TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA

Desarrollo de un chatbot mediante la plataforma RASA

Autor
Raúl Del Pozo Moreno

Director
David Griol Barres

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 7 de septiembre de 2022

Resumen

En este TFG se ha desarrollado un asistente conversacional de tipo chatbot, con la finalidad de proporcionar apoyo a negocios ligados al campo de la restauración como bares, restaurantes y cafeterías, gestionando las reservas de las mesas y permitiendo que se establezca un menú asociado a dicha reserva, de forma que facilite estas funcionalidades a los usuarios interesados en acudir al establecimiento.

Uno de los principales objetivos del trabajo ha sido buscar una solución económica y profesional para el despliegue del chatbot, de forma que tenga una alta disponibilidad y que permita redimensionar la carga de trabajo que pueda tener en un momento determinado según el número de usuarios que lo estén usando, ya sea desplegando más servidores o eliminándolos.

También se pretende estudiar e investigar nuevas tecnologías no aprendidas durante el grado de forma que se muestre la capacidad de adaptación y resolución a este tipo de situaciones, como puede ser el desarrollo del mismo chatbot o el despliegue avanzado del chatbot.

Palabras clave

Chatbot, Asistente conversacional, Restauración, Reservas, Disponibilidad

Abstract

In this TFG, a chatbot-type conversational assistant has been developed, with the aim of providing support to businesses linked to the field of catering such as pubs, restaurants and cafe, managing table reservations and allowing a menu associated with said to be established, in a way that it facilitates these functionalities to users interested in visiting the establishment.

One of the main objectives of the work has been to find an economical and professional solution for the deployment of the chatbot, so that it has high availability and that allows resizing the workload that it may have at a given time according to the number of users that are using the system, deploying more servers or destroying them.

It is also intended to study and investigate new technologies not learned during the degree so as to show the ability to adapt and resolve these types of situations, such as the development of the chatbot itself or the advanced deployment of the chatbot.

Keywords

Chatbot, Conversational assistant, Restoration, Reservations, Availability

Agradecimientos

A mis padres, por apoyarme siempre de manera incondicional y animarme a seguir hacia delante.

Índice de contenidos

Índice de Figuras

Capítulo 1. Introducción

1.1. Motivación

Con la llegada del COVID-19, los confinamientos domiciliarios, las olas de nuevos infectados y ahora con la viruela del mono, uno de los sectores que más afectada ha visto su profesión es la restauración. Aunque las medidas de seguridad ya no sean obligatorias en su mayoría, sigue habiendo personas a las que la situación les sigue pareciendo peligrosa o incluso incómoda, al encontrarse con gente sin mascarilla o rodeada de demasiada gente.

Creo que sería interesante estudiar el desarrollo de un chatbot que actúe como intermediario entre los puntos de restauración y los clientes, de forma que estos se puedan sentir más atraídos y cómodos para acudir a dichos lugares al facilitar un medio de reserva.

Otra de las motivaciones para realizar este desarrollo ha sido la de proporcionar una herramienta genérica la cual se pueda personalizar para cada usuario, de forma que el código de la aplicación actúa como una plantilla sobre la cual trabajar, como modificar la forma de conexión a la base de datos del propio establecimiento para las reservas o la forma en que se quiere que el chatbot interactúe con los usuarios.

Además, este desarrollo proporciona una oportunidad para explorar y aprender tecnologías y herramientas que no he utilizado antes, como Rasa, además de obtener experiencia con el despliegue de una aplicación funcional en la nube.

1.2. Objetivos

Con este trabajo se pretenden conseguir los siguientes objetivos:

- Aprender sobre la herramienta Rasa.
- Estudiar y aprender la metodología necesaria para elaborar un chatbot.
- Investigar el uso de diferentes herramientas y elegir el software más adecuado para el desarrollo.
- Consolidar una metodología de CI/CD.

1.3. Justificación del uso de chatbots

A la hora de interactuar con una persona podemos encontrarnos con muchas situaciones diferentes en función de la actitud de esta persona, es posible que haya tenido un mal día y nos responda negativamente o puede que esté estresada por lo que la interacción con ella será difícil.

Este comportamiento se puede aplicar muy fácilmente a trabajador que estén de cara al público en su puesto de trabajo, más en entornos estresantes como puede ser un restaurante, por el cual pasan muchas personas a lo largo del día, las cuales puede sean comprensivas o pueden que quieran las cosas al momento sin esperas, lo que repercute en el trabajador a pesar de que este no tenga la culpa de los problemas que puedan surgir.

El uso de chatbots para la interacción con personas al realizar procesos es una herramienta muy útil ya que se puede limitar la interacción con el usuario de una manera personalizada, lo cual proporciona algunas ventajas:

- Libera a trabajadores de carga laboral permitiendo que atiendan tareas más importantes.
- Los trabajadores no tienen que lidiar con clientes agresivos por lo que disminuye su nivel de ansiedad y estrés.
- Los usuarios disponen a cualquier hora de un medio para interactuar con el establecimiento sin tiempos de espera

El uso de chatbots no solo trae ventajas, también tiene algunos inconvenientes como:

- En caso de problema, es posible que el usuario se sienta frustrado al no poder conseguir o adquirir un soporte inmediato para solucionar su problema.
- Tiene un coste de mantenimiento.
- Elimina responsabilidades sobre los empleados, lo que puede conducir a despidos.

1.4. Presupuesto

Para el desarrollo de este trabajo no se ha dispuesto de ningún presupuesto al contar de una fuente de ingresos limitada, para ello, se han utilizado en la medida de lo posible herramientas Open Source en su plan gratuito.

En el caso del despliegue del sistema, se ha recurrido a Google Cloud que, aunque no es Open Source, dispone de un plan de prueba que proporciona $300 durante 3 meses.

El coste estimado de un despliegue mínimo de este sistema basado en el uso actual es de aproximadamente $50 al mes, compuesto por los siguientes servidores:

- Servidor Nginx para balanceo de carga y reverse proxy.
- Servidor Redis para almacenar base de datos en memoria.
- Servidor Rasa donde se gestionan las peticiones de los usuarios.
- Servidor Rasa Actions utilizado por el servidor de Rasa.

1.5. Planificación

Para planificar y desarrollar este proyecto se ha utilizado GitHub mediante el uso de sus herramientas.

Primero se estableció una serie de requisitos mínimos que debería tener el chatbot y se reflejaron en issues mediante una descripción de la tarea a realizar.

Una vez que se tuvo una serie de tareas iniciales, se investigó la forma de clasificar estas issues de forma que se pudieran desarrollar en un contexto, para esto se estudió el uso de proyectos y milestones.

Puesto que los milestones están orientados a establecer fechas límites para ser completados y los proyectos actúan de una forma más general, se decidió crear milestones donde clasificar las issues y usar proyectos para issues más generales o que no están en un punto de realización inmediato, en este caso, para issues a realizar en el futuro.

Así, se crearon issues tanto para investigación como para desarrollo, reflejando el proceso realizado en cada una de ellas mediante comentarios en la propia issue, esto se puede consultar accediendo al repositorio mediante el enlace proporcionado en el **Anexo 1. Repositorio de GitHub**.

1.6. Estructura del documento

Este documento se estructura de la siguiente forma:

- Capítulo 1. Introducción

En este capítulo se introduce de forma general al trabajo realizado, exponiendo la motivación por la que realizarlo, qué objetivos se pretenden conseguir con su realización, una justificación derivada de la motivación expuesta por la cual se indica el problema concreto a resolver, la planificación seguida en el desarrollo del trabajo y finalmente la estructura del documento.

- Capítulo 2. Estado del Arte

Este capítulo trata sobre la historia de los chatbots, desde su origen hasta la actualidad, especificando sus características, aplicaciones en el mercado actual y herramientas utilizadas hoy en día para la construcción de chatbots.

También se tratan las bases de datos, utilizadas para almacenar información de la aplicación y que al igual que los chatbots, se introduce sobre su historia y los diferentes tipos que hay.

Finalmente se hace referencia al cloud computing, donde se explica su historia, utilidad y características.

- Capítulo 3. Desarrollo

En este capítulo se explica el desarrollo seguido en la realización del proyecto, empezando sobre la estructura general de la aplicación, las herramientas usadas y una pequeña explicación del tipo de despliegue realizado, seguido de un análisis de las herramientas usadas y la forma en la que han sido usadas.

- Capítulo 4. Conclusiones y Trabajo Futuro

El capítulo 4 pone de manifiesto las conclusiones obtenidas y los objetivos conseguidos con el desarrollo de este trabajo, así como los futuros desarrollos que hasta el momento se han planteado.

- Bibliografía

Recopilación de fuentes bibliográficas consultadas para la realización del trabajo y la memoria.

- Anexo

Apartados adicionales a la memoria como el repositorio de código utilizado y un pequeño manual de usuario.

Capítulo 2. Estado del Arte

En este capítulo se procederá a estudiar la historia de las diferentes herramientas contempladas para el desarrollo del trabajo, como los chatbots, las bases de datos o el cloud computing, incidiendo en los tipos y características de dichas herramientas.

2.1. Chatbots

2.1.1. ¿Qué es un chatbot? Tipos y características de los chatbots

Un chatbot es una herramienta utilizada para mantener una conversación con una persona mediante texto o voz.

Podemos encontrar dos principales categorías de chatbots [13]:

- Basados en reglas
    - Los bots basados en reglas o bots de árbol de decisión tienen como característica que obedecen pautas predefinidas según la situación que se presente.
    - Es el tipo de bot más sencillo de realizar ya que no requiere de conocimientos avanzados de programación y, por ende, tiene la ventaja de su rapidez y de su precio económico, en cambio, su funcionamiento es menos flexible en general, sobre todo en situaciones no controladas.
    - Este tipo de bots se caracterizan por ser utilizados principalmente en interacciones simples, el usuario hace una pregunta sencilla y se le responde con una respuesta preestablecida.

- Basados en inteligencia artificial
    - Estos bots utilizan aprendizaje automático, son capaces de comprender el contexto y patrones de comportamiento, permitiendo respuestas más enriquecidas para el usuario.
    - Utilizan el procesamiento de lenguaje natural (PLN) y evolucionan según recopilan información, adaptándose a nuevas situaciones.
    - A pesar de tener las ventajas de la naturalidad del lenguaje y su adaptabilidad al usuario, tiene como desventaja el requerimiento de conocer lenguajes de programación más avanzados y en inteligencia

artificial, lo que supone que el desarrollo de este tipo de bot sea más costoso.

Entre sus características se encuentran las siguientes [3]:

- Disponibilidad
    - Un chatbot debe estar disponible las veinticuatro horas los siete días de la semana, no necesita descansar o parar, salvo por algún motivo imprevisto como caída de servidor o problema en el mismo chatbot.

- Personalización
    - Según la temática o los requisitos que haya, el chatbot se puede realizar de forma que se adapte a una temática a medida que haya más interacciones.

- Identificación
    - Aunque uno de los propósitos de un chatbot sea proporcionar una experiencia lo más natural posible, es necesario que la otra persona sea consciente de que habla con un programa, así, en caso de un problema, esta persona sabrá que existe y podrá reportarlo.

- Lenguaje natural
    - La intención de un chatbot es proporcionar naturalidad en el lenguaje, esto es fácilmente posible si el chatbot cuenta con una IA, de forma que reconozca ciertos aspectos de la conversación y se adapte a ellos.

- Claridad y brevedad
    - Cuando se realiza una pregunta a un chatbot, este debe dar una respuesta acorde a lo preguntado, es decir, la respuesta debe contener un valor importante para el usuario.

2.1.2. Historia

El primer asistente conversacional apareció en 1960 de la mano de Joseph Weienbaum, profesor en el MIT, este chatbot se basó en el reconocimiento de pregunta-respuesta, determinando la respuesta según palabras clave extraídas de la frase introducida. Por ejemplo, si la frase contenía la palabra “madre”, la respuesta sería referida a “familia” [1].

Esto supone que el chatbot debería superar estas fases:

1. Identificación de palabras clave.
2. Descubrir un contexto mínimo.
3. Elección de transformaciones apropiadas.
4. Generación de respuesta adecuada.
5. Capacidad de reacción ante la ausencia de palabras críticas.

El siguiente chatbot en aparecer fue Parry en 1972, creado por un psiquiatra llamado Kenneth Colby, con la intención de simular un enfermo de esquizofrenia, en el proyecto “Overcoming Depression”, cuyo objetivo era comprender el funcionamiento de dicha enfermedad.

En 1989 apareció Chatterbot, creado para TinyMud, un videojuego sobre mazmorras, donde el bot proporcionaba conversaciones multiusuario y escenarios simulados, de esta forma, los usuarios creyeron que todos los jugadores eran una sola persona, cuando no era así.

En 1995 se creó Alice (Artificial Linguistic Internet Computer Entity) a partir de Eliza, este nuevo chatbot utilizaba ejemplos de lenguaje natural obtenidos desde una web, cuyos datos categorizados usando el formato AIML (Artificial Intelligence Mark-up Language), en un intento de simular una conversación los más natural posible.

En 1997 Microsoft creó su primer agente conversacional disponible en Windows, llamado Clippy, con el objetivo de ayudar en el uso de Microsoft Office.

En 2011 Apple desarrolló a Siri, basado en versiones desarrolladas por DARPA (Agencia de Proyectos de Investigación Avanzados de Defensa) perteneciente al Departamento de Defensa de Estados Unidos. Este asistente aumenta su conocimiento según la cantidad de usuarios que lo usen, lo que le permite aprender con su uso. Siri sigue disponible hoy en día y es ampliamente usado, con funciones que van desde responder preguntas, gestionar una agenda, hacer recomendaciones o aprender las preferencias del usuario, permitiendo dar una experiencia más personalizada.

Ese mismo año aparece Watson, una IA desarrollada por IBM con una finalidad de aprendizaje, de forma que puede ayudar con el descubrimiento de conocimientos relacionados en diferentes áreas como puede ser la sanidad y los servicios financieros.

Finalmente, en 2014 y 2016, aparece Alexa y Google Assistant respectivamente, estos dos asistentes tienen la particularidad de relacionarse con el usuario a través de reconocimiento de voz, lo que permite que el lenguaje natural usado para responder el usuario proporcione una experiencia más natural [2].

2.1.3. Aplicaciones

Como se ha podido observar a lo largo de la historia de los chatbots, las aplicaciones de estos aumentan conforme avanza el tiempo, al principio, no había una necesidad imperante para su uso, más allá que el de investigación.

Pero una vez que el tiempo pasa, y la tecnología avanza, se puede observar que un chatbot tiene infinitas posibilidades de uso, ya sea desde tareas simples de pregunta-respuesta, hasta ayudar en investigaciones científicas.

Nombrar todas las aplicaciones en las que se puede usar un bot sería casi imposible, pero algunas de ellas serían las siguientes:

- Soporte (información, ayudas)
- Encuestas
- Juegos
- Investigación
- Salud
- Seguridad
- Reservas
- Ahorrar costes

2.1.4. Frameworks para el desarrollo de chatbots

Un framework también es conocido como un esquema o marco de trabajo que ofrece una estructura base para elaborar un proyecto específico [4].

En el ámbito de los chatbots, existen numerosos frameworks a elegir, tanto de código privado como Open Source, a continuación, nombraré algunos y sus principales características [5].

- Dialog Flow
    - Este framework utiliza la inteligencia artificial de Google y es una de las herramientas de Google Cloud Platform (GCP). No es un framework Open Source.
    - Tiene un precio de uso según las características que se utilice, generalmente en función de las peticiones realizadas o del tiempo de procesamiento usado. [7]
    - Es muy intuitivo de usar ya que proporciona una interfaz de usuario desde donde configurar el chatbot.
    - Es compatibles con múltiples SDK, como Node.js, Go, Java, Python, C#, Ruby [6].

- Microsoft Bot Framework
    - Este framework Open Source está diseñado por Microsoft, accesible a través de GitHub https://github.com/microsoft/botframework-sdk y de la plataforma Azure.
    - Proporcionan una versión de pago (subscripción $0.50 cada 1000 mensajes) y otra gratuita con limitación de uso (10000 mensajes por mes) [11].
    - Permite la configuración del bot mediante plantillas y acceso a todos los productos Microsoft, incluido su IDE Visual Studio o Visual Studio Code.

- Amazon Lex
    - Servicio de inteligencia artificial de AWS de código privado.
    - Proporciona herramientas para Deep Learning y se integra con AWS Lambda [8].
    - Al igual que otros framework tiene una limitación de uso según el plan usado, con costes desde $1,50 para 2000 solicitudes de texto ($0,00075 la solicitud) hasta $32 para 8000 solicitudes de texto ($0,004 la solicitud). En el campo de streaming, los precios varían entre $4 y $56, mientras que con el diseñador automatizado del chatbot los precios van en función del tiempo de computación ($0,50 por minuto)
    - Según su propio “Pricing”, el gasto varía desde $150 hasta $500 [9].

- Rasa
    - Framework totalmente Open Source que permite ejecutar el asistente en casi cualquier plataforma, incluyendo una infraestructura propia.
    - Permite el despliegue del asistente directamente sobre el sistema de computación, o utilizar herramientas como Docker o Kubernetes, lo que le proporciona una escalabilidad enorme.
    - Tiene una documentación clara y directa, además de una comunidad activa en su foro.
    - Está preparada para que el usuario solo tenga que preocuparse de gestionar la interacción del usuario (preguntas, respuestas) y la relación entre ellas.
    - Soporta aprendizaje interactivo, se tiene la capacidad de crear el bot mientras conversas con él, indicando a que tiene que reaccionar o cómo interpretar ciertas características.
    - Dispone de un plan empresarial (Rasa Enterprise) antes llamado Rasa X, que no revela coste salvo que se contacte. El plan gratuito se llama Rasa Open Source, el cual no cuenta con la interfaz de usuario que Rasa Enterprise proporciona, mediante la cual se puede crear el bot en tiempo real [10].

Hay muchos otros frameworks para el desarrollo de chatbots que no han sido mencionados, pero todos ellos comparten casi las mismas características, que son fáciles de usar, que con ellos se puede lograr lo último en el mercado, que utilizan la última tecnología… al final, por lo que he podido ver e intuir, a la hora de escoger un framework para el desarrollo de un bot habría que fijarse principalmente en lo siguiente:

- Open Source vs Closed Source
    - Los proyectos Open Source tienen por lo general una comunidad mucho más amplia y colaborativa que los proyectos no Open Source, esto se debe a que, en dichos proyectos, cuando existe un problema cualquier usuario va a poder ayudarte además de los empleados de dicha compañía. En un proyecto que no es Open Source, la atención al usuario suele ser más directa y formal, utilizando canales más cerrados o no tan accesibles como puede ser GitHub.
    - Como menciono en el siguiente punto, los proyectos Open Source suelen ser por lo general mucho más económicos (si no gratis) que los proyectos cuyo código fuente es cerrado.

- Precio
    - Independientemente de que un proyecto sea Open Source o no, el precio de una herramienta es algo a tener en cuenta siempre.
    - Un proyecto Open Source por lo general cuenta con un presupuesto más limitado y cuenta con el apoyo de la comunidad para sobrevivir, en cambio, un proyecto de código cerrado perteneciente a una compañía que no sigue la filosofía Open Source, suele disponer de una cartera más potente con la que realizar desarrollos, permitiendo el uso de más funcionalidades.
    - Así, si bien es cierto que hay que abogar por el uso de tecnologías que permitan la transparencia de sus acciones, el uso de herramientas de código cerrado por lo general, va a proporcionar más facilidades y seguridad en el uso de dicha herramienta, por ello, es necesario realizar un adecuado estudio de las herramientas a usar.

- Características
    - Se debe analizar si el uso de dicho framework se ajusta al propósito del desarrollo, si el asistente a desarrollar requiere de ciertas características como el reconocimiento de voz, se debe buscar una herramienta que lo proporcione, si se sabe que dicho bot nunca va a utilizar reconocimiento de voz, sería interesante estudiar alternativas que no dispongan de dicha característica ya que ello puede influir en el

precio que pueda llegarse a pagar o incluso en la propia funcionalidad del bot.

- Despliegue
    - Hay que estudiar si el asistente desarrollado puede desplegarse en la plataforma en la que queremos que almacene el asistente, por ejemplo, si se desarrolla bot sencillo, basado en reglas que se encarga de informar de la hora, ¿tiene sentido albergar ese bot en una instancia de AWS EC2? Este despliegue implicaría tener una máquina de computación de un sistema determinado y una serie de características que el bot no va a utilizar y, por lo tanto, se estaría perdiendo dinero y desaprovechando los recursos. En este caso concreto, se podría crear una imagen Docker, con lo mínimo y necesario para funcionar y albergarse en una plataforma que permita costear y ejecutar contenedores Docker, como puede ser AWS ECR para almacenar la imagen y AWS ECS para ejecutar el contenedor, o usar una plataforma que esté diseñada para albergar despliegues ligeros de Docker como Heroku, aunque ya no disponga de un plan gratuito.

2.1.5. Lenguaje natural

En un asistente conversacional, una de las partes más importantes es el reconocimiento y comprensión de los mensajes del usuario, esto se consigue a través del Procesamiento de lenguaje natural (PLN).

Al participar en una conversación, lo que las personas analizamos inconscientemente no es solo lo que sería el texto del mensaje, sino que también analizamos otras características intrínsecas a dicho mensaje como es el tono, expresiones específicas, contexto…

En un asistente virtual, este procesamiento del lenguaje natural, suele ir acompañado de dos elementos:

- Comprensión del Lenguaje Natural - Natural Language Understanding (NLU)
- Generación de Lenguaje Natural - Natural Language Generation (NLG)

El primer elemento tiene como objetivo detectar el significado del mensaje transmitido por el usuario, mientras que el segundo elemento busca la capacidad de

generar mensajes mediante el uso de estructuras de datos acordes al diálogo del usuario.

Así, podemos encontrar cinco etapas principales en el procesamiento del lenguaje natural:

1. Presentación
2. Conversión a datos estructurados
3. Decisión
4. Datos
5. Conversión de datos estructurados a texto

La primera etapa se sitúa en el intercambio de mensajes entre el usuario y el asistente, como puede ser WhatsApp, Telegram o Slack.

La segunda etapa se encarga de convertir el mensaje del usuario en datos estructurados de forma que el chatbot pueda entender.

La tercera etapa consiste en el proceso de decisión del chatbot, determina que quiere el usuario y la forma en la que puede responder.

La cuarta etapa analiza los datos preprogramados y datos aprendidos mediante Aprendizaje Automático con la finalidad de generar decisiones y textos.

La quinta etapa convierte los datos estructurados y en texto mediante NLG [12].

2.1.6. Chatbots en entornos profesionales

Los chatbots hoy en día están muy extendidos en los entornos profesionales, ya que principalmente aportan una función que una persona no puede realizar al completo, por ejemplo, un chatbot está disponible en cualquier momento, no importa que sea de madrugada o a mediodía, mientras el servidor donde esté alojado el chatbot esté en funcionamiento, el chatbot estará disponible. Además, aun en caso de que el servidor no lo estuviera por algún incidente no previsto, existen mecanismos de respaldo para asegurar la disponibilidad.

Otro punto importante es el coste, un chatbot puede realizar tareas repetitivas sin esfuerzo, por ejemplo, un chatbot puede sustituir, no solamente a un trabajador de atención al cliente telemático, sino a múltiples de ellos. Esto puede incurrir en dos situaciones, la primera es el despido de dichos trabajadores al no ser ya necesarios, lo que genera malestar. La otra situación podría ser que dichos trabajadores que ya no cumplen la función de atención al cliente, puedan ser reubicados para realizar otra tarea, permitiendo una mejor gestión de recursos.

El aprendizaje es una característica importante en los chatbots, con el tiempo y el uso, un chatbot puede ser capaz de aprender comportamientos no previstos, aumentando la funcionalidad del mismo. Por otro lado, esto requiere un alto nivel de programación y es muy difícil hacer que un chatbot se comporte como una persona, por lo que siempre es susceptible a fallos y puede encontrarse situaciones donde no sabrá como responder, mientras que una persona no tendría ese problema.

2.2. Bases de datos

2.2.1. Historia

Podríamos definir una base de datos como cualquier elemento capaz de almacenar información, como pueden ser libros o registros [18].

En términos de computación, podríamos decir que la primera Base de datos apareció en 1884, cuando Herman Hollerith crea una máquina automática de tarjetas perforadas con la finalidad de realizar un censo.

Más tarde, alrededor de 1950, se crean las cintas magnéticas con la finalidad de automatizar la información y la creación de respaldos de seguridad. Una década más tarde, con la bajada de precios en los computadores, se popularizó el uso de discos, de forma que el acceso a la información almacenada en ellos se hizo más factible.

En la década de 1960, Charles Bachman llevó a cabo el desarrollo del IDS, que supuso la creación de un nuevo tipo de base de datos conocido como modelo en red, estableciendo un estándar en los sistemas de bases de datos.

EN la década de 1970, Edgar Frank Codd, definió el modelo relacional, lo que supuso el nacimiento de la segunda generación de los Sistemas Gestores de Bases de Datos (SGDB), dando lugar a la creación de la empresa “Relational Software System”, conocida hoy en día como “Oracle Corporation”.

En 1980 aparece por primera vez SQL, estableciéndose como un estándar en la industria, ya que el nivel de programación de una base relacional era significativamente más sencillo que las bases jerárquicas y de red.

En la década de 1990, se apostó por la investigación de bases de datos orientadas a objetos, dando lugar a herramientas como Microsoft Excel y Access, lo que creó la tercera generación de sistemas gestores de bases de datos.

Hoy en día existen tres compañías que dominan el mercado de las bases de datos: IBM, Microsoft y Oracle [14].

2.2.2. Base de datos jerárquicas

Las bases de datos jerárquicas almacenan información en una estructura jerarquizada, parecida a un árbol. El objetivo de esta base de datos es gestionar grandes volúmenes de datos.

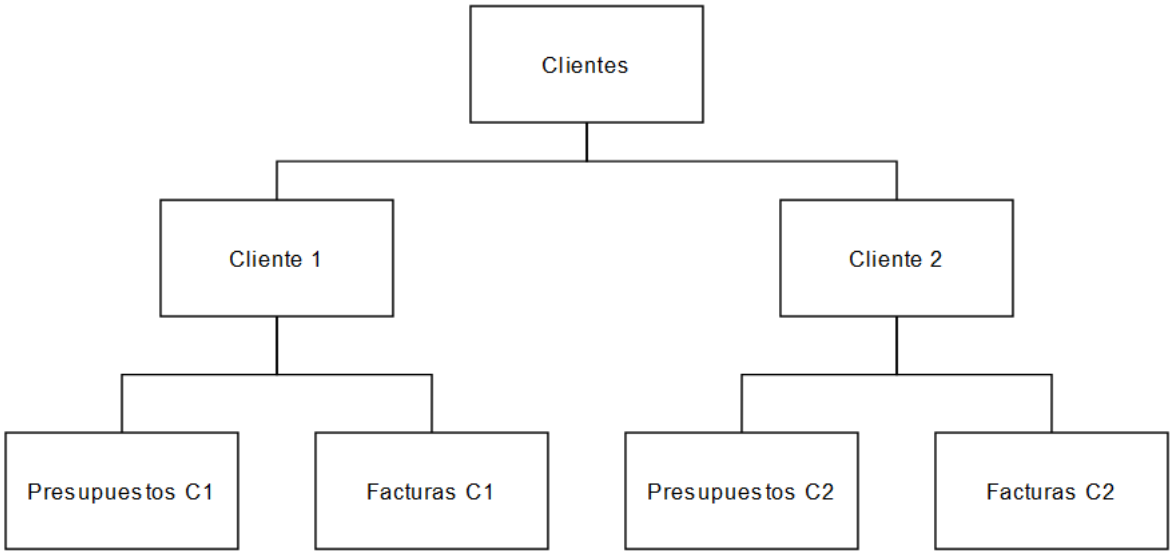

Figura 1: Estructura de base de datos jerárquica.

En la figura 1 se puede observar la estructura típica de este tipo de bases de datos, en ella, se tiene un nodo padre “Clientes”, el cual tiene como hijos los nodos de “Cliente 1” y “Cliente 2”, así mismo, cada uno de estos nodos hijos tiene otros nodos hijos de forma que los nodos van ramificándose.

Proporcionan una navegación rápida entre los nodos del árbol ya que sus conexiones no son dinámicas, es fácil de ver y comprender, establecen relaciones y globaliza la información, de forma que cualquier usuario tiene acceso a los datos, también permite mantener una integridad de información y una independencia de datos.

Sin embargo, esto provoca ciertas situaciones donde presentan desventajas a su uso, por ejemplo, al no haber una independencia entre nodos, si se quiere acceder a ciertos datos hay que pasar por todos los padres de dicho nodo.

Al no poder los hijos referenciar a más de un padre, obliga a una mala gestión de la redundancia de datos, ya que se duplica información, y, por tanto, un mayor volumen de datos a almacenar [15].

2.2.3. Base de datos en red

Este tipo de base de datos está formada por registros enlazados entre sí creando una red, de forma similar a los campos de bases de datos relacionales, con la particularidad de que estos sí permiten a los registros tener múltiples precedentes, a diferencia del modelo jerárquico [16].

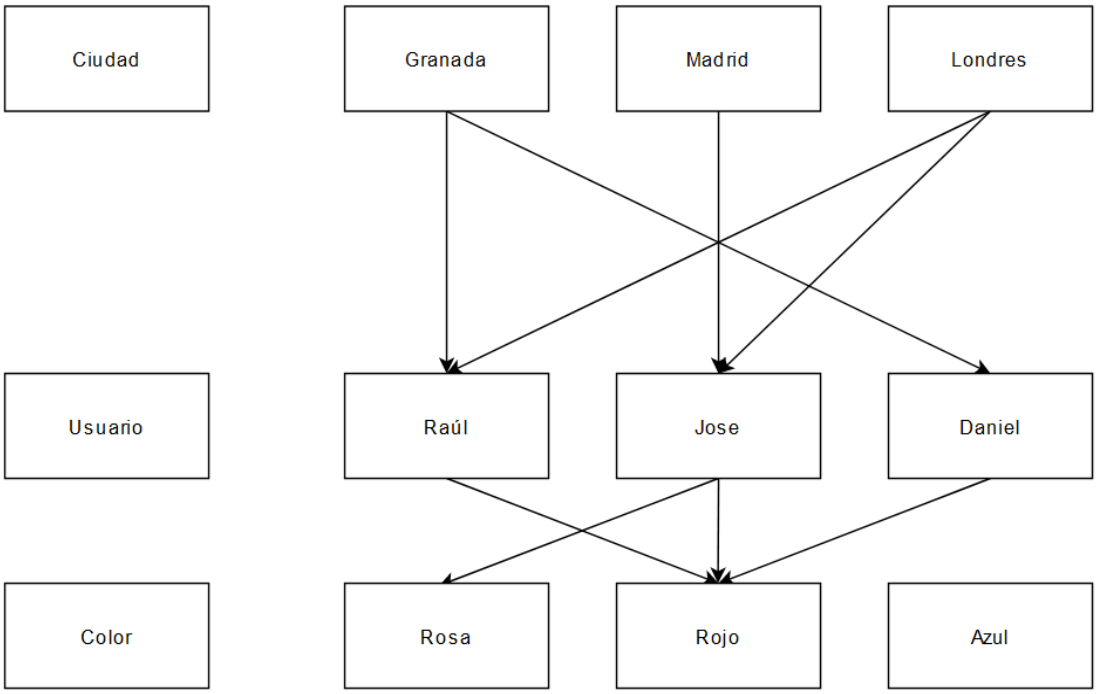

Figura 2: Estructura de base de datos en red.

En la figura superior se puede observar la estructura de este tipo de bases de datos, cada nodo padre puede tener varios hijos y cada nodo hijo tener varios padres, en el ejemplo, el registro “Granad” referencia al usuario “Raúl” y al

usuario “Daniel”, mientras que el usuario “Raúl” es a la vez referenciado por la ciudad “Londres”.

2.2.4. Base de datos relacionales

Las bases de datos relacionales se basan en el modelo relacional, permitiendo establecer relaciones de datos entre sí de forma directa e intuitiva en forma de tabla, en dicha tabla, se utilizan filas con un registro único (clave) y columnas que almacenan atributos de los datos [17].

DNI	Nombre	Ciudad	Edad
12345678H	Raul	Granada	12
98765432Q	Daniel	Jaen	15
12983476L	Clara	Granada	12
34985612F	Jonas	Cadiz	11

Figura 3: Estructura de base de datos relacional.

En la figura anterior se puede apreciar la estructura de este tipo de base de datos, en ella se tiene una columna DNI que almacena valores correspondientes a DNI, los cuales son únicos en dicha columna (no hay dos valores iguales), esta sería la clave primaria, mediante la cual se puede acceder al resto de datos de la fila correspondiente, si quisiéramos conocer el nombre de un usuario, podríamos acceder a la tabla mediante su DNI y obtener los valores del resto de columnas para la fila de dicho DNI.

2.2.5. Base de datos orientado a objetos (BDOO)

Las bases de datos orientadas a objetos se caracterizan ya que la información está representada mediante objetos, de la misma forma en la que se basa la programación basada en objetos (POO) [19].

Esto permite integrar las características de una BDOO con una POO, dando lugar a un sistema gestor de base de datos orientada a objetos (ODBMS), donde los objetos de la base de datos pueden ser gestionados con un objeto de lenguaje de programación, permitiendo dar soporte a más tipos de lenguaje, como C++ o Java.

Entre sus características se encuentran características obligatorias y opcionales:

Obligatorias	Opcionales
Soporte a objetos complejos	Herencia múltiple
Los objetos deben llevar identificador	Diseño de transacciones y versiones
Encapsulación	Inferencia de distribución
Concurrencia	
Recuperación	
Persistencia	
Facilidad de query	

Las ventajas de este modelo de base de datos es que permite una gran adaptabilidad a lenguajes de programación orientados a objetos, la manipulación de datos complejos es rápida, no hace falta indicar un código de identificación a cada objeto ya que se asigna automáticamente, además cuentan con un mecanismo de caché para crear réplicas parciales, mejorando el rendimiento y son capaces de manejar grandes volúmenes de datos.

Por otro lado, no existe una gran documentación actualmente y su uso está poco extendido hoy en día.

2.2.6. Base de datos en memoria

Existe otro tipo de base de datos que se almacena en la memoria principal del ordenador.

Este tipo de base de datos permite analizar los datos más rápido que las bases de datos tradicionales, cuyo almacenamiento es en disco, esto se debe a que la CPU tiene acceso de escritura y lectura sobre la memoria principal [20]. Un ejemplo puede verse en la figura siguiente:

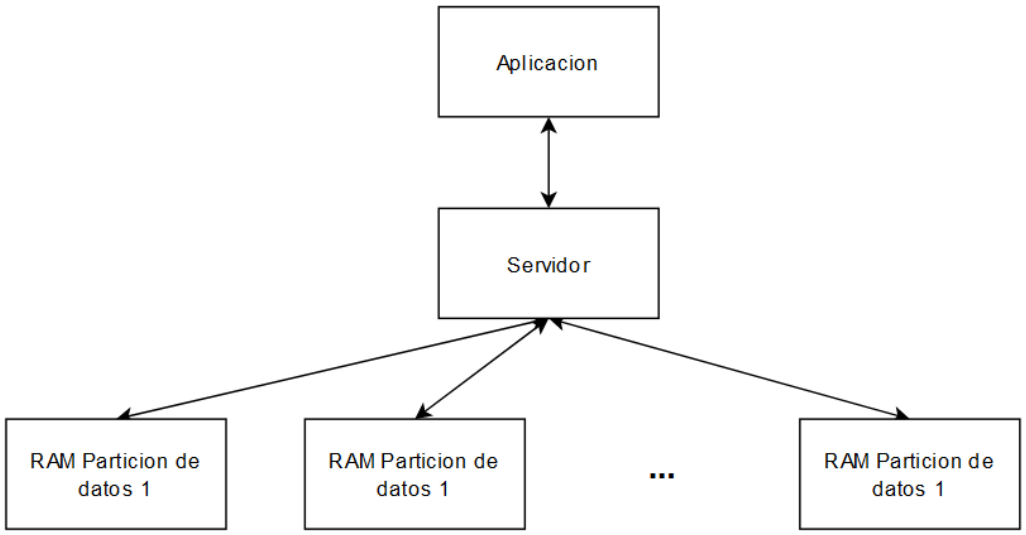

Figura 4: Estructura de base de datos en memoria.

Con la aparición del internet de las cosas (IoT) y el mayor uso de soluciones basadas en la nube, es necesario procesar datos en tiempo real. Esto se consigue mediante estas bases de datos.

Uno de los problemas que presenta este tipo de base de datos es la volatilidad de la información, al no estar almacenada en disco, dicha información no es persistente y en caso de fallo de sistema se producirá la pérdida de información. Una de las soluciones que se utilizan para este problema, es crear copias de seguridad periódicamente en disco, de forma que se minimiza la pérdida de datos.

Otro problema importante sobre este modelo de base de datos, es la cantidad de memoria usada, al almacenarse todo en memoria principal, es fácil que la memoria se llene provocando pérdida de información y problemas en el propio sistema, lo que requiere la implementación de procesos para evitar el colapso del sistema.

2.3. Cloud computing

2.3.1. Historia

El término cloud computing (computación en la nube) es el término utilizado para referirse a que los programas se ejecutan en un servidor remoto en Internet. En 1960, John McCarthy propuso que un día la informática sería realizada por “empresas de servicios públicos a nivel nacional”, dando nacimiento al término de cloud computing.

En 1999, la empresa Salesforce se convirtió en la primera compañía que ofrece un servicio a través de internet, consistiendo en la descarga de aplicaciones.

Más adelante, empresas como Google se empezaron a interesar en el cloud computing, proporcionando herramientas como Google Docs, pero no sería hasta 2006 cuando apareció la primera cloud moderna a manos de Amazon, bajo el nombre de Amazon Web Services (AWS), proporcionando una amplia gama de servicios, como potencia de cálculo y almacenamiento.

A raíz de esto otras empresas como Microsoft, Google, Apple e IBM empezaron a aparecer en el mercado de la computación [21].

2.3.2. Características

Respecto a cloud computing, se pueden definir las siguientes características:

- Posibilidad de autoservicio sin intervención humana por parte del proveedor.
- Disponibilidad de acceso de banda ancha a la web.
- Concentración de recursos en sitios concretos
- Escalabilidad: los recursos se pueden asignar y liberar de forma eficiente y rápida.
- Servicio gestionable: el servicio de gestión de la nube controla los recursos y los optimiza automáticamente.

2.3.3. Tipos de Cloud Computing

Al hablar de cloud computing podemos encontrarnos con determinada terminología para hacer referencia a ciertas situaciones, que han ido surgiendo unos de otros con el paso del tiempo, algunos modelos de implementación son los siguientes [22]:

- Nube pública
    - Es la más común, todos los recursos son propiedad de un proveedor que presta sus servicios a los usuarios y se encarga de su gestión.
    - Generalmente los servicios se ubican en instalaciones dedicadas de computación repartidas por regiones.
    - Algunos ejemplos de nube pública son: AWS, Azure, Google Cloud Platform.

- Nube privada
    - Debido a requisitos de seguridad, es posible que ciertas compañías no quieran usar los servicios de la nube pública, en este tipo de casos, la empresa crea la infraestructura necesaria en sus propios centros de datos, es decir, los servidores físicos que proporcionan estos recursos se encuentran en la misma instalación física que la empresa.
    - Proporciona un mayor control sobre los recursos, pero a cambio de un coste mayor debido al coste de obtener la infraestructura, gestionarla y mantenerla.
    - Algunas herramientas Open Source para replicar este modelo de cloud computing son: OpenStack, OpenNebula y CloudStack.

- Nube híbrida
    - Este tipo de nube es una mezcla de la nube pública y de la nube privada, de forma que se pueden combinar servicios de diferentes proveedores.

- Multicloud
    - Se basa en la combinación de dos o más implementaciones del mismo tipo, resultando en las combinaciones de varios proveedores simultáneamente.
    - Por ejemplo, si se está utilizando una nube pública que no ofrece un servicio requerido por un cliente, se puede buscar otro proveedor que sí ofrezca dicha funcionalidad y adaptar solamente lo necesario para su funcionamiento, de forma que se ajuste a las necesidades.
    - Otro ejemplo sería la disponibilidad, un proveedor proporciona acceso a regiones africanas y no europeas, la solución consistiría en buscar un proveedor que sí proporcione acceso a dichas regiones además de mantener el proveedor de las regiones africanas.

Aparte de dichos modelos de implementación de cloud computing, es posible encontrarse con modelos de servicio que permiten elegir el nivel de control, flexibilidad y administración de la información [23].

- IaaS (Infrastructure-as-a-Service)
    - Utilizado principalmente por administradores de sistemas.
    - Proporciona recursos fundamentales, redes, servidores, almacenamiento y firewalls.
    - El cliente tiene mayor control de la información, puesto que puede implementar y ejecutar software según sus preferencias.
    - No se controla la infraestructura subyacente, solamente la capa de virtualización.
    - Ejemplo: AWS, Microsoft Azure, Google Cloud.

- PaaS (Platform-as-a-Service)
    - Utilizado principalmente por desarrolladores de software.
    - Implica un nivel de abstracción más elevado del IaaS.
    - Se proporciona acceso a entornos cloud en los que se pueden crear, desarrollar, gestionar y distribuir aplicaciones.
    - Ejemplo: Google App Engine, Heroku, AWS Lambda.

- SaaS (Software-as-a-Service)
    - Solo se proporciona software o aplicaciones en la nube a través de internet.
    - No hay una infraestructura física.
    - Ejemplo: Gmail, Office 365, Dropbox.

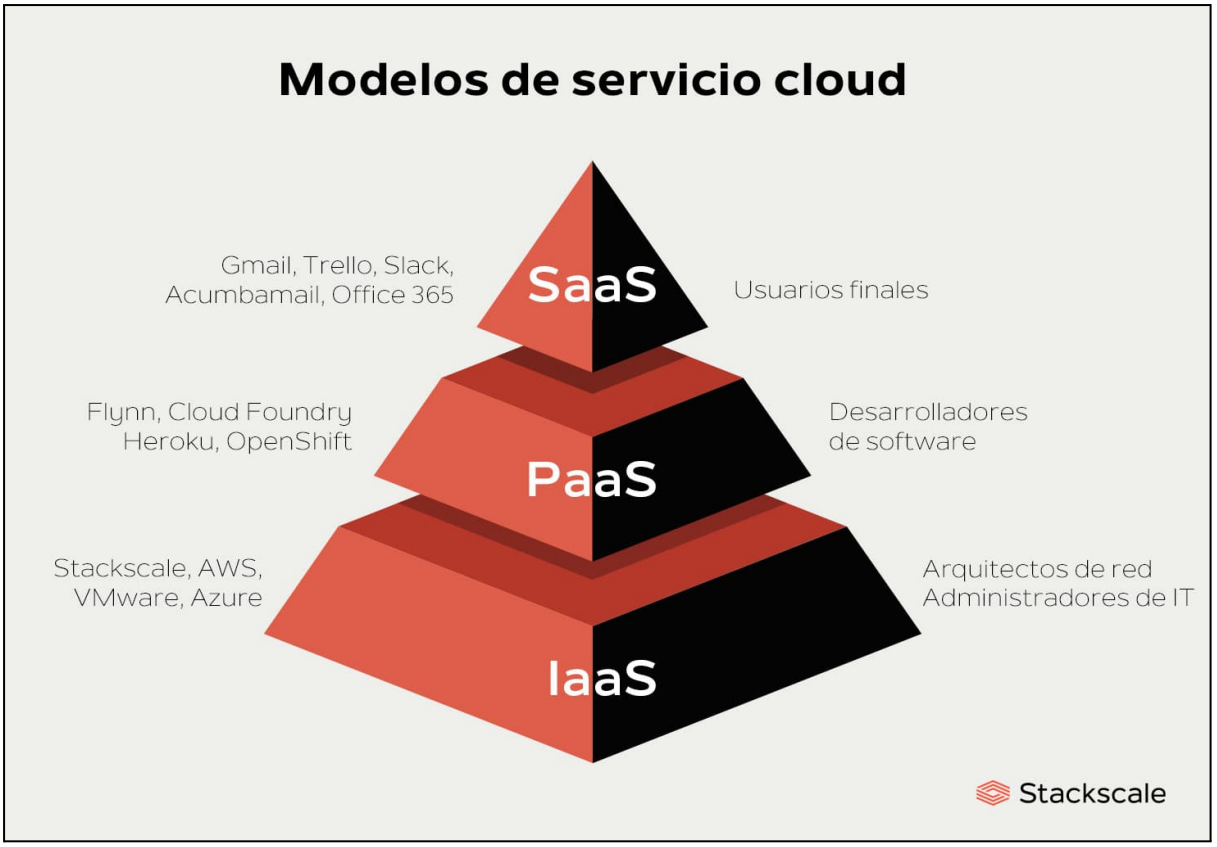

Figura 5: Clasificación entre IaaS, PaaS y SaaS [58].

La figura superior obtenida de la web referenciada en la referencia 58 de la bibliografía, representa claramente la relación entre estas tres tecnologías, donde IaaS es la base de todo, proporcionando la arquitectura de red, dentro de esta arquitectura se puede encontrar los PaaS y finalmente los SaaS.

Capítulo 3. Desarrollo

En este capítulo se explica el desarrollo seguido en la realización del proyecto, empezando con una introducción sobre la estructura general de la aplicación, las herramientas usadas y cómo se han usado, una explicación del despliegue realizado y los tests diseñados.

3.1. Introducción

Para empezar a desarrollar el proyecto, es necesaria la elección de un framework que permita la construcción de un chatbot, para esto, se ha elegido Rasa, un framework Open Source que será explicado en profundidad más adelante, esta herramienta permite construir un asistente conversacional compuesto en dos partes principales, un servidor principal, encargado de recibir peticiones y gestionarlas a través de un motor de análisis basado en un modelo construido previamente y un servidor de acciones basado en clases Python cuyo objetivo es gestionar determinados aspectos del asistente como pueden ser acciones personalizadas (se quiere hacer una acción que no es del ámbito directo del bot, como consulta de un horario específico), o acciones de validación de componentes del propio bot (como los slots, donde se almacena la información extraída).

Para la comunicación con el asistente se ha decidido utilizar Telegram, de forma que desde Telegram, se envían peticiones al servidor de Rasa, esta las gestiona y devuelve la respuesta a Telegram.

Para conseguir una disponibilidad 24x7, se ha utilizado Google Cloud Engine donde cada instancia almacena dos contenedores Docker, uno del servidor de Rasa y otro de Rasa Actions.

Puesto que es necesario proveer de servicio según el número de usuarios, se ha establecido un proceso automático de despliegue de servidores desde un balanceador de carga (Nginx), de forma que, si el número de usuarios es superior al número total que pueden soportar los servidores desplegados, una nueva instancia se despliega.

Por el contrario, si dicho número de usuarios totales es inferior a cierta cantidad, se destruirán tantas instancias como sean necesarias para mantener siempre el número necesario de servidores sin desperdiciar recursos.

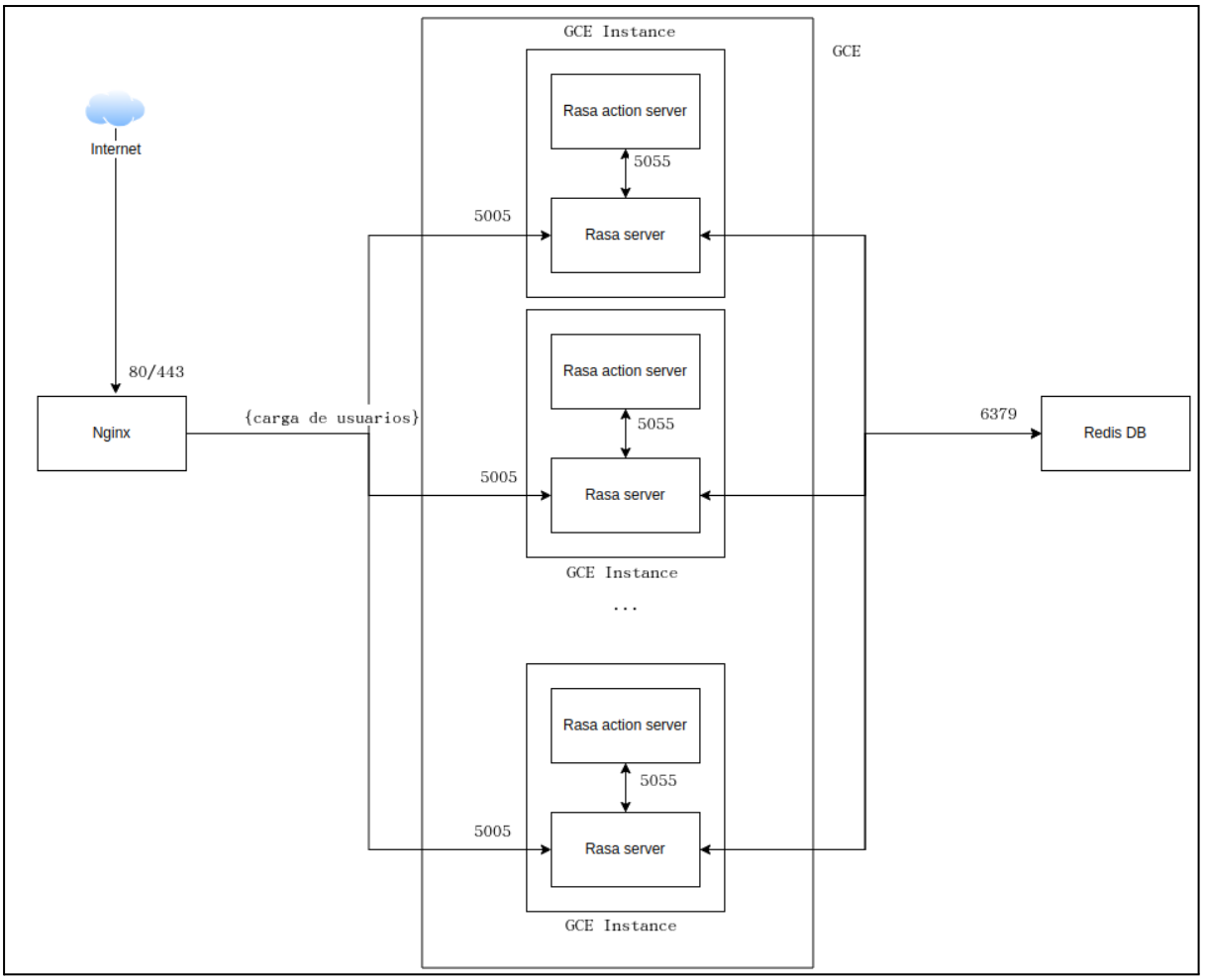

Figura 6: Relación y conexión entre servidores del proyecto.

En la figura anterior, se puede ver el despliegue de forma gráfica, en él se puede observar que desde la nube (Internet) se realizan conexiones a los puertos 80 y 443 de Nginx, el cual balancea la carga entre diferentes instancias de Google Compute Engine al puerto 5005, donde está escuchando el servidor de Rasa. Este servidor realiza acciones en función de la petición que ha llegado, haciendo conexiones a la base de datos de Redis en el puerto 6379 y a su Action server particular a través del puerto 5055.

Para el almacenamiento de los datos del asistente conversacional, como puede ser mantener el historial de mensajes o mantener un estado de la conversación de un usuario, de forma que cualquier servidor pueda acceder a él, se ha utilizado una base de datos basada en memoria (Redis) desplegada en un contenedor Docker.

Puesto que para el despliegue es necesario disponer del código, se ha utilizado la plataforma GitHub, de forma que se puede llevar un control de versiones y disponer siempre del último código actualizado, de esta forma cada despliegue descarga dicho repositorio y configura el servidor con sus propios parámetros.

A continuación, voy a dar una explicación de cada herramienta usada, el motivo de su elección y su modo de uso.

3.2. Herramientas

3.2.1. Rasa Server

Como ya se ha explicado anteriormente, Rasa es un framework de desarrollo de asistentes conversacionales de código abierto, el cual dispone de diferentes planes de desarrollo de forma gratuita, al contrario que otras herramientas que establecen límites en la funcionalidad o incluso pagos.

Entre estos planes se puede encontrar diferentes opciones como puede ser Rasa Enterprise o Rasa Open Source.

Rasa Enterprise, antes conocida como Rasa X, es una solución empresarial más completa que Rasa Open Source y que dispone de más funcionalidades que su versión gratuita, una de las funcionalidades más llamativas es la interfaz que proporciona, la cual permite construir y entrenar al bot en tiempo real.

En la siguiente figura se puede ver un ejemplo de la interfaz de Rasa Enterprise, se dispone de un espacio de conversación, como el usado en una aplicación de mensajería, donde se introducen mensajes y se recibe un mensaje del bot.

También se puede observar la historia actual que se ha escrito, permitiendo guardarla e incluso información sobre un “slot” usado, mostrando su nombre y valor almacenado.

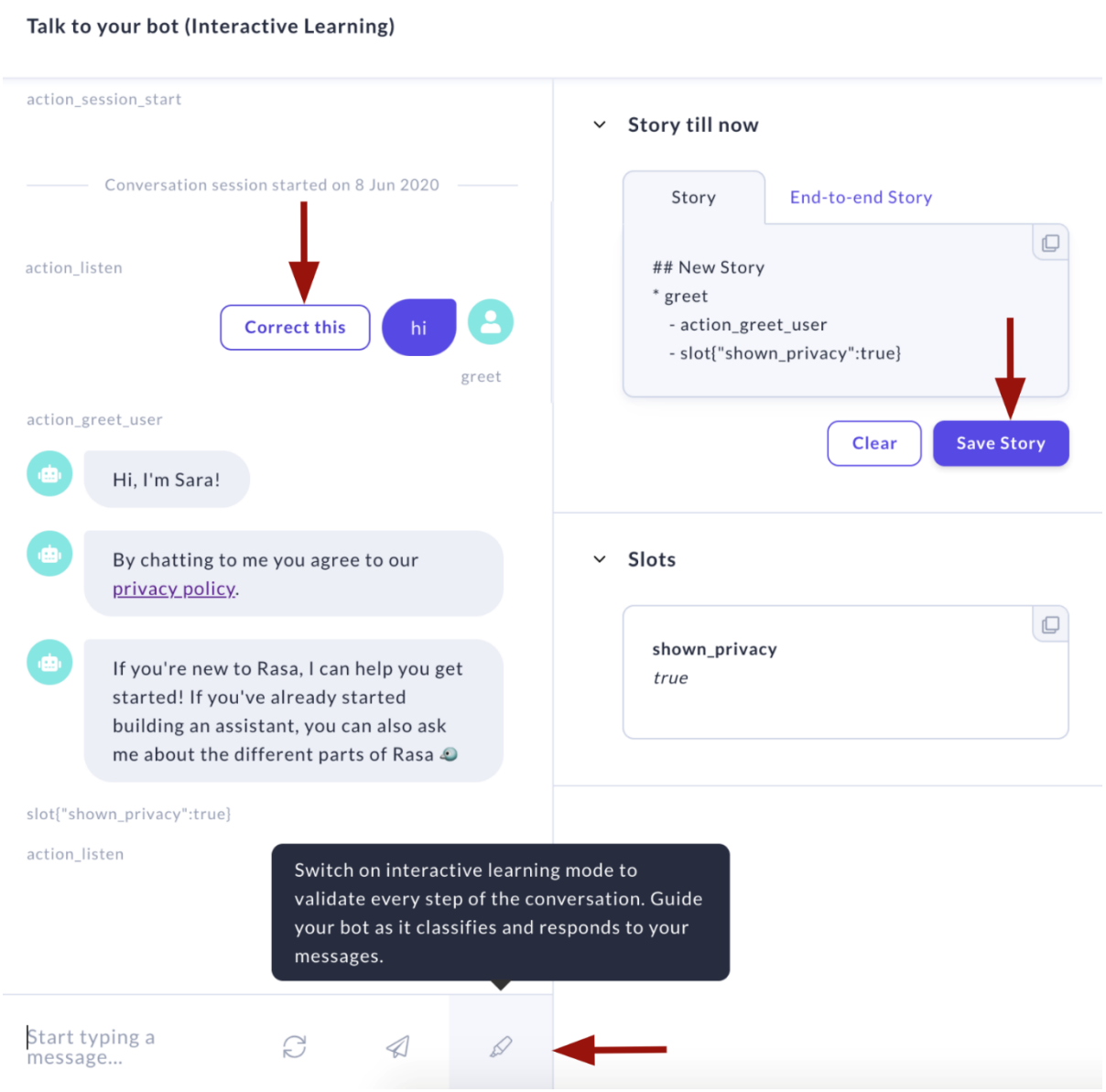

Figura 7: Ejemplo de interfaz de Rasa Enterprise [59].

La otra herramienta que se proporciona es Rasa Open Source, la cual es de uso gratuito en su totalidad, disponiendo del código fuente en GitHub, esta herramienta no cuenta con una interfaz como Rasa Enterprise, por lo que su desarrollo debe realizarse a través de una terminal de usuario usando la línea de comandos de Rasa.

Por la naturaleza del proyecto y mi interés en el código Open Source, ha sido esta solución la elegida para el desarrollo.

Rasa dispone de una amplia comunidad distribuida entre sus repositorios de GitHub, su foro público, YouTube, eventos, etc.
Al analizar Rasa, se pueden encontrar los dos componentes más importantes, Rasa core y Rasa NLU:

- Rasa core

Se encarga de la gestión de diálogos utilizando modelos creados mediante machine learning, conversaciones previas y datos almacenados hasta el momento actual de la conversación.

El flujo de trabajo de rasa core sería el siguiente:

1. El usuario envía un mensaje, se crea un tracker para dicha conversación, el cual almacena un historial.
2. Se clasifican los intents, reconoce y clasifica el mensaje.
3. Realiza las predicciones según los datos, estas predicciones se clasifican según un valor establecido en función del contexto.
4. Se ejecutan las predicciones realizadas y se almacenan en un tracker [24].

Un “tracker store” es un mecanismo interno que utiliza Rasa para almacenar las conversaciones que mantiene el bot en tiempo real, de forma que con cada interacción del usuario y acción del bot, se mantiene un historial de lo sucedido.

Este mecanismo va acompañado de otro mecanismo llamado “lock store”, el cual se encarga de manejar el acceso a las conversaciones almacenadas de forma que asegura que las conversaciones de un usuario no se mezclan con las de otro usuario y se procesan en un orden correcto.

No ha sido necesario implementar estos dos mecanismos puesto que Rasa los proporciona, aunque se permite crear un mecanismo personalizado ya que van asociados a la base de datos usada.

En el caso de este desarrollo, la base de datos Redis permite separar dichos mecanismos en diferentes bases de datos en el fichero credentials.yml como se verá más adelante.

- Rasa NLU

Se encarga del procesamiento de los mensajes y da una respuesta adecuada al usuario.

Una vez que se recibe el mensaje del usuario, se analiza la información y se cataloga en intents, que contienen ejemplos de mensaje del usuario, extrayendo información de ellos, por ejemplo, si el mensaje informa de una característica como un color, podremos almacenar el valor del color en un slot para recordarlo.

Esta clasificación de intents se realiza asignando un valor porcentual a los intents establecidos en el bot, de forma que se elige el intent con mayor porcentaje.

Es posible que el bot no sea capaz de reconocer un intent con un alto porcentaje, por lo cual, si el porcentaje del intent elegido es inferior a un valor configurado en el fichero config.yml, un intent interno llamado “default_fallback”, será elegido, el cual tiene como función realizar un paso atrás en la conversación devolviéndole a un estado correcto.

Si este mecanismo no se controla correctamente o no se tiene en cuenta, puede darse el problema de pérdida de datos, por ejemplo, si el usuario está terminando de establecer una reserva y antes de terminar se reacciona a este intent, ningún dato de la reserva introducido por el usuario será guardado, lo que, además, confundirá al usuario.

Una vez elegido un intent, entra en funcionamiento Rasa core, con la finalidad de elegir una acción a realizar. Esto lo realiza utilizando casos establecidos por el usuario en reglas e historias, las cuales indican un patrón de acción-respuesta, de forma que se establecen flujos de conversación en función del estado actual de bot.

La siguiente acción que se realiza es determinar la respuesta al usuario, la cual puede ser un mensaje predefinido, los cuales se denominan “utter_responses” o puede realizar una acción personalizada mediante el servidor de Rasa Actions, proporcionando una respuesta más enriquecida [25].

En la siguiente figura se puede observar un esquema general del funcionamiento de Rasa:

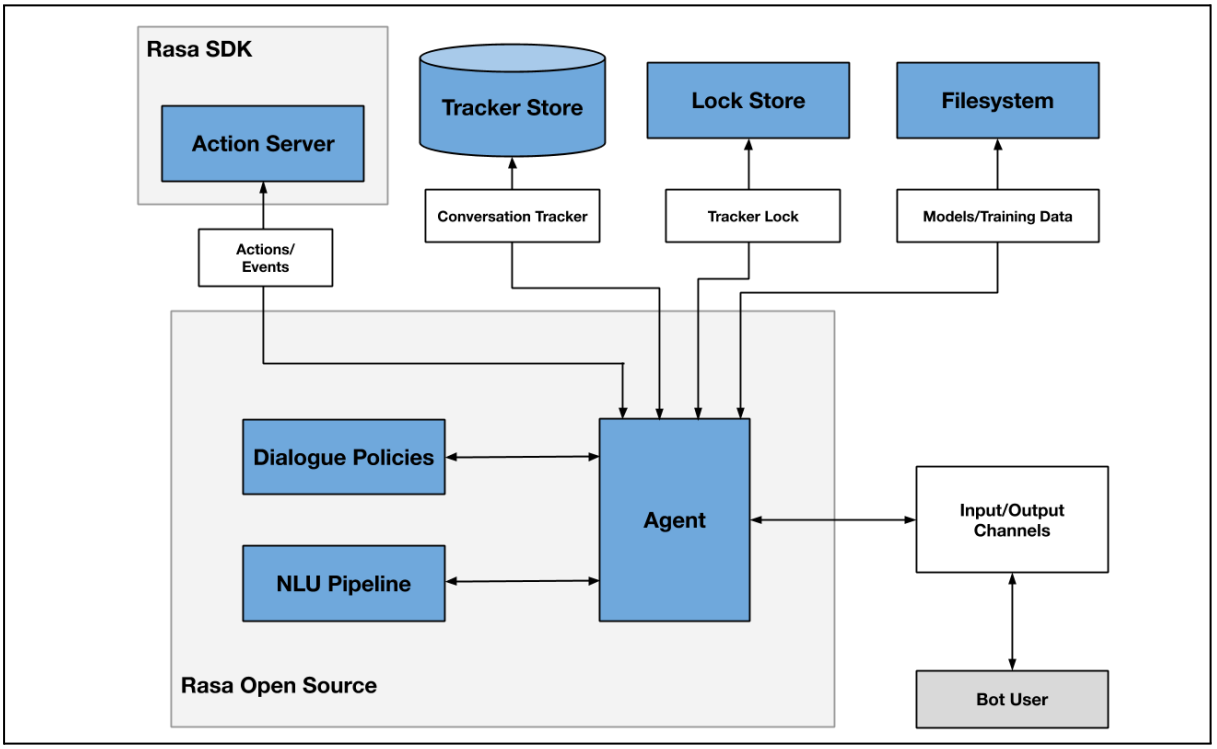

Figura 8: Flujo descriptivo de Rasa [26].

En esta figura se puede observar lo siguiente:

1. Se realiza una comunicación entre el agente de Rasa y el bot.
2. El agente de Rasa accede a diferentes módulos internos
   a. Accede a ficheros del sistema para consultar el modelo entrenado donde se especifican las historias a seguir.
   b. Accede a los módulos de Tracker Store y Lock Store para guardar la conversación actual de forma que no pierda el progreso del usuario.
   c. Se comunica con el servidor de Rasa Action Server para gestionar acciones personalizadas y eventos.
   d. Accede a la NLU para identificar la respuesta del usuario en función de las políticas de diálogo establecidas.

En esta explicación previa se han hecho referencia a ciertos objetos como intents, historias, reglas, responses, slots…, los cuales voy a explicar a continuación.

- Entities

Declaran todas las entidades que pueden ser extraídas de los mensajes de los usuarios [30].

Algunos ejemplos de “entities” usados en el desarrollo son: “first_name”, “menu_hay_plato”, “menu_establecido”, “reserva_dia”.

- Slots [29]

Se corresponden con la memoria del bot, almacenan valores extraídos de los intents en formato key-value, de forma que pueden ser accedidos en cualquier momento de la conversación.

Existen diferentes tipos de slots:

1. Texto: Almacenan strings.
2. Bool: Almacenan True o False.
3. Categorical: Obtiene un valor previamente establecido en una lista, esto es útil si un slot puede albergar solamente ciertos valores inmutables.
4. Float: Almacena números reales.
5. List: Almacena una lista de valores.
6. Any: Almacena listas o diccionarios.
7. Custom: Permite establecer un objeto personalizado mediante la creación de una Clase en Python, de esta forma se puede personalizar el tipo de valor a almacenar

Una de las opciones más importantes que se pueden utilizar con los slots es el parámetro “influence_conversation”, el cual permite al bot ser influenciado a la hora de realizar predicciones, por ejemplo, si se establece a False y se ha establecido un nombre de usuario, el bot puede reaccionar saludando al usuario, en cambio, si se establece a True, es posible que la acción que realice el bot no sea la de saludar, sino otra diferente.

- Intents

Relacionan ejemplos de mensajes de usuario, permiten establecer valores en los slots definidos de forma que es posible extraer información del usuario [31].

A la hora de extraer información del usuario, es posible que nos encontremos con el siguiente problema, con el cual me he tenido que enfrentar al realizar el reconocimiento del nombre del usuario:

- Se tiene un intent llamado “repetir_nombre”
- Se quiere extraer el nombre de la frase introducida por el usuario.

Se establecen 5 mensajes de ejemplo asociados a este intent, los cuales extraen el valor encapsulado entre corchetes y lo asignan al slot “first_name”:

- Me llamo [aslan](first_name)
- Me llamo [raul](first_name)
- Me llamo [nimra](first_name)
- Me llamo [roxan](first_name)
- Me llamo [tayana](first_name)

Si el usuario introduce un nombre que hace match con uno de estos mensajes, el valor se asigna directamente, pero si el usuario indica otro valor, por ejemplo, introduce el siguiente mensaje:

- Me llamo Elisabeth

El bot no sabrá que tiene que extraer el valor “Elisabeth” de dicha frase, puesto que dicho nombre no concuerda con el tipo de valor que se busca, ya que los valores establecidos entre corchetes actual como valores ejemplos de los que se analizan combinaciones, pero al ser este nombre más largo que los ejemplos, no se conseguirá un match.

Este problema ha estado presente durante el desarrollo del reconocimiento de nombres de usuario, donde la solución ha pasado por utilizar un diccionario predefinido de nombres válidos y una lista de ejemplos más larga, combinando nombres cortos y largos.

Se puede consultar más información sobre este problema en los comentarios de la siguiente issue: https://github.com/rauldpm/ChatbotTFG/issues/4

Otro de los problemas que he encontrado ha sido el cómo diferenciar la extracción de valores de distintos intents, cuando el posible valor de dichos intents tiene las mismas características, por ejemplo, el reconocimiento de números.

La primera opción, como ya se ha comentado antes, es especificar un número de ejemplos lo suficientemente alto para que el valor no pueda ser confundido, la otra opción es utilizar regex.

Rasa permite añadir patrones de expresiones regulares asociados a un slot determinado, de forma que cuando se hace match con un intent, el valor extraído se valida primero con los regex disponibles, asociándose al slot que corresponda.

- Responses

Son acciones que envían un mensaje al usuario sin ejecutar un código personalizado. Se definen directamente en el fichero domain.yml y permite especificar una respuesta, pudiendo enriquecerla con condiciones, botones, imágenes y otros parámetros.

Un ejemplo sería, por ejemplo: “utter_saludo”, este response podría estar condicionado a la identificación del usuario, de forma que, si el usuario es conocido, el mensaje enviado contiene el nombre del usuario y si no lo conoce, simplemente responde con un saludo.

También permiten establecer varias respuestas de forma que una de ellas es respondida al azar [32].

- Forms

Son un tipo especial de acción diseñado para obtener información del usuario, se definen asociando un slot, de forma que cuando se activan en una historia, automáticamente se pide una entrada de usuario de la que extraer el valor para ese slot y pasar un proceso de extracción y validación.

Este proceso de extracción y validación se realiza mediante clases establecidas en un fichero.

Dichas clases deben contener tres métodos para realizar este proceso:

- required_slots: Se encarga de añadir al dominio el slot que se pide
- extract_{slot_name}: Debe encargarse de extraer el slot requerido y prepararlo para la validación.
- validate_{slot_name}: Debe encargarse de validar el slot extraído.

Estos dos métodos se ejecutan automáticamente si “slot_name” corresponde con un slot establecido, ejecutándose primero la extracción y luego la validación.

Una vez se ha extraído la información validada o se para el proceso, es necesario desactivar el form, en el caso de que la extracción y validación haya sido correcta, el form se desactiva automáticamente, en caso de que no sea así, es necesario especificar explícitamente que debe parar [33].

- Actions

Se declara en el fichero domain.yml y define una lista de posibles acciones a realizar para responder al usuario, estas pueden ser respuestas predefinidas (responses), acciones custom (clases personalizadas en python) y formularios [34].

- Rules (rules.yml)

Las reglas se utilizan para manejar situaciones donde la conversación no es prolongada, generalmente se usan para dar casos de pregunta-respuesta (Intent-Action), como pueden ser [35]:

Usuario	Respuesta
¿Qué hora es?	Son las 15:00 horas
¿Cuál es el menú?	El menú del día es: … (menú)
¿A qué hora abrís el jueves?	El jueves abrimos a las 08:00 horas

En este trabajo se han utilizado para realizar acciones como empezar la conversación con el usuario, mostrar las opciones disponibles, saludar, proporcionar horarios, mostrar menús, mostrar reserva e identificar al usuario.

Esta última acción de identificar al usuario debería haberse realizado mediante una historia, pero se ha querido mostrar que es posible utilizar formularios junto a las reglas.

Esto implica tener una regla para activar el form y otra regla para terminar el formulario, mientras que, si se hubiera hecho con historias, se hubiera hecho de forma nativa.

- Stories (stories.yml)

Las historias, al contrario de las reglas, son utilizadas cuando la conversación se prolonga en el tiempo o es necesario especificar patrones de conversación, por ejemplo, si el usuario desea reservar una mesa, se contemplaría la siguiente historia [36]:

- Usuario: Quiero hacer una reserva
- Chatbot: ¿Cuándo quieres reservar?
- Usuario: El lunes
- Chatbot: ¿Ha pensado en una hora?
- Usuario: Si, a las 15:00 horas
- Chatbot: Muy bien, ¿cuántos comensales seréis?
- Usuario: Seremos 3 personas
- Chatbot: Vale, ese día tengo disponible una mesa, he guardado tu reserva

En esta conversación, se puede observar que hay varias preguntas por parte del chatbot y varias respuestas del usuario, se podría definir que hay secciones en ella:

- Intención de reserva
- Establecer hora
- Establecer comensales

Estas 3 secciones podrían especificarse mediante reglas de forma que cada vez que el usuario habla, hay que calcular la respuesta adecuada, pudiéndose dar el caso de un mal reconocimiento del intent.

Las historias ayudan a que el bot sepa que existe un camino a seguir y cómo seguirlo.

Un problema presente en las historias es la divergencia, por ejemplo, teniendo de ejemplo la historia anterior, podría ser que el usuario en vez de responder “Si, a las 15:00 horas” dijera: “No, no lo he hecho”, por lo que, en ese caso, el bot en vez de

preguntar por el número de comensales mostraría o propondría una hora para la reserva.

Story 1	Story 2
Usuario: Quiero hacer una reserva	Usuario: Quiero hacer una reserva
Chatbot: ¿Cuándo quieres reservar?	Chatbot: ¿Cuándo quieres reservar?
Usuario: El lunes	Usuario: El lunes
Chatbot: ¿Ha pensado en una hora?	Chatbot: ¿Ha pensado en una hora?
Usuario: Si, a las 15:00 horas	Usuario: No, no lo he hecho
Chatbot: Muy bien, ¿cuántos comensales seréis?	Chatbot: Vale, el lunes tenemos estas mesas disponibles estas horas
Usuario: Seremos 3 personas	…
Chatbot: Vale, ese día tengo disponible una mesa, he guardado tu reserva	

Esto es lo que se conoce como “Happy path” y “Unhappy path”, el primero determina historias que acaban según lo planeado, mientras que las segundas son derivaciones de la historia “happy”, donde no se llega a terminar la historia, ya sea por error o por intención del usuario.

Si en el caso de la tabla, las historias hubieran sido rules, es posible que tras el input del usuario “No, no lo he hecho”, la respuesta hubiera sido una no deseada, como, por ejemplo: “Vaya, lo siento”, en el caso de que se tenga una regla creada para responder a intents negativos.

- Conectores

Usados para conectar aplicaciones externas al chatbot de forma que se establece un canal de entrada y salida por el cual poder comunicarse [37].

Rasa dispone de una serie de conectores preestablecidos, los cuales necesitan solamente la información de conexión. También permite desarrollar un conector propio usando Python de forma que podría conectarse a endpoints customizados.

Para poder conectarse a estos endpoints, hace falta tener unas credenciales de acceso, como Tokens, y un webhook del que recibir y al que enviar los mensajes.

Algunos de los conectores que proporciona por defecto son:

- Facebook Messenger
- Slack
- Telegram
- Twillo
- Google Hangouts Chat

En este trabajo se ha decidido usar Telegram al ser una plataforma open source, de gran uso y reconocimiento, lo que consigue que la gran mayoría de los usuarios tengan acceso al mismo.

El proceso de esta integración comienza usando el bot “Bot Father” de Telegram, el cual gestiona los bots de Telegram creados por los usuarios. El proceso es muy sencillo y solo requiere que el usuario introduzca un nombre de usuario para el bot, mediante el cual los usuarios van a poder encontrarlo en Telegram.

Una vez creado se proporciona un Token con el cual se obtendrá acceso al mismo (recepción y envío de mensajes).

Una vez se tiene el Token y el nombre del bot, se debe acudir al fichero credentials.yml para establecer los datos de conexión.

```
telegram:
access_token: ${TELEGRAM_TOKEN}
verify: ${TELEGRAM_BOT_NAME}
webhook_url: ${TELEGRAM_WEBHOOK}
```

Figura 9: Ejemplo de configuración de credenciales [60].

De base, Rasa indica que los valores se establezcan en dicho fichero, lo que puede ser un problema de seguridad, en su lugar, se pueden establecer dichos valores en el entorno de la máquina y ser accedidos de la forma ${variable}.

Uno de los problemas encontrados ha sido la realización de peticiones usando https en vez de http. Por defecto, Rasa utiliza el protocolo http en sus conexiones, por lo que un desarrollo local no se va a ver afectado por esto al no necesitarlo.

En cambio, Telegram realiza las peticiones por el protocolo https, por lo que es imposible que el chatbot se comunique con el endpoint, para esto, se ha aprovechado el balanceador de carga Nginx usado para balancear la carga entre los distintos servidores, para que actúe como “reverse proxy” utilizando certificados autofirmados creados con “Letsencrypt”.

- Domain (domail.yml)

Este fichero de configuración contiene algunos de los apartados anteriormente explicados, en concreto, este fichero se encarga de definir la configuración de la sesión (como el tiempo de expiración), los intents que se definan en el fichero nlu.yml, las entidades que se pueden extraer de los mensajes, los slots que almacenan los valores de dichas entidades, los forms usados para extraer determinados valores del mensaje del usuario, las respuestas predefinidas que pueden darse al usuario y una lista de acciones que puede realizar el bot frente a una interacción del usuario [28].

- Endpoints (endpoints.yml)

Este fichero de configuración establece endpoints que el bot puede usar, como conexiones a bases de datos.

Por defecto, Rasa utiliza una base de datos SQL, la cual almacena los datos de la conversación, como el tracker y el lock store. Esta conexión es una conexión local exclusiva del servidor de Rasa que gestiona internamente [38].

También permite configurar otras bases de datos como MongoDB o Redis, que se explicaran en la sección 3.2.3.

3.2.2. Rasa Action Server

Este servidor proporciona un mecanismo para utilizar acciones personalizadas en el bot o procesos de validación de “slots” en “forms”, su funcionamiento general es simple: cuando el asistente predice una acción personalizada, el servidor de Rasa envía una petición POST al Action server mediante un payload en formato JSON [39].

Este payload en formato JSON incluye entre otros, el nombre de la acción predecida, el identificador de la conversación, el contenido del tracker y el contenido del dominio.

Cuando la acción personalizada ha terminado de realizarse, se retorna otro payload en formato JSON con las respuestas y los eventos resultantes hacia el servidor de Rasa, el cual responde al usuario y añade los eventos al tracker de la conversación. En este punto, el servidor de Rasa solo conoce lo que el servidor de acciones le devuelve.

Principalmente, Rasa Action server utiliza el lenguaje Python como Rasa SDK, pero permite la utilización de muchos otros lenguajes siempre y cuando dispongan de las librerías necesarias.

Para ejecutar Rasa Action server, se puede utilizar el comando:

- rasa run actions

3.2.3. Base de datos de Rasa

Rasa permite utilizar determinadas bases de datos de forma nativa sin necesidad de configurarla, requiriendo solamente la configuración de las credenciales y del propio tipo de base de datos.

3.2.1.1. SQL

Esta base de datos relacional está configurada por defecto para ser usada localmente junto al servidor de Rasa, aunque también permite configurar ciertos parámetros para establecer una conexión con una base de datos ya existente.

3.2.1.2. Mongo

Otra de las opciones que proporciona Rasa para conectar una base de datos es MongoDB, una base de datos NoSQL y orientada a objetos, que almacena la información en estructuras de datos BSON con un esquema dinámico.

3.2.1.3. Redis

Debido a la necesidad de que los servidores deben acceder a la información lo más rápido posible, se ha estudiado el uso de la base de datos Redis debido a sus características.

Redis almacena la información en memoria principal, realizando copias de seguridad periódicas en disco, esto produce que el acceso a los datos sea eficiente y rápido [40].

Al desarrollar el despliegue del proyecto, se ha encontrado el problema de que las bases de datos anteriormente mencionadas, establecen inicialmente la base de datos de forma local, teniendo acceso a dicha base de datos solo el servidor que la contiene.

Al balancear el acceso a los servidores, aparece el problema de que un usuario tenga diferentes comportamientos según el servidor al que se conecte.

Si un usuario se identifica con el servidor A, este almacena el nombre del usuario en la base de datos del servidor A, en la siguiente iteración, al volver a interactuar el usuario, es posible que sea otro servidor el que responda, el cual no conoce al usuario.

Para solucionar esto, cada servidor se configura para que se conecte a una base de datos externa almacenada en otra instancia, de esta forma, la base de datos gestiona el acceso al mismo y todos los servidores conocen la información que necesiten, ya que si un usuario con ID 0001 se identifica en el servidor A, la base de datos solo sabrá que el tracker 0001 tendrá un valor de identificación, cuando otro servidor atienda a dicho usuario, podrá obtener el valor de dicho usuario usando su mismo ID.

Puesto que es primordial poder acceder a la información rápidamente, me he decantado por esta base de datos.

Para configurar esta base de datos solo es necesario indicar los siguientes datos en el fichero “endpoints.yml”, tanto en el “tracker store” como en el “lock store”, aunque permite que ambos módulos estén separados en diferentes bases de datos:

- url: Debe indicar la dirección IP donde esté ejecutándose la base de datos.
- port: Debe indicar el puerto en el que se sirve el servicio de la base de datos.
- db: Indica el identificador de la base de datos, en caso de usar la misma base de datos para el tracker como para el lock, deben indicar valores diferentes.
- password: La contraseña establecida al iniciar la base de datos, de esta forma se evita tener la base de datos desprotegida

Un comportamiento no deseado a la hora de usar esta base de datos en un despliegue en la nube ha sido descubrir mediante los logs del contenedor, accesos no deseados, ya que inicialmente, el parámetro “password” no estaba presente, lo que al realizar el despliegue y al no tener configurado un firewall inicial, permitía que cualquier persona pudiera conectarse a ella.

Esto ha sido solucionado añadiendo una contraseña a la base de datos y cerrando el acceso de la instancia que contiene la base de datos de forma que solo sea accesible desde la red interna de la Google Cloud.

Durante la investigación para arreglar este problema, se hizo uso del foro público de Rasa para consultar posibles alternativas, pero no se recibió contestación. Al final,

sen encontró la solución aquí descrita y se dejó constancia en el post creado para que fuera de ayuda a otros usuarios: https://forum.rasa.com/t/add-conversation-persistence-with-load-balancer-nginx-redis-found-possible-solution/54569/2

3.2.4. Python

Python es un lenguaje de programación de alto nivel interpretado, es el lenguaje de programación principal que utiliza Rasa a la hora de definir acciones personalizadas.

Se ha utilizado debido a su gran simplicidad, manejabilidad y utilidad, además de ser uno de los lenguajes más demandados profesionalmente.

3.2.5. Git y GitHub

En el desarrollo de cualquier proyecto es necesario mantener un control de versiones de forma que se pueda ver una historia en los cambios del proyecto, esto se puede conseguir mediante múltiples herramientas y la elegida para este por proyecto es Git [41].

Git almacena cada versión guardada como una instantánea identificada mediante una clave denominada commit. Esta herramienta permite realizar acciones como push y pull para actualizar cambios, ya sea hacia remoto (push) o local (pull), de forma que es posible trabajar de forma paralela entre varios desarrolladores.

Por otro lado, GitHub es una plataforma que mantiene repositorios de código en la nube, de forma que el proyecto sea accesible a diferentes desarrolladores y pueda ser consultado con facilidad en tiempo real.

Además, GitHub proporciona diferentes herramientas para gestionar el código, como la organización y gestión de proyectos, establecer roles, realizar procesos de automatización, etc.

A pesar de que GitHub se usa mayoritariamente con proyectos de código Open Source, él mismo no lo es, siendo de código cerrado.

La mayoría de las acciones que se realizan en GitHub se pueden realizar a través de Git, en este trabajo, Git se ha utilizado para controlar los cambios que se realizan en el código y subirlos al repositorio creado en GitHub.

Estas herramientas también permiten la gestión de ramas de forma que se pueden realizar desarrollos independientes para más adelante ser fusionados una vez que estos terminen.

En lo que se refiere a la planificación del proyecto, se ha seguido un versionado de ramas numeradas de forma que cada rama numerada corresponde con una versión (major.minor), de estas ramas enumeradas acarrean el último desarrollo de dicha versión (major.minor.release), por ejemplo, si se está desarrollando la versión 3.5.4, se trabajará sobre una rama obtenida de 3.5, cuando el desarrollo termine y la versión 3.5.4 esté lista, los cambios se mergear en la rama origen 3.5 y se creará un tag y una release..

Un tag es igual que una rama, pero inmutable, es una screenshot de un commit del repositorio, de forma que corresponde una versión final del producto en un momento determinado.

Una vez creado el tag, se genera una release publica a partir del código de dicho tag, donde se pueden consultar un changelog de la versión.

Una vez que ha sido publicada la nueva release, todas las ramas se actualizan en cascada de forma ascendente, por ejemplo, se tiene la rama “main”, la rama 4.0 y la rama 3.5, la cual contiene el último desarrollo publicado, en el momento de publicar la release, la rama 4.0 (desarrollo a futuro superior de 3.5) se actualiza con la rama 3.5, luego la rama “main”, que puede contener el desarrollo más a futuro se actualiza con la rama 4.0.

Este sería el caso de tener múltiples ramas de desarrollo con diferentes versiones, pero en el repositorio no se ha seguido este esquema, sino que como solamente se ha estado trabajando en una rama numerada a la vez, se usa la rama “main” como rama “stable”, la cual contiene los últimos cambios publicados.

Como ya se ha explicado en la planificación del desarrollo del **Capítulo 1. Introducción**, se han utilizado las herramientas de milestones, project e issues para gestionar el desarrollo constante del trabajo, de forma que las issues quedan clasificadas por temática y desarrolladas dentro de un periodo de tiempo o deadline.

Puesto que esto es un trabajo académico, se ha desarrollado en un repositorio privado para evitar que se plagie el trabajo, siendo este repositorio liberado (público) a día de entrega de la memoria.

Al hacer público el repositorio, se consigue además que el repositorio active el apartado de Discussions y Wiki, en el primero se puede discutir con los usuarios

ideas que estos tengan y en el segundo apartado se puede mantener una documentación del repositorio.

3.2.6. Docker

Otra de las herramientas usadas es Docker, una herramienta Open Source que permite la automatización de aplicaciones dentro de un contenedor software, es decir, permite el despliegue de una máquina virtual altamente personalizable [42].

Entre los principales usos que tiene Docker, podemos definir los siguientes casos:

- Abstraer servicios del sistema operativo.
- Escalar servicios de forma automatizada.
- Desarrollo ágil.
- Integración continua.
- Testing.

En este proyecto se utiliza Docker para múltiples tareas, algunas de las cuales han sido mencionadas, como el despliegue de servidores y recursos necesarios en la nube, creación de ficheros como el fichero de modelo generado con el entrenamiento de Rasa o incluso para realizar testing a la hora de incorporar cambios a una rama principal de GitHub.

La principal utilidad de Docker es la creación de imágenes a partir de ficheros llamados Dockerfile, el cual contiene una serie de órdenes donde se especifica el sistema operativo a usar y diversas órdenes como la copia de archivos y ejecución de comandos.

Para poder ser ejecutado, este archivo requiere de un punto de entrada, que puede ser un fichero en formato yml o una orden directa.

Así, podríamos decir que un Dockerfile tiene dos partes, la primera parte se encarga de especificar cómo se construye la imagen y la segunda parte se especifica mediante un fichero denominado “entrypoint”, el cual indica lo que hace la imagen al ser ejecutada.

Una vez se ha construido la imagen, es posible ejecutarla, de forma que se produce la creación de un volumen para persistencia de datos y de un contenedor asociado.

Un contenedor es una instancia obtenida de la imagen donde se ejecuta el servicio al que está destinado, este contenedor puede ser parado, reiniciado y modificado (aunque esto vaya en contra de la filosofía Docker) entre otras cosas.

El volumen es un punto de montaje fuera del contenedor que actúa como almacenamiento compartido, de esta forma se puede lograr una persistencia de datos cuando el contenedor se para [43].

Cuando se tienen múltiples servicios, se puede recurrir a la herramienta "docker-compose", la cual ejecuta ficheros yml en los que se especifican dichos servicios y sus características, por ejemplo, un fichero yml puede contener información de dos servicios, una base de datos y un servidor.

Con dicho fichero es posible enlazarlos de forma que primero se inicie la base de datos y luego el servidor, evitando que pueda existir un periodo de tiempo donde el servidor no tiene utilidad.

En este proyecto se ha utilizado la herramienta docker-compose para realizar el despliegue de los servidores, de la base de datos y del balanceador de carga de una forma automatizada.

Es posible que se quiera que determinadas imágenes estén almacenadas en la nube, de forma que sean accesibles públicamente o para procesos automatizados, esto se puede conseguir mediante determinadas plataformas como puede ser AWS ECR o Docker Hub [44].

En este caso se ha utilizado la plataforma Docker Hub para aprovechar la integración con GitHub, que al igual que este, se caracteriza por almacenar versiones de imágenes.

3.2.7. Nginx

Como ya se ha mencionado anteriormente, un chatbot tendrá una cantidad de usuarios que querrán acceder al mismo servicio ya sea en un momento dado o en momentos diferentes.

Si muchos usuarios intentan acceder al chatbot al mismo tiempo, es posible que el servidor se sature resultando en la pérdida de datos al no poder atender todas las peticiones.

Debido a esto se ha desarrollado un despliegue automatizado del servidor de Rasa en función de la cantidad de usuarios activos en un determinado momento, este proceso será explicado más adelante.

Para lograr este balanceo de carga entre los servidores y evitar la saturación del sistema, se ha recurrido a Nginx, un software Open Source que actúa como servidor web, proxy inverso, caché de HTTP y balanceador de carga [45].

Para especificar a qué servidores se debe balancear la carga, es necesario especificarlo en un fichero de configuración, que, por defecto, es el mismo fichero principal de Nginx, esto no es deseado ya que implica realizar un “hardcoding” de los servidores en la configuración y al escalar el sistema, dicho fichero de configuración debe ser modificado, lo cual no es deseable.

Nginx dispone de una versión de pago llamada Nginx+ que permite establecer servidores a los que balancear la carga de forma dinámica, pero al ser de pago, no se tiene acceso a esta funcionalidad.

Como alternativa, los servidores se almacenan en un fichero llamado “servers” que se incluye en la configuración mediante la directiva “include”, de esta forma, cuando se escale el sistema, solo habrá que modificar dicho fichero “servers” y reiniciar el servicio de Nginx.

Otro de los problemas ya mencionados anteriormente, es la comunicación segura (https) entre Telegram y el servidor de Rasa, el cual por defecto utiliza http. Este problema puede ser solucionado mediante la creación de certificados autofirmados y el uso de un “reverse proxy” en la configuración de Nginx.

Para la creación de certificados es necesario utilizar el software Open Source “LetsEncrypt”, el cual asocia los certificados a un DNS, el cual debe coincidir o con el servidor principal de Rasa en caso de que no se utilice Nginx, o con la instancia donde Nginx este desplegado.

Aquí se encontró el siguiente problema en las dos plataformas que se probaron para realizar el despliegue (AWS y Google Cloud):

- AWS proporciona un DNS para cada instancia levantada, pero el equipo de LetsEncrypt no considera estas DNS seguras y no permite la creación de certificados con ellas, más información en este post de la comunidad de LetsEncrypt.
- Google Compute Engine no proporciona DNS a las instancias levantadas (solo IP privada y pública), por lo que es necesario utilizar un servicio de GC para establecer los DNS (rechazado por costo) o utilizar un servicio como NOIP para enlazar la IP publica con un DNS dinámico.

La solución elegida fue utilizar el servicio de NOIP, el cual proporcionó la siguiente dirección: “tfgrasa.ddns.net”, la cual es usada al especificar el webhook de Telegram.

3.2.8. Cloud

En este apartado voy a listar y explicar los diferentes recursos contemplados para su uso en el proyecto.

3.2.8.1. Open Source platforms

Este proyecto se ha desarrollado con la idea de usar todas las herramientas Open Source posibles, en el caso de plataformas en la nube, se han contemplado las siguientes:

3.2.8.1.1. Heroku

Esta plataforma de servicios en la nube (PaaS) que permite realizar despliegues de contenedores Docker en su plataforma y gestionarlos de una forma rápida y eficaz.

Esta plataforma ofrece distintos medios de suscripción, y aunque ahora mismo dispone de un plan gratuito, este dejará de estar disponible próximamente en favor de un plan de estudiantes [46].

Los planes que ofrece esta plataforma son los siguientes [47]:

- Free: Sin coste, limitación de peticiones mensuales, 512MB de RAM, 2 núcleos de CPU
- Hobby: $7 al mes, 512MB de RAM y 10 núcleos CPU
- Standart: $25-$50 al mes, 512MB de RAM
- Performance: $250-$500 al mes, 1GB de RAM
- Private: No especifica el coste, 2.5GB de RAM
- Shield: No especifica el coste, 14GB de RAM

Los requerimientos recomendados de Rasa para su despliegue varían según el uso, pero se especifica que Rasa server debería usar 2CPU y 4GB de RAM mientras que Rasa Action 2CPU y 2GB de RAM.

Debido a esto, esta plataforma queda descartada para su uso.

3.2.8.1.2. OpenStack

OpenStack es otro PaaS Open Source utilizado para diseñar nubes privadas y públicas. Su funcionamiento consiste en la ejecución de una serie de comandos conocidos como scripts, que se agrupan en paquetes llamados proyectos los cuales transmiten tareas para generar los entornos en la nube.

OpenStack depende de sistemas de virtualización y un sistema operativo base que ejecuta los comandos provenientes de los scripts de OpenStack [48].

Esta herramienta no te proporciona directamente una nube, sino que se utiliza la virtualización para diseñarlas.

Una vez se ha establecido la nube se puede disponer de una serie de servicios con diferente finalidad, entre ellos están:

- Nova: Gestión y acceso a recursos.
- Neutron: conectividad de redes entre servicios OpenStack.
- Swift: Almacenamiento de objetos.
- Cinder: Almacenamiento permanente de bloques.
- Keystone: Autentica y autoriza los servicios de OpenStack.

Esta herramienta es muy potente, pero también es muy compleja y para el propósito de este trabajo no tiene sentido usarla, además, la nube que crea debe alojarse en una infraestructura que no se dispone.

3.2.8.1.3. Ngrok

Aunque Ngrok no es una plataforma en la nube, se ha utilizado para realizar pruebas locales con Telegram. Este software es un servidor Web Socket que proporciona un medio de enlace público a un servicio.

Como ya se ha dicho anteriormente, el servidor de Rasa tiene configurado las credenciales de Telegram para poder recibir y mandar mensajes, el problema reside en que, en las primeras etapas del desarrollo, realizar tests de funcionamiento con Telegram desde una máquina local, requiere configurar el router para aceptar conexiones por determinados puertos.

Otro problema es la IP pública, hoy en día muy pocos proveedores proporcionan una IP pública estática, y cada vez que se reinicia el router se asigna otra, lo que obligaría a estar reiniciando el servidor para que pueda redirigir correctamente las peticiones.

Este problema no solo se da en entornos locales, sino que, en despliegues en la nube, también se requiere una IP pública para esta conexión.

Así, existen dos problemas a resolver:

1. IP pública no estática
2. Router doméstico abierto

Este software enlaza la IP pública con un subdominio y enlaza un protocolo y un puerto, de forma que no hace falta utilizar la IP pública explícitamente y no hay necesidad de modificar la configuración de router [49].

Una vez ejecutado, proporciona un subdominio el cual habrá que introducir en el servidor de Rasa para que este pueda comunicarse con Telegram.

Este software no es una solución definitiva, pero es una herramienta útil para realizar desarrollos en etapas tempranas del proyecto.

En general, he encontrado pocas soluciones Open Source en las que realizar el despliegue de los recursos del proyecto, por lo que he contemplado soluciones de código cerrado, las cuales, aunque requieren de una suscripción mensual para utilizar sus recursos, proporcionan una fiabilidad y manejabilidad que no se van a encontrar tan fácilmente en otras plataformas.

3.2.8.2. Closed source

3.2.8.2.1. AWS

AWS es un IaaS, una de las plataformas en la nube más reconocidas y adaptadas que hay, ofrece más de 200 servicios en centros de datos a nivel global.

Entre los servicios que ofrece se puede encontrar:

- AWS EC2 (Amazon Elastic Compute Cloud 2)

Proporciona una capacidad de computación escalable en la nube de AWS, lo que elimina la necesidad de invertir en hardware.

Permite, a través de AMIs (Amazon Machine Images) realizar despliegues de determinados sistemas operativos pudiendo configurar plenamente el despliegue de la instancia, desde los recursos del sistema, hasta métodos de seguridad como los Grupos de Seguridad (Security Groups) que actúan como firewalls externos a las instancias.

Los precios de estos despliegues varían según el tipo de AMI elegida, como mínimo existe un coste de infraestructura, pudiendo cobrar adicionalmente por el software que tenga.

El despliegue de dicha instancia proporciona una conexión privada y pública a través de la red, además de un subdominio asociado a dicha instancia para no tener que utilizar la IP pública de la misma. Como se ha comentado en el apartado de Nginx, estos subdominios presentan un problema con la creación de certificados autofirmados para permitir conexiones https desde Telegram al balanceador de carga [50].

- AWS ECR (Amazon Container Registry)

Este servicio proporciona un servicio similar a Docker Hub, con la diferencia de que no se basa en la gestión de versiones sino simplemente en el almacenamiento de las imágenes, las cuales pueden organizarse en repositorios privados o públicos [51].

- AWS ECS (Amazon Container Service)

El servicio anterior permite almacenar imágenes, pero ¿cómo podemos ejecutar dichas imágenes? El servicio de ECS se encarga de gestionar clústeres de contenedores y ejecutarlos [52].

La particularidad de este servicio es que elimina la necesidad de desplegar una instancia y desaprovechar recursos, pudiendo simplemente utilizar contenedores con lo necesario.

- IAM

Este servicio tiene la responsabilidad de gestionar a los usuarios y sus permisos, durante el desarrollo y el testing de esta plataforma, se ha necesitado crear una cuenta root con la cual se tiene acceso a todos los recursos, pero dicha cuenta no debe usarse para desarrollos, solamente para gestionar la propia cuenta [53].

Debido a esto, se creó un usuario regular que inicialmente no disponía de permisos, así, mediante este servicio, se puede configurar dicho usuario y asignarle roles, que proporcionen permisos de EC2, ECS y ECR.

- AWS S3

Una de las necesidades encontradas durante el desarrollo ha sido la necesidad de almacenar los modelos generados al entrar el asistente conversacional en la nube, de forma que al desplegarse los servidores estos no tengan que entrenar al momento, produciendo un aumento significativo del tiempo de despliegue.

Este servicio permite almacenar datos en la nube de una forma sencilla y personalizable, para ello, es necesario crear un bucket o contenedor el cual cuente los archivos almacenados [54].

Estos son solo algunos de los servicios que ofrece AWS, pero son los mínimos y necesarios para el proyecto.

Otro punto importante a la hora de decidir si usar esta plataforma es que no dispone de un plan gratuito o prueba, teniendo que realizar pagos mensuales en función de lo usado.

Una de las particularidades de AWS EC2, es que se paga lo que usas, y se paga por hora. Si la instancia indica que tiene un gasto de $0.004 por hora y se usa

cuatro horas, el costo será de $0.016, pero si dicha instancia de para (que no eliminar) no se realizará un cobro por ella.

El costo mensual dependerá de los servicios que se usen, siendo el servicio de AWS “Elastic IP Address” uno de los servicios que más coste produce.

Se estima que, para un despliegue mínimo del proyecto, tendría un costo mensual de $50.

3.2.8.2.2. Google Cloud

Otra de las grandes plataformas que ofrecen servicios de computación es Google Cloud, mediante Google Compute Engine. Esta plataforma también es de código cerrado, pero al igual que AWS es una de las plataformas que mejor servicio ofrece.

Esta plataforma ofrece servicios de computación como Google Compute Engine para desplegar sistemas de computación y Google Cloud Storage para almacenar datos en la nube.

Al contrario que AWS, Google Cloud no ofrece un servicio como AWS ECR o AWS ECS, los despliegues de contenedores deben realizarse en instancias de Google Compute Engine, de hecho, al crear una instancia, se ofrece una opción para hacer un despliegue de un contenedor dentro de la instancia.

Si se revisan los precios de Google Cloud, se puede ver que las ofertas son muy parecidas a AWS, se paga por lo que se usa, pero existe una gran diferencia que ha hizo que me decantara por usar Google Cloud y es que ofrece $300 gratis y más de 20 productos de forma gratuita durante 3 meses [55].

Si comparamos con AWS, se van a encontrar muy pocas diferencias entre los servicios que ofrecen en lo que a computación y almacenamiento se refiere. En un desarrollo profesional, la plataforma elegida hubiera sido AWS por los servicios de AWS ECR y AWS ECS, puesto que el despliegue se basa en contenedores, pero dada la situación Google Cloud es la alternativa mejor preparada y que más beneficios ofrece.

3.3. Despliegue

El despliegue del chatbot se ha realizado en Google Compute Engine utilizando instancias que se ajusten a los servicios que van a servir, de forma que se utilizan los siguientes tipos de instancias para cada servicio:

	Nginx	Redis DB	Rasa
Instance type	e2-micro	E2-small	E2-small
CPU	1	1	1
RAM	1	2	2

Para la configuración de las instancias se ha utilizado la región “europe-southwest1 (Madrid)” y la zona “europe-southwest1-a”, la cual está marcada con una huella de consumo “CO2 bajo”.

Los scripts utilizados para la automatización se pueden consultar en el repositorio mencionado en el Anexo punto 1, en la carpeta “scripts”.

Estos scripts escritos en bash, necesitan ser configurados para el primer despliegue de forma que se establezcan determinadas variables como la región y la zona antes mencionada, además, en el caso del proceso de despliegue, se establece que los nombres de los servidores de Rasa se llamen de la forma “server-timestamp”, donde “timestamp” es el valor horario asignado en el momento de la creación, lo que proporciona un método visual para identificarlas.

El primer paso necesario es crear la instancia que contendrá el servicio de Nginx mediante Google Compute Engine, el cual proporciona una conexión ssh a la instancia a través del navegador.

Una vez dentro de la instancia creada, es necesario instalar Docker siguiendo su documentación oficial y clonar el repositorio del chatbot en la máquina, una vez clonado y configurado los scripts de despliegue y los ficheros de configuración de Nginx, así como los certificados, se levantará el servicio de Nginx usando el fichero docker-compose-nginx.yml, el cual aún no contendrá ningún servidor al que balancear.

El siguiente paso es levantar en esta misma instancia o en otra el servicio de la base de datos de Rasa, en este caso se ha preferido crearla aparte para simular la independencia de los contenedores Dockers.

Para desplegar este servicio, no es necesario clonar el repositorio sino solamente ejecutar el fichero docker-compose-redis.yml una vez se ha configurado con la contraseña deseada.

Ahora, en la instancia que contiene el servicio de Nginx, es necesario terminar de configurar los scripts de despliegue para que los servicios Docker de Rasa que se desplieguen remotamente, conozcan la dirección IP del servidor de Redis, así como las credenciales para Telegram y el servidor de correo SMTP usado para enviar emails con información de la reserva a los usuarios.

Una vez configurados completamente los scripts de despliegue, se puede añadir un cronjob en la instancia de Nginx, encargado de comprobar cada 30 minutos (este tiempo es configurable) si el número de usuarios activos en el balanceador de carga supera la carga máxima o está por debajo de lo necesario, de modo que, dependiendo de la situación, desplegará servidores Rasa o los destruirá.

Para comprobar si la carga del sistema es adecuada, se ha establecido que cada servidor de Rasa puede servir a 10 usuarios (esta cantidad es personalizable en función de la carga esperada y el hardware de la instancia), de forma que se realiza el siguiente cálculo:

- Si (Nº de servidores * 10) > (Nº de usuarios activos)
- Entonces, se despliega servidor Rasa
- Repetir mientas (Nº de servidores * 10) > (Nº de usuarios activos)

De esta forma, si se tiene en un momento determinado 21 usuarios y hay desplegado un solo servidor de Rasa, se desplegarán dos servidores adicionales dando soporte hasta 30 usuarios.

- 1 servidor – [1 a 10 usuarios]
- 2 servidores – [11 a 20 usuarios]
- 3 servidores – [21 a 30 usuarios]

En el caso contrario, en el que el número de usuarios sea muy inferior a la capacidad del servidor, se destruirán tantos servidores como sea necesario, manteniendo siempre un servidor en funcionamiento.

Si se tienen 9 usuarios y 3 servidores desplegados, quiere decir que se da soporte para 30 usuarios, esto es una pérdida de recursos innecesaria, por lo que se eliminarán 2 servidores.

Se ha elegido este procedimiento puesto que Rasa no proporciona información de los usuarios conectados, en su lugar, se podría consultar la base de datos asociada a los servidores, pero esta mantiene un conteo de los usuarios conectados y desconectados acumulativamente.

Por esto, considero que la mejor opción es comprobar los usuarios activos a través del balanceador de carga a través de un proceso automático y repetitivo como es cron, ya que Nginx está configurado para redirigir las peticiones al puerto 5005 de los servidores especificados en su configuración, siendo los usuarios activos los que han sido aceptados por el servidor de Rasa (petición JSON válida).

En un futuro, cuando se disponga de financiación, se plantea trasladar este despliegue a AWS, de forma que los servicios Docker estén almacenados en AWS ECS sin necesidad de utilizar una instancia, lo cual ahorraría costes de computación.

Además, este proceso de despliegue mediante scripts automatizados sería sustituido por el software Open Source “Jenkins”.

Este software de Integración Continua, permite la automatización de tareas de forma individualizada y clara. Se basa en el desarrollo de “pipelines”, los cuales ejecutan tareas independientes, por ejemplo, se tendrían los siguientes pipelines:

- Deploy_rasa_server
- Destroy_rasa_server
- Deploy_nginx_server
- Deploy_redis_server
- Build_containers (Redis, testing, training, rasa server, rasa action)
- Train_model (entrena el chatbot y lo publica en la nube para que sea accesible)

Estos pipelines tendrían sus parámetros y serían llamados mediante un pipeline que se ejecute cada X tiempo de la misma manera que lo hace cron.

La gran ventaja de este software, es que permite una gran personalización y dispone de una interfaz desde la cual se puede consultar el estado de cada proceso ejecutado, pudiendo observar y guardar el correspondiente log.

Además, este software no solo serviría para realizar despliegues, sino que también sustituiría a GitHub Actions, el cual actualmente tiene ciertas limitaciones como un máximo de minutos que puede usarse, o que no es todo lo transparente que debería ser al realizar determinados procesos.

Esta integración sería sustituida de la siguiente forma:

- En GitHub, se establece un Webhook a AWS Lambda cada vez que se realiza o se mergea un Pull Request, esto es totalmente personalizable por parte de GitHub.
- AWS Lambda recibe la petición y mediante una función serverless manda una petición JSON a Jenkins, de forma que se ejecuta un pipeline u otro.

3.4. Tests

Una de las partes más importantes a la hora de realizar un desarrollo son los test y Rasa proporciona ciertas herramientas para probar la funcionalidad del bot que se está desarrollando.

3.4.1. Data validation

La validación de datos comprueba que no haya errores en los elementos establecidos en los ficheros nlu.yml, domain.yml, rules.yml, de forma que se puedan detectar inconsistencias que puedan llevar a un mal funcionamiento del bot [56].

Para ejecutar este tests, basta con utilizar el comando “rasa data validate”.

En el caso de que se encuentre algún error, se verá un mensaje como el siguiente:

```
/Documents/TFG/ChatbotTFG/venv/lib/python3.7/site-packages/rasa/shared/
utils/io.py:99: UserWarning: The form 'HorarioGet' is used in the 'El
usuario pregunta por horario' block, but it is not listed in the domain
file. You should add it to your domain file!
 More info at https://rasa.com/docs/rasa/forms
Project validation completed with errors.
```

Figura 10: Error al validar la estructura NLU.

El bloque “El usuario pregunta por horario”, es un bloque del fichero rules.yml, que tiene la siguiente estructura:

```
- rule: El usuario pregunta por horario
  steps:
    - intent: horario
    - action: HorarioGet
    - active_loop: HorarioGet
```

Figura 11: Estructura de una regla en Rasa.

Al comprobar el fichero domain.yml, se puede ver que efectivamente, no hay un form con este nombre, al corregir los problemas que vayan apareciendo, llegará un momento donde no habrá más problemas y se verá el siguiente output:

```
rasa.validator  - Validating intents...
rasa.validator  - Validating uniqueness of intents and stories...
rasa.validator  - Validating utterances...
rasa.validator  - Story structure validation...
Processed story blocks: 100%
rasa.core.training.story_conflict  - Considering all preceding turns
for conflict analysis.
rasa.validator  - No story structure conflicts found.
```

Figura 12: Validación de NLU exitosa.

3.4.2. Stories tests

Rasa también proporciona un método para poder probar las historias que se escriben, para ello, es necesario crear un fichero cuyo nombre empiece por “test” y guardarlo en el directorio con nombre: “tests”.

Para mantener el dominio de las historias, he creado cuatro ficheros tests:

1. Tests de reservas exitosas.
2. Tests de reservas fallidas.
3. Tests de menús exitosos.
4. Tests de menús fallidos.

Los ficheros exitosos hacen referencia a historias que acaban según lo planeado, el usuario quiere algo y lo consigue, en cambio, las historias fallidas hacen referencia a historias que han divergido en algún punto de la conversación separándose de su objetivo principal, o que simplemente el propio usuario ha decidido parar.

Para poder realizar los tests a las historias es necesario modificar las historias originales introduciendo un intent con una de las frases de ejemplo, de forma que simula un mensaje del usuario, de esta forma, cada acción realizada debe corresponder con el intent anterior que se espera.

```
stories:
 - story: Reserva happy path
   steps:
     - user: |
         quiero reservar mesa
       intent: reservar
     - or:
         - slot_was_set:
             - reserva_completa: false
         - slot_was_set:
             - reserva_completa: null
     - action: utter_reserva_dia
```

Figura 13: Ejemplo de interacción de usuario dentro de un test de historia.

En este caso, se simula la entrada de usuario con la frase “quiero reservar mesa” el cual corresponde con el intent “reservar”, en la historia que se utiliza para entrenar el bot, ese es el intent especificado, esto es solo una forma de forzar una conversación para averiguar si dicha conversación es funcional [57].

Para ejecutar los tests, Rasa proporciona el siguiente comando: “rasa test”

Al ejecutar dicho comando, se lanza un proceso de evaluación que evalúa no solo las historias, sino las acciones, los intents y demás parámetros que pueda extraer el bot. Los resultados del test se almacenan en un directorio llamado “results”, donde se pueden encontrar diversos archivos, como análisis de los intents, las historias y TEDpolicy.

Algunos ejemplos son los siguientes:

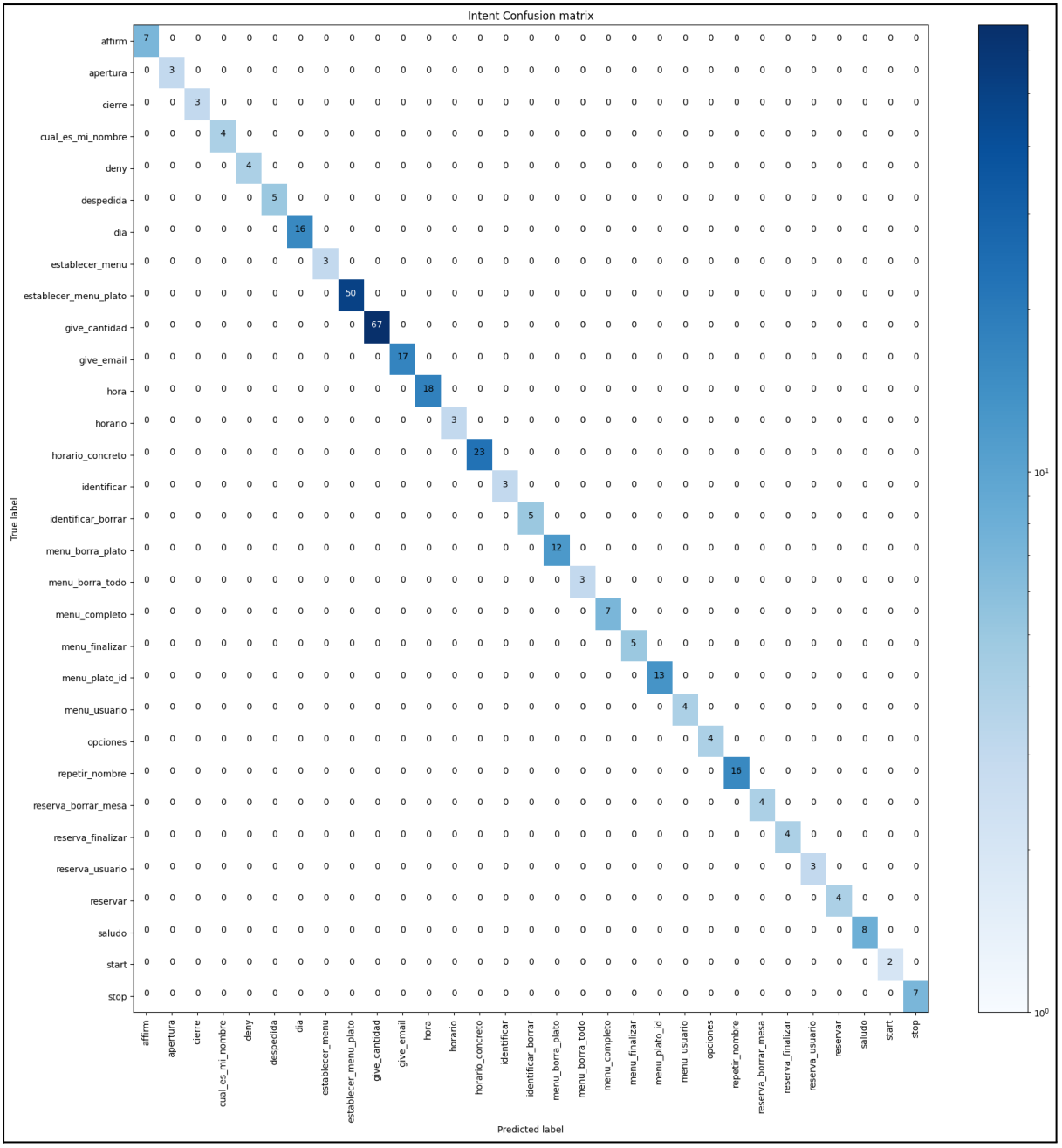

Figura 14: Intent Confusion matrix.

En la figura superior, se puede observar el número de veces que cada intent ha sido reconocido, un color más oscuro corresponde con un mayor número de veces.

La forma de leer dicha gráfica es la siguiente:

- “El intent predecido ‘affirm’ ha sido reconocido 7 veces con el intent ‘affirm’”
- “El intent ‘establecer_manu_plato’ ha sido reconocido 50 veces con el intent ‘establecer_manu_plato”

Si encontramos algún cuadrante que no estuviera en la diagonal con un valor superior a 0, significa que ha habido un intent que ha sido reconocido erróneamente con otro. En este caso, todo ha sido reconocido correctamente.

Como ejemplo de dicho comportamiento se puede observar la siguiente gráfica:

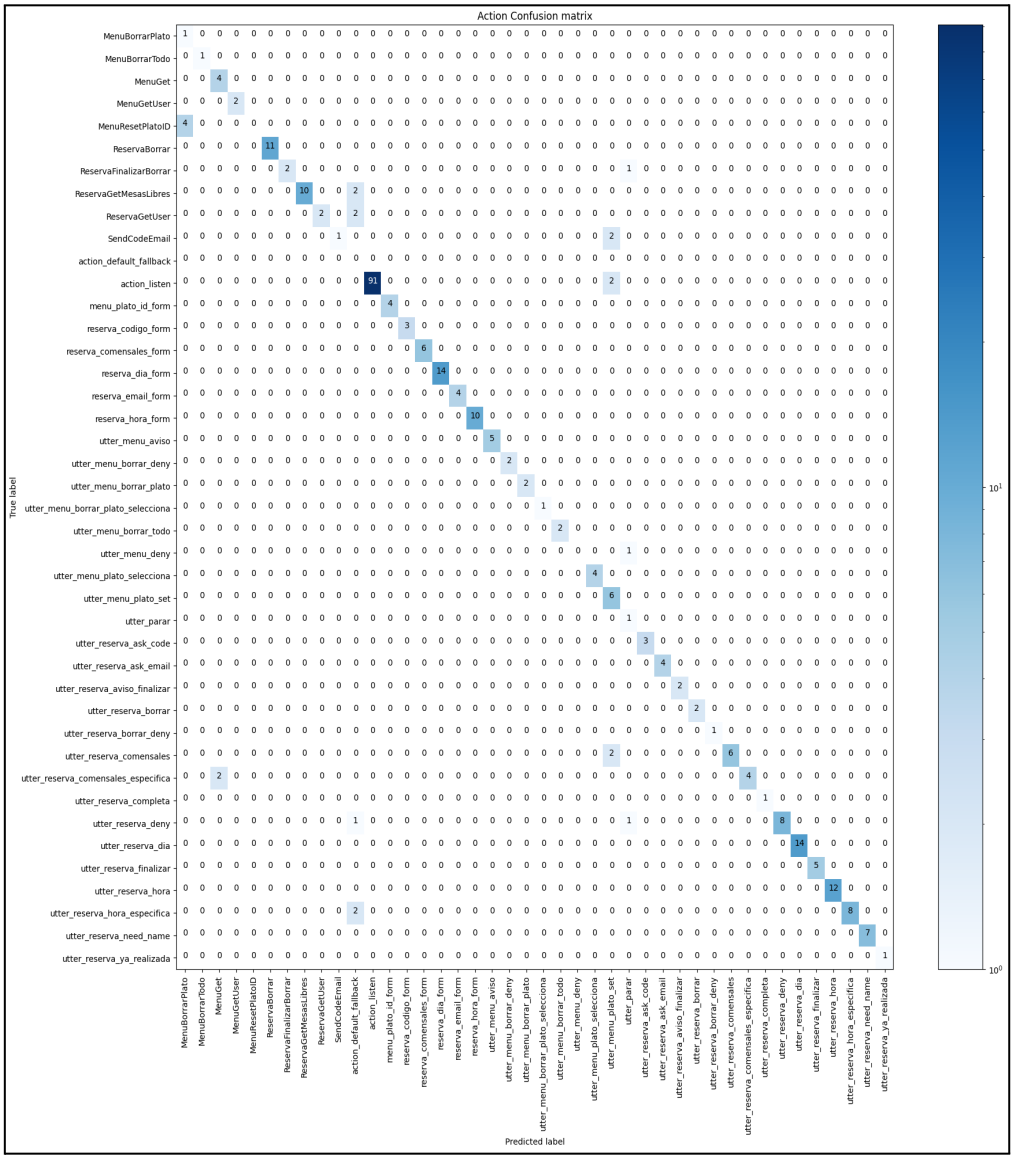

Figura 15: Action Confusion matrix.

Esta gráfica representa las acciones de historias que han sido confundidas con otras acciones, donde al igual que la gráfica de intents, un valor alto en la diagonal es bueno, y un valor alto fuera de la diagonal es malo.

Capítulo 4. Conclusiones y Trabajo Futuro

4.1. Conclusiones

El objetivo del desarrollo que se ha llevado a cabo era la creación de un asistente conversacional tipo chatbot que fuera capaz de gestionar las reservas de los clientes, el cual creo que se ha cumplido en una gran medida ya que es capaz de conversar con una persona de una forma lo más natural posible, es capaz de analizar lo que quiere dicha persona y proveer de una respuesta adecuada a dicha persona, guiándola a través de todo el proceso de reserva.

Es cierto, que en lo que se refiere al chatbot, quedan muchas cosas que mejorar y por hacer, si de algo me he dado cuenta durante el desarrollo del mismo, es que cuando hablamos con otra persona, no somos conscientes de la cantidad de información y de casuísticas que pueden darse sin previo aviso, sin necesidad de que exista una lógica de por medio, controla esto en una IA requiere mucho esfuerzo.

Otro de los objetivos era conseguir un despliegue avanzado del sistema para el acceso al mismo dependiendo del número de usuarios. Este objetivo también se ha cumplido ya que se ha conseguido automatizar, a través de datos recolectados por un balanceador de carga, el despliegue o destrucción de servidores, consiguiendo que la capacidad de respuesta de los servidores desplegados siempre esté ajustada a la necesidad del momento.

Hay que tener presente, que realmente el objetivo más importante a conseguir es el aprendizaje conseguido durante todo el proceso y la metodología de desarrollo del mismo, yo no conocía Rasa hasta el momento de realizar este TFG y el único agente conversacional que había hecho anteriormente fue en la asignatura de Inteligencia Artificial usando lenguaje AIML de una forma muy superficial. Esto me ha hecho darme cuenta de las posibilidades que tienen realmente este tipo herramientas y como alguien que actualmente trabaja realizando procesos de automatización de CI/CD, estoy viendo posibles aplicaciones prácticas que podrían ayudar en el trabajo del día a día, como, por ejemplo, obtener información una

infraestructura o hacer que lance determinados procesos que requeriría una tarea repetitiva.

4.2. Trabajo Futuro

Algunas de las opciones que se contemplan para futuros desarrollos y mejoras son las siguientes:

- Mejora en el reconocimiento de intents, la forma de hablar de las personas es muy variada y poder reconocer correctamente un mensaje de un usuario es complicado, por esto, se seguirá trabajando en la mejora de reconocimiento de texto

- Desplegar el bot en webs, de esta forma, el bot puede estar presente en la página del establecimiento, proporcionando una interacción más directa.

- Mejora de los tests, aumentar la cantidad de tests realizados sobre los componentes del chatbot, como pueden ser tests sobre las acciones personalizadas.

- Mejorar la integración continua, actualmente se está utilizando GitHub actions debido a recursos, principalmente económicos. Una vez se tenga financiación, se puede desarrollar la automatización de los procesos en Jenkins, que proporciona una calidad y personalización sobre el proceso mucho mayor que GHA.

- Mejorar el despliegue del bot, al igual que se nombra Jenkins para la integración continua, también se puede utilizar para realizar despliegues mediante pipelines independientes en diferentes plataformas, como AWS o Azure.

- Incluir la habilidad de reconocer audios y responder de la misma forma.

Bibliografía

[1] ChatCompose - 2019 | Historia de los chatbots
https://www.chatcompose.com/historia.html

[2] Automatichat - Daniel Marcovich | Chatbots que marcaron historia
https://automatichat.com/blog/la-historia-de-los-chatbots

[3] Aunoa | Características de Chatbots con IA vs Chatbots guiados
https://aunoa.ai/caracteristicas-de-los-chatbots-guiados-vs-chatbots-con-ia/

[4] Edix - 2022/07/26 | Frameworks https://www.edix.com/es/instituto/framework/

[5] GeekFlare - 2022 | 10 mejores marcos de desarrollo de chatbot para construir potentes bots https://geekflare.com/es/chatbot-development-frameworks/

[6] Tribalyte - Álvaro Gómez | Como crear un chatbot con Dialogflow
https://tech.tribalyte.eu/blog-crear-un-bot-conversacional-con-dialogflow

[7] Google Cloud Documentation | Dialog flow pricing
https://cloud.google.com/dialogflow/pricing?hl=es-419

[8] Amazon Lex Documentation | Crear chatbots con IA conversacional
https://aws.amazon.com/es/lex/

[9] Amazon Lex Documentation | Amazon Lex Pricing
https://aws.amazon.com/es/lex/pricing/

[10] Rasa | Rasa Features https://rasa.com/product/features/

[11] Microsoft Bot Framework | Pricing
https://azure.microsoft.com/es-es/pricing/details/bot-services/

[12] Centribal | NPL chatbots https://centribal.com/es/nlp-chatbot/

[13] Iberdrola | Tipos de chatbots
https://www.iberdrola.com/innovacion/que-es-un-chatbot

[14] Blog UPV - 2011 | Historia de las bases de datos
https://histinf.blogs.upv.es/2011/01/04/historia-de-las-bases-de-datos/

[15] Ayudaley | Bases de datos jerárquicas ¿Qué son? Ejemplos
https://ayudaleyprotecciondatos.es/bases-de-datos/jerarquicas/

[16] Ayudaley | Base de datos de red. ¿Qué es? Ejemplos
https://ayudaleyprotecciondatos.es/bases-de-datos/red/

[17] Oracle Documentation | ¿Qué es una base de datos relacional (sistema de gestión de bases de datos relacionales)?https://www.oracle.com/es/database/what-is-a-relational-database/

[18] NotJustBI - Francisco Martinsky - 2020/06/24 | ¿Qué son las bases de datos?
https://notjustbi.com/que-son-las-bases-de-datos/

[19] Ayudaley | Base de datos orientadas a objetos ¿Qué son?
https://ayudaleyprotecciondatos.es/bases-de-datos/orientas-a-objetos/

[20] Tibco 2022 | ¿Qué es una base de datos en memoria?
https://www.tibco.com/es/reference-center/what-is-an-in-memory-database

[21] CESUMA - Administrador CESUMA | Cloud computing: Historia y propiedades
https://www.cesuma.mx/blog/cloud-computing-historia-y-propiedades.html

[22] OpenWebinars - Frankier Flores - 2021/03/22 | Cloud computing: Tipos de nubes, servicios y proveedores
https://openwebinars.net/blog/tipos-de-cloud-computing/

[23] CloudCenterAndalucia - cca-admin |
https://www.cloudcenterandalucia.es/blog/iaas-paas-y-saas-que-son-ejemplos-y-diferencias/#Ejemplos_de_PaaS

[24] Rasa Forum - Tobias_Wochinger - 2019/06/21 | Rasa Core internal code explanation https://forum.rasa.com/t/rasa-core-internal-code-explanation/10912

[25] Rasa Open Source Documentation | Rasa NLU
https://rasa.com/docs/rasa/nlu-training-data/

[26] Rasa Open Source Documentation | Rasa Architecture Overview
https://rasa.com/docs/rasa/arch-overview/

[28] Rasa Open Source Documentation | Domain https://rasa.com/docs/rasa/domain/

[29] Rasa Open Source Documentation | Slots
https://rasa.com/docs/rasa/domain/#slots

[30] Rasa Open Source Documentation | Entities
https://rasa.com/docs/rasa/domain/#entities

[31] Rasa Open Source Documentation | Intents
https://rasa.com/docs/rasa/domain/#intents

[32] Rasa Open Source Documentation | Responses
https://rasa.com/docs/rasa/responses

[33] Rasa Open Source Documentation | Forms https://rasa.com/docs/rasa/forms

[34] Rasa Open Source Documentation | Actions https://rasa.com/docs/rasa/actions

[35] Rasa Open Source Documentation | Rules https://rasa.com/docs/rasa/rules/

[36] Rasa Open Source Documentation | Stories https://rasa.com/docs/rasa/stories/

[37] Rasa Open Source Documentation | Connecting to Messaging and Voice Channels https://rasa.com/docs/rasa/messaging-and-voice-channels/

[38] Rasa Open Source Documentation | Endpoints
https://rasa.com/docs/rasa/tracker-stores/

[39] Rasa Open Source Documentation | Introduction to Rasa Action Servers
https://rasa.com/docs/action-server/

[40] Redis Documentation | About https://redis.io/docs/about/

[41] Kinsta - 2020/12/29 | Git y GitHub: ¿Cuál es la Diferencia y cómo Empezar?
https://kinsta.com/es/base-de-conocimiento/git-vs-github/

[42] Redhat Documentation - 2018/01/09 | ¿Qué es DOCKER?
https://www.redhat.com/es/topics/containers/what-is-docker

[43] Irontec - Rubén Gómez Olivencia - 2018/10/29 | Introducción, muy breve y desenfadada, a Docker
https://blog.irontec.com/introduccion-muy-breve-y-desenfadada-a-docker/

[44] Ondho - Óscar Villacampa - 2021/01/29 | Qué es Docker y para qué sirve
https://www.ondho.com/que-es-docker-para-que-sirve

[45] Kinsta - 2022/02/21 | ¿Qué es Nginx y Cómo funciona?
https://kinsta.com/es/base-de-conocimiento/que-es-nginx/

[46] Heroku Web | Heroku pricing https://www.heroku.com/pricing

[47] Heroku Help | Removal of Heroku Free Product Plans FAQ
https://help.heroku.com/RSBRUH58/removal-of-heroku-free-product-plans-faq

[48] Redhat Documentation | OpenStack
https://www.redhat.com/es/topics/openstack

[49] Alexi A.C.V | ¿Qué es Ngrok y cómo utilizarlo?
https://www.elcursodelhacker.com/ngrok/

[50] AWS Documentation | ¿Qué es Amazon EC2?
https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/concepts.html

[51] AWS Documentation | ¿Qué es Amazon Elastic Container Registry?
https://docs.aws.amazon.com/es_es/AmazonECR/latest/userguide/what-is-ecr.html

[52] AWS Documentation | ¿Qué es Amazon Elastic Container Service?
https://docs.aws.amazon.com/es_es/AmazonECS/latest/developerguide/Welcome.html

[53] AWS Documentation | ¿Qué es IAM?
https://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/introduction.html

[54] AWS Documentation | ¿Qué es Amazon S3?
https://docs.aws.amazon.com/es_es/AmazonS3/latest/userguide/Welcome.html

[55] Google Cloud | Google Cloud pricing https://cloud.google.com/pricing/

[56] Rasa Open Source Documentation | Validating Data and Stories
https://rasa.com/docs/rasa/testing-your-assistant/#validating-data-and-stories

[57] Rasa Open Source Documentation | Writing test stories
https://rasa.com/docs/rasa/testing-your-assistant/#writing-test-stories

[58] Stackscale | Imagen Modelos de servicio cloud IaaS, PaaS y SaaS
https://www.stackscale.com/es/blog/modelos-de-servicio-cloud/

[59] Rasa Enterprise Documentation | Share assistant
https://rasa.com/docs/rasa-enterprise/user-guide/share-assistant

[60] Rasa forum - Greg Stephens | Avoid hardcoding secrets in credentials.yml file
https://forum.rasa.com/t/avoid-hardcoding-secrets-in-credentials-yml-file/54320

Anexo

1. Repositorio de GitHub

Todo el código desarrollado se puede encontrar en la plataforma GitHub, en el repositorio:

- https://github.com/rauldpm/ChatbotTFG

2. Manual de usuario

2.1. Instalación de Rasa

Para los despliegues que no se realicen en un contenedor Docker, es necesario instalar y configurar Rasa, puedes seguir la documentación oficial para ello: https://rasa.com/docs/rasa/installation

- Crear un nuevo entorno virtual con python3
    - python3 -m venv ./venv

- Activar el entorno virtual
    - source ./venv/bin/activate

- Instalar Rasa Open Source
    - pip3 install -U pip && pip3 install rasa

2.2. Despliegue del chatbot

Para realizar un despliegue local del chatbot se necesita descargar el repositorio de GitHub mediante el comando:

- git clone https://github.com/rauldpm/ChatbotTFG.git

Una vez clonado es necesario acceder al directorio del chatbot:

- cd ChatbotTFG

Ahora dispones de diferentes opciones de despliegue (se requiere disponer de un bot de Telegram creado mediante BotFather)

2.2.1. Despliegue local de Rasa sin Docker

Este despliegue describe los pasos necesarios para ejecutar los servidores de Rasa en tu máquina local directamente:

- Configura tu base de datos en el fichero “endpoints.yml”, en caso de que no dispongas de una base de datos Redis propia puedes desplegar un contenedor mediante docker-compose.

- En caso de que no dispongas de base de datos propia, ejecuta: “docker-compose -f docker-compose-redis.yml” dentro de la carpeta “Docker/docker-compose”.

- Configura las credenciales de acceso a tu bot de Telegram en el fichero “credentials.yml”.

- Antes de iniciar el servidor de Rasa es necesario entrenar el chatbot, para ello ejecuta: “rasa train”.

- Ahora en un terminal ejecuta el comando: “rasa run actions” para ejecutar el servidor de acciones.

- Ejecuta en otro terminal el comando: “rasa shell”, para iniciar el servidor de Rasa.

2.2.2. Despliegue local de Rasa con Docker

- Configura las credenciales de acceso a tu bot de Telegram en el fichero “secrets/telegram_secrets.env”.

- Configura las credenciales de tu servidor smtp en el fichero “secrets/email_secrets.env”.

- Ejecuta el contenedor de la base de datos de Redis.
    - docker-compose -f docker-compose-redis.yml up --build

- Configura en el fichero endpoints.yml la IP de la base de datos de Redis.

- Ejecuta el contenedor de Rasa server y Rasa actions.
    - docker-compose -f docker-compose-rasa.yml up --build

2.2.3. Despliegue del balanceador de carga Nginx

En caso de querer usar un balanceador de carga, se proporciona un fichero docker-compose-nginx.yml mediante el cual se despliega un contenedor con Nginx. Debes asegurarte que todas las peticiones se realizan al contenedor en vez de al servidor o servidores.

Por seguridad, es necesario disponer de dos certificados autofirmados con letsencrypt asociados a un DNS correspondiente a la máquina donde Nginx está desplegado.

- Coloca en “Docker/nginx/certs/” tus certificados con el nombre de “fullchain.pem” y “privkey.pem”

- Modifica el fichero “Docker/nginx/conf/servers” introduciendo la IP de tu servidor de Rasa.

- Inicia el servicio de Nginx usando docker-compose:
    - docker-compose -f docker-compose-nginx.yml up --build

El balanceador de carga actúa además como proxy inverso, permitiendo que todas las conexiones se realicen mediante https.

2.2.4. Despliegue de Redis DataBase

Si no dispones de una base de datos propia para el servidor de Rasa, puedes realizar un despliegue rápido de un contenedor con esta base de datos.

Para realizar esto, navega al directorio “Docker/docker-compose/” y sigue los siguientes pasos:

- Modifica el fichero docker-compose-redis.yml cambiando “redis-password” por la contraseña establecida en el fichero “endpoints.yml”.

- Construye e inicia el contenedor:
    - docker-compose -f docker-compose-redis.yml up --build

2.3. Como conversar con el bot

2.3.1. Localmente

Una vez tengas desplegado el bot localmente, puedes conversar con él de dos maneras diferentes.

La primera es mediante lenguaje natural, esto lo puedes realizar si ejecutas el servidor mediante el comando “rasa shell”.

El segundo caso es mediante peticiones curl en formato JSON al servidor, este método debe ser usado si el servidor ha sido desplegado usando Docker, como ejemplo, se pueden enviar mensajes de la forma:

- curl -XPOST http://RASA_SERVER_IP:5005/webhooks/rest/webhook -H "Content-type: application/json" -d '{"sender": "test", "message": "Hola"}'

Si estás usando el balanceador de carga provisto en el repositorio, las peticiones se pueden hacer mediante http o https sin necesidad de especificar el puerto 5005:

- curl -XPOST http://LOAD_BALANCER_IP/webhooks/rest/webhook -H "Content-type: application/json" -d '{"sender": "test", "message": "Hola"}'

- curl -XPOST https://LOAD_BALANCER_IP/webhooks/rest/webhook -H "Content-type: application/json" -d '{"sender": "test", "message": "Hola"}'

2.3.2. Telegram

Accede a Telegram y busca: “TFG_Restaurante”, al abrir la conversación verás lo siguiente:

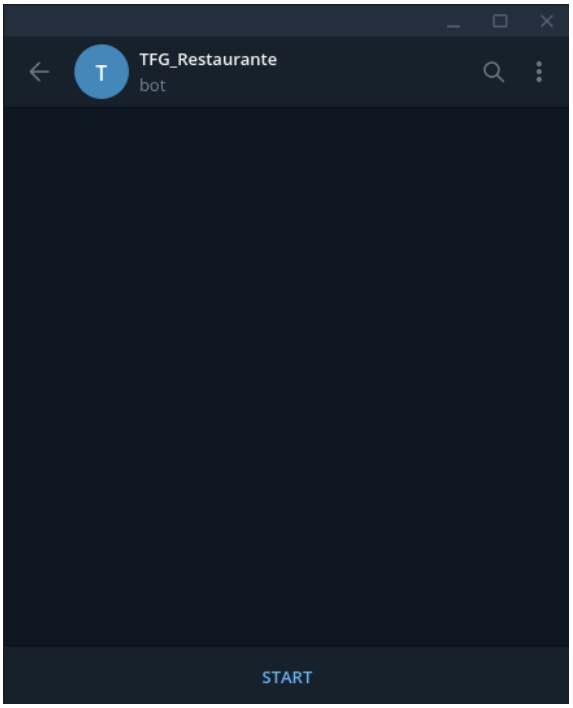

Figura 16: Conversación con el bot en Telegram sin iniciar.

Una vez iniciado el bot al pulsar START, este se presentará y te mostrará una lista de opciones en forma de botones:

Figura 17: Bot inicia la conversación mostrando las opciones iniciales.

Puedes interactuar con el bot ya sea mediante mensajes de texto o usando los botones mostrados al introducir “Opciones”, un ejemplo de conversación sería el siguiente:

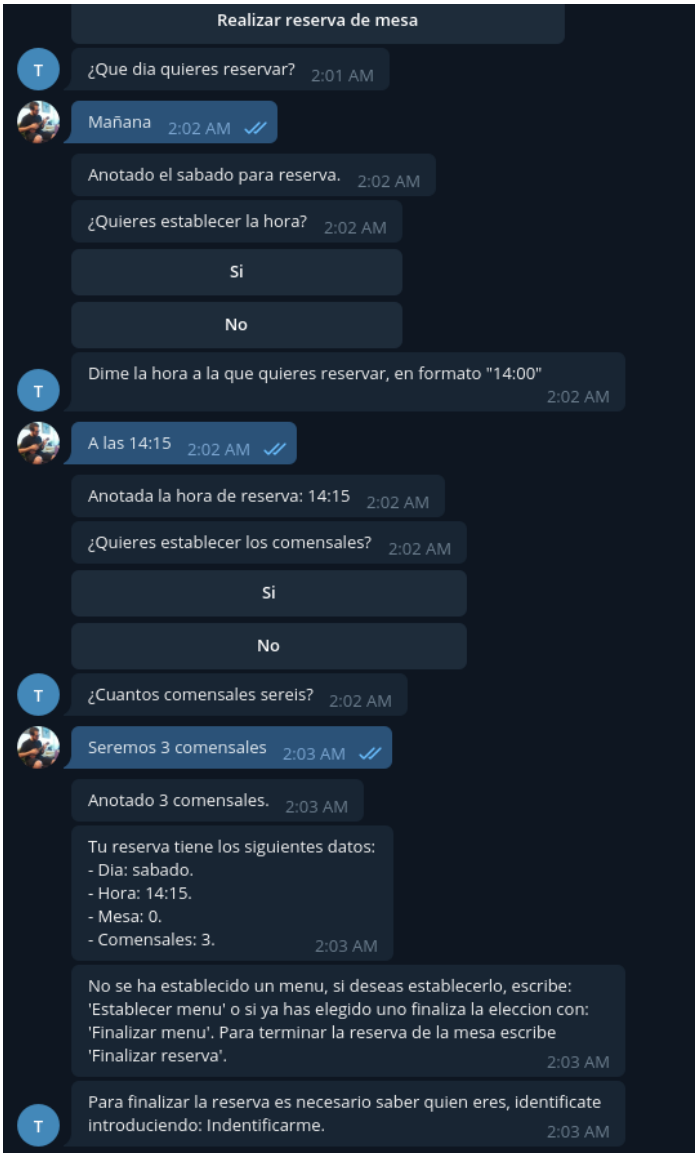

Figura 18: Conversación con el bot.