Automatización de pruebas de software basadas en propiedades mediante ingeniería de prompts e inteligencia artificial

Property-based software testing automation through prompt engineering and artificial intelligence

Ricardo Rafael Quintero Meza
Instituto Tecnológico de Culiacán, Culiacán, México
ricardo.qm@culiacan.tecnm.mx

Erven Germán Gil García
Instituto Tecnológico de Culiacán, Culiacán, México
erven.gg@culiacan.tecnm.mx

https://doi.org/10.36825/RITI.13.31.005

Recibido: Junio 22, 2025
Aceptado: Septiembre 30, 2025

Resumen: La creciente complejidad del desarrollo de software moderno exige metodologías de prueba más eficientes, completas y adaptativas, que garanticen la fiabilidad, robustez y calidad de las aplicaciones, al tiempo que optimicen los costos y tiempos asociados con el mantenimiento y evolución del sistema. Aunque los enfoques de prueba tradicionales siguen siendo ampliamente utilizados, presentan limitaciones en cobertura, escalabilidad y adaptabilidad, especialmente frente a sistemas dinámicos y en constante evolución. En este contexto, la integración de modelos de lenguaje de gran tamaño (LLMs) mediante técnicas de ingeniería de *prompts* surge como una alternativa prometedora e innovadora para la automatización, mejora y ampliación de los procesos de prueba de software. Este trabajo presenta una herramienta que combina las capacidades generativas de los LLMs, accesibles mediante APIs especializadas, con el rigor de las pruebas basadas en propiedades (PBT). Esta sinergia permite la generación automática de propiedades de prueba y la validación inteligente del código, facilitando la detección temprana de errores y contribuyendo al desarrollo de software más robusto y confiable desde las etapas iniciales del ciclo de vida. A través de la ingeniería de *prompts*, la herramienta guía la formulación precisa de propiedades de prueba y orquesta la generación de datos diversos y relevantes. Este enfoque busca superar las limitaciones de las metodologías tradicionales, mejorando la cobertura de pruebas, reduciendo el esfuerzo manual y aumentando la escalabilidad. El resultado es un proceso de verificación más optimizado que promueve mayores estándares de calidad y confiabilidad del software. Esta propuesta representa un avance hacia la automatización inteligente de pruebas, integrando la inteligencia artificial con metodologías formales de validación y abriendo nuevas posibilidades para su aplicación en la ingeniería de software.

Palabras clave: *Prompt, Propiedades, Prueba, IA, LLM.*

Abstract: The increasing complexity of modern software development demands more efficient, comprehensive, and adaptive testing methodologies that ensure the reliability, robustness, and quality of applications, while also optimizing the costs and time associated with system maintenance and evolution. Although traditional testing approaches remain widely used, they present limitations in terms of coverage, scalability, and adaptability,

especially when dealing with dynamic and constantly evolving systems. In this context, the integration of large language models (LLMs) through prompt engineering techniques emerges as a promising and innovative alternative for automating, enhancing, and expanding software testing processes. This work presents a tool that combines the generative capabilities of LLMs, accessible through specialized APIs, with the rigor of property-based testing (PBT). This synergy enables the automatic generation of test properties and the intelligent validation of code, facilitating early error detection and contributing to the development of more robust and reliable software from the early stages of the development lifecycle. Through prompt engineering, the tool guides the precise formulation of test properties and orchestrates the generation of diverse and relevant data. This approach aims to overcome the limitations of traditional methodologies by improving test coverage, reducing manual effort, and increasing scalability. The result is a more optimized verification process that promotes higher standards of software quality and reliability. This proposal represents a step forward in the intelligent automation of testing, integrating artificial intelligence with formal validation methodologies and opening new possibilities for its application in software engineering.

Keywords: *Prompt, Properties, Test, IA, LLM.*

1. Introducción

La industria en general se encuentra siempre en transición tratando de generar nuevos productos y servicios, que según el tipo de mercado son requeridos por los clientes. Desde el punto de vista de software, los requerimientos son uno de los puntos iniciales para el desarrollo, pues imponen las pautas a cumplir con el funcionamiento del mismo, pero que cumpla con el funcionamiento de manera temporal no es suficiente; el producto debe tener la versatilidad para futuras modificaciones, requerimientos o añadiduras que sean necesarias y es ahí donde la calidad del software tiene que ser evaluada.

La calidad del software no debe entenderse como una etapa aislada del ciclo de vida, sino como un aspecto transversal que se extiende desde el análisis de viabilidad hasta el despliegue, pasando por el mantenimiento y la evolución del sistema [1]. Dentro de las estrategias para garantizar esta calidad, las pruebas de software desempeñan un papel fundamental. Estas consisten en la ejecución de un programa bajo distintas condiciones de entrada para verificar su comportamiento esperado. Por lo tanto, una prueba tiene un resultado negativo si un error es detectado durante las ejecuciones, y el resultado es positivo si en el conjunto de prueba no se tiene un error [2]. La creciente complejidad de los sistemas de software modernos exige metodologías de prueba más eficientes, exhaustivas y adaptativas que aseguren la confiabilidad, la robustez y la calidad de las aplicaciones, al tiempo que optimizan los costos y el tiempo asociados con el mantenimiento y la evolución del sistema [3]. Aunque los enfoques de prueba tradicionales siguen siendo ampliamente utilizados, presentan limitaciones significativas en términos de cobertura, escalabilidad y adaptabilidad, especialmente ante sistemas dinámicos y en constante evolución [4].

Entre los enfoques más prometedores en los últimos años se encuentran las pruebas basadas en propiedades (*Property-Based Testing*, PBT), que permiten una validación más exhaustiva del comportamiento del software mediante la definición de propiedades generales que deben cumplirse en todos los casos, facilitando así la detección temprana de errores y la automatización de los casos de prueba. A pesar de las ventajas de las PBT, la dificultad para definir propiedades significativas a menudo dificulta su adopción más amplia. Este trabajo explora un nuevo enfoque que aprovecha las capacidades de los Modelos de Lenguaje de gran tamaño (LLMs, *Large Language Models*) y la ingeniería de *prompts* para ayudar en la creación de propiedades para Pruebas Basadas en Propiedades, específicamente dentro del ecosistema de Python utilizando la biblioteca Hypothesis. Paralelamente, los Modelos LLMs están transformando diversas áreas del desarrollo de software.

Entrenados con enormes volúmenes de texto, estos modelos son capaces de generar código, documentar funciones, sugerir mejoras e incluso asistir en la creación de pruebas automatizadas [5]. La interacción con los LLMs se realiza a través de *prompts* instrucciones en lenguaje natural cuya formulación precisa y contextualizada resulta clave para obtener resultados útiles. Este proceso ha dado origen a la disciplina de la ingeniería de *prompts*, centrada en la optimización de estas instrucciones para maximizar la efectividad de los modelos [6]. Para abordar este desafío, presentamos una extensión de Visual Studio Code que utiliza la API Mistral AI e interactúa con la biblioteca Hypothesis. Esta herramienta permite a los desarrolladores utilizar *prompts* en lenguaje natural para guiar la generación de propiedades comprobables, lo que reduce la barrera de entrada a las PBT y mejora la calidad

del software. Esta obra contribuye con una cadena de herramientas práctica que integra capacidades de IA de vanguardia con metodologías de prueba sólidas, ofreciendo una forma más intuitiva y eficiente de garantizar la confiabilidad y la corrección del software.

Como resultado, la combinación entre la ingeniería de *prompts* y las pruebas basadas en propiedades se presenta como una propuesta innovadora para mejorar la calidad del software. Esta integración no solo permite automatizar etapas clave del proceso de validación, sino que también reduce significativamente la intervención manual, facilita la detección temprana de errores y contribuye a un desarrollo más ágil, confiable y alineado con las nuevas capacidades de la inteligencia artificial.

2. Estado del arte

2.1. Ingeniería de prompts

Un *prompt* es la instrucción de texto que los usuarios deben introducir cuando interactúan con un LLM tipo chat; este debe contener la información clave para dirigir nuestra conversación hacia las respuestas de los modelos [7]. En entornos donde los LLMs operan como agentes autónomos o asistentes cognitivos, los *prompts* funcionan como una interfaz de programación semántica: especificaciones en lenguaje natural o estructurado que indican no solo qué resultado se desea obtener, sino también cómo debe generarse [8]. La ingeniería de *prompts*, también conocida como diseño de *prompts* o *prompt crafting*, se ha erigido como una disciplina crucial en la interacción efectiva con modelos de inteligencia artificial generativa, especialmente aquellos basados en modelos de redes Transformers [9] como los LLMs [10]. A medida que estos sistemas han demostrado una capacidad notable para comprender y generar texto complejo, la habilidad para formular instrucciones precisas y detalladas, conocidas como *prompts*, se ha vuelto fundamental para desbloquear su potencial y obtener resultados de alta calidad.

A pesar de que los sistemas de inteligencia artificial generativa han avanzado significativamente en su capacidad para comprender preguntas complejas, la calidad y precisión de las respuestas dependen en gran medida de cómo se formulan las preguntas. Comprender cómo interactuar con estos sistemas para construir *prompts* precisos permite obtener respuestas más claras y relevantes. Por el contrario, el uso de *prompts* confusos y poco elaborados puede limitar el potencial de la herramienta, resultando en respuestas superficiales. La ingeniería de prompts es esencial para optimizar la interacción con la IA, ya que un prompt bien diseñado mejora la calidad del contenido generado, aumentando la eficiencia y ampliando los escenarios de aplicación de los modelos de IA [10]. Además, proporcionar contexto adecuado y especificidad en los *prompts* es fundamental para que la IA comprenda plenamente la solicitud y genere respuestas más precisas y útiles [11].

La importancia de la ingeniería de *prompts* radica en su capacidad para optimizar la comunicación entre el usuario y la IA. Al proporcionar un contexto adecuado y ser específico en las solicitudes, se facilita que el modelo comprenda plenamente la intención del usuario y genere respuestas más precisas y útiles [11]. Esto implica considerar diversos aspectos al construir un *prompt*, como:

- La claridad del lenguaje.
- La inclusión de detalles relevantes.
- La definición de restricciones.
- El contexto previo.
- La especificación del formato de salida deseado.

Además, la ingeniería de *prompts* no es un proceso estático; requiere experimentación y ajuste iterativo. Diferentes modelos de IA pueden responder de manera distinta al mismo *prompt*, y su efectividad puede variar según la tarea específica. Por lo tanto, es crucial comprender las capacidades y limitaciones del modelo utilizado y adaptar las estrategias de diseño de *prompts* en consecuencia. Técnicas como la inclusión de ejemplos, la especificación del tono y el estilo, y la descomposición de tareas complejas en *prompts* más pequeños y manejables son parte del arsenal de un ingeniero de *prompts*.

2.2. Pruebas basadas en propiedades

Las Pruebas Basadas en Propiedades (*Property-Based Testing* o PBT) representan un cambio de paradigma en la forma en que se aborda la verificación del software [12]. A diferencia de las pruebas tradicionales basadas en

ejemplos específicos, donde se definen entradas y salidas concretas para un número limitado de casos, PBT se centra en la definición de propiedades o invariantes que deben cumplirse para un amplio rango de entradas generadas automáticamente [13].

En esencia, una prueba basada en propiedades describe el comportamiento esperado de una unidad de código en términos de relaciones que siempre deben ser verdaderas, independientemente de los datos de entrada específicos. El marco de pruebas PBT se encarga de generar una gran cantidad de entradas aleatorias (o pseudoaleatorias) dentro de los tipos de datos definidos y verifica si la propiedad especificada se mantiene para todas estas entradas.

La generación aleatoria de entradas es fundamental para PBT, permitiendo una exploración exhaustiva del espacio de pruebas y la detección de casos extremos [13]. Herramientas como QuickCheck [14], Hypothesis [12] y PropEr [15] automatizan este proceso, facilitando la implementación y ejecución de pruebas.

2.2.1. Beneficios de las pruebas basadas en propiedades

La adopción de PBT ofrece varios beneficios significativos:

- Mayor cobertura: Al generar automáticamente una gran cantidad de entradas diversas, PBT puede descubrir casos borde y escenarios inesperados que las pruebas basadas en ejemplos manuales podrían pasar por alto [16].
- Detección de errores sutiles: PBT es especialmente eficaz para encontrar errores que surgen de interacciones complejas entre diferentes entradas o en condiciones límite.
- Mejor comprensión de las especificaciones: El proceso de definir propiedades obliga a los desarrolladores a pensar profundamente sobre el comportamiento esperado del código, lo que lleva a una especificación más clara y precisa.
- Automatización eficiente: Una vez definidas las propiedades y los generadores, el proceso de prueba se automatiza, lo que reduce el esfuerzo manual requerido para crear y mantener un gran número de casos de prueba.

2.2.2. Desafíos de las pruebas basadas en propiedades

A pesar de sus ventajas, PBT también presenta ciertos desafíos:

- Definición de propiedades significativas: Identificar las propiedades correctas que capturen completamente el comportamiento deseado del sistema puede ser una tarea compleja y requiere una comprensión profunda del dominio.
- Curva de aprendizaje: La adopción de PBT requiere un cambio en la mentalidad de las pruebas y puede tener una curva de aprendizaje para los desarrolladores acostumbrados a las pruebas basadas en ejemplos.

Estas dificultades técnicas y cognitivas hacen que muchos equipos de desarrollo opten por enfoques más tradicionales y ad-hoc, es decir, sin una metodología formalizada, confiando en el conocimiento tácito del programador o en ejemplos particulares que pueden no representar adecuadamente el rango completo de posibles entradas. Sin embargo, estos métodos suelen ser limitados en cobertura, más susceptibles a errores sutiles y menos efectivos frente a condiciones cambiantes [14]. En cambio, PBT ofrece beneficios sustanciales al centrarse en invariantes generales y explorar masivamente el espacio de entradas, lo que conduce a una detección más temprana de errores complejos y a una mayor robustez del software [12].

El desafío, entonces, no es solo técnico, sino también práctico: ¿cómo facilitar la adopción de esta poderosa técnica sin exigir un alto nivel de experiencia en lógica formal o en diseño de propiedades?. Una de las principales barreras en la adopción generalizada de PBT es precisamente la dificultad de formular propiedades adecuadas, completas y expresivas. Esta actividad suele depender de la pericia y el entendimiento profundo del sistema bajo prueba, lo cual no siempre está documentado formalmente ni es fácil de trasladar a expresiones lógicas o invariantes [17].

En este contexto, LLMs emergen como una herramienta complementaria de gran valor gracias a su capacidad para comprender estructuras de código y generar descripciones funcionales [18]. Estos modelos pueden asistir en la formulación de propiedades útiles y la generación automatizada de casos de prueba [19]. Además, se ha

demostrado que los LLMs pueden interpretar correctamente la intención del desarrollador a partir de descripciones en lenguaje natural, facilitando la transición desde requisitos informales hasta propiedades formales [20].

Por lo tanto, la integración de los LLMs con los enfoques de PBT no solo representa una solución viable a los desafíos mencionados, sino que también marca una evolución significativa en las prácticas de aseguramiento de calidad. Esta combinación puede potenciar la adopción de PBT, automatizar tareas complejas y promover una validación más robusta del software desde las etapas tempranas del desarrollo.

2.3. Introducción a Hypothesis: una librería de Python

Hypothesis [12] se ha consolidado como una librería fundamental en el ecosistema de pruebas de Python, introduciendo y popularizando el paradigma de las pruebas basadas en propiedades PBT [13]. A diferencia de las pruebas unitarias tradicionales, que verifican el comportamiento del código mediante la ejecución de ejemplos específicos definidos manualmente, Hypothesis adopta un enfoque más declarativo y generativo. En esencia, permite a los desarrolladores especificar las propiedades o invariantes que deben cumplirse para un amplio rango de entradas, dejando que la propia librería se encargue de generar automáticamente una gran cantidad de casos de prueba para verificar estas propiedades. Este enfoque se inspira en herramientas pioneras como QuickCheck [21]. El desarrollo de Hypothesis surgió de la necesidad de superar las limitaciones de las pruebas basadas en ejemplos. Si bien estas últimas son útiles para verificar casos de uso comunes y escenarios bien definidos, a menudo resultan insuficientes para descubrir errores sutiles que se manifiestan en entradas inesperadas o en combinaciones complejas de datos.

David MacIver, el autor principal de Hypothesis [12], concibió la librería como una herramienta que permitiría a los desarrolladores especificar el comportamiento de su código, en lugar de limitarse a ejemplos particulares. Esta abstracción conduce a una cobertura de pruebas mucho más amplia y a una mayor probabilidad de detectar errores que, de otra manera, podrían pasar desapercibidos [13]. El funcionamiento de Hypothesis se centra en un ciclo iterativo de generación y verificación. El desarrollador comienza por definir una propiedad, que es esencialmente una función en Python que toma uno o más argumentos y devuelve un valor booleano (verdadero si la propiedad se cumple, falso en caso contrario). Estos argumentos están tipados, y para cada tipo, Hypothesis cuenta con una variedad de “estrategias” o generadores capaces de producir automáticamente una amplia gama de valores [12].

Durante la ejecución de la prueba, Hypothesis invoca repetidamente la función de propiedad, suministrando en cada llamada un conjunto diferente de entradas generadas por sus estrategias. Uno de los beneficios más distintivos de Hypothesis es su capacidad para realizar una “búsqueda inteligente” del espacio de entrada. Si una entrada generada revela una violación de la propiedad, Hypothesis no se limita a informar del fallo. En cambio, emplea algoritmos sofisticados para reducir el caso de prueba a su forma más simple y concisa que aún reproduce el error [12]. Este proceso de reducción es invaluable para la depuración, ya que aísla la causa raíz del problema y facilita su comprensión y corrección por parte del desarrollador. Esta capacidad de reducción es una de las fortalezas que distingue a las librerías de PBT como Hypothesis y su precursor, QuickCheck [21], en el panorama de las pruebas de software.

Los beneficios de adoptar Hypothesis en el flujo de trabajo de pruebas son numerosos. En primer lugar, mejora significativamente la cobertura de las pruebas al explorar automáticamente una gran variedad de entradas, incluyendo casos borde y combinaciones inesperadas que las pruebas manuales raramente cubren [13]. En segundo lugar, ayuda a descubrir errores sutiles que podrían permanecer ocultos en pruebas basadas en ejemplos limitados. En tercer lugar, el proceso de definir propiedades obliga a los desarrolladores a pensar más profundamente sobre las especificaciones y el comportamiento esperado de su código, lo que a menudo conduce a una mejor comprensión del problema y a un diseño más robusto.

Finalmente, la automatización de la generación de casos de prueba reduce la carga de trabajo manual asociada a la creación y mantenimiento de un gran número de pruebas individuales [12]. Su capacidad para reducir los casos de fallo a su mínima expresión facilita la depuración y contribuye a la construcción de software más confiable y robusto. Al centrarse en el comportamiento general en lugar de en ejemplos particulares, Hypothesis empodera a los desarrolladores para descubrir errores de manera más efectiva y construir sistemas de mayor calidad, siguiendo la senda marcada por herramientas como QuickCheck [21].

3. Materiales y métodos

En el contexto de la inteligencia artificial, el aprovechamiento de modelos de lenguaje grandes (LLMs) como Mistral Large [22] para generar código ha ganado tracción significativa. Sin embargo, la generación automática de código plantea una preocupación central: ¿cómo garantizar su corrección y robustez sin intervención manual intensiva? Para abordar esta problemática, se propone MistralHypothesis, una extensión para Visual Studio Code que integra generación de código mediante LLMs con validación automatizada basada en pruebas por propiedades, usando la biblioteca Hypothesis [12]. El propósito principal de MistralHypothesis es automatizar y facilitar la adopción de pruebas basadas en propiedades (PBT) durante la generación de código con LLMs, permitiendo a desarrolladores validar implementaciones sin salir de su entorno de trabajo. Esta extensión busca no solo verificar la corrección funcional, sino también promover buenas prácticas y robustez en el software generado.

3.1. Arquitectura de MistralHypothesis

1. Interfaz WebView integrada en Visual Studio Code: una ventana interactiva que se abre dentro del entorno de desarrollo, desde la cual el usuario puede ingresar su código en Python como si fuera un *prompt* de lenguaje natural o estructurado. Esta interfaz permite al usuario especificar claramente la funcionalidad deseada, ya sea mediante una descripción en lenguaje natural, un fragmento de código existente que se quiere extender o probar, o incluso un conjunto de requisitos.
2. Módulo de generación con Mistral Large: el código ingresado se interpreta como una especificación inicial que Mistral Large transforma en una implementación funcional en Python. Este módulo se encarga de comunicarse con la API de Mistral Large, enviando el código del usuario como un *prompt* diseñado para obtener una implementación coherente y funcional.
3. Módulo de validación con Hypothesis: el código generado es analizado para extraer propiedades lógicas, a partir de las cuales se diseñan pruebas con Hypothesis. El módulo examina la estructura del código, identifica posibles invariantes y utiliza el poder de Mistral Large para inferir propiedades relevantes que deberían cumplirse. También se genera una sección de sugerencias con mejoras de diseño, posibles errores lógicos y refactorizaciones recomendadas. Por ejemplo, si el código generado tiene una lógica condicional compleja, MistralHypothesis podría sugerir pruebas para cubrir diferentes ramas o identificar posibles condiciones de borde no manejadas. Además, podría recomendar refactorizaciones para mejorar la legibilidad o el rendimiento del código generado.

3.2. Selección de API para la modificación de código

Considerando que los diferentes LLMs pueden generar respuestas totalmente distintas, optamos por buscar la mejor manera de analizar las características de cada uno, y qué mejor que hacerlo de forma paralela modificando parámetros principales y analizando las salidas generadas. En este contexto, GitHub Models se presentó como la plataforma elegida, ofreciendo un entorno que facilita la consulta de algunas de las APIs de LLMs de mayor demanda actual. Su enfoque permite no solo acceder de manera centralizada a modelos de última generación, sino también establecer comparaciones sistemáticas entre ellos, garantizando que las decisiones sobre su implementación se basen en datos objetivos.

La Tabla 1 ilustra el desempeño de algunos de los modelos más significativos del mercado en los reconocidos *benchmarks* de codificación HumanEval [24] y MBPP [25].

Tabla 1. Rendimiento en los puntos de referencia de codificación populares de los modelos LLM líderes del mercado.

Modelo	HumanEval	MBPP
Mistral Large	45.1%	73.1%
LLaMA 2 70B	29.3%	49.8%
GPT-3.5	48.1%	-
GPT-4	67.0%	-
Claude 2	-	-
Gemini Pro 1.0	67.7%	-

Fuente: Mistral AI Team. (2024, febrero). Au large. Mistral AI.

Estos puntos de referencia, ampliamente adoptados por la comunidad científica para evaluar la habilidad de los modelos en la resolución de problemas de programación a partir de descripciones en lenguaje natural, nos ofrecieron una perspectiva inicial sobre las capacidades relativas de cada arquitectura. Los resultados revelaron una destacada aptitud de Mistral Large en el *benchmark* MBPP, centrado en la resolución de problemas de programación. Por otro lado, Gemini Pro demostró un liderazgo en HumanEval, un *benchmark* diseñado para evaluar la capacidad de los modelos para abordar problemas más intrincados que requieren un razonamiento más profundo.

3.3. Funcionamiento y flujo de trabajo

El flujo de trabajo propuesto es simple, pero funcional:

1. El usuario abre un archivo Python en VS Code y lanza la extensión.
2. Desde la interfaz proporcionada, el usuario ingresa un fragmento de código (o descripción funcional) como un *prompt*.
3. Al enviar el *prompt*, la extensión interactúa con la API de Mistral Large, que responde con una implementación funcional del código objetivo aplicando la lógica que implica el uso de la librería Hypothesis.
4. Inmediatamente, este nuevo código es evaluado bajo una batería de pruebas generadas automáticamente con Hypothesis, basadas en propiedades inferidas o definidas previamente.
5. El resultado del análisis se presenta al usuario en dos apartados:
    - Código generado con PBT: Implementación funcional con pruebas Hypothesis incorporadas.
    - Área de mejora: Comentarios sobre calidad del código, posibles mejoras y áreas de oportunidad.

El vasto ecosistema de proyectos y herramientas de evaluación presentes en GitHub [26] nos proporcionó el marco necesario para analizar objetivamente el rendimiento de modelos como Mistral Large, Llama y ChatGPT en tareas relevantes para nuestro objetivo de manera paralela, bajo las mismas entradas de *prompts*. Estos resultados, junto con consideraciones sobre el costo y la disponibilidad de las APIs, influyeron en nuestra decisión de enfocar nuestra herramienta en la integración con Mistral Large, explorando sus capacidades para la generación de propiedades de prueba que capturen las invariantes esenciales del software bajo análisis.

Durante el desarrollo de MistralHypothesis, se diseñaron diferentes tipos de *prompts* para evaluar su efectividad en distintos contextos de desarrollo:

- *Prompts* funcionales: Solicitan implementar funciones con condiciones específicas, como: “Escribe una función que invierta una cadena y valide que siempre regrese la cadena original si se invierte dos veces.”
- *Prompts* de propiedades: Describen comportamientos generales del código como: “Genera una función que sume dos listas y asegúrate de que la longitud del resultado sea igual a la suma de ambas longitudes.”
- *Prompts* mixtos: Mezclan descripciones en lenguaje natural con fragmentos de código para guiar al modelo otorgando mayor contexto.
    - Un ejemplo de esto sería: *Escribe una función en Python que reciba dos listas y las concatene. El resultado tendrá una longitud igual a la suma de ambas. Usa como base la siguiente línea de código:*
    - *def concatenar_listas(lista1, lista2):*

Estos *prompts* no solo permiten generar código, sino que actúan como catalizadores para el diseño automático de propiedades, lo cual representa una innovación clave de esta herramienta. Para tener mayor entendimiento del mismo, se llevó a cabo una prueba de *prompt* con el siguiente ejemplo:

Crear un script en Python para analizar archivos CSV y extraer los nombres y correos electrónicos con los siguientes requisitos:

- El script debe ser capaz de leer archivos CSV grandes y manejarlos de manera eficiente.
- Debe extraer los nombres y correos electrónicos de las columnas específicas en el archivo CSV.
- La salida debe estar en formato JSON para facilitar su uso en otros programas.
- Haz un ejemplo de una ejecución y muestra el JSON resultante.

Con el estudio de los resultados se tomaron puntos de relevancia, los cuales se muestran en la Tabla 2.

Tabla 2. Comparativa de modelos LLMs.

Criterio	Mistral Large	GPT-4	Gemini Pro
¿Explica cómo lo implementa o el funcionamiento?	Sí	Sí	Sí
¿Genera código sencillo?	Sí	Sí	Sí
¿El JSON de la salida es correcto?	Sí	No	Sí
¿Muestra un ejemplo de ejecución?	Sí	Sí	Sí
¿Ejecuta correctamente?	No, falta de generación de datos	No, inventa datos	No, ha sido necesario modificar datos
¿Lenguaje sencillo?	Sí	Sí	Sí
¿Es veraz la información?	Sí	Sí	Sí
¿Libre uso (coste)?	Sí	No, uso limitado	No
Total	7/8	5/8	6/8

Fuente: Tomado de [27].

Considerando lo anterior, dentro de esta plataforma y con los datos de rendimiento obtenidos, se definieron los actores clave para la interacción con la API, estableciendo los roles y responsabilidades dentro del proceso de prueba automatizada:

- Sistema: Actúa como el orquestador principal, siendo el asistente para el desarrollo de software que utiliza una librería de Python para la ejecución de pruebas basadas en propiedades. El Sistema es responsable de enviar los *prompts* a la API y procesar las respuestas.
- Usuario: Es quien proporciona el *prompt* inicial que será evaluado por la API del LLM. Este *prompt* guía la generación o modificación de código y la posible sugerencia de propiedades de prueba.
- Asistente: Representa el modelo de lenguaje (inicialmente Mistral Large) que recibe el *prompt* y realiza las modificaciones solicitadas, además de generar comentarios relevantes con relación al código.

Con estos parámetros definidos, y a través de la experimentación con distintos *prompts*, se optó por el uso de la API Mistral Large para la generación de código y la asistencia en la definición de propiedades de prueba en las etapas iniciales de desarrollo de nuestra herramienta. Esta decisión se fundamentó en su capacidad demostrada para un mejor entendimiento del código Python, el lenguaje principal de nuestra librería de pruebas basadas en propiedades, así como en su sólida comprensión del lenguaje natural utilizado en los *prompts*, lo que facilitó la obtención de respuestas más precisas y útiles para nuestro objetivo.

3.4. Desarrollo de la extensión

En el proceso de indagación para la creación de la extensión, encontramos un manual oficial de Microsoft donde se explica a detalle la manera en que se puede crear una nueva extensión, incluyendo las posibles instalaciones según las necesidades de los desarrolladores y pasos específicos [28], como se muestra en la Figura 1. Con esto en cuenta, comenzamos el desarrollo y la implementación de la API de Mistral Large.

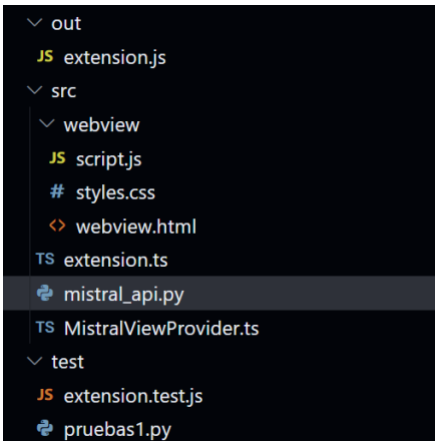

Figura 1. Organización de la extensión.

Con el uso de la plataforma GitHub Models, obtuvimos el token de acceso para la API de Mistral Large y, de igual manera que para la creación de la extensión, se utilizó una plantilla base de la configuración del LLM. Esta se puede visualizar en la Figura 2.

```
"""Run this model in Python

> pip install mistralai>=1.0.0
"""
import os
from mistralai import Mistral, UserMessage, SystemMessage

# To authenticate with the model you will need to generate a personal access token (PAT) in your GitHub settings.
# Create your PAT token by following instructions here: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens
client = Mistral(
    api_key=os.environ["GITHUB_TOKEN"],
    server_url="https://models.github.ai/inference"
)

response = client.chat(
    model="mistral-ai/Mistral-Large-2411",
    messages=[
        SystemMessage(""),
        UserMessage("What is the capital of France?"),
    ],
    temperature=0.8,
    max_tokens=2048,
    top_p=0.1
)

print(response.choices[0].message.content)
```

Figura 2. Plantilla base que se ofrece en GitHub Models.

4. Resultados

Para facilitar la obtención de la extensión, se optó por publicarla directamente en el Marketplace de Visual Studio Code. Tal como se muestra en la Figura 3, esta puede ser descargada de forma directa y recibir actualizaciones automáticas cuando estén disponibles.

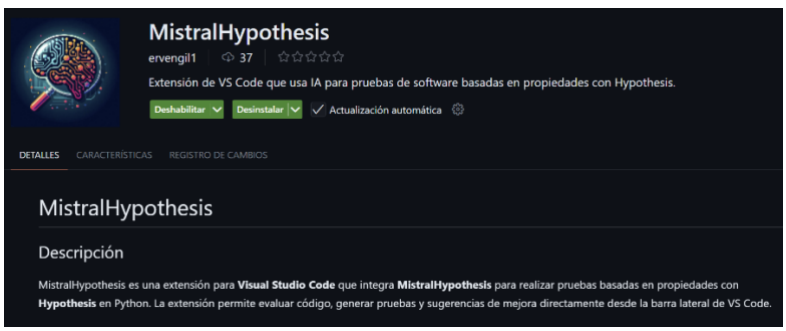

Figura 3. Extensión publicada.

En el área de desarrollo, y ya con la extensión descargada, se ejecuta de la siguiente manera, como se muestra en la Figura 4.

- Tomamos el código a evaluar.
- Copiamos el código y lo pegamos en el recuadro blanco de la extensión.
- Hacemos clic en el botón Ejecutar.
- Con esto, la extensión evalúa el código y expone las respuestas generadas con el uso de la API Mistral.
- El resultado brinda indicaciones a realizar, presenta el código que deberá ser copiado para realizar la respectiva prueba, genera comentarios de posibles mejoras y explica el funcionamiento general del código.

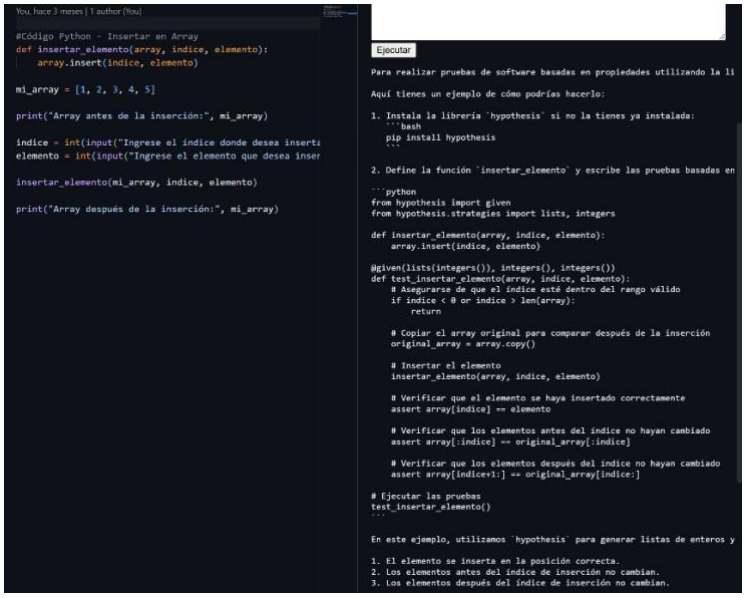

Figura 4. Extensión en uso.

4.1. Validación de resultados de MistralHypothesis

Esta validación se genera considerando el código generado por la extensión, el cual se copia y pega en el segundo archivo de pruebas llamado pruebas2.py.

- Ejecución Directa: En esta modalidad, se copia y pega directamente el código generado, se ejecuta de manera normal y no se arroja mensaje de error alguno. Esto significa que el código es correcto y no presenta fallas, como se muestra en la Figura 5.

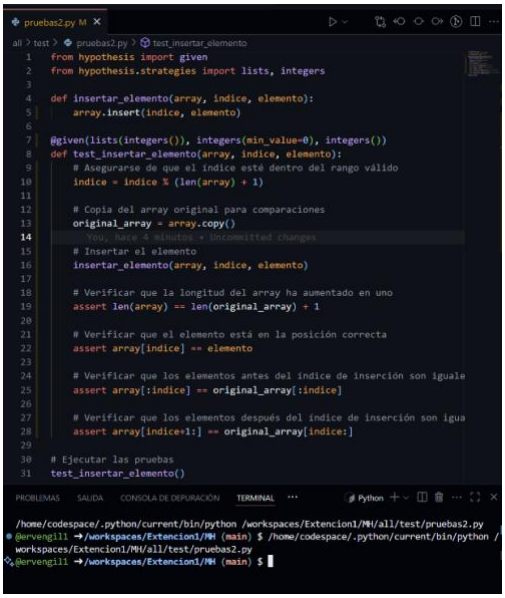

Figura 5. Validación sin error.

- Ejecución con error: Se tomó el mismo código y se modificó la sección donde el arreglo (array) incrementa de 1 en 1, sustituyéndolo por un incremento de 2 en 2. Esto ocasionó un error, el cual Hypothesis identificó y describió la falla, como se muestra en la Figura 6.

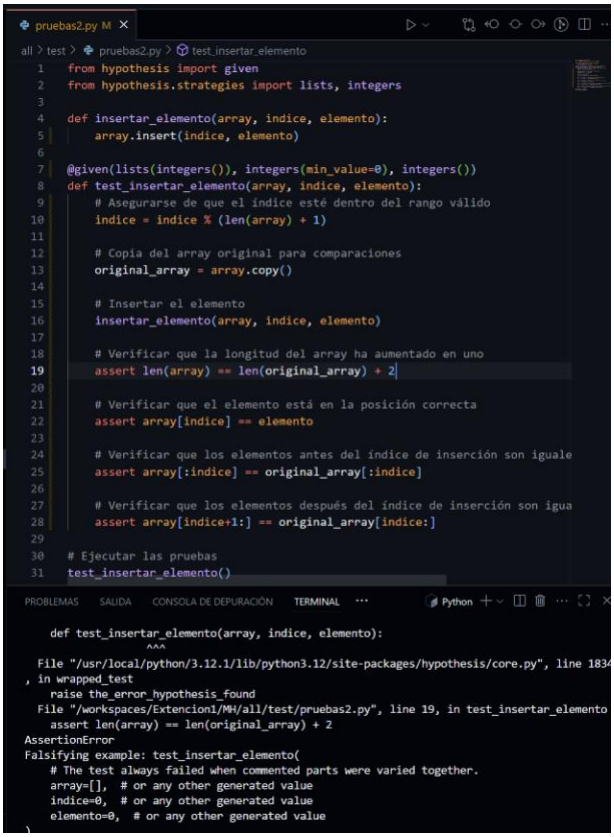

Figura 6. Validación con error.

Posterior a esto, se solicita a la extensión que corrija el error e indique en qué falla, así como que explique dichas correcciones mediante un *prompt*. El resultado generado se muestra en la Figura 7.

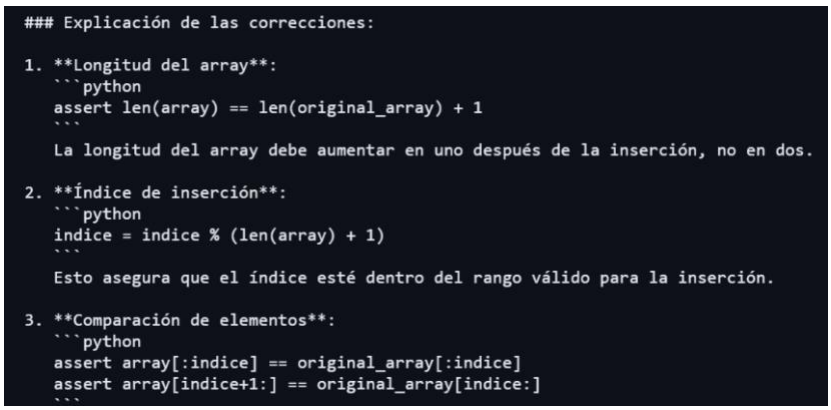

Figura 7. Validación con error.

Con esto podemos observar que, aun cuando se resuelve el error del aumento del rango de inserción, también se comentan mejoras sobre el mismo código generado por la extensión. Esto ocurre porque la API busca mejorar la entrada del *prompt*, en cualquier caso.

5. Conclusiones

MistralHypothesis representa una herramienta innovadora que democratiza el acceso a las pruebas basadas en propiedades (PBT), permitiendo que desarrolladores con distinta experiencia integren prácticas robustas de calidad directamente desde Visual Studio Code.

Al combinar la capacidad generativa de un modelo de lenguaje de gran tamaño como Mistral Large con el rigor de la verificación automatizada mediante Hypothesis, la extensión facilita la creación de software más confiable y mantenible, reduciendo la carga manual de generar pruebas. Esta colaboración entre humanos y modelos inteligentes establece un nuevo paradigma en el desarrollo, en el que la automatización y la retroalimentación continua fomentan una cultura de calidad desde las etapas iniciales.

Se plantea para futuro la implementación de pruebas en tiempo real dentro del editor, lo que permitirá a los desarrolladores observar inmediatamente el comportamiento del código bajo diversas entradas y corregir errores de forma ágil. Además, se explorarán nuevos protocolos como *Model Context Protocol* (MCP) para mejorar la interpretación del contexto y la generación de propiedades de prueba más efectivas, así como mejoras en la interfaz para facilitar el ingreso de *prompts* y la visualización clara de resultados y sugerencias. Aunque persisten desafíos en la definición precisa de *prompts* y propiedades que guíen adecuadamente a los modelos hacia un código útil, el potencial de esta herramienta para transformar el flujo de trabajo y elevar la calidad del software es significativo.

7. Referencias

[1] Baresi, L., Pezze, M. (2006). An introduction to software testing. *Electronic Notes in Theoretical Computer Science, 148* (1), 89–111. https://doi.org/10.1016/j.entcs.2005.12.014

[2] Fink, G., Bishop, M. (1997). Property-based testing: A new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes, 22* (4), 74–80. https://doi.org/10.1145/263244.263267

[3] Kaner, C., Bach, J., Pettichord, B. (2002). *Testing computer software*. Wiley.

[4] Beizer, B. (1990). *Software testing techniques*. Van Nostrand Reinhold Company. https://dl.acm.org/doi/10.5555/79060

[5] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ..., Amodei, D. (2020). *Language models are few-shot learners*. 34th International Conference on Neural Information Processing Systems. Vancouver BC, Canada. https://dl.acm.org/doi/abs/10.5555/3495724.3495883

[6] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys, 55* (9), 1–35. https://doi.org/10.1145/3560815

[7] White, J., Fu., Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D. C. (2023). Prompt engineering techniques for large language models: A survey. *arXiv preprint*. https://arxiv.org/abs/2302.11382

[8] Zhou, X., Schärli, N., Hou, L. Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O. Le, Q., Chi, E. (2022). Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint*. https://doi.org/10.48550/arXiv.2205.10625

[9] Díaz Benito, G. (2024). *Análisis sobre la utilización de transformers y modelos generativos para generación de anuncios* (Trabajo de Fin de Grado). Universidad Politécnica de Madrid, España. https://oa.upm.es/82703/

[10] Jason ZK. (2024*). Ingeniería de prompts y prompts de IA: Conceptos, diseño y optimización*. https://es.blog.jasonzk.com/ai/aipromptengineering/

[11] Kippel01. (2024). *La importancia del "prompt engineering" en la calidad de respuestas de herramientas de inteligencia artificial*. https://www.kippel01.com/tecnologia/importancia-prompt-engineering-calidad-respuestas-herramientas-inteligencia-artificial

[12] MacIver, D. R., Hatfield-Doods, Z. (2018). Hypothesis: A new approach to property-based testing. *Journal of Open Source Software, 4* (43). https://doi.org/10.21105/joss.01891

[13] Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., Head, A. (2024). *Property-Based Testing in Practice*. IEEE/ACM 46th International Conference on Software Engineering (ICSE). Lisbon, Portugal. https://doi.org/10.1145/3597503.3639581

[14] Claessen, K., Hughes, J. (2000). *QuickCheck: A lightweight tool for random testing of Haskell programs*. Fifth ACM SIGPLAN international conference on Functional programming. Singapore. https://doi.org/10.1145/351240.351266

[15] Papadakis, M., Sagonas, K. (2011). *A PropEr integration of types and function specifications with property-based testing*. 10th ACM SIGPLAN workshop on Erlang. Tokyo, Japan. https://doi.org/10.1145/2034654.2034663

[16] keploy. (2024). *Property-based testing: A comprehensive guide*. https://dev.to/keploy/property-based-testing-a-comprehensive-guide-lc2

[17] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F. (2010). *Model checking lots of systems: Efficient verification of temporal properties in software product line*s. 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. Cape Town South Africa. https://doi.org/10.1145/1806799.1806850

[18] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., Sutton, C. (2023). Program synthesis with large language models. *arXiv preprint*. https://doi.org/10.48550/arXiv.2108.07732

[19] Chen, M., Tworek, J., Jun, H., ... (2021). Evaluating large language models trained on code. *arXiv preprint*. https://doi.org/10.48550/arXiv.2107.03374

[20] Kazemitabaar, M., Williams, J., Drosos, I., Grossman, T., Henley, A. Z., Negreanu, C., Sarkar, A. (2024). *Improving steering and verification in AI-assisted data analysis with interactive task decomposition*. 37th Annual ACM Symposium on User Interface Software and Technology (UIST). Pittsburgh PA, USA. https://doi.org/10.1145/3654777.3676345

[21] Claessen, K., Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices, 35* (9), 268–279. https://doi.org/10.1145/357766.351266

[22] Higginbotham, G. Z., Matthews, N. S. (2024). *Prompting and in-context learning: Optimizing prompts for Mistral Large*. https://doi.org/10.21203/rs.3.rs-4430993/v1

[23] Mistral AI Team. (2024). *Au large*. https://mistral.ai/news/mistral-large

[24] Zheng, Q., Guo, Y., ... (2023). Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. *arXiv preprint*. https://doi.org/10.48550/arXiv.2303.17568

[25] Yu, Z., Wang, Z., ... (2024). Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint*. https://doi.org/10.48550/arXiv.2412.21199

[26] Dohmke, T. (2024). *Introducing GitHub models: A new generation of AI engineers building on GitHub*. https://github.blog/news-insights/product-news/introducing-github-models/

[27] Pérula, R., Calleja, T., Hernández de la Cruz, J. M. (2024). *Versus: OpenAI GPT-4 vs. Google Gemini Pro vs. Mistral AI Large*. https://www.paradigmadigital.com/dev/versus-openai-gpt4-google-gemini-pro-mistral-ai-large/

[28] Visual Studio. (2025). *Your first extension*. https://code.visualstudio.com/api/get-started/your-first-extension

[29] Hou, X., Zhao, Y., Wang, S., Wang, H. (2025). Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint*. https://doi.org/10.48550/arXiv.2503.23278

