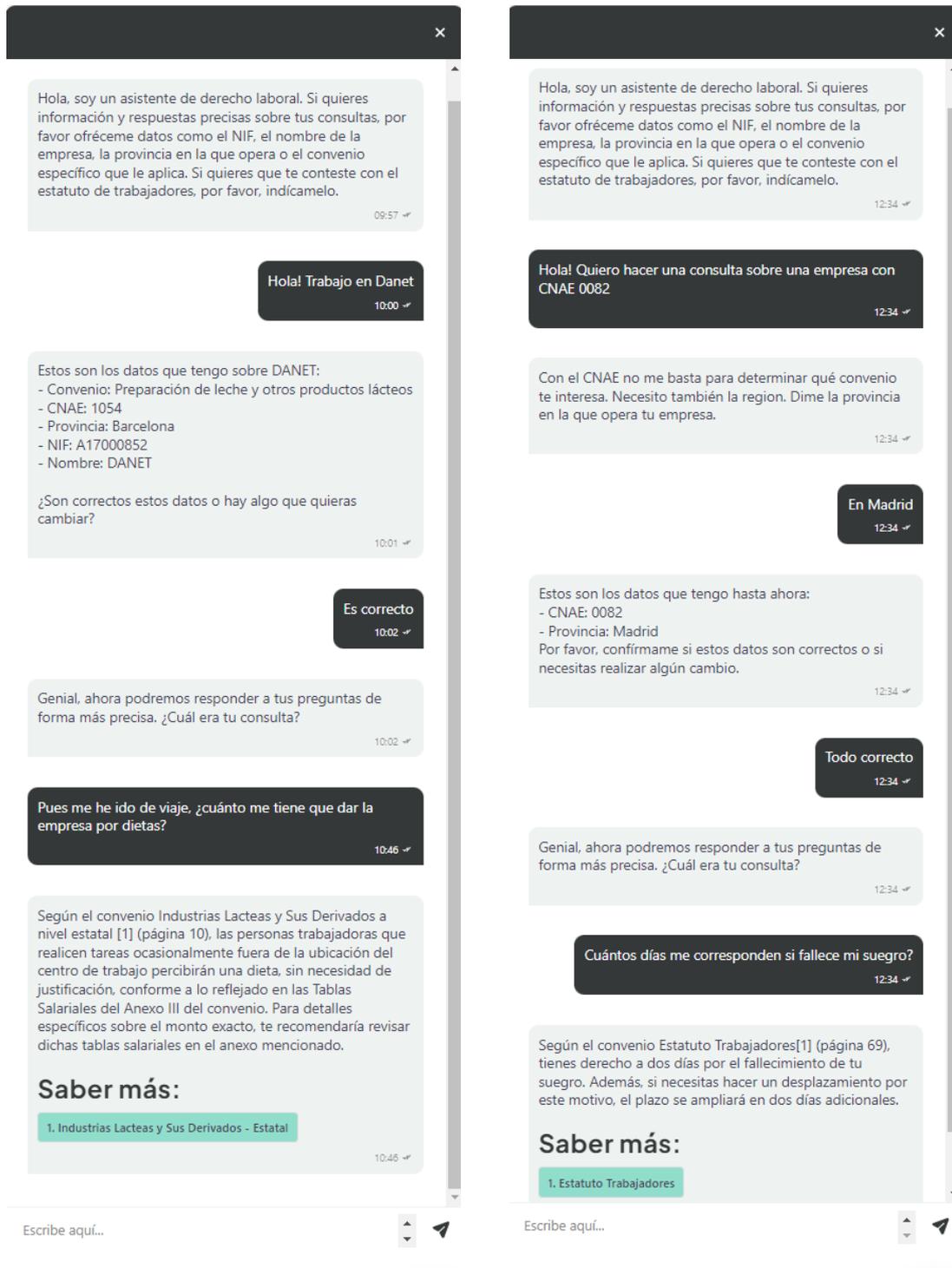


Figura 4.2: Ejemplos de consultas sobre convenios

Capítulo 5

Desafíos y soluciones implementadas

Una vez presentada la arquitectura propuesta y descritos dos de los casos de uso más trabajados, es momento de discutir los **desafíos** a los que nos hemos enfrentado durante su desarrollo, así como las **soluciones** a los mismos.

5.1. Escalabilidad

El primer requisito fundamental para el sistema de *Skills* era la **escalabilidad**, es decir, la posibilidad de que cualquier cliente creara sus propias *skills* de forma sencilla, y que ello no supusiera ningún cambio en el software, sino que la configuración de cada *Skill* se almacenara en una **base de datos**, y el proceso recuperara la información necesaria de la misma.

Este requisito se resolvió con el diseño de un modelo de base de datos cuyo diagrama entidad relación se recoge en la Figura 5.1, el cual fue implementado usando el *framework Django* [14].

Como se puede apreciar, a cada componente de la arquitectura le corresponde una clase en el modelo. De esta forma, una *skill* es un conjunto de objetos de dichas clases, las cuales pueden crearse de manera **sencilla**, modificando los **atributos** de cada uno para dar lugar a casos de uso distintos.

Como se explicó anteriormente, los objetos pueden crearse mediante llamadas a API, o directamente desde el portal de administrador de *Django*. El desarrollo de un *front-end* para permitir a los clientes configurar sus propias *skills* de forma sencilla está en los planes del equipo, pero aún no se ha llevado a cabo.

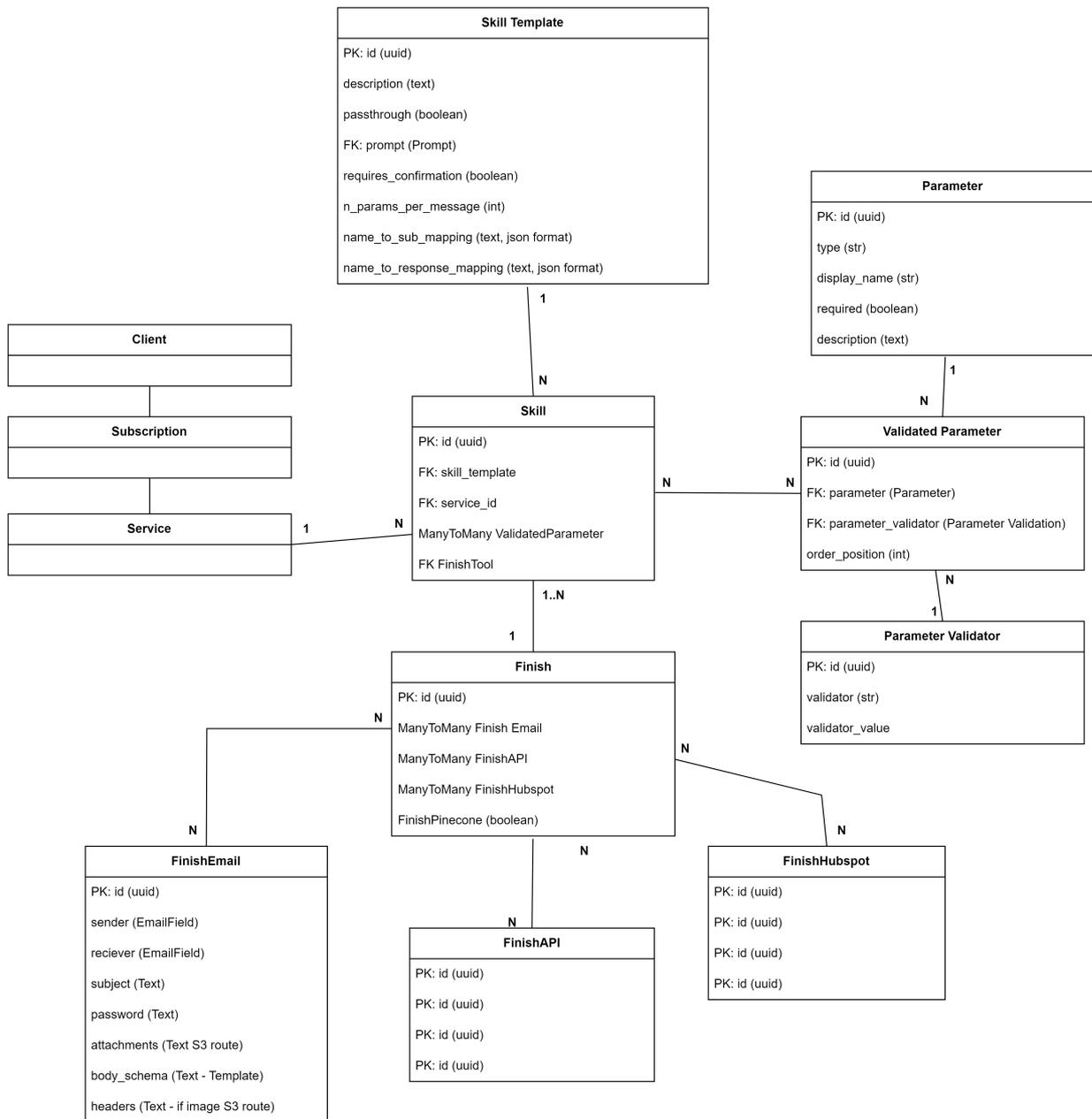


Figura 5.1: Diagrama Entidad Relación de la Base de Datos

5.2. Robustez

En una primera versión, era el propio *AgentExecutor* de LangChain el que, partiendo de los parámetros de cada *tool* que lo componía, **decidía** qué valores debía pedir y cómo pedirselos al usuario. Esta libertad que se le daba al agente permitía un comportamiento muy humano, con conversaciones muy parecidas a las que tendría una persona.

Sin embargo, el agente no era lo suficientemente **robusto**: funcionaba bien, aproximada-

mente, la mitad de las veces. La otra mitad, sufría de **alucinaciones**. Las alucinaciones [40] se dan cuando los LLM responden con información falsa haciéndola pasar por verdadera o, en nuestro caso, cuando se inventan valores de parámetros. Un ejemplo curioso y gracioso, aunque agotador para el equipo de Votconnect, es que el agente se inventaba que el nombre de un cliente que hacía una reserva era Juan. En ningún momento se le decía, pero el agente **lo asumía** y no pedía el parámetro nombre, sino que directamente le asignaba ese valor.

De igual manera, una vez recopilados los parámetros necesarios, el agente debe ser capaz de comunicar al cliente lo que ha ocurrido. Esto, con la primera versión, era un **punto de fallo**. Siguiendo con el ejemplo de las reservas, aunque la llamada a API de *covermanager* devolviera un mensaje de error, en muchas ocasiones el agente comunicaba al cliente que la reserva había sido realizada. ¿Cómo puede ponerse en producción un bot que confirma reservar sin que se hayan registrado en el sistema? Esto daría una imagen **muy mala** al restaurante cliente.

Es por ello que gran parte de nuestro esfuerzo se destinó a robustecer el modelo de *skills*. El primer paso fue que el agente no gestionara la pedida de parámetros: él solo decidiría qué *tool* usar, lo cual sí que **se le da bien**. Por tanto, pedir los parámetros es un proceso que debe gestionarse dentro de la *tool*. Para ello, se usa la ya mencionada librería *Guardrails*. Dicha librería, mediante un refinado *prompt* y una serie de repreguntas a LLM, es capaz de devolver un **diccionario** con los parámetros que le especifiquemos.

De este modo, el agente llama a la *tool* porque, dada la descripción de la misma, considera que debe hacerlo. Sin embargo, como a sus ojos la *tool* no tiene parámetros, **no pregunta nada**, solo ejecuta el código. En ese código se introduce la llamada a *Guardrails*, que, a partir de la conversación, obtiene de manera robusta los parámetros. Si falta alguno, se indica al agente, de forma explícita, **qué parámetros debe preguntar** y en qué orden.

En cuanto a la gestión de los *Finish*, para que el agente informe de manera correcta al usuario de lo que ha ocurrido, se construye el mensaje **en el propio código** en función de los mensajes recibidos de la API correspondiente. Construyendo el mensaje nosotros, podemos ser más **específicos** y conseguir que el agente lo entienda mejor y, por tanto, pueda comunicarlo de forma más clara y **sin errores**.

5.3. Latencia y costes

Las soluciones a los desafíos hasta ahora planteados no son, ni mucho menos, perfectas, y dan lugar a nuevos problemas, como lo son la latencia y los costes. Puesto que los costes son ínfimos en el caso de los modelos GPT, no se tendrán en cuenta en este apartado, pues estamos hablando de precios del orden de 50 céntimos a 5 euros por millón de *tokens*.

Las llamadas a LLM que tienen lugar durante la ejecución del agente de LangChain implican una **latencia**, que dependerá del LLM en cuestión. En el caso de GPT-3.5 turbo, modelo con el que se empezó, la latencia era de unos 1.9 segundos. Sin embargo, no se lleva a cabo una sola llamada: también se hace otra para *Guardrails*, e incluso varias más como parte de algunos validadores.

Aunque unos **seis segundos**, que es la media del total empleado en las interacciones dentro de una *skill*, no parezca un tiempo muy alto, sí que se puede hacer eterno en ciertas situaciones, como una conversación **telefónica**. Por tanto, la latencia es definitivamente un aspecto a mejorar.

Por desgracia, **no hay una solución** a este problema más allá de esperar que surjan nuevos modelos más rápidos que sus predecesores. Parece desolador, pues la solución no depende de nosotros, pero nada más lejos de la realidad. Por ejemplo, durante el proyecto, OpenAI lanzó GPT-4o, como ya se ha mencionado anteriormente. Este lanzamiento permitió **mejores respuestas con una menor latencia**. Por tanto, la solución a la latencia de nuestro sistema pasa por las grandes empresas que entrenan y ofrecen sus LLM... y, por suerte, estamos en buenas manos.

Capítulo 6

Conclusiones y líneas de trabajo futuro

En este capítulo, el último del trabajo, se van a presentar las **conclusiones** extraídas de este trabajo, así como las **líneas de trabajo** que se surgen del mismo.

6.1. Conclusiones

Tras nueve meses de trabajo y la construcción de una arquitectura desde cero, son muchas las conclusiones extraídas de este trabajo. La primera es que, en un campo tan novedoso y en el que tienen lugar tantos avances, **estar en la vanguardia es muy costoso** e implica un proceso continuo de documentación y revisión de artículos y propuestas de otros compañeros de profesión.

Sin embargo, precisamente por la novedad de los LLM y, más en particular, de los agentes autónomos, muchos de los problemas que surgieron durante el desarrollo de este trabajo no habían sido resueltos aún o, al menos, **no estaban documentados**. Por tanto, en el equipo de Votconnect hemos tenido que valorar múltiples propuestas y tomar **decisiones trascendentales** para el proyecto sin una base establecida externamente.

Eso me lleva a la siguiente conclusión: la mejor manera de determinar si una propuesta es o no buena es mediante un proceso iterativo de construcción de un MVP (*Minimum Viable Product*), **medición y análisis** de los resultados. Este análisis permite determinar qué puntos cambiar del MVP de cara a la siguiente iteración... Así, hasta alcanzar el producto final. Este proceso es en el que se basa la metodología *SCRUM* [35].

En cuanto a los LLM, una de las conclusiones más importantes a la que he llegado tras trabajar con ellos todo este tiempo es que hay que entenderlos como **modelos probabilísticos**, al igual que cualquier otro modelo de IA. Esto quiere decir que **no siempre van a funcionar bien**, por mucho que nos esforcemos en que así sea. Las alucinaciones son una realidad, como se ha explicado en el apartado anterior, y es un aspecto que se debe asumir cuando se trabaja con estos modelos.

Por tanto, y centrándonos, ahora sí, en la arquitectura propuesta, otra conclusión a la que llego es que **aún no se ha alcanzado el nivel de robustez necesario**, en cuanto al funcionamiento de agentes autónomos, para la puesta en producción de los mismos. Un agente autónomo que gestione las reservas de un restaurante no debe poder inventarse valores o confirmar una reserva que realmente no ha hecho porque ha alucinado. Sin embargo, con los agentes autónomos que existen en la actualidad, **estos escenarios son posibles**.

Sin embargo, en Votconnect consideramos que nuestro sistema sí que **garantiza la suficiente robustez**, aunque, como se ha explicado, a costa de una “robotización” del bot. Por ello, y esta es la última conclusión que voy a exponer en este trabajo, existe un *trade-off* entre la seguridad de un agente y la naturalidad de la conversación. Actualmente, para llevar un agente a producción, es necesario **sacrificar la fluidez en la conversación** y hacer que dicho agente se parezca más a un robot.

6.2. Líneas de trabajo futuro

En un campo tan cambiante como lo es el PLN, y más aún dentro de los LLM, la arquitectura de *skills* propuesta en este trabajo puede quedar obsoleta en muy poco tiempo, tanto por el avance de los propios LLM como por **nuevas herramientas** propuestas por grandes equipos como el que hay detrás de *LangChain*.

La filosofía de Votconnect respecto al primer caso, la llegada de nuevos y mejores LLM, siempre ha sido la misma: construir un *framework* que permita, con facilidad, **integrar nuevos LLM** sin necesidad de modificar el software. Este objetivo es crucial en un mundo tan cambiante, y se ha demostrado durante este trabajo, en el que se pasó de que el estándar en los modelos GPT fuera GPT-3.5, a contar con GPT-4, hasta llegar al que actualmente es el modelo más rápido y con mejor funcionamiento: GPT-4o. Este cambio **no requirió ningún tipo de adaptación** de nuestro código fuente, por lo que se pudieron integrar las

mejoras de los modelos desde el día de su salida.

Esta idea hace que el producto de Votconnect **mejore con los avances en LLM**, por lo que, en lugar de verse como “competencia”, los lanzamientos de OpenAI, Google o Meta se contemplan como saltos de calidad también en Votconnect. Al fin y al cabo, **el LLM es tan solo el núcleo** alrededor del cual se encuentra el producto de Votconnect.

Sin embargo, ante nuevas propuestas en el campo de los agentes autónomos, el software sí que requiere cambios para adaptarse a las mismas. Es en estas propuestas en las se enmarcan las líneas de trabajo futuro.

Aunque cuando se diseñó la arquitectura de *skills* el *AgentExecutor* de LangChain era el estandarte de los agentes autónomos, en pocos meses **el panorama ha cambiado** y otras arquitecturas similares han surgido. Este es el caso de LangGraph, de los mismos desarrolladores de LangChain.

LangGraph [23] es un *framework* que permite construir sistemas complejos, en los que el control pasa de unos agentes a otros, cada uno con un *prompt*, información y *tools* distintas. Estos sistemas se definen como **grafos** en los que los nodos se corresponden con los agentes y *tools*, mientras que las aristas son las conexiones entre ellos: condiciones que, de satisfacerse, llevan el flujo a un nodo u otro.

De este modo, el proceso **no es tan secuencial** como en la arquitectura propuesta en este trabajo, sino que se cede el control al LLM y este decide qué caminos tomar. Esto, a pesar de suponer una mayor flexibilidad y naturalidad en la interacción, también tiene un inconveniente, como venimos repitiendo en el trabajo: **la falta de robustez**.

Después de hacer algunas pruebas con LangGraph hemos detectado fallos similares a los que se observaron con el *AgentExecutor* antes de introducir *Guardrails*: a veces no pide parámetros necesarios, como el nombre, y opta por **inventárselo** (en este caso, lo ponía como “Usuario”).

Por tanto, el trabajo futuro se vería enfocado a **robustecer LangGraph**, pues se trata de un *framework* capaz de simplificar arquitecturas muy complejas, por lo que merece la pena trabajar sobre él. Para ello, podríamos volver a usar *Guardrails* e introducir nodos dentro del grafo cuya función sea validar los parámetros.

En definitiva, LangGraph parece prometedor y es la línea de trabajo principal a seguir. Sin embargo, hasta entonces, seguiremos refinando el sistema de *skills*, que tiene un funcio-

namiento muy bueno y, sobre todo, con unos estándares de calidad **compatibles con la puesta en producción.**

Referencias

- [1] Hello gpt-4o, May 2024.
- [2] Introducing meta llama 3: The most capable openly available llm to date, Apr 2024.
- [3] Introducing the next generation of claude, Mar 2024.
- [4] ALMEIDA, F., Y XEXÉO, G. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069* (2019).
- [5] ALMUTIRI, T., Y NADEEM, F. Markov models applications in natural language processing: a survey. *Int. J. Inf. Technol. Comput. Sci* 2 (2022), 1–16.
- [6] BARLOW, H. B. Unsupervised learning. *Neural computation* 1, 3 (1989), 295–311.
- [7] BEBIS, G., Y GEORGIPOULOS, M. Feed-forward neural networks. *Ieee Potentials* 13, 4 (1994), 27–31.
- [8] BIRYUKOV, A., DINU, D., Y KHOVRATOVICH, D. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 292–302.
- [9] CHOPRA, A., PRASHAR, A., Y SAIN, C. Natural language processing. *International Journal of Technology Enhancements and Emerging Engineering Research* 1 (2013), 131–134.
- [10] CHUNG, J., GULCEHRE, C., CHO, K., Y BENGIO, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
- [11] CHURCH, K. W. Word2vec. *Natural Language Engineering* 23, 1 (2017), 155–162.

- [12] COVERMANAGER. Hospitality technology partner.
- [13] CUNNINGHAM, P., CORD, M., Y DELANY, S. J. Supervised learning. In *Machine learning techniques for multimedia: case studies on organization and retrieval*. Springer, 2008, pp. 21–49.
- [14] DJANGO SOFTWARE FOUNDATION. Django.
- [15] ESPAÑA, Y ESPAÑA. *Estatuto de los trabajadores*. C.S. de CCOO,, Madrid, 1980.
- [16] FORNEY, G. D. The viterbi algorithm. *Proceedings of the IEEE* 61, 3 (1973), 268–278.
- [17] GROSSBERG, S. Recurrent neural networks. *Scholarpedia* 8, 2 (2013), 1888.
- [18] HARISUDHAN.S. Crewai: A framework for building agents, Jun 2024.
- [19] HASSAN, M. H. Autogpt - empowering automation and beyond, Jun 2023.
- [20] HOCHREITER, S. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6, 02 (1998), 107–116.
- [21] HOCHREITER, S., Y SCHMIDHUBER, J. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
- [22] HUTCHINS, W. J. The georgetown-ibm experiment demonstrated in january 1954. In *Conference of the Association for Machine Translation in the Americas* (2004), Springer, pp. 102–114.
- [23] JIKADARA, B. Langgraph: A comprehensive guide for beginners, May 2024.
- [24] JINDAL, N., Y LIU, B. Review spam detection. In *Proceedings of the 16th international conference on World Wide Web* (2007), pp. 1189–1190.
- [25] KIROS, R., SALAKHUTDINOV, R., Y ZEMEL, R. Multimodal neural language models. In *International conference on machine learning* (2014), PMLR, pp. 595–603.
- [26] LANGCHAIN. Langgraph: Multi-agent workflows, Jan 2024.

- [27] LEWIS, P., PEREZ, E., PIKTUS, A., PETRONI, F., KARPUKHIN, V., GOYAL, N., KÜTTLER, H., LEWIS, M., YIH, W.-T., ROCKTÄSCHEL, T., ET AL. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems 33* (2020), 9459–9474.
- [28] LIN, T., WANG, Y., LIU, X., Y QIU, X. A survey of transformers. *AI open 3* (2022), 111–132.
- [29] MARRERO, M., URBANO, J., SÁNCHEZ-CUADRADO, S., MORATO, J., Y GÓMEZ-BERBÍS, J. M. Named entity recognition: fallacies, challenges and opportunities. *Computer Standards & Interfaces 35*, 5 (2013), 482–489.
- [30] MARTINEZ, A. R. Part-of-speech tagging. *Wiley Interdisciplinary Reviews: Computational Statistics 4*, 1 (2012), 107–113.
- [31] MCKEOWN, K. *Text generation*. Cambridge University Press, 1992.
- [32] MENDEZ, M. F. G. ¿qué es langchain?, Mar 2024.
- [33] NADKARNI, P. M., OHNO-MACHADO, L., Y CHAPMAN, W. W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association 18*, 5 (2011), 544–551.
- [34] RODRIGUEZ, J. Inside guardrails ai: A new framework for safety, control and validation of llm applications, Sep 2023.
- [35] SCHWABER, K. Scrum development process. In *Business Object Design and Implementation: OOPSLA '95 Workshop Proceedings 16 October 1995, Austin, Texas* (1997), Springer, pp. 117–134.
- [36] SHAH, H., WARWICK, K., VALLVERDÚ, J., Y WU, D. Can machines talk? comparison of eliza with modern dialogue systems. *Computers in Human Behavior 58* (2016), 278–295.
- [37] TEAM, G. Gemini: A family of highly capable multimodal models, 2024.

- [38] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., Y POLOSUKHIN, I. Attention is all you need. *Advances in neural information processing systems 30* (2017).
- [39] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., XIA, F., CHI, E., LE, Q. V., ZHOU, D., ET AL. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems 35* (2022), 24824–24837.
- [40] YAO, J.-Y., NING, K.-P., LIU, Z.-H., NING, M.-N., Y YUAN, L. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469* (2023).

Apéndice A

Payload para creación de skill

A continuación se recoge el *payload*, en formato JSON, necesario para crear una *skill* capaz de gestionar pedidos de pizzas en un restaurante.

```
1 {
2   "service_id": "880ee1d5-c192-434e-a2fc-85341de1568a",
3   "name": "Pedir pizza",
4   "description": "Usa esta tool cuando el cliente quiera pedir una pizza
5   para recoger en el local",
6   "requires_confirmation": true,
7   "params_per_message": 2,
8   "response": "Gracias por pedir tu pizza!",
9   "parameters": [
10    {
11      "name": "nombre",
12      "type": "str",
13      "required": true,
14      "description": "El nombre del cliente",
15      "order_position": 1,
16      "get_from_history": true
17    },
18    {
19      "name": "sabor",
```

```
19     "type": "str",
20     "required": true,
21     "description": "El sabor de la pizza",
22     "order_position": 2,
23     "validator_type": "choice",
24     "validator_value": "{ 'Barbacoa': ['Barbacoa', 'BBQ'], 'Cuatro
Quesos': ['Cuatro Quesos'], 'Mediterranea': ['Mediterranea'] }"
25 },
26 {
27     "name": "dia",
28     "type": "str",
29     "required": true,
30     "description": "El dia que el cliente recogerá la pizza",
31     "order_position": 3,
32     "validator_type": "date",
33     "validator_value": "{ 'allows_past': 'false' }"
34 },
35 {
36     "name": "hora",
37     "type": "str",
38     "required": true,
39     "description": "La hora a la que el cliente recogerá la pizza",
40     "order_position": 4,
41     "validator_type": "hour",
42     "validator_value": "{ 'default_range': 'pm' }"
43 },
44 {
45     "name": "local",
46     "type": "str",
47     "required": true,
48     "description": "El local en el que el cliente va a recoger la
```

```

pizza",
49     "order_position": 5,
50     "validator_type": "choice",
51     "validator_value": "{ 'Las Rozas': ['Las Rozas'], 'Alonso Cano':
['Alonso Cano', 'Calle Jose Abascal', 'Spaces'], 'Francos Rodriguez': [
'Francos Rodriguez']}"
52     }
53 ],
54 "composed_validators": [
55     {
56         "name": "fecha_hora",
57         "type": "date_hour",
58         "parameters": [
59             {
60                 "name": "dia"
61             },
62             {
63                 "name": "hora"
64             }
65         ],
66         "value": "{ 'allows_past': false}",
67         "param_name_mapping": "{ 'fecha': 'dia' }"
68     }
69 ],
70 "finishes": {
71     "FinishEmail": [
72         {
73             "sender": "manuel.oliveria@votconnect.com",
74             "receiver": "{miguellangel.davila@votconnect.com}",
75             "subject": "Se ha hecho una reserva",
76             "body": "etc, etc"

```

```
77     }
78   ],
79   "FinishDatasource": true
80 }
81 }
```