UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

APLICACIÓN DE IA GENERATIVA PARA EL MANTENIMIENTO DE PRUEBAS AUTOMÁTICAS

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Ascó Cholbi, Iván

Tutor/a: Canós Cerdá, José Hilario

Cotutor/a: Llavador Campos, Manuel

CURSO ACADÉMICO: 2024/2025

Resum

A mesura que evoluciona el desenvolupament de *software*, les proves automàtiques esdevenen un component essencial per a garantir la qualitat i la fiabilitat dels sistemes. No obstant això, amb el pas del temps, apareix la necessitat de mantindre aquestes proves de manera contínua. Quan el nombre de proves és reduït, aquesta tasca no representa una dificultat significativa. Tanmateix, a mesura que el conjunt de proves augmenta, el procés de revisió i actualització individual de cadascuna d'elles es converteix en un repte considerable.

L'objectiu d'aquest treball és proposar una solució a aquesta problemàtica mitjançant l'ús d'*Intel·ligència Artificial Generativa* (IAG). L'enfocament plantejat pretén que la IAG siga capaç d'analitzar un cas d'ús i, a partir de les dades d'entrada, generar automàticament un conjunt de proves que proporcione el major grau de cobertura possible.

Amb aquesta finalitat, es desenvoluparà una eina de generació de definicions de proves mitjancant la IA complementaria a l'eina de codi obert **Wakamiti**. Aquesta eina, basada en el llenguatge Gherkin, implementa el paradigma d'especificació per comportament (*Behavior Driven Development*, BDD per les seues sigles en anglès). Aquest enfocament permet la definició de proves en llenguatge natural, la qual cosa no sols facilita la seua creació, sinó que també millora la seua comprensió i accessibilitat per a tots els perfils implicats en el procés de desenvolupament de programari.

Paraules clau: IA, Inteligència Artificial, test, automatització, LLM, tester

Resumen

A medida que evoluciona el desarrollo de software, las pruebas automáticas se convierten en un componente esencial para garantizar la calidad y fiabilidad de los sistemas. Sin embargo, con el paso del tiempo, surge la necesidad de mantener dichas pruebas de manera continua. Cuando el número de pruebas es reducido, esta tarea no representa una dificultad significativa. No obstante, a medida que el conjunto de pruebas crece, el proceso de revisión y actualización individual de cada una de ellas se transforma en un desafío considerable.

El propósito de este trabajo es proponer una solución a esta problemática mediante el uso de *Inteligencia Artificial Generativa* (IAG). El enfoque planteado busca que la IAG sea capaz de analizar un caso de uso y, a partir de los datos de entrada, generar automáticamente un conjunto de pruebas que proporcione el mayor grado de cobertura posible.

Con este fin, se desarrollará una herramienta de generación de definiciones de pruebas mediante el uso de IA complementaria a la herramienta de código abierto **Wakamiti**. Esta herramienta, basada en el lenguaje Gherkin, implementa el paradigma de especificación por comportamiento (*Behavior Driven Development*, BDD por sus siglas en inglés). Dicho enfoque permite la definición de pruebas en lenguaje natural, lo cual no solo facilita su creación, sino que también mejora su comprensión y accesibilidad para todos los perfiles involucrados en el proceso de desarrollo de software.

Palabras clave: IA, Inteligencia artificial, test, automatización, LLM, tester

Abstract

As software development evolves, automated testing becomes an essential component to ensure the quality and reliability of systems. However, over time, the need to continuously maintain these tests arises. When the number of tests is small, this task does not pose a significant challenge. Nevertheless, as the test suite grows, the process of individually reviewing and updating each test turns into a considerable challenge.

The purpose of this work is to propose a solution to this problem through the use of *Generative Artificial Intelligence* (GAI). The proposed approach aims for GAI to be able to analyze a use case and, based on the input data, automatically generate a set of tests that provides the highest possible degree of coverage.

To this end, a test definition generation tool will be developed using AI, complementary to the open-source tool **Wakamiti**. This tool, based on the Gherkin language, implements the behavior-driven development paradigm (*Behavior Driven Development*, BDD). This approach allows the definition of tests in natural language, which not only facilitates their creation but also improves their comprehension and accessibility for all stakeholders involved in the software development

Key words: AI, Artificial intelligence, test, automation, LLM, tester

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

Este trabajo ha sido posible gracias al apoyo de innumerables personas, pero en primer lugar tengo que mencionar a mis padres. Sin vosotros, esto no habría sido posible, sin vuestro ánimo y apoyo en los momentos más difíciles. Sois la base de todo lo que hago y nunca podré agradeceros lo suficiente por todo lo que habéis hecho por mí.

También quiero agradecer a mis tutores, José Hilario, Auxi Carlos, y Manuel Llavador sin ellos este trabajo no habría salido adelante y me han servido de mucha ayuda con sus comentarios y aportes.

Glosario

***Swagger*:** Conjunto de herramientas de software de código abierto para diseñar, construir, documentar y consumir APIs. Actualmente se conoce como OpenAPI Specification (OAS), un estándar que describe de forma estructurada los endpoints, parámetros y respuestas de una API.

***API*:** Conjunto de reglas y especificaciones que permiten la comunicación entre diferentes aplicaciones o componentes de software, facilitando la integración y el intercambio de datos.

***Framework*:** Entorno de trabajo compuesto por librerías, componentes y reglas que proporcionan una estructura base para desarrollar aplicaciones de manera más eficiente y estandarizada.

***Pipeline*:** Flujo automatizado de procesos en desarrollo de software que incluye etapas como compilación, pruebas, integración y despliegue. Se emplea en metodologías de CI/CD para asegurar entregas rápidas y de calidad.

***Frontend*:** Parte visible de una aplicación con la que interactúa directamente el usuario. Incluye la interfaz gráfica y la lógica de presentación.

***Backend*:** Parte no visible de una aplicación que gestiona la lógica de negocio, el acceso a bases de datos y la comunicación con servicios externos.

***Testing*:** Conjunto de actividades destinadas a verificar y validar que un sistema cumple con los requisitos especificados y funciona de manera correcta.

***Tester*:** Profesional encargado de diseñar, ejecutar y mantener pruebas de software, con el objetivo de garantizar la calidad y fiabilidad del producto.

***Smoke test*:** Conjunto reducido de pruebas básicas destinadas a comprobar rápidamente si una nueva versión de software es estable y funcional antes de aplicar pruebas más exhaustivas.

***DevOps*:** Conjunto de prácticas y herramientas que integran el desarrollo de software y las operaciones de sistemas para mejorar la colaboración, la automatización y la entrega continua de software.

***Prompt*:** Instrucción o entrada textual que se proporciona a un modelo de lenguaje o sistema de IA para obtener una respuesta o acción específica.

***Step definition*:** En entornos BDD, fragmento de código que implementa los pasos escritos en lenguaje natural, conectando los escenarios de prueba con su ejecución real.

***Endpoint*:** Dirección específica de una API a la que se puede acceder para realizar una operación determinada (por ejemplo, consultar datos o enviar información).

***Fixture*:** Conjunto de datos o configuraciones iniciales necesarias para ejecutar pruebas de software de forma controlada y reproducible.

***Token*:** Cadena de caracteres que actúa como credencial de seguridad en procesos de autenticación y autorización en sistemas informáticos.

***Plugin*:** Componente de software que añade funcionalidades específicas a una aplicación principal, extendiendo sus capacidades sin modificar su núcleo.

***Script*:** Archivo de texto que contiene una secuencia de instrucciones escritas en un lenguaje interpretado. Se utiliza para automatizar tareas, ejecutar procesos o definir pruebas sin necesidad de compilación previa.

***Stakeholders*:** Personas, grupos u organizaciones que tienen interés o se ven afectados por un proyecto, producto o sistema de software.

CAPÍTULO 1
Introducción

1.1 Motivación

Hoy en día, el proceso de pruebas de aplicaciones software adquiere una gran relevancia a lo largo de todo el ciclo de vida del desarrollo, desde la validación de requisitos hasta las pruebas posteriores al despliegue. En cada fase se requieren distintos tipos de verificación, como las pruebas unitarias durante la construcción de la aplicación o las pruebas de integración en fases posteriores.

Con el incremento en la complejidad y número de desarrollos, la carga de trabajo del *tester* funcional aumenta proporcionalmente. A medida que se incorporan nuevas pantallas y funcionalidades, el profesional debe verificar cada una de ellas de manera aislada y en conjunto, lo que implica realizar pruebas unitarias e integradas. Además, cada vez que se añaden funcionalidades adicionales, se hace necesario repetir pruebas previamente ejecutadas para garantizar que no se introduzcan errores en componentes ya validados, lo que corresponde a las denominadas pruebas de regresión.

En el ámbito de la automatización sucede una situación análoga. El profesional encargado de automatizar pruebas en el *frontend* o en el *backend* debe crear casos de prueba asociados a los distintos escenarios de uso, programando interacciones con las pantallas de la aplicación o validando los distintos *endpoints*. A medida que el desarrollo progresa, la ejecución de las baterías de pruebas de regresión se complica, llegando a requerir minutos, horas o incluso días. Estos procesos son susceptibles de interrupciones debido a fallos de ejecución que obligan a realizar correcciones y reiniciar el ciclo, lo que repercute directamente en la productividad.

Tanto el *backend*, por su naturaleza cambiante, como el *frontend*, en proyectos sujetos a entregas frecuentes, demandan una ejecución continua de pruebas. Una vez obtenidos los resultados, el *tester* debe analizar los fallos y adaptar los casos de prueba afectados. Este proceso supone dedicar una parte considerable del tiempo a revisar y mantener pruebas ya existentes, en lugar de destinarlo a validar los nuevos desarrollos.

Esta situación genera un problema de eficiencia, pues se invierte un esfuerzo significativo en la validación de errores previamente corregidos, reduciendo la capacidad de verificación de nuevas funcionalidades. En consecuencia, se hace necesario explorar soluciones que permitan optimizar al máximo el tiempo destinado al aseguramiento de la calidad del software.

La irrupción de la Inteligencia Artificial Generativa (IAG) en los últimos años abre nuevas oportunidades para abordar este reto. Gracias a su rápida evolución, esta tecnología ofrece la posibilidad de asistir en la creación y mantenimiento de pruebas automáticas, reduciendo la carga manual del proceso. Herramientas basadas en modelos

generativos, como ChatGPT, DeepSeek o Gemini, permiten a partir de descripciones detalladas del formato de las pruebas y del funcionamiento de las herramientas empleadas, generar casos básicos que proporcionan una cobertura inicial amplia.

El objetivo de este proyecto se centra en aplicar técnicas de inteligencia artificial para desarrollar un sistema complementario a la herramienta de pruebas automáticas **Wakamiti**, con el propósito de generar de manera eficiente definiciones de pruebas asociadas al software en desarrollo.

1.2 Objetivos

El objetivo general de este proyecto consiste en aplicar técnicas de Inteligencia Artificial Generativa (IAG) para desarrollar una herramienta complementaria a la herramienta *open source* de pruebas automáticas **Wakamiti**, con el propósito de generar definiciones de pruebas automáticas que proporcionen la mayor cobertura posible, así como mantenerlas de manera eficiente frente a los cambios en la API sobre la que se ejecutan. De este modo, se persigue optimizar el tiempo actualmente invertido en tareas de mantenimiento manual.

Asimismo, en este trabajo se pretende:

- Analizar el estado del arte del *software testing*, tanto en enfoques manuales como automáticos, e identificar sus limitaciones.
- Estudiar herramientas actuales de automatización de pruebas, así como aquellas que integran inteligencia artificial para la generación de casos de prueba.
- Describir la arquitectura y el funcionamiento de Wakamiti como base para la implementación de la herramienta de generación de pruebas.
- Diseñar e implementar un prototipo que emplee la API de ChatGPT para la generación automática de definiciones de escenarios *Gherkin*.
- Evaluar la utilidad de la herramienta mediante ejemplos prácticos.
- Proponer posibles mejoras y líneas de trabajo futuro que permitan ampliar las funcionalidades desarrolladas.

1.3 Metodología seguida

Para la elaboración de este documento se ha procedido de la siguiente manera:

La elaboración de la parte teórica del documento se ha realizado mediante el uso de herramientas de búsqueda de documentos académicos como *Google Scholar* o *IEEE Xplore*. Para la búsqueda en estas plataformas se han utilizado palabras clave como *test, test automation, testing, LLM, machine learning, tester, software development*, etc.

Para el desarrollo de la herramienta se ha hecho uso del entorno de trabajo *VSCode*, el cual facilita el desarrollo de software por su clara interfaz y por la facilidad que proporciona para la integración de extensiones que proporcionan ayuda en el proceso de desarrollo.

Finalmente, para la planificación del desarrollo se ha hecho uso de la plataforma *Azure DevOps* en la cual se han establecido los casos de uso con sus respectivas tareas a realizar.

1.4 Estructura de la memoria

Este documento se estructura de la siguiente manera

- **Capítulo 1 – Introducción:** En primer lugar, se da a conocer el problema a resolver con este proyecto, los objetivos a cumplir y, finalmente, el contexto temporal en el que se encuentra. Por otro lado, también se presentan las metodologías aplicadas, así como la estructura del mismo.

- **Capítulo 2 – Contextualización de las pruebas de software en el desarrollo de aplicaciones software:** En este segundo capítulo se describirán detalladamente los conceptos principales del *testing* (proceso de pruebas de software) aplicado al desarrollo, y se contextualizará dicho proceso durante los últimos años, destacando la relevancia que tiene en todo el ciclo de desarrollo. Asimismo, se analizarán los diferentes tipos de pruebas existentes según el enfoque que se les dé.

  A continuación, se realizará una serie de comparaciones entre el *testing* tradicional y el *testing* automático. En primer lugar, se hablará de las pruebas manuales, valorando tanto sus ventajas como sus desventajas. En segundo lugar, se abordarán las pruebas automáticas de manera análoga. Finalmente, se llevará a cabo una comparación global, evaluando los pros y contras de ambas modalidades.

  En la siguiente sección se tratará la Inteligencia Artificial, profundizando en los LLM, sobre los cuales se explicará su historia reciente para, posteriormente, abordar su funcionamiento en términos generales. Para concluir con los LLM, se presentarán sus limitaciones y se describirán brevemente las posibles aplicaciones de esta tecnología.

  En el último apartado de esta sección, se unirán los conceptos de pruebas automatizadas e Inteligencia Artificial con el fin de analizar los posibles roles que pueda desempeñar la IA en el desarrollo y mantenimiento de dichas pruebas.

- **Capítulo 3 – Descripción de la herramienta Wakamiti:** En este capítulo se hablará sobre BDD (Behaviour Driven Development). En primer lugar, se explicará qué es, para qué se usa y cuáles son los beneficios que aporta. En segundo lugar, se abordará *Gherkin*, el núcleo sintáctico en el que se basa el BDD para la creación de pruebas. Finalmente, se presentará la herramienta sobre la cual se va a trabajar: *Wakamiti*.

  Primero, se hará una introducción a la herramienta, explicando qué es Wakamiti y el tipo de lenguaje que utiliza en sus pruebas (*Gherkin*). A continuación, se describirá su funcionamiento interno, así como su arquitectura y modelo de pruebas.

- **Capítulo 4 – Estado del arte y herramientas existentes:** El objetivo de este capítulo es la evaluación y comparación de diferentes herramientas de *testing* de software.

  En primer lugar, se evaluarán diversas herramientas de *testing* automático que no hacen uso de inteligencia artificial. Se analizarán opciones como Selenium, Postman, Cypress, Dredd, entre otras. Se presentará una breve descripción de cada herramienta junto con un análisis de sus ventajas y desventajas. Seguidamente, se elaborará una tabla comparativa de las herramientas presentadas, exponiendo ciertas características con el fin de visualizar gráficamente sus diferencias. Finalmente, se incluirá una breve conclusión donde se resumirán los resultados de la comparación.

  En la siguiente sección se procederá de manera similar. En este caso, se analizarán diferentes herramientas que incorporan inteligencia artificial durante el proceso de

testing. Algunas de ellas serán Qodo, testRigor, Functionize, Mabl y Katalon Studio. Como en la sección anterior, se describirá cada herramienta y se expondrán sus ventajas y desventajas. Finalmente, se presentará la comparativa en forma de tabla, destacando las características principales con el objetivo de identificar oportunidades de mejora que puedan aplicarse a la herramienta desarrollada.

- **Capítulo 5 – Uso de la API de ChatGPT para la generación de pruebas. Explicación de la herramienta desarrollada:** En este capítulo se describen los requisitos, estructura y diseño de la herramienta desarrollada, que utiliza la API de ChatGPT para la generación automática de definiciones de pruebas.

  Se detallará el desarrollo de la herramienta. En esta parte se abordarán los requisitos funcionales establecidos, la estructura diseñada, el funcionamiento del código implementado y el *prompt* (mensaje o petición que se pasa a un LLM) sobre el cual se basa la generación de pruebas. Además, se explicarán las mejoras introducidas para ampliar el alcance de la herramienta (incluyendo el uso de otros LLM).

  Seguidamente, se tratarán los aspectos de privacidad y seguridad. Este apartado cobra especial relevancia en el ámbito profesional, donde es necesario manejar con cautela los datos de los clientes al trabajar con LLM. Finalmente, se expondrá la estructura y organización de las pruebas, así como la gestión de carpetas del directorio. Para concluir, se revisarán los estándares, convenciones y buenas prácticas aplicadas en el desarrollo.

- **Capítulo 6 – Evaluación y resultados:** Se presentan los resultados de aplicar la herramienta a una API de pruebas, analizando la respuesta generada, verificando si cumple con las nomenclaturas, la cobertura que ofrece la definición generada, su utilidad y comparándola con lo solicitado en el *prompt*.

- **Capítulo 7 – Conclusiones:** Se resumen las principales aportaciones del trabajo, se evalúan los objetivos impuestos al principio del documento, se analizan sus limitaciones y se plantean posibles líneas de mejora o investigación futura.

- **Bibliografía:** Se presenta la bibliografía utilizada en este documento.

- **Anexos:** Se incluye información relacionada con el tratamiento de los Objetivos de Desarrollo Sostenible (ODS).

CAPÍTULO 2

Contextualización de las pruebas de software en el desarrollo de aplicaciones software

Las pruebas de software constituyen una de las actividades fundamentales dentro del ciclo de vida del desarrollo, ya que permite garantizar que los sistemas cumplen con los requisitos funcionales y no funcionales establecidos, y que lo hacen de manera fiable y sostenible en el tiempo. Lejos de ser una fase aislada que se lleva a cabo al final del desarrollo, el proceso de pruebas de software se concibe hoy como un proceso transversal que acompaña a todas las etapas, desde la definición de requisitos hasta el despliegue y mantenimiento del producto.

En este capítulo se contextualiza el papel del proceso de pruebas dentro de la ingeniería del software, abordando su evolución histórica, su relevancia actual y las principales tipologías que se han consolidado a lo largo de los años. Se trata de sentar las bases teóricas necesarias para comprender no solo los enfoques tradicionales, sino también las limitaciones que han motivado la búsqueda de soluciones más avanzadas, entre ellas la automatización y el uso de inteligencia artificial.

El análisis que aquí se presenta servirá, por tanto, para enmarcar el trabajo en un contexto amplio, mostrando cómo las técnicas de verificación y validación han pasado de enfoques manuales a estrategias sistemáticas y automáticas, y cómo estas tendencias se conectan con la propuesta desarrollada en este TFG.

2.1 La importancia del testing en el ciclo de vida del software

Las pruebas de software constituyen una actividad esencial dentro del ciclo de vida del desarrollo de software (*Software Development Life Cycle*, SDLC), ya que asegura que el producto final cumpla con los requisitos esperados y permite detectar defectos en fases tempranas. Corregir errores en estas etapas iniciales resulta considerablemente más económico que hacerlo en producción, como evidencian diversos estudios [1]. De hecho, el *IBM Systems Science Institute* determinó que el coste de corregir un error durante la fase de implementación puede ser hasta seis veces mayor que en la fase de diseño, y hasta cien veces superior si se identifica tras el despliegue del sistema [2].

La literatura académica coincide en que las actividades de verificación y validación consumen una parte sustancial del esfuerzo de desarrollo. Harrold [3] destaca que las pruebas pueden representar entre un 40 % y un 50 % del esfuerzo total de un proyecto. Asimismo, Afzal et al. [4], en su revisión sistemática sobre mejora de procesos de prue-

Figura 2.1: Ciclo de vida del desarrollo de software. Fuente
Fuente:https://bit.ly/3Ia6wTQ

bas, subrayan la magnitud del esfuerzo requerido, lo que refleja el carácter intensivo en recursos del *software testing*.

Para mitigar estos costes, una de las prácticas más reconocidas en la ingeniería del software contemporánea es el *shift-left testing*, que propone desplazar las actividades de verificación y validación hacia etapas tempranas del desarrollo. Este enfoque busca detectar defectos cuanto antes, reduciendo los retrabajos y evitando su propagación a fases posteriores, lo que repercute directamente en la disminución de los costes asociados [1, 2]. Con frecuencia, se complementa con estrategias de pruebas concurrentes y prácticas de mejora continua, que permiten disminuir el gasto global del proyecto sin alargar de forma significativa los tiempos de desarrollo [1].

Además, la introducción de técnicas de automatización en estas fases iniciales contribuye a ampliar la cobertura, reducir la duración de los ciclos de prueba y mejorar la calidad del producto software. La automatización no solo ayuda a disminuir el esfuerzo manual, sino que también aborda los principales desafíos del sector: una encuesta entre profesionales reveló que la gestión del testing y la automatización son consideradas las actividades más complejas dentro del proceso de pruebas [5]. Este hallazgo evidencia que optimizar estas áreas no solo genera beneficios cuantificables en términos de costes y tiempo, sino que también incide en los cuellos de botella que limitan la productividad de la industria.

2.1.1. Beneficios globales del testing

Los principales beneficios de incluir *software testing* desde etapas tempranas del ciclo de vida del desarrollo son:

- **Reducción de los costes totales de desarrollo y mantenimiento.** La corrección de defectos en fases iniciales resulta mucho menos costosa que cuando los errores llegan a producción, donde el impacto económico se multiplica [1, 3].
- **Mejor adherencia a los requisitos y mayor confianza en el sistema.** La verificación temprana permite validar que el software cumple con las especificaciones desde el principio, reduciendo la probabilidad de desviaciones y aumentando la confianza de usuarios y clientes [6].
- **Disminución de riesgos de fallo en producción.** Una estrategia de testing continuo minimiza la probabilidad de errores críticos en entornos reales, evitando tanto pérdidas económicas como impactos negativos en la reputación de la organización [7].
- **Mejora en la mantenibilidad y escalabilidad del software.** Un sistema probado de manera rigurosa es más fácil de modificar, refactorizar y escalar, lo que resulta fundamental en proyectos que evolucionan bajo metodologías ágiles e integración continua [4].

2.1.2. Testing como inversión estratégica

Lejos de ser un coste prescindible, el proceso de validación de software se debe entender como una inversión. Los defectos corregidos en las fases iniciales pueden reducir hasta 100 veces el coste comparado con errores en producción [2]. Además, estándares internacionales como ISO/IEC/IEEE 29119 [8] establecen procesos formales para asegurar la calidad desde la planificación hasta la evaluación de pruebas.

Finalmente, en entornos ágiles o de integración continua (CI/CD), la automatización del testing permite iteraciones rápidas sin comprometer la calidad, lo que es especialmente relevante en contextos donde se usan herramientas automatizadas [9].

2.1.3. Importancia de una estrategia equilibrada

Seleccionar el conjunto adecuado de pruebas depende del tipo de proyecto, los riesgos asociados y los recursos disponibles. Una estrategia de testing bien diseñada combina pruebas unitarias, de integración, funcionales y de regresión, automatizadas en lo posible, y complementadas con pruebas exploratorias y no funcionales [10].

Las pruebas de software pueden clasificarse desde diversas perspectivas, cada una relacionada con distintos objetivos, niveles del sistema o técnicas de ejecución. A continuación, se describen los principales tipos de pruebas utilizados en ingeniería del software.

Según el nivel de abstracción

- **Pruebas unitarias (*Unit Testing*):**

  Constituyen la primera capa de validación en el ciclo de vida del testing de software, enfocadas en verificar de manera aislada cada unidad mínima de código (funciones, métodos, clases), asegurando su comportamiento correcto antes de su integración [10]. Diversos estudios empíricos muestran que, aunque requieren esfuerzo, las pruebas unitarias resultan altamente eficaces en la detección de errores [58]. Frameworks ampliamente adoptados como JUnit (Java), NUnit (C#) o CUnit estructuran, ejecutan y reportan resultados de manera estandarizada [30].

- **Pruebas de integración (*Integration Testing*):**

  Tienen como propósito validar la correcta interacción entre unidades de software previamente verificadas mediante pruebas unitarias. Su objetivo principal es garantizar que los módulos se comunican de manera adecuada, que los datos se transfieren correctamente y que las interfaces cumplen con los contratos establecidos [10].

- **Pruebas de sistema (*System Testing*):**

  Constituyen la validación del software como un todo integrado, verificando que satisface tanto los requisitos funcionales como no funcionales definidos en la fase de especificación. A diferencia de las pruebas unitarias o de integración, su objetivo no es validar módulos aislados, sino evaluar el comportamiento global desde la perspectiva del usuario final [10].

- **Pruebas de aceptación (*Acceptance Testing*):** Verifican que el sistema, como producto integrado, satisface los criterios de aceptación definidos por los representantes de negocio o usuarios finales. Su finalidad es determinar la *aptitud para el uso* antes de la liberación, reduciendo el riesgo de rechazo o de desalineación con las necesidades operativas [10].

Según el propósito

- **Pruebas funcionales:**

  Tienen como objetivo verificar que el software cumple con los requisitos funcionales establecidos en la fase de análisis y diseño. Se centran en evaluar el comportamiento del sistema desde la perspectiva del usuario, sin considerar su implementación interna, lo que las convierte en pruebas de tipo *caja negra* [11].

  Estas pruebas pueden aplicarse en diferentes niveles del ciclo de vida del testing [12] (unidad, integración, sistema y aceptación) y se diseñan a partir de casos de uso, historias de usuario o documentación de requisitos. Entre las técnicas habituales se encuentran la partición de equivalencia, el análisis de valores límite y el modelado de transiciones de estado [13].

- **Pruebas no funcionales:**

  Se orientan a evaluar las características de calidad del software que no dependen de la funcionalidad explícita, sino de atributos como el rendimiento, la seguridad, la usabilidad o la mantenibilidad. Su objetivo es garantizar que el sistema cumple con los requisitos de calidad definidos en normativas, acuerdos de nivel de servicio o expectativas de los usuarios finales [15].

  De acuerdo con el estándar ISO/IEC 25010 [15], estas pruebas se organizan en categorías como rendimiento y escalabilidad, seguridad, usabilidad y accesibilidad, fiabilidad, compatibilidad y mantenibilidad. Por ejemplo, las pruebas de rendimiento incluyen ensayos de carga y estrés; las pruebas de seguridad abarcan la detección de vulnerabilidades y ataques de penetración; y las de usabilidad comprenden la evaluación de la experiencia de usuario y de la accesibilidad según directrices como WCAG (Pautas de Accesibilidad para el Contenido Web, https://www.w3.org/WAI/standards-guidelines/wcag/es).

- **Pruebas de regresión:**

  Tienen como finalidad garantizar que las funcionalidades previamente implementadas y validadas siguen funcionando de manera correcta después de introducir

cambios en el sistema, ya sean nuevas características, correcciones de errores o refactorizaciones. De esta forma, se busca prevenir que modificaciones recientes generen defectos en componentes que anteriormente eran estables [10].

Según el conocimiento del código

- **Pruebas de caja blanca:**

  Consisten en validar la lógica interna del software, considerando el código fuente, los flujos de control y el uso de datos. A diferencia de las pruebas de caja negra, que se centran en la funcionalidad observada desde el exterior, las pruebas de caja blanca requieren acceso al código y conocimiento de su implementación [10].

- **Pruebas de caja negra:**

  Validan el comportamiento del sistema desde la perspectiva externa, comprobando que las salidas cumplen los requisitos para entradas específicas sin considerar la implementación interna [10]. Su diseño se basa en documentación de requisitos, casos de uso o contratos (por ejemplo, especificaciones de API) y pueden aplicarse en distintos niveles: funcional, integración, sistema y aceptación.

- **Pruebas de caja gris:**

  Constituyen un enfoque intermedio entre la caja blanca y la caja negra. En este caso, el tester dispone de un conocimiento parcial de la estructura interna del sistema (como diagramas de flujo de datos, contratos de APIs o documentación técnica), lo que le permite diseñar casos de prueba más enfocados y eficientes que en la caja negra, pero sin necesidad de analizar la totalidad del código fuente [10].

2.2 Testing tradicional vs Testing automático

En esta sección se tratarán dos tipos de pruebas en el marco de pruebas según su grado de automatización las cuales no se han mencionado en el apartado anterior. Estas constan de dos tipos de pruebas: pruebas manuales y pruebas automatizadas.

El *testing* de software ha experimentado una transformación considerable desde sus inicios siendo parte constantemente del ciclo de vida del software. Tradicionalmente, las pruebas se realizaban y siguen realizándose de forma manual en gran parte del ciclo de vida por *testers* humanos, los cuales interactúan directamente con la aplicación validando su correcto funcionamiento según se indique en los respectivos documentos funcionales.

Con la adopción de metodologías ágiles y la necesidad de ciclos de entrega más rápidos, el testing automático ha ganado una gran relevancia en la industria proporcionando grandes mejoras en el tiempo dedicado a probar el software desarrollado. [14].

A continuación se estudiarán ambos tipos de pruebas, tanto manuales como automatizadas. Se explicará en que consisten, se valorarán tanto ventajas como desventajas presentes en ambos tipos de pruebas con el fin de averiguar qué tipo de pruebas es más relevante en los desarrollos actuales.

2.2.1. Testing tradicional (manual)

El ***testing* manual** consiste en la ejecución de casos de prueba sin la intervención de herramientas automatizadas, basándose en la interacción directa del *tester* o (ingeniero de pruebas) con el sistema bajo prueba. El ingeniero o los ingenieros de pruebas, se encargan

personalmente de realizar las pruebas necesarias sobre el sistema a probar en base a los requisitos establecidos en las fases anteriores del ciclo de desarrollo [16].

Este enfoque, es especialmente adecuado para escenarios donde se requiere una validación visual minuciosa, la evaluación de la experiencia de usuario (UI/UX) o la realización de pruebas exploratorias, en las que la creatividad y el criterio del ingeniero de pruebas son fundamentales puesto que, a pesar de que el sistema puede cumplir los requisitos especificados, es posible encontrar errores inesperados. El proceso implica seguir un conjunto de pasos definidos o utilizar el sistema de manera libre para verificar que el software cumple con los requisitos especificados.

Ventajas:

- **Adaptabilidad a condiciones no previstas:** El **testing manual** destaca por su capacidad de adaptación frente a escenarios imprevistos. A diferencia de la automatización, que se limita a ejecutar *scripts* predefinidos, los *testers* pueden reaccionar en tiempo real, explorar caminos alternativos y diseñar nuevas pruebas sobre la marcha. Este rasgo lo hace especialmente útil en pruebas exploratorias y en fases tempranas de desarrollo, cuando los requisitos aún son inestables y la automatización resulta poco eficiente [16].
- **Bajo coste inicial de infraestructura:** Otra característica destacable del *testing* manual es que su adopción no implica una inversión inicial elevada. A diferencia de la automatización, que requiere configurar entornos, adquirir herramientas específicas o desarrollar *scripts*, el enfoque manual puede iniciarse prácticamente con los recursos ya disponibles en el equipo. Esta inmediatez lo convierte en una opción adecuada para validar prototipos o realizar comprobaciones preliminares en fases tempranas, donde aún no compensa destinar esfuerzo a la construcción de una infraestructura de pruebas automatizadas [16].
- **Relevancia en validaciones UI/UX:** Un aspecto en el que el testing manual mantiene una relevancia indiscutible es en la validación de características subjetivas del software. Elementos como la experiencia de usuario, la facilidad de uso o la accesibilidad no pueden evaluarse únicamente con métricas objetivas ni mediante *scripts* automatizados. La percepción humana es capaz de identificar problemas de ergonomía, claridad en la interfaz o frustraciones en la interacción que, aunque no rompan la funcionalidad, afectan de manera significativa a la calidad percibida del sistema [16].

  En este sentido, las pruebas manuales aportan un valor añadido que complementa a la automatización, ya que introducen una dimensión cualitativa en la evaluación del software. Esto resulta especialmente importante en aplicaciones orientadas a usuarios finales, donde la satisfacción y la confianza dependen tanto de aspectos funcionales como de la experiencia subjetiva del usuario.

Desventajas:

- **Propenso a errores humanos:** la ejecución manual de pruebas depende de la concentración y precisión del *tester*, lo que la hace susceptible a fallos derivados de fatiga, descuidos o interpretaciones subjetivas de los resultados. Este factor reduce la fiabilidad y la consistencia de los ensayos, especialmente cuando deben repetirse de manera extensa o en sistemas complejos [16].
- **Poca escalabilidad y dificultad de repetición:**el *testing* manual presenta serias dificultades para crecer al mismo ritmo que la complejidad del software. A medida

que aumentan las funcionalidades, también lo hace el número de casos de prueba, lo que convierte en inviable repetirlos manualmente en cada iteración. En entornos ágiles o con ciclos de entrega continua, esta limitación se traduce en retrasos y cuellos de botella que afectan a la productividad de los equipos de desarrollo y calidad [16].

- **Coste elevado en el largo plazo:** aunque el testing manual no requiere grandes inversiones iniciales, su dependencia de la intervención humana incrementa los costes a medida que el proyecto avanza. La necesidad de repetir las pruebas en cada iteración o tras cada cambio en el sistema multiplica el esfuerzo requerido, lo que lo convierte en un enfoque poco sostenible en proyectos de gran escala o con entregas frecuentes [16].

2.2.2. Testing automático

El ***testing* automatizado** constituye uno de los pilares fundamentales del aseguramiento de la calidad en el desarrollo de software contemporáneo. A diferencia del *testing* manual, donde la verificación depende de la ejecución directa por parte de personas, este enfoque se basa en la utilización de herramientas, marcos de trabajo y *scripts* capaces de ejecutar casos de prueba de forma repetitiva y sin intervención humana. Su finalidad principal es garantizar que los cambios introducidos en el software no comprometen funcionalidades ya existentes y que el sistema cumple de manera continua con los requisitos establecidos [18].

El origen puede situarse en los años noventa, cuando comenzaron a popularizarse las primeras herramientas de soporte para pruebas unitarias y funcionales. Sin embargo, su consolidación se produjo con la llegada de las metodologías ágiles y, especialmente, con la adopción masiva de prácticas de integración y despliegue continuo (*CI/CD*). En este contexto, la automatización se convirtió en un requisito indispensable para poder ejecutar suites de pruebas de forma frecuente, rápida y confiable, acompañando a cada iteración del desarrollo [17].

Actualmente, abarca un amplio espectro de técnicas y niveles: desde pruebas unitarias que verifican componentes individuales del sistema, hasta pruebas de regresión, de integración o de rendimiento que validan el comportamiento global de la aplicación. Además, su integración en *pipelines* de desarrollo permite detectar defectos de manera temprana, reduciendo los costes de corrección y mejorando la calidad del producto final. Como señalan diversos estudios, esta capacidad de proporcionar retroalimentación continua se ha convertido en un factor crítico para el éxito de proyectos de software en entornos competitivos y de alta demanda [18, 16].

El *testing* automatizado no debe entenderse únicamente como una alternativa más eficiente al *testing* manual, sino como un elemento central dentro del ciclo de vida del software moderno. Su capacidad para ofrecer repetibilidad, cobertura amplia y velocidad lo convierte en una práctica imprescindible, cuya relevancia seguirá aumentando con la evolución de metodologías ágiles y la incorporación de nuevas tecnologías como la inteligencia artificial.

Ventajas:

- **Alta repetibilidad y velocidad:** las pruebas automatizadas pueden ejecutarse tantas veces como sea necesario sin pérdida de precisión ni variación en los resultados, lo que garantiza una alta consistencia en la validación del sistema. Además, su velocidad de ejecución es muy superior a la de las pruebas manuales, ya que suites

completas pueden ejecutarse en minutos o incluso segundos, frente a las horas o días que requerirían de forma manual. Esta capacidad resulta crítica en entornos ágiles y de *CI/CD*, donde se demandan ciclos de retroalimentación rápidos para validar cada integración de código [16].

- **Ideal para pruebas de regresión:** uno de los usos más relevantes del *testing* automatizado es la ejecución de pruebas de regresión, cuyo objetivo es garantizar que nuevas funcionalidades o correcciones no afectan negativamente al comportamiento previamente validado del sistema. Debido a que estas pruebas deben repetirse con frecuencia tras cada cambio en el código, el enfoque manual resulta poco práctico y costoso. En contraste, la automatización permite ejecutar suites completas de regresión de forma rápida y continua, reduciendo el riesgo de introducir defectos y proporcionando confianza en cada nueva entrega del software [19].

- **Mejora de la cobertura y confiabilidad:** el *testing* automatizado permite ejecutar un gran número de casos de prueba en cada iteración, lo que incrementa significativamente la cobertura respecto al enfoque manual. Esta capacidad posibilita validar múltiples combinaciones de entradas, escenarios de uso y configuraciones que serían inviables de comprobar de manera manual. Además, al eliminar la variabilidad asociada a la intervención humana, los resultados obtenidos son más consistentes y confiables, reduciendo la probabilidad de errores no detectados. En consecuencia, la automatización contribuye a mejorar tanto la calidad del software como la confianza del equipo en cada entrega [16].

- **Integración con procesos ágiles y CI/CD:** el *testing* automatizado se adapta de forma natural a las metodologías ágiles y a los entornos de integración y despliegue continuo. En estos contextos, donde los equipos entregan nuevas versiones del software de manera frecuente, resulta imprescindible contar con suites de pruebas que se ejecuten automáticamente tras cada cambio en el código. Esta integración garantiza retroalimentación inmediata, facilita la detección temprana de defectos y permite mantener un ciclo de entrega ágil y confiable. Gracias a ello, la automatización se ha consolidado como un componente esencial en la práctica de pruebas continuas, que constituye uno de los pilares de *DevOps* [17].

Desventajas:

- **Esfuerzo de configuración inicial:** la implantación de un entorno de *testing* automatizado requiere una inversión inicial considerable en términos de tiempo y recursos. Es necesario seleccionar las herramientas adecuadas, configurar entornos de ejecución, diseñar los *scripts* de prueba y garantizar su integración con el ciclo de desarrollo. Aunque este esfuerzo se compensa con los beneficios a medio y largo plazo, puede suponer una barrera para proyectos pequeños o en fases muy tempranas, donde la automatización aún no resulta prioritaria [19].

- **Coste de mantenimiento de los *scripts*:** las pruebas automatizadas requieren actualización constante para reflejar los cambios en el sistema bajo prueba. Cada modificación en la interfaz, la lógica de negocio o las dependencias puede invalidar *scripts* existentes, generando costes de mantenimiento significativos si no se gestionan adecuadamente. En proyectos de gran escala, este esfuerzo puede convertirse en una carga considerable para los equipos de calidad, lo que subraya la importancia de aplicar buenas prácticas de diseño y refactorización de pruebas [20, 19].

- **No sustituye el juicio humano:** aunque ofrece gran eficacia en la validación funcional y técnica del software, presenta limitaciones en escenarios donde la percepción

humana resulta esencial. Aspectos como la usabilidad, la accesibilidad o la experiencia de usuario (UI/UX) requieren interpretación subjetiva, creatividad y criterio, cualidades que las herramientas automatizadas aún no pueden reproducir con la misma fidelidad. Por esta razón, incluso en proyectos altamente automatizados, las pruebas manuales siguen siendo necesarias como complemento en estas áreas [16].

2.2.3. Comparación general

El análisis de ambas aproximaciones permite observar que el *testing* manual ofrece valor en contextos muy específicos, como las pruebas exploratorias o la validación de aspectos subjetivos relacionados con la experiencia de usuario. Sin embargo, sus limitaciones en términos de escalabilidad, repetibilidad y coste lo convierten en un enfoque insuficiente para proyectos de gran envergadura o con entregas frecuentes.

Por el contrario, el *testing* automatizado proporciona ventajas decisivas en los entornos actuales: su capacidad para ejecutar pruebas de manera rápida, repetible y con amplia cobertura lo hace indispensable en prácticas de desarrollo ágil y DevOps. Aunque requiere un esfuerzo inicial de configuración y mantenimiento, su retorno de inversión es elevado, ya que garantiza retroalimentación continua y reduce significativamente los riesgos de regresión. En consecuencia, mientras que el *testing* conserva un papel complementario, el automatizado se ha consolidado como el estándar en la industria moderna para asegurar calidad de forma sostenible.

La siguiente tabla resume las diferencias clave entre el *testing* manual y el *testing* automático, considerando aspectos como la velocidad, la repetibilidad, la cobertura o el coste a largo plazo. Esta comparación no implica que uno de los enfoques sea intrínsecamente superior al otro; de hecho, en la mayoría de los proyectos, ambos se combinan para aprovechar sus ventajas respectivas.

Característica	**Testing Manual**	**Testing Automatizado**
Velocidad de ejecución	✗	✓
Repetibilidad	✗	✓
Validación de UX/Usabilidad	✓	✗
Escalabilidad	✗	✓
Coste inicial reducido	✓	✗
Cobertura amplia	✗	✓
Confiabilidad	✗	✓
Adaptabilidad a imprevistos	✓	✗
Coste a largo plazo bajo	✗	✓
Integración con CI/CD	✗	✓

Tabla 2.1: Comparación de características entre *testing* manual y *testing* automatizado.

En la práctica, la elección entre *testing* manual y automático depende de factores como la fase del ciclo de desarrollo, el presupuesto disponible, la criticidad del sistema y la experiencia del equipo. Una estrategia equilibrada que combine ambos métodos puede maximizar la eficiencia y la calidad del producto final.

2.3 Introducción a la Inteligencia Artificial y LLM

Vista la primera parte de la literatura que se presenta, el *testing*, es momento de abordar el segundo elemento con el que se va a trabajar, la Inteligencia Artificial.

La incorporación de la **Inteligencia Artificial (IA)** en el ámbito del desarrollo de software ha supuesto un punto de inflexión en numerosos procesos, desde la generación automática de código hasta la optimización de arquitecturas. Dentro de este espectro de aplicaciones, el *testing* automatizado ha emergido como uno de los campos donde la IA está mostrando un impacto especialmente relevante. Mientras que las herramientas tradicionales de automatización permiten ejecutar pruebas predefinidas de manera rápida y repetitiva, la IA introduce la posibilidad de *generar*, *adaptar* y *mantener* casos de prueba de forma dinámica, a partir de descripciones en lenguaje natural o documentación técnica [21].

En este contexto, los **Modelos de Lenguaje Extensos** (*Large Language Models*, LLMs) han demostrado ser una tecnología particularmente disruptiva. Gracias a su entrenamiento en volúmenes masivos de datos textuales y a su capacidad para comprender y producir lenguaje natural, los LLMs son capaces de traducir requisitos escritos en lenguaje humano en estructuras formales utilizables por marcos de *testing*. Esta característica abre la puerta a una nueva generación de herramientas que no solo automatizan la ejecución de pruebas, sino que también intervienen en las fases de diseño y mantenimiento de los escenarios de validación [22].

La relevancia de este apartado radica en que la comprensión del papel de la Inteligencia Artificial y los LLMs en el *testing* automatizado resulta esencial para contextualizar la herramienta propuesta en este trabajo. Antes de describir en detalle la solución implementada, es necesario establecer un marco conceptual que permita entender qué son los LLMs, cómo funcionan y cuáles son sus limitaciones. De esta forma, se dispondrá de una base sólida para valorar las oportunidades y retos que plantea su aplicación en el proceso de verificación de software.

2.3.1. ¿Qué es un LLM?

Los **Modelos de Lenguaje Extensos** (*Large Language Models*, LLMs) son sistemas de aprendizaje profundo entrenados con enormes volúmenes de datos textuales, cuyo objetivo es modelar el lenguaje natural y generar texto coherente y contextualizado. Su funcionamiento se basa en la arquitectura *transformer*, presentada por Vaswani et al. en 2017, que introdujo el mecanismo de autoatención como alternativa a las redes recurrentes y convolucionales. Esta innovación permitió manejar dependencias de largo alcance en secuencias de texto y sentó las bases de los actuales modelos a gran escala [23].

La popularización de los LLMs se dio a partir de 2018 con la familia **GPT (Generative Pretrained Transformer)** desarrollada por OpenAI, que marcó un antes y un después en el procesamiento de lenguaje natural. Modelos como GPT-2, GPT-3 y, más recientemente, GPT-4 han demostrado una capacidad sin precedentes para generar código, analizar documentación y mantener interacciones conversacionales de alta calidad. Este avance ha impactado de forma directa en el desarrollo de software, donde estas tecnologías ya se emplean para asistir en la redacción de código, la documentación automática y, de manera cada vez más relevante, en la generación y mantenimiento de pruebas de software [21, 24, 25, 26, 27].

Desde un punto de vista técnico, los LLMs se entrenan inicialmente en grandes corpus textuales mediante un proceso de **preentrenamiento** no supervisado, cuyo objetivo principal es predecir la siguiente palabra en una secuencia. Posteriormente, estos mode-

los pueden adaptarse a tareas concretas a través de técnicas de *fine-tuning* (re-entrenar el modelo con datos más específicos) o aprovechar enfoques más recientes como el *in-context learning*, que permiten guiar sus respuestas utilizando ejemplos incluidos directamente en el *prompt*, sin necesidad de modificar el modelo base. Esta característica resulta especialmente valiosa en el ámbito del *testing*, ya que posibilita condicionar la generación de casos de prueba a partir de requisitos o descripciones específicas, sin tener que entrenar de nuevo el sistema.[28, 26, 29]

En el ámbito del **testing**, los LLMs no solo facilitan la traducción de descripciones en lenguaje natural (requisitos, historias de usuario, documentación de APIs) en *casos de prueba estructurados*, sino que también abren nuevas posibilidades en el **mantenimiento de pruebas**. Ante cambios en los requisitos o en la especificación técnica, un modelo de lenguaje puede sugerir la actualización o re-escritura de los escenarios afectados, reduciendo así uno de los mayores cuellos de botella de las estrategias tradicionales de automatización. De esta forma, los LLMs no solo complementan a las herramientas existentes, sino que amplían su alcance hacia etapas tempranas del ciclo de vida del software y hacia actividades de mantenimiento continuo, reforzando el papel del *testing* como actividad transversal en todo el proceso de desarrollo.[30, 22]

En resumen, los LLMs constituyen una tecnología que combina la capacidad de modelar el lenguaje natural con aplicaciones prácticas de gran valor en el desarrollo de software, especialmente en el ámbito del testing. Sin embargo, para comprender mejor su potencial y sus limitaciones, resulta necesario examinar los fundamentos técnicos que explican cómo son capaces de generar texto coherente y adaptarse a distintos contextos. En el siguiente apartado se aborda el funcionamiento básico de estos modelos, centrándose en la arquitectura *transformer*, el proceso de entrenamiento y los mecanismos que hacen posible su uso en tareas como la generación de pruebas automáticas.

2.3.2. Funcionamiento básico

El funcionamiento de los **Modelos de Lenguaje Extensos** (LLMs) se fundamenta en la arquitectura *transformer*, introducida por Vaswani et al. en 2017. A diferencia de las redes neuronales recurrentes tradicionales, los *transformers* incorporan el mecanismo de **autoatención**, que permite a cada palabra de una secuencia considerar simultáneamente la relación con todas las demás. Esto facilita capturar dependencias a largo plazo en el texto y otorga mayor eficiencia en el entrenamiento paralelo de grandes volúmenes de datos [23].

Durante la fase de **preentrenamiento**, los LLMs son expuestos a corpus textuales de enorme escala con el objetivo de aprender a predecir la siguiente palabra en una secuencia dada. A partir de este proceso no supervisado, el modelo adquiere una representación estadística muy rica del lenguaje. Posteriormente, puede especializarse en tareas concretas mediante *ajuste fino (fine-tuning)* o bien adaptarse a instrucciones específicas usando *aprendizaje en contexto (in-context learning)*, donde el comportamiento del modelo se guía a través de ejemplos incluidos en el propio prompt.[26, 28]

En el momento de la **inferencia**, los LLMs generan texto palabra por palabra, eligiendo en cada paso la opción más probable dentro de su vocabulario. Parámetros como la *temperatura* o el *top-p sampling* permiten modular la creatividad y diversidad de las respuestas, lo que resulta esencial para tareas en las que se busca un equilibrio entre precisión y variedad.[31]

Aplicado al **testing**, este funcionamiento permite que los LLMs no solo comprendan descripciones en lenguaje natural, sino que las transformen en estructuras formales como casos de prueba en lenguaje natural o *scripts* automatizados. La capacidad de inter-

pretar contexto y de ajustarse a ejemplos proporcionados hace posible generar pruebas consistentes a partir de requisitos de usuario, documentaciones de APIs o especificaciones técnicas, reduciendo así la carga manual y mejorando la cobertura de validación del software.[30]

2.3.3. Limitaciones y retos

A pesar de su enorme potencial, los **Modelos de Lenguaje Extensos** presentan una serie de limitaciones y retos que condicionan su aplicación práctica en el ámbito del *testing*. Uno de los problemas más destacados es la generación de **respuestas incorrectas o alucinadas**, es decir, salidas que aparentan ser plausibles pero que no se corresponden con la realidad. En el contexto de las pruebas de software, esto puede traducirse en casos de prueba mal estructurados, inconsistentes o que no reflejan fielmente los requisitos del sistema.[27]

Otro desafío importante es la **variabilidad en las respuestas**, dado que los LLMs no son completamente deterministas. Aunque esta característica puede aportar creatividad en ciertas tareas, en *testing* puede derivar en dificultades para obtener resultados consistentes y reproducibles. Asimismo, su conocimiento depende de los datos con los que fueron entrenados, lo que limita su capacidad para manejar dominios muy específicos o tecnologías emergentes sin un ajuste adicional.[27]

En el plano operativo, el uso de LLMs implica **costes computacionales** elevados, tanto en el entrenamiento como en la ejecución, así como una dependencia de servicios externos en muchos casos. Esto plantea cuestiones de **privacidad y seguridad**, especialmente cuando los *prompts* contienen requisitos sensibles o fragmentos de código que no deberían salir del entorno de la organización.[32]

Por último, aunque la integración de LLMs en procesos de *testing* abre oportunidades inéditas, también exige **nuevas prácticas de control de calidad y validación de resultados**. No basta con aceptar la salida del modelo como correcta: se requiere un marco de verificación adicional que garantice la utilidad y coherencia de las pruebas generadas. Estos retos no eliminan el valor de los LLMs, pero sí señalan la necesidad de un enfoque complementario en el que su potencial se combine con mecanismos de supervisión y con el criterio humano.[30]

2.3.4. Conclusión

Los **Modelos de Lenguaje Extensos** representan un avance disruptivo en el tratamiento del lenguaje natural y ofrecen un gran potencial para su aplicación en el ámbito del desarrollo de software. Su capacidad para transformar descripciones en lenguaje natural en estructuras formales abre nuevas posibilidades en la generación y mantenimiento de pruebas, aunque también plantea limitaciones relacionadas con la variabilidad de resultados, la dependencia de datos de entrenamiento y los aspectos de privacidad. Estas consideraciones permiten enmarcar el papel de la Inteligencia Artificial dentro del *testing* automatizado, tema que se desarrolla en el siguiente apartado, donde se analizan las principales aportaciones de la IA a esta disciplina.

2.4 Rol de la inteligencia artificial en el *testing* automatizado

La incorporación de la **Inteligencia Artificial (IA)** al *testing* automatizado ha transformado de manera significativa la forma en que se diseñan, ejecutan y mantienen las

pruebas de software. A diferencia de la automatización tradicional, cuyo alcance se limitaba principalmente a la ejecución repetitiva de casos de prueba predefinidos, la IA introduce la capacidad de **generar, priorizar y adaptar pruebas** en función de requisitos cambiantes o del comportamiento real del sistema bajo prueba. Esta evolución permite abordar uno de los principales retos de la industria: garantizar calidad en ciclos de desarrollo cada vez más rápidos y complejos, propios de los entornos ágiles y *DevOps* [33].

La IA desempeña un papel clave en distintas fases del ciclo de *testing*. Puede asistir en la **generación automática de casos de prueba** a partir de especificaciones en lenguaje natural, en la **optimización y priorización** de grandes suites de pruebas para reducir costes de ejecución, o en el **mantenimiento** de casos existentes cuando cambian los requisitos o la arquitectura del software. Asimismo, técnicas de aprendizaje automático han demostrado ser útiles en la **detección de defectos** y en la creación de oráculos de prueba capaces de evaluar resultados de forma automática [29].

En este apartado se analizan los principales roles de la IA en el *testing* automatizado, prestando atención tanto a sus beneficios como a las limitaciones y retos que todavía persisten. Este análisis servirá como base para comprender las oportunidades que ofrecen los LLMs y, posteriormente, justificar la herramienta propuesta en este trabajo.

2.4.1. Generación automática de casos de prueba

Esta es una de las áreas donde la Inteligencia Artificial ha mostrado un mayor impacto. Tradicionalmente, la automatización se había limitado a la ejecución de pruebas diseñadas manualmente, lo que suponía un alto coste de preparación y mantenimiento. Con la introducción de técnicas de IA, el enfoque cambia hacia la creación de casos de prueba de manera **automática o semiautomática**, reduciendo la dependencia del trabajo manual y mejorando la cobertura de validación.[34]

En un primer momento, la generación automática se abordó mediante algoritmos de búsqueda y técnicas basadas en modelos. Fraser y Arcuri (2012), por ejemplo, demostraron el uso de algoritmos evolutivos para generar *suites* completas de pruebas (conjuntos de pruebas) que maximizaban la cobertura estructural del software. Estos enfoques supusieron un avance importante, pero seguían dependiendo de modelos específicos o de heurísticas costosas de diseñar y ajustar.[35]

La irrupción del **deep learning** y de los **modelos de lenguaje** ha permitido avanzar hacia un paradigma distinto: generar pruebas directamente a partir de **documentación técnica o descripciones en lenguaje natural**. De este modo, los LLMs son capaces de transformar requisitos, historias de usuario o especificaciones OpenAPI en casos de prueba estructurados, abriendo la puerta a una mayor automatización del proceso y reduciendo la brecha entre analistas y *testers* [30, 22].

2.4.2. Priorización y optimización de suites

La **priorización y optimización de pruebas** constituye otro de los ámbitos donde la Inteligencia Artificial aporta un valor significativo. En sistemas de gran tamaño, ejecutar la totalidad de la suite de pruebas puede resultar costoso en términos de tiempo y recursos, lo que dificulta mantener ciclos de integración y entrega continua. En este contexto, la IA permite seleccionar de forma inteligente los casos de prueba más relevantes o reordenarlos en función de criterios como la criticidad del sistema, la probabilidad de fallo o el impacto esperado de un cambio en el código.[37]

Los enfoques tradicionales de priorización se basaban en heurísticas definidas manualmente, como la frecuencia de uso de un módulo o la criticidad de una funcionalidad.

Sin embargo, con la incorporación de técnicas de aprendizaje automático, es posible entrenar modelos que predicen qué casos de prueba son más propensos a detectar errores en cada iteración, lo que incrementa la eficiencia del proceso.[36]

La investigación reciente también ha explorado la aplicación de IA en la **optimización dinámica de suites de regresión**, en la que los conjuntos de pruebas se adaptan en tiempo real según los cambios en el código y los resultados previos [36, 37, 38].

2.4.3. Mantenimiento y evolución de suites de pruebas

El **mantenimiento y evolución de suites de pruebas** es uno de los aspectos más costosos y críticos en el ciclo de vida del *testing* automatizado. A medida que el software evoluciona, los requisitos cambian y las interfaces se modifican, lo que genera la necesidad de revisar constantemente los casos de prueba existentes. Sin mecanismos de actualización, las suites tienden a llenarse de casos obsoletos, redundantes o inconsistentes, lo que degrada su eficacia y aumenta los costes de validación.

La Inteligencia Artificial ofrece soluciones innovadoras para mitigar este problema. Modelos de aprendizaje automático pueden identificar **pruebas redundantes o ineficaces**, priorizar aquellas que siguen siendo relevantes y sugerir modificaciones en las que han quedado desfasadas. [22]. Los LLMs, en particular, aportan un valor añadido al facilitar la **regeneración de pruebas a partir de nuevas especificaciones**. Esta capacidad no solo reduce la carga de trabajo manual de los *testers*, sino que también asegura que la suite de pruebas se mantenga alineada con los requisitos actuales, minimizando el riesgo de ejecutar pruebas obsoletas o irrelevantes.[22, 30]

2.4.4. Análisis avanzado de resultados

El **análisis avanzado de resultados** es un área donde la Inteligencia Artificial está transformando la forma en que se interpretan y gestionan los resultados de pruebas. En los enfoques tradicionales, los *testers* deben revisar manualmente los registros de ejecución o confiar en reglas predefinidas para identificar errores, lo que puede ser ineficiente y propenso a omisiones. La IA introduce técnicas que permiten **automatizar la interpretación de salidas**, identificar patrones de fallo y generar oráculos de prueba más sofisticados.[39]

Los LLMs amplían este enfoque al permitir un **análisis contextual de los resultados**, relacionando fallos con los requisitos o documentación de la aplicación. Esto facilita la explicación de errores en un lenguaje cercano al usuario y la sugerencia de pasos de corrección, lo que convierte al análisis de resultados en un proceso más inteligente y útil para la toma de decisiones.

2.4.5. Limitaciones del uso de IA en el *testing* automatizado

El uso de **Inteligencia Artificial en el *testing* automatizado** ofrece ventajas notables frente a los enfoques tradicionales como se ha mostrado anteriormente. No obstante, su uso también puede conllevar ciertas limitaciones o riesgos a considerar. En primer lugar el uso de IA puede **producir resultados incorrectos o imprecisos** reduciendo la calidad de las pruebas generadas, lo que hace necesaria la validación manual de un ingeniero de pruebas capacitado[27]. No solo eso, sino que en casos donde se requiera entrenar el modelo utilizado con nuevos datos acarrea el **aumento significativo del coste computacional así como de posibles brechas de seguridad**, sobretodo en ámbitos empresariales cuando se hace uso de datos de cliente y/u organizaciones externas [27, 30].

2.4.6. Conexión con el caso de estudio

La aplicación de la Inteligencia Artificial al campo del *testing* automatizado ha abierto un nuevo escenario en el que es posible no solo ejecutar pruebas más rápido, sino también **generarlas, priorizarlas, mantenerlas y analizarlas de manera más inteligente**. Aunque las ventajas en términos de eficiencia, cobertura y adaptabilidad son evidentes, persisten retos relacionados con la explicabilidad, la fiabilidad y la privacidad de los datos. Estas limitaciones no anulan el potencial de la IA, sino que refuerzan la necesidad de enfoques híbridos en los que la automatización tradicional se complemente con nuevas soluciones basadas en modelos de aprendizaje automático y LLMs.

Precisamente, estas oportunidades y desafíos constituyen el punto de partida del caso de estudio presentado en este trabajo. La propuesta desarrollada busca aprovechar las capacidades de los LLMs para asistir en la generación y mantenimiento de pruebas, al tiempo que se diseñan mecanismos de control que mitiguen algunas de las limitaciones detectadas en la literatura. De este modo, el caso de estudio se plantea como una aproximación práctica que conecta la investigación teórica con una aplicación real en el ámbito del *testing* automatizado.

CAPÍTULO 3
Wakamiti: bases y funcionamiento

En este capítulo se aborda un cambio de perspectiva en el análisis del *testing* automatizado. Tras la revisión del papel de la inteligencia artificial en este ámbito, resulta pertinente detenerse en un enfoque metodológico ampliamente adoptado en la industria: el **Behaviour Driven Development (BDD)**. Este paradigma, concebido como una evolución del desarrollo guiado por pruebas (TDD), enfatiza la colaboración entre los distintos perfiles que intervienen en el proceso de desarrollo y el uso de un lenguaje común que facilite la comunicación de requisitos.

Con el fin de comprender de manera adecuada el funcionamiento de herramientas modernas de *testing*, como **Wakamiti**, se considera necesario introducir previamente los fundamentos de BDD y el lenguaje **Gherkin**, que constituye la base sintáctica sobre la que se construyen los escenarios de prueba. A partir de esta contextualización resulta posible examinar en detalle las características de Wakamiti, su arquitectura y las posibilidades que ofrece como entorno de pruebas extensible.

La estructura de este capítulo se organiza en tres apartados: en primer lugar, se presentan los principios de BDD; en segundo lugar, se analiza el lenguaje Gherkin y su aplicación en la definición de escenarios de prueba; y, por último, se expone Wakamiti como caso representativo de herramienta que implementa estos conceptos en un marco real de automatización.

3.1 Behaviour Driven Development (BDD)

El ***Behaviour Driven Development (BDD)*** es una metodología de desarrollo ágil que busca mejorar la comunicación entre los diferentes perfiles de un equipo de software (desarrolladores, equipo de pruebas y partes interesadas) mediante el uso de un lenguaje común y accesible. A diferencia de enfoques tradicionales de validación de software, BDD integra la definición de pruebas con la especificación de requisitos, reduciendo la brecha entre negocio y tecnología.

Surge a mediados de la década de 2000 como una evolución natural del **Test Driven Development (TDD)**. Mientras que TDD ponía el énfasis en escribir pruebas unitarias antes del código de producción, BDD amplió la perspectiva al incorporar un enfoque más cercano al lenguaje del negocio. Fue propuesto por **Dan North (2006)**, quien planteó que muchos equipos tenían dificultades para aplicar TDD de manera efectiva debido a la falta de claridad en la comunicación de requisitos.[40]

El aporte fundamental de BDD fue el uso de un **lenguaje ubicuo** basado en expresiones claras y estructuradas, que permitía describir el comportamiento esperado del sistema en términos entendibles para todos los actores del proyecto. Con el tiempo, BDD se

consolidó como un pilar en metodologías ágiles, integrándose en herramientas como Cucumber y extendiéndose más allá de las pruebas unitarias hacia escenarios de aceptación y validación de negocio [40, 41].

Actualmente, BDD se emplea no solo como una técnica de *testing*, sino como una filosofía de colaboración que ayuda a alinear las expectativas de negocio con la implementación técnica. Su evolución refleja la tendencia de la ingeniería de software hacia metodologías más iterativas, centradas en la comunicación y la entrega de valor continuo.

3.1.1. Principios fundamentales

Los **principios fundamentales del Behaviour Driven Development (BDD)** giran en torno a la idea de que el desarrollo de software debe estar guiado por el **comportamiento esperado del sistema**, expresado en un lenguaje accesible tanto para perfiles técnicos como no técnicos. A partir de esta premisa, BDD se apoya en los siguientes pilares:

- **Lenguaje ubicuo:** la comunicación entre desarrolladores, *testers* y *stakeholders* debe realizarse con un vocabulario común, evitando ambigüedades. El uso de un lenguaje estructurado pero natural facilita que todos los participantes comprendan los objetivos del sistema [40].
- **Colaboración continua:** BDD promueve la interacción constante entre negocio y tecnología, fomentando la definición de requisitos de manera conjunta. Esta práctica evita malentendidos y asegura que las funcionalidades implementadas aporten valor real al usuario final.
- **Especificación mediante ejemplos:** los requisitos no se expresan de forma abstracta, sino como **ejemplos concretos** que describen el comportamiento esperado del sistema. Esto permite verificar de manera objetiva si una funcionalidad cumple con lo solicitado [40].
- **Pruebas ejecutables como documentación viva:** los escenarios definidos en BDD, generalmente mediante el lenguaje Gherkin, no solo guían el desarrollo, sino que además funcionan como documentación actualizada y verificable automáticamente. De esta manera, la documentación no se queda obsoleta, sino que evoluciona junto al software.[41]
- **Iteración y retroalimentación rápida:** BDD está alineado con los principios ágiles, ya que busca entregar software de forma incremental, con ciclos cortos de validación y aprendizaje continuo[40].

En conjunto, estos principios convierten a BDD en una metodología que trasciende el *testing*, situándose como un marco de trabajo para **alinear requisitos, desarrollo y validación** en un único proceso colaborativo.

3.1.2. El lenguaje Gherkin

El **lenguaje Gherkin** constituye el núcleo sintáctico sobre el que se sustentan la mayoría de herramientas de BDD. Se trata de un lenguaje de especificación estructurado pero legible por humanos, cuyo propósito es describir el comportamiento esperado del software mediante **escenarios** fácilmente entendibles tanto por perfiles técnicos como no técnicos.[42]

Gherkin se caracteriza por diferentes aspectos. Se destaca principalmente por su simplicidad: utiliza un conjunto reducido de palabras clave (*Given*,*When* , *Then*, *And*, *But*) para expresar de manera clara las condiciones iniciales, las acciones realizadas y los resultados esperados de un escenario de prueba. No solo con eso, Gherkin permite el uso de diferentes idiomas en sus palabras clave, facilitando así el entendimiento de las pruebas, independientemente del idioma hablado. Además, Gherkin es capaz de ejecutar los escenarios desarrollados si estos se vinculan a implementaciones de pruebas automatizadas mediante *step definitions*. [42, 40]

Gracias a este enfoque, Gherkin cumple una doble función: por un lado, facilita la comunicación entre los distintos miembros del equipo al establecer un lenguaje común; y por otro, actúa como una **documentación viva**, ya que los escenarios pueden ejecutarse de manera automática con el soporte de *frameworks* de *testing* como Cucumber, SpecFlow o, en el caso que nos ocupa, **Wakamiti**.

Estructura de un escenario

La **estructura de un escenario en Gherkin** sigue un patrón simple y estandarizado que facilita tanto la comprensión como la ejecución automática de las pruebas. El esquema central se basa en las palabras clave vistas anteriormente (*Given*, *When*, *Then*, *And*, *But*):

- **Scenario (Escenario):** define el comienzo de un caso de prueba y describe el comportamiento esperado del programa.
- **Given (Dado):** describe el estado inicial del sistema o las precondiciones necesarias para ejecutar el escenario.
- **When (Cuando):** especifica la acción o evento que desencadena el comportamiento a probar.
- **Then (Entonces):** define el resultado esperado tras la ejecución de la acción, permitiendo validar si el sistema responde de forma correcta.
- **And (Y), But(Pero):** permite añadir condiciones a cada escenario además de las descritas por el resto de palabras clave.

Ejemplos ilustrativos

Para comprender mejor el funcionamiento de Gherkin, resulta útil observar ejemplos representativos que muestran cómo se describen diferentes tipos de escenarios. Estos ejemplos reflejan la simplicidad del lenguaje y su capacidad para expresar tanto casos básicos como situaciones más complejas.

En ejemplo típico de escenario que se puede plantear es el caso de validar un inicio de sesión en una aplicación:

```
1 Scenario: Inicio de sesión válido en la aplicación
2   Given que el usuario se encuentra en la página de inicio de sesió
      n
3   When introduce credenciales válidas
4   Then el sistema muestra la pantalla principal
```

Listing 3.1: Ejemplo de escenario en Gherkin

Otro ejemplo mas detallado es el siguiente:

```
Scenario: Retiro válido en un cajero automático
  Given que la tarjeta es válida
  And el usuario dispone de saldo suficiente
  When solicita retirar 100 euros
  Then el cajero entrega el dinero
  And el saldo de la cuenta se actualiza correctamente
```

Listing 3.2: Ejemplo de escenario positivo en Gherkin

En este caso, se utilizan las palabras clave *Given, When, Then* y *And* para detallar las condiciones previas, la acción del usuario y los resultados esperados de manera clara y verificable.

Gherkin también permite expresar **escenarios alternativos o excepcionales**, lo que facilita cubrir tanto casos de éxito como de fallo, como este caso, en el que se produce un inicio de sesión erróneo:

```
Scenario: Inicio de sesión con credenciales inválidas
  Given que el usuario está en la página de inicio de sesión
  When introduce una contraseña incorrecta
  Then el sistema muestra un mensaje de error
  But no permite el acceso al sistema
```

Listing 3.3: Ejemplo de escenario simple en Gherkin

Finalmente, el lenguaje incluye la palabra clave **Scenario Outline (Esquema del escenario)** para generalizar escenarios mediante el uso de parámetros, lo que resulta especialmente útil en pruebas de regresión ya que para una misma prueba se pueden probar diferentes combinaciones de datos:

```
Scenario Outline: Inicio de sesión con múltiples credenciales
  Given que el usuario está en la página de inicio de sesión
  When introduce <usuario> y <contraseña>
  Then el resultado debe ser <mensaje>

Examples:
  | usuario   | contraseña | mensaje                  |
  | user1     | pass1      | Acceso concedido         |
  | user2     | passX      | Credenciales incorrectas |
  | admin     | admin123   | Acceso concedido         |
```

Listing 3.4: Ejemplo de Outline en Gherkin

Estos ejemplos muestran cómo Gherkin permite describir comportamientos de manera estructurada, clara y ejecutable, abarcando desde pruebas unitarias hasta escenarios de aceptación. Su versatilidad explica por qué se ha convertido en el lenguaje estándar en la mayoría de *frameworks* BDD.

3.2 Wakamiti

Wakamiti (GitHub oficial: https://github.com/iti-ict/wakamiti) es una herramienta de automatización de pruebas de software de carácter *open source*, desarrollada por el **Instituto Tecnológico de Informática (ITI)** e inspirada en *Cucumber*. Esta herramienta esta desarrollada en Java y enfocada a las pruebas de caja negra haciendo uso del lenguaje natural gracias a la gramática **Gherkin** mencionada anteriormente.

Figura 3.1: Inicio de una ejecución en Wakamiti.
Fuente: http://bit.ly/3JPnc3I

En los apartados siguientes se analizarán su estructura de entorno, modelo de pruebas, arquitectura general y posibilidades de integración, con el fin de comprender cómo esta herramienta materializa los principios de BDD en un contexto práctico.

3.2.1. Filosofía y objetivos de Wakamiti

Su propósito principal es facilitar la definición, organización y ejecución de pruebas en distintos niveles (desde APIs hasta bases de datos). Wakamiti se sustenta en la idea de que las pruebas de software deben ser **accesibles, modulares y colaborativas**, siguiendo los principios del **BDD**. Bajo este enfoque, el *testing* deja de ser una tarea exclusiva de desarrolladores o *testers* y pasa a convertirse en una actividad compartida, en la que los requisitos funcionales se expresan en un lenguaje común y verificable. Por ello, Wakamiti adopta **Gherkin** como lenguaje ubicuo para la definición de pruebas, alineándose con la filosofía de BDD propuesta por North y ampliamente difundida en la industria [40].

En cuanto a sus **objetivos principales**, Wakamiti busca:

- **Unificación del lenguaje de pruebas:** permitir que los escenarios definidos en Gherkin sean comprensibles tanto para perfiles técnicos como para perfiles externos.
- **Extensibilidad y modularidad:** ofrecer una arquitectura abierta que permita integrar distintos tipos de pruebas (API, bases de datos, etc.) mediante *plugins* y pasos *custom*.
- **Automatización ágil:** facilitar la ejecución de pruebas en entornos de integración y entrega continua (CI/CD), reduciendo el tiempo de validación y mejorando la calidad del software entregado.
- **Documentación viva:** garantizar que los escenarios de prueba actúen como especificación y verificación al mismo tiempo, evitando la obsolescencia documental.

3.2.2. Arquitectura de Wakamiti

Wakamiti se basa en una arquitectura modular cuyo núcleo es un motor central (`wakamiti-engine`) que coordina la ejecución de pruebas y la gestión de plugins. Esta arquitectura se orga-

niza en tres capas principales: **Core Framework**, **Execution Layers** y **Extension Framework**. La Figura 3.2 muestra una visión general de la estructura y la interacción entre sus componentes.

En términos generales, Wakamiti se organiza en torno a un **núcleo central (core)** encargado de orquestar la ejecución de pruebas,las interfaces esenciales y la gestión de dependencias:

- **wakamiti-engine**: motor principal de ejecución de pruebas, responsable de coordinar la lógica interna y la comunicación entre módulos.
- **wakamiti-api**: define las interfaces y tipos necesarios para extender y utilizar el *framework*.
- **wakamiti-starter**: gestiona las dependencias básicas, facilitando la integración con otros módulos.

La **capa de ejecución** incluye diferentes puntos de entrada que permiten ejecutar pruebas en entornos variados. Ofrece flexibilidad en la ejecución de pruebas mediante distintos entornos y herramientas de desarrollo:

- **wakamiti-launcher**: interfaz de línea de comandos (CLI) que facilita la ejecución directa de pruebas.
- **wakamiti-junit**: integración con *JUnitRunner*, que permite ejecutar pruebas en entornos basados en JUnit.
- **wakamiti-maven-plugin**: integración con *Maven*, posibilitando la ejecución dentro de un ciclo de construcción estándar.

La **capa de extensión** permite el desarrollo de *plugins* (extensiones) y la integración con servicios adicionales. Su objetivo es facilitar la extensibilidad del *framework*, permitiendo a desarrolladores integrar nuevas funcionalidades y servicios personalizados:

- **wakamiti-plugin-starter**: arquetipo base para la creación de nuevos *plugins*.
- **jext**: componente encargado del descubrimiento de servicios.
- **jext-spring**: integración con el ecosistema *Spring*, que amplía las capacidades de configuración y despliegue.

La arquitectura se completa con un conjunto de **adaptadores o plugins**, que permiten extender la funcionalidad de Wakamiti para soportar distintos dominios de prueba, como APIs REST o bases de datos.

Además, Wakamiti incorpora un sistema de *reporting modular*: el motor escribe por defecto un archivo de resultados (archivo en formato JSON) junto con la salida por consola, y además, *plugins* de informe (p.ej., HTML) consumen esos resultados para generar distintos formatos para los resultados.

Gracias a este diseño, Wakamiti se diferencia de otras herramientas tradicionales al poner el énfasis en la **flexibilidad** y en su capacidad de **evolucionar** junto a las necesidades de cada proyecto.

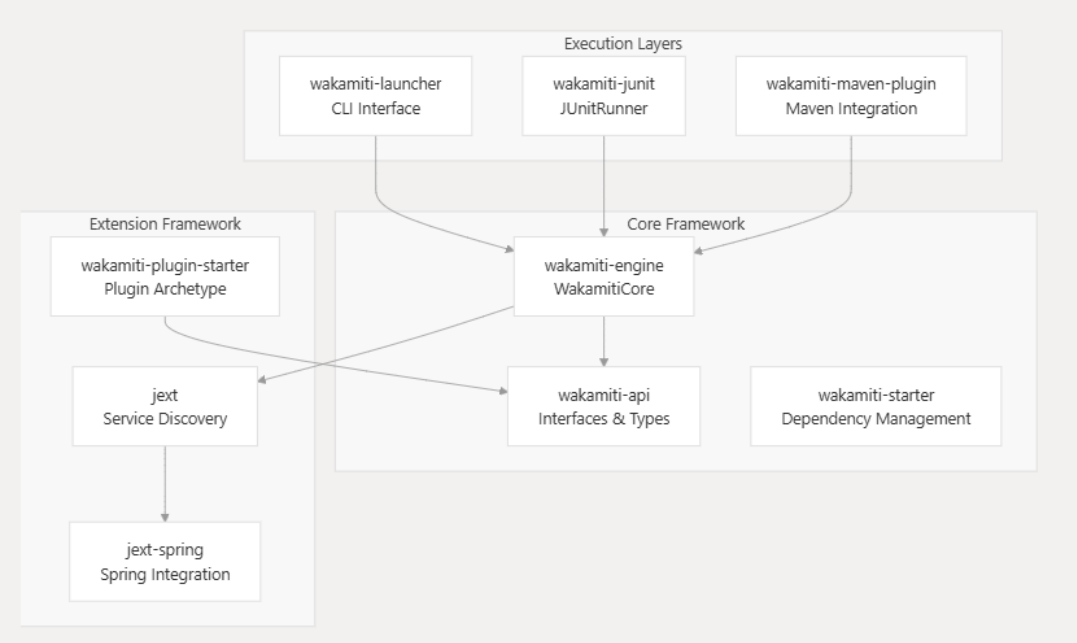

Figura 3.2: Arquitectura del *framework* Wakamiti
Fuente: http://bit.ly/4naDaDG.

3.2.3. Proceso de trabajo

El **proceso de trabajo en Wakamiti** sigue el flujo característico de las metodologías basadas en **Behaviour Driven Development (BDD)**, en el que la definición de escenarios de prueba precede a la automatización de su ejecución. Este enfoque garantiza que las pruebas se formulen en un lenguaje accesible para negocio y desarrollo, y que posteriormente puedan ejecutarse de manera automática sobre el sistema bajo prueba.

En términos generales, el ciclo de uso de Wakamiti puede resumirse en las siguientes fases:

1. **Definición de escenarios en Gherkin:** el proceso comienza con la redacción de ficheros `.feature` que describen, mediante la sintaxis *Dado-Cuando-Entonces*, los comportamientos esperados del sistema.

2. **Preparación del entorno de pruebas:** el usuario configura el proyecto de Wakamiti indicando los adaptadores o *plugins* necesarios (por ejemplo, para pruebas de APIs REST o bases de datos).

3. **Ejecución de pruebas:** una vez definidos los escenarios y configurado el entorno, Wakamiti procesa los ficheros `.feature`, planifica la ejecución y coordina el motor de ejecución con los adaptadores correspondientes.

4. **Generación de informes:** durante la ejecución, Wakamiti emite eventos que son consumidos por distintos módulos, lo que permite obtener resultados en múltiples formatos (consola, JSON, HTML).

3.2.4. Estructura del entorno de pruebas

La **estructura del entorno de pruebas en Wakamiti** refleja su filosofía modular y extensible. Wakamiti organiza sus proyectos en torno a los archivos `.feature` escritos y los

complementa con configuraciones y componentes adicionales que permiten ejecutar las pruebas de manera controlada y reproducible.

Un entorno típico de pruebas con Wakamiti incluye los siguientes elementos:

- **Archivos `.feature`:** constituyen el núcleo del entorno, ya que describen en lenguaje Gherkin los comportamientos esperados del sistema. Cada fichero contiene una o varias características que agrupan escenarios relacionados.
- **Configuración del proyecto:** a través de archivos de configuración (por ejemplo, en formato YAML) se definen aspectos como los *plugins* a utilizar y su respectiva configuración, las rutas de los ficheros de prueba, las variables de entorno y parámetros de ejecución.
- **Archivos de datos:** se encargan de preparar los datos necesarios para cada una de las pruebas que lo necesite. Principalmente se tratan de *scripts SQL* que se ejecutan durante la ejecución de las pruebas descritas en los archivos `.feature`.
- **Informes:** como resultado de la ejecución, el entorno produce salidas en diversos formatos (consola, JSON, JUnit XML), que pueden almacenarse localmente o integrarse en *pipelines* de CI/CD.

Esta organización garantiza que el entorno de pruebas sea **reproducible, extensible y portable**, permitiendo que distintos equipos puedan compartir configuraciones y escenarios sin pérdida de consistencia. En este sentido, Wakamiti convierte los proyectos de *testing* en un recurso reutilizable y que se puede mantener a lo largo del ciclo de vida del software.

3.2.5. Fortalezas y limitaciones de Wakamiti

El análisis de **Wakamiti**, basado en su diseño arquitectónico, documentación oficial y la comparativa con otras herramientas de pruebas, permite identificar una serie de fortalezas y limitaciones que ayudan a comprender su posición en el ecosistema de la automatización de pruebas.

Fortalezas

- **Arquitectura modular y extensible:** el núcleo es ligero y permite ampliar sus capacidades mediante *plugins*, lo que facilita la incorporación de nuevos dominios de prueba sin necesidad de modificar la lógica central.
- **Flexibilidad tecnológica:** existen adaptadores para múltiples ámbitos (APIs REST, bases de datos, Azure, entre otros), lo que incrementa su versatilidad en entornos heterogéneos.
- **Enfoque BDD y legibilidad:** el uso de Gherkin asegura que los escenarios sean comprensibles tanto para perfiles técnicos como de negocio, reforzando la comunicación entre las partes interesadas.
- **Integración con CI/CD y ecosistema *DevOps*:** la compatibilidad con ejecución mediante CLI, Maven y JUnit, junto con la exportación de resultados en formatos estándar (JSON, HTML, JUnit XML), facilita su uso en *pipelines* de integración continua.

- **Soporte para validaciones avanzadas:** incluye comparadores capaces de manejar estructuras complejas (JSON, XML, expresiones regulares) y permite enmascarar propiedades sensibles en los reportes.
- **Proyecto abierto y en evolución:** su naturaleza *open source* bajo licencia MPL 2.0 y la disponibilidad de repositorio público en GitHub garantizan transparencia y la posibilidad de contribuciones externas.

Limitaciones

- **Curva de aprendizaje inicial:** al ser menos conocido que *frameworks* más consolidados, existen menos recursos externos, tutoriales o foros activos de soporte.
- **Dependencia de *plugins*:** el núcleo no incluye pasos básicos integrados, por lo que es necesario instalar y configurar los adaptadores adecuados. En caso de que un *plugin* no exista, debe desarrollarse, lo que requiere conocimientos técnicos específicos.
- **Ecosistema reducido:** su comunidad es todavía pequeña en comparación con alternativas más populares, lo que limita la disponibilidad de integraciones inmediatas con herramientas de terceros.
- **Mantenimiento y estabilidad:** algunas sugerencias abiertas en el repositorio muestran que aún existen aspectos en evolución. La continuidad depende de la actividad del ITI y de la participación comunitaria.

Recapitulando, Wakamiti destaca por su **modularidad**, **flexibilidad tecnológica** y su alineación con los principios de **BDD**, lo que lo convierte en una herramienta potente para equipos que buscan adaptabilidad y trazabilidad en sus pruebas. Sin embargo, su **ecosistema reducido**, la **dependencia de plugins** y su menor difusión internacional limitan, por el momento, su competitividad frente a *frameworks* más consolidados.

3.2.6. Ejemplos ilustrativos

```
Escenario: Inicio de sesión válido y acceso al dashboard
  Dado que el usuario abre la aplicación
  Que introduce usuario "alice" y contraseña válida
  Entonces visualiza el dashboard con el mensaje "Bienvenida, Alice
      "
```

Listing 3.5: Ejemplo de escenario sencillo

```
Esquema del escenario: Validación de contrato y negocio en /orders
  Dado que existe el endpoint "/orders/<id>"
  Cuando realizo una petición GET a "/orders/<id>"
  Entonces la respuesta tiene código 200
  Y el cuerpo cumple el esquema "schemas/order.json"
  Y el campo "status" es "<estadoEsperado>"

Ejemplos:
  | id  | estadoEsperado |
  | 101 | SHIPPED        |
  | 202 | PENDING        |
```

Listing 3.6: Ejemplo de Escenario outline

3.2.7. Conclusión

Wakamiti constituye una plataforma versátil y extensible para la automatización de pruebas de software. Su enfoque modular y el uso de Gherkin como lenguaje unificado lo convierten en una alternativa interesante frente a otras herramientas más específicas. No obstante, su dependencia de la creación manual de escenarios y de las *step definitions* abre la puerta a la búsqueda de mejoras y ampliaciones, que se abordarán en capítulos posteriores.

CAPÍTULO 4
Estado del arte y herramientas existentes

4.1 Herramientas actuales para *testing* automático

El *testing* automático ha sido una práctica consolidada durante décadas, con herramientas que han permitido reducir el tiempo de ejecución de pruebas y mejorar la calidad del software. Estas soluciones, ampliamente adoptadas en la industria, requieren que los casos de prueba sean diseñados y mantenidos manualmente, lo que implica un esfuerzo considerable en proyectos de larga duración. A continuación, se presentan algunas de las herramientas más representativas, cubriendo distintos enfoques y tipos de pruebas.

En primer lugar encontramos **Selenium** [43] es un *framework* de código abierto diseñado para la automatización de navegadores web. Su principal fortaleza radica en la compatibilidad con múltiples lenguajes de programación, como Java, Python o C#, así como en la integración con *frameworks* de *testing* tales como TestNG y JUnit. Gracias a estas características, se ha consolidado como una de las soluciones más utilizadas en la industria para la validación de aplicaciones web. Selenium no está conformado por un único complemento, si no que esta compuesto por diferentes herramientas y bibliotecas para poder aplicarlo a diferentes navegadores. Selenium se destaca por el amplio soporte de navegadores que ofrece y su flexibilidad para adaptarse a diferentes entornos. No obstante, presenta limitaciones asociadas a una curva de aprendizaje pronunciada y al elevado coste de mantenimiento cuando las interfaces cambian con frecuencia.

Por otro lado **JUnit y NUnit** [44, 45] son *frameworks* de pruebas unitarias ampliamente utilizados en los lenguajes Java y C#, respectivamente. Ambos proporcionan anotaciones y mecanismos de aserción que permiten estructurar y ejecutar pruebas unitarias. Gracias a estas características, se han convertido en herramientas fundamentales en la construcción de software orientado a la calidad, favoreciendo la detección temprana de errores y la automatización de procesos de validación. Entre sus principales ventajas destacan su ligereza, la facilidad de integración con entornos de desarrollo integrados (IDE) y la compatibilidad con flujos de integración y entrega continua (CI/CD). No obstante, presentan limitaciones derivadas de su enfoque restringido a pruebas unitarias, lo que implica la necesidad de codificar manualmente cada caso de prueba.

Otra herramienta dentro de este ámbito es el conjunto entre **Postman [47] y Newman [48]**. Postman es una herramienta ampliamente utilizada para la validación y documentación de APIs REST, mientras que Newman constituye su motor de línea de comandos (CLI) que permite la ejecución automatizada de pruebas. Postman ofrece la posibilidad de crear colecciones de peticiones, asociarles pruebas automatizadas y organizar la do-

cumentación correspondiente, lo que facilita la colaboración entre equipos de desarrollo. Newman, por su parte, amplía estas capacidades al integrarse de manera fluida en *pipelines* de integración y entrega continua (CI/CD), consolidándose como una solución práctica tanto para la fase de desarrollo como para la de despliegue.Este ecosistema se destaca por la interfaz gráfica intuitiva de Postman, la facilidad para documentar y compartir pruebas, y la integración sencilla con entornos CI/CD mediante Newman. No obstante, presentan limitaciones asociadas a la necesidad de redactar *scripts* en JavaScript para automatizaciones más avanzadas y a su orientación exclusiva hacia pruebas de APIs.

A continuación se abordará la herramienta **Dredd** [49]. Es una herramienta orientada a la validación de APIs mediante pruebas basadas en contrato. Su funcionamiento consiste en comparar las respuestas reales de una API con las definiciones establecidas en especificaciones como OpenAPI o Swagger, lo que permite verificar la coherencia entre el comportamiento implementado y el contrato acordado. Entre sus funcionalidades principales se encuentran la validación de códigos de estado HTTP, la verificación de la estructura de los datos devueltos y la comprobación del cumplimiento de las reglas definidas en la especificación. Gracias a estas capacidades, Dredd se ha consolidado como una solución precisa y confiable para garantizar la calidad de las interfaces de programación.

Sus principales ventajas incluyen la alta precisión en la validación de contratos, la facilidad de integración con múltiples lenguajes y la compatibilidad con flujos de integración y entrega continua (CI/CD). No obstante, presenta limitaciones derivadas de su dependencia de especificaciones actualizadas y de la ausencia de pruebas funcionales que evalúen aspectos externos al contrato.

Finalmente se describe **Cypress** [46]. Se trata de un *framework* moderno orientado a la automatización de pruebas *end-to-end (E2E)* en aplicaciones web. Se distingue por su capacidad de ejecutar pruebas directamente en un navegador real, lo que permite una mayor fidelidad en la validación del comportamiento de las interfaces. Además, ofrece un entorno de depuración visual y una API de *scripting* en JavaScript que facilita la escritura de pruebas, lo que lo convierte en una herramienta accesible para desarrolladores que ya trabajan con dicho lenguaje. Gracias a estas características, Cypress ha ganado popularidad en la industria como una alternativa ágil frente a soluciones tradicionales de *testing* web.

Destaca por la experiencia de desarrollo integrada, la ejecución de pruebas en tiempo real y la disponibilidad de documentación clara y extensa que respalda su adopción. Sin embargo, presenta limitaciones derivadas de su enfoque exclusivo en aplicaciones web y de su elevado consumo de recursos en proyectos de gran escala con un alto volumen de pruebas.

4.1.1. Resumen comparativo de tipos de pruebas

Herramienta	UI	APIs	Pruebas unitarias	CI/CD	Lenguaje natural
Selenium	✓	✗	✗	✓	✗
JUnit / NUnit	✗	✗	✓	✓	✗
Postman+Newman	✗	✓	✗	✓	✗
Dredd	✗	✓	✗	✓	✗
Cypress	✓	✗	✗	✓	✗
Wakamiti	✗	✓	✓	✓	✓

Tabla 4.1: Comparación de herramientas de *testing* convencionales según los tipos de pruebas a las que se aplican.

4.1.2. Conclusión

Las herramientas convencionales de *testing* automático han demostrado ser esenciales para la calidad del software moderno, permitiendo validar de forma repetitiva y fiable diferentes aspectos del sistema. Sin embargo, su eficacia depende de la capacidad del equipo para diseñar, escribir y mantener los casos de prueba, lo que puede convertirse en un cuello de botella en entornos con cambios frecuentes. En este contexto, la integración de IA —como en el caso del *plugin* para Wakamiti— ofrece la oportunidad de reducir estos costes, acelerar la creación de pruebas y mejorar la adaptabilidad del proceso de *testing*.

4.2 Herramientas con IA para generación de pruebas automáticas

En los últimos años, el uso de *Inteligencia Artificial* (especialmente modelos de lenguaje extenso (LLM)) ha impulsado el desarrollo de herramientas capaces de generar, mantener y optimizar casos de prueba de software de forma automática o semiautomática. Estas soluciones interpretan requisitos escritos en lenguaje natural, analizan especificaciones técnicas o examinan el propio código para producir pruebas listas para su ejecución.

Aunque muchas de estas herramientas aún se encuentran en una fase de madurez temprana, su adopción está creciendo rápidamente, especialmente en entornos ágiles y de *DevOps*, gracias a ventajas como la reducción del tiempo de desarrollo de las pruebas y la capacidad de adaptación a cambios frecuentes en el software.

A continuación, se describen algunas de las herramientas más relevantes en este ámbito, seleccionadas tanto por su notoriedad en el sector como por la oportunidad que ofrecen para contrastar sus características con las de Wakamiti.

Un primer ejemplo es **Qodo**, cuyo objetivo principal es la generación de pruebas unitarias e integraciones simples directamente a partir del código fuente. Su mayor ventaja radica en la integración con entornos de desarrollo populares como VS Code o JetBrains, lo que facilita que el programador obtenga rápidamente un conjunto de pruebas iniciales basados en la lógica del código. No obstante, se trata de una herramienta centrada casi exclusivamente en el nivel unitario, sin soporte nativo para escenarios expresados en lenguajes legibles como Gherkin ni para pruebas de mayor alcance funcional [50].

Por otro lado, **TestRigor** propone un enfoque distinto: traducir instrucciones en lenguaje natural a pruebas funcionales ejecutables. Este planteamiento permite que perfiles

no técnicos participen en la definición de casos de prueba, lo que constituye una de sus principales fortalezas. Sin embargo, la expresividad de este lenguaje natural está limitada en escenarios muy complejos y, en la práctica, requiere un aprendizaje progresivo por parte del usuario para redactar instrucciones adecuadas [51].

En una línea similar, **Functionize** busca transformar requisitos de negocio en pruebas funcionales web. Su valor añadido se encuentra en la integración con herramientas de gestión de requisitos y la capacidad de ejecutar pruebas en múltiples navegadores, lo que resulta útil en entornos corporativos. Aun así, su aplicación está restringida principalmente al ámbito de las aplicaciones web, lo que reduce su aplicabilidad en otros dominios [52].

Otra herramienta representativa es **Mabl**, enfocada en pruebas de regresión y validación visual apoyadas en IA. Su propuesta de valor reside en la capacidad de detectar automáticamente cambios en la interfaz y ofrecer soporte para pruebas de accesibilidad, lo que la convierte en una solución interesante para equipos centrados en la experiencia de usuario. Su limitación, al igual que Functionize, es que está orientada de forma predominante al *testing* de aplicaciones web, con menor alcance en otros tipos de pruebas [53].

Finalmente, **Katalon Studio** constituye una de las propuestas más amplias al incorporar capacidades de inteligencia artificial en una plataforma ya consolidada para pruebas automatizadas. Se trata de una solución multiplataforma que abarca aplicaciones web, móviles, APIs e incluso de escritorio, lo que supone una ventaja competitiva respecto a las anteriores. No obstante, su soporte para lenguajes naturales es solo parcial y depende de configuraciones adicionales [54].

En conjunto, estas herramientas evidencian el potencial de la inteligencia artificial en el ámbito del *testing*, pero también muestran limitaciones comunes. Todas ellas resultan útiles en contextos concretos (ya sea pruebas unitarias, funcionales E2E o validación visual), pero tienden a estar especializadas en un tipo de pruebas y carecen de un soporte integral que combine de forma completa la expresividad BDD, la automatización multiplataforma y la capacidad de integración fluida en diferentes flujos de desarrollo.

4.2.1. Resumen comparativo sobre tipos de pruebas y lenguaje natural

Característica	Pruebas unitarias	Pruebas E2E	Prueba APIs	Lenguaje natural
Codium	✓	✗	✗	✗
TestRigor	✗	✓	✗	✗
Functionize	✗	✓	✗	✗
Mabl	✗	✓	✗	✗
Katalon	✗	✓	✓	✓
Wakamiti	✓	✓	✓	✓

Tabla 4.2: Comparación de herramientas con IA para generación de pruebas automáticas según tipos de pruebas que se abordan.

4.2.2. Conclusión

Las herramientas analizadas demuestran el potencial de la IA para agilizar la creación y mantenimiento de pruebas de software. Sin embargo, presentan limitaciones significa-

tivas: muchas dependen de plataformas propietarias, están especializadas en un solo tipo de prueba o no utilizan lenguajes de especificación estructurados como Gherkin. En este contexto, la propuesta de integrar generación de pruebas mediante IA en Wakamiti ofrece una ventaja competitiva clara: combina un modelo *open source* con soporte multiárea y la posibilidad de mantener las pruebas de forma más accesible y estandarizada.

CAPÍTULO 5

Uso de la API de ChatGPT para la generación automática de pruebas

Este capítulo aborda el desarrollo de una herramienta para la **generación asistida de pruebas de software mediante el uso de la API de ChatGPT**, que constituye la principal aportación práctica de este trabajo. La motivación para esta propuesta surge de las limitaciones detectadas en los enfoques tradicionales de automatización: la necesidad de redactar manualmente escenarios de prueba en lenguajes como Gherkin, el esfuerzo asociado al mantenimiento de *suites* extensas y la dificultad de adaptar las pruebas a cambios frecuentes en los requisitos.

La solución presentada no sustituye a los *frameworks* de pruebas como Wakamiti, sino que los complementa. Su función actual se centra en **producir definiciones de escenarios en un formato cercano al BDD**, expresados en lenguaje natural estructurado, a partir de descripciones funcionales o requisitos textuales. Estas definiciones pueden integrarse posteriormente en proyectos de Wakamiti para su ejecución, pero en esta versión inicial **no se generan implementaciones de pasos (*step definitions*) ni se ejecutan directamente las pruebas**.

El empleo de **Modelos de Lenguaje Extenso (LLMs)**, y en particular de la API de ChatGPT (OpenAI), permite reducir el esfuerzo de diseño de pruebas y aumentar la cobertura mediante la generación automática de escenarios representativos. En este capítulo se describen el **diseño del sistema** y su integración con la API, el **proceso de construcción de *prompts* y extracción de contexto** y la **transformación de respuestas en definiciones de escenarios**.

5.1 Requisitos funcionales

En esta sección se describen los requisitos funcionales relacionados con el desarrollo del plugin de generación automática de casos de prueba utilizando un modelo de lenguaje (ChatGPT). El objetivo principal es facilitar la creación de pruebas a partir de la definición de un *swagger* de una API, reduciendo así el esfuerzo manual de los ingenieros de prueba y garantizando una mayor cobertura.

5.1.1. CU.0001 - Definición del funcionamiento de la herramienta utilizando ChatGPT4

Descripción
Para que el *plugin* de creación automática funcione de forma correcta, se necesitan las siguientes entradas: JSON del swagger o URL.

Excepciones
N/A

Parámetros de entrada
JSON del swagger de los sistemas a probar o URL donde está desplegado.

Parámetros de salida
Casos de test generados.

Actores
Ingeniero de test.

Precondiciones
El ingeniero de test debe disponer de un *token* válido de ChatGPT4.

Escenario
Para que los ingenieros de test puedan disminuir el tiempo que pasan en la definición de casos de test, se va a crear un generador de definiciones de pruebas que utilice un LLM y genere los casos de test automáticos a partir de una definición de un swagger. Para poder utilizar ChatGPT4 se requiere de un *token* válido. A partir de ahí, se deberá programar un *prompt* para indicar al LLM el formato necesario en la definición de los casos de test.

Diseño del *prompt*

Una vez aislado el fragmento de especificación correspondiente a cada operación, la herramienta construye el ***prompt*** que se enviará al modelo de lenguaje. El diseño de este *prompt* es un aspecto crítico, ya que condiciona la calidad, la coherencia y la utilidad de los escenarios generados.

Para garantizar resultados consistentes, el sistema utiliza un ***prompt* fijo y versionado** almacenado en el repositorio (`src/features/generate_test_level_1/prompt.txt`). Este enfoque asegura **reproducibilidad** y facilita la trazabilidad, dado que cualquier modificación en la plantilla de generación queda registrada en el control de versiones.

El *prompt* se organiza en secciones diferenciadas que cumplen funciones específicas:

- **Contexto general:** explica al modelo el objetivo de la tarea, el público al que va dirigido (*product owners*) y el encaje de los escenarios en un flujo de desarrollo ágil con CI/CD.
- **Pasos de análisis:** instruyen al modelo para examinar el `schema` de entrada, identificar parámetros, cuerpos de petición, códigos de respuesta y posibles validaciones, y a partir de ellos generar un conjunto de pruebas que cubra tanto casos de éxito como de error.
- **Restricciones:** definen reglas estrictas de estilo (por ejemplo, no usar primera persona, no finalizar pasos con punto, no inventar valores de parámetros) y de cobertura (priorizar el caso feliz y los errores comunes, generar hasta 20 escenarios no redundantes).

- **Formato de salida:** obliga al modelo a devolver la respuesta en estructura Gherkin, incluyendo cabecera de idioma (`#language: es` cuando procede), *tags* o etiquetas como `@definition`, `@smoke` y `@error`, así como identificadores únicos por escenario siguiendo un patrón.
- **Esquema y ejemplo:** muestran al modelo cómo debe estructurarse la salida final, facilitando que las generaciones cumplan con el estilo deseado.

El *prompt* es el siguiente:

```
<CONTEXTO>

Objetivo: Generar la definición de casos de prueba para asegurar que el API cumpla con todas sus especificaciones y manejar todos los casos de posibles.

Público: Product owners.

Flujo de trabajo: Parte del flujo de desarrollo ágil, donde se integran pruebas automáticas como parte de la CI/CD.

Finalización exitosa: Obtener un conjunto completo de escenarios de prueba exhaustivo y específico para cada endpoint y comportamiento del API. Debes incluir tantos escenarios como sea posible y cada uno contendrá en formato gherkin. Se debe seguir todos los pasos y cumplir con el formato y restricciones indicadas para que el resultado sea válido.

<PASOS>
1. Analiza detalladamente el schema que te indicaré como input para identificar la ruta y su método HTTP, sus parámetros de entrada y salida, tipos de parámetros (query, path, body), validaciones asociadas (requeridos, formatos, etc.), códigos de respuesta de éxito y error (excepto 5xx), tipos de datos y modelos definidos en los esquemas de entrada y salida.
2. Diseña todos los posibles casos de prueba funcionales:
    - Caso feliz (smoke) que incluya la información mínima para asegurar que la operación funciona.
    - Si la información disponible lo permite, valora posibles casos funcionales exploratorios y/o hipotéticos que puedan ser de interés.
    - Casos que validen todos los caminos lógicos de la operación que terminen en éxito, jugando con las posibles combinaciones de parámetros y cuerpos de solicitud (parámetros opcionales ausentes, límites, valores extremos, máximos, mínimos, longitudes máximas y mínimas, etc).
3. Casos de validación de entradas inválidas o mal formadas (parámetros omitidos, tipos de datos incorrectos, valores fuera de rango, pruebas límite de máximos y mínimos, etc).
4. Pruebas funcionales inválidos (recursos no encontrados, autenticación inválida, etc.)
5. Valida que las pruebas diseñadas cubran todos los escenarios prosibles y si hay algún vacío en la cobertura.
6. Haz un listado de tantas dudas como sea posible que se te ocurran sobre el comportamiento funcional de la operación que se le puedab plantear al product owner.

<RESTRICCIONES>
```

```
Es extremamente importante que cumplas estos puntos:
- Incluye todos los casos encontrados y, si y sólo si la operación lo permite, es obligatorio que lleguen a 20.
- No incluir escenarios que no aporten valor, como por ejemplo no indicar un path parameter en la url o que se repita una funcionalidad que ya se prueba en otro escenario.
- No incluir valores de ejemplo o datos ficticios (salvo en los casos límite de validación de datos), sólo la descripción de cómo serán los datos. Hay que limitarse a indicar los pasos necesarios de una forma genérica y breve para que perfiles no técnicos entiendan el funcionamiento del escenario.
- Guíate siempre por las buenas prácticas de gherkin e ISTQB. Nunca me escribas los pasos en primera persona. No acabes los pasos con punto o coma.
- No debe haber referencias al valor de parámetros en los pasos (indicados con '{}').
- Prioriza los casos principales (escenario de éxito y errores más comunes) antes de considerar casos exploratorios o hipotéticos, pero incluye alguno si encuentras.
- Cuando interpretes el comportamiento de los parámetros del contrato, o alguno de sus valores, justifícamelo en la descripción del escenario.
- Los escenarios deben ser generados en el siguiente orden: 1-Caso feliz (smoke), 2-Casos funcionales que terminen en éxito, 3-Validaciones de entradas válidas, 4-Validaciones de entradas inválidas, 5-Casos funcionales que terminen en error.

<FORMATO>
Quiero que me generes una respuesta única en la que se incluya el feature descrito en formato gherkin, en el idioma que te indique en el input.
- Si el idioma no es el inglés, deberás indicarlo al principio del feature, expresado como un comentario, de la forma '#language: xx', si es inglés no incluyas esta línea.
- El feature debe contener una serie de tags: siempre el tag '@definition', y el input indicaré opcionalmente un 'apiId' (si no está, no incluir ese tag) y un 'operationId' (siempre se deberá informar).
- Cada escenario deberá contener un tag identificativo y único para todos los escenario que generes con el patrón '@ID-*'.
- Los escenarios que prueben casos de error (con codigos 4xx), deben contener siempre el tag '@error' y se colocarán al final, el caso principal (caso feliz) el tag '@smoke' y solo habrá 1 por feature, el primero.

<ESQUEMA>
#language: {lenguaje}
@definition @{apiId} @{operationId}
Característica: {descripción breve de la característica, máximo 10 palabras}
  {descripción funcional detallada de la característica}

  @ID-{operationId}-01 @smoke
  Escenario: {descripción breve del escenario, máximo 10 palabras}
   {descripción funcional detallada del escenario, razon de ser}
   {pasos Dado, Y, Cuando, Entonces, O, *, máximo 15 palabras}

  {resto de escenarios}
```

```
  @ID-{operationId}-XX @error
  Escenario: {descripción breve del escenario erroneo (codigo 4xx),
      máximo 10 palabras}

# {dudas}

<EJEMPLO>
input:
language=es
operationId=something

resultado:
```gherkin
#language: es
@definition @something
Característica: Una descripción de ejemplo
  Una descripción del comportamiento funcional detallada del
     escenario

  @ID-something-01 @smoke
  Escenario: Descripción breve del caso feliz
    Una descripción detallada del comporamiento y razón de ser del
       escenario
       Dado un paso de ejemplo


  @ID-something-02
  Escenario: Decripción breve de un caso funcional
    Una descripción detallada del comporamiento y razón de ser del
       escenario
    Cuando un paso de ejemplo

  @ID-something-03 @error
  Esquema del escenario: Descripción breve casos de error validació
     n de datos
    Una descripción detallada del comporamiento y razón de ser del
       escenario
    Entonces un paso de ejemplo parametro <param1>
    * resto de pasos <param2>
    Ejemplos:
      | param1 | param2 |
# Dudas:
# Una duda sobre al comportamiento funcional del contrato
# Otra duda

```
<INPUT>
```

Listing 5.1: Prompt utilizado en la herramienta de generación de definiciones de pruebas

Además del *prompt* base, la herramienta añade información contextual dinámica:

- El fragmento de especificación OpenAPI de la operación (`schema`).
- El idioma objetivo (`language`, por defecto español).
- El `operationId` y, opcionalmente, el `apiId`.

La combinación de estos elementos produce un *prompt* completo que instruye al modelo sobre qué generar y en qué formato exacto devolverlo.

Este enfoque se conoce como **ingeniería de prompts** (*prompt engineering*), práctica mediante la cual se estructura cuidadosamente la instrucción al modelo para mejorar la precisión y reducir las ambigüedades [55, 56]. En este caso, la separación en secciones claras y la imposición de restricciones de formato persiguen disminuir el riesgo de respuestas incoherentes o incompatibles con el flujo de pruebas.

5.1.2. CU.0002 - Generación de definiciones multiproveedor

Descripción
Se debe poder cambiar de proveedor de LLM en el código fuente de la herramienta introduciendo uno de los siguientes términos: 'openai', 'deepseek-chat', 'gemini-2.0-flash'
Excepciones

No se puede hacer uso de ningún proveedor no implementado.
Parámetros de entrada

N/A
Parámetros de salida

N/A
Actores

Ingeniero de test.
Precondiciones

El ingeniero de test debe disponer de un *token* válido de OpenAI, Gemini o Deepseek según el proveedor. seleccionado.
Escenario

El ingeniero de test entra en el archivo runPrompChatGPT.ts y cambia el proveedor de LLM cambiando el valor de la constante OpenAI_PROVIDER.

El programa enviará la petición de generación al modelo seleccionado y este devolverá el resultado generado.

5.1.3. CU.0003 - Gestión de escritura de los archivos generados

Descripción
Se debe poder forzar la sobre-escritura de las carpetas de las pruebas generadas mediante el comando –force por la consola al ejecutar la herramienta
Excepciones

N/A
Parámetros de entrada

N/A
Parámetros de salida

N/A
Actores

Ingeniero de test.
Precondiciones

N/A
Escenario

El ingeniero de test quiere forzar la sobre-escritura de la carpeta resultado. Para ello ejecuta el comando de inicio con la opción –force.

Resultado: Se elimina la carpeta de resultados antes de generar los test y se crea una carpeta nueva.

5.2 Estructura y diseño del sistema

El sistema desarrollado se concibe como una **herramienta de generación asistida de pruebas automáticas**, diseñada para transformar contratos de API (en formato OpenAPI/Swagger) en artefactos Gherkin listos para ser integrados en un flujo de pruebas BDD. Su objetivo principal no es ejecutar las pruebas, sino **automatizar la fase de diseño y redacción**, reduciendo la carga manual que tradicionalmente afrontan los equipos de calidad y los *product owners*.

La estructura del sistema se inspira en principios de modularidad y separación de responsabilidades, lo que facilita su mantenibilidad y evolución. La Figura 5.1 muestra una visión esquemática de la organización del sistema.

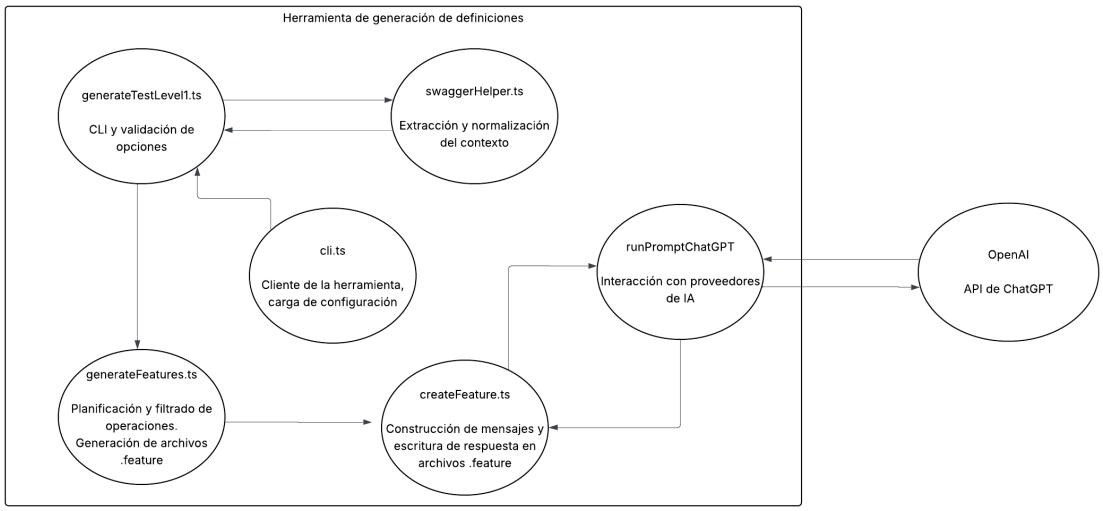

Figura 5.1: Estructura modular del sistema de generación de pruebas

5.2.1. Explicación de la estructura modular

La estructura adoptada en la herramienta responde a un enfoque **modular y desacoplado**, donde cada componente desempeña un rol específico dentro del flujo de generación de pruebas. Esta separación permite mejorar la mantenibilidad, facilitar la extensibilidad y reducir el acoplamiento entre partes del sistema. La Figura 5.1 refleja esta división de responsabilidades.

Módulo de interacción con el usuario

El módulo `cli.ts` constituye la puerta de entrada al sistema. Su responsabilidad es la de **gestionar la comunicación con el usuario**, interpretando parámetros proporcionados por línea de comandos o cargados desde el archivo `wakamiti.yaml`. De este modo, el resto de módulos quedan aislados de detalles de interacción, centrándose únicamente en la lógica de negocio.

Módulo de validación de opciones y gestión de contexto

El módulo `generateTestLevel1.ts` es el encargado de interpretar las opciones que recibe del CLI, gestiona el envío de datos para su normalización, carga el *prompt* fijo y envía los datos tratados a la capa de planificación y filtrado.

Módulo de extracción y normalización de contexto

El módulo `swaggerHelper.ts` implementa la **fragmentación del contrato OpenAPI/Swagger**. Su función es convertir una especificación extensa y potencialmente compleja en fragmentos autocontenidos por operación. Separar este paso en un módulo propio permite mantener la lógica de extracción independiente de la generación de pruebas, facilitando su reutilización en futuros escenarios (p.ej., validación del contrato, análisis de cobertura o integración con otros generadores de pruebas).

Módulo de planificación y filtrado

El módulo `generateFeatures.ts` actúa como **orquestador intermedio**. Recibe los fragmentos procesados por el fragmentador y se encarga de filtrar aquellos relevantes (según el controlador indicado) y planificar la creación de los archivos de salida. Esta separación permite mantener el control sobre el flujo de generación sin sobrecargar ni al fragmentador ni al generador de *features*. Además, al integrar una barra de progreso y gestionar rutas de salida, este módulo concentra la lógica de planificación y trazabilidad, sin mezclarse con la lógica de interacción con el modelo de IA.

Módulo de construcción de mensajes y persistencia

El módulo `createFeature.ts` encapsula la responsabilidad de **construir los mensajes a enviar al modelo de IA y procesar la respuesta**. Aquí se combinan el *prompt* base (fijo) con la entrada dinámica (fragmento del contrato y metadatos) y se gestiona la escritura en disco del archivo `.feature`. Al independizar este paso, se logra aislar la lógica relacionada con el ciclo de vida de cada archivo generado. Esto facilita introducir futuras mejoras, como validación del contenido Gherkin o normalización de etiquetas, sin necesidad de modificar el planificador ni el *parser*.

Módulo de integración con proveedores de IA

Finalmente, el módulo `runPromptChatGPT.ts` actúa como una **capa de abstracción hacia la API de OpenAI y otros proveedores compatibles**. Centralizar la interacción con el *endpoint* de *chat completions* permite gestionar posibles errores y respuestas de manera uniforme y añadir soporte a múltiples proveedores (como es el caso de DeepSeek y Gemini) modificando unicamente este componente.

5.2.2. Librerías y dependencias clave

El sistema emplea librerías que cubren diferentes necesidades:

- `openai`: cliente oficial para interactuar con el *endpoint* `/v1/chat/completions` de OpenAI, el cual se encarga de recibir y procesar el *prompt* recibido.
- `@google/genai`: soporte para Gemini, utilizando la capa de compatibilidad con el contrato OpenAI.
- `fs`, `fs/promises`, `path`: módulos estándar de Node.js para E/S de archivos y gestión de rutas.
- `openapi3-ts/oas31`: tipado seguro para manejar objetos de especificaciones OpenAPI.

La combinación de estas librerías permite construir un **flujo de trabajo completo**, que abarca desde la lectura del contrato hasta la persistencia de los escenarios generados.

5.3 Interacción con la API de OpenAI

5.3.1. *Endpoint* de Chat Completions

La herramienta utiliza el ***endpoint* de Chat Completions** de OpenAI (`/v1/chat/completions`), el cual crea una respuesta del modelo en base a los parámetros introducidos en la llamada. Este *endpoint* recibe dos parámetros:

- `messages`: se especifican los diferentes mensajes a enviar al servicio. Estos mensajes deben especificar el rol (`role`) con el que se envía y el contenido del mensaje (`content`). Los roles utilizados son `system` y`user`. El primero se envia con el *prompt* fijo con reglas de formato, restricciones de estilo y estructura esperada de los escenarios. Y el segundo rol se envia con la información dinámica(`schema`, `operationId`, `language`).
- `model`: contiene el modelo que se quiere utilizar para la generación de las definiciones.

5.3.2. Gestión de la respuesta

La respuesta del modelo que devuelve la llamada se recibe con el siguiente formato:


```
{
  "id": "chatcmpl-B9MBs8CjcvOU2jLn4n570S5qMJKcT",
  "object": "chat.completion",
  "created": 1741569952,
  "model": "gpt-4.1-2025-04-14",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "Hello! How can I assist you today?",
        "refusal": null,
        "annotations": []
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 19,
    "completion_tokens": 10,
    "total_tokens": 29,
    "prompt_tokens_details": {
      "cached_tokens": 0,
      "audio_tokens": 0
    },
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "audio_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  },
  "service_tier": "default"
```

```
35 }
```

Listing 5.2: Respuesta a la llamada a OpenAI

De la respuesta proporcionada, el programa recoge (`choices[0].message.content`) y se escribe sin modificaciones en el archivo `.feature` correspondiente.

5.4 Funcionamiento del sistema

El flujo de trabajo puede resumirse en las siguientes fases:

1. El usuario invoca el comando CLI (`node dist/cli.js generate`) con los parámetros deseados.
2. El contrato OpenAPI se divide en fragmentos independientes por operación.
3. Se filtran operaciones según el controlador, y se prepara la ruta de salida para cada operación.
4. Para cada operación:
   - Se construye un *prompt* que combina reglas fijas y contexto dinámico.
   - Se envía una petición al modelo seleccionado mediante `runPrompt`.
   - La respuesta (en Gherkin) se guarda en un archivo `.feature`.
5. Al finalizar, se obtiene una carpeta estructurada de definiciones de prueba.

5.5 Estructura, organización y mantenimiento de los tests

Tras explicar el funcionamiento de la herramienta desarrollada, se va a explicar en detalle la estructura, organización y mantenimiento así como los estándares y convenciones utilizados, así como las diferentes mejoras que se han aplicado a la herramienta desarrollada para la gestión de las pruebas generadas.

5.5.1. Estándares y convenciones utilizadas

Uno de los pilares de la mantenibilidad de las pruebas es la existencia de estándares claros que definan cómo deben escribirse, organizarse y documentarse los escenarios. Sin estas normas, el repositorio tiende a fragmentarse en estilos heterogéneos que dificultan la comprensión por parte de los equipos y comprometen la reutilización de los artefactos. En este caso se han establecido los estándares propios de BDD y de Gherkin. Además, se han definido convenciones específicas para reforzar la uniformidad:

- **Idioma:** se adopta de forma predeterminada el español, aunque se mantiene compatibilidad multilingüe gracias a la directiva `#language`.
- **Estructura de escenarios:** se limita la longitud de las descripciones (10 palabras en títulos y 15 en pasos), lo que evita escenarios excesivamente verbosos.
- **Etiquetado:** cada escenario incorpora un identificador único (`@ID-operationId-NN`), con etiquetas adicionales como `@smoke` para el caso principal y `@error` para los casos negativos.
- **Consistencia de estilo:** se prohíbe el uso de primera persona y de puntuación al final de los pasos.

5.5.2. Organización del repositorio de pruebas

El repositorio de pruebas constituye el núcleo donde se almacenan y gestionan los artefactos necesarios para la definición, ejecución y seguimiento de las pruebas automáticas. En este trabajo, dicho repositorio se encuentra estructurado en tres carpetas principales: `features`, `data` y `results`. Cada una de ellas cumple un rol específico dentro del ciclo de pruebas, lo que facilita la trazabilidad, el mantenimiento y la escalabilidad del sistema de *testing*.

- **Carpeta** `features`: contiene los archivos que describen los escenarios de prueba siguiendo el estándar Gherkin. La organización se realiza por niveles: en el primer nivel se incluyen carpetas con el nombre de la *API*, y dentro de cada una se definen diferentes archivos `.feature`, cada uno correspondiente a un *endpoint*. El nombre de cada archivo sigue la convención `<nombre_endpoint>.feature`.
- **Carpeta** `data`: contiene los datos de prueba necesarios para la ejecución de los escenarios definidos en `features`. Su organización también es jerárquica: en el primer nivel se crean carpetas con el nombre de la *API*, en el segundo nivel cada carpeta corresponde a un *endpoint*, y finalmente en el tercer nivel se almacenan los archivos de datos (principalmente scripts SQL) bajo la convención `createData.sql`.
- **Carpeta** `results`: almacena los informes de ejecución de las pruebas. Este directorio contiene, por un lado, archivos en formato HTML generados automáticamente tras la ejecución de los escenarios (cuyo nombre se define en el archivo de configuración `.yaml`), y por otro, una sub-carpeta donde se guardan los resultados en formato JSON correspondientes a cada prueba individual.

El Listing 5.3 muestra un ejemplo de la estructura de carpetas del repositorio de pruebas utilizado en este trabajo.

```
Repositorio de pruebas
+-- features/
|   +-- API_X/
|   |   +-- endpoint1.feature
|   |   \-- endpoint2.feature
|   \-- API_Y/
|       +-- endpoint1.feature
|       \-- endpoint2.feature
|
+-- data/
|   +-- API_X/
|   |   +-- endpoint1/
|   |   |   \-- createData.sql
|   |   \-- endpoint2/
|   |       \-- createData.sql
|   \-- API_Y/
|       \-- endpoint1/
|           \-- createData.sql
|
\-- results/
    +-- informe1.html
    +-- informe2.html
    \-- json/
```

```
24          +-- resultado1.json
25          \-- resultado2.json
```

Listing 5.3: Estructura de carpetas del repositorio de Wakamiti

5.5.3. Nomenclatura, *fixtures* y estructura base

La coherencia en la nomenclatura y en la estructura de los artefactos de prueba es un elemento clave para que las suites resulten comprensibles y fáciles de mantener. En el caso de pruebas generadas automáticamente, esta necesidad se vuelve aún más relevante, dado que los escenarios no son escritos manualmente por un equipo homogéneo, sino producidos por un modelo de lenguaje que debe ajustarse a convenciones predefinidas.

En lo que respecta a la **nomenclatura** en primer lugar, cada escenario generado incorpora un identificador único siguiendo el patrón `@ID-{operationId}-NN`, donde `{operationId}` corresponde a la operación OpenAPI asociada y `NN` es un contador incremental. Este esquema asegura la trazabilidad entre prueba y contrato técnico. Asimismo, se utilizan etiquetas adicionales que permiten clasificar los escenarios: `@smoke` identifica el caso principal de éxito y `@error` los casos negativos asociados a códigos de respuesta 4xx. Gracias a este sistema, los equipos pueden ejecutar subconjuntos específicos de pruebas en *pipelines* de integración continua o filtrar rápidamente escenarios por su propósito.

Por último, la **estructura base** de cada archivo `.feature` mantiene un formato homogéneo: declaración de la característica, breve descripción funcional, y casos ordenados por prioridad (caso feliz, escenarios de éxito adicionales, validaciones de entradas y casos de error).

5.5.4. Gestión de carpetas y reproducibilidad de artefactos

Con el fin de asegurar **reproducibilidad** e **idempotencia** en la generación de artefactos, la herramienta valida el estado del directorio de salida antes de escribir nuevos ficheros. El comportamiento por defecto es conservador: si el directorio no existe, se crea; si existe y está vacío, se escribe; si existe y contiene archivos, la ejecución se interrumpe con un mensaje informativo, evitando sobre-escrituras accidentales y mezclas con resultados previos. Mediante el uso de la opción `-force` se puede eliminar la carpeta destino antes de generar las definiciones para poder sobre-escribir los archivos.

5.5.5. Buenas prácticas aplicadas en el desarrollo

El análisis realizado en este capítulo ha puesto de manifiesto que la generación automática de pruebas, aunque poderosa, no es suficiente por sí sola para garantizar *suites* fiables y sostenibles. Los estándares de escritura, la organización del repositorio y los mecanismos de mantenimiento constituyen piezas clave, pero su éxito depende en última instancia de la adopción de un conjunto de buenas prácticas que guíen la forma en que se generan, revisan y utilizan dichas pruebas en el día a día del desarrollo.

Una primera buena práctica fundamental es la búsqueda constante de **claridad y simplicidad**. Los escenarios deben ser concisos y comprensibles, evitando descripciones excesivamente técnicas o repetitivas. La finalidad última de una prueba BDD no es solo verificar un comportamiento, sino también servir como **documentación viva** que pueda ser entendida por todos los perfiles del equipo, incluidos los no técnicos. La legibilidad es, por tanto, un valor esencial: cuanto más clara sea una prueba, más fácil será mantenerla y confiar en ella.

Un segundo principio es la **trazabilidad**. Cada escenario de prueba debe estar vinculado de manera inequívoca con un requisito, un contrato de API o una funcionalidad concreta. Esta conexión asegura que las pruebas mantengan su propósito y que no se conviertan en artefactos aislados sin relación directa con el sistema que validan. El uso de identificadores únicos y convenciones consistentes facilita este seguimiento, pero la buena práctica consiste en interiorizar la idea de que toda prueba debe responder a la pregunta: *¿qué requisito estoy validando?*.

Las pruebas generadas automáticamente deben integrarse de manera natural en el **ciclo de vida del software**. Esto implica adoptar un enfoque de *shift-left testing*, ejecutando las validaciones desde fases tempranas del desarrollo, y garantizando que los escenarios se ejecuten de forma sistemática en los *pipelines* de integración continua. De esta manera, las pruebas no solo detectan errores en etapas avanzadas, sino que acompañan la evolución del sistema desde sus primeras iteraciones, reduciendo el coste de corrección y reforzando la calidad.

No obstante, la automatización no elimina la necesidad de supervisión humana por un profesional. Los modelos de lenguaje pueden generar escenarios técnicamente correctos pero funcionalmente irrelevantes, o incluso redundantes. La revisión manual por parte de ingenieros de pruebas, desarrolladores y responsables de producto es indispensable para filtrar, priorizar y ajustar las pruebas generadas. Esta revisión no debe entenderse como una carga, sino como un mecanismo de control de calidad que complementa la capacidad de generación automática.

En pocas palabras, las buenas prácticas aplicadas en el desarrollo de pruebas automáticas con apoyo de inteligencia artificial pueden resumirse en cuatro ejes: **claridad**, **trazabilidad**, **integración en el ciclo de vida** y **supervisión humana**. Estos principios, combinados con una mentalidad de mejora continua, garantizan que las pruebas generadas no se conviertan en simples artefactos desechables, sino en una parte integral y confiable del proceso de desarrollo. Solo de esta manera es posible extraer el máximo valor de la automatización, asegurando que la calidad del software no solo se mantenga, sino que evolucione en paralelo al propio sistema.

5.6 Conclusiones del capítulo

A lo largo de este capítulo se ha descrito en detalle la herramienta desarrollada para la generación automática de pruebas mediante la API de ChatGPT. Se han abordado los requisitos funcionales, la estructura de la herramienta, la interacción con el modelo, la transformación de las respuestas en artefactos de prueba y los mecanismos actuales de manejo de errores, calidad, privacidad y ética.

El análisis ha puesto de manifiesto que, aunque la herramienta se encuentra en una fase inicial, su arquitectura modular y extensible sienta las bases para un sistema capaz de integrarse en flujos de desarrollo reales.

Finalmente, en este capitulo se ha descrito la organización, la estructura que deben tener las pruebas,la forma en la que se deben de guardar las pruebas así como las ampliaciones de la herramienta para su propósito. También, se han visto los diferentes estándares y convenciones utilizadas para las definiciones así como las buenas practicas aplicadas al desarrollo.

CAPÍTULO 6
Evaluación del sistema

La utilidad real de una herramienta de generación automática de pruebas no se demuestra únicamente en su diseño o estructura, sino en su capacidad para producir escenarios de calidad aplicables en un contexto práctico. Por este motivo, este capítulo se dedica a la **evaluación de la herramienta desarrollada**, poniendo a prueba su funcionamiento frente a un caso de estudio concreto.

El objetivo de esta evaluación es analizar la calidad de los escenarios generados, valorar su cobertura funcional y discutir tanto las ventajas como las limitaciones detectadas. De este modo, se pretende determinar hasta qué punto la herramienta puede considerarse útil en entornos reales de desarrollo de software y cuáles son las áreas en las que aún requiere mejoras.

6.1 Objetivo de la evaluación

La evaluación de la herramienta desarrollada debe entenderse en el marco de sus capacidades actuales. A diferencia de un *framework* de pruebas completo, cuyo propósito es ejecutar automáticamente los escenarios y devolver resultados verificables, esta propuesta se centra en la **generación de definiciones en formato Gherkin**, que sirven como punto de partida para la construcción de suites de pruebas más amplias. Por tanto, lo que se analiza en este capítulo no es la ejecución de las pruebas en sí, sino la **calidad, pertinencia y utilidad de las definiciones generadas**.

Este enfoque es especialmente relevante en un contexto en el que las organizaciones buscan optimizar el esfuerzo inicial de redacción de pruebas. La generación automática no elimina la necesidad de implementación ni de validación manual, pero puede **acelerar de forma significativa la fase inicial**, trasladando el esfuerzo de los equipos desde la escritura desde cero hacia la revisión, ajuste e implementación de los escenarios propuestos.

La evaluación persigue, por tanto, responder a tres preguntas principales:

- **¿Cumplen los escenarios generados los estándares formales establecidos?** Esto incluye comprobar la corrección de la sintaxis Gherkin, el uso adecuado de etiquetas y la coherencia en la estructura de cada escenario. Un formato consistente es fundamental para garantizar que las pruebas puedan integrarse sin fricciones en *frameworks* como Wakamiti.
- **¿La cobertura funcional es suficiente?** Se busca determinar si los escenarios recogen los casos más relevantes: tanto los flujos de éxito como los principales errores esperables. Una buena cobertura asegura que las pruebas definidas no se limiten a

un caso ideal, sino que reflejen el abanico real de situaciones que puede encontrar el sistema.

- **¿En qué medida estas definiciones resultan útiles en la práctica?** Más allá de la corrección formal, es importante valorar si los escenarios generados proporcionan una base sólida para el trabajo de los equipos. Una definición clara y completa reduce el tiempo y el esfuerzo necesarios para transformarla en una prueba ejecutable, y actúa como catalizador para la conversación entre desarrolladores, *testers* y *product owners*.

De este modo, el objetivo último de la evaluación es comprobar hasta qué punto la herramienta contribuye a **mejorar la productividad y la calidad del proceso de *testing***. Se trata de identificar las fortalezas actuales (claridad, coherencia, velocidad de generación) y, al mismo tiempo, reconocer las limitaciones que todavía deben resolverse (cobertura incompleta, ausencia de ejemplos de datos, dependencia de revisión humana) para consolidar la solución como un apoyo efectivo en entornos profesionales de desarrollo de software.

6.2 Caso de estudio: PetClinic API

Para evaluar la herramienta se ha seleccionado como caso de estudio la **API PetClinic**, un sistema de ejemplo ampliamente utilizado en el ámbito académico e industrial para la demostración de *frameworks* y metodologías de desarrollo. PetClinic, desarrollado originalmente como una aplicación de referencia para el ecosistema Spring, proporciona un dominio sencillo pero suficientemente representativo: la gestión de mascotas, propietarios, visitas y veterinarios en una clínica veterinaria.

La elección de esta API se justifica por varias razones. En primer lugar, se trata de un sistema conocido y disponible públicamente, lo que garantiza la transparencia y reproducibilidad de la evaluación. En segundo lugar, ofrece una estructura de recursos y operaciones típica de muchas aplicaciones empresariales, incluyendo entidades con relaciones jerárquicas, operaciones CRUD (Create, Read, Update, Delete) y validaciones de seguridad, lo que la convierte en un terreno adecuado para poner a prueba la generación automática de escenarios. Finalmente, su nivel de complejidad es intermedio: suficiente para que la evaluación sea significativa, pero no excesivo hasta el punto de dificultar la interpretación de los resultados.

En este caso, la evaluación se ha centrado en el *endpoint* `GET /api/pets`, cuya finalidad es recuperar la lista de todas las mascotas registradas en el sistema. Este recurso resulta idóneo para el experimento por dos motivos. Por un lado, permite comprobar cómo la herramienta genera escenarios en torno a una operación de consulta con múltiples posibles respuestas: colecciones vacías, colecciones con elementos, errores de autenticación o de permisos, etc. Por otro lado, refleja un patrón común en APIs REST (la recuperación de listas de recursos) que aparece en prácticamente cualquier sistema real, lo que refuerza la generalización de los resultados obtenidos.

La **especificación OpenAPI** de este *endpoint* es la siguiente:

```
paths:
  /api/pets:
    get:
      summary: Obtener lista de mascotas
      operationId: getPets
      tags:
        - Pet
```

```
      responses:
        '200':
          description: Lista de mascotas recuperada correctamente
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Pet'
        '401':
          description: No autenticado
        '403':
          description: Acceso no autorizado
        '404':
          description: Recurso no encontrado
```

Listing 6.1: Especificación OpenAPI de el endpoint GET/api/pets

La herramienta procesó esta especificación y generó automáticamente la **definición de escenarios en formato Gherkin**, que será analizada en detalle en los apartados siguientes. Esta salida constituye la base sobre la que se evaluará la adecuación a los estándares, la cobertura funcional y la utilidad práctica de las pruebas generadas.

6.3 Resultados de la generación automática

Tras procesar la especificación del *endpoint* `GET /api/pets`, la herramienta generó automáticamente la siguiente definición de escenarios en formato Gherkin:

```
#language: es
@definition @get__api_pets
Característica: Obtención de lista de mascotas
  Permite recuperar el conjunto completo de mascotas registradas en
      el sistema

  @ID-get__api_pets-01 @smoke
  Escenario: Éxito al obtener todas las mascotas
    Dado que existe al menos una mascota registrada
    Cuando el cliente solicita todas las mascotas
    Entonces la respuesta tiene código 200
    Y el cuerpo contiene una colección de objetos mascota

  @ID-get__api_pets-02
  Escenario: Éxito con lista vacía
    Dado que no existen mascotas registradas
    Cuando el cliente solicita todas las mascotas
    Entonces la respuesta tiene código 200
    Y el cuerpo contiene una colección vacía

  @ID-get__api_pets-03
  Escenario: Éxito con múltiples mascotas
    Dado que existen varias mascotas registradas
    Cuando el cliente solicita todas las mascotas
    Entonces la respuesta tiene código 200
    Y el cuerpo contiene varios elementos en la colección

  @ID-get__api_pets-04
  Escenario: Solicitud sin cabecera Accept
```

```
29      Dado que la cabecera Accept no está presente
30      Cuando el cliente solicita todas las mascotas
31      Entonces la respuesta tiene código 200
32      Y el cuerpo contiene una colección de mascotas
33
34    @ID-get__api_pets-05 @error
35    Escenario: No autenticado retorna 401
36      Dado que no se envían credenciales válidas
37      Cuando el cliente solicita todas las mascotas
38      Entonces la respuesta tiene código 401
39
40    @ID-get__api_pets-06 @error
41    Escenario: Usuario sin permisos retorna 403
42      Dado que el usuario no tiene permisos de acceso
43      Cuando el cliente solicita todas las mascotas
44      Entonces la respuesta tiene código 403
45
46    @ID-get__api_pets-07 @error
47    Escenario: Ruta no encontrada retorna 404
48      Dado que la ruta de la API no existe
49      Cuando el cliente realiza la petición a una ruta inválida
50      Entonces la respuesta tiene código 404
51
52  # Dudas:
53  # Se aplican parámetros de filtrado o paginación en esta operación?
54  # Cómo se comporta el endpoint si la cabecera Accept tiene un valor
      distinto de application/json?
55  # Qué roles adicionales, si los hay, son necesarios para acceder al
      recurso?
56  # Existe algún límite en el tamaño de la colección de mascotas
     devuelta?
57  # Qué error devuelve el servicio si la base de datos no está
     disponible?
```

Listing 6.2: Definiciones dgeneradas por la herramienta desarrollada

El resultado evidencia varios aspectos relevantes. En primer lugar, se observa que la herramienta genera un conjunto de **siete escenarios diferenciados**, que cubren tanto casos de éxito (lista con resultados, lista vacía, múltiples elementos, ausencia de cabecera) como casos de error (no autenticado, usuario sin permisos, recurso no encontrado). Además, el primer escenario se marca con la etiqueta `@smoke`, lo que permite identificar rápidamente el flujo principal de funcionamiento.

Otro aspecto destacable es la inclusión de una sección final de **dudas abiertas**, en la que el modelo propone preguntas que podrían plantearse a los responsables del producto (por ejemplo, sobre paginación, roles adicionales o manejo de errores internos).

En general, la salida muestra una estructura clara, coherente con las convenciones de Gherkin, y un nivel de cobertura inicial suficiente para servir como base en la construcción de pruebas ejecutables. No obstante, como se analizará en los apartados siguientes, la definición presenta limitaciones propias de un proceso automático, como la ausencia de detalles sobre parámetros de entrada o la necesidad de revisión manual para validar la pertinencia de cada escenario.

6.4 Evaluación de la calidad de los escenarios

La salida generada por la herramienta para el *endpoint* `GET /api/pets` se puede evaluar en tres dimensiones: **adecuación a estándares formales, cobertura funcional y utilidad práctica**.

6.4.1. Adecuación a estándares formales

Los escenarios producidos se ajustan correctamente a la sintaxis de Gherkin. Cada uno presenta la secuencia de pasos típica (*Dado, Cuando, Entonces*) sin errores formales y en un lenguaje claro y comprensible. Además, se respeta la convención de etiquetado establecida: el primer escenario incluye la marca `@smoke` y los casos de error están identificados con `@error`. También se emplea un patrón de identificadores único basado en el `operationId` (`@ID-get__api_pets-NN`), lo que facilita la trazabilidad. En este sentido, la adecuación formal puede considerarse satisfactoria.

6.4.2. Cobertura funcional

La herramienta ha generado siete escenarios que cubren tanto los flujos principales de éxito (lista de mascotas con elementos, lista vacía, múltiples registros, ausencia de cabecera `Accept`) como los errores más habituales (401, 403 y 404). Esto demuestra que el modelo ha capturado tanto la respuesta normal como las condiciones de fallo previstas en la especificación. No obstante, se identifican lagunas: no se contemplan parámetros de filtrado o paginación, ni casos de límite de tamaño en la colección. Estas carencias fueron parcialmente detectadas por el propio modelo, que las incluyó en la sección final de "dudas", lo que refuerza la utilidad de este mecanismo como herramienta de refinamiento.

6.4.3. Utilidad práctica

Desde una perspectiva práctica, los escenarios generados ofrecen una base sólida sobre la que un equipo podría construir pruebas ejecutables. El valor principal radica en que los casos básicos y de error ya están definidos, lo que evita que los equipos tengan que redactarlos desde cero. Sin embargo, la herramienta no incluye ejemplos de datos concretos ni detalles adicionales que permitan ejecutar directamente los escenarios en un *runner* como Wakamiti. Por ello, el esfuerzo no desaparece, sino que se traslada de la escritura inicial a la validación y extensión de las definiciones.

6.4.4. Comparación con el prompt

El análisis es más revelador al contrastar los resultados con el contenido del *prompt*. En primer lugar, se cumple la instrucción de generar un escenario "smoke" y de añadir casos de éxito y de error. Asimismo, se respeta la restricción de no utilizar datos ficticios en los pasos, limitándose a descripciones genéricas. También se observa la correcta aplicación de etiquetas y de la estructura ordenada en la generación de escenarios. No obstante, el *prompt* establecía como requisito generar "tantos escenarios como sea posible hasta alcanzar 20, si la operación lo permite". En este caso, la herramienta generó únicamente siete escenarios, lo que indica una limitación en la capacidad de expansión del modelo. Además, aunque el *prompt* exige contemplar validaciones de entradas válidas e inválidas en mayor detalle, el modelo se limitó a errores genéricos de autenticación y permisos, sin explorar casos de datos mal formados o parámetros ausentes.

6.4.5. Síntesis

Los resultados muestran una **adecuación formal alta**, una **cobertura funcional aceptable pero incompleta** y una **utilidad práctica considerable aunque limitada por la falta de ejemplos ejecutables**. Al compararlos con el *prompt*, se observa que la herramienta cumple los requisitos esenciales (estructura, etiquetas, smoke test, dudas), pero no alcanza plenamente el nivel de exhaustividad solicitado. Esto refuerza la idea de que la herramienta proporciona un punto de partida valioso, pero que la supervisión y revisión humanas siguen siendo indispensables para garantizar la completitud de las pruebas.

6.5 Discusión de resultados

Los resultados obtenidos en la evaluación muestran que la herramienta cumple adecuadamente con su propósito principal: generar definiciones de escenarios en formato Gherkin de forma rápida y con un nivel de calidad suficiente como para servir de base en la construcción de pruebas ejecutables. La salida generada a partir de la especificación del *endpoint* `GET /api/pets` evidencia que el sistema es capaz de cubrir los casos de uso fundamentales, estructurarlos de acuerdo con las convenciones establecidas y aportar valor añadido mediante la inclusión de dudas que fomentan el diálogo entre los distintos actores del proyecto.

En comparación con la redacción manual de pruebas, la principal ventaja radica en el **ahorro de tiempo y esfuerzo en la fase inicial**. Mientras que un equipo de desarrollo debería invertir varias horas en analizar la especificación, identificar casos de éxito y de error, y plasmarlos en un formato BDD coherente, la herramienta es capaz de producir un borrador completo en cuestión de segundos. Este borrador no es perfecto ni definitivo, pero constituye un punto de partida sólido sobre el que trabajar, trasladando el esfuerzo desde la creación ex novo hacia la revisión y ajuste.

La evaluación también confirma que la **claridad y consistencia formal** de los escenarios es elevada: se respetan las estructuras de Gherkin, los pasos son legibles y las etiquetas están correctamente aplicadas. Esto significa que los escenarios pueden integrarse sin dificultad en *frameworks* de ejecución como Wakamiti, siempre que se implementen las definiciones de pasos necesarias.

Sin embargo, también se identifican limitaciones importantes. La cobertura funcional, aunque razonable, no alcanza el nivel de exhaustividad que se esperaba según el *prompt*: en lugar de 20 escenarios, el modelo generó siete, dejando sin cubrir aspectos como parámetros de entrada mal formados, límites de paginación o validaciones de formatos alternativos. Este déficit subraya la necesidad de supervisión humana para completar los huecos y garantizar que las pruebas reflejen de manera fidedigna la lógica de negocio.

En términos prácticos, puede afirmarse que la herramienta **no sustituye al trabajo manual**, pero sí lo **complementa eficazmente**, reduciendo la carga cognitiva y mejorando la productividad. El rol del *tester* o del desarrollador pasa de ser el de un redactor exhaustivo a convertirse en un **validador crítico**, centrado en ajustar los detalles que la generación automática no contempla.

Resumiendo, la discusión de resultados confirma que la herramienta constituye un apoyo valioso en la fase de diseño de pruebas, aportando rapidez, claridad y consistencia, pero también pone de manifiesto que su eficacia depende de una integración adecuada en los procesos de desarrollo y de la participación activa de profesionales que supervisen y completen las definiciones generadas.

6.6 Conclusión del capítulo

La evaluación realizada ha permitido comprobar que la herramienta desarrollada cumple con su objetivo principal: **generar definiciones de escenarios de prueba en formato Gherkin de manera automática** a partir de especificaciones OpenAPI. Los resultados muestran que las definiciones obtenidas son claras, consistentes y útiles como punto de partida para la construcción de pruebas ejecutables, lo que supone un ahorro significativo de esfuerzo en la fase inicial de diseño de test.

Se ha verificado que la herramienta respeta las convenciones establecidas (sintaxis Gherkin, etiquetas, estructura de escenarios) y que ofrece una cobertura adecuada de casos de éxito y error. Asimismo, la inclusión de dudas abiertas refuerza su papel como facilitador en la comunicación entre los distintos perfiles del equipo. Sin embargo, también se ha constatado que la cobertura funcional es incompleta respecto a lo solicitado en el *prompt*, lo que obliga a mantener la supervisión manual como complemento indispensable.

La evaluación confirma que la herramienta constituye un apoyo valioso en el ámbito del *testing* automatizado, reduciendo tiempos de redacción y garantizando consistencia formal, aunque todavía depende de la revisión y extensión humanas para alcanzar un nivel de completitud comparable al de una *suite* de pruebas madura. Este análisis sienta las bases para el siguiente capítulo, en el que se abordarán las implicaciones más amplias de los resultados y se discutirán posibles líneas de mejora y trabajo futuro.

CAPÍTULO 7
Conclusiones

Este capítulo recoge las conclusiones finales del trabajo y constituye el cierre del documento. En él se presentan, en primer lugar, el grado de cumplimiento de los objetivos inicialmente planteados, seguido de un resumen de los resultados y aportaciones más relevantes. A continuación, se exponen las limitaciones detectadas durante el desarrollo del proyecto y se plantean posibles líneas de evolución futura que permitirían ampliar y consolidar la propuesta presentada. Finalmente, se incluye una breve reflexión personal sobre el aprendizaje alcanzado a lo largo del proceso. Con este capítulo se pretende ofrecer una visión global y sintética que sirva de balance final del trabajo realizado y de orientación para futuros desarrollos.

7.1 Grado de cumplimiento de los objetivos

El objetivo general de este Trabajo Fin de Grado consistía en **diseñar y evaluar una herramienta capaz de generar definiciones de pruebas automáticas en formato Gherkin a partir de especificaciones OpenAPI, con el apoyo de modelos de lenguaje de gran tamaño (LLM), y enmarcada en el paradigma del Behaviour Driven Development (BDD)**. Dicho objetivo puede considerarse alcanzado en su totalidad, ya que se ha implementado un prototipo funcional que demuestra la viabilidad técnica de este enfoque y se ha validado mediante un caso de estudio representativo.

En cuanto a los **objetivos específicos**, el grado de cumplimiento puede detallarse de la siguiente manera:

- **Analizar el estado del arte en *testing* automatizado e inteligencia artificial.** Este objetivo se ha cumplido mediante una revisión exhaustiva de herramientas actuales de *testing* convencional (como Dredd o Postman) y de aproximaciones basadas en inteligencia artificial. Esta revisión permitió identificar carencias en el panorama actual (particularmente, la falta de soluciones orientadas a la generación automática de pruebas a partir de especificaciones formales) y, al mismo tiempo, situar la propuesta dentro de la tendencia emergente del *testing* asistido por IA.
- **Describir y documentar la arquitectura de Wakamiti.** La investigación sobre Wakamiti permitió comprender en profundidad su diseño modular y extensible, su enfoque en torno a BDD y su integración con diferentes tipos de pruebas. Este análisis fue esencial, no solo para contextualizar la herramienta desarrollada, sino también para justificar que las definiciones generadas fueran compatibles con *frameworks* existentes. El cumplimiento de este objetivo contribuyó a garantizar que la solución propuesta no fuera un ejercicio aislado, sino que pudiera incorporarse de forma coherente en ecosistemas de testing ya consolidados.

- **Diseñar un prototipo capaz de generar definiciones de escenarios en Gherkin.** Este objetivo se alcanzó con éxito mediante la implementación de un sistema que toma como entrada fragmentos de especificaciones OpenAPI y devuelve un archivo `.feature` en Gherkin. Se verificó que la salida cumple las convenciones de nomenclatura, etiquetado y estructura establecidas en el trabajo, lo que demuestra la validez de la aproximación. Este cumplimiento supone la principal contribución técnica del TFG.

- **Evaluar la herramienta en un caso de estudio.** La evaluación, centrada en la API PetClinic, puso de manifiesto que la herramienta genera escenarios con cobertura aceptable, legibles y consistentes. Aunque no se alcanzó el número de escenarios máximos que sugería el *prompt* (20), se consiguió un conjunto suficientemente representativo de casos de éxito y error aunque no excesivo en pruebas. Estos escenarios cubren los casos mas generales (como pueden ser pruebas smoke, pruebas de seguridad (códigos HTTP 401, 403...)). La inclusión de dudas como parte de la salida reforzó la utilidad del sistema al promover la discusión con los responsables de producto y en un futuro se podría conseguir resolver esas dudas ya que de momento no se puede mantener una conversación con el modelo debido a que el funcionamiento es en base al *prompt* fijo que se le pasa al modelo. De este modo, este objetivo puede considerarse satisfecho, aunque con margen de mejora en términos de exhaustividad.

En conjunto, puede afirmarse que el grado de cumplimiento de los objetivos planteados en la fase inicial del trabajo ha sido **muy alto**. La única diferencia respecto al planteamiento original radica en que la herramienta, en su estado actual, se limita a la **generación de definiciones de prueba** y no aborda la creación de implementaciones ejecutables. Este aspecto constituye una limitación reconocida, pero no impide considerar alcanzado el objetivo central del TFG: demostrar la viabilidad y el valor práctico de la generación automática de pruebas asistida por inteligencia artificial.

7.2 Resultados y aportaciones más relevantes

Los resultados obtenidos a lo largo del trabajo permiten extraer varias aportaciones significativas. En primer lugar, se ha logrado **desarrollar una herramienta funcional capaz de transformar especificaciones OpenAPI en definiciones de escenarios de prueba en formato Gherkin**, aplicando modelos de lenguaje de gran tamaño (LLM) como núcleo del proceso de generación. Este resultado constituye una innovación clara, dado que en la actualidad existen pocas herramientas que automaticen este proceso de forma tan directa y estandarizada.

En segundo lugar, la herramienta ha sido **evaluada en un caso de estudio representativo** (la API PetClinic), lo que ha permitido comprobar en un entorno realista que los escenarios generados cumplen con los criterios de claridad, coherencia formal y cobertura básica de casos de éxito y error. Esta validación práctica confirma la viabilidad de la aproximación y demuestra que la herramienta es capaz de reducir el esfuerzo inicial de redacción manual de pruebas.

Otra aportación destacable es la **formalización de un marco de estándares y convenciones** para la generación de escenarios. A lo largo del trabajo se ha definido un conjunto de reglas de nomenclatura, etiquetado y estructura que no solo garantizan la legibilidad de las pruebas, sino que también facilitan su integración en *frameworks* como Wakamiti. Este marco de buenas prácticas constituye en sí mismo un valor añadido, pues asegura

que las pruebas generadas no sean meros artefactos aislados, sino componentes consistentes de una suite mantenible.

Desde una perspectiva académica y profesional, el trabajo aporta evidencia de la **viabilidad de aplicar técnicas de inteligencia artificial al ámbito del *testing* de software**, un área tradicionalmente muy dependiente del esfuerzo humano. Al reducir la carga de redacción inicial y estandarizar la documentación de escenarios, se contribuye a mejorar tanto la productividad de los equipos como la calidad de los procesos de verificación.

En síntesis, los resultados más relevantes del trabajo pueden resumirse en tres aportaciones principales:

1. La creación de una herramienta innovadora para la generación automática de pruebas en formato BDD.
2. La validación de su utilidad en un caso práctico realista.
3. El establecimiento de un marco de estándares y buenas prácticas que refuerza la calidad y sostenibilidad de las pruebas generadas.

7.3 Limitaciones identificadas

Aunque los resultados del trabajo son satisfactorios y demuestran la viabilidad de la propuesta, la evaluación también ha permitido identificar una serie de limitaciones que condicionan el alcance actual de la herramienta:

- **Generación de definiciones, no implementaciones.** La herramienta produce únicamente escenarios en formato Gherkin, pero no genera las implementaciones necesarias para su ejecución en un *runner* como Wakamiti. Esto obliga a que los equipos de desarrollo deban completar manualmente las *step definitions* antes de poder ejecutar las pruebas.
- **Cobertura incompleta.** Si bien la herramienta genera casos de éxito y de error representativos, no alcanza la exhaustividad esperada en algunos ámbitos. Aspectos como la validación de entradas inválidas, la paginación de resultados, los valores límite o las pruebas de rendimiento no son contemplados en la salida inicial.
- **Dependencia de la supervisión humana.** Los escenarios generados no siempre cubren todos los casos relevantes, ni garantizan la ausencia de redundancias. Por este motivo, la revisión manual sigue siendo imprescindible para asegurar la completitud y pertinencia de las pruebas.
- **Uso de un modelo externo.** El sistema depende de un proveedor externo de modelos de lenguaje. Esto plantea cuestiones relacionadas con la privacidad de los datos empleados, el coste de uso en escenarios intensivos y la posibilidad de sesgos en las salidas generadas.
- **Limitación del número de escenarios generados.** Aunque el *prompt* indicaba la posibilidad de generar hasta 20 escenarios por operación, en la práctica el modelo produjo un número más reducido (siete en el caso evaluado). Esto evidencia una limitación en la capacidad de expansión del modelo que afecta a la cobertura funcional.

Estas limitaciones no invalidan los resultados obtenidos, pero sí marcan las fronteras del alcance actual del trabajo. Constituyen, además, oportunidades claras para orientar las líneas de mejora y evolución futura de la herramienta.

7.4 Posibles líneas de trabajo futuro

Las limitaciones identificadas durante el desarrollo del proyecto permiten delinear varias líneas de evolución que podrían reforzar la utilidad y el alcance de la herramienta:

- **Generación de implementaciones ejecutables.** Una primera línea de mejora consistiría en ampliar la herramienta para que, además de generar definiciones en Gherkin, pueda proponer o incluso generar automáticamente *step definitions* compatibles con *frameworks* como Wakamiti. Esto permitiría pasar de simples descripciones a pruebas directamente ejecutables, reduciendo aún más la intervención manual.
- **Ampliación de la cobertura funcional.** Otra mejora clave consistiría en incrementar el nivel de cobertura de los escenarios generados, incluyendo casos como validaciones de entradas inválidas, parámetros omitidos, pruebas de paginación, valores límite o validaciones de rendimiento. De esta forma, la herramienta abarcaría un espectro más amplio de situaciones.
- **Validación automática de escenarios.** Se podría integrar un mecanismo de validación interna que analizara la sintaxis y consistencia de los escenarios generados, detectando redundancias, pasos incompletos o errores de formato antes de guardar el resultado.
- **Optimización del proceso de generación.** Una línea de evolución natural es la mejora de los *prompts* utilizados, incorporando técnicas de *prompt engineering* o incluso enfoques *few-shot learning*, que permitan guiar al modelo hacia resultados más ricos y completos.
- **Exploración de modelos *open-source*.** Para reducir la dependencia de proveedores externos, sería interesante experimentar con modelos de lenguaje de código abierto que puedan ejecutarse en entornos locales o privados. Esto mitigaría riesgos de privacidad y reduciría los costes de uso en entornos productivos.

Estas líneas de trabajo no solo permitirían consolidar los resultados obtenidos, sino también situar la herramienta en un nivel de madurez superior, acercándola a su integración real en procesos de desarrollo de software industriales.

7.5 Valor personal y formativo

El desarrollo de este Trabajo Fin de Grado ha supuesto una experiencia enriquecedora tanto en el ámbito de la **computación** como en el de la **ingeniería del software**, favoreciendo un aprendizaje combinado que resulta especialmente relevante en el contexto actual de la disciplina.

Desde la perspectiva de la **computación**, el proyecto ha permitido profundizar en el funcionamiento de los **modelos de lenguaje de gran tamaño (LLM)** y en su integración mediante APIs, comprendiendo sus capacidades, limitaciones y el papel que desempeñan en la automatización de tareas complejas. Esta experiencia ha facilitado la adquisición de competencias en el diseño de *prompts*, la extracción de contexto y la adaptación de los resultados generados a un dominio tan específico como el del *testing* de software.

Por otro lado, desde la **ingeniería del software**, el trabajo ha reforzado los conocimientos relacionados con el ***testing* automatizado**, las metodologías ágiles y, en particular, el enfoque **Behaviour Driven Development (BDD)**. La necesidad de garantizar

que las pruebas generadas fueran legibles, coherentes y mantenibles ha implicado aplicar principios propios de la calidad del software, como la estandarización, la trazabilidad y la integración en procesos de desarrollo reales.

El valor añadido de este proyecto reside precisamente en la **intersección de ambos ámbitos**: aprovechar técnicas avanzadas de computación (IA, procesamiento del lenguaje natural) para abordar un reto clásico de la ingeniería del software (el elevado coste y complejidad del *testing*). Este aprendizaje híbrido ha permitido consolidar una visión más completa y transversal, en la que la innovación tecnológica se combina con la aplicación rigurosa de principios de ingeniería.

Además de los conocimientos técnicos, el proyecto ha favorecido el desarrollo de competencias personales y profesionales como la **capacidad de análisis crítico, la planificación y gestión del tiempo, y la comunicación técnica**. En conjunto, la experiencia ha supuesto un paso importante en la formación como ingeniero informático, contribuyendo a integrar de manera práctica los conocimientos adquiridos en diferentes áreas del grado.

7.6 Síntesis final

En conjunto, este Trabajo Fin de Grado ha demostrado la **viabilidad de aplicar modelos de lenguaje al ámbito del *testing* automatizado**, aportando una herramienta capaz de generar de forma automática definiciones de escenarios en formato Gherkin a partir de especificaciones OpenAPI. Los resultados obtenidos confirman que la propuesta reduce de manera significativa el esfuerzo inicial de redacción de pruebas, garantiza consistencia formal y fomenta la comunicación entre los distintos perfiles del equipo, en línea con los principios del BDD.

No obstante, también se han identificado limitaciones que condicionan su alcance actual, como la ausencia de implementaciones ejecutables, la cobertura incompleta en escenarios complejos y la dependencia de la supervisión humana. Estas restricciones, lejos de restar valor al trabajo, se convierten en oportunidades claras para guiar futuras líneas de investigación y desarrollo.

Las aportaciones del proyecto pueden resumirse en tres ejes principales: la creación de una herramienta innovadora de generación de pruebas, la validación de su utilidad mediante un caso de estudio realista y la formalización de un marco de convenciones y buenas prácticas aplicables al *testing* automatizado. Desde una perspectiva académica y profesional, el trabajo refuerza la idea de que la integración de técnicas de inteligencia artificial en la ingeniería del software no sustituye a la labor humana, pero sí constituye un apoyo valioso para aumentar la productividad y mejorar la calidad.

Como conclusión global, el TFG ha cumplido sus objetivos, ha abierto nuevas vías de aplicación de la inteligencia artificial en el desarrollo de software y ha aportado tanto en el plano técnico como en el formativo. Con ello, se sientan las bases para la evolución hacia sistemas de *testing* más inteligentes, adaptativos y sostenibles, en los que la colaboración entre computación e ingeniería del software seguirá siendo la clave.

Bibliografía

[1] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34:1:135–137, 2001. Consultado en https://ieeexplore.ieee.org/document/962984.

[2] Maurice Dawson, Darrell N. Burrell, Emad Rahim and Stephen Brewster. Integrating Software Assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Technology and Planning*, 3:6:49–53, 2010. Consultado en https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC.

[3] Mary Jean Harrold. Testing: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering*, 61–72, 2000. ACM. Consultado en https://dl.acm.org/doi/10.1145/336512.336532.

[4] Wasif Afzal, Snehal Alone, Kerstin Glocksien and Richard Torkar. Software Test Process Improvement Approaches: A Systematic Literature Review and an Industrial Case Study. *Journal of Systems and Software*, 1–33, September, 2015. Consultado en http://www.es.mdu.se/publications/3997-.

[5] V. Garousi, Michael Felderer, Marco Kuhrmann, Kadir Herkiloğlu and Sigrid Eldh. Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process*, 32, February, 2020. Consultado en https://www.researchgate.net/publication/339095645_Exploring_the_industry's_challenges_in_software_testing_An_empirical_study.

[6] Lester Lobo and James D. Arthur. Local and Global Analysis: Complementary Activities for Increasing the Effectiveness of Requirements Verification and Validation. *arXiv preprint cs/0503002*, 2005. Consultado en https://arxiv.org/abs/cs/0503002.

[7] Maximiliano A. Mascheroni and Emanuel Irrazábal. Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review. *Computación y Sistemas*, 2025. Consultado en https://www.scielo.org.mx/scielo.php?pid=S1405-55462018000301009&script=sci_arttext.

[8] ISO/IEC/IEEE. ISO/IEC/IEEE 29119-1:2022 – Software and systems engineering – Software testing – Part 1: General concepts. 2022. Consultado en https://www.iso.org/standard/81291.html.

[9] Yuqing Wang, Mika Mäntylä, Zihao Liu and Jouni Markkula. Test Automation Maturity Improves Product Quality — Quantitative Study of Open Source Projects Using Continuous Integration. *arXiv preprint arXiv:2202.04068*, 2022. Consultado en https://arxiv.org/abs/2202.04068.

[10] Mubarak Albarka Umar. Comprehensive Study of Software Testing: Categories, Levels, Techniques, and Types. *International Journal of Advanced Research, Ideas and Innovations in Technology*, 2020. Consultado en https://www.researchgate.net/publication/342579919_A_Study_of_Software_Testing_Categories_Levels_Techniques_and_Types.

[11] Srinivas Nidhra and Jagruthi Dondeti. Black Box and White Box Testing Techniques – A Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2:2:29–32, 2012. Consultado en https://airccse.org/journal/ijesa/papers/2212ijesa04.pdf.

[12] Nickolay Bakharev. Black Box Testing: Types, Techniques, Pros and Cons. 2023. Consultado en https://brightsec.com/blog/black-box-testing-types-techniques-pros-and-cons.

[13] Glenford J. Myers, Corey Sandler and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[14] Divya Kumar and Krishn Mishra. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*, 79:8–15, December, 2016. Consultado en https://www.sciencedirect.com/science/article/pii/S1877050916001277.

[15] International Organization for Standardization. ISO/IEC 25010:2011 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. 2011. Consultado en https://www.iso.org/standard/35733.html.

[16] Khin Shin Thant and Hlaing Htake Khaung Tin. The Impact of Manual and Automatic Testing on Software Testing Efficiency and Effectiveness. *Indian Journal of Science and Research*, 3:3:88–93, 2023. Consultado en https://www.researchgate.net/publication/370526552_THE_IMPACT_OF_MANUAL_AND_AUTOMATIC_TESTING_ON_SOFTWARE_TESTING_EFFICIENCY_AND_EFFECTIVENESS.

[17] Mojtaba Shahin, Muhammad Ali Babar and Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017. Consultado en https://doi.org/10.1109/ACCESS.2017.2685629.

[18] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). *Future of Software Engineering, FOSE 2014 - Proceedings*, May, 2014. Consultado en https://courses.cs.washington.edu/courses/csep503/19wi/schedule/papers/SoftwareTestingTravelogue.pdf.

[19] Vahid Garousi and Mika V. Mäntylä. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, 76:92–117, 2016. Consultado en https://www.sciencedirect.com/science/article/pii/S0950584916300702.

[20] Jussi Kasurinen, Ossi Taipale and Kari Smolander. Software Test Automation in Practice: Empirical Observations. *Advances in Software Engineering*, 2010. Consultado en https://doi.org/10.1155/2010/620836.

[21] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng and Neel Sundaresan. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv:2009.05617*, 2021. Consultado en https://arxiv.org/abs/2009.05617.

[22] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh and Michel C. Desmarais. Effective test generation using pre-trained Large Language Models and mutation testing. *Information and Software Technology*, 171:107468, 2024. Consultado en https://www.sciencedirect.com/science/article/pii/S0950584924000739.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. Attention Is All You Need. *Advances in Neural Information Processing Systems (NeurIPS)*, 5998–6008, 2017. Consultado en https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[24] Alec Radford, Karthik Narasimhan, Tim Salimans and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. OpenAI, Technical Report, 2018. Consultado en https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[25] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. OpenAI, Technical Report, 2019. Consultado en https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[26] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. Consultado en https://arxiv.org/abs/2005.14165.

[27] OpenAI. GPT-4 Technical Report. OpenAI, Technical Report, 2023. Consultado en https://arxiv.org/abs/2303.08774.

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 4171–4186, 2019. Association for Computational Linguistics. Consultado en https://aclanthology.org/N19-1423.

[29] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego R. C. Dias and Marcelo P. Guimarães. Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Reliability*, 68:3:1189–1212, 2019. Consultado en https://ieeexplore.ieee.org/document/8638573.

[30] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang and Qing Wang. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, 50:4:911–936, 2024. Consultado en https://doi.org/10.1109/TSE.2024.3368208.

[31] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes and Yejin Choi. The Curious Case of Neural Text Degeneration. *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020. Consultado en https://arxiv.org/abs/1904.09751.

[32] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill et al. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021. Consultado en https://arxiv.org/abs/2108.07258.

[33] Prasanna Rasal. Usage Of Machine Learning Algorithms In Software Testing. 2022. Consultado en https://www.researchgate.net/publication/371577007_Usage_Of_Machine_Learning_Algorithms_In_Software_Testing.

[34] Afonso Fontes and Gregory Gay. The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study. *arXiv preprint arXiv:2206.10210*, 2022. Consultado en https://arxiv.org/abs/2206.10210.

[35] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 416–419, 2011. Association for Computing Machinery. Consultado en https://dl.acm.org/doi/10.1145/2025113.2025179.

[36] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22:2:67–120, 2012. Consultado en https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.430.

[37] Helge Spieker, Arnaud Gotlieb, Dusica Marijan and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*, 12–22, 2017. Association for Computing Machinery. Consultado en https://doi.org/10.1145/3092703.3092709.

[38] Dusica Marijan, Arnaud Gotlieb and Sagar Sen. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, 540–543, 2013. Consultado en https://doi.org/10.1109/ICSM.2013.91.

[39] Soneya Binta Hossain and Matthew Dwyer. TOGLL: Correct and Strong Test Oracle Generation with LLMs. *arXiv preprint arXiv:2405.03786*, 2024. Consultado en https://arxiv.org/abs/2405.03786.

[40] Dan North. Introducing BDD. 2006. Consultado en https://dannorth.net/blog/introducing-bdd/.

[41] Matt Wynne and Aslak Hellesøy. *The Cucumber Book: Behaviour-driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012. Consultado en https://books.google.es/books?id=oMswygAACAAJ.

[42] Magnus Härlin. *Testing and Gherkin in Agile Projects*. Master's Thesis, Linköping University, 2016. Consultado en https://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-125772.

[43] Selenium. Selenium Official Website. 2025. Consultado en https://www.selenium.dev/.

[44] JUnit. JUnit Official Website. 2017. Consultado en https://junit.org/junit5/.

[45] xUnit. xUnit Official Website. 2007. Consultado en https://xunit.net/.

[46] Cypress. Cypress Official Website. 2015. Consultado en https://www.cypress.io/.

[47] Postman. Postman Official Website. 2012. Consultado en https://www.postman.com/.

[48] Newman. Newman doc. on npm. 2014. Consultado en https://www.npmjs.com/package/newman.

[49] Dredd. Dredd Official Website. 2013. Consultado en https://dredd.org/en/latest/.

[50] Qodo. Qodo Official Website. 2023. Consultado en https://www.qodo.ai.

[51] TestRigor. TestRigor Official Website. 2023. Consultado en https://testrigor.com.

[52] Functionize. Functionize Official Website. 2023. Consultado en https://www.functionize.com.

[53] Mabl. Mabl Official Website. 2023. Consultado en https://www.mabl.com.

[54] Katalon. Katalon Official Website. 2023. Consultado en https://katalon.com.

[55] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi and Graham Neubig. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Computing Surveys*, 55:9:1–35, 2023. Consultado en https://dl.acm.org/doi/10.1145/3560815.

[56] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith and Douglas C. Schmidt. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *arXiv preprint arXiv:2302.11382*, 2023. Consultado en https://arxiv.org/abs/2302.11382.

[57] OpenAI. ChatGPT API. 2025. Consultado en https://platform.openai.com/docs/overview.

[58] Michael Ellims, J. Bridges and D.C. Ince. Unit testing in practice. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 3–13, December, 2004. Consultado en https://doi.org/10.1109/ISSRE.2004.44.

Anexo: Contribución del trabajo a los Objetivos de Desarrollo Sostenible (ODS)

Los **Objetivos de Desarrollo Sostenible (ODS)** forman parte de la Agenda 2030 de Naciones Unidas, aprobada en 2015. Su finalidad es establecer un marco común de actuación para gobiernos, empresas y sociedad civil con el fin de afrontar los grandes retos globales: erradicación de la pobreza, protección del planeta, acceso universal a la educación y a la salud, reducción de desigualdades y fomento de la innovación, entre otros.

Los ODS son un conjunto de **17 objetivos** y 169 metas específicas, interconectados entre sí, que buscan garantizar un desarrollo sostenible en sus tres dimensiones: social, económica y medioambiental. Su existencia responde a la necesidad de coordinar esfuerzos internacionales en torno a desafíos que no pueden resolverse de manera aislada ni unilateral.

Lista de los 17 ODS

1. **Fin de la pobreza:** Erradicar la pobreza en todas sus formas y dimensiones, garantizando acceso a recursos básicos y protección social.
2. **Hambre cero:** Poner fin al hambre, lograr la seguridad alimentaria, mejorar la nutrición y promover una agricultura sostenible.
3. **Salud y bienestar:** Garantizar una vida sana y promover el bienestar en todas las edades, asegurando acceso universal a la salud.
4. **Educación de calidad:** Garantizar una educación inclusiva, equitativa y de calidad, y promover oportunidades de aprendizaje permanente para todos.
5. **Igualdad de género:** Lograr la igualdad entre géneros y empoderar a todas las mujeres y niñas.
6. **Agua limpia y saneamiento:** Garantizar la disponibilidad de agua y su gestión sostenible, así como el acceso al saneamiento para todos.
7. **Energía asequible y no contaminante:** Asegurar el acceso universal a energía asequible, fiable, sostenible y moderna.
8. **Trabajo decente y crecimiento económico:** Promover el crecimiento económico sostenido, inclusivo y sostenible, con empleo pleno y productivo y trabajo decente para todos.

9. **Industria, innovación e infraestructura:** Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación.
10. **Reducción de las desigualdades:** Reducir la desigualdad dentro y entre los países, promoviendo la inclusión social, económica y política.
11. **Ciudades y comunidades sostenibles:** Lograr que las ciudades y asentamientos humanos sean inclusivos, seguros, resilientes y sostenibles.
12. **Producción y consumo responsables:** Garantizar modalidades de consumo y producción sostenibles, fomentando el uso eficiente de recursos.
13. **Acción por el clima:** Adoptar medidas urgentes para combatir el cambio climático y sus impactos.
14. **Vida submarina:** Conservar y utilizar sosteniblemente los océanos, mares y recursos marinos.
15. **Vida de ecosistemas terrestres:** Gestionar sosteniblemente los bosques, combatir la desertificación, detener la degradación de la tierra y la pérdida de biodiversidad.
16. **Paz, justicia e instituciones sólidas:** Promover sociedades pacíficas e inclusivas, facilitar acceso a la justicia y crear instituciones eficaces y responsables.
17. **Alianzas para lograr los objetivos:** Revitalizar la alianza mundial para el desarrollo sostenible, reforzando la cooperación internacional.

Tabla de relación con el TFG

ODS	Descripción	Relación con el trabajo
1	Fin de la pobreza	
2	Hambre cero	
3	Salud y bienestar	
4	Educación de calidad	X
5	Igualdad de género	
6	Agua limpia y saneamiento	
7	Energía asequible y no contaminante	
8	Trabajo decente y crecimiento económico	X
9	Industria, innovación e infraestructura	X
10	Reducción de las desigualdades	
11	Ciudades y comunidades sostenibles	
12	Producción y consumo responsables	X
13	Acción por el clima	
14	Vida submarina	
15	Vida de ecosistemas terrestres	
16	Paz, justicia e instituciones sólidas	
17	Alianzas para lograr los objetivos	

Tabla 1: Relación de los 17 ODS con el presente trabajo.

Justificación de los ODS seleccionados

En este trabajo se han identificado como relevantes los ODS **4, 8, 9 y 12**.

El **ODS 4 (Educación de calidad)** se refleja en la medida en que la herramienta fomenta el aprendizaje de buenas prácticas de *testing* automatizado y del uso de la inteligencia artificial aplicada al desarrollo de software, contribuyendo a la capacitación técnica de estudiantes y profesionales.

El **ODS 8 (Trabajo decente y crecimiento económico)** está presente porque la automatización de pruebas reduce costes y tiempos de desarrollo, favoreciendo entornos laborales más eficientes y sostenibles, en los que los equipos pueden dedicar más tiempo a tareas de mayor valor añadido.

El **ODS 9 (Industria, innovación e infraestructura)** se materializa en la innovación tecnológica que supone integrar modelos de lenguaje en la generación de pruebas, impulsando metodologías más modernas y robustas dentro de la industria del software.

El **ODS 12 (Producción y consumo responsables)** se vincula con el uso eficiente de recursos en el desarrollo de software, evitando la redundancia de pruebas manuales y optimizando la eficiencia en la validación de sistemas.