ShowMeTheModel: Plataforma para la ejecución desde dispositivo móvil de modelos de ML/DL

Trabajo de Fin de Grado

Ingeniería Informática

Julio de 2023

Juan Carlos Velasco Sánchez

_____

Ángeles María Moreno Montero

Armando Iban Sánchez

_____

VNiVERSiDAD D SALAMANCA
CAMPUS DE EXCELENCIA INTERNACIONAL

Facvltad D Ciencias
VNiVERSiDAD D SALAMANCA

Certificado del/los tutor/es TFG

Dña.Ángeles Mª Moreno Montero, profesora del Departamento de Informática y Automáticade la Universidad de Salamanca, y D. Armando Ibán Sánchez , ingeniero de Software de HP,

HACE/N CONSTAR:

Que el trabajo titulado "ShowMeTheModel: Plataforma para la ejecución desde dispositivo móvil de modelos de ML/DL", que se presenta, ha sido realizado por Juan Carlos Velasco Sánchez, con DNI ****9966K y constituye la memoria del trabajo realizado para la superación de la asignatura Trabajo de Fin de Grado en Ingeniería Informática en esta Universidad.

Salamanca, 3 de julio de 2023

Firmado digitalmente por Ángeles Mª Moreno Montero
Fecha: 2023.07.04 11:40:12 +02'00'

Fdo.: _____

Fdo.: Armando Ibán Sánchez _____

Resumen

En un mundo cada vez más influenciado por la Inteligencia Artificial, los equipos que desarrollan modelos de *Machine Learning* buscan encontrar formas cómodas de mostrar los resultados de sus investigaciones. Sin embargo, estos modelos suelen requerir una gran potencia de cálculo, lo que dificulta exponerlos públicamente.

El objetivo del proyecto ShowMeTheModel es abordar este desafío mediante el desarrollo de una plataforma que permita la ejecución remota de modelos de *Machine Learning.* A través de esta plataforma, los usuarios finales podrán acceder desde un navegador web y utilizar texto o imágenes como entrada. Esto es posible gracias al uso de un agente externo, un servidor con mucha potencia de cálculo y que es el responsable de ejecutar los modelos.

De esta manera, los usuarios podrán utilizar los modelos de *Machine Learning*, ya que es el agente externo se encargará de ejecutar los modelo. Esta solución permite a los usuarios utilizar los modelos en cualquier lugar, resolviendo así el problema mencionado.

Palabras clave: *Machine Learning*, remoto, modelos, inteligencia artificial

Abstract

In a world increasingly influenced by Artificial Intelligence, teams developing Machine Learning models are looking for convenient ways to display their research results. However, these models often require a great deal of computational power, which makes it difficult to expose them publicly.

The goal of the ShowMeTheModel project is to address this challenge by developing a platform that enables the remote execution of Machine Learning models. Through this platform, end users will be able to access from a web browser and use text or images as input. This is possible thanks to the use of an external agent, a server with a lot of computing power, which is responsible for executing the models.

In this way, users will be able to use the Machine Learning models, since it is the external agent that will be in charge of running the models. This solution allows users to use the models anywhere, thus solving the aforementioned problem.

Key words: Machine Learning, remote, models, artificial intelligence, artificial intelligence.

Agradecimientos

Me gustaría agradecer en primer lugar a mis tutores de este TFG, tanto de la USAL, como de HP, por ayudarme en el desarrollo del proyecto, y por haberme acompañado en este viaje.

Me gustaría dar las gracias también a la USAL, que me ha dado la oportunidad de hacer este proyecto con una empresa tan reconocida a nivel mundial, y que creo que la he aprovechado al máximo para aprender, ya que al final, el aprendizaje es lo que me llevo de todo este proyecto.

También agradecer a todos los profesores que he tenido, que me han enseñado tanto durante todos estos años, tanto conocimiento en sí como valores humanos.

Por último, agradecer a todos los compañeros de la carrera, que han sido parte de todo este viaje, a mi familia y amigos, y todas las personas que me han apoyado y que me han llevado hasta aquí.

Tabla de contenido

Tabla de figuras

1. Introducción

Una de las tareas más habituales que desea realizar un equipo enfocado al desarrollo de modelos de *Machine Learning*, *Deep Learning* e IA (Inteligencia Artificial) en general es poder enseñar de una forma rápida sus resultados a terceros.

No obstante, uno de los problemas a los que se enfrentan este tipo de equipos es la imposibilidad de exponer sus máquinas internas de desarrollo al mundo exterior cuando acuden a ferias y congresos.

En el proyecto ShowMeTheModel se propone desarrollar una plataforma que permita consumir desde una aplicación móvil o un navegador estos modelos de forma remota, aunque se encuentren dentro de una red privada. Así, se podrán utilizar como entrada archivos del propio móvil, un texto introducido en él o una imagen capturada con su cámara.

Por otro lado, la ejecución remota de los modelos la llevará a cabo un agente, que será un servidor que ejecuta los modelos de *Machine Learning*.

Este proyecto se ha realizado en colaboración con HP, empresa muy reconocida a nivel mundial.

2. Objetivos del proyecto

Los objetivos del proyecto que se proponen son:

- Desarrollo de un servicio web, desplegado en la nube, al que se conectará la aplicación móvil y que será encargada de enviar las peticiones de trabajo a los diferentes agentes. También recibirá los resultados por parte de estos, y gestionará su registro y estado.

- Desarrollo de un agente configurable que se ejecutará en las máquinas donde residan los modelos de *Machine Learning*, normalmente desarrollados en Python. Este agente se encargará de recibir peticiones en forma de mensajes descargar los archivos que sean necesarios (imágenes, vídeo, sonido), ejecutar el modelo de *Machine Learning* y subir los resultados donde sea necesario, comunicando el resultado.

3. Técnicas y herramientas

Para hablar de las técnicas y herramientas utilizadas, es preciso que primero se hable de la primera aproximación al proyecto deseado.

3.1 Esquema del proyecto inicial

La idea inicial consiste en hacer una aplicación a la que los usuarios accederán a través de un navegador web. Allí, éste creará una petición, seleccionando un modelo cuya entrada puede ser un texto o una imagen, y que la aplicación enviará a una cola de mensajes. Una vez enviada la petición, el cliente se quedará a la espera de recibir el resultado, que vendrá en otro mensaje que estará en la cola de mensajes.

Por otro lado, las peticiones serán recibidas por un agente, el cual estará alojado en un ordenador de HP, en una red privada. Este agente es el encargado de procesar la entrada de los usuarios y aplicar a esa entrada los modelos de *Machine Learning.*

El agente se ejecuta continuamente y se encarga de recibir los mensajes enviados por el usuario, aplicarles los modelos correspondientes, y devolver el resultado en un mensaje que irá a la cola de mensajes.

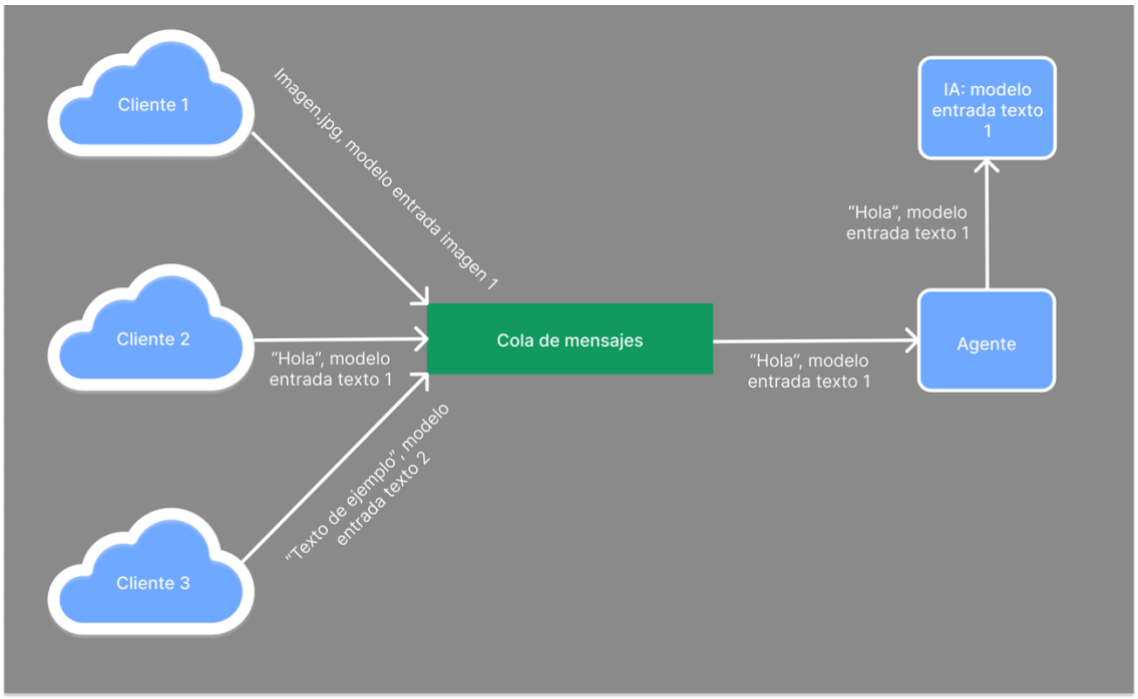


Figura 1: esquema del envío de mensajes

En la figura anterior se puede apreciar como tres clientes mandan su petición, con lo cual se enviarán tres mensajes a la cola. El agente, en este caso, recoge uno de los tres, el mensaje del cliente dos, y le aplica el modelo correspondiente.

El modelo que aplica el agente será seleccionado por el usuario, pero de manera coherente, es decir, si el usuario ha elegido un modelo cuya entrada es texto, la

aplicación permitirá que el usuario introduzca texto. Los modelos que el usuario puede elegir son los modelos que el agente puede ejecutar.

Una vez el agente ha enviado su respuesta a la cola, el cliente lo recogerá y mostrará el resultado de la petición.

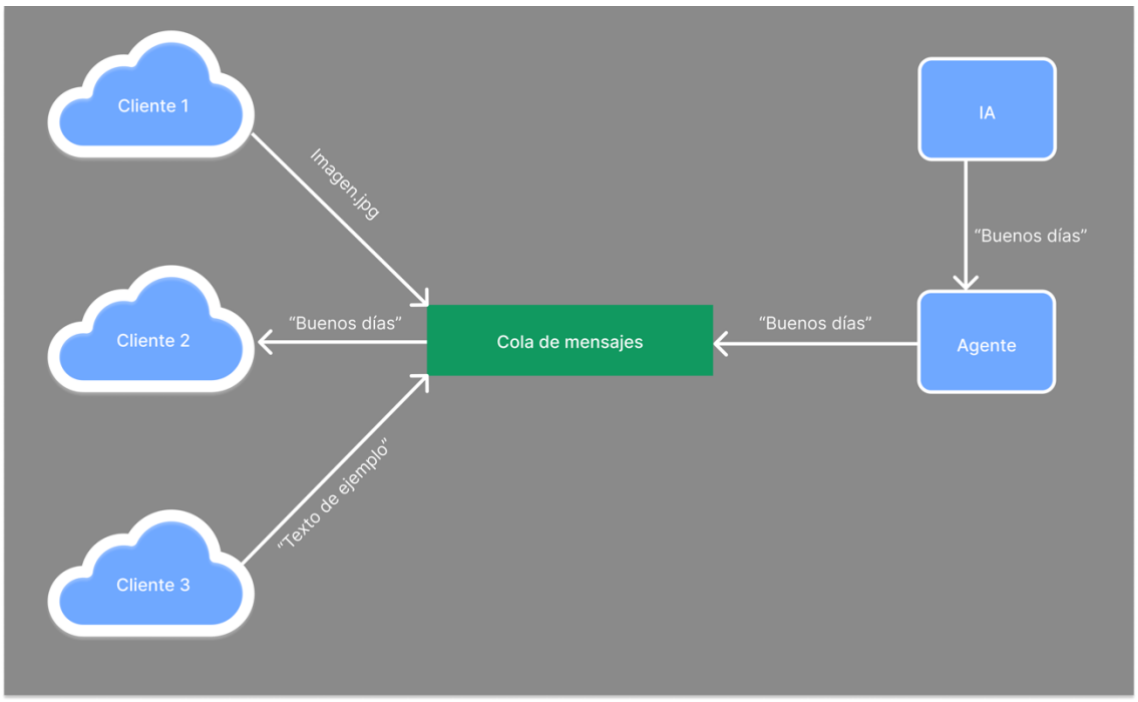


Figura 2: esquema del envío de mensajes (parte 2)

En la imagen anterior se puede observar como el agente obtiene el resultado de aplicar el modelo cuya entrada fue el mensaje del cliente 2, y que envía a la cola de mensajes. El cliente 2 recoge el mensaje y lo mostrará al usuario.

Para que la aplicación pueda ser accesible a todos los usuarios, está aplicación deberá de estar desplegada en la nube.

3.2 Herramientas utilizadas

En este apartado se explicarán las herramientas utilizadas para el proyecto.

Al final del proyecto, las herramientas utilizadas han sido las siguientes:

- Gitlab [1]: es el repositorio utilizado para almacenar el código. Este repositorio es el repositorio que proporcionó HP para alojar el código del proyecto.

- AWS SQS [2]: sistema de colas de mensajes de Amazon Web Services [3]. Este sistema de colas de mensajes se usa para que el agente y el cliente se puedan comunicar de manera remota. La principal ventaja de este servicio es que la configuración es muy sencilla y es un servicio con un alto grado de escalabilidad.

  Se investigaron otras alternativas como Apache Kafka [4] o RabbitMQ [5], pero estas no proponen tener un servidor de cola de mensajes en la nube, sino en local. Además, la configuración de estos servidores era mucho más complejo.

- Render [6]: servicio web de *hosting* de páginas web, bases de datos, servidores, etc. En este proyecto se utiliza para alojar en la nube la base de datos del proyecto y el servidor de *backend* del proyecto. Basta con pasar a esta herramienta el enlace del repositorio y la rama del proyecto que se quiere desplegar.

  Otra alternativa muy popular es la herramienta Heroku [7], que funciona de la misma manera, pero requería introducir una tarjeta de crédito incluso para tener un plan gratuito.

- Flask [8]: biblioteca de Python orientada a proporcionar un servicio REST. De esta aplicación se utiliza para comunicar la parte *frontend* y *backend* de ésta. Con esta biblioteca se crea un servidor que actúa como intermediario entre los usuarios y la aplicación web, manejando las solicitudes de los usuarios y devolviendo a éstos las respuestas pertinentes.

  Existen otras bibliotecas como FastAPI [9] pero se decidió usar Flask porque es la más popular, por lo que hay más información online sobre posibles fallos, configuraciones, etc.

- React [10]: tecnología para el desarrollo de *frontend* basada en el uso de componentes reutilizables. Con esta tecnología se ha desarrollado el *frontend* de la aplicación, usando el lenguaje Typescript [11] y la biblioteca de diseño Ant Design [12].

- Python [13]: lenguaje de programación utilizado para el desarrollo del *backend* de la aplicación. Se decidió usar este lenguaje ya que las bibliotecas de AWS para implementar sus servicios, como SQS, están escritas en su mayoría para Python.

- Axios [14]: es la biblioteca que utiliza la aplicación para hacer las operaciones GET y POST que se envían al servidor de Flask.

- LocalStack [15]: es una herramienta con la cual se puede emular un servicio de AWS como si fuera local, de manera que las llamadas al api se hacen de la misma forma. Para este proyecto se utilizan para el almacenamiento de archivos mediante *buckets*. Localstack utiliza a su vez Docker [16], una herramienta que permite crear y ejecutar entornos aislados, conocidos como contenedores.

  Esta herramienta se utiliza para emular un servicio de AWS de almacenamiento persistente de archivos, denominado *bucket*, ya que el servicio de AWS requiere de una suscripción de pago.

- AWS Cognito [17]: es un API de Amazon Web Services que se utiliza para realizar la gestión de usuarios. Se ha decidido usar esta herramienta ya que, al usar otra herramienta de AWS para la cola de mensajes, es una buena idea utilizar servicios de la misma compañía.

  Esto es debido a que en caso de que se quisiera pagar por el servicio de cola de mensajes y el servicio de usuarios para aumentar sus prestaciones, como son del mismo servicio saldrían más baratos, ya que las compañías normalmente proporcionan suscripciones para poder mejorar un conjunto de sus servicios.

3.3 Técnicas utilizadas

Metodología SCRUM

Para el proyecto se usó la metodología SCRUM. Scrum es una metodología ágil que propone un desarrollo de manera iterativa. Las iteraciones son llamadas *sprints*, y éstas suelen tener una duración de entre una y cuatro semanas, con preferencia por *sprints* con poca duración.

En Scrum hay varios roles:

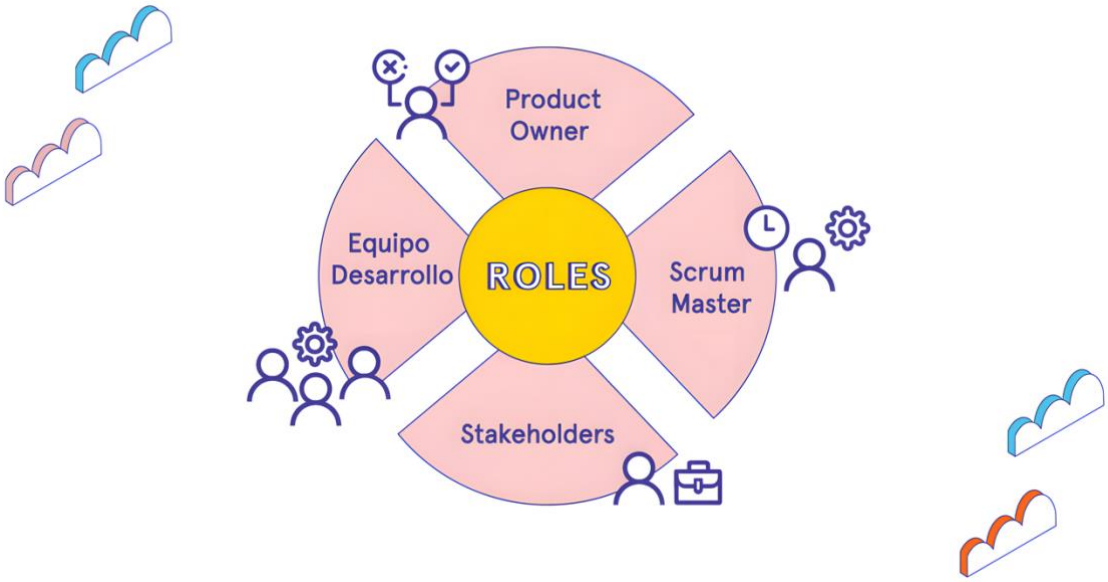


Figura 3: roles de Scrum. Fuente: https://www.lacreativeria.com/blog/los-roles-del-equipo-de-scrum

- El ***Product Owner*** es la persona que es dueña del proyecto, y, por tanto, decide el rumbo de éste. Es la persona que busca maximizar el valor del proyecto. Además, el *Product Owner* es la persona que se encarga de comunicarse con el cliente, y recoger las ideas de éste. Las ideas se recogen mediante historias de usuario.

- El ***Scrum Master*** es aquel que se encarga de coordinar y organizar al equipo de la mejor manera posible.

- El equipo de desarrollo es el conjunto de personas contratadas por el ***Product Owner*** que se encargan de desarrollar en proyecto de manera conjunta. Al final de cada sprint, el equipo de desarrollo entrega un incremento del producto.

- Por último, los ***Stakeholders*** son aquellas personas que, son ajenos al equipo del producto, pero que tienen intereses relacionados con el producto. Esto significa que el resultado del producto tiene una influencia en ellos, ya sea un beneficio o perjuicio económico o algún otro tipo de cambio.

Los resultados de aplicación de la metodología Scrum se denominan artefactos. Estos son tres:

- El ***Product Backlog*** es la lista de tareas o peticiones del proyecto. Pueden ser de diferentes tipos, como una tarea que consista en hacer una nueva funcionalidad o una petición que sea arreglar un fallo en la aplicación que se esté desarrollando.

- El **Sprint Backlog** es aquella lista de elementos sobre los que se trabaja en el sprint actual. Además, el equipo de desarrollo podrá visualizar en todo momento aquellas tareas que no han empezado a hacerse, aquellas que están en proceso y las personas asignadas para hacer estas tareas, y las tareas que ya están completadas.

- Por último, el **incremento** es el resultado del sprint, que será una versión del proyecto con el añadido de las tareas completadas del Sprint Backlog.

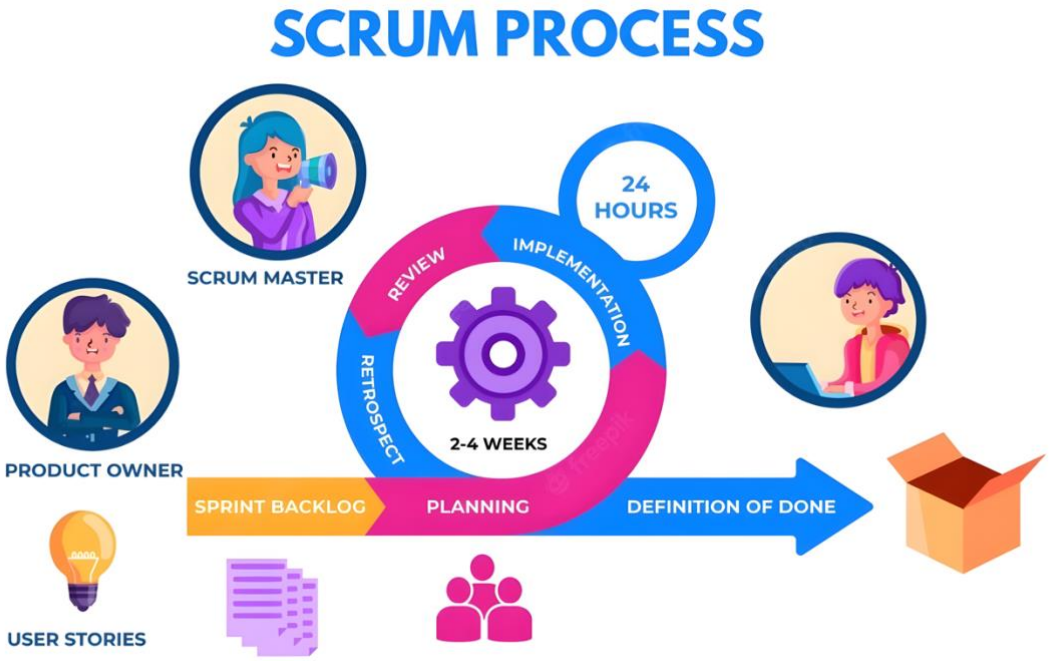


Figura 4: proceso Scrum. Fuente: https://www.cyberclick.es/numerical-blog/scrum-que-es-como-funciona-y-mejora-el-trabajo-en-equipo

En resumen, la metodología SCRUM es una de las metodologías ágiles más usadas, y que permite que al final de cada sprint se entregue un incremento respecto al sprint anterior.

Es un desarrollo en el que al final de cada entrega se añaden nuevas funcionalidades. A lo largo del sprint, además de las tareas ya propuestas se descubrían nuevas tareas que se podían realizar.

Al final de cada sprint, se hace una reunión con los tutores del proyecto, que son las personas más cercanas al concepto de cliente. En colaboración con ellos, se iban planeando las tareas a realizar para el siguiente sprint, y se planeaba el rumbo de la aplicación.

Integración continua (CI/CD)

El concepto de integración continua (CI/CD) trata sobre como en proyectos de gran escala, en los que trabaja mucha gente, los cambios que se hacen sobre este son analizados, de manera que el código pasa por varias fases: compilación del código, ejecución de pruebas unitarias y de integración y análisis de calidad del código.

Para llevar esto a cabo, hay herramientas como Jenkins [18] que compilan y construyen el código y que pueden pasar las pruebas.

4. Aspectos relevantes sobre el desarrollo del proyecto

En este apartado se exponen los aspectos relevantes del proyecto, donde se analizarán y discutirán las diferentes partes que componen la aplicación en cuestión.

4.1 Uso de SCRUM en el proyecto

Para poder explicar bien cómo se ha abordado la realización de este proyecto, lo primero que se va a explicar es cómo se ha aplicado la metodología SCRUM a este proyecto.

Los roles que se han desempeñado en el proyecto son:

El alumno del proyecto ha asumido los roles de Equipo de Desarrollo y *Product Owner*, ya que es el único desarrollador del proyecto y tiene el control directo sobre la construcción y entrega del producto (Equipo de Desarrollo), y es responsable de definir los requisitos y funcionalidad del software, priorizando además las tareas más importantes a realizar.

Por otro lado, mis tutores desempeñan el rol de *Stakeholders*, ya que son personas interesadas en el proyecto, participando en reuniones de realimentación o siendo supervisores académicos.

En cuanto a los *sprints*, en este proyecto se han ido realizando *sprints* de una duración media de dos semanas. Al final de cada sprint, se ha ido haciendo una entrega del proyecto que propone un incremento en cuanto a funcionalidad.

Al final, se ha tenido una reunión con el tutor de la empresa para evaluar el estado del proyecto, proponer nuevas tareas a realizar para incluirlas en el *Product Backlog* y definir el rumbo del proyecto.

4.2 Uso de la integración continua

En este proyecto, se ha creado un archivo en el repositorio en el cual, si un cambio realizado pasa por todas las tareas que se proponen de manera correcta, este cambio se desplegará automáticamente. Estos pasos son:

- Se instalan todas las dependencias del proyecto de manera adecuada. Estas son todas las bibliotecas usadas para el desarrollo de la aplicación.
- Se realizan todas las pruebas unitarias que se hayan escrito.
- Si ha pasado todas las pruebas, en caso de que la rama en la que se haya hecho el cambio sea la rama que está desplegada en la nube, se despliega el nuevo cambio realizado.

En el caso de este proyecto, la rama *main* es la rama que se despliega, por lo que un cambio en esta rama (*commit*) tendrá que pasar por la tubería y pasar las pruebas unitarias. En caso de que pase estas pruebas, se desplegará la aplicación en la plataforma Render.

4.3 Arquitectura del proyecto

En cuanto a la arquitectura del proyecto, desde el primer momento se planteó como una arquitectura cliente-servidor, donde varios clientes se comunican con varios servidores para que la aplicación funcione correctamente.

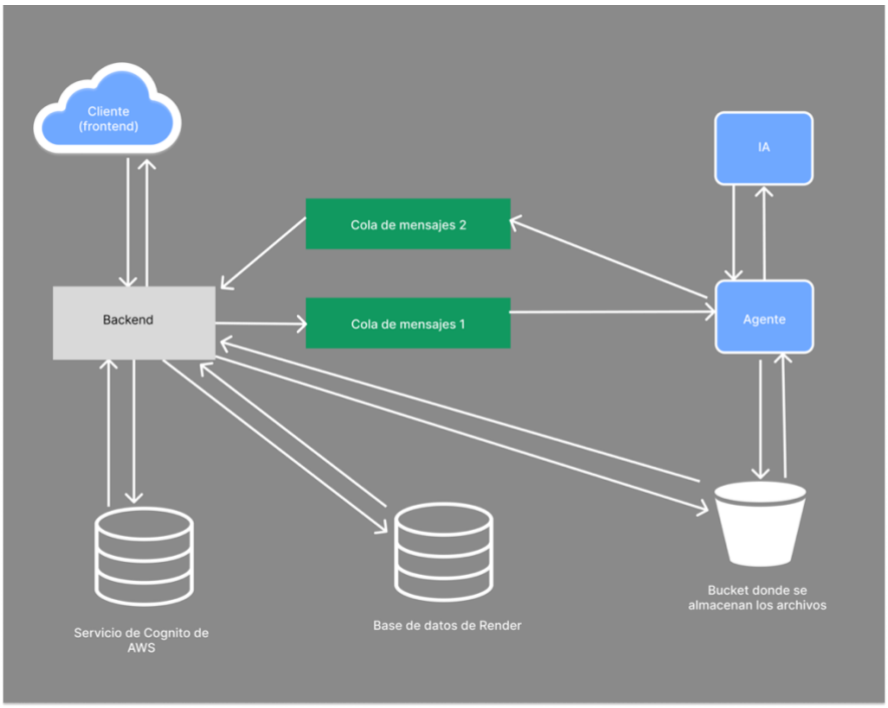


Figura 5: esquema de comunicación entre los diferentes componentes de la aplicación

Se tiene:

- Un servidor Flask que actúa como API REST para comunicar frontend y backend a través de operaciones GET y POST.
- Servidores de Cognito para la autenticación de usuarios.
- Una base de datos alojada en un servidor, que almacenará de manera persistente la información relativa a las peticiones de los usuarios.

- Un agente que ejecuta los modelos de IA, que se comunica con el *backend* a través de mensajes.
- Un servidor (*bucket* local) donde están almacenados los archivos de manera persistente.

El uso de una arquitectura diferente como una arquitectura P2P no tiene sentido en esta aplicación, ya que las funciones que tiene esta aplicación como el servicio de usuarios por lo general se implementan con servicios que usan esta arquitectura. Además, se prevé el número de usuarios de la aplicación será reducido, por lo que no tiene sentido utilizar una arquitectura P2P, cuya potencia se sostiene en número de usuarios considerable.

No obstante, el uso de una arquitectura cliente-servidor tiene algunas desventajas como los cuellos de botella que se pueden ocasionar en los servidores en caso de que haya muchas peticiones entrantes.

Sin embargo, en caso de que el número de usuarios de esta aplicación escalara, se podría hacer una replicación de los agentes con balance de carga, o replicar el servidor de Flask o de base de datos. De esta manera, las prestaciones de la aplicación podrían aumentar. Más adelante se verá cómo se llegó a este esquema de comunicación.

4.4 Desarrollo del frontend y backend

Durante la primera fase del desarrollo del proyecto, se debía tomar una decisión sobre qué camino tomar en relación a cómo desarrollar el *frontend* y *backend* del proyecto:

- Desarrollar un *frontend* y un *backend* completamente separados el uno del otro y comunicarlos posteriormente mediante el uso de *APIs*. Esto permite hacer una interfaz de usuario completamente personalizada y medida. A cambio, la complejidad para hacerlo es mucho mayor, especialmente teniendo en cuenta que la experiencia desarrollando una interfaz de usuario es nula. Por lo tanto, el tiempo de construcción sería elevado.

- Usar una biblioteca que permita escribir un *frontend* de manera sencilla. La principal ventaja es el tiempo de desarrollo, que es mucho más bajo al tratarse de algo mucho más sencillo. Como contra, el número de opciones es mucho más limitado, no pudiendo desarrollar el *frontend* con total libertad.

En este caso, se decidió optar por la segunda opción, ya que, aunque no se tenía total flexibilidad, crear una interfaz de usuario era una tarea mucho más sencilla, y menos costosa. De esta manera, se podría dedicar el tiempo de desarrollo que se perdería con la primera opción en incluir más funcionalidades. Por otro lado, en el que caso de que sobrara tiempo, se podría hacer la primera opción.

Habiendo tomado esta decisión, se investigaron las herramientas Streamlit [19] y Gradio [20], y se decantó por usar Streamlit, un *framework* de código abierto para Python que permite elaborar interfaces de usuario muy visuales con pocas líneas de código. Con ello, no se necesita crear un *frontend* y *backend* al uso, si no que se puede asociar código de Python a cualquier elemento de interacción del usuario.

Esta biblioteca está basada en el uso de componentes (widgets), que prácticamente se declaran de la misma manera que las variables. La herramienta es muy potente, sobre todo en el ámbito de la visualización de datos.

Sin embargo, a medida que avanzaba el desarrollo del proyecto, el proceso de implementar nuevas funcionalidades se hacía cada vez más complejo en el apartado del desarrollo del *frontend* en sí. Y es que Streamlit estaba dando muchos problemas para implementar nuevas opciones que requerían de la aparición/ocultamiento de nuevos componentes como botones, o campos de entrada de texto. Además, la navegación entre las diferentes ventanas era sumamente compleja de implementar mediante unas variables de estado.

Es por ello por lo que finalmente, se hizo una segunda versión en la que se hizo un *frontend* a medida para la aplicación.

Tras la investigación de las diferentes bibliotecas y herramientas disponibles, las tecnologías que se usarán en esta nueva versión serán las siguientes:

- Uso de React, que es una biblioteca de código abierto para construir interfaces interactivas y reactivas. Gran parte de su éxito reside en que esta biblioteca se basa en el uso de componentes que son reutilizables.

- Uso de Typescript como lenguaje de programación. Con React se puede utilizar Javascript o Typescript como lenguaje de programación. El tutor de HP indicó que Typescript era algo más fácil para aprender al ser un lenguaje tipado.

- Ant Design como lenguaje de diseño para el *frontend*. En lugar de realizar el *frontend* completamente a mano, se usará la Ant Design y sus componentes para aportar una estructura de diseño consistente. Las razones para usar Ant Design fueron dos:

    - Por un lado, es un lenguaje de diseño muy popular.
    - Por otro lado, su lenguaje de diseño está basado en Material Design [21], que es el lenguaje predecesor de Material You, el lenguaje de diseño que se usó para realizar el prototipo digital.

- Flask como API REST para comunicar *backend* y *frontend*. Esta es la principal diferencia con respecto a la versión anterior, ya que ahora si hay un *frontend* y un *backend* separados por completo, y, por tanto, necesitamos un servidor para comunicar ambas partes.

  Se decidió utilizar Flask ya que es el *framework* más usado para *backend* escrito en Python.

- Axios, una biblioteca que se usa para hacer las operaciones GET y POST, y que permite realizar las llamadas al servidor de Flask, comunicando así la parte *frontend* y la parte *backend* de la aplicación.

- Node.js [22] como entorno de ejecución del *frontend*.

Tras haberse avanzado bastante en la implementación, a la hora de probar este nuevo *frontend*, la diferencia en términos de velocidad respecto a la versión anterior era notoria.

4.5 Sistema de cola de mensajes

Para el sistema de colas de mensajes, se ha utilizado SQS de Amazon AWS. El API para utilizar este servicio pertenece a la biblioteca boto [23] de Python.

Como se mencionó anteriormente, AWS tiene dos tipos de colas: las normales y las FIFO. En este caso se decantó por utilizar colas normales, ya que el orden de entrega no tiene demasiada relevancia en esta aplicación.

Como se mencionó también en el esquema del proyecto inicial (véase apartado 3.1), se tiene una única cola de mensajes que se encargaba de comunicar a los distintos clientes con el agente, y viceversa.

Sin embargo, al hacer distintas pruebas, se comprobó que la recepción de los mensajes era muy lenta. Esto en parte se debe a que, en esta versión, tanto el agente como el *backend* hacen sondeos de la misma cola para recibir mensajes cada muy poco tiempo, creando un cuello de botella en el servidor de SQS. De esta manera, el servidor no puede atender a todas las peticiones del *backend* y del agente, y, por tanto, no puede proporcionarles los mensajes.

Por lo tanto, para mejorar el tiempo de recepción, se decidió usar dos colas de mensajes en lugar de una, como se muestra en el siguiente esquema:

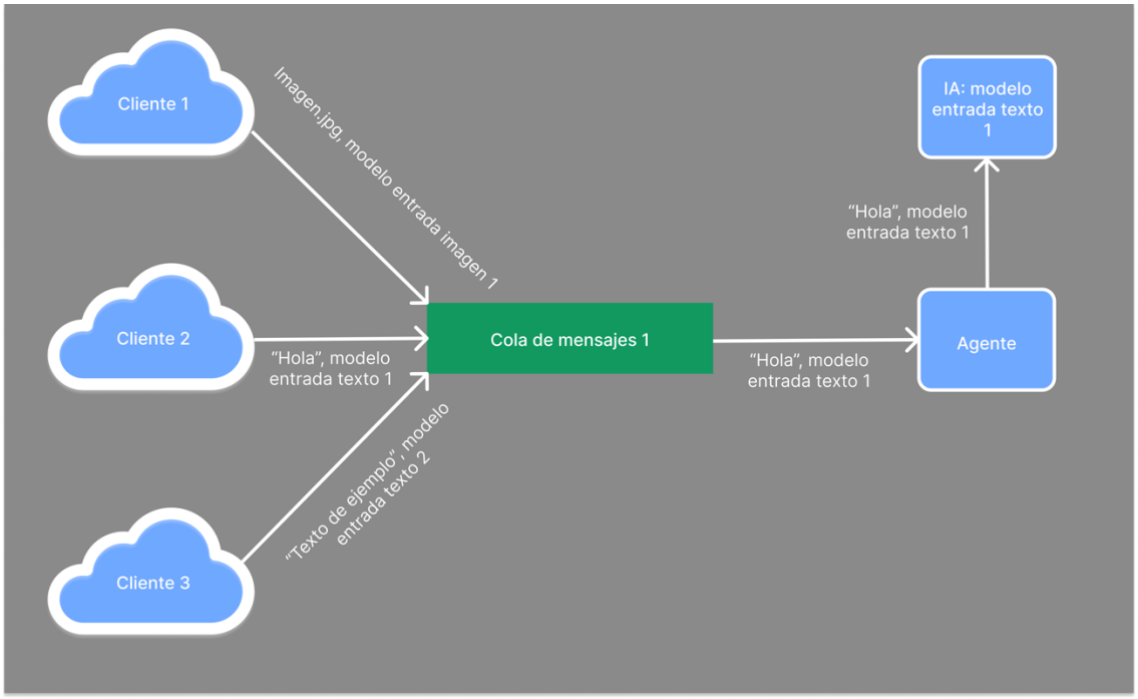


Figura 6: nuevo esquema de mensajes (cola de mensajes 1)

El envío de mensajes de parte de los clientes que contiene las peticiones se hará en la cola de mensajes 1, al igual que antes.

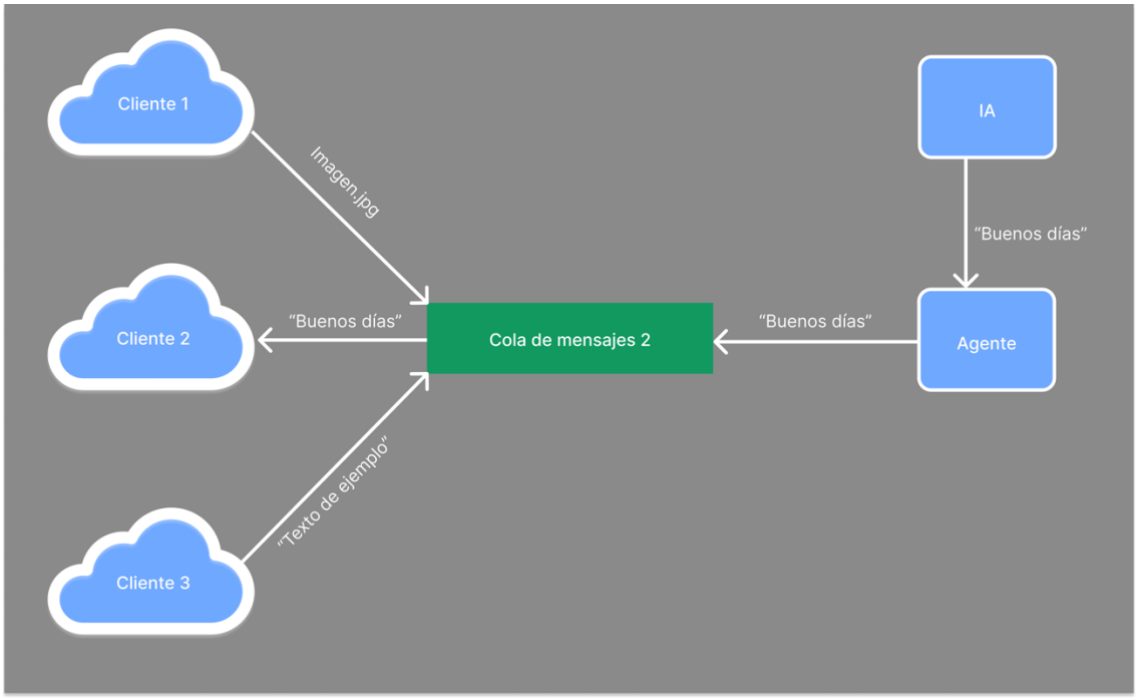


Figura 7: nuevo esquema de mensajes (cola de mensajes 2)

Sin embargo, el cambio reside en que el mensaje que contiene el resultado de la petición será enviado por el agente en la nueva cola de mensajes, de la que harán un sondeo los distintos clientes.

Con este cambio, el agente hará un sondeo sobre la cola de mensajes 1, y los clientes sobre la cola de mensajes 2. Por otro lado, los clientes enviarán mensajes a la cola de mensajes 1 y el agente enviará mensajes a la cola de mensajes 2.

Ahora, los métodos que se encargan de enviar un mensaje a la cola de mensajes y de recibir mensajes de la cola de mensajes tendrán un nuevo parámetro, que será la cola a la que hace referencia.

Adicionalmente, se aumentó el tiempo de sondeo de la aplicación para que se haga cada 5 segundos, y en caso de que no se encuentren mensajes, adicionalmente se creará otro tiempo de espera de 10 segundos.

Cada mensaje tiene los siguientes atributos:

- ID: identificador del mensaje. Los mensajes de SQS tienen un identificador único como atributo predeterminado. En este caso, se ha creado otro identificador, que no será único, y que servirá para realizar la comunicación.

  En este caso, el cliente envía el mensaje a la cola de mensajes 1 con un ID. El agente, una vez reciba este mensaje y procese el resultado, enviará un mensaje a la cola de mensajes 2 con ese mismo ID.

  De esta manera, el cliente se queda esperando a recibir el mensaje con el resultado de la petición de la cola de mensajes 2 que tenga el identificador del primer mensaje que envío.

  Resumiendo, es una forma de que el cliente sepa cuál es el mensaje que contiene el resultado de su petición.

- *Model*: nombre del modelo de *Machine Learning* que selecciona el usuario.

- *Time*: la fecha y hora en la que se envía el mensaje.

- *URL*: contiene una URL a un archivo almacenado en el *bucket*.

4.6 Tipos de datos de los mensajes

En las primeras versiones del trabajo, el texto o imagen que introduce el usuario se envía dentro de un mensaje. Meter el texto es realmente fácil, ya que es una cadena de texto. Sin embargo, la imagen no se puede enviar como tal. Por ello, se decidió enviar texto e imagen convertidos a base64. Base64 es sistema de codificación que permite convertir datos en binario (como una imagen) en forma de texto, utilizando letras mayúsculas,

minúsculas, números y algunos símbolos especiales. Además, esto un proceso reversible (se puede convertir una cadena de base64 a binario), permitiendo que el *backend* de la aplicación envíe en un mensaje la cadena en base64 que represente el texto o la imagen, y el agente, que es el receptor del mensaje, obtenga el contenido original en binario.

4.7 Limitaciones

El servicio de cola de mensajes de Amazon en su plan gratuito tiene una gran limitación en cuanto al tamaño de los mensajes, siendo el máximo 256 kb. Esto implica que, a la hora de enviar imágenes, el tamaño de éstas deberá de ser muy pequeño para que esto funcione. En el siguiente apartado se hablará de la solución a este problema.

4.8 Persistencia de la información

Persistencia de los archivos

Como se ha mencionado en el apartado anterior, incluir el archivo de la entrada del usuario en el contenido de los mensajes estaba muy limitado, ya que el tamaño de los mensajes es de 256kb.

Para solucionar este problema, existe una biblioteca denominada *SQS Extended Library [24]*. Sin embargo, esta biblioteca no se ha usado para el desarrollar la aplicación, ya que, para poder funcionar, esta biblioteca requiere de un *bucket* de Amazon, y este servicio no es gratuito.

Un *bucket* es un servicio que ofrece Amazon que es un objeto de almacenamiento en la nube. Se podría decir que es un sistema parecido a plataformas de almacenamiento en la nube como Google Drive [25] o Microsoft OneDrive [26], pero orientado a otro tipo de usos.

Un *bucket* se utiliza para almacenar cantidades de datos muy grandes y está más enfocado en la seguridad y la escalabilidad, mientras que las plataformas anteriores están enfocadas en almacenar archivos personales y compartir archivos.

Como este servicio no se ha usado debido a que es de pago, se intentó pensar y buscar otras alternativas. Una alternativa que se pensó fue fragmentar los mensajes en partes de 256 kB, pero se descartó ya que los tiempos de recepción de mensajes por parte del agente aumentaban exponencialmente cuanto mayor era el tamaño del archivo que se quería enviar.

Como solución a este problema, la aplicación usa la plataforma LocalStack, que permite emular de manera gratuita un *bucket* de manera local, usando el mismo código que con un bucket en la nube.
Por lo tanto, mediante el uso de esta plataforma, se pueden almacenar archivos de cualquier tamaño, y no se necesita enviar el archivo como tal en el mensaje, sino la URL

donde está el archivo, resolviendo de esta manera el problema de la limitación del tamaño de los mensajes.

4.9 Persistencia de la información de las peticiones

La necesidad de la persistencia de la información surge cuando un usuario de la aplicación quiere poder acceder en cualquier momento a la información relacionada con sus peticiones anteriores.

Para ello, se usará la plataforma Render, que permite tener en la nube un sistema gestor de base de datos PostgreSQL [27]. Para conectarse a la base de datos se usará la biblioteca de Python Psycopg2 [28].

En cuanto a la información almacenada en base de datos, la aplicación no necesita más que una tabla, que está orientada a que la información de las peticiones de los usuarios sea persistente para que estos puedan acceder a ella en todo momento.

En la única tabla existente, que almacena la información relativa a las peticiones, se tienen los siguientes campos:

- Identificador de la petición: es la clave primaria de la tabla.
- Identificador de usuario: necesario para la lógica de la aplicación, de manera que cuando un usuario quiera ver sus peticiones, se recojan las filas de la tabla cuyo campo identificador de usuario coincida con el identificador del usuario.
- Fecha de la petición: necesario para mostrar los resultados en orden descendente.
- URL al archivo de entrada y URL al archivo de salida: necesarios para acceder al *bucket* donde están almacenados los archivos de manera persistente.

4.10 Gestión de usuarios

Esta necesidad surge como complemento a la implementación de un método para la persistencia de la información. Mediante la gestión de usuarios, se puede hacer que cada usuario pueda crear peticiones y ver los resultados de sus propias peticiones.

Inicialmente se hizo con el servicio Firebase [29] de Google, pero posteriormente se cambió por Cognito, que es un servicio de AWS. Ambos son servicios para gestionar usuarios que funcionan de manera similar, pero ha decidido usar esta herramienta ya que, al usar otra herramienta de AWS para la cola de mensajes, es una buena idea utilizar servicios de la misma compañía.

Esto es debido a que en caso de que se quisiera pagar por el servicio de cola de mensajes y el servicio de usuarios para aumentar sus prestaciones, como son del mismo servicio saldrían más baratos, ya que las compañías normalmente proporcionan suscripciones para poder mejorar un conjunto de sus servicios.

El almacenamiento de la información podría haber sido más complejo en caso de que se hubiera decidido por tratar el apartado de los usuarios de manera manual, pero en lugar de eso, se utiliza el servicio Cognito, ya que ofrece un mayor grado de seguridad, es más escalable y la implementación es más simple.

Por lo tanto, existen tres tipos de información a la que el cliente accede: la relativa a usuarios para autenticarse, la información de las peticiones, guardada en una base de datos que aloja la plataforma Render, y el *bucket* que almacena los archivos relativos a las peticiones.

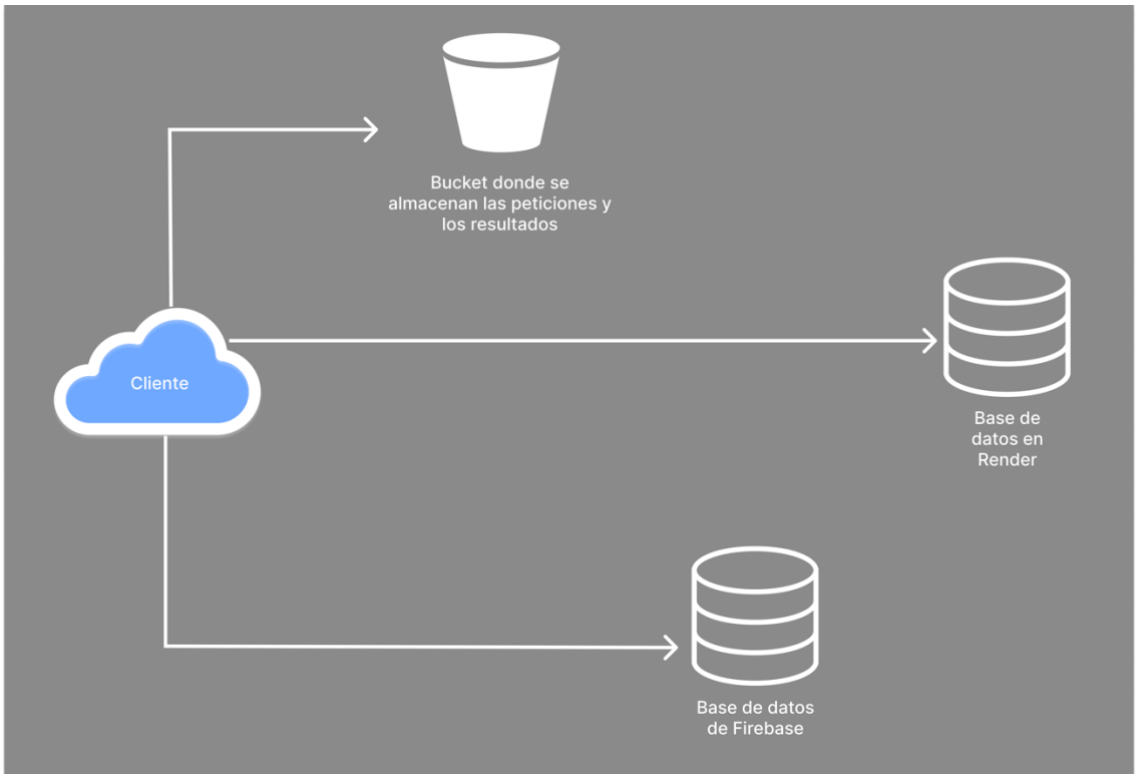


Figura 8: comunicación entre el cliente con el resto de los componentes.

4.11 Diseño del agente

El agente que se ha diseñado constará de dos hilos: un hilo se encargará de recibir mensajes de la cola 1 y procesarlos, para después enviar los resultados a la cola 2. Por otro lado, el otro hilo se encargará de eliminar mensajes antiguos y se ejecutará cada media hora, el resto del tiempo permanecerá dormido.

Esto parece relativamente una tarea sencilla, pero hay un problema. Ambos hilos harían un sondeo de la misma cola de mensajes. Además, si en un hipotético caso, la tarea principal obtiene un mensaje, y justo la otra tarea lo elimina, podría haber un problema, ya que están accediendo al mismo recurso.

Es por ello por lo que se ha implementado un semáforo para que solo haya un hilo sondeando la cola de mensajes al mismo tiempo, solucionando el problema mencionado.

4.12 Estrategias para mejorar la mantenibilidad del código

Para mejorar la mantenibilidad del código, se han implementado las siguientes estrategias:

- Encapsulación de las operaciones GET y POST en funciones externas. Esto implica que en el caso de que se quiera realizar un cambio en la configuración de estas operaciones, solo hay que modificar un archivo, que es el que contiene estas funciones.

- Dentro del agente, se ha creado un archivo de configuración en el que se defina el tipo de entrada y salida del modelo, denominado “config.json”. De esta manera, para añadir nuevos modelos al agente solamente se necesita añadirlos al archivo de configuración, e incluir el modelo nuevo, sin necesidad de crear nuevo código

  Con este sistema, cuando el usuario acceda al desplegable donde puede seleccionar uno de los modelos disponibles, esté desplegable mostrará los modelos del archivo de configuración, de manera que si se hace un cambio en este archivo de configuración se mostrará inmediatamente reflejado en la aplicación.

- Uso de variables de entorno para almacenar la información sensible, como puede ser la URL de la base de datos o la URL de las colas de mensajes. No sólo es necesario para impedir que cualquier persona tenga acceso a estos datos, si no que modificar el valor de estas variables es muy sencillo. De esta manera, se evitan hacer modificaciones en el código en caso de que se necesite modificar el valor de una variable de entorno.

4.13 Estructura de ficheros de la aplicación

La versión final de la aplicación tiene la siguiente estructura de ficheros:

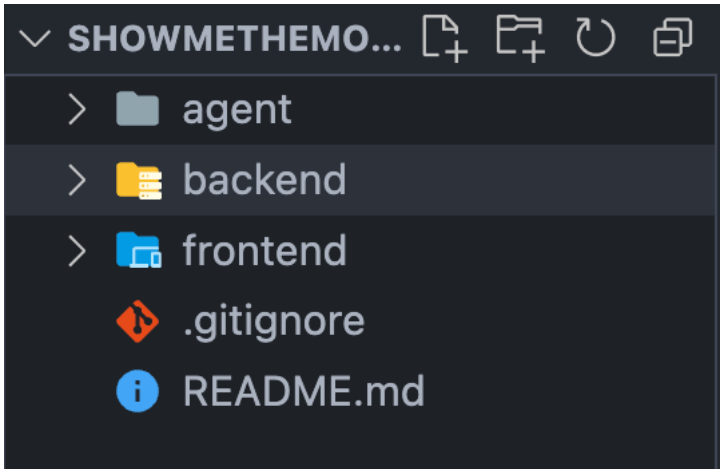


Figura 9: estructura general de las carpetas del proyecto.

Se tienen tres módulos completamente separados: el agente, el *backend* y el *frontend*. Estos podrían estar en tres repositorios directamente separados entre ellos.

Estructura del backend

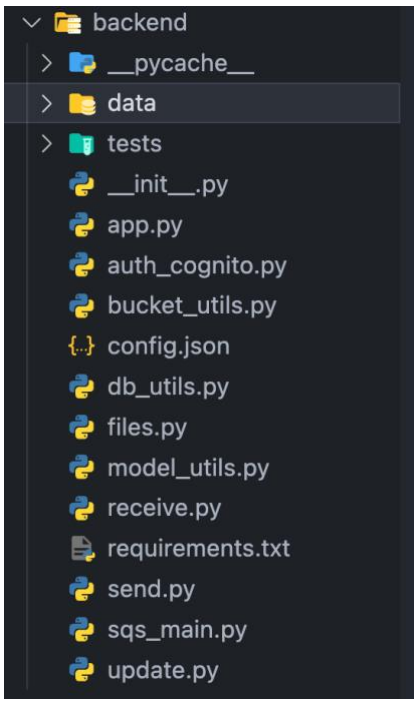


Figura 10: estructura de archivos de la carpeta backend.

El directorio *backend* tiene, a su vez, dos carpetas:

- La carpeta data almacena los archivos que se descargan del *bucket*.
- La carpeta test almacena las pruebas unitarias de este modulo

El resto de los archivos contienen funcionalidad referente al *backend*: por ejemplo, db_utils.py se encarga de realizar la conexión con base de datos, app.py es el servidor de Flask o *files*, que se encarga de guardar/eliminar/modificar los archivos.

Estructura del frontend

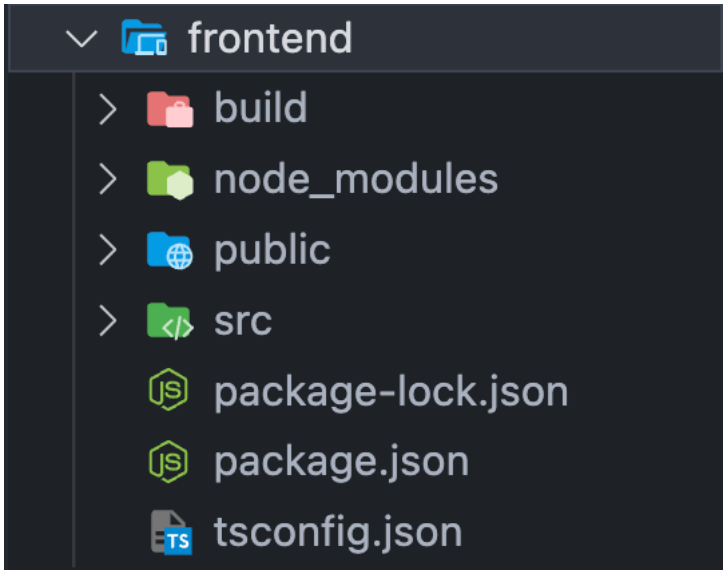


Figura 11: : estructura de archivos de la carpeta frontend.

Esta es la estructura de casi cualquier proyecto de *frontend* que utilize React. Lo importante se encuentra en el directorio src:

Figura 12: estructura de archivos de la carpeta src.

Dentro de la carpeta src, se tiene la carpeta componentes, donde están los componentes que forman la aplicación. Cada componente es una ventana de ésta. En la carpeta /Api existe un componente en el que se encuentran las funciones que realizan las llamadas GET y POST.

Estructura del agente

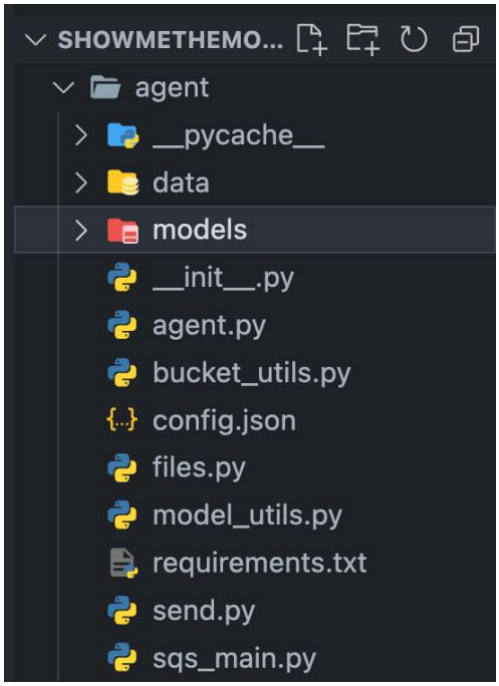


Figura 13: estructura de archivos de la carpeta agente.

El agente no tiene funcionalidad nueva respecto al *backend*, sino que reutiliza todas las funciones del *backend* que necesita. En cuanto a las carpetas:

- Tiene una carpeta data, donde almacena las entradas de las peticiones de los clientes para poder procesarlas.
- La carpeta *models* contiene los modelos que puede ejecutar el agente. Estos son archivos .py.
- El archivo "config.json" es el archivo que contiene la información de los modelos, como se mencionó en el apartado anterior, cuando el usuario acceda al desplegable donde puede seleccionar uno de los modelos disponibles, esté desplegable mostrará los modelos del archivo de configuración.

En el anexo 4 se explica el funcionamiento de la aplicación con más detalle.

4.14 Configuración del pipeline

Como se mencionó anteriormente, se puede configurar una tubería por la que el código pasará por unos pasos, y si pasa por todos sin problema se desplegará el código. La configuración de una tubería depende del repositorio de software utilizado. En este caso, al utilizar Gitlab como repositorio software se deberá de crear un archivo denominado "gitlab-ci.yml" en la carpeta raíz del repositorio.

Como se vio anteriormente, la plataforma Render es la que se encarga de desplegar la aplicación. Cuando el usuario de Render proporciona el enlace su repositorio, y escoja la rama que quiera desplegar, Render despliega la aplicación con los nuevos cambios nada más se haga un *commit* en el código del repositorio.

Es decir, si se le ha dicho a Render que despliegue la rama "main" de nuestro repositorio, cada *commit* que se haga en la rama *main* se despliega automáticamente. En este caso, se va a cambiar esto en los ajustes, de manera que Render no despliegue la aplicación automáticamente, y que en su lugar esté controlado por nosotros.

Para ello, Render proporciona al usuario una URL con la cual cada vez que se acceda a ella, se despliega la aplicación con los últimos cambios.

Por lo tanto, el archivo "gitlab-ci.yml" está compuesto por tres fases:

- La fase inicial, donde se descargan todas las dependencias que usa el proyecto, es decir, todas las bibliotecas. Estas estarán en un archivo denominado "requirements.txt", situado en el directorio raíz del proyecto.

- La fase de prueba, que se ejecuta siempre que se haga un *commit* de cualquier rama. Esta fase ejecuta las pruebas unitarias, y, en caso de que las pase, se pasa a la siguiente fase.

- La fase de despliegue. Esta fase solo se ejecuta en caso de que la rama en la que se haya hecho el cambio sea la rama principal, que es la que Render despliega. Se dirá al pipeline que se acceda a la URL correspondiente y Render desplegará la aplicación con los cambios nuevos.

Por último, hay que tener en cuenta que el proyecto que se ha desarrollado utiliza variables de entorno. Estas variables de entorno tienen como objetivo guardar información sensible de manera que esté protegida ante accesos no autorizados. En el caso de esta aplicación, existen variables de entorno que almacenan información sensible como la URL de la base de datos que se utiliza en el proyecto o la URL que se utiliza para desplegar la aplicación.

Por lo tanto, para que el despliegue funcione, en Gitlab, se tienen que introducir estas mismas variables de entorno. Esto se puede hacer en la configuración del proyecto, si el usuario de Gitlab tiene permisos suficientes en el repositorio.

El archivo tiene el siguiente estilo:

```yaml
image: python:latest

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

stages:
  - test
  - deploy

cache:
  paths:
    - .cache/pip
    - venv/

before_script:
  - pip install virtualenv
  - virtualenv myenv
  - source myenv/bin/activate
  - pip install -r requirements.txt
  - export SECRET_KEY=$SECRET_KEY
  - export SECRET_ACCESS_KEY=$SECRET_ACCESS_KEY
  - export SECRET_URL=$SECRET_URL
  - export SECRET_URL2=$SECRET_URL2
  - export BUCKET_URL=$BUCKET_URL
  - export BUCKET_NAME=$BUCKET_NAME
  - export REGION_NAME=$REGION_NAME
  - export USER_POOL_ID=$USER_POOL_ID
  - export CLIENT_ID=$CLIENT_ID
  - export CLIENT_SECRET=$CLIENT_SECRET
  - export DOMAIN=$DOMAIN
  - export DBNAME=$DBNAME
  - export USER=$USER
  - export PASSWORD=$PASSWORD
  - export HOST=$HOST
  - export POST=$POST

test:
  stage: test
  script:
    - python -m unittest

deploy_job:
  stage: deploy
  only:
    - main
  script:
    - curl -s "${RENDER_URL1}?${RENDER_URL2}"
```

Figura 14: vista del archivo de configuración de la tubería

4.15 Despliegue de la aplicación

Mediante el despliegue de la aplicación se puede hacer que ésta sea accesible para todo el mundo. Como se ha mencionado anteriormente, este proyecto se apoya en varios servicios para funcionar, y mediante el uso de los planes gratuitos que éstos ofrecen, se ha podido desplegar la base de datos, el sistema de colas de mensajes, el *backend* y el *frontend*.

Por tanto, lo único que no se ha podido desplegar es el *bucket* para almacenar los archivos. Para ello habría que pagar una suscripción de AWS, y ya tendríamos la

aplicación desplegada. No hay que rescribir nada de código respecto al uso del *bucket*. Solamente habría que cambiar la URL del *bucket*, que cambiará de localhost a otra URL.

Con el resto de los planes gratuitos, la aplicación podría funcionar para un número significativo de usuarios.

Al final, el grado de escalabilidad de la aplicación va a depender de los planes de los servicios que se usan. Por lo tanto, para un mayor número de usuarios se necesitará usar planes más caros.

En el anexo 1 se explica cómo podríamos abordar económicamente esta situación.

5. Descripción del producto final

Se va a explicar el funcionamiento de la aplicación. En caso de que la aplicación estuviera desplegada al completo, se accederá a la URL de la aplicación, que llevará al usuario a la pantalla de inicio de sesión:

Figura 15: ventana de inicio de sesión

Si el usuario no tiene una cuenta de usuario, pulsará en el botón “regístrese ahora”, que redirigirá al usuario a la pantalla de registro:

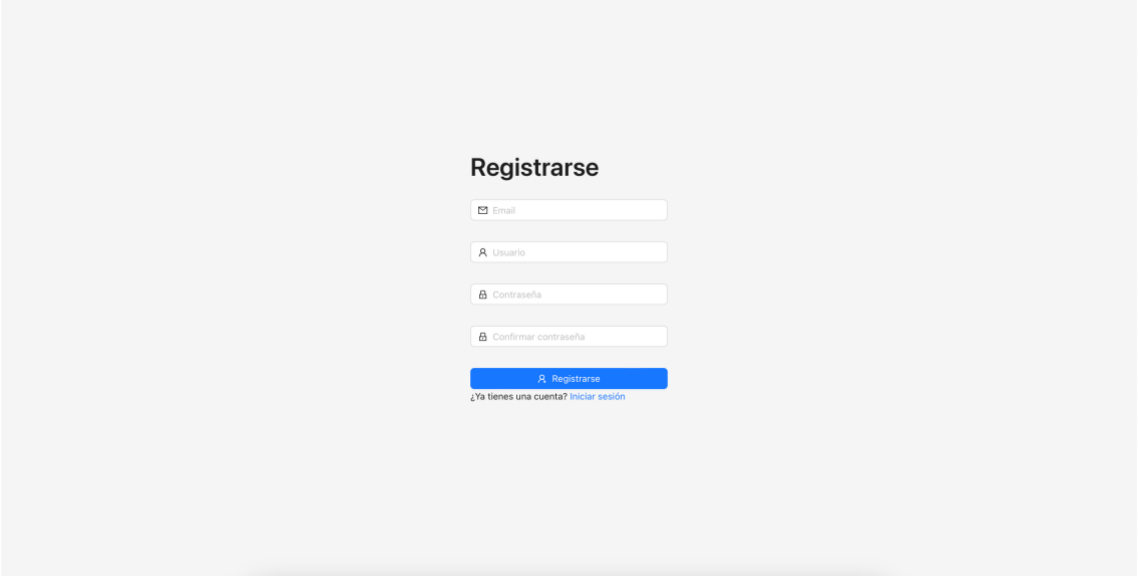


Figura 16: ventana de registro de la aplicación

Cuando el usuario introduzca las credenciales pertinentes, aparecerá la siguiente notificación y se redirige al usuario al inicio de sesión:

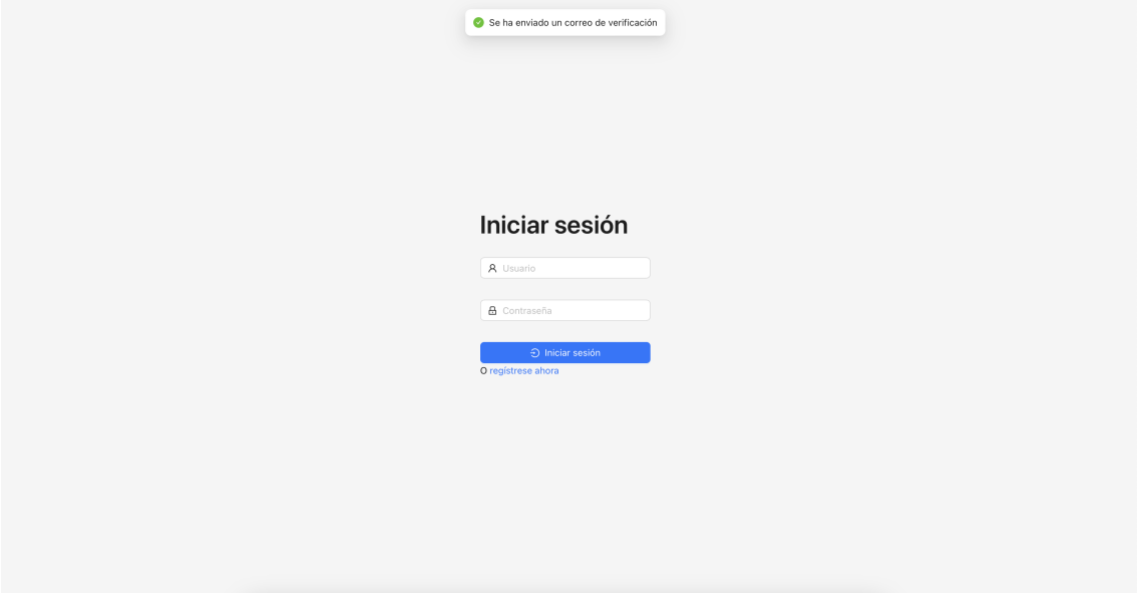


Figura 17: notificación de correo de verificación

Se enviará el siguiente correo de verificación:

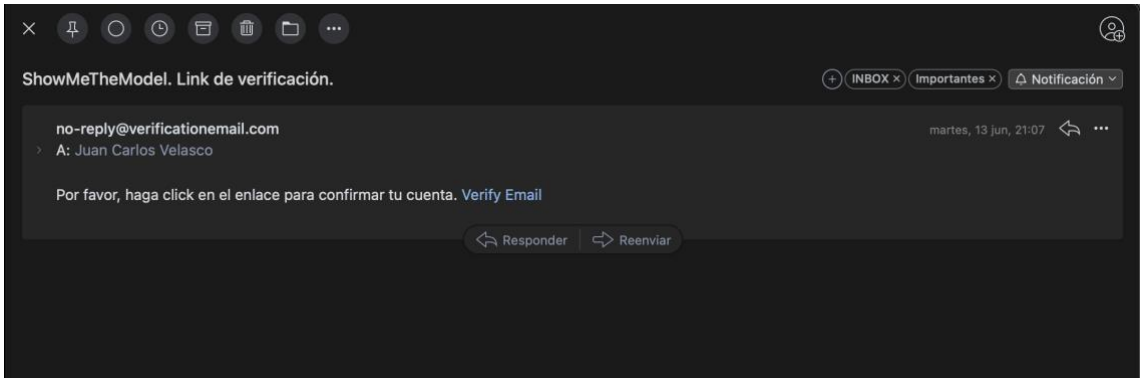


Figura 18: correo de verificación de usuario

Una vez se pulse en "*Verify Email*" saldrá la siguiente ventana en el navegador:

Figura 26: ventana de verificación de cuenta.

Ahora este usuario, como ha creado una cuenta, podrá acceder a la aplicación.

Si el usuario ya tiene una cuenta, se introducirán las credenciales de usuario, y se accederá a la pantalla principal.

Lo que ocurre por debajo es que, al pulsar el botón de iniciar sesión, los campos introducidos al usuario se enviarán en una operación POST al *backend,* donde se llamará al servicio de Cognito para intentar iniciar sesión con esos campos, y en caso afirmativo, reenviar al usuario a la página principal.

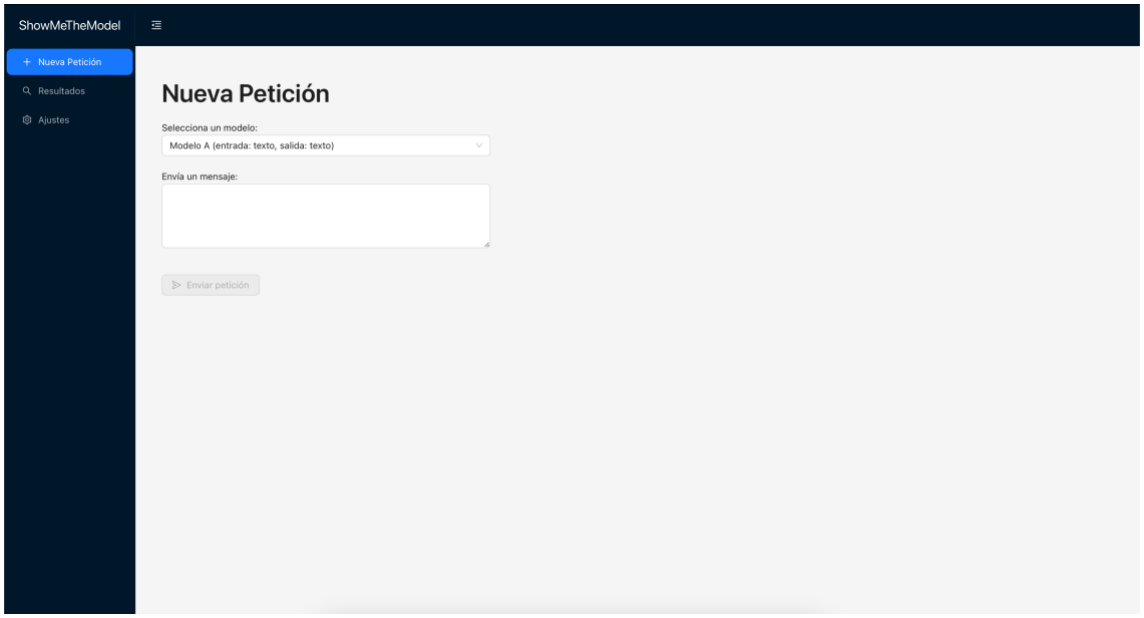


Figura 19: página principal de la aplicación

En la página principal de la aplicación se tienen tres pestañas, “Nueva Petición”, “Resultados” y “Ajustes”.

En la pestaña de “Nueva Petición”, se tiene un desplegable, directamente conectado al archivo llamado “config.json” que se mencionó al hablar de la estructura del agente. El *frontend* solicitará al *backend* mediante una operación GET este archivo de configuración. En este caso el *frontend* mostrará el atributo “descripción” de cada modelo.

Como se puede ver, la información del desplegable coincide con la del archivo de configuración:

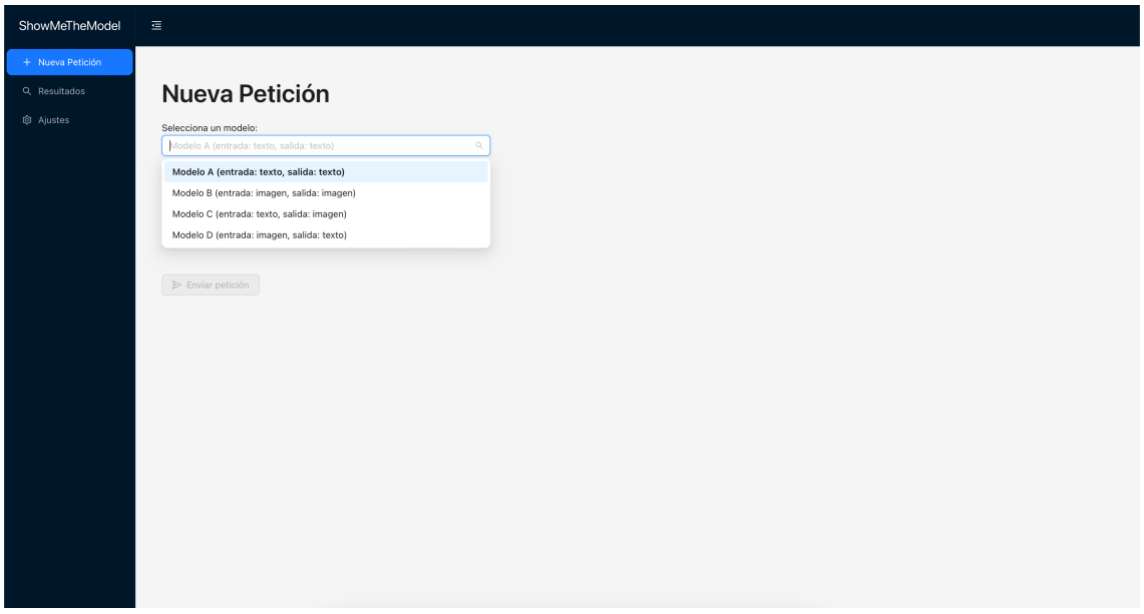


Figura 20: ventana de Nueva Petición (desplegable abierto)

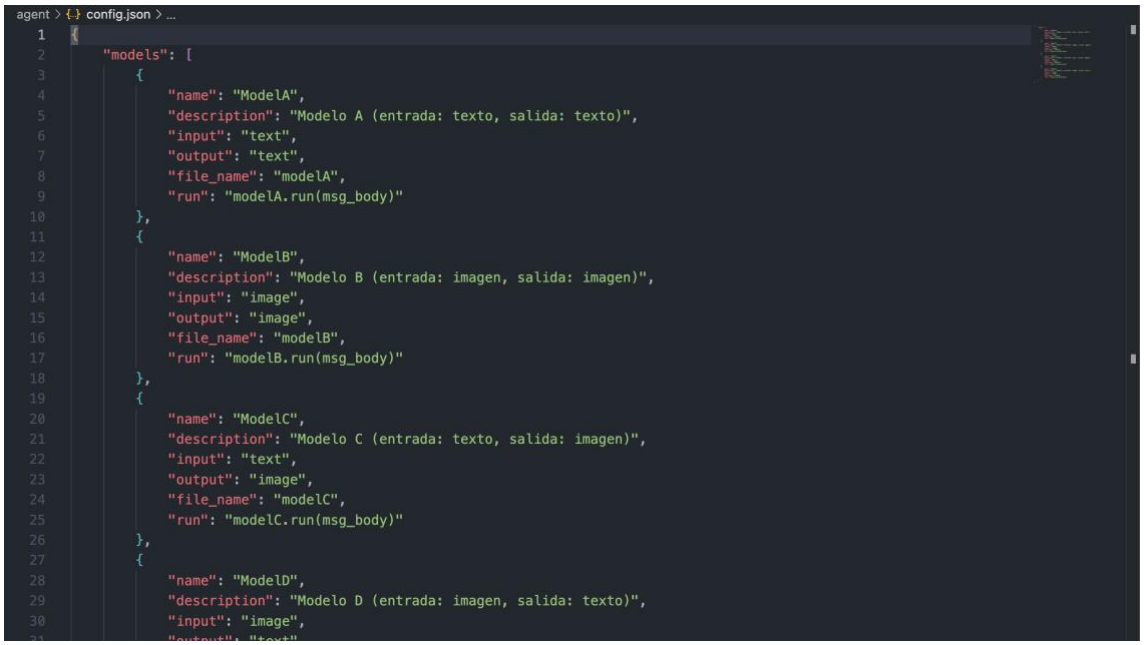
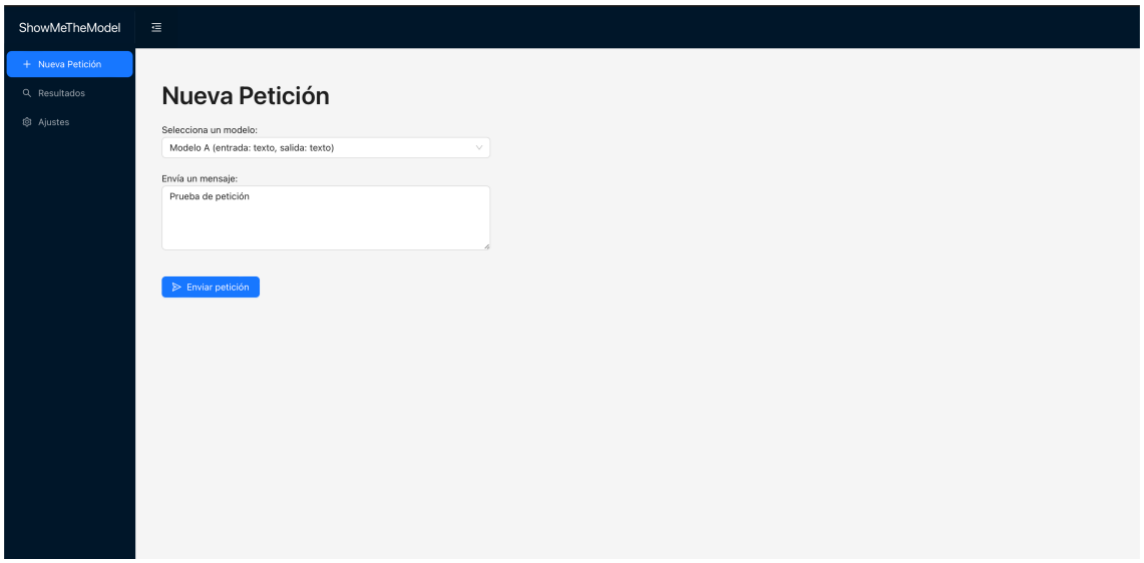


Figura 21: archivo de configuración de los modelos

Se va a crear una nueva petición de ejemplo:



Se pulsa en el botón “Enviar petición” y aparecerá la siguiente pantalla:

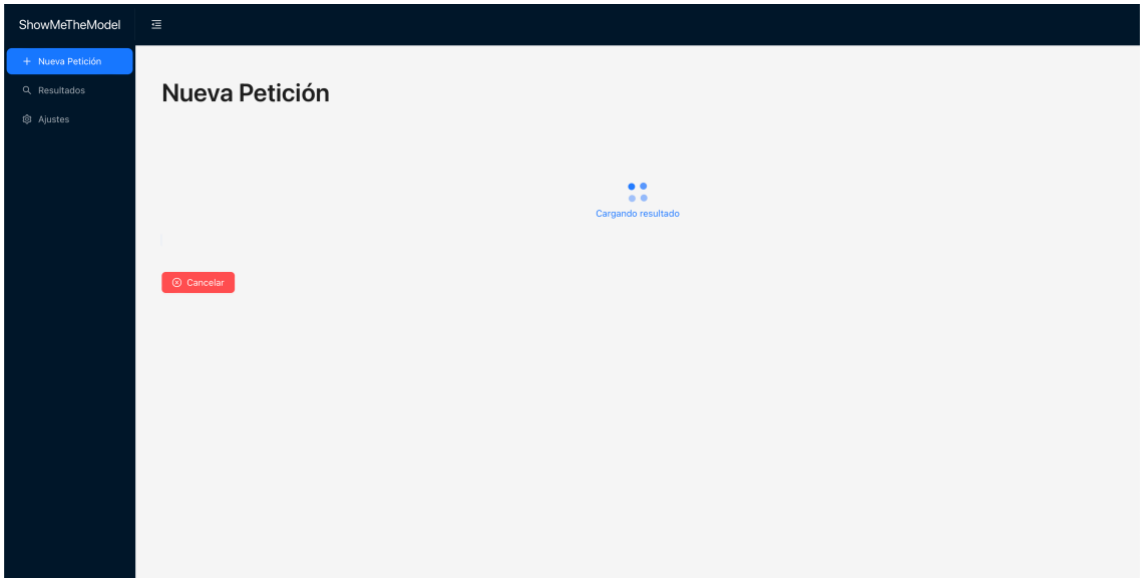


Figura 22: ventana de Nueva Petición (cargando resultado)

Lo que está pasando por debajo es lo siguiente:

En el cliente:

- Se guarda la entrada del usuario en un archivo en la carpeta data de *backend.*
- Guarda el archivo de la entrada en el *bucket* y lo borra del almacenamiento local.
- Se crea una entrada en base de datos con estado “Pendiente”.
- Se envía un mensaje a la cola 1 con atributos: id(generado), nombre del modelo, tiempo, y URL del archivo, y actualiza el estado de la base de datos a “Procesando”.

En el agente:

- Sondea la cola de mensajes uno y obtiene el mensaje enviado por el cliente.
- Coge el atributo URL contenido en el mensaje y se descarga el archivo.
- Ejecuta el modelo de *Machine Learning* sobre el archivo descargado, y guarda el resultado en el sistema de archivos local.
- Guarda el resultado en el *bucket*, del que obtiene la URL al archivo.
- Borra de su sistema de archivos el archivo que se descargó de la URL del mensaje del cliente.
- Borra de su sistema de archivos el archivo que generó como resultado al ejecutar el modelo.
- Envía un nuevo mensaje a la cola 2 que contendrá la URL del resultado, y tendrá el mismo ID que el mensaje que envío el cliente.
- Borra el mensaje recibido de la cola 1.

Después de que el cliente envía la petición:

- Se queda esperando a recibir un mensaje de la cola 2 con el mismo ID que el mensaje que envío a la cola 1.

- Cuando lo obtiene, coge la URL de los atributos del mensaje.
- Se descarga el archivo de la URL.
- Actualiza el estado de la entrada de la base de datos a “Acabado”.
- Procesa los atributos del mensaje para mostrar el resultado en el *frontend* y lo devuelve.
- Borra el mensaje recibido de la cola 2.

Cuando el resultado de la petición esté listo se mostrará la siguiente ventana:

Figura 23: ventana de Nueva Petición (resultados)

También hay modelos que dan como resultado imágenes:

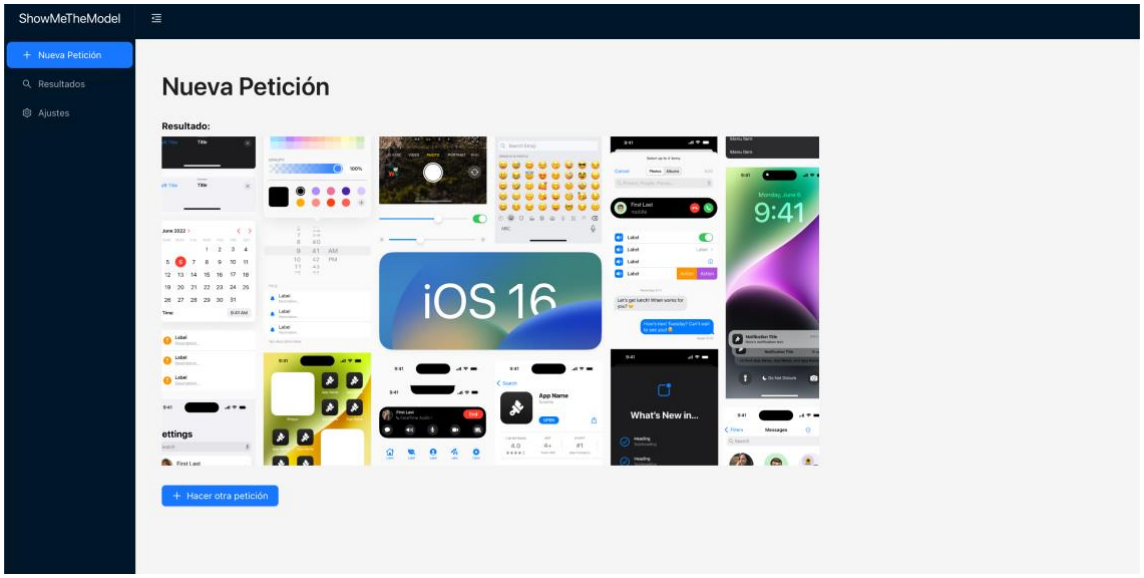


Figura 24: ventana de Nueva Petición (resultados: imagen)

Si hubiera modelos de *Machine Learning* reales implementados, el procesamiento de la entrada del usuario podría tardar bastante tiempo, dependiendo del modelo que se aplique. Es por ello por lo que el usuario no tiene por qué tener la aplicación abierta en el navegador. Una vez haya enviado la petición puede cerrar el navegador, ya que se le enviará un correo cuando la petición esté lista:

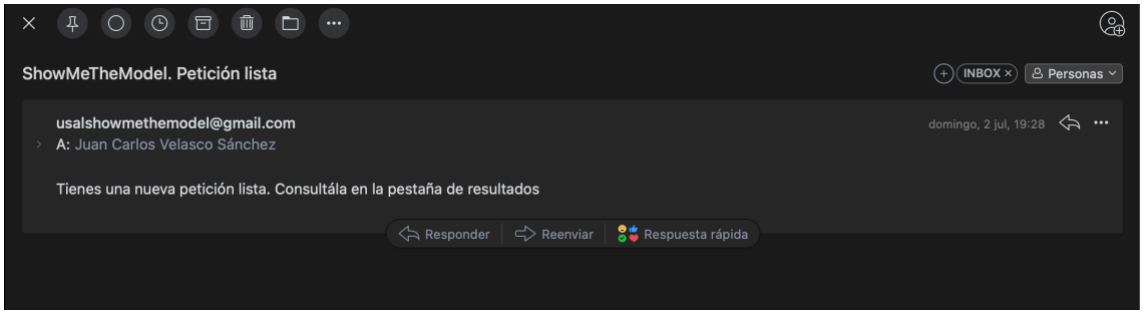


Figura 25: correo de aviso al usuario

Por otro lado, si el usuario accede a la pestaña de resultados podrá ver sus resultados de otras peticiones:

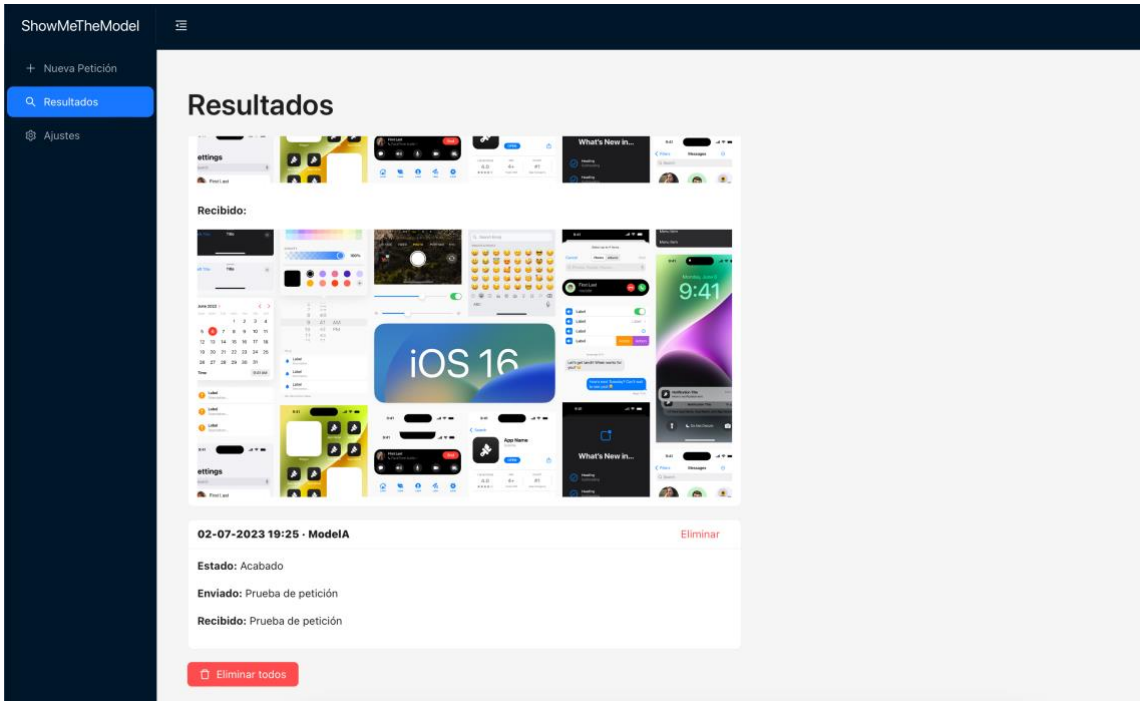


Figura 26: ventana de Resultados

Nada más cargar la ventana, lo que hace la aplicación el llamar al *backend* con una operación POST pasando como parámetro el identificador de usuario, de manera que devuelva como respuesta aquellas entradas de la base de datos de Render que tengan ese identificador de usuario.

Por último, si el usuario accede a la pestaña de “Ajustes”, podrá tener acceso a dos funciones: cerrar sesión y eliminar su cuenta de usuario.

Figura 27: ventana de Ajustes

Con la primera función, simplemente se redirige al usuario a la pantalla de inicio de sesión. Con la segunda función, se mostrará otro recuadro de confirmación:

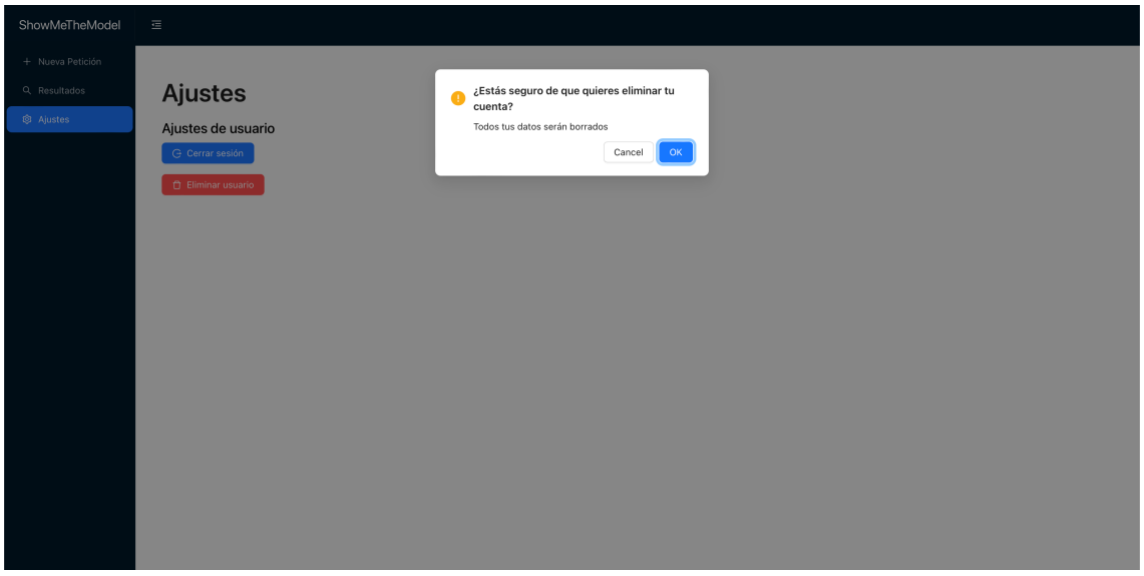


Figura 28: ventana de Ajustes (eliminar cuenta de usuario)

Si el usuario pulsa en el botón “OK”, se borrará su cuenta de usuario de Cognito, los archivos de sus peticiones del *bucket*, y las entradas de la base de datos que tengan su identificador de usuario.

6. Conclusiones y líneas de trabajo futuras

Se ha desarrollado una aplicación web que permite ejecutar modelos de *Machine Learning* de manera remota, accesible desde cualquier dispositivo, ya sea un dispositivo móvil o un equipo de escritorio.

Desde el punto de vista del desarrollo, la aplicación es fácil de mantener y de realizar nuevos incrementos que introduzcan nuevas funcionalidades o aporten un valor añadido al proyecto. Adicionalmente, la aplicación presenta un alto grado de escalabilidad, en caso de que hubiera aumento considerable en el número de usuarios que utiliza la aplicación.

Para desarrollar la aplicación se ha utilizado la metodología ágil SCRUM, que ha permitido desarrollar la aplicación de manera iterativa e incremental.

Se ha implementado un sistema de cola de mensajes, mediante el cual el agente y el *backend* se comunican, realizado con el servicio Amazon SQS.

Para almacenar los datos necesarios que maneja la aplicación, se usa una base de datos PostgreSQL desplegada en la nube mediante la plataforma Render, mediante la cual se alojan las peticiones de los usuarios, y un servicio de almacenamiento en la nube de archivos (bucket) emulado de manera local mediante la plataforma LocalStack que almacena las entradas y salidas de los usuarios.

La parte *frontend* de la aplicación se ha diseñado con la tecnología React y la biblioteca Ant Design, que dotan a la aplicación de una interfaz de usuario fácil de entender, con un lenguaje de diseño moderno, y un diseño adaptable a todo tipo de dispositivos. Mediante el uso de un servidor Flask, se comunican las partes *frontend* y *backend* de la aplicación.

Realizar este proyecto ha supuesto todo un reto, ya que no conocía prácticamente ninguna de las herramientas que he utilizado para desarrollar la aplicación.

Gracias a esto, he podido aprender muchísimo sobre nuevas tecnologías y herramientas que conocía, por lo que creo que en general ha sido una experiencia muy positiva.

Realizar este proyecto en una empresa como HP y con la ayuda de sus profesionales me ha permitido expandir mis conocimientos y aprender sobre todos los ámbitos del desarrollo de software, sobre todo, de cómo se desarrollan productos de gran escala en el ámbito empresarial. Sin su asesoramiento este proyecto no hubiera salido adelante.

En general, creo que he aprovechado al máximo la oportunidad que he tenido para aprender y siempre he estado abierto a investigar nuevas herramientas, probar diferentes alternativas y hacer nuevos cambios en la aplicación.

En cuanto a las ampliaciones futuras, se podrían introducir nuevas funcionalidades, como incluir una sección de filtros que permita realizar una búsqueda de peticiones más

precisa o la opción de copiar el resultado de una petición en el portapapeles, para poder usarlo en otras aplicaciones.

Otra característica que se podría implementar es la inclusión de modelos que permitan tengan como entrada o salida archivos de audio, aportando así nuevas opciones al usuario.

7. Bibliografía

[1] Gitlab: *The DevSecOps Platform*. GitLab. https://gitlab.com/

[2] Colas de mensajes completamente administradas - Amazon Simple Queue Service – Amazon Web Services. Amazon Web Services, Inc. https://aws.amazon.com/es/sqs/

[3] AWS | Cloud Computing - Servicios de informática en la nube. Amazon Web Services, Inc. https://aws.amazon.com/es

[4] Apache Kafka. Apache Kafka. https://kafka.apache.org/

[5] Messaging that just works — RabbitMQ. https://www.rabbitmq.com/

[6] Cloud Application Hosting for Developers | Render. Cloud Application Hosting for Developers | Render. https://render.com/

[7] Cloud Application Platform | Heroku. https://www.heroku.com/

[8] Welcome to Flask — Flask Documentation (2.3.x). (s. f.). https://flask.palletsprojects.com/en/2.3.x/

[9] FastAPI. (s. f.). https://fastapi.tiangolo.com/

[10] React. https://es.react.dev/

[11] JavaScript With Syntax for Types. (s. f.). https://www.typescriptlang.org/

[12] Ant Design - The world's second most popular React UI framework. (s. f.). https://ant.design/

[13] Welcome to Python.org. Python.org. https://www.python.org/

[14] Axios. (s. f.). https://axios-http.com/

[15] LocalStack. *LocalStack*. https://LocalStack.cloud/

[16] Docker: Accelerated, Containerized Application Development. (2023). *Docker*. https://www.docker.com/

[17] AWS | Gestión de identidades y autenticación de usuario en la nube. Amazon Web Services, Inc. https://aws.amazon.com/es/cognito/

[18] Jenkins. Jenkins. https://www.jenkins.io/

[19] Streamlit • A faster way to build and share data apps. https://streamlit.io/

[20] Team, G. *Gradio*. https://gradio.app/

[22] Node.js. Node.js. https://nodejs.org/en

[23] Material Design. (s. f.-c). Material Design. https://m3.material.io/

[24] SQS - Boto3 1.26.151 documentation. https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sqs.html

[25] *Personal Cloud Storage & File Sharing Platform - Google*. (s. f.). https://drive.google.com/

[26] *Personal Cloud Storage – Microsoft OneDrive*. (s. f.). https://onedrive.live.com/

[27] *PostgreSQL*. (2023, 3 julio). PostgreSQL. https://www.postgresql.org/

[28] psycopg2. PyPI. https://pypi.org/project/psycopg2/

[29] Firebase. *Firebase*. https://firebase.google.com/?hl=es-419