

Trabajo fin de grado

Diseño y desarrollo de una aplicación de web scraping de propósito general asistida por inteligencia artificial.



Antonio David Delgado Santiago

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C/ Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

**Diseño y desarrollo de una aplicación de web
scraping de propósito general asistida por
inteligencia artificial.**

Autor: Antonio David Delgado Santiago

Tutor: Alejandro Bellogín Kouki

mayo 2026

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© mayo de 2026 por UNIVERSIDAD AUTÓNOMA DE MADRID

Francisco Tomás y Valiente, n.º 1

Madrid, 28049

Spain

Antonio David Delgado Santiago

Diseño y desarrollo de una aplicación de web scraping de propósito general asistida por inteligencia artificial.

Antonio David Delgado Santiago

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

AGRADECIMIENTOS

Primero de nada, agradecer a Alejandro Bellogín por su apoyo y consejos a lo largo de este año. Si no fuera por él no hubiera sido capaz de realizar este Trabajo de Fin de Grado. Nunca olvidaré las tardes resolviendo problemas y la sonrisa que se me quedaba cuando encontrábamos una solución.

Mi más sincero agradecimiento a mis padres y a mi hermana, quienes siempre lo han dado todo por apoyarme y motivarme durante toda mi carrera académica. Gracias a ellos doy siempre lo mejor de mí. Además, quiero dar especial agradecimiento a mi hermana por darme la idea sobre la que acabaría realizando este proyecto.

Asimismo, me gustaría dar las gracias a los docentes de la Escuela Politécnica Superior por el esfuerzo y compromiso en enseñarnos de la mejor manera posible, a aprender y a ser individuos independientes y resolutivos.

Finalmente, a mis amigos y a mi novia. Ellos son los que me han impulsado cada día a mejorar y me han enseñado lo bonito que es disfrutar de este camino estando bien acompañado.

RESUMEN

Internet ha revolucionado la forma en la que obtenemos información. Esto ha provocado que la mayoría de las personas prefieran buscar datos con un ordenador o un dispositivo móvil, antes que utilizar los medios tradicionales, convirtiendo a las páginas web en fuentes de datos muy útiles para cualquier persona. Bajo este contexto, la comparación de catálogos de ropa, o análisis de noticias, se han convertido en información de mucho interés. Sin embargo, hay muchas personas que consideran que buscar grandes cantidades de datos es una tarea muy tediosa y aburrida, por ello, buscan automatizar estas búsquedas, pero no saben cómo sin saber programar.

Para cubrir esta necesidad se ha desarrollado una aplicación web llamada Kosmos, destinada a todos los públicos, con una serie de características que garantizan una experiencia intuitiva. La aplicación dispone de un asistente virtual conversacional encargado de guiar al usuario en la aplicación. Este permite a los usuarios nuevos entender el funcionamiento de la herramienta principal de la web, la extracción de datos a través de web scraping, mediante el chat. La guía del asistente comienza explicando los datos que debe insertar el usuario, siendo estos: dos enlaces a una página web, y la intención de búsqueda. Ambos enlaces deben ser del mismo dominio, teniendo cada uno un papel distinto en el proceso. El primero servirá de ejemplo para que una inteligencia artificial aprenda el patrón de datos de la página web. El segundo se referirá a la página de catálogo o categoría donde se empezará la búsqueda de datos de forma autónoma.

Un estudio de usabilidad con dieciocho participantes evaluó las funciones clave de Kosmos mediante tareas guiadas que incluyeron la configuración de una búsqueda, el refinamiento de los datos mediante un comentario correctivo, la consulta de los resultados y la eliminación de los resultados seleccionados. La plataforma obtuvo una calificación de “Excelente” en las preguntas sobre la satisfacción general al usarla, lo que confirma lo eficaz e intuitivo que es su diseño. De esta manera, Kosmos ofrece una vía para obtener información de forma automática y sencilla para todo el público.

PALABRAS CLAVE

Web scraping, Extracción de datos, Asistente virtual, Inteligencia artificial, Aplicación web, Estudio de usabilidad

ABSTRACT

The Internet has revolutionized the way we obtain information. As a result, most people now prefer to search for information using a computer or mobile device rather than traditional media, making websites extremely useful sources of information for everyone. In this context, comparing clothing catalogs or analyzing news stories has become highly relevant. However, many people find searching through large amounts of data to be a tedious and boring task, so they want to automate these searches but do not know how to do so without knowing how to program.

To address this need, a web application called Kosmos has been developed. Designed for all users, it features a range of capabilities that ensure an intuitive experience. The application includes a conversational virtual assistant that guides users through the platform. This assistant helps new users understand how the website's main tool, data extraction via web scraping, works through a chat interface. The assistant's guide begins by explaining the data the user must enter, which consists of two links to a webpage and the search intent. Both links must be from the same domain, with each playing a distinct role in the process. The first will serve as an example for the artificial intelligence to learn the data pattern of the webpage. The second will refer to the catalog or category page where the data search will begin autonomously.

A usability study involving eighteen participants evaluated Kosmos's key features through guided tasks that included setting up a search, refining the data using a corrective comment, reviewing the results, and deleting selected results. The platform received an "Excellent" rating on questions regarding overall satisfaction with its use, confirming how effective and intuitive its design is. In this way, Kosmos offers a way for everyone to obtain information automatically and easily.

KEYWORDS

Web scraping, Data extraction, Virtual assistant, Artificial intelligence, Web application, Usability study

ÍNDICE

1	Introducción	1
1.1	Motivación del proyecto	1
1.2	Objetivos	2
1.3	Estructura del documento	2
2	Estado del arte	3
2.1	Evolución histórica y tecnológica	3
2.2	Metodologías	4
2.2.1	Comparativa entre extracción manual y automatizada	4
2.2.2	Técnicas tradicionales de extracción	4
2.3	Sistemas de seguridad y evasión antibot	6
2.4	Evolución de las herramientas de automatización	6
2.5	Web scraping con inteligencia artificial	7
2.6	Aspectos legales y normativas	7
2.6.1	Tratamiento de datos personales en el RGPD y la LOPDGDD	8
2.6.2	Minería de datos y propiedad intelectual en la Directiva DSM	8
2.6.3	Buenas prácticas y límites técnicos	8
3	Diseño	9
3.1	Estructura de la aplicación	9
3.2	Requisitos de la aplicación	10
3.2.1	Requisitos funcionales	11
3.2.2	Requisitos no funcionales	11
3.3	Ciclo de vida	12
3.4	Diseño de la aplicación	12
3.4.1	Diseño de la base datos	12
3.4.2	Diagrama de casos de uso	13
3.4.3	Diagramas de flujo	14
4	Implementación	17
4.1	Entorno de desarrollo y herramientas	17
4.1.1	Hardware y sistema operativo	17
4.1.2	Herramientas de desarrollo	17
4.1.3	Entorno virtual y librerías	19

4.1.4 Integración de la inteligencia artificial	20
4.2 Desarrollo de los módulos del sistema	20
4.2.1 Análisis de una página web	20
4.2.2 Extracción automática a gran escala	23
4.2.3 Página de inicio	26
4.2.4 Sobre el proyecto	26
4.2.5 Características	26
4.2.6 Registro e inicio de sesión	27
4.2.7 Notificaciones de la aplicación	27
4.2.8 Listar páginas web y mostrar productos	28
4.2.9 Borrar páginas web	28
4.3 Decisiones técnicas	29
4.4 Complicaciones encontradas	31
5 Experimentos	33
5.1 Estudio de usuarios	33
5.1.1 Consentimiento	33
5.1.2 Datos demográficos y experiencia en búsqueda web	33
5.1.3 Motivaciones y preferencias	34
5.1.4 Recorrido guiado de tareas	35
5.1.5 Experiencia y comentarios posteriores a la prueba	36
5.2 Análisis de los resultados	38
6 Conclusiones	39
6.1 Conclusiones	39
6.2 Trabajo Futuro	40
Bibliografía	42
Apéndices	43
A Apendice	45

LISTAS

Lista de figuras

3.1	Diagrama de los módulos de la aplicación	10
3.2	Diagrama de los objetos de la base de datos	13
3.3	Diagrama de los casos de uso del sistema	14
3.4	Diagrama de flujo del enrutador de páginas web	15
3.5	Diagrama de flujo de la extracción automática de datos	16
5.1	Consentimiento.	33
5.2	Datos demográficos	34
5.3	Experiencia en búsqueda web	34
5.4	Motivaciones y preferencias	35
5.5	Tipos de datos a extraer	35
5.6	Configuración y refinamiento	36
5.7	Consulta y eliminación	36
5.8	Facilidad y navegación	37
5.9	Satisfacción y utilidad	37
5.10	Estética y claridad	38
A.1	Consentimiento	45
A.2	Distribución por edad	45
A.3	Género	46
A.4	Experiencia en búsqueda web	46
A.5	Motivaciones principales	47
A.6	Preferencia de interacción	47
A.7	Tipos de datos a extraer	48
A.8	Configuración y análisis	48
A.9	Refinamiento mediante asistente	49
A.10	Consulta de resultados	49
A.11	Eliminación de resultados	50
A.12	Facilidad de uso	50
A.13	Claridad de la navegación	50
A.14	Satisfacción general	51
A.15	Utilidad de los resultados	51
A.16	Estética de la aplicación	52

A.17 Claridad de presentación	52
---	----

INTRODUCCIÓN

En el mundo actual, internet trae una gran cantidad de datos, información y artículos a su alcance desde cualquier dispositivo. Sin embargo, al intentar recopilar y procesar estos contenidos manualmente, los usuarios rápidamente se enfrentan a un cuello de botella considerable. Por lo tanto, diseñar herramientas para automatizar la extracción de datos en la web tiene cada vez más relevancia.

1.1. Motivación del proyecto

En este contexto, la información pública de internet es uno de los recursos más valiosos para cualquier sector. Existen diferentes vías para acceder a estos datos de forma estructurada, como el uso de bases de datos públicas o APIs oficiales. Sin embargo, muchas plataformas no ofrecen estas herramientas, limitan su acceso, o simplemente no tienen los datos exactos que se buscan. Cuando estas opciones oficiales no están disponibles, la solución es extraer la información directamente de las páginas web. El problema es que, aunque hay mucha información, sacar todos estos datos a gran escala es un proceso largo, manual y que está limitado a personas que saben programar.

Para automatizar esta extracción se utiliza una técnica conocida como *web scraping*. Esta técnica se encarga de simular el comportamiento de un usuario para extraer el contenido visual de las páginas y transformarlo en información estructurada, lista para guardarse en bases de datos. Su utilidad hace que sea el motor de muchas industrias. Una de sus aplicaciones es el comercio electrónico, en este se usa para observar los cambios de precios de la competencia, otra aplicación es el sector inmobiliario que recopila los datos de las propiedades en venta, para tener las mejores ofertas. Incluso tiene un gran protagonismo en los comparadores de vuelos, que agrupan las ofertas de muchas marcas en una misma pantalla. A pesar de que recopilar datos sea muy útil para cualquier negocio, obtenerlos sigue siendo un obstáculo para la mayoría de los usuarios, y más si no saben programar. Para solucionar este problema, este proyecto desarrolla Kosmos, una aplicación diseñada para ayudar a superar esta barrera técnica. El principal valor de esta herramienta es su universalidad. De esta manera, cualquier persona puede extraer cualquier tipo de contenido, enfocándose en catálogos de tiendas online y artículos de noticias, de forma totalmente automática y sin importar los conocimientos previos que

tenga.

1.2. Objetivos

El principal objetivo de este trabajo es afianzar los conocimientos adquiridos durante el grado en distintas asignaturas. En primer lugar, se estudia la extracción automática de datos, conocida como web scraping, desde un enfoque que integra grandes modelos de lenguaje, siendo estos Gemini y Llama 3 para superar la rigidez de los extractores tradicionales basados en reglas fijas. Posteriormente, estas capacidades de análisis semántico se combinan con sistemas de navegación automatizada y colas de tareas asíncronas, lo que permite generar selectores dinámicos, evadir medidas de seguridad antibot y realizar rastreos a gran escala. Finalmente, se desarrolla una aplicación web completa y funcional, Kosmos, para poder utilizar este motor de extracción de datos en un entorno real, incorporando un asistente virtual y una interfaz accesible para usuarios sin conocimientos de programación.

1.3. Estructura del documento

El documento se encuentra dividido en seis capítulos:

Capítulo 1. Introducción: se presenta la motivación y el problema sobre el que gira el trabajo, los objetivos a completar durante el proyecto y la estructura del documento.

Capítulo 2. Estado del arte: se realiza un recorrido por los fundamentos técnicos del web scraping y el impacto que ha tenido la reciente integración de los grandes modelos de lenguaje en la automatización de estos procesos.

Capítulo 3. Diseño del sistema: muestra el diseño que se ha llevado a cabo durante el proyecto incluyendo el análisis de requisitos, su estructura general y su ciclo de vida.

Capítulo 4. Implementación: se explica la construcción de la aplicación y del motor de scraping web, además del desarrollo de los algoritmos de navegación automatizada.

Capítulo 5. Pruebas y resultados: se describen las pruebas de usabilidad realizadas y se analizan sus resultados.

Capítulo 6. Conclusiones y trabajo futuro: se exponen las conclusiones finales del trabajo, así como las posibles mejoras que se pueden investigar como trabajo futuro.

ESTADO DEL ARTE

En este capítulo se hace una introducción a la tecnología de la extracción automática de datos. Para ello, se comienza en la Sección 2.1 con un recorrido histórico sobre la evolución del *web scraping*. A continuación, en la Sección 2.2, se analizan las formas tradicionales de extraer información. Posteriormente, en la Sección 2.3, se examinan los sistemas de seguridad actuales y cómo se evitan. Seguidamente, en la Sección 2.4, se explica la evolución de las herramientas de automatización web, para después abordar en la Sección 2.5 el impacto de la inteligencia artificial. Finalmente, en la Sección 2.6, se resumen las leyes que regulan la extracción de datos web.

2.1. Evolución histórica y tecnológica

El inicio de la extracción de datos web o *web scraping* comenzó en los años 90. El web scraping empezó siendo una herramienta con unos fines distintos a los de hoy en día. Este empezó siendo utilizado para recopilar toda la información posible para los primeros buscadores web. Los programadores crearon bots conocidos como crawlers que se dedicaban a entrar en páginas web y descargar el código de estas. Los crawlers eran especialistas en seguir los enlaces de cada página web. Esto desembocó en que en mediados de los años 90, buscadores pioneros como Lycos o AltaVista emplearan estos bots para mapear el internet [1].

Durante aquellos primeros años, las páginas eran simples documentos de texto sin movimiento. Sin embargo, con el paso del tiempo, la web se transformó en una plataforma interactiva llena de aplicaciones complejas. Este cambio obligó a que las técnicas de extracción también tuvieran que evolucionar para poder adaptarse a páginas mucho más pesadas y difíciles de leer.

Más adelante, en los años 2000, esto dio un gran giro. Ya no servía de nada guardar páginas enteras, sino que las empresas necesitaban sacar datos muy concretos para pasarlos del código HTML a bases de datos ordenadas. Hoy en día, el proceso está tan automatizado que se extraen millones de datos cada día. El paso más reciente en esta evolución ha sido juntar la extracción de datos con la inteligencia artificial. Según estudios recientes llevados a cabo por Barba et al. [2], esta combinación ha crecido más de un 35 % cada año desde 2017, convirtiéndose en la principal herramienta

para conseguir la inmensa cantidad de información necesaria para entrenar a los nuevos modelos de lenguaje.

2.2. Metodologías

En esta sección se explican las diferencias entre extraer información a mano y hacerlo de forma automática, así como las técnicas clásicas para buscar datos dentro de una página web.

2.2.1. Comparativa entre extracción manual y automatizada

Existen dos formas principales de recopilar datos en internet: la manual y la automatizada. Cada una tiene sus ventajas e inconvenientes dependiendo de la cantidad de información que se necesite:

Precisión vs. Escalabilidad: Cuando una persona recopila información a mano, es muy precisa. Los humanos somos muy buenos entendiendo el contexto de una página e ignorando la publicidad. Sin embargo, hacerlo a mano es un proceso demasiado lento. Si hace falta conseguir miles de datos, usar personas es poco práctico. El *scraping* automatizado soluciona esto, ya que puede leer cientos de páginas en unos pocos segundos.

Consistencia y reducción de errores humanos: Cuando una persona realiza tareas repetitivas durante horas, como copiar y pegar precios, es inevitable que acabe cometiendo errores por fatiga. Un solo número mal escrito en una hoja de cálculo puede arruinar por completo un estudio de mercado. Por el contrario, un sistema automatizado ejecuta las mismas instrucciones millones de veces con una precisión absoluta, garantizando que no haya fallos al guardar los datos.

Datos anticuados: En sectores como la venta de billetes de avión, los precios cambian constantemente. La lentitud de la extracción manual provoca que los datos guardados en un Excel se queden anticuados casi al instante. Usar programas automatizados soluciona este retraso, consiguiendo que las bases de datos tengan la información exacta en tiempo real.

2.2.2. Técnicas tradicionales de extracción

Hasta hace poco, la forma clásica de extraer datos consistía en descargar el código HTML de la página web y moverse por su estructura, la cual se conoce como árbol DOM, por sus siglas en inglés *Document Object Model*. Tal y como detallan Gunawan et al. [3], para extraer un dato concreto con este método, el programador tiene que revisar el código a mano y escribir una regla fija para decirle al programa dónde tiene que buscar.

Rastreo web

El primer paso para sacar datos de internet es el rastreo. El programa tiene que ir saltando de un enlace a otro para encontrar la información. Como advierte Najork [4], el mayor problema a la hora de hacer esto es conseguir que el *bot* descargue información muy rápido, pero teniendo cuidado de no saturar ni tumbar los servidores de las webs que visita.

Para organizarse de forma eficiente, los programas usan una lista de tareas pendientes llamada frontera de rastreo, denominada en inglés *Crawl Frontier*. Esta lista controla qué enlace visitar después y filtra los enlaces inútiles, como las políticas de privacidad o las páginas de contacto. Además, esta cola de tareas evita que el programa se quede atascado dando vueltas en bucles infinitos de la web, un fallo técnico muy común conocido como *Spider Traps*.

Selectores CSS y Expresiones XPath

Para moverse por el código HTML y encontrar el dato, se usan principalmente dos métodos. Por un lado, están los selectores CSS, que aprovechan los nombres de diseño y colores de la página para ir directos al dato que interesa. Se pueden traducir estos selectores a expresiones regulares, consiguiendo que el programa vaya más rápido y gaste mucha menos memoria [5].

Por otro lado, se usan las expresiones XPath. A diferencia de CSS, XPath busca guiándose por el lugar exacto que ocupa el dato en la página, por ejemplo: el tercer elemento de la segunda caja. Aunque gasta más recursos, su gran ventaja es que es mucho más preciso cuando se busca en páginas web que están mal organizadas [3].

El problema de los selectores fijos

El gran fallo que tienen tanto CSS como XPath es que dependen de que la página no cambie nunca. Si la web cambia un poco su diseño, el programa de *scraping* se rompe por completo. Actualmente, este problema es muy común porque se usan herramientas, como Tailwind CSS, que cambian los nombres de las etiquetas del código automáticamente cada vez que los creadores actualizan la página.

A este problema técnico hay que sumar que muchas tiendas online alteran su código a propósito para confundir a los programas y evitar que sus rivales copien sus precios. Por culpa de todos estos cambios, los programadores tienen que perder mucho tiempo arreglando las reglas a mano todos los días, lo que hace que mantener estos sistemas sea muy caro y frustrante [6].

2.3. Sistemas de seguridad y evasión antibot

Hoy en día, las páginas web se defienden de los *bots* para evitar que sus servidores se colapsen o que roben su información. Al principio, la única defensa era bloquear la dirección IP si el sistema detectaba que alguien hacía demasiadas peticiones muy rápido [4]. Sin embargo, ahora se usan técnicas mucho más avanzadas, como la Huella Dactilar del Navegador o *Browser Fingerprinting*. Empresas de seguridad como Cloudflare analizan en milisegundos el ordenador del visitante: miran la tarjeta gráfica, las fuentes instaladas o los núcleos del procesador. Si los datos no cuadran con los de una persona real, bloquean el acceso al instante.

Dentro de estas técnicas de seguridad, destaca la detección de los navegadores invisibles o *headless*. Normalmente, los programas de extracción utilizan el navegador en un modo oculto para no gastar recursos y funcionar más rápido. Las páginas web modernas lanzan pequeños retos invisibles para comprobar si el navegador está dibujando realmente la pantalla. Si el programa falla esta prueba, salta la alarma y se corta la conexión.

Para evitar que estos sistemas bloqueen la recopilación de datos, hacen falta estrategias de evasión. El estudio de Cuevas et al. [7] demuestra que si el programa choca contra las barreras *anti-bot*, se puede llegar a perder hasta el 46 % de los datos importantes de la página. Para solucionar esto, se suelen usar *proxies* rotatorios, los cuales cambian la IP para no saturar a la página, y técnicas de sigilo, que consiguen que el programa se comporte exactamente igual que un usuario normal leyendo en su pantalla [8].

2.4. Evolución de las herramientas de automatización

Para automatizar la extracción, se han ido creando diferentes herramientas con el tiempo. Al principio, se usaban librerías como BeautifulSoup en Python, que eran muy rápidas para leer código HTML estático. Sin embargo, no sabían ejecutar código JavaScript. En este contexto, Mesbah et al. [9] señalan que, con la llegada de las webs modernas que cargan los datos poco a poco en la pantalla, las llamadas aplicaciones AJAX, estas herramientas antiguas dejaron de funcionar.

Para solucionar este problema, se empezó a usar Selenium, una herramienta que abre un navegador real y ejecuta todo el JavaScript como si fuera un usuario haciendo clics. El inconveniente es que consume mucha memoria y es bastante lento, ya que usa un intermediario para mandar las órdenes al navegador, lo que además hace que los sistemas de seguridad lo detecten fácilmente. Como paso intermedio para solucionar estos problemas de lentitud, la industria creó herramientas como Puppeteer, que permitían un control más directo, pero estaban demasiado limitadas a un solo tipo de navegador.

Hoy en día, el estándar definitivo es usar Playwright. A diferencia de Selenium, Playwright se conecta directamente al núcleo del navegador. Evaluaciones recientes, como la de Halilaj y Norlin [10],

demuestran que esto lo hace muchísimo más rápido y le permite trabajar en tiempo real sin atascos en cualquier plataforma. Aun así, aunque es una herramienta muy potente, sigue teniendo el problema de depender de reglas fijas que se rompen cuando la web cambia de diseño.

2.5. Web scraping con inteligencia artificial

La llegada de los Grandes Modelos de Lenguaje, conocidos como LLM, ha traído una solución real al problema de que los programas se rompan. Tal y como apuntan Nguyen y Maag [11], la IA es capaz de entender cómo está organizada visualmente una página sin depender de reglas fijas.

Los primeros intentos de usar la IA para esto funcionaban por fuerza bruta: consistían en pasarle a la IA todo el código de la página de golpe para que buscara la información. El problema es que hacer esto es muy lento y económicamente inviable. El motivo por el que procesar todo el código es tan caro se debe a que empresas como OpenAI o Google cobran por cada pequeño fragmento de texto, llamado *token*, que la inteligencia artificial tiene que leer. Una sola página web puede contener decenas de miles de estos fragmentos. Si se multiplica este coste por todos los artículos de una tienda, el precio se dispara. Además, darle tanto texto inútil hace que la IA se confunda fácilmente y se invente los datos, lo que se conoce como alucinar.

Para solucionar estos problemas, Sivakumar [12] explica que es obligatorio limpiar la información antes. A este truco se le llama poda del DOM, y consiste en borrar todo el código que no interesa, como estilos de colores o *scripts*, antes de pasárselo a la IA. Esto reduce el tamaño del texto más de un 70 %, haciendo que sea muchísimo más barato de procesar y evitando que la inteligencia artificial cometa errores.

Otra idea mucho mejor es cambiar la forma de usar la IA: en lugar de usarla para leer todos los productos, se usa como si fuera un programador. La IA mira una sola página de ejemplo y ella misma crea la regla CSS o XPath. Después, se usa Playwright para ejecutar esa regla miles de veces sin gastar apenas dinero. Investigaciones recientes de Dagdelen et al. [13] confirman que las inteligencias artificiales actuales son muy buenas extrayendo estas reglas exactas a partir de código desordenado.

2.6. Aspectos legales y normativas

Saber si la extracción automatizada es legal depende del tipo de información que se recopile y de cuánto se sature a la página de destino. A continuación se resumen las normas actuales:

2.6.1. Tratamiento de datos personales en el RGPD y la LOPDGDD

El Reglamento Europeo de Protección de Datos exige el permiso de los usuarios para tratar sus datos personales. Sin embargo, los estudios legales de Bonvin [14] dejan claro que hacer *scraping* es legal siempre que se extraiga información comercial pública, como catálogos, precios o descripciones, y no se usen los programas para intentar recopilar información privada de personas reales con el fin de crear perfiles a sus espaldas.

2.6.2. Minería de datos y propiedad intelectual en la Directiva DSM

La ley europea de derechos de autor permite extraer datos y textos de forma automática, técnica conocida como minería de datos. Aparte de los derechos de autor, la jurisprudencia internacional, como el famoso juicio de hiQ contra LinkedIn, ha determinado que sacar información que es totalmente pública en internet no es un delito.

Muchas veces las páginas web prohíben la extracción en sus términos y condiciones, pero los jueces han dictaminado que un simple aviso escrito no sirve como barrera legal frente a información abierta a todo el mundo. Como resumen Brown et al. [15], la verdadera línea roja de la ley solo se cruza si el programa se salta sistemas técnicos de seguridad, usa contraseñas sin permiso o intenta evitar muros de pago de forma fraudulenta.

2.6.3. Buenas prácticas y límites técnicos

Si un programa hace miles de peticiones en un segundo, puede colapsar el servidor de la página web. Para evitar que esto se considere un ciberataque, es obligatorio programar los *bots* con límites estrictos de velocidad y políticas de cortesía. Estas pausas obligatorias aseguran que el rastreo sea suave y no sature a los servidores que se están visitando [4].

DISEÑO

En esta sección se presenta gráficamente la estructura y procesos clave del sistema, junto con los requisitos de la aplicación.

3.1. Estructura de la aplicación

La arquitectura está diseñada en módulos independientes para facilitar la comprensión de la aplicación y el diseño. La modularización permite una mayor limpieza del código y la corrección de este, además de simplificar el añadir nuevas funcionalidades a futuro. Esta arquitectura se puede observar en el diagrama de la Figura 3.1. Siendo los módulos que componen el sistema los siguientes:

Aplicación web y frontend: Desarrollada con el framework Vue.js. Es la parte visible del sistema y se encarga de mostrar la interfaz gráfica. Desde aquí, el usuario interactúa con la plataforma para configurar extracciones, conversar con el chatbot y consultar las tablas de resultados.

Servidor y API backend: Desarrollado en python utilizando Django. Funciona como el centro de control del proyecto. Se comunica con la aplicación web mediante peticiones HTTP y JSON, recibiendo las órdenes del usuario y distribuyéndolas al resto del sistema.

Base de Datos: Almacenamiento local mediante SQLite. El servidor de Django guarda aquí, de forma organizada, los registros de usuarios, el historial de las páginas, los selectores aprendidos y los datos finales extraídos.

Gestor de Tareas Asíncronas: Implementado con Celery. Esta pieza es fundamental para evitar que la aplicación se bloquee. Su función es recoger las tareas pesadas, como navegar por múltiples enlaces, y ejecutarlas en segundo plano, permitiendo que el servidor principal siga disponible para atender al usuario.

Módulo de Inteligencia Artificial: El sistema de análisis se divide en dos herramientas distintas para ganar velocidad y reducir costes:

- **Chatbot Local Ollama:** Una inteligencia artificial que funciona directamente en el entorno local. Gestiona la interacción con el usuario y traduce los requisitos del usuario a instrucciones técnicas.

- **API de Google Gemini en la nube:** El modelo encargado de las tareas de análisis complejas. Se ocupa de revisar el código de la web y obtener los selectores exactos necesarios para extraer la información.

Motor de scraping con Playwright: Herramienta que controla un navegador de forma automática para navegar por las páginas y aplicar los selectores generados por la IA.

Enrutador de tráfico con ProtonVPN y Proxies: Capa de seguridad del proyecto. Se encarga de ocultar y cambiar la dirección IP de origen antes de realizar las peticiones a la web objetivo, evitando así bloqueos por parte de los sistemas de defensa.

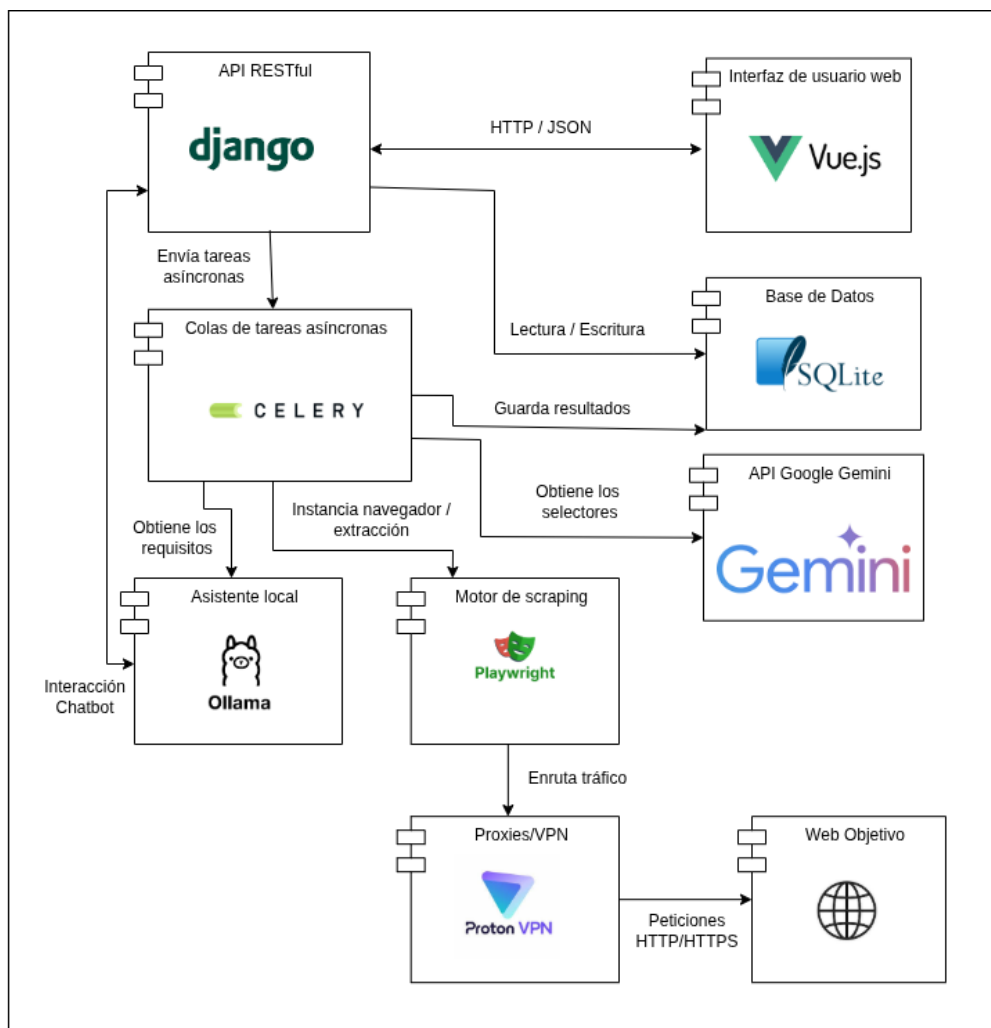


Figura 3.1: Diagrama con los diferentes módulos de la aplicación y su relación.

3.2. Requisitos de la aplicación

Esta sección detalla los requisitos funcionales y no funcionales del sistema.

3.2.1. Requisitos funcionales

- RF-1.**— El sistema debe posibilitar a los usuarios el registrarse, iniciar sesión y cerrar sesión en la aplicación.
- RF-2.**— El sistema debe contener una interfaz que ofrezca: una página de inicio, una página que explique el proyecto, una página que detalle las características principales del proyecto, una página para conversar con el chatbot y una página que muestre los resultados de las extracciones.
- RF-3.**— El sistema debe permitir al usuario recibir notificaciones vía web o del sistema operativo para avisar de que un proceso de extracción en segundo plano ha finalizado.
- RF-4.**— El sistema debe integrar un chatbot local que actúe como intermediario para gestionar las peticiones del usuario.
- RF-5.**— El sistema debe permitir al usuario iniciar una nueva conversación con el chatbot en mitad de una sesión.
- RF-6.**— El sistema debe permitir al usuario introducir una URL de producto como página de ejemplo, una URL de categoría para la extracción automatizada y definir qué datos específicos desea extraer mediante la conversación con el chatbot.
- RF-7.**— El sistema debe mostrar al usuario una prueba inicial de extracción. Si los datos son erróneos, el sistema debe permitir al usuario escribir un comentario correctivo para que la inteligencia artificial regenere y afine los selectores de los productos.
- RF-8.**— El sistema debe descargar el código HTML de las páginas y limpiarlo automáticamente, eliminando anuncios, menús, scripts y bloques de aceptación de cookies, para generar un esqueleto HTML reducido.
- RF-9.**— El sistema debe enviar el esqueleto HTML limpio al modelo de inteligencia artificial extractor para que este identifique automáticamente las rutas exactas de los datos solicitados.
- RF-10.**— El sistema debe incluir un worker/crawler rastreador que utilice los selectores aprendidos para navegar de forma autónoma por la URL de categoría y recolectar la información de todos los productos del catálogo.
- RF-11.**— El sistema debe almacenar en la base de datos local el catálogo extraído, los selectores aprendidos, las URLs introducidas y el historial de la conversación con el chatbot.
- RF-12.**— El sistema debe permitir al usuario consultar el historial completo de sesiones anteriores, mostrando los resultados obtenidos, la conversación asociada y un contador del total de productos encontrados.
- RF-13.**— El sistema debe permitir al usuario eliminar resultados específicos almacenados en su cuenta.
- RF-14.**— El sistema debe permitir al usuario actualizar en tiempo real los resultados mostrados en pantalla mediante un botón de refresco.

3.2.2. Requisitos no funcionales

- RNF-1.**— El backend del sistema debe estar desarrollado en el lenguaje Python, utilizando una versión 3.10 o superior para garantizar la compatibilidad con librerías asíncronas utilizadas.
- RNF-2.**— El frontend de la aplicación debe estar desarrollado utilizando el framework Vue.js.
- RNF-3.**— La comunicación entre el frontend y el backend debe realizarse a través de una API REST desarrollada con Django.
- RNF-4.**— El sistema debe delegar la extracción automatizada de datos a una cola de tareas en segundo plano utilizando Celery, evitando así bloquear la interfaz del usuario.
- RNF-5.**— El sistema debe integrarse con la API externa de Google Gemini para delegar el aprendizaje de selectores.
- RNF-6.**— La persistencia de datos, incluyendo la información de usuarios, resultados y configuraciones, debe gestionarse mediante una base de datos relacional SQLite.
- RNF-7.**— El sistema de extracción de datos debe simular tiempos de espera y comportamientos de un navegador

humano estándar para minimizar el riesgo de ser detectado y bloqueado.

RNF-8.— El sistema debe emplear proxies de WebShare.io, junto con una VPN de ProtonVPN, para enmascarar la IP del servidor de origen y evitar bloqueos del servicio.

RNF-9.— El sistema prioriza la precisión a la velocidad, mostrando tiempos de espera variables por las inteligencias artificiales en función de la solicitud y la página web.

RNF-10.— La extracción a gran escala debe operar de manera secuencial y garantizando un consumo ético del tráfico de la página web destino.

3.3. Ciclo de vida

Para el desarrollo de este proyecto, se ha elegido un modelo de ciclo de vida iterativo e incremental, dividiendo el trabajo en pequeños bloques. De esta forma se comprobaba su correcto funcionamiento antes de añadir nuevas funcionalidades más complejas. Se ha decidido esta forma de trabajar por los habituales imprevistos en los resultados aportados por la IA y las extracciones de datos automáticas por la web.

3.4. Diseño de la aplicación

En esta subsección se presenta en primer lugar el esquema de la base de datos, seguido por el diagrama de los casos de uso y, finalmente, los flujos internos de ejecución.

3.4.1. Diseño de la base datos

La base de datos funciona con el ORM de Django y se organiza en cuatro tablas principales, como se puede ver en el diagrama de la Figura 3.2. La primera de ellas es User, que separa el trabajo de cada cliente para garantizar la privacidad de sus búsquedas y datos en la aplicación. Esta parte está desarrollada utilizando el sistema de autenticación nativo de Django.

La tabla central del sistema es Website. Aquí se guardan los datos de cada una de las páginas web visitadas, como sus URLs, y se crea una conexión con el usuario que las ha solicitado. Además de estos atributos, Website se relaciona con la tabla Blueprint, que es la encargada de guardar los selectores utilizados para extraer la información de cada producto. Finalmente está la tabla Product, contiene los datos finales del producto, como precios o nombres, en formato JSON. Esta última tiene una conexión con Blueprint para saber qué selectores han obtenido ese producto, y otra con Website para identificar de qué página web proviene.

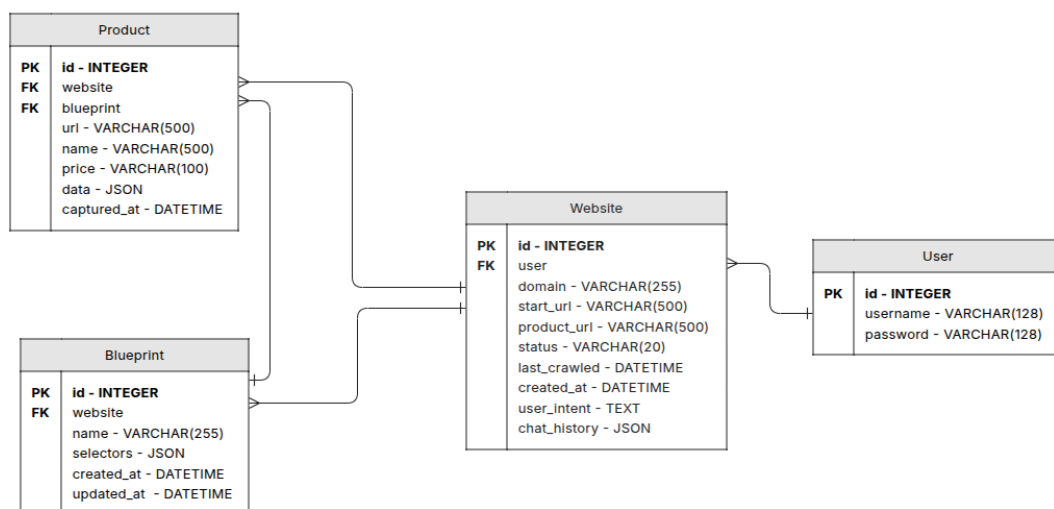


Figura 3.2: Diagrama de los objetos de la base de datos de la aplicación y su relación.

3.4.2. Diagrama de casos de uso

Lo primero que debe hacer el usuario es registrarse e iniciar sesión. Este paso es necesario para que cada persona tenga su propio espacio de trabajo, donde puede guardar su historial de búsquedas y gestionar sus propias extracciones. Una vez dentro, se accede a la parte principal del sistema, el extractor de datos, que se puede ver resumida en el diagrama de la Figura 3.3. Aquí, el usuario pone la URL de la categoría de la web que quiere analizar y dice qué datos busca, pudiendo hablar con un chatbot para que le ayude a concordar mejor su objetivo de búsqueda.

Con la configuración lista, el usuario no lanza la extracción de datos automática directamente, sino que el sistema hace una prueba con inteligencia artificial para mostrar cómo quedarían los datos sobre una URL de un producto de ejemplo. Si algo está mal, el usuario puede escribir un comentario correctivo explicando el error para que la IA lo corrija y vuelva a probar. Cuando el usuario esté satisfecho con el resultado, se lanza la extracción automática. Esta funciona en segundo plano para que el usuario no tenga que esperar y pueda seguir usando la aplicación tranquilamente sin notar esperas en la interfaz. Por último, en la sección de resultados se puede ver el historial, actualizar los datos que van llegando en el momento o borrar lo que ya no sirva para tenerlo todo organizado.

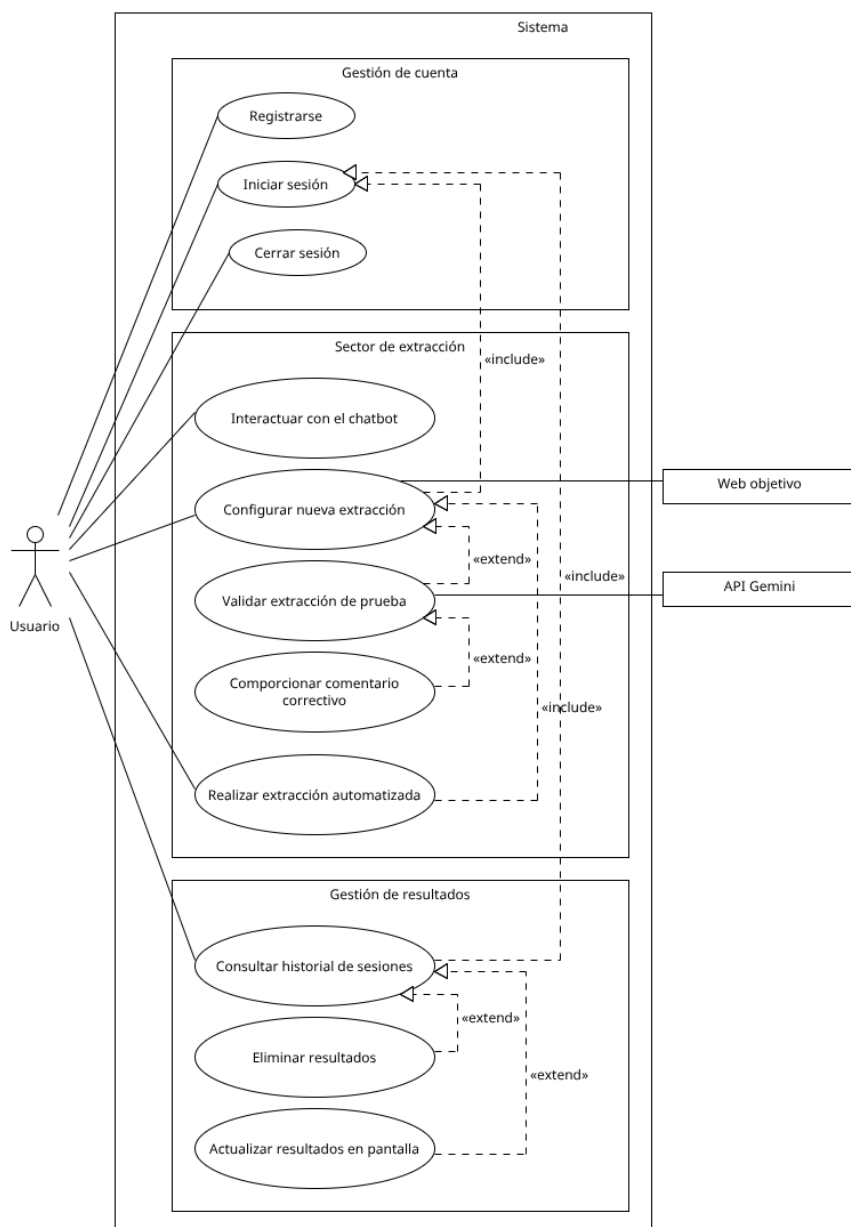


Figura 3.3: Diagrama de casos de uso donde se muestran las funcionalidades de la aplicación y cómo interactúan con el usuario y los servicios externos.

3.4.3. Diagramas de flujo

En esta subsección se muestran los procesos internos de la aplicación. Específicamente, se detallan el flujo de decisión frente a nuevas URLs y el ciclo repetitivo de la exploración automática de datos.

Enrutador de URLs y fase de aprendizaje

En la Figura 3.4 se ve cómo es el funcionamiento del enrutador de URLs. Se muestra cómo el sistema recibe las URLs y decide el camino apropiado dependiendo del caso. Si ya conoce la página, el sistema recicla las reglas que ya tiene para ir más rápido y evitar consultas innecesarias a la IA. Si la web es nueva o hay algún fallo, usa la inteligencia artificial para crear selectores desde cero antes de enseñar una prueba del producto.

Scraping automatizado

La Figura 3.5 enseña los pasos de la extracción automática. Esta muestra cómo el programa empieza a trabajar en segundo plano justo después de guardar la configuración, creando una lista de los enlaces aptos dentro de la URL de categoría solicitada. Luego entra en un bucle: visita las páginas y guarda los productos que sirven, repitiendo esto hasta terminar la lista y enseñar los resultados.

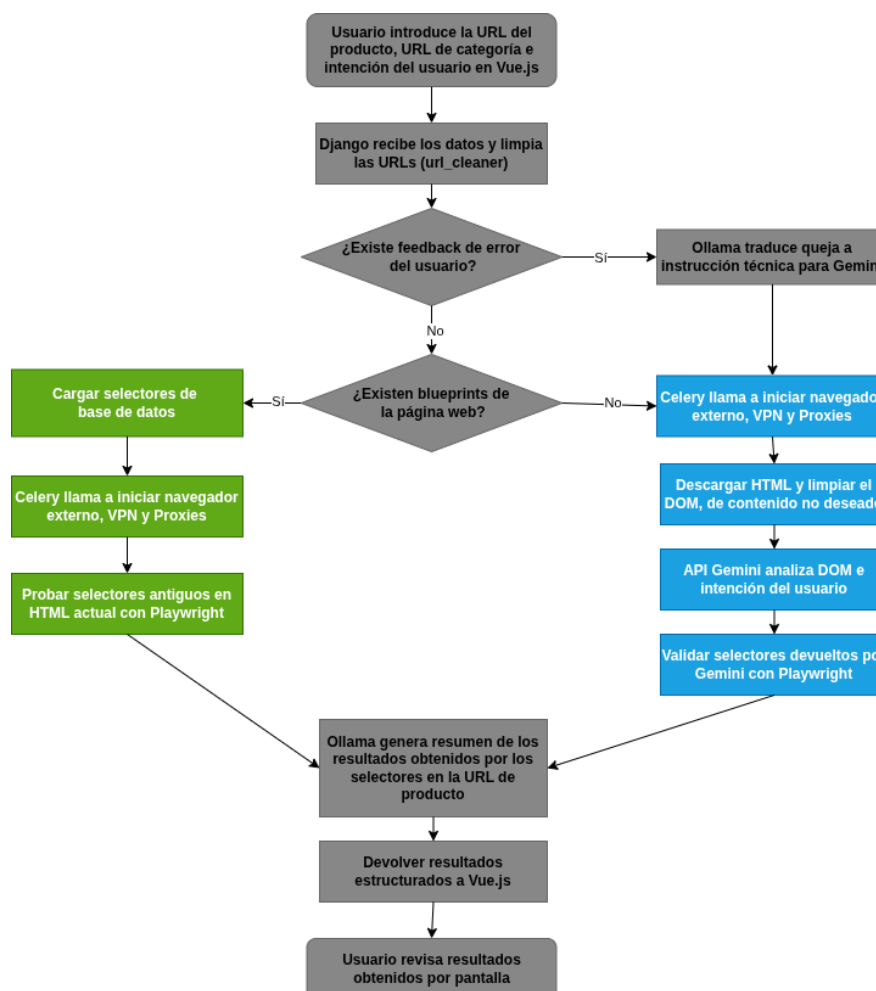


Figura 3.4: Diagrama de flujo del análisis de una URL, incluyendo las vías que ofrece el sistema para devolver la configuración apropiada de la búsqueda.

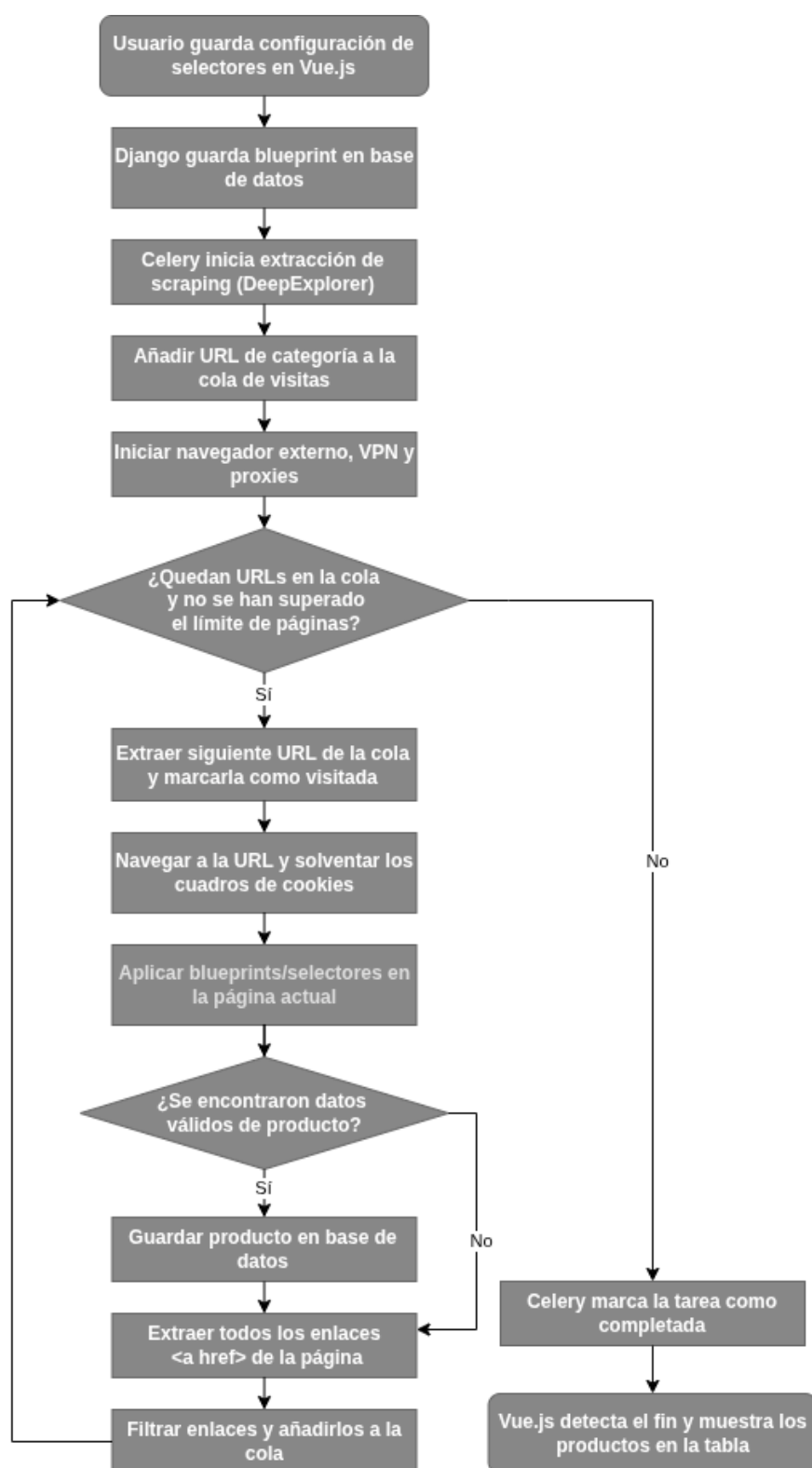


Figura 3.5: Diagrama de flujo sobre la extracción a gran escala, comenzando en el guardado de la configuración y terminando en la muestra de los resultados obtenidos.

IMPLEMENTACIÓN

4.1. Entorno de desarrollo y herramientas

4.1.1. Hardware y sistema operativo

El proyecto se ha desarrollado en un ordenador portátil personal. El ordenador cuenta con un microprocesador Intel Core i7-1255U, 16 GB de memoria RAM, 1 TB de almacenamiento y una tarjeta gráfica dedicada Nvidia RTX 2050 con 4 GB de VRAM. Las características de este son importantes para el proyecto, debido a que se utiliza una inteligencia artificial en local para la función de chatbot en la aplicación. De esta manera, la principal encargada de este trabajo es la tarjeta gráfica. La inteligencia artificial empleada es LLama 3, utilizada mediante la herramienta Ollama.

Como sistema operativo base se ha elegido Ubuntu 24.04.2 LTS. Esta decisión se tomó, en primer lugar, por la familiaridad y soltura con los entornos Linux adquirida durante los años de carrera. Además, este sistema destaca por la sencillez que ofrece su terminal para la instalación de dependencias, la gestión de servidores locales, y el manejo general de proyectos de programación orientados al desarrollo web y a la automatización.

4.1.2. Herramientas de desarrollo

El entorno de desarrollo integrado (IDE) principal utilizado para escribir y gestionar todo el código del proyecto ha sido Visual Studio Code, desarrollado por Microsoft. En este se ha escrito todo el código empleado en la aplicación desde el servidor hasta la interfaz web. Para mantener un historial del código e ir actualizando el proyecto se ha utilizado GitHub. Para ser más específicos se ha empleado la herramienta GitHub Desktop, para facilitar la creación de commits y la actualización del proyecto en un repositorio.

Como extra de seguridad al repositorio en GitHub, se mantuvieron copias del proyecto de manera periódica tanto en el ordenador local, como en la nube de Google Drive.

Como motor de base de datos para guardar información en la aplicación se ha seleccionado SQ-

Lite. El motivo principal de esta decisión es la gran ligereza de esta, además de la integración nativa en Python. Esta ligereza permite que los datos se almacenen en un único archivo, y así eliminando la necesidad de configurar servidores externos. La versión utilizada en el proyecto es la 3.45.1, decidida por su completa compatibilidad con el motor de mapeo objeto-relacional (ORM) de Django. Esta compatibilidad permite gestionar con gran eficiencia la autorización de los usuarios y sus sesiones en la aplicación.

Es en la fase de diseño en la que se empezó a plantear la forma en la que se guardarían los datos extraídos. En la mayoría de las aplicaciones, los datos guardados en la base de datos tienen unas tablas fijas. De esta manera es sencilla la organización de los datos pues las columnas que componen sus tablas son fijas y están ya establecidas. El problema con esta aplicación es que no puede utilizar este sistema, pues depende del usuario qué datos va a solicitar almacenar. Debido a que también hay una serie de campos que siempre se quieren tener fijos, se decidió implementar un esquema híbrido que combina la robustez de las tablas relacionales tradicionales con la flexibilidad del formato JSON. De esta forma, mientras que la estructura operativa de la aplicación y la privacidad de los usuarios se gestionan mediante SQL, los datos variables de cada catálogo se almacenan en campos de tipo JSON. Este formato resulta ideal al ser ligero, fácil de leer y permite la representación de estructuras complejas de datos sin necesidad de modificar la arquitectura de las tablas cada vez que el sistema añade una página web nueva.

Esta solución mixta ha resultado ser la mejor decisión técnica para el proyecto frente a otros sistemas de bases de datos como MongoDB. Aunque los sistemas NoSQL ofrecen una gran flexibilidad para datos no estructurados, carecen de la integración nativa que ofrece el sistema de autenticación de Django sobre una base de datos SQL. Al optar por este modelo en SQLite, se ha conseguido un equilibrio perfecto: la integridad y el orden de una base de datos relacional para la gestión de los clientes y la versatilidad de los documentos JSON para la extracción web.

Un factor determinante para consolidar esta elección fue la optimización de las peticiones a la inteligencia artificial, ya que el uso de JSON en la base de datos se alinea directamente con el funcionamiento del motor de extracción. El sistema se ha configurado para que Google Gemini trabaje nativamente en su “JSON Mode”, una funcionalidad que obliga al modelo a responder siguiendo un esquema estructurado que el software puede leer de forma automática. Esta estrategia reduce drásticamente las posibilidades de “alucinación” de la inteligencia artificial, asegurando que los selectores y datos generados sean siempre coherentes con el formato esperado. De esta manera, la arquitectura no solo hace que la extracción sea mucho más fiable y eficiente, sino que facilita la comunicación directa entre la inteligencia artificial y el resto del sistema, garantizando un flujo de datos limpio.

4.1.3. Entorno virtual y librerías

Para aislar el proyecto y evitar conflictos con otras aplicaciones del sistema, se ha creado un entorno virtual de Python (venv). Dentro de este entorno se han instalado únicamente las librerías necesarias para el funcionamiento del servidor y del motor de extracción. El backend está compuesto de las librerías de Django (v6.0.1) y Django REST Framework (v3.16.1), las cuales se han utilizado para levantar el servidor, gestionar el sistema de autenticación nativo y construir la API que expone y comunica los datos con la interfaz web.

Durante las primeras fases de diseño de la arquitectura, se planteó un modelo de ejecución síncrono, asumiendo que el flujo de extracción funcionaría de manera rápida y directa. Sin embargo, al integrar la automatización del navegador web y las llamadas a los modelos de inteligencia artificial, se cambió el formato, debido a que procesar un catálogo completo podía tardar una decena de minutos. Si esta carga de trabajo se ejecutara en el proceso principal de Django, la petición HTTP del usuario quedaría en espera hasta provocar un error por límite de tiempo (timeout) en el navegador, dando la sensación de que la aplicación se congela. Además, esto generaría un cuello de botella crítico: un solo usuario ejecutando un scraping bloquearía los recursos del servidor, impidiendo que otros clientes pudieran utilizar la web o guardar sus solicitudes al mismo tiempo.

Para solucionar este problema, se evaluaron diferentes alternativas. Inicialmente se consideró el uso de hilos de Python (threading). Esta idea se descartó porque los procesos lanzados de esta manera son inestables; si el servidor experimentase un reinicio, todas las tareas de los usuarios se perderían en el momento y no habría forma de monitorizar su progreso. La conclusión fue que el sistema requería desacoplar por completo la atención al cliente web de la ejecución del trabajo pesado, haciendo una buena idea el uso de una arquitectura basada en colas de tareas.

Para implementar esta solución se incorporaron Celery (v5.6.2) y Redis (v7.1.0). Celery es la librería encargada de gestionar la cola de tareas; actúa como un sistema de trabajadores (workers) independientes que operan en segundo plano. Gracias a Celery, el servidor puede recibir la petición del usuario, delegarla inmediatamente y devolverle el control para que siga usando la aplicación libremente sin esperas. Por otro lado, dado que Django y Celery son procesos independientes que se ejecutan por separado, necesitan un mecanismo rápido y fiable para comunicarse. Ese es el papel de Redis, una base de datos en memoria extremadamente rápida que funciona como intermediario o broker de mensajes. El flujo resultante es limpio y seguro: Django recibe la orden del usuario y deposita un mensaje en Redis; Celery lee dicho mensaje del intermediario, procesa la extracción del catálogo, y actualiza la base de datos una vez ha finalizado.

Playwright (versión 1.57.0) es la librería fundamental sobre la que se construye el motor de extracción del sistema. Se ha instalado con el objetivo de automatizar el control de un navegador web real, permitiendo entrar a las páginas objetivo y simular el comportamiento de un usuario humano para sortear posibles bloqueos de red. Aunque esta herramienta incluye opciones para extraer información

por sí sola, tras una fase inicial de investigación se llegó a la conclusión de que su rol más eficiente era trabajar en conjunto con un modelo de lenguaje, LLM. De esta forma, Playwright se encarga de cargar la web, limpiar el código HTML y entregárselo a Gemini para que este entienda el contexto. Una vez la inteligencia artificial deduce las reglas, Playwright actúa como el ejecutor, utilizando esos selectores para navegar y extraer los datos exactos tal y como lo haría una persona escribiendo en el teclado.

4.1.4. Integración de la inteligencia artificial

Para dotar al sistema de sus capacidades de razonamiento y procesamiento de lenguaje natural, se han integrado dos librerías fundamentales que comunican el código de python con los modelos de inteligencia artificial. En primer lugar, se ha utilizado el paquete oficial google-generativeai (versión 0.8.3) para establecer la conexión con la API externa de Google Gemini. Para habilitar este servicio sin costes de infraestructura, se ha configurado un entorno de trabajo bajo el nivel de acceso gratuito en la plataforma para desarrolladores Google AI Studio. Desde allí se gestionan las credenciales necesarias para que el sistema envíe el código web y reciba los resultados estructurados.

Por otra parte, para gestionar el asistente conversacional y las tareas de procesamiento interno sin depender exclusivamente de servicios en la nube, se ha implementado la librería ollama (versión 0.6.1). Este paquete funciona como puente de comunicación con el motor local de inteligencia artificial. Una de las grandes ventajas de esta herramienta es que permite cargar y alternar entre distintos modelos de forma dinámica según las necesidades del momento.

En concreto, para este proyecto se ha decidido utilizar dos versiones distintas de la familia de modelos de Meta: Llama 3.0 y Llama 3.2. Esta decisión se debe a la necesidad de equilibrar la velocidad y la capacidad de razonamiento del sistema. Por un lado, las interacciones en tiempo real del chat con el usuario se gestionan mediante Llama 3.2, un modelo más ligero y rápido que garantiza una conversación fluida y sin apenas tiempos de espera. Por otro lado, aquellas tareas que ocurren en segundo plano y requieren un mayor nivel de comprensión o razonamiento complejo como la interpretación del comentario correctivo del usuario (feedback) o el resumen de los selectores web se delegan a Llama 3.0, priorizando así la calidad del resultado por encima de la velocidad de respuesta.

4.2. Desarrollo de los módulos del sistema

4.2.1. Análisis de una página web

Análisis de la URL de ejemplo y generación de selectores

El análisis de la página web con la URL de ejemplo es el motor principal de la aplicación. Este es el proceso más largo de la aplicación, en el que convergen todas las tecnologías de inteligencia artificial,

scraping y navegación. Este flujo es complejo, ya que transforma una simple URL introducida por el usuario en información estructurada, extraída de forma automática. Para comprender esta arquitectura, es necesario seguir el ciclo de vida de la petición a través de las distintas capas del sistema, desde la interfaz hasta los modelos de lenguaje.

Interacción inicial y el asistente virtual

El ciclo de vida comienza en el componente `UrlAnalyzer.vue`. Esta vista ha sido diseñada para guiar al usuario a poder rellenar el formulario de búsqueda. En la parte superior, el usuario interactúa con un asistente virtual que maneja el modelo de lenguaje Llama 3.2. Este asistente está configurado de forma que introduzca al usuario a la interfaz y le oriente sobre cómo utilizarla mediante el chat. El usuario puede expresar sus intenciones en lenguaje natural, por ejemplo, “Quiero sacar el título y el precio de los relojes de esta página web”. En el momento que el usuario manda un mensaje, `Vue.js` envía una petición a Django a través de la ruta `/api/chat/`. Django procesa la petición y llama al modelo de lenguaje Llama 3.2 para que conteste al mensaje. En caso de que el modelo no haya identificado el objetivo del usuario, le solicitará que sea más preciso. En el momento en que el usuario haya introducido su objetivo de búsqueda y las dos URLs, la de ejemplo y la de categoría, el usuario deberá pulsar el botón “Iniciar Análisis”.

A partir de este momento, el usuario debe esperar con la pestaña en el navegador abierta, hasta que el sistema le notifique sobre la finalización del proceso de extracción. Dado que la lectura, renderizado y análisis de una página web mediante inteligencia artificial puede tomar varios minutos, mantener la conexión HTTP abierta provocaría un error de Timeout en el navegador. Para solucionarlo, `Vue.js` envía la petición inicial con el método POST, delegando el trabajo al servidor de Django y activando la función `pollStatus`, un mecanismo que pregunta al servidor cada dos segundos para conocer el estado del proceso en segundo plano, manteniendo la interfaz fluida y activa.

Enrutador de URLs y cola de tareas

La solicitud HTTP llega a la vista `analyze-url()` en el archivo `views.py` de Django. Antes de iniciar cualquier conexión externa, el servidor hace una limpieza de las URLs. Las URLs crudas pasan por el módulo `url-cleaner-web.py`, donde se aplican expresiones regulares y funciones de la librería `urllib.parse` para eliminar parámetros de rastreo o acortar enlaces complejos de plataformas como Amazon, para así tener URLs más acortadas, que llevan al mismo lugar, sin guardar parámetros innecesarios del usuario.

A continuación, la vista actúa como un enrutador de URLs consultando la base de datos. Si el sistema detecta que el dominio de la URL introducido ya posee un plano de extracción guardado, Blueprint, y el usuario no ha enviado un mensaje de corrección, la petición se enruta por la vía de reciclaje llamando a la función `task-test-blueprint()`. Esta vía añade la tarea a la cola de tareas de

Celery y ahorra tiempo de cómputo y consumo de tokens de la API de Gemini. Sin embargo, si el dominio es desconocido o se solicita un reintento de análisis, el sistema toma la vía del aprendizaje y llama a la función `task-train-domain()`, además de meter la tarea a la cola de tareas de Celery.

Django devuelve un identificador de tarea (`task-id`) a Vue.js, para que pueda comprobar el estado de la tarea constantemente con `pollStatus()`. Además de finalizar su ejecución síncrona y cediendo el control absoluto al gestor de colas Celery y a la base de datos en memoria Redis.

El proceso es recogido por un trabajador de Celery en el archivo `tasks.py`, el cual invoca al organizador de scraping (`organizer.py`). El organizador puede actuar de dos formas distintas, dependiendo de si se le llama para una extracción con blueprints en la base de datos, o si tiene que llamar a la API de Gemini para que los cree.

Vía de reciclaje

En caso de que hayan blueprints del dominio, lo primero que hace el organizador es lanzar un navegador Chromium con Playwright simulando la navegación de un humano. Para que el navegador se parezca a un humano este es lanzado con una configuración modificada, que incluye el cambio del `user-agent`, para simular un ordenador Linux. Además de esta configuración hay un sistema para la protección de la IP de origen, contando con una conexión a la API del proveedor de proxies WebShare, y el uso de una VPN de ProtonVPN.

Una vez que el navegador carga la página web y espera a que el tráfico de red se estabilice, este se encarga de los paneles de cookies que saltan en pantalla. Estos paneles no permiten al navegador acceder a los datos de la página web, y es por ello que el navegador utiliza una serie de estrategias para despejar la pantalla. Algunas de estas estrategias incluyen: aplicar selectores CSS de plataformas conocidas como OneTrust o utilizar una heurística que haga click en cuadros de texto flotantes con botones que incluyan las palabras (“Agree” o “Allow”).

Ya con la pantalla despejada, y la conexión asegurada, el navegador inserta los selectores CSS almacenados, y manda los resultados obtenidos a Celery. Ahora, en el proceso de entrega de los resultados, se utiliza el modelo de lenguaje LLaMA 3, para presentar los resultados en texto plano. Celery manda los resultados a Django, y este a Vue.js, además de actualizar el estado de la tarea a “analyzed”. De esta manera Vue.js reacciona al cambio de estado y se los presenta al usuario.

Vía del aprendizaje

Por otro lado tenemos el caso en el que el dominio sea nuevo, y por tanto no tengamos planos de extracción almacenados. Esta vía sigue el mismo sistema que la vía del reciclaje, exceptuando que cuando la página está renderizada y libre de obstrucciones, lo que hace es descargar todo el DOM (Document Object Model) de la página. El organizador le entrega el DOM al componente selector-

learner.py encargado de obtener los selectores CSS y expresiones XPath.

Este componente comienza ejecutando un script de JavaScript, encargado de limpiar el DOM, para dejar solo la información del HTML que interesa. De esta manera podemos entregar a Gemini un DOM filtrado, que ocupa mucho menos y hace viable la cantidad de tokens utilizados. Este script elimina etiquetas estéticas y de comportamiento (<script>, <style>, <svg>), elimina atributos no esenciales e identifica y destruye contenedores <div> o que estén vacíos, reduciendo drásticamente el peso del código sin perder la estructura semántica.

El proceso sigue con el envío a la API de Gemini, para utilizar el modelo Gemini 2.5 Flash. A este modelo se le envía un prompt diseñado con las características necesarias para realizar un scraping web profesional, incluyendo el DOM filtrado y el objetivo de búsqueda del usuario. A diferencia de un scraper web tradicional que requiere reglas fijas, Gemini analiza visual y semánticamente el documento y devuelve un objeto JSON con propuestas de selectores CSS o expresiones XPath resistentes a ligeros cambios en la página web. Además, se configura la llamada a la API habilitando el formato de salida JSON (response-mime-type="application/json"). Esta configuración demuestra por qué se ha seleccionado Gemini para esta tarea, pues este modelo de lenguaje permite un formato de salida en JSON nativo, mejorando la credibilidad de la respuesta.

Para asegurar la máxima fiabilidad, el código no confía ciegamente en la respuesta de la IA. Inmediatamente después de recibir los selectores generados, el navegador de Playwright los prueba en milisegundos en la página web abierta para verificar si realmente apuntan a elementos válidos y extrae un valor de muestra.

Finalmente, los selectores verificados retornan a Celery, donde siguen el mismo curso que en la vía de reciclaje, donde sus valores de muestra son presentados al usuario.

Verificación de los resultados

El siguiente paso después de cualquiera de las dos vías, es la verificación de los resultados obtenidos. El usuario tiene la posibilidad de, en el caso de no estar de acuerdo con los resultados mostrados, escribir un comentario correctivo, dirigiendo al sistema a un ciclo por la vía del aprendizaje. Sin embargo, si el usuario está conforme con los resultados, debe pulsar el botón de "Guardar Configuración", y dar inicio al proceso de la extracción automática a gran escala.

4.2.2. Extracción automática a gran escala

El motor de exploración

En cuanto el usuario comprueba en la interfaz que los selectores propuestos por Gemini, o reciclados por el sistema, son correctos y extraen adecuadamente los campos solicitados, el ciclo de

análisis da paso a la fase de extracción automática a gran escala. Este proceso abandona el enfoque de una única página para desplegar un motor de navegación recursivo capaz de recorrer y descargar categorías enteras de una tienda online de forma autónoma.

Guardado del plano de extracción

La transición comienza cuando el usuario pulsa el botón “Guardar Configuración” en la interfaz de análisis. Esta acción envía una petición HTTP POST a la ruta `/api/guardar-blueprint/` del servidor Django, transmitiendo los datos de la configuración y activando la función `save-blueprint`. Esta función intercepta la información y hace uso del ORM de Django para guardarla en la base de datos SQLite. En concreto, el sistema genera y almacena un nuevo objeto Blueprint (plano de extracción) en la base de datos, asociado a la página web objetivo. A partir de este momento, el servidor delega la extracción a Celery, llamando a la función `task-scrape-domain()`, la cual inserta una nueva tarea en la cola de tareas de Celery.

Inicialización del motor de exploración

Esta tarea actúa como puente hacia el organizador de scraping (`organizer.py`). Este invoca la función encargada de iniciar la exploración de la categoría de la página web indicada por el usuario. Es en este punto donde se llama al encargado de la extracción a gran escala, la clase `DeepExplorer`. A esta clase se le introducen los parámetros fundamentales para su funcionamiento: la URL de la categoría por donde debe empezar a navegar, la Blueprint con los selectores confirmados, y un límite de seguridad máximo de páginas a visitar para prevenir bucles infinitos durante la extracción.

Antes de comenzar a explorar, la clase `DeepExplorer` replica las mismas medidas de seguridad antibloqueo empleadas en la función de análisis de la url de ejemplo. Para proteger la identidad del servidor, se conecta a una IP rotatoria a través de la API de Webshare, y lanza el navegador Chromium mediante Playwright con la configuración modificada del user-agent.

Algoritmo de la navegación recursiva

El núcleo de este explorador funciona mediante un algoritmo de búsqueda en anchura iterativa. Para ello, se apoya en una lista que actúa como cola de próximas URLs a visitar, y un conjunto que registra las páginas ya analizadas para garantizar que el sistema no procese la misma URL dos veces. Al arrancar, la cola se inicia únicamente con la URL de la categoría introducida por el usuario.

El proceso principal consiste en un ciclo que extrae el primer enlace de la cola y redirige al navegador. Para simular el comportamiento de un usuario humano y evadir las protecciones de los servidores web, como los limitadores de peticiones por segundo, el explorador introduce un tiempo de espera asíncrono, pausando la ejecución de forma aleatoria entre 1,5 y 4 segundos antes de interactuar con

la página web. Una vez transcurrido este margen, el explorador despliega de nuevo las estrategias de eliminación de paneles de cookies para garantizar que la pantalla quede despejada.

Con la página limpia, el explorador aplica los selectores CSS sobre el DOM actual. Si el sistema consigue extraer información válida, deduce de forma automática que se encuentra en una página de producto y procesa los datos. Si por el contrario, los selectores devuelven un resultado vacío, el motor asume que está ante una página de listado o categoría.

En ambos escenarios, el explorador necesita descubrir nuevos enlaces para continuar su misión. Para lograrlo, inyecta un pequeño script de JavaScript que recopila todos los enlaces (atributos href de las etiquetas <a>) presentes en el documento. Para evitar que el explorador se salga de la tienda online o acabe analizando secciones irrelevantes como las redes sociales, cada enlace descubierto pasa por un filtro que comprueba si está contenido dentro de la URL de la categoría que especificó el usuario. Solo las URLs válidas y no visitadas previamente se añaden al final de la cola de pendientes.

Almacenamiento dinámico

Según el explorador va extrayendo datos válidos de los productos, necesita almacenarlos de forma persistente. Este paso presenta un importante desafío de arquitectura: Playwright funciona sobre un bucle de eventos completamente asíncrono, mientras que el ORM de Django opera de forma síncrona. Ejecutar una consulta a la base de datos directamente detendría el hilo principal, provocando un bloqueo crítico en el servidor. Para solucionar este conflicto, la función de guardado se encapsula utilizando el decorador nativo de Django `@sync-to-async`, creando una capa de adaptación segura que permite guardar la información en la base de datos sin interrumpir al explorador.

Debido a que el usuario solicita la extracción utilizando lenguaje natural en el chat con el asistente virtual, los nombres de los campos extraídos son variables (el usuario puede haber pedido “precio”, “price” o “coste”). Para manejar esta variabilidad, el algoritmo de guardado incorpora un mapeo dinámico. Antes de guardar el producto en la base de datos, el sistema busca palabras clave representativas como “nombre”, “título” o “name”. Si no encuentra una coincidencia exacta, utiliza el primer valor disponible como nombre representativo del producto, evitando así que la tabla de la interfaz muestre registros anónimos. Finalmente, los datos se guardan mediante el ORM de Django, relacionándolos con su respectiva Blueprint.

Cuando la cola de enlaces se vacía o se alcanza el límite de seguridad de páginas a visitar estipulado, el navegador se cierra y el trabajador de Celery marca el proceso como exitoso. De forma paralela, Vue.js en la interfaz, que seguía consultando el estado de la tarea mediante `pollStatus()`, detecta la finalización, oculta la pantalla de carga y renderiza inmediatamente la tabla con los productos extraídos, dejándolos listos para que el usuario pueda consultarlos y gestionarlos.

4.2.3. Página de inicio

El usuario entra a la aplicación con la página de inicio, que está diseñada como la presentación al proyecto. Esta página de inicio, desarrollada en el archivo `LandingPage.vue`, menciona el lema del proyecto “Convierte cualquier Web en una API de Datos”, además de los pilares principales del proyecto: la navegación segura sin bloqueos; el uso de inteligencia artificial para el análisis del DOM de la página web; y la extracción automatizada en segundo plano. Entre los botones que componen esta página está el botón principal de “Lanzar Aplicación”, el cual se refiere a la funcionalidad principal del proyecto de analizar una búsqueda de datos y comenzar su extracción. Además de ello hay otros botones como el de “About”, que lleva al usuario a una página que resume qué contiene el sistema. También está el botón de “Features”, que lleva al usuario a una página que explica las características del proyecto.

4.2.4. Sobre el proyecto

Además de las funcionalidades principales de la aplicación, se ha creado una página informativa en el componente `AboutPage.vue`. Esta sección sirve como una pequeña documentación, ofreciendo al usuario una visión clara de lo que hace este trabajo de fin de grado. En ella se presenta a “Kosmos” como un sistema de extracción web moderno y se explica su principal ventaja frente a los programas de scraping clásicos: el uso de inteligencia artificial para entender el código de las páginas y crear reglas de búsqueda que no se rompan fácilmente. También incluye los datos básicos del proyecto, como el autor y el grado universitario.

Además de ser un resumen, sirve como esquema para entender el funcionamiento del proyecto. En este se muestra el recorrido que hacen los datos. Desde el usuario con la interfaz web de `Vue.js`, pasando por el servidor `Django` y el gestor de tareas `Celery`, hasta el motor de scraping de `Playwright` y la API de `Gemini`. Además se muestra una lista de las tecnologías utilizadas en el proyecto, para poder entender mejor todas las piezas del software.

4.2.5. Características

La página informativa “Características” del componente `FeaturesPage.vue` menciona las tres características que mejor definen a este proyecto: la generación de selectores CSS y expresiones XPath con `Gemini`; la navegación segura gracias a los proxies y a la VPN; y el procesamiento asíncrono de las tareas en segundo plano, como el aprendizaje de selectores o la extracción automática.

Además de estas características, se muestra una tabla comparativa, para mostrar las ventajas de este proyecto versus un scraper tradicional en diversos aspectos. Para empezar, en el mantenimiento: el scraper tradicional tiene un alto coste, pues se rompe con ligeros cambios en la web, mientras este proyecto tiene un coste bajo, pues la IA se adapta a los cambios y genera los nuevos selectores.

Segundo, los bloqueos: un scraper tradicional utilizaría su IP de origen siendo así bloqueada rápidamente, mientras este proyecto usa varios proxies y VPN para evitar los bloqueos de red. Tercero, la configuración: un scraper tradicional tendría que revisar lentamente el código para buscar lo que quiere, mientras este proyecto tiene un chatbot con el que el cliente explica qué es lo que busca, y la IA se lo encuentra.

4.2.6. Registro e inicio de sesión

Desde la página de inicio, el usuario puede pulsar el botón “Lanzar Aplicación”. Este activa el sistema de identificación de usuario. Para saber si el usuario ha iniciado sesión, este sistema utiliza un token almacenado en la memoria local del navegador, `localStorage`. De esta manera, si el sistema detecta que el usuario ya está autenticado, le dirige directamente a la ventana principal de análisis en el componente `UrlAnalyzer.vue`, omitiendo las ventanas de validación. Por el contrario, si es la primera vez que accede o su sesión ha expirado, el sistema muestra la ventana del componente `LoginPage.vue` para que el usuario inicie sesión.

Desde la ventana de inicio de sesión, hay un botón subrayado para que en el caso de que el usuario nunca se haya registrado, pueda crear una cuenta. En esta el usuario puede rellenar un formulario con un nombre de usuario y contraseña. El sistema ofrece este formulario desde el componente `SignUpPage.vue`, el cual envía una petición HTTP por el método POST hacia la ruta `/api/register/` del servidor de Django. El servidor procesa esta solicitud utilizando el modelo de autenticación nativo de Django. Este modelo de autenticación aprovecha la función `(user.objects.create-user)`, la cual aplica un algoritmo de hashing seguro a la contraseña antes de guardarla en la base de datos SQLite. De esta manera, hay una seguridad de los datos personales mucho mayor frente a guardar las credenciales en texto plano, confiando la seguridad en las herramientas nativas de Django y así evitando las posibles vulnerabilidades.

Cuando el registro del usuario está completado, la aplicación redirige al usuario a la página de inicio de sesión, para que este pueda introducir sus credenciales y con ello tener acceso a todas las funcionalidades de la aplicación.

4.2.7. Notificaciones de la aplicación

La aplicación web cuenta con un par de notificaciones para sus principales procesos: la extracción de la url de ejemplo y la extracción automática a gran escala. Este sistema de notificaciones mejora la experiencia del usuario, permitiendo a este continuar con otras actividades, mientras la aplicación está trabajando en segundo plano.

Esta decisión de diseño se ha tomado debido al tiempo moderado de respuesta que requiere la aplicación, tanto en la extracción de prueba como en la extracción a gran escala.

Al abrir la aplicación, esta pregunta al usuario si quiere recibir notificaciones. En caso de aceptar, el usuario recibirá notificaciones de sistema operativo. Sin embargo, si el cliente rechaza las notificaciones de sistema operativo, este será notificado con un cuadro de texto verde en la esquina inferior derecha de la interfaz, junto con el mensaje “Extracción finalizada”.

4.2.8. Listar páginas web y mostrar productos

Los resultados obtenidos de la extracción a gran escala se visualizan en otro componente del sistema, `WebList.vue`. En esta ventana se le proporciona acceso al usuario a los datos de su historial de búsquedas, junto con sus extracciones de datos. El diseño de esta ventana se divide en dos espacios interactivos, que se comunican con el servidor y la base de datos.

Nada más entrar en la interfaz, la aplicación carga la lista de Websites de la base de datos mediante la función `fetchWebs`. Esta función manda una petición al servidor a través de la ruta `/api/webs/`, donde Django filtra la base de datos para mostrar solo las páginas que pertenecen al usuario que ha iniciado sesión. Antes de hacer click en la Website que se quiere visualizar, se muestra su estado, pudiendo ser “finalizado” si la extracción a gran escala se ha completado o “en proceso” si aún no ha terminado y el sistema aún no ha notificado al usuario.

El usuario puede elegir el dominio del que quiere visualizar sus datos. Esta acción activa el método `selectWeb` de `Vue.js`, que envía una petición GET a la ruta `/api/productos/` del servidor Django. Esta petición devuelve la lista de búsquedas que ha tenido el usuario sobre esa Website, junto con los resultados de la extracción a gran escala. Los resultados de los productos se muestran en una tabla con las columnas de los atributos que el cliente ha solicitado, como por ejemplo: el nombre, el precio y la descripción. Además, para los usuarios que quieran mayor detalle, cada fila permite desplegar un visor de datos en bruto para ver la información completa en formato JSON.

Para mantener al usuario actualizado con los datos en el momento, el sistema cuenta con un refresco de los datos de `WebList.vue` mediante un botón en el espacio superior izquierdo de la interfaz, que vuelve a recopilar los datos de la base de datos, para mostrar los datos más recientes.

4.2.9. Borrar páginas web

La aplicación permite eliminar datos almacenados de los dominios desde el componente `WebList.vue`. Al pulsar el botón “Borrar Scraping”, desde `Vue.js` la función `deleteWeb`, envía una petición HTTP DELETE hacia la ruta `/api/webs/<dominio>/`. Django responde con la vista `delete-website` que recibe la petición y valida que la Website especificada pertenece al usuario que solicita el borrado “`user=request.user`”. Esta restricción garantiza que ningún usuario pueda borrar datos de otros usuarios, proporcionando un aislamiento total de los datos entre los distintos clientes de la aplicación.

Esta eliminación se realiza con gran eficiencia, utilizando el ORM de Django, por la propiedad “on-delete=models.CASCADE”, configurada en los objetos secundarios: Blueprint y Product. De esta manera, al borrar de raíz la Website seleccionada, hace una limpieza automática de los objetos secundarios conectados a ella en SQLite. En una única transacción se eliminan los selectores CSS generados por Gemini, los historiales de los chats y los catálogos de objetos vinculados. Así se evitan los registros huérfanos y se mantiene la coherencia de los datos. Tras el borrado de los datos en el servidor, Vue.js actualiza su estado y refresca la ventana para la correcta visualización de la interfaz.

4.3. Decisiones técnicas

Durante la fase inicial de investigación del proyecto, se evaluaron diversas herramientas clásicas de extracción de datos, como Scrapy, Selenium o BeautifulSoup. Aunque son estándares en la industria, presentaron limitaciones críticas para un sistema que pretende ser universal. BeautifulSoup, por ejemplo, carece de capacidad para ejecutar JavaScript, lo que lo invalida para la inmensa mayoría de tiendas web modernas que son altamente reactivas. Selenium, por su parte, resultó ser una herramienta excesivamente pesada y con tiempos de respuesta lentos. De esta forma, se optó finalmente por Playwright en su modalidad asíncrona, ya que la versión síncrona no ofrecía el rendimiento necesario para gestionar el flujo interno del servidor. Con Playwright también se probó su herramienta de generación de código, la cual graba los clics del usuario y escribe los selectores automáticamente. Sin embargo, esta vía se descartó rápidamente al comprobar que generaba selectores extremadamente frágiles, muy acoplados a la estructura visual de ese instante y que se rompían ante el menor rediseño de la página.

Tras descartar la generación manual de selectores, se exploró la viabilidad de crear reglas mediante programación heurística y expresiones regulares. La idea era desarrollar un algoritmo capaz de identificar patrones comunes en el texto, como los símbolos del dolar (\$) para localizar precios. Este enfoque fue desechado debido al caos estructural del desarrollo web actual, donde cada programador escribe y organiza el código HTML a su manera. La heurística demostró ser capaz de encontrar cifras con formato de precio, pero era totalmente ciega al contexto: resultaba imposible enseñarle a diferenciar de forma fiable cuál era el precio principal del producto objetivo y cuáles pertenecían a una lista lateral de artículos recomendados. Esta limitación confirmó que se necesitaba capacidad de comprensión semántica. Siguiendo esta línea, se probó a utilizar librerías especializadas en extracción con IA como LangExtract, pero se descartaron porque introducían un nivel de abstracción demasiado alto, ocultando el proceso de comunicación con la API y reduciendo el control sobre los errores.

Una vez establecido que la inteligencia artificial sería el núcleo del sistema, se produjo el cambio de paradigma más importante en la arquitectura del proyecto. En el planteamiento original, se diseñó la herramienta para que Gemini actuara como el recolector directo de la información; es decir, la idea era iterar por cada enlace de la tienda y enviar el código HTML de cada producto a la inteligencia artificial

para que esta devolviera los datos. Rápidamente se comprobó que esta aproximación era inviable y extremadamente ineficiente. Procesar el código de cientos de páginas web suponía un consumo de tokens inasumible, saturando los límites de la API y multiplicando los tiempos de espera del sistema. Ante este problema, se decidió cambiar radicalmente la misión de la inteligencia artificial: Gemini pasó de ser un recolector masivo a ser un creador de selectores. Bajo este nuevo enfoque, el sistema solo consulta a la inteligencia artificial una única vez enviándole un producto de ejemplo. Gemini deduce las rutas de extracción y, a partir de ahí, es el motor local de Playwright el que recolecta el resto del catálogo de forma autónoma y sin gastar un solo token adicional.

Incluso con este cambio, donde la inteligencia artificial solo analiza una página de prueba, surgió un nuevo reto técnico: el excesivo peso del código fuente. Enviar a Gemini el DOM entero de esa única página seguía siendo problemático, ya que las webs modernas contienen enormes bloques de código destinados a scripts de analítica, menús desplegados ocultos y extensos textos legales de cookies. Esta sobrecarga de información generaba ruido irrelevante que confundía al modelo y rozaba el límite de tokens permitido por petición. La decisión final para optimizar por completo la comunicación con la API fue desarrollar un script intermedio que poda todos estos elementos visuales e interactivos. De esta forma, se entrega al modelo un DOM filtrado, garantizando que la generación de selectores sea precisa y económica.

Otro reto técnico significativo surgió en la gestión de los tiempos de carga de la navegación automatizada. Inicialmente, el sistema utilizaba la función de Playwright (`network-kindle`) que espera a que el tráfico de red se detenga por completo antes de capturar el código, asegurando así la carga de todos los elementos dinámicos. No obstante, al realizar pruebas en entornos reales como la página web de la compañía Multiópticas, se descubrió que algunos servidores mantienen conexiones abiertas de forma perpetua para refrescar publicidad. Esto provocaba que el motor explorador se quedara congelado de manera indefinida esperando un silencio en la red que nunca iba a ocurrir. Para solucionarlo, se descartó la espera pasiva a cambio de un modelo mixto: el sistema sigue intentando esperar a que la red se estabilice, pero incluye un tiempo de espera máximo programado por página. Si se alcanza ese límite temporal, se procesa la información disponible, garantizando que la ejecución de la aplicación nunca se bloquee.

Por último, se tomó una decisión de diseño consciente respecto a la concurrencia y paralelización del sistema de extracción automatizada. A nivel técnico, el proyecto podría haberse diseñado para utilizar múltiples hilos de ejecución, permitiendo analizar decenas de URLs de forma simultánea. Esto habría multiplicado notablemente la eficiencia del sistema, reduciendo los tiempos de respuesta. Sin embargo, se optó por un modelo secuencial por dos motivos fundamentales. El primero responde a una cuestión de ética digital: lanzar peticiones concurrentes podría sobrecargar los recursos del servidor destino, asimilándose a un ataque de denegación de servicio. Tratándose de un proyecto de carácter académico y no lucrativo, el respeto al tráfico de las páginas web es un requisito indispensable. El segundo factor determinante es la supervivencia del propio explorador. Las páginas web actuales

cuentan con rígidos sistemas antibot que detectan inmediatamente los patrones de peticiones en paralelo. Al operar de forma controlada y secuencial, el sistema simula mejor el comportamiento humano, esquivando estos bloqueos y manteniendo un porcentaje alto de extracciones exitosas.

4.4. Complicaciones encontradas

El desarrollo de un sistema de extracción automatizada y su integración con modelos de inteligencia artificial ha presentado numerosos desafíos técnicos a lo largo del proyecto. A diferencia de un entorno académico controlado, el despliegue de esta aplicación en internet implica interactuar con arquitecturas web impredecibles, sistemas de seguridad restrictivos y el comportamiento no determinista de los modelos de lenguaje. La resolución de estos obstáculos ha requerido constantes iteraciones de diseño y cambios en el código fuente.

El primer gran obstáculo surgió al realizar las primeras pruebas de navegación automatizada con Playwright. Las tiendas de comercio electrónico modernas implementan severos sistemas de mitigación de bots, como Cloudflare o Datadome, diseñados específicamente para detectar y bloquear accesos automatizados. En las fases iniciales, tras realizar unas pocas peticiones consecutivas, la dirección IP del servidor quedaba inmediatamente bloqueada. El sistema recibía códigos de error de acceso denegado o se quedaba atrapado en bucles infinitos de resolución de captchas, interrumpiendo por completo la extracción. Estos problemas siguen siendo existentes, pero en menor medida gracias a la arquitectura del proyecto.

Una vez estabilizada la conexión de red, las tareas en segundo plano de Celery generaron sus propias complicaciones. Aunque Celery es una herramienta robusta, su integración simultánea con Django, SQLite y un navegador asíncrono como Playwright provocó varios fallos de concurrencia. El primer problema detectado fue el error de “base de datos bloqueada” (database is locked). Al ser SQLite una base de datos de un solo archivo, no maneja bien las escrituras concurrentes. Cuando un usuario intentaba guardar una configuración desde la interfaz al mismo tiempo que un trabajador de Celery intentaba guardar un catálogo de cien productos, el sistema colapsaba. La solución técnica pasó por limitar la concurrencia de los trabajadores de Celery y ajustar los tiempos de espera de las conexiones a la base de datos dentro de la configuración de Django, asegurando que las escrituras se realizaran de forma ordenada.

Finalmente, el desafío más complejo a nivel lógico fue gestionar la fiabilidad de la inteligencia artificial. Trabajar con modelos de lenguaje como Gemini implica lidiar con su naturaleza probabilística, lo que en ocasiones lleva a “alucinaciones”. En el contexto de este proyecto, una alucinación consiste en que la inteligencia artificial, al no encontrar la ruta exacta de un precio o un título en el HTML, se inventase un selector CSS o una expresión XPath inexistente para intentar cumplir con la petición. Cuando el explorador intentaba utilizar estas reglas falsas en el navegador, el sistema fallaba en cade-

na devolviendo resultados vacíos. Esto resultaba inaceptable para un proyecto que requiere precisión absoluta.

Para evitar que el modelo tuviera alucinaciones y asegurar su precisión, se aplicaron varias estrategias. En primer lugar, además de filtrar el código de la página web, se ajustaron cuidadosamente las instrucciones enviadas a Gemini. Se le pidió que verificara siempre la información antes de contestar y que devolviera los datos exclusivamente en un formato estructurado JSON, sin añadir texto conversacional. Junto a esto, se configuró la llamada a la API con el parámetro `temperature=0.0`. Este ajuste es fundamental, ya que elimina la creatividad de la inteligencia artificial, forzándola a dar respuestas mucho más literales, exactas y deterministas.

A pesar de estas medidas preventivas, la solución definitiva para garantizar la fiabilidad del sistema fue darle al usuario la capacidad de corregir los errores. Si al hacer la prueba el usuario nota que los datos extraídos no son los deseados, puede indicarlo de forma conversacional a través del chat. Es aquí donde Llama 3 interpreta la corrección del usuario, reformula la instrucción añadiendo el contexto del fallo, y manda a Gemini a buscar los selectores de nuevo. Este ciclo de retroalimentación logró convertir un proceso impredecible en una herramienta precisa y fiable.

EXPERIMENTOS

En este capítulo, se presenta el diseño y ejecución del estudio de usuarios realizado, con el objetivo de evaluar la usabilidad de las características principales de la aplicación web Kosmos.

5.1. Estudio de usuarios

Esta sección describe el estudio realizado con 18 participantes para evaluar la usabilidad de Kosmos, su utilidad real y recopilar sugerencias de mejora (las imágenes con mayor tamaño se pueden encontrar en el Apéndice A). La muestra, seleccionada en entornos sociales y universitarios, poseía un nivel básico en el uso de aplicaciones web.

El procedimiento consistió en un cuestionario inicial demográfico y de motivaciones, seguido de un recorrido guiado por las tareas principales: registro, extracción de datos mediante el chatbot, ajuste de resultados por IA y gestión de la información obtenida. Al completar cada tarea, los participantes las valoraron en una escala de 1 a 5 puntos. Finalmente, se realizó una evaluación sobre la facilidad de uso, navegación y satisfacción general.

5.1.1. Consentimiento

Antes de comenzar con el estudio, se verificó que todos los participantes aceptaban contestar a las preguntas de forma anónima. La Figura 5.1, muestra que todos los participantes aceptaron su consentimiento de manera libre y voluntaria.



Figura 5.1: Consentimiento.

5.1.2. Datos demográficos y experiencia en búsqueda web

Esta subsección detalla el perfil sociodemográfico de los encuestados y su nivel de familiaridad previo con las herramientas de búsqueda de información web.

Distribución por edad

Se les pidió a los participantes que rellenaran una pregunta sobre su edad para obtener los datos demográficos del estudio. Se puede observar en la Figura 5.2a que el 66.7 % de los participantes se sitúan entre los 18 y los 29 años, el 11.1 % entre los 30 y los 45 años, y el 22.2 % entre los 46 y los 65 años.



Figura 5.2: Datos demográficos de los participantes.

Género

La Figura 5.2b) muestra la distribución de género de los participantes, siendo esta del género masculino con un 61.1 % y femenino con un 38.9 %.

Experiencia en búsqueda web

En la Figura 5.3 se expone el nivel de experiencia de los usuarios realizando búsquedas en internet. Se destaca un mediano y alto nivel de experiencia entre los participantes.

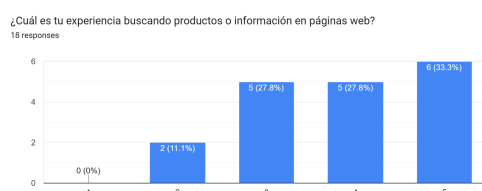


Figura 5.3: Experiencia en búsqueda web.

5.1.3. Motivaciones y preferencias

En la siguiente sección del formulario se les preguntó a los participantes sobre la razones y métodos principales para utilizar una aplicación de extracción automática de datos web.

Motivaciones para usar web scraping

La Figura 5.4a expone que la motivación principal es la comparación de precios, seguida por el ahorro de tiempo frente a métodos manuales y la recopilación de información académica.

Preferencia de interacción

Según la Figura 5.4b, los encuestados muestran una clara tendencia hacia preferir conversar con un chatbot antes que buscar ellos a mano.

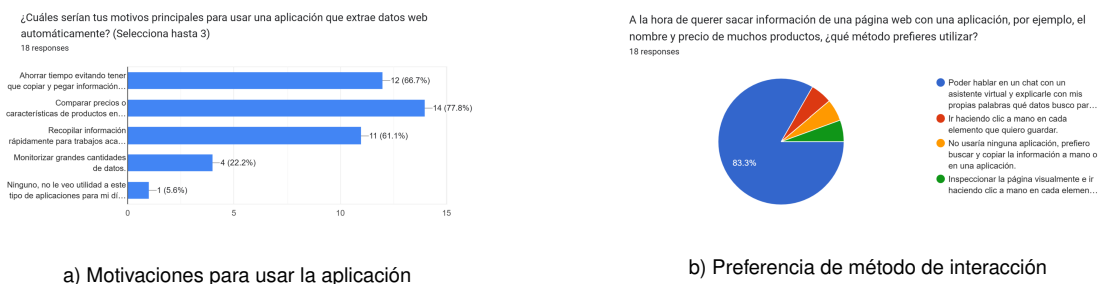


Figura 5.4: Motivaciones de uso y preferencia de interacción de los usuarios.

Tipos de datos a extraer

Los participantes mostraron una clara tendencia a la hora de decidir su tipología favorita (ver Figura 5.5), siendo esta los catálogos de tiendas online.

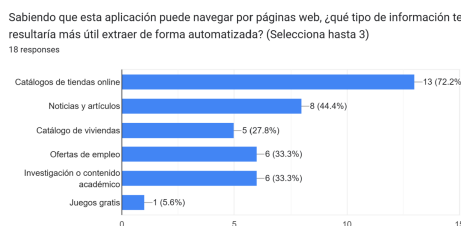


Figura 5.5: Tipos de datos a extraer.

5.1.4. Recorrido guiado de tareas

Cada participante siguió un recorrido de las funcionalidades principales del sistema. Tras cada una, se evaluó la dificultad encontrada sobre una escala del 1 al 5, siendo 1 muy difícil y 5 muy fácil.

Configuración y análisis de una página web

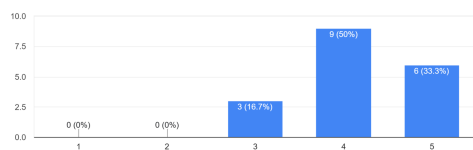
La Figura 5.6a muestra que la mayoría de los usuarios consideró sencillo el proceso inicial de introducir las URLs y lanzar el análisis.

Refinamiento de los resultados mediante el asistente virtual

Según la Figura 5.6b, la capacidad de corregir errores de extracción utilizando peticiones en lenguaje natural fue valorada como una tarea con dificultad en algunos casos mediana, y en otros casos

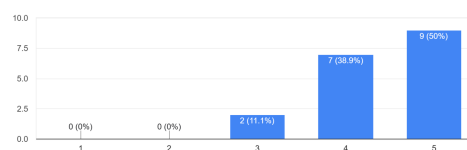
muy sencilla.

Configuración y análisis de una página web
18 respuestas



a) Configuración y análisis de una página web

Refinamiento de resultados con el asistente virtual
18 respuestas



b) Refinamiento mediante el asistente virtual

Figura 5.6: Configuración de la extracción y refinamiento mediante el asistente.

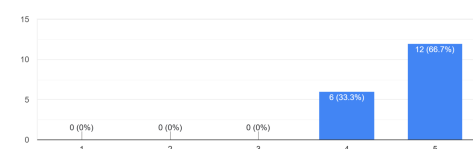
Consulta de los resultados

La visualización de los datos extraídos en la interfaz resultó comprensible y accesible, tal y como reflejan las elevadas calificaciones ilustradas en la Figura 5.7a.

Eliminación de los resultados

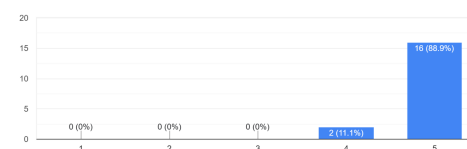
La funcionalidad de borrado de una Website en el sistema fue valorada como muy sencilla entre los participantes, pudiendo observarse en la Figura 5.7b.

Consulta de los resultados
18 respuestas



a) Consulta de los resultados

Eliminación de los resultados seleccionados
18 respuestas



b) Eliminación de los resultados

Figura 5.7: Consulta y eliminación de los resultados obtenidos.

5.1.5. Experiencia y comentarios posteriores a la prueba

Tras finalizar el circuito de tareas operativas, se solicitó una valoración global en una escala de 1 a 5 estrellas para medir aspectos cualitativos y perceptivos de la herramienta.

Facilidad de uso

El sistema fue considerado muy accesible, logrando una sobresaliente nota media de 4,67 sobre 5 en la métrica de facilidad de uso general (Figura 5.8a).

Claridad de la navegación

La Figura 5.8b muestra que la mayoría de los encuestados consideraron que la aplicación tiene una navegación clara o muy clara, siendo su nota media 4,45.

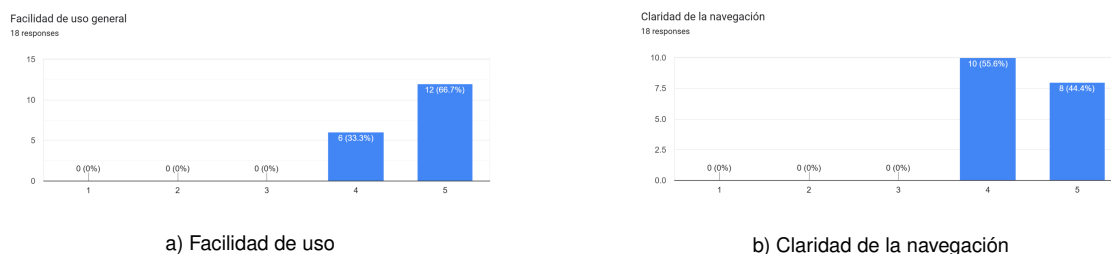


Figura 5.8: Facilidad de uso y claridad de la navegación.

Satisfacción general

El grado de satisfacción global con Kosmos resultó ser elevado, promediando una nota de 4,67 sobre 5 y destacando una gran aceptación del recorrido guiado (Figura 5.9b).

Utilidad de los resultados generados

Los participantes consideraron en un 77 % que los resultados generados les fueron de gran utilidad, mientras que un 22.2 % consideró no estar tan satisfecho con los mismos (ver Figura 5.9b).

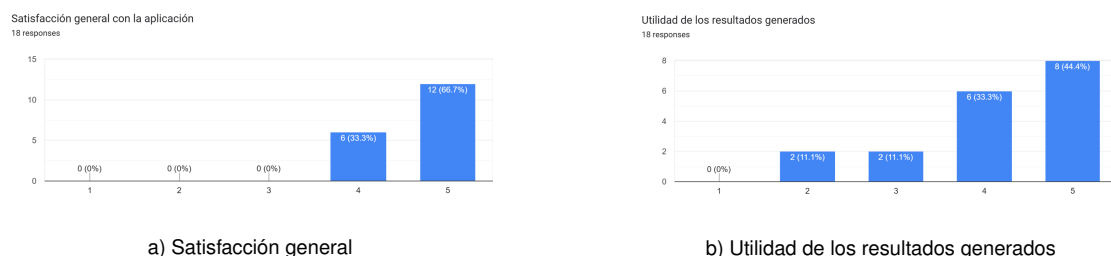


Figura 5.9: Satisfacción general y utilidad de los resultados.

Estética de la aplicación

El apartado visual y de diseño de la plataforma obtuvo una recepción muy positiva con una nota media de 4,72, siendo catalogado como limpio y bonito (Figura 5.10a).

Claridad de la presentación de los resultados

Todos los encuestados afirmaron comprender tanto la configuración como la forma de mostrar los resultados en la aplicación (Figura 5.10b).

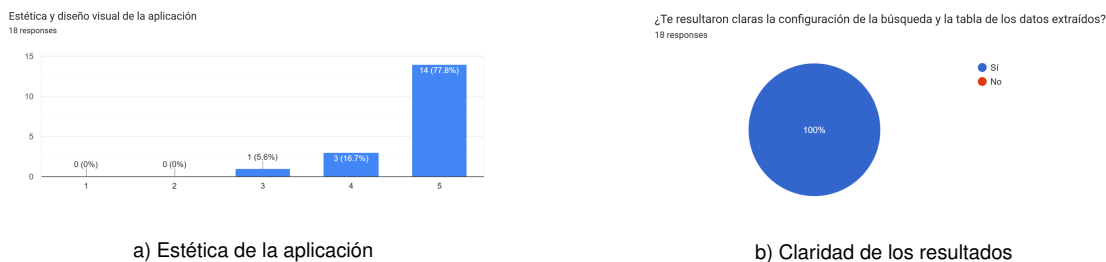


Figura 5.10: Estética de la aplicación y claridad de la presentación.

Mejoras propuestas

Las principales aportaciones de los encuestados fueron la inclusión de filtros de búsqueda más avanzados (precio <10 €) y la reducción de los tiempos de espera en las ejecuciones.

Ideas para funcionalidades adicionales

Los participantes propusieron varias ideas para futuras mejoras. Entre ellas están la implementación de algún tipo de acceso por sesión y credenciales a portales privados para monitorizar valores como las notas de asignaturas, y la exportación de los resultados por descarga en formato PDF, CSV o JSON.

5.2. Análisis de los resultados

En conclusión, los resultados del estudio confirman por la satisfacción de los usuarios, la facilidad de uso y la claridad de la navegación, que la gran mayoría de los participantes prefieren conversar con un chatbot y acceder a datos de manera automática para extraer datos a gran escala. Además de estar satisfechos con el diseño y estética de la aplicación. Con este estudio se muestra la satisfacción de una muestra de público que podrá querer utilizar una herramienta enfocada en la obtención de datos mediante el scraping web para todos los públicos.

CONCLUSIONES

6.1. Conclusiones

Hoy en día trabajamos con información automatizada de internet recopilada a través de *web scraping*, una tecnología que es utilizada en diversos campos. Cuando las personas comparan precios o necesitan información para investigar, surge el desafío de acceder rápidamente a la información. El propósito de la aplicación desarrollada es conseguir facilitar la obtención de la información de forma automatizada a todos sin ningún conocimiento de programación. Por lo tanto, Kosmos en este trabajo ha sido desarrollada para que, utilizando páginas web, organice la información que los usuarios desean.

Para implementar la plataforma, se han integrado modelos de inteligencia artificial conversacional, con el fin de evitar selectores fijos para la identificación del código de las páginas web. El sistema proporciona un asistente virtual que se comunica a través del lenguaje natural permitiendo a los usuarios ajustar la configuración de su búsqueda con una prueba a una URL de ejemplo, y así recibir los resultados y en caso de no estar satisfecho modificar la propuesta con comentarios correctivos.

Del estudio de usuarios realizado, se ha observado que este sistema reduce la dificultad de realizar web scraping. Los datos recopilados verifican que la herramienta funciona correctamente, lo que llevó a una alta calificación en características como claridad de navegación, utilidad de resultados y satisfacción general. Esto demuestra que la aplicación permite a los más inexpertos recopilar una gran cantidad de datos por su cuenta con esta herramienta. En cuanto al rendimiento de la aplicación, este estuvo condicionado por el tiempo de espera de la extracción y el análisis de páginas web más grandes. Además, el sistema no está configurado para la concurrencia, esto se debe a que es un proyecto educativo, y prioriza el no generar problemas de red a los servidores de las páginas web accedidas. Alternativamente, la herramienta aún tiene margen de mejora, algunos usuarios del estudio han ofrecido ideas como incluir filtros de búsqueda avanzados, inicio de sesión con contraseña a portales privados y resultados para ser descargados directamente en formatos como CSV o JSON.

Por último, pero no menos importante, el código fuente de Kosmos y la documentación técnica están accesibles en el repositorio del proyecto. Se ha decidido utilizar GitHub como plataforma para el

repositorio debido a la experiencia con ella a lo largo de la carrera.

Repositorio público del proyecto:

https://github.com/DavidDelgadoSantiago/Kosmos_web_scraping

6.2. Trabajo Futuro

Se han presentado varios temas que podrían ser dignos de desarrollo futuro, en esta sección se van a discutir las principales propuestas. Estas mejoras avanzarían a Kosmos de un prototipo académico a una herramienta escalable a futuro.

En primer lugar, para el almacenamiento de información se encontró que SQLite era la mejor opción para este proyecto ya que es ligero y fácil de configurar. Pero, durante un escenario donde la aplicación necesitara ser escalable para grandes cantidades de datos o miles de usuarios concurrentes, la migración a un motor de base de datos más grande (como MySQL o PostgreSQL) sería una mejor opción.

En segundo lugar, creo que una de las áreas donde hay una gran posible mejora es en la concurrencia y los tiempos de espera. El sistema actual emplea una organización mediante una cola de tareas con extracciones que se procesan una tras otra. De esta manera, si hay dos o más usuarios al mismo tiempo, la actividad de extracción del segundo usuario necesita ser procesada después del primero. Para resolver este problema, una sugerencia sería expandir el número de trabajadores y también la rotación de User-Agents dinámicos. Una mayor concurrencia también permitiría interactuar con el usuario más rápido sin afectar al servidor de destino del usuario, ya que el flujo de tráfico a cada web seguiría siendo bajo y respetuoso debido a que cada usuario estaría en su mayor parte accediendo a páginas web distintas.

En cuanto a la inteligencia artificial, aunque los modelos generales tipo Gemini tienen potencial, puede ser más interesante entrenar un modelo propio de inteligencia artificial para web scraping. Este permitiría aplicar funciones propuestas por los usuarios en nuestro estudio, como por ejemplo, filtros inteligentes, además de una respuesta más rápida. Estos filtros serían reglas que filtran los productos que cumplan condiciones como por ejemplo un precio superior a 10 €. Por otro lado estaría la opción de implementar algún tipo de acceso por sesión y credenciales a portales privados. Esta funcionalidad haría posible monitorizar páginas que requieren inicio de sesión, como portales de calificaciones universitarias o áreas de clientes privados. Por último, está la mejora al respecto de la exportación de datos, permitiendo descargar los datos en formatos PDF, CSV o JSON.

En resumen, Kosmos proporciona una idea clara para un sistema de extracción de datos accesible mediante web scraping, además de ser un software cómodo y fácil de usar.

BIBLIOGRAFÍA

- [1] T. Seymour, D. Frantsvog, S. Kumar, *et al.*, “History of search engines,” *International Journal of Management & Information Systems (IJMIS)*, vol. 15, no. 4, pp. 47–58, 2011.
- [2] G. Barba, M. Lezzi, M. Lazoi, and A. Corallo, “Combined use of web scraping and ai-based models for business applications: research evolution and future trends,” *Management Review Quarterly*, pp. 1–49, 2025.
- [3] R. Gunawan, A. Rahmatulloh, I. Darmawan, and F. Firdaus, “Comparison of web scraping techniques: regular expression, html dom and xpath,” in *2018 International Conference on Industrial Enterprise and System Engineering (ICoIESE 2018)*, pp. 283–287, Atlantis Press, 2019.
- [4] M. Najork, “Web crawler architecture.,” 2009.
- [5] E. Uzun, “A regular expression generator based on css selectors for efficient extraction from html pages,” *Turkish Journal of Electrical Engineering and Computer Sciences*, vol. 28, no. 6, pp. 3389–3401, 2020.
- [6] S. Ortona, G. Orsi, T. Furche, and M. Buoncristiano, “Joint repairs for web wrappers,” in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 1146–1157, IEEE, 2016.
- [7] A. Cuevas, F. Miedema, K. Soska, N. Christin, and R. van Wegberg, “Measurement by proxy: On the accuracy of online marketplace measurements,” in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2153–2170, 2022.
- [8] X. Li, B. A. Azad, A. Rahmati, and N. Nikiforakis, “Good bot, bad bot: Characterizing automated browsing activity,” in *2021 IEEE symposium on security and privacy (sp)*, pp. 1589–1605, IEEE, 2021.
- [9] A. Mesbah, A. Van Deursen, and S. Lenselink, “Crawling ajax-based web applications through dynamic analysis of user interface state changes,” *ACM Transactions on the Web (TWB)*, vol. 6, no. 1, pp. 1–30, 2012.
- [10] A. Halilaj and V. Norlin, “A comparative evaluation of end-to-end testing frameworks for cross-browser web applications: Cypress vs playwright,” 2025.
- [11] D. P. Nguyen and S. Maag, “Codeless web testing using selenium and machine learning,” in *ICSOF 2020: 15th International Conference on Software Technologies*, pp. 51–60, ScitePress, 2020.
- [12] S. Sivakumar, “Performance optimization of large language models (llms) in web applications,” *International Journal of Advanced Scientific Research*, vol. 8, pp. 1077–1096, 2024.
- [13] J. Dagdelen, A. Dunn, S. Lee, N. Walker, A. S. Rosen, G. Ceder, K. A. Persson, and A. Jain, “Structured information extraction from scientific text with large language models,” *Nature communications*, vol. 15, no. 1, p. 1418, 2024.

- [14] P. A. V. Bonvin, “La compatibilidad del web scraping con los principios de la protección de datos personales,” *IUS ET SCIENTIA*, vol. 11, no. 2, pp. 74–103, 2025.
- [15] M. A. Brown, A. Gruen, G. Maldoff, S. Messing, Z. Sanderson, and M. Zimmer, “Web scraping for research: Legal, ethical, institutional, and scientific considerations,” *Big Data & Society*, vol. 12, no. 4, p. 20539517251381686, 2025.

APÉNDICES

APENDICE

¿Aceptas participar de forma libre, voluntaria y anónima?

18 responses

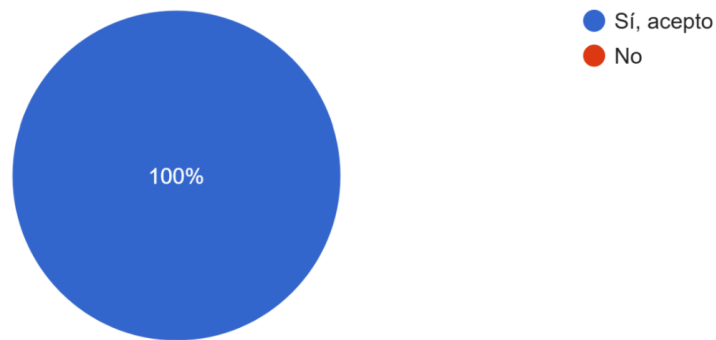


Figura A.1: Consentimiento

¿Cuál es tu rango de edad?

18 responses

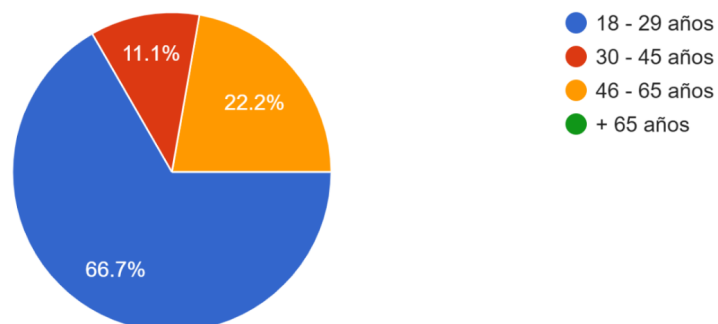


Figura A.2: Distribución por edad

Género

18 responses

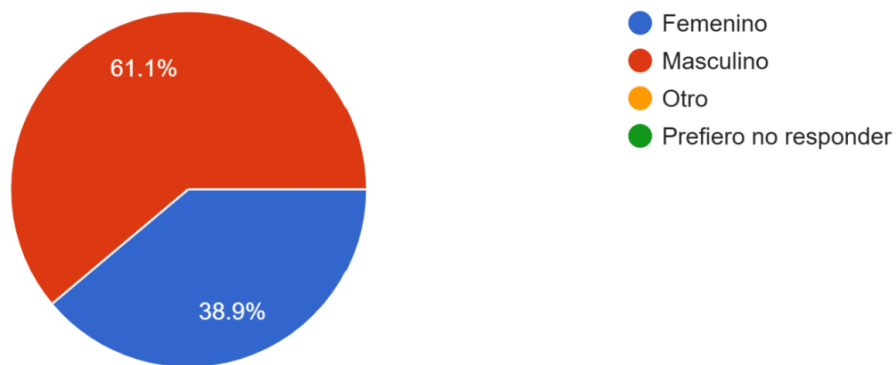


Figura A.3: Género

¿Cuál es tu experiencia buscando productos o información en páginas web?

18 responses

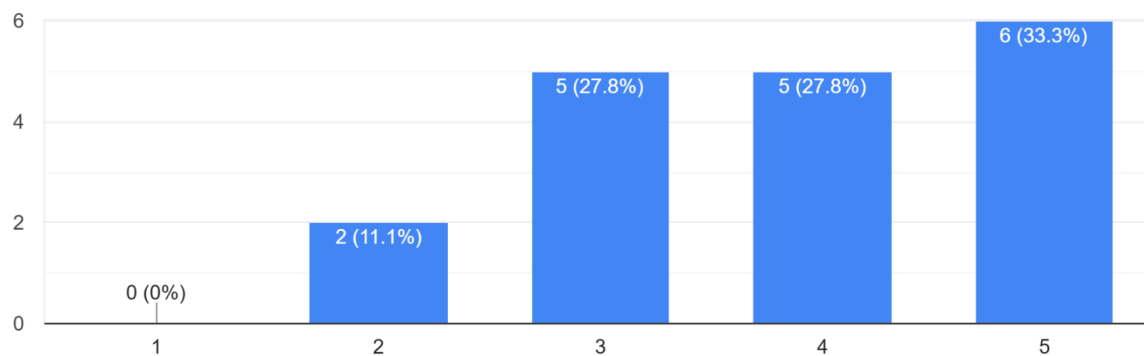


Figura A.4: Experiencia en búsqueda web

¿Cuáles serían tus motivos principales para usar una aplicación que extrae datos web automáticamente? (Selecciona hasta 3)

18 responses

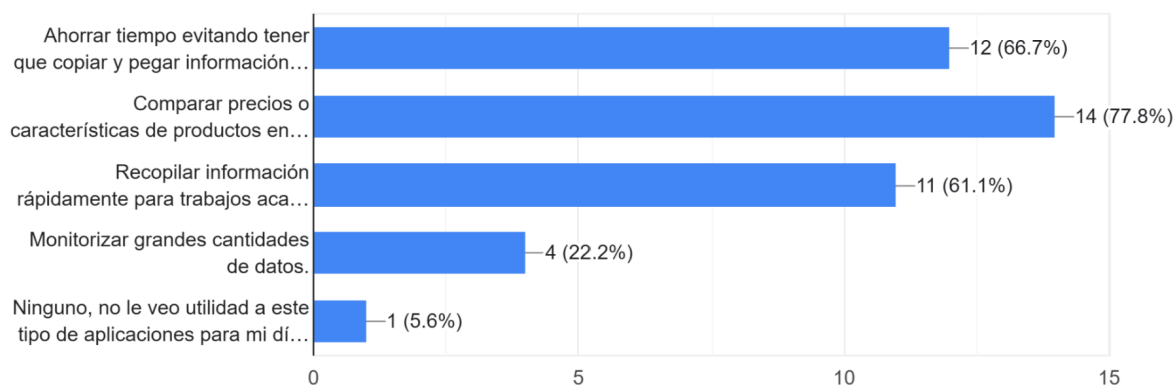


Figura A.5: Motivaciones principales

A la hora de querer sacar información de una página web con una aplicación, por ejemplo, el nombre y precio de muchos productos, ¿qué método prefieres utilizar?

18 responses

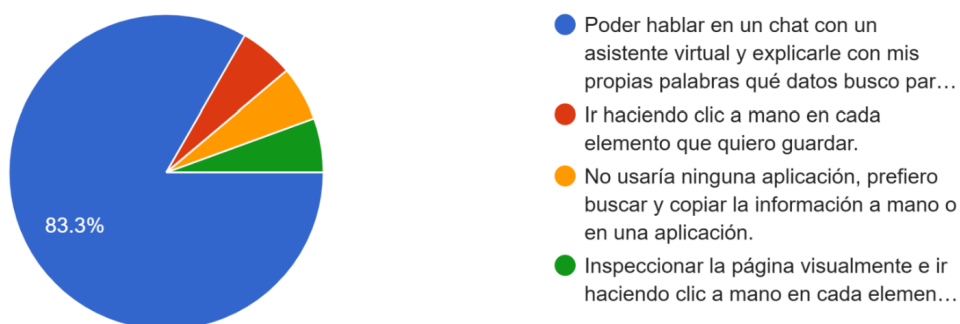


Figura A.6: Preferencia de interacción

Sabiendo que esta aplicación puede navegar por páginas web, ¿qué tipo de información te resultaría más útil extraer de forma automatizada? (Selecciona hasta 3)

18 respuestas

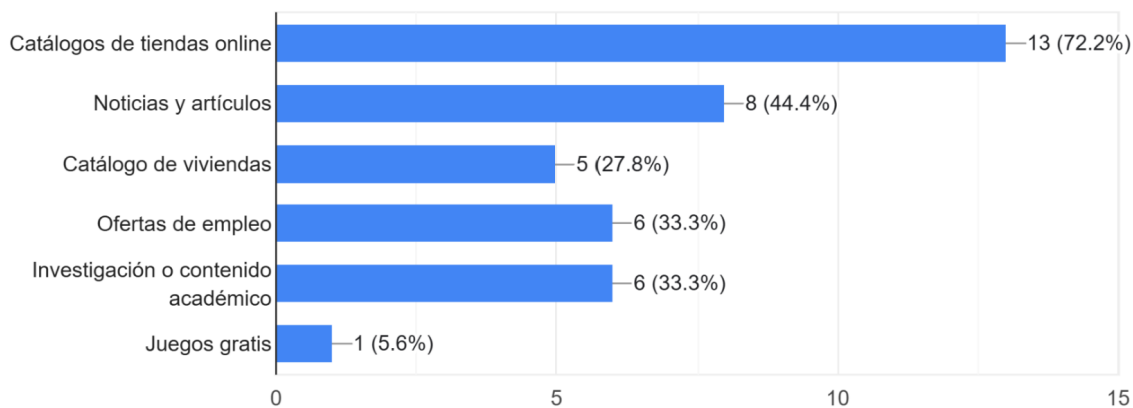


Figura A.7: Tipos de datos a extraer

Configuración y análisis de una página web

18 respuestas

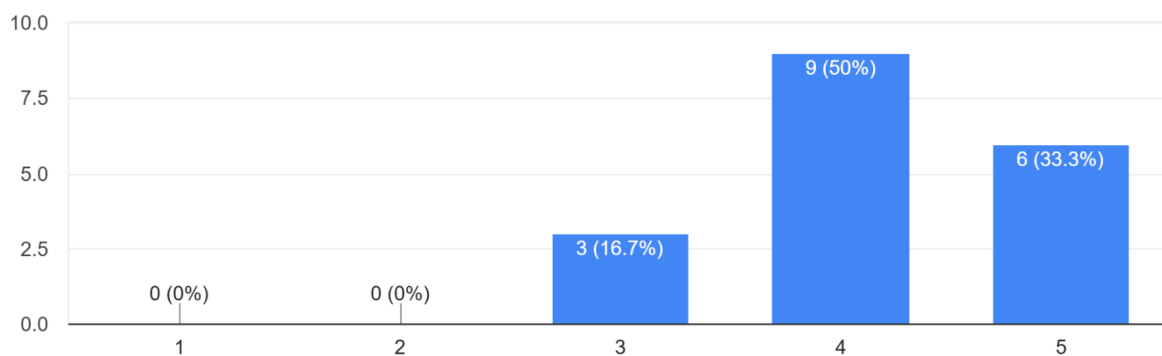


Figura A.8: Configuración y análisis

Refinamiento de resultados con el asistente virtual

18 responses

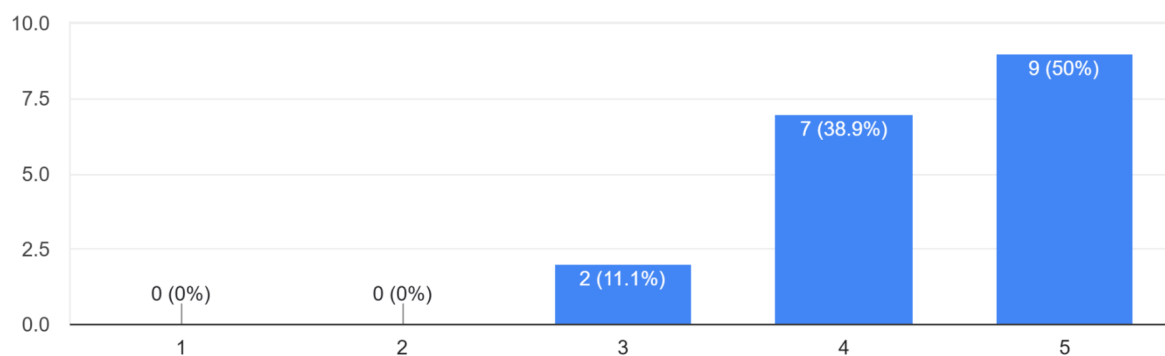


Figura A.9: Refinamiento mediante asistente

Consulta de los resultados

18 responses

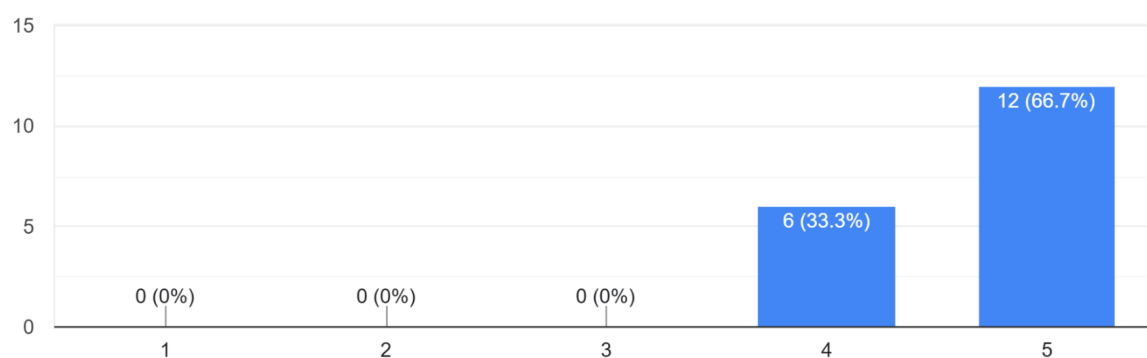


Figura A.10: Consulta de resultados

Eliminación de los resultados seleccionados

18 responses

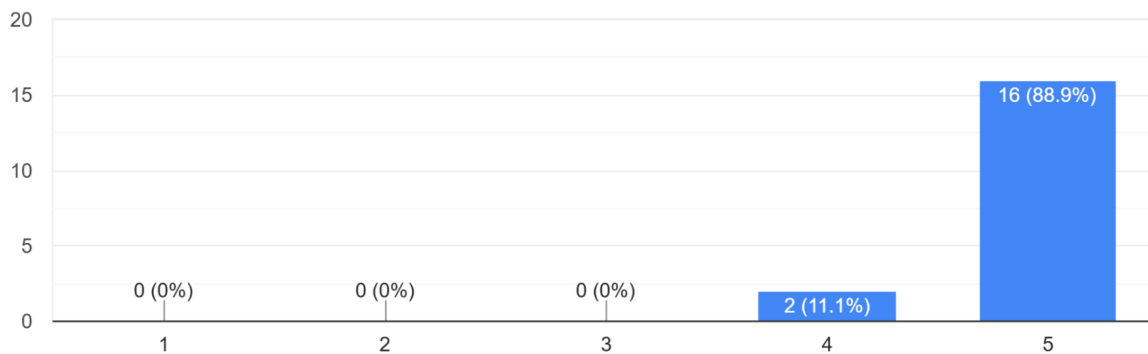


Figura A.11: Eliminación de resultados

Facilidad de uso general

18 responses

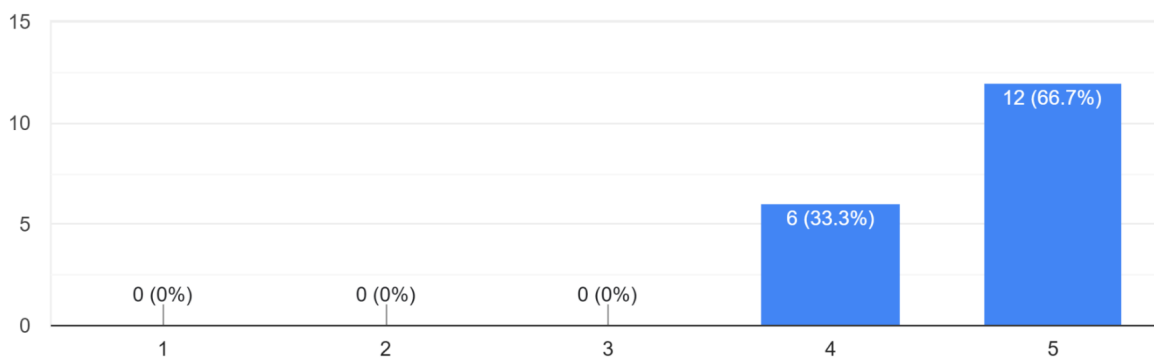


Figura A.12: Facilidad de uso

Claridad de la navegación

18 responses

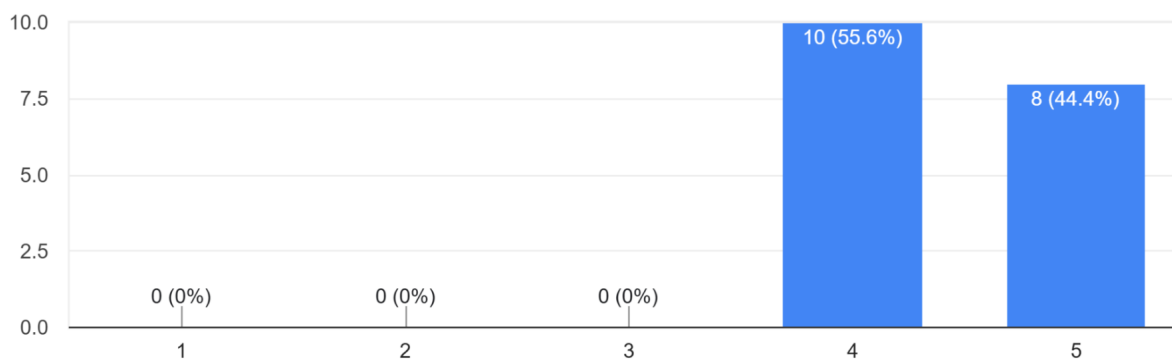


Figura A.13: Claridad de la navegación

Satisfacción general con la aplicación

18 responses

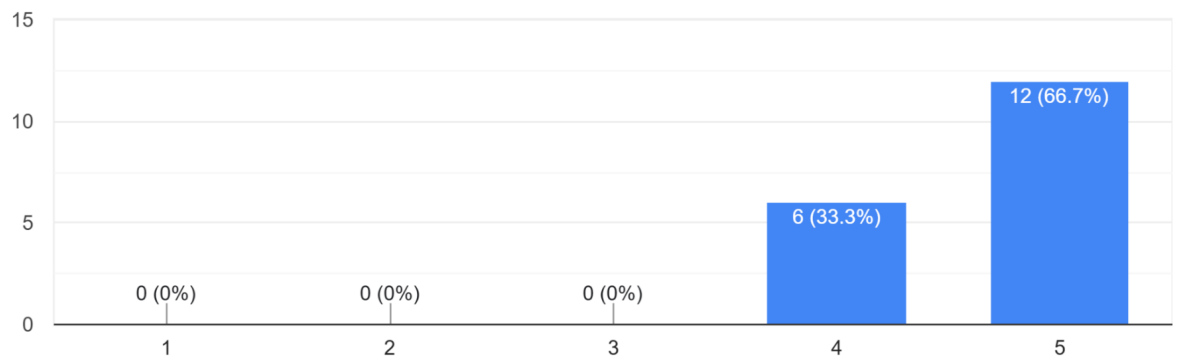


Figura A.14: Satisfacción general

Utilidad de los resultados generados

18 responses

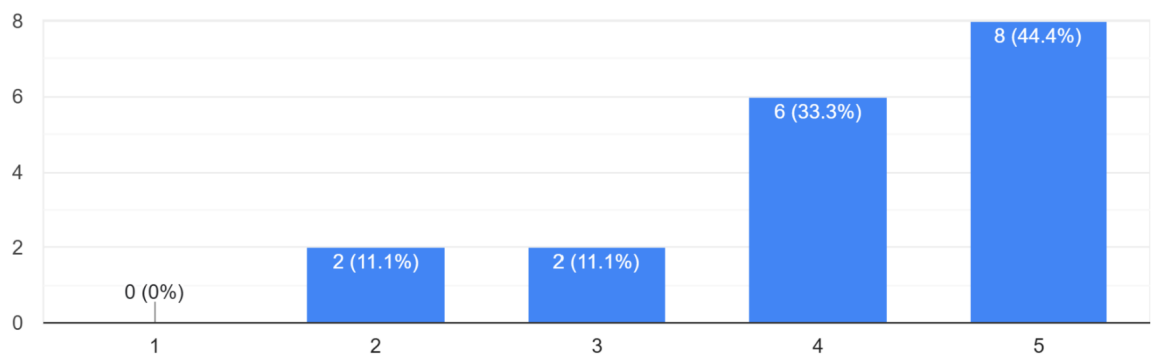


Figura A.15: Utilidad de los resultados

Estética y diseño visual de la aplicación

18 responses

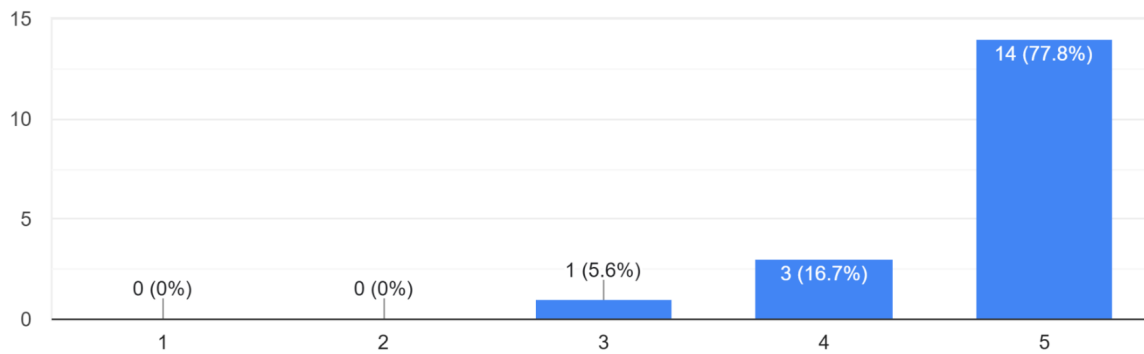


Figura A.16: Estética de la aplicación

¿Te resultaron claras la configuración de la búsqueda y la tabla de los datos extraídos?

18 responses

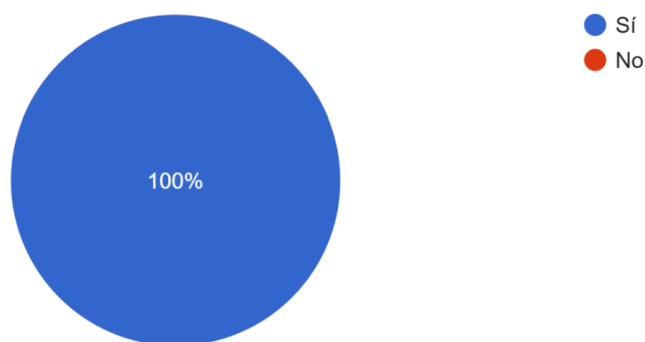


Figura A.17: Claridad de presentación



Universidad Autónoma
de Madrid