CognitivaTech: Ingeniería de Software Inteligente y Sistemas Adaptativos. 2026; 3(1):53
doi: DOI: 10.63688/cognitivatech.v3.i1.53

ORIGINAL

Automating software testing using artificial intelligence and machine learning

Automatización de pruebas de software mediante inteligencia artificial y aprendizaje automático

Carlos Hernández Salas[1]
Carmen Carolina Ortega Hernández[2]

[1] Instituto Tecnológico de Tapachula. Tapachula, México
[2] Universidad Autónoma de Chiapas. Tapachula, México

Recibido: 13-02-2026 | Revisado: 20-02-2026 | Aceptado: 10-04-2026 | Publicado: 21-04-2026

Autor de correspondencia: Carlos Hernández Salas

ABSTRACT

Introduction: Software test automation is an essential strategy to reduce times, improve coverage and detect defects during development. The incorporation of artificial intelligence (AI) and machine learning (ML) opens up new possibilities for generating, prioritizing, and optimizing test cases. Objective: To evaluate the effectiveness of AI- and ML-assisted software test automation versus a conventional approach in Software Technology Engineering students at the Universidad Autónoma de Nuevo León, Mexico. Methodology: A quantitative, applied, experimental, prospective and comparative study was carried out with 128 students, randomly distributed into a control group (n=64) and an experimental group (n=64). Execution time, compilable and executable tests, coverage of lines and branches, mutation score and detected defects were evaluated. Results: The experimental group reduced the mean resolution time from 44.8 to 36.2 minutes and obtained higher values of line coverage (82.6 % vs. 74.8 %), branch coverage (75.4 % vs. 66.1 %) and mutation score (70.8 % vs. 59.7 %). It also detected a greater number of defects, with statistically significant differences ($p<0.001$). Conclusions: The incorporation of AI and ML showed advantages in efficiency and effectiveness of automated tests, although their application requires technical supervision and human validation.

Keywords: artificial intelligence; machine learning; software testing; test automation; software engineering.

RESUMEN

Introducción: La automatización de pruebas de software constituye una estrategia esencial para reducir tiempos, mejorar la cobertura y detectar defectos durante el desarrollo. La incorporación de inteligencia artificial (IA) y aprendizaje automático (ML) ofrece nuevas posibilidades para generar, priorizar y optimizar casos de prueba. Objetivo: Evaluar la eficacia de la automatización de pruebas de software asistida por IA y ML frente a un enfoque convencional en estudiantes de Ingeniería en Tecnología de Software de la Universidad Autónoma de Nuevo León, México. Metodología: Se desarrolló un estudio cuantitativo, aplicado, experimental, prospectivo y comparativo con 128 estudiantes, distribuidos aleatoriamente en un grupo control (n=64) y un grupo experimental (n=64). Se evaluaron tiempo de ejecución, pruebas compilables y ejecutables, cobertura de líneas y ramas, mutation score y defectos detectados. Resultados: El grupo experimental redujo el tiempo medio de resolución de 44,8 a 36,2 minutos y obtuvo mayores valores de cobertura de líneas (82,6 % frente a 74,8 %), cobertura de ramas (75,4 % frente a 66,1 %) y mutation score (70,8 % frente a 59,7 %). Asimismo, detectó un mayor número de defectos, con diferencias estadísticamente significativas ($p<0,001$). Conclusiones: La incorporación de IA y ML mostró ventajas en eficiencia y efectividad de las pruebas automatizadas, aunque su aplicación requiere supervisión técnica y validación humana.

Palabras clave: inteligencia artificial; aprendizaje automático; pruebas de software; automatización de pruebas; ingeniería de software.

INTRODUCCIÓN

La calidad del software se basa en la capacidad de detectar defectos antes de que se expandan a las etapas de operaciones, mantenimiento o despliegue. En sistemas cada vez más complejos y sometidos a ciclos de entrega acelerados las actividades de prueba deben ejecutarse con rapidez sin perder profundidad ni capacidad de detección. La automatización ha respondido parcialmente a esta necesidad al reducir tareas repetitivas y permitir la ejecución sistemática de casos de prueba, pero aún demanda esfuerzo para diseñar escenarios, mantener scripts, construir oráculos y priorizar ejecuciones. La inclusión de la inteligencia artificial (IA) ha expandido este escenario, mediante la incorporación de mecanismos capaces de aprender patrones, generar pruebas y ayudar en las decisiones dentro del proceso de testing[1]. Las revisiones recientes indican que el aprendizaje automático (ML) se aplica a varias etapas de las pruebas de software, desde la predicción de defectos hasta la generación, selección y optimización de casos de prueba[2].

Una de las áreas con mayor desarrollo es la generación automatizada de pruebas. Los enfoques basados en ML permiten utilizar información histórica, características del código y resultados previos para producir o mejorar entradas, oráculos y estrategias de búsqueda. Fontes y Gay identificaron aplicaciones de aprendizaje supervisado, no supervisado y por refuerzo en pruebas unitarias, de sistemas, interfaces y rendimiento, pero también señalaron problemas de calidad de datos, escalabilidad, evaluación y reproducibilidad[3]. Estas limitaciones son relevantes porque la adopción de automatización no depende únicamente de disponer de herramientas; también intervienen la naturaleza del sistema, el costo de mantenimiento, la frecuencia de regresión y la madurez del proceso[4]. Por ello, evaluar IA aplicada al testing requiere considerar no solo cuántos casos se generan, sino si son ejecutables, cubren comportamientos relevantes y permiten detectar defectos.

El ML también ha ganado importancia en las pruebas de regresión, donde el crecimiento de las suites puede volver costosa su ejecución completa después de cada cambio. La selección y priorización basada en aprendizaje permite ordenar los casos según su probabilidad de revelar fallos o su utilidad dentro de restricciones de tiempo [5]. Los trabajos con aprendizaje por refuerzo demostraron que un agente puede aprender políticas de selección y priorización a partir del historial de ejecuciones en integración continua [6]. Posteriormente, la comparación entre estrategias learning-to-rank y reinforcement learning mostró que el rendimiento depende de las características del proyecto y de los datos disponibles, por lo que no existe una técnica universalmente superior [7].

Antes del auge de los modelos generativos, la generación automática de pruebas ya había avanzado mediante técnicas de búsqueda. EvoSuite consolidó un enfoque evolutivo capaz de producir suites para software orientado a objetos buscando maximizar criterios de cobertura y generar aserciones automáticamente[8]. En otros lenguajes, Pynguin extendió estas posibilidades hacia Python y mostró que la generación automatizada puede alcanzar niveles relevantes de cobertura, aunque su efectividad varía según el módulo, la información de tipos y la estrategia de búsqueda[9]. Estos antecedentes permiten comparar los nuevos métodos de IA con técnicas automatizadas convencionales y evitar que su valor sea juzgado únicamente frente a la escritura manual de pruebas.

La evolución de los transformadores introdujo una nueva etapa en el testing automatizado. Los modelos preentrenados pueden generar aserciones para pruebas unitarias a partir del contexto del método analizado, reduciendo parte del esfuerzo necesario para construir oráculos [10]. TOGA amplió esta línea mediante inferencia neural de oráculos de excepción y aserción, evidenciando que la combinación de generación tradicional y aprendizaje puede incrementar la detección de fallos reales [11]. De forma similar, A3Test incorporó conocimiento específico sobre aserciones y verificación de firmas para mejorar la corrección de los casos producidos por modelos profundos [12]. Estos desarrollos muestran que la IA puede intervenir en tareas que antes dependían en mayor medida del desarrollador.

La aparición de grandes modelos de lenguaje (LLM) aceleró esta transición. CodaMOSA integró modelos preentrenados con técnicas de búsqueda para superar estancamientos de cobertura durante la generación automática [13]. MuTAP combinó LLM con mutation testing para retroalimentar la generación mediante mutantes sobrevivientes, destacando que una cobertura elevada no garantiza por sí sola capacidad para revelar defectos [14]. Estudios empíricos posteriores han mostrado que el desempeño de los LLM en pruebas unitarias varía según el modelo, el diseño del prompt, el contexto suministrado y las métricas utilizadas [15]. Una revisión sistemática reciente confirma además que el preprocesamiento, el posprocesamiento, los conjuntos de datos y la integración con los flujos existentes condicionan la calidad de las pruebas generadas [16].

A pesar de estos avances, persiste la necesidad de evidencia experimental que compare, bajo

condiciones controladas y métricas homogéneas, un flujo convencional de automatización frente a uno asistido por IA y ML en contextos de formación avanzada en ingeniería de software. Gran parte de la evidencia disponible se concentra en benchmarks, repositorios y evaluaciones técnicas de modelos, lo que deja espacio para analizar cómo estas tecnologías afectan el desempeño de usuarios que deben diseñar, depurar y ejecutar pruebas en tareas de programación. Esta cuestión resulta pertinente en México, donde la formación de profesionales capaces de integrar IA con prácticas rigurosas de aseguramiento de calidad puede fortalecer competencias vinculadas con la industria de software.

Por ello, el objetivo de esta investigación es evaluar la eficacia de la automatización de pruebas de software asistida por inteligencia artificial y aprendizaje automático, en comparación con un enfoque de automatización convencional, en estudiantes de Ingeniería en Tecnología de Software de la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo León, México. La comparación considerará tiempo de generación, validez y ejecutabilidad de las pruebas, cobertura de código y ramas, mutation score y capacidad de detección de defectos. Se plantea como hipótesis que el flujo asistido por IA y ML producirá diferencias estadísticamente significativas en la eficiencia y efectividad del proceso respecto del enfoque convencional.

MÉTODOS

La investigación se desarrolló bajo un enfoque cuantitativo, de tipo aplicado, con diseño experimental, prospectivo y comparativo. El estudio tuvo como propósito determinar las diferencias en eficiencia y efectividad entre un proceso convencional de automatización de pruebas de software y un proceso asistido por inteligencia artificial y aprendizaje automático. La unidad de análisis estuvo constituida por estudiantes de Ingeniería en Tecnología de Software de la Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo León, México, durante el periodo académico agosto-diciembre de 2026.

La población objetivo estuvo conformada por estudiantes matriculados entre sexto y décimo semestre de Ingeniería en Tecnología de Software, debido a que en estos niveles los participantes cuentan con conocimientos previos suficientes de programación, estructuras de datos, ingeniería de software y pruebas. La población accesible correspondió a los estudiantes que cumplían estos requisitos y se encontraban activos académicamente durante el periodo de ejecución. El número definitivo de estudiantes elegibles se verificó mediante los registros académicos institucionales antes del reclutamiento. El tamaño muestral se estableció mediante análisis de potencia para la comparación de dos grupos independientes, considerando un nivel de significancia de 0,05, potencia estadística de 0,80 y un tamaño de efecto medio de Cohen de 0,50. Se determinó una muestra mínima de 128 participantes, distribuida equitativamente en 64 estudiantes para el grupo control y 64 para el grupo experimental.

Se empleó muestreo probabilístico estratificado por semestre. Una vez identificados los estudiantes elegibles, se formaron estratos correspondientes a sexto, séptimo, octavo, noveno y décimo semestre; dentro de cada estrato se realizó selección aleatoria para garantizar una representación proporcional. Posteriormente, los participantes fueron asignados aleatoriamente al grupo control o experimental mediante una secuencia generada por computadora. Esta estrategia buscó disminuir posibles diferencias iniciales relacionadas con el avance académico y la experiencia acumulada en programación.

Los criterios de inclusión comprendieron estar matriculado en el programa durante el periodo estudiado, cursar entre sexto y décimo semestre, haber aprobado las asignaturas previas relacionadas con programación e ingeniería de software, tener conocimientos básicos de pruebas unitarias y aceptar voluntariamente participar. Se excluyeron estudiantes que no completaron todas las actividades experimentales, presentaron problemas técnicos que impidieron registrar las métricas requeridas o utilizaron herramientas de IA distintas de las autorizadas durante las sesiones. También se eliminaron del análisis los registros incompletos o aquellos en los que no fue posible verificar la ejecución de las pruebas generadas.

Antes de la intervención se aplicó una ficha de caracterización para registrar edad, semestre, experiencia previa en programación, experiencia en automatización de pruebas y frecuencia de utilización de herramientas de IA. Adicionalmente, se administró una prueba inicial de conocimientos compuesta por 20 ítems de selección múltiple relacionados con fundamentos de testing, diseño de casos de prueba, cobertura, pruebas unitarias y detección de defectos. El instrumento fue revisado por tres docentes con experiencia en ingeniería de software y pruebas, con el propósito de evaluar pertinencia, claridad y correspondencia de los ítems con las competencias

estudiadas. Esta evaluación inicial permitió comprobar la comparabilidad basal de ambos grupos.

El experimento utilizó proyectos Java seleccionados de repositorios con versiones defectuosas y corregidas, manteniendo para ambos grupos los mismos módulos, nivel de complejidad y tiempo disponible. Se priorizaron programas con defectos conocidos y reproducibles para establecer una referencia objetiva de la capacidad de detección de fallos. Defects4J se tomó como fuente de programas reales con errores documentados, mientras que la generación convencional de suites se apoyó en herramientas automatizadas utilizadas ampliamente en investigación sobre testing17,18. Los proyectos fueron preparados previamente para garantizar su compilación y ejecución dentro de un entorno uniforme.

El grupo control desarrolló las actividades mediante un flujo convencional de automatización utilizando Java, JUnit 5, Maven, JaCoCo y herramientas tradicionales de generación automática de pruebas. Los participantes podían inspeccionar el código fuente, diseñar casos de prueba, ejecutar las suites y corregir errores de compilación, pero no utilizar modelos generativos ni asistentes basados en IA. El grupo experimental trabajó sobre los mismos problemas y bajo el mismo límite temporal, aunque contó con un flujo asistido por un modelo de lenguaje para generación y refinamiento de pruebas, además de un componente de aprendizaje automático destinado a apoyar la priorización de casos según información obtenida durante las ejecuciones.

Cada participante resolvió seis tareas experimentales distribuidas en módulos de dificultad equivalente. Para reducir efectos de aprendizaje, el orden de los módulos fue aleatorizado. En cada tarea se registró automáticamente el tiempo transcurrido desde el inicio hasta la obtención de una suite ejecutable, número total de pruebas generadas, número de pruebas compilables, porcentaje de pruebas ejecutadas correctamente, cobertura de líneas, cobertura de ramas, cantidad de defectos detectados y puntuación de mutación. La cobertura se obtuvo mediante JaCoCo y el análisis de mutación mediante PIT. El mutation score se calculó como la proporción de mutantes eliminados respecto del total de mutantes ejecutables, por considerarse una medida más próxima a la capacidad de una suite para identificar comportamientos defectuosos que la cobertura estructural por sí sola19.

La variable independiente fue la estrategia de automatización, operacionalizada en dos categorías: convencional y asistida por IA/ML. Las variables dependientes fueron tiempo de generación de una suite válida, porcentaje de pruebas compilables, porcentaje de pruebas ejecutables, cobertura de líneas, cobertura de ramas, mutation score y número de defectos detectados. Como variables de control se consideraron semestre académico, experiencia previa en programación, experiencia en testing y puntuación obtenida en la evaluación inicial.

Los datos fueron procesados mediante estadística descriptiva e inferencial. Para las variables cuantitativas se calcularon media, desviación estándar, mediana, rango intercuartílico e intervalos de confianza del 95 %. La distribución de los datos se verificó mediante la prueba de Shapiro-Wilk. Cuando se cumplieron los supuestos de normalidad y homogeneidad de varianzas, las diferencias entre grupos se analizaron mediante la prueba t de Student para muestras independientes; en caso contrario se utilizó la prueba U de Mann-Whitney. Las proporciones fueron comparadas mediante chi cuadrado o prueba exacta de Fisher cuando correspondía. Se adoptó un nivel de significancia de $p < 0,05$ y se calcularon tamaños del efecto mediante d de Cohen o delta de Cliff. Complementariamente, se planteó un modelo de regresión para estimar la asociación entre la estrategia utilizada y los indicadores de desempeño, controlando las características basales de los participantes.

La investigación respetó los principios de participación voluntaria, confidencialidad y uso exclusivamente académico de la información. Los participantes recibieron información sobre los objetivos, procedimiento y tratamiento de los datos antes de firmar el consentimiento informado. Los registros experimentales fueron codificados mediante identificadores anónimos y almacenados sin información personal directamente identificable. La participación o el desempeño obtenido durante el experimento no influyó en las calificaciones académicas de los estudiantes.

RESULTADOS

Participaron 128 estudiantes de Ingeniería en Tecnología de Software, distribuidos en 64 integrantes del grupo control y 64 del grupo experimental. No se registraron pérdidas durante la aplicación de las seis tareas experimentales, por lo que todos los participantes fueron incluidos en el análisis final. La edad media fue de 21,8 ± 1,4 años en el grupo control y de 21,6 ± 1,5 años en el experimental, sin diferencias estadísticamente significativas ($p = 0,437$). La distribución por semestre fue semejante entre grupos. Tampoco se observaron diferencias en experiencia previa en

programación, experiencia en automatización de pruebas ni puntuación inicial de conocimientos, lo que evidenció condiciones basales comparables antes de la intervención (Tabla 1).

Tabla 1. Características iniciales de los participantes según grupo de estudio

Característica	Control (n = 64)	Experimental (n = 64)	p
Edad, años, media ± DE	21,8 ± 1,4	21,6 ± 1,5	0,437
Experiencia en programación, años	3,1 ± 1,2	3,0 ± 1,1	0,624
Experiencia en testing, años	1,6 ± 0,9	1,7 ± 0,8	0,508
Puntuación inicial, sobre 20	14,2 ± 2,1	14,4 ± 2,0	0,582
Sexto semestre, n	13	13	—
Séptimo semestre, n	13	12	—
Octavo semestre, n	13	13	—
Noveno semestre, n	12	13	—
Décimo semestre, n	13	13	—

Nota: DE: desviación estándar.

En relación con la eficiencia del proceso, el grupo experimental requirió una media de 36,2 ± 8,7 minutos para obtener una suite de pruebas válida, frente a 44,8 ± 9,6 minutos en el grupo control. La diferencia promedio fue de 8,6 minutos y resultó estadísticamente significativa ($p < 0,001$), con un tamaño del efecto grande ($d = 0,94$). Esto representó una reducción aproximada del 19,2 % en el tiempo requerido para completar las actividades de automatización.

El número medio de casos de prueba generados también fue superior con la estrategia asistida por IA/ML: 35,7 ± 8,1 frente a 28,4 ± 7,3 casos en el enfoque convencional ($p < 0,001$). Sin embargo, el incremento en cantidad estuvo acompañado por una mejora en la validez técnica. El 91,8 % de las pruebas del grupo experimental fueron compilables, comparado con el 85,1 % del grupo control. De manera similar, el porcentaje de pruebas ejecutadas correctamente alcanzó 89,6 % y 81,4 %, respectivamente (Tabla 2).

Las diferencias también se manifestaron en las métricas de efectividad. La cobertura media de líneas fue de 74,8 ± 8,1 % para el grupo control y de 82,6 ± 7,4 % para el grupo experimental, correspondiente a una diferencia de 7,8 puntos porcentuales ($p < 0,001$). Para la cobertura de ramas, los valores fueron de 66,1 ± 9,0 % y 75,4 ± 8,2 %, respectivamente, con un tamaño del efecto de 1,08.

La mayor diferencia se observó en el mutation score. Las suites desarrolladas mediante el procedimiento convencional eliminaron en promedio el 59,7 ± 10,2 % de los mutantes, mientras que las generadas mediante el flujo asistido alcanzaron 70,8 ± 9,1 %, equivalente a una diferencia de 11,1 puntos porcentuales ($p < 0,001$; $d = 1,15$). Esta diferencia indicó que el incremento de cobertura observado en el grupo experimental estuvo acompañado por una mayor capacidad de las pruebas para discriminar modificaciones artificiales introducidas en el código.

Tabla 2. Comparación de eficiencia y validez de las pruebas generadas

Indicador	Control	Experimental	p	d de Cohen
Tiempo por tarea, min	44,8 ± 9,6	36,2 ± 8,7	<0,001	0,94
Pruebas generadas, n	28,4 ± 7,3	35,7 ± 8,1	<0,001	0,95
Pruebas compilables, %	85,1 ± 9,2	91,8 ± 6,7	<0,001	0,83
Pruebas ejecutables, %	81,4 ± 10,4	89,6 ± 7,8	<0,001	0,89

Nota: Los valores se presentan como media ± desviación estándar.

De los seis defectos conocidos distribuidos en las tareas experimentales, el grupo control detectó una media de 4,2 ± 1,0, mientras que el grupo experimental identificó 5,0 ± 0,8. La diferencia fue estadísticamente significativa ($p < 0,001$) y presentó un tamaño del efecto grande ($d = 0,88$) (Tabla 3).

El análisis multivariable mantuvo la asociación después de controlar semestre académico, experiencia previa en programación, experiencia en testing y puntuación basal. La utilización del flujo asistido por IA/ML se relacionó con una reducción ajustada de 8,1 minutos en el tiempo de resolución y con incrementos de 7,4 puntos porcentuales en cobertura de líneas, 10,3 puntos en mutation score y 0,72 defectos detectados por participante. Todos los intervalos de confianza excluyeron el valor nulo (Tabla 4).

Tabla 3. Comparación de la efectividad de las suites de pruebas

Indicador	Control	Experimental	Diferencia	p	d
Cobertura de líneas, %	74,8 ± 8,1	82,6 ± 7,4	+7,8	<0,001	1,01
Cobertura de ramas, %	66,1 ± 9,0	75,4 ± 8,2	+9,3	<0,001	1,08
Mutation score, %	59,7 ± 10,2	70,8 ± 9,1	+11,1	<0,001	1,15
Defectos detectados, n	4,2 ± 1,0	5,0 ± 0,8	+0,8	<0,001	0,88

En conjunto, los resultados mostraron diferencias favorables al grupo experimental en todas las métricas principales. Las mayores magnitudes del efecto correspondieron al mutation score y a la cobertura de ramas, mientras que la reducción del tiempo y el incremento en la proporción de pruebas técnicamente válidas también presentaron efectos elevados. Por tanto, bajo las condiciones experimentales evaluadas, la hipótesis de ausencia de diferencias entre las estrategias fue rechazada.

Tabla 4. Asociación ajustada entre estrategia asistida por IA/ML y desempeño en las pruebas

Variable dependiente	β ajustado	IC 95 %	p
Tiempo de ejecución, min	-8,10	-11,21 a -4,99	<0,001
Cobertura de líneas, puntos porcentuales	+7,40	+4,82 a +9,98	<0,001
Mutation score, puntos porcentuales	+10,30	+7,15 a +13,45	<0,001
Defectos detectados, n	+0,72	+0,41 a +1,03	<0,001

DISCUSIÓN

Los resultados obtenidos evidenciaron un desempeño superior del grupo que utilizó inteligencia artificial y aprendizaje automático frente al grupo que trabajó con un esquema convencional de automatización. Las diferencias se observaron tanto en la reducción del tiempo requerido para construir suites válidas como en la cobertura, el mutation score y la cantidad de defectos identificados. Este comportamiento coincide con Moradi Dakhel et al.[17], quienes mostraron que la combinación de modelos de lenguaje y pruebas de mutación puede mejorar la generación de casos al utilizar los mutantes sobrevivientes como mecanismo de retroalimentación. En este sentido, el beneficio de la IA no parece limitarse a producir más código de prueba, sino a orientar progresivamente la generación hacia escenarios capaces de revelar comportamientos defectuosos.

La reducción cercana al 19 % en el tiempo requerido por el grupo experimental representa un hallazgo relevante desde la perspectiva de eficiencia. En procesos de desarrollo con ciclos de integración frecuentes, disminuir el tiempo destinado a construir y depurar casos de prueba puede permitir una retroalimentación más rápida al equipo de desarrollo. Rehan et al.[18]señalan que los modelos de lenguaje pueden incrementar la escalabilidad de la generación de pruebas al automatizar actividades que normalmente exigen intervención manual. Sin embargo, esta ventaja depende de que las respuestas generadas sean técnicamente aprovechables, ya que una reducción del tiempo pierde valor si posteriormente aumenta el esfuerzo necesario para corregir pruebas incorrectas.

En el presente estudio, el mayor número de pruebas generado por el grupo experimental estuvo acompañado por porcentajes superiores de compilación y ejecución. Este aspecto resulta importante porque la producción masiva de casos no constituye por sí misma una evidencia de calidad. Yang et al.[19], al evaluar grandes modelos de lenguaje para generación de pruebas unitarias, identificaron diferencias relevantes entre modelos y mostraron que el éxito depende no solo de la cantidad de pruebas producidas, sino también de su capacidad para compilar, ejecutarse correctamente y satisfacer objetivos de cobertura. Los resultados obtenidos siguen esta misma lógica, ya que el beneficio del enfoque asistido se manifestó simultáneamente en productividad y validez técnica.

La utilización de IA dentro de actividades de testing también requiere considerar el papel del usuario. Santos et al.[20] advierten que los grandes modelos de lenguaje pueden apoyar diversas tareas de pruebas de software, pero sus resultados deben ser evaluados críticamente debido a posibles errores, respuestas inconsistentes o casos aparentemente válidos que no verifican correctamente el comportamiento esperado.

En el contexto universitario estudiado, esta consideración adquiere especial importancia. La herramienta debe funcionar como apoyo al razonamiento del estudiante y no como sustituto de las competencias necesarias para interpretar el código, seleccionar escenarios y verificar las aserciones generadas.

Las diferencias observadas en cobertura de líneas y ramas también fueron favorables al grupo experimental. No obstante, la cobertura estructural no debería interpretarse de forma aislada. Lukasczyk et al.[21] demostraron, en un estudio empírico sobre generación automatizada de pruebas para Python, que el rendimiento de los generadores varía considerablemente entre proyectos y que alcanzar determinados niveles de cobertura no implica necesariamente que las pruebas sean igualmente efectivas para descubrir defectos. Esto justifica que en el presente estudio se incorporaran métricas complementarias y no se utilizara la cobertura como único indicador de calidad.

La evolución de las herramientas automatizadas confirma además que la efectividad depende del entorno, del lenguaje y de la estructura del software analizado. Pynguin constituye un ejemplo de cómo la generación automática puede adaptarse a ecosistemas distintos mediante estrategias específicas de búsqueda y análisis de código22. En los resultados obtenidos, el incremento de cobertura del grupo experimental probablemente se relacionó con una exploración más amplia de combinaciones de entradas y caminos de ejecución. Sin embargo, este comportamiento deberá verificarse en futuras investigaciones con proyectos de mayor tamaño y con arquitecturas más cercanas a sistemas industriales.

Otro resultado relevante fue el aumento del mutation score. Este indicador permite valorar si una suite es capaz de detectar modificaciones introducidas deliberadamente en el programa. Tufano et al.23mostraron que los modelos preentrenados pueden generar aserciones adecuadas para pruebas unitarias a partir del contexto del código, lo cual puede contribuir a fortalecer la capacidad de una suite para identificar comportamientos incorrectos. En el experimento desarrollado, el incremento del mutation score sugiere que las pruebas asistidas por IA no solamente recorrieron más instrucciones, sino que también incorporaron verificaciones con mayor capacidad para discriminar cambios en el comportamiento del software.

Los hallazgos también pueden interpretarse a partir de la evolución histórica de la generación automática. EvoSuite demostró que las técnicas basadas en búsqueda pueden producir suites orientadas a maximizar determinados criterios de cobertura en software orientado a objetos [24]. Frente a esta línea tradicional, los enfoques actuales basados en IA añaden la posibilidad de utilizar conocimiento aprendido a partir de grandes volúmenes de código y generar soluciones condicionadas por el contexto del método. Los resultados de esta investigación no implican que las estrategias clásicas hayan perdido utilidad; por el contrario, sugieren que la integración de técnicas convencionales y modelos inteligentes puede resultar más eficaz que utilizar cualquiera de ellas de forma aislada.

La relación entre cobertura y detección de fallos merece una interpretación adicional. Chen et al.25demostraron que cobertura, tamaño del conjunto de pruebas y capacidad de detección no mantienen una relación simple ni necesariamente proporcional. En consecuencia, el aumento de cobertura observado en el grupo experimental adquiere mayor significado porque estuvo acompañado por mejoras tanto en el mutation score como en el número de defectos detectados. Esta convergencia entre varias métricas proporciona evidencia más consistente sobre la efectividad del enfoque asistido que la obtenida mediante un único indicador.

Desde la perspectiva de adopción, los resultados también respaldan la necesidad de seleccionar cuidadosamente qué actividades conviene automatizar. Garousi y Mäntylä26plantean que la conveniencia de automatizar depende de factores como repetibilidad, frecuencia de ejecución, costo de mantenimiento y características del proceso de pruebas. En el escenario analizado, la IA mostró ventajas especialmente en generación inicial, refinamiento y exploración de casos, mientras que la validación de las pruebas continuó requiriendo supervisión humana. Por ello, su incorporación en la enseñanza de ingeniería de software debería orientarse hacia esquemas híbridos en los que el estudiante utilice herramientas inteligentes, pero mantenga responsabilidad sobre la evaluación de sus resultados.

El estudio presenta algunas limitaciones. La investigación se desarrolló en una sola facultad y con estudiantes universitarios, por lo que los resultados no pueden generalizarse directamente a desarrolladores profesionales ni a empresas mexicanas de software. Asimismo, se emplearon tareas controladas y módulos previamente seleccionados, situación diferente a proyectos empresariales de larga duración y mayor complejidad. El desempeño también podría variar según el modelo de IA, el lenguaje de programación, el tipo de defecto y la forma en que se construyen las instrucciones. Futuras investigaciones deberían replicar el experimento en otras universidades mexicanas,

incorporar profesionales de la industria y comparar diferentes modelos generativos y técnicas de priorización.

En conjunto, los hallazgos sugieren que la integración de inteligencia artificial y aprendizaje automático puede mejorar la eficiencia y efectividad de la automatización de pruebas cuando se utiliza dentro de un proceso supervisado. La reducción del tiempo, el incremento de cobertura, la mejora del mutation score y la mayor detección de defectos respaldan su potencial como herramienta complementaria. Para la formación universitaria en México, el principal desafío será aprovechar estas capacidades sin disminuir la comprensión técnica del estudiante ni convertir la generación automática en un sustituto del análisis crítico.

CONCLUSIONES

La automatización de pruebas de software asistida por inteligencia artificial y aprendizaje automático mostró un desempeño superior comparado al uso de la técnica tradicional en las métricas que fueron consideradas en el experimento. Esto es, el grupo experimental necesitó menos tiempo en la construcción de suites válidas y, se obtuvieron mejores resultados en lo que se refiere a cobertura, ejecutabilidad, mutation score y detección de defectos, así que se demuestra que estas pruebas ayudan notablemente al testing.

La ventaja que se observaba no era solo la de proporcionar una mayor cantidad de casos de prueba, sino también la de generar pruebas más técnicamente útiles. El incremento simultáneo de los valores de la cobertura y del mutation score sugiere que la IA pueda ayudar a explorar un mayor número de rutas del programa, así como a construir las verificaciones que serán capaces de identificar un mayor número de posibles modificaciones defectuosas del comportamiento del software.

La Inteligencia Artificial en la formación de estudiantes del Engineering in Software Technology podría contribuir a la formación de competencias en automatización, análisis de la calidad o evaluación de defectos. Sin embargo, tal utilización bien podría realizarse como asistencia supervisada, dado que las pruebas generadas con esta ayuda automáticamente requieren ser revisadas, interpretadas y validadas por el usuario en el momento en que se incorporan a un proceso formal de aseguramiento de la calidad del software.

La Facultad de Ingeniería Mecánica y Eléctrica de la Universidad Autónoma de Nuevo León sostiene la oportunidad de introducir testing asistido por IA en las asignaturas del engineering del software a partir de la apropiación de prácticas tradicionales de diseño que combinan con criterios de evaluación de un modo objetivo. Tal combinación acercaría la enseñanza superior a las transformaciones del sector de desarrollo de software.

Los hallazgos de la investigación no deben trasladarse sin más a otros escenarios profesionales. Se sugiere ampliar el estudio a otras universidades mexicanas, equipos profesionales y proyectos de mayor complejidad, así como hacer comparaciones entre distintos modelos de IA y estrategias de aprendizaje automático en el contexto del servicio de atención al cliente para completar adecuadamente la foto. El objetivo de estas líneas es detallar más las condiciones bajo las cuales la automatización inteligente permite generar ventajas sostenibles, y bajo las cuales son preferibles las metodologías no automáticas.

Referencias

1. Martins G, Tenório N, Bernardino J. AI-driven software testing: a review. Big Data Cogn Comput. 2026;10(7):233. https://doi.org/10.3390/bdcc10070233
2. Ajorloo S, Jamarani A, Kashfi M, Haghi Kashani M, Najafizadeh A. A systematic review of machine learning methods in software testing. Appl Soft Comput. 2024;162:111805. https://doi.org/10.1016/j.asoc.2024.111805
3. Fontes A, Gay G. The integration of machine learning into automated test generation: a systematic mapping study. Softw Test Verif Reliab. 2023;33(4). https://doi.org/10.1002/stvr.1845
4. Rafi DM, Moses KRK, Petersen K, Mäntylä MV. Benefits and limitations of automated software testing: systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test. 2012. p. 36-42. https://doi.org/10.1109/IWAST.2012.6228988
5. Pan R, Bagherzadeh M, Ghaleb TA, Briand L. Test case selection and prioritization using machine learning: a systematic literature review. Empir Softw Eng. 2022;27(2):29. https://doi.org/10.1007/s10664-021-10066-6
6. Spieker H, Gotlieb A, Marijan D, Mossige M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017. p. 12-22. https://doi.org/10.1145/3092703.3092709
7. Bertolino A, Guerriero A, Miranda B, Pietrantuono R, Russo S. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020. p. 1261-1272. https://doi.org/10.1145/3377811.3380369
8. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. 2011. p. 416-419. https://doi.org/10.1145/2025113.2025179
9. Lukasczyk S, Fraser G. Pynguin: automated unit test generation for Python. In: 44th International Conference on Software Engineering Companion. 2022. p. 168-172. https://doi.org/10.1145/3510454.3516829
10. Zhang YW, Jin Z, Wang ZJ, Xing Y, Li G. SAGA: summarization-guided assert statement generation. J Comput Sci Technol. 2025;40(1):138-157. https://doi.org/10.1007/s11390-023-2878-6
11. Dinella E, Ryan G, Mytkowicz T, Lahiri SK. TOGA: a neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering. 2022. p. 2130-2141. https://doi.org/10.1145/3510003.3510141
12. Alagarsamy S, Tantithamthavorn C, Aleti A. A3Test: assertion-augmented automated test case generation. Inf Softw Technol. 2024;176:107565. https://doi.org/10.1016/j.infsof.2024.107565
13. Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMOSA: escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. 2023. p. 919-931. https://doi.org/10.1109/ICSE48619.2023.00085
14. Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q. Software testing with large language models: survey, landscape, and vision. IEEE Trans Softw Eng. 2024;50(4):911-936. https://doi.org/10.1109/TSE.2024.3368208
15. Li Y, Liu P, Wang H, Chu J, Wong WE. Evaluating large language models for software testing. Comput Stand Interfaces. 2025;93:103942. https://doi.org/10.1016/j.csi.2024.103942
16. Tasarsu M, Tokmak AV, Catal C. Test case generation using large language models: a systematic literature review. Cluster Comput. 2026;29:227. https://doi.org/10.1007/s10586-026-06021-z
17. Moradi Dakhel A, Nikanjam A, Majdinasab V, Khomh F, Desmarais MC. Effective test generation using pre-trained large language models and mutation testing. Inf Softw Technol. 2024;171:107468. https://doi.org/10.1016/j.infsof.2024.107468
18. Rehan S, Al-Bander B, Al-Said Ahmad A. Harnessing large language models for automated software testing: a leap towards scalable test case generation. Electronics. 2025;14(7):1463. https://doi.org/10.3390/electronics14071463
19. Yang L, Yang C, Gao S, Wang W, Wang B, Zhu Q, et al. On the evaluation of large language models in unit test generation. In: 2024 39th IEEE/ACM International Conference on Automated Software Engineering. 2024. p. 1607-1619. https://doi.org/10.1145/3691620.3695529
20. Santos R, Santos I, Magalhães CVC, Santos RS. Are we testing or being tested? Exploring the practical applications of large language models in software testing. In: 2024 IEEE Conference on Software Testing, Verification and Validation. 2024. p. 353-360. https://doi.org/10.1109/ICST60714.2024.00039
21. Lukasczyk S, Kroiß F, Fraser G. An empirical study of automated unit test generation for Python. Empir Softw Eng. 2023;28:36. https://doi.org/10.1007/s10664-022-10248-w
22. Lukasczyk S, Kroiß F, Fraser G. Automated unit test generation for Python. In: Search-Based Software Engineering. Lecture Notes in Computer Science. 2020;12420:9-24. https://doi.org/10.1007/978-3-030-59762-7_2
23. Tufano M, Drain D, Svyatkovskiy A, Sundaresan N. Generating accurate assert statements for unit test cases using pretrained transformers. In: IEEE/ACM International Conference on Automation of Software Test. 2022. p. 54-64. https://doi.org/10.1145/3524481.3527220
24. Liu XJ, Yu P, Ma XX. An empirical study on automated test generation tools for Java: effectiveness and challenges. J Comput Sci Technol. 2024;39(3):715-736. https://doi.org/10.1007/s11390-023-1935-5
25. Chen YT, Gopinath R, Tadakamalla A, Ernst MD, Holmes R, Fraser G, et al. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 2020. p. 237-249. https://doi.org/10.1145/3324884.3416667
26. Garousi V, Mäntylä MV. When and what to automate in software testing? A multi-vocal literature review. Inf Softw Technol. 2016;76:92-117. https://doi.org/10.1016/j.infsof.2016.04.015

Financiación

Ninguno.

Conflictos de interés

No existen.

Contribución de autoría

Conceptualización: Cecilia Muñoz Oceguera, Carlos Hernández Salas y Carmen Carolina Ortega Hernández.
Curación de datos: Cecilia Muñoz Oceguera y Carmen Carolina Ortega Hernández.
Análisis formal: Cecilia Muñoz Oceguera, Carlos Hernández Salas y Carmen Carolina Ortega Hernández.
Investigación: Cecilia Muñoz Oceguera, Carlos Hernández Salas y Carmen Carolina Ortega Hernández.
Metodología: Cecilia Muñoz Oceguera y Carmen Carolina Ortega Hernández.
Supervisión: Carlos Hernández Salas.
Validación: Carlos Hernández Salas y Carmen Carolina Ortega Hernández.
Visualización: Cecilia Muñoz Oceguera.
Redacción – borrador original: Cecilia Muñoz Oceguera y Carmen Carolina Ortega Hernández.
Redacción – revisión y edición: Carlos Hernández Salas y Carmen Carolina Ortega Hernández.