DOI: https://doi.org/10.36910/6775-2524-0560-2026-64-04
УДК 004.415.53
Чижевич Владислав Олексійович, аспірант
https://orcid.org/0009-0007-8030-7991
Луцький національний технічний університет, м. Луцьк, Україна

ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ЕТАПАХ ЖИТТЄВОГО ЦИКЛУ РОЗРОБКИ (SDLC): АНАЛІТИЧНИЙ ОГЛЯД

Чижевич В. О. Використання штучного інтелекту для автоматизації тестування програмного забезпечення на етапах життєвого циклу розробки (SDLC): аналітичний огляд. У статті було проведено аналіз і огляд сучасних досліджень та практичних методів застосування штучного інтелекту (ШІ) для автоматизації тестування програмного забезпечення (ПЗ). Предмет аналізу — використання методів машинного навчання (ML), глибокого навчання (DL) та великих мовних моделей (LLM) для автоматизації процесу тестування і перевірки якості на різних етапах життєвого циклу розробки (SDLC), а саме на етапах розробки, тестування, розгортання та підтримки ПЗ. Було детально проаналізовано 37 джерел, пов'язаних з темою роботи. Для кожного з джерел описано і проаналізовано проблему, яка розглядалася, і отримані в ньому результати. Були виділені ключові підходи і методи, зокрема генерація тестів на основі LLM, гібридизація з пошуковим тестуванням (SBST), автоматичне рев'ю коду, передбачення дефектів, інтелектуальна пріоритизація тестів у CI/CD та self-healing автоматизація. Наведено порівняння і кількісні метрики ефективності інструментів, визначено відкриті проблеми в цій області.
Ключові слова: штучний інтелект, автоматизація тестування, великі мовні моделі, генерація тестів, SDLC, CI/CD, self-healing тестування, передбачення дефектів.

Chyzhevych V. Use of artificial intelligence for software testing automation across the software development lifecycle (SDLC): an analytical review. The article analyzes and reviews modern research and practical methods for applying artificial intelligence (AI) to automate software testing (SW). The subject of the analysis is the use of machine learning (ML), deep learning (DL), and large language models (LLM) methods to automate the testing and quality assurance process at different stages of the software development life cycle (SDLC), namely at the stages of software development, testing, deployment, and maintenance. 37 sources related to the work's topic were analyzed in detail. For each source, the problem under consideration is described and analyzed, and the results obtained are described. Key approaches and methods are highlighted, including LLM-based test generation, hybridization with search testing (SBST), automatic code review, defect prediction, intelligent test prioritization in CI/CD, and self-healing automation. Comparisons and quantitative metrics of the effectiveness of the tools are presented, and open problems in this area are identified.
Keywords: artificial intelligence, test automation, large language models, test generation, SDLC, CI/CD, self-healing testing, defect prediction.

Постановка наукової проблеми. Розробка ПЗ в сучасному світі відбувається дуже швидко, що провокує стрімке збільшення обсягу написаного коду, короткі й часті релізи та перехід до складних архітектур (мікросервіси, хмарні застосунки, розподілені системи). Це все робить традиційне тестування, як мануальне так і за написаними сценаріями, з використанням різних тестових інструментів, недостатньо гнучким і швидким. Галузеві та профільні дослідження оцінюють, що такий підхід до тестування займає до 40% витраченого часу на розробку, а кількість дефектів, з якими випускається ПЗ, може сягати 15% [1]. Разом з тим, опитування фахівців показують, що приблизно 70% респондентів вважають, що якість застосунків погіршилася через прискорену розробку і генерацію коду за допомогою ШІ, а 68% респондентів відповіли, що такий підхід до розробки робить саме етап тестування вузьким місцем [2].

Щоб відповідати вимогам, активно почав розвиватися напрямок Artificial Intelligence for Software Engineering (AI4SE) у сфері, що стосується тестування. AI-driven test automation використовує методи машинного навчання, глибокого навчання, обробки природної мови (NLP) та великих мовних моделей, щоб автоматизувати тестування на всіх можливих етапах SDLC. Це в свою чергу включає в себе широкий спектр функцій, від генерації тестів на етапі написання коду до підтримки стабільності у продакшн-середовищі [3]. Тому забезпечення якості ПЗ в умовах, коли швидкість створення значно перевищує швидкість тестування і перевірки , є важливим практичним завданням, а створення і побудова моделей, що узгоджують використання ШІ з етапами життєвого циклу розробки ПЗ, є цінним науковим завданням. Відповідно, систематизація цієї сфери і напряму це важливий початковий етап.

Аналіз останніх досліджень і публікацій. При написанні даної статті були взяті дослідження і публікації з наступних ресурсів: arXiv (розділ cs.SE), IEEE Xplore, ACM Digital Library, ScienceDirect та ResearchGate. Пошук здійснювався за такими ключовими словами: AI-

driven test automation, LLM unit test generation, AI code review, defect prediction, self-healing test automation, test case prioritization CI, AI4SE, search-based software testing. Пріоритет у пошуку надавався статтям, які були опубліковані у 2023–2026 роках, коли активно почав застосовуватися генеративний ШІ. Головні критерії відбору джерел: рецензовані статті з високим рівнем цитування, безпосередня пов'язаність з автоматизацією тестування ПЗ засобами ШІ, наявність емпіричних даних (метрик покриття, показників якості) або структурованого огляду літератури. В результаті було відібрано 37 джерел для аналізу, які були поділені на сім категорій відповідно до етапу SDLC та ролі ШІ.

Основними для напряму є узагальнювальні огляди. У найповнішому на сьогодні огляді «AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges» [3] було проаналізовано 76 галузевих інструментів і наукових робіт та запропоновано lifecycle-орієнтовану класифікацію, що розподіляє AI-підходи за етапами тестового циклу: планування, генерація, виконання, виправлення та підтримка. Також окремо виділено технологічні складові, серед яких класичне машинне навчання, глибоке навчання, великі мовні моделі та навчання з підкріпленням (RL). В статті «The Future of Software Testing: AI-Powered Test Case Generation and Validation» [4] проаналізовано інтеграцію ШІ у генерацію та валідацію тестових кейсів і проблеми ручного тестування, які воно вирішує: тривалі терміни виконання, можливість людської помилки та неповне покриття функціоналу. Також розглянуто використання ШІ з точки зору можливої трансформації, а не тільки для покращення існуючих інструментів і підходів. У дослідженні AI-Native SDLC [5] зазначається, що ефективність створення тестових кейсів з використанням систем, які базуються на ChatGPT-подібних моделях,у деяких випадках перевищує 70%. Також виявлено, що зі зростанням обсягів згенерованого коду приблизно 50% компаній мали щонайменше один інцидент протягом року, пов'язаний з безпекою. В роботі «Breaking Barriers in Software Testing: The Power of AI-Driven Automation» [1] задокументовано, які економічні фактори підштовхують до впровадження ШІ для тестування. Галузеві опитування [2, 6] показують наявність практичного запиту на використання ШІ: близько 34% організацій застосовують його для підвищення якості тестування. Основними проблемами для впровадження ШІ є можливі витоки конфіденційних даних (58%), складність інтеграції (55%) та неточність і помилки моделей (47%).

Були розглянуті статті, пов'язані з генерацією unit-тестів [7–14], гібридизацією LLM із пошуковим тестуванням [15–20], застосуванням ШІ для рев'ю коду та передбачення дефектів [21–26], пріоритизацією тестів у CI [27–31], self-healing автоматизацією [32–35] та індустріальним контекстом [36, 37]. Оскільки темою роботи є аналітичний огляд, детальний розбір кожної з цих статей наведено в основній частині.

Виділення невирішених раніше частин загальної проблеми. Незважаючи на активний розвиток напрямку, аналіз літератури виявляє низку невирішених частин загальної проблеми, які описані в цій статті. По-перше, більшість наявних досліджень зосереджена на окремих етапах SDLC, тоді як наскрізні підходи, що охоплюють увесь процес, починаючи від розробки до підтримки застосунків у реальних умовах, залишаються слабо описаними і проаналізованими [1]. В таких роботах немає узагальненої класифікаційної моделі, що узгоджувала б використання ШІ на різних етапах життєвого циклу. По-друге, оцінки ефективності LLM-генерації тестів у різних дослідженнях суперечать й відрізняються на порядок залежно від методології й способу вимірювання [19], що ускладнює їх пряме порівняння. По-третє, велика частина тестів, згенерованих через LLM, є нефункціональною або містить галюцинації[12], а якісні показники не враховуються при систематичному оцінюванні [13]. Також кількісні дані щодо ефективності self-healing-тестування походять переважно від постачальників комерційних рішень і не мають незалежної верифікації [33, 35]. Таким чином, існує потреба в систематизованому огляді, що об'єднав би розрізнені результати в єдину lifecycle-орієнтовану модель із детальним аналізом кожного джерела.

Формулювання мети дослідження. Метою статті є огляд і систематизація моделей, методів та інструментів, що використовують ШІ для автоматизації тестування ПЗ. Акцент в роботі зроблений на етапі розробки ПЗ, оскільки саме на цьому етапі найбільше використовується ШІ.

Для досягнення мети сформульовано такі питання:

1. Як класифікується застосування ШІ в тестуванні ПЗ відповідно до етапів SDLC?
2. Які моделі та методи ШІ використовуються на етапі розробки — зокрема для генерації тестів, рев'ю коду та передбачення дефектів?
3. Які підходи ШІ застосовуються на етапі інтеграції та безперервного тестування (CI/CD)?

4. Як ШІ застосовується на етапі підтримки застосунків у реальних умовах (self-healing, GUI-тестування)?
5. Які обмеження та відкриті проблеми характерні для цього напрямку?

Виклад основного матеріалу й обґрунтування отриманих результатів дослідження.

1. Термінологія та класифікаційна модель

Широко розповсюдженим поняттям для позначення застосування ШІ в задачах програмної інженерії є термін AI4SE (Artificial Intelligence for Software Engineering). У межах AI4SE окремим напрямком виступає AI-driven test automation. AI-driven test automation відповідає за використання ML/DL/LLM-моделей для автоматизації одного чи кількох етапів тестувального циклу, наприклад, планування тестів, генерації тестових кейсів, виконання тестів та їх підтримки [1].

Для структуризації матеріалу в цій статті використано lifecycle-орієнтовану класифікаційну модель, що узгоджує етапи SDLC з відповідними задачами тестування та типовими ШІ-підходами (Рис. 1).

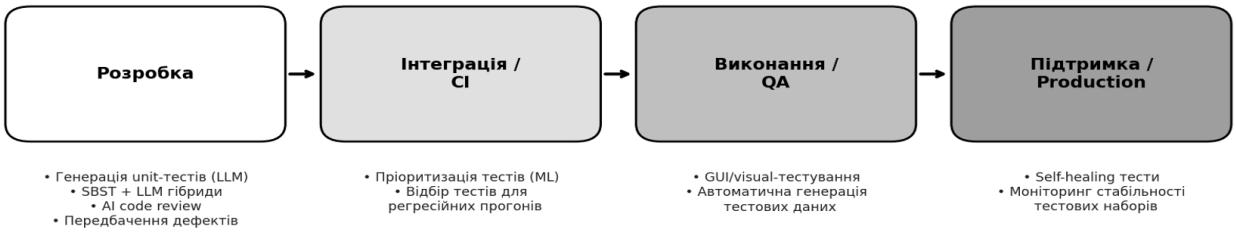

Рис. 1. Застосування ШІ в автоматизації тестування ПЗ на етапах SDLC

Ця модель є основою для розбиття подальшого матеріалу на окремі частини. Кожна частина відповідає одному з блоків. Кожне з джерел розглядається окремо, вказуються метод, ключові результати і обмеження для того, щоб забезпечити максимально детальний огляд.

2. Загальна картина: AI-driven test automation як напрямок

У найповнішому на сьогодні огляді «AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges» [3] автори проаналізували 76 інструментів і наукових робіт та запропонували lifecycle-орієнтовану класифікацію, що розподіляє AI-техніки за етапами тестового циклу. Вона поділяє їх на планування, генерацію, виконання та підтримку. Окремо виділили технологічні складові напрямку, такі як класичне машинне навчання, глибоке навчання, великі мовні моделі та навчання з підкріпленням (RL). Також автори наголосили на пояснювальному ШІ (XAI) у контексті тестування та NLP-орієнтованого скриптингу тестів, які залишаються малодослідженими напрямками. Автори підкреслюють, що більшість наявних робіт зосереджена на окремих етапах циклу, тоді як наскрізні підходи досі залишаються малопоширеною практикою.

Робота «The Future of Software Testing: AI-Powered Test Case Generation and Validation» [4] присвячена аналізу інтеграції ШІ у генерацію та валідацію тестових кейсів. У статті описано вирішення традиційних проблем ручного тестування, а саме: тривалі терміни виконання, людський фактор і неповне покриття функціоналу. Автори підкреслюють, що традиційні методи, попри розвиток, усе ще значно відстають від швидкості розробки. Автори розглядають ШІ як підхід до подолання цього розриву і можливість для трансформування .

«The AI-Native Software Development Lifecycle: A Theoretical and Practical New Methodology» [5]. У даній роботі в розділі, присвяченому тестуванню та забезпеченню якості в межах повністю «AI-нативного» SDLC, автори зазначають, що ефективність генерації тестових наборів системами на основі ChatGPT-подібних моделей перевищує 70% у деяких дослідженнях. Але також наголошується на проблемах, які провокує такий підхід. Зі зростанням обсягів AI-згенерованого коду близько 50% організацій повідомляють щонайменше про один інцидент безпеки протягом 12 місяців. Це прямо вказує на потребу в посиленні перевірки якості та тестування ПЗ.

Дослідження «Breaking Barriers in Software Testing: The Power of AI-Driven Automation» [1] детально розглядає економічні передумови впровадження ШІ в тестування. Наприклад, мануальне тестування може споживати до 40% витрат на розробку, а частка дефектів, що не виявляються до релізу, у складних системах може досягати 15%. Автори показують, що ці проблеми загострюються з переходом до мікросервісів і хмарних застосунків. Такі архітектурні рішення, де обсяг коду та кількість залежностей між компонентами є значними, вимагають глибшого тестування, ніж може

забезпечити ручне тестування. ШІ розглядається як інструмент, що використовує машинне навчання, обробку природної мови та адаптивні алгоритми для автоматизації тестування саме в умовах підвищеної складності.

В опитуванні «SmartBear. AI Software Quality Gap Report» [2] було задіяно 273 керівників у сфері тестування та якості ПЗ (січень 2026 р.). За результатами опитування 70% респондентів вважають, що якість застосунків уже погіршується через прискорення розробки засобами ШІ. А 68% респондентів побоюються виникнення вузьких місць у тестуванні. У звіті вводиться поняття цілісності застосунку. Це означає зміщення фокусу з питання функціональної роботи коду на питання відповідності реалізації і досвіду використання заданим вимогам. Це, своєю чергою, передбачає безперервний, вимірюваний контроль відповідності зі швидкістю, властивою AI-розробці.

«Sogeti / Capgemini / OpenText / Coleman Parkes» [6]. Глобальне опитування 1775 керівників IT- та бізнес-напрямків показало, що 71% організацій уже інтегрували певну форму ШІ чи генеративного ШІ у свою діяльність. Причому 34% застосовують ШІ безпосередньо для покращення якості тестування, ще 34% розробили план впровадження після тестового застосування, а 19% усе ще перебувають на етапі випробовування. Серед головних перешкод для впровадження ШІ респонденти назвали занепокоєння щодо витоку даних (58%), складність інтеграції (55%), обсяг необхідних ресурсів (53%), проблему галюцинацій моделей (47%) та непередбачувані витрати (43%).

3. Етап розробки: генерація unit-тестів за допомогою LLM

Це найбільш активно досліджуваний напрямок застосування ШІ в тестуванні на етапі розробки, тому в цьому розділі розглянуто найбільшу кількість джерел.

«Schäfer M. et al. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation»[7] є однією з найбільших робіт в даному напрямку. Автори застосували модель GPT-3.5-turbo без додаткового навчання чи few-shot прикладів для автоматичної генерації unit-тестів для JavaScript API. У промпт включаються сигнатура функції, її реалізація та приклади використання з документації, а в разі невдалого прогону тест ітеративно змінюється через повторний запит моделі з текстом помилки. На вибірці з 25 npm-пакетів і 1684 API-функцій TestPilot досяг медіанного покриття операторів 70,2% і покриття гілок 52,8%. це значно вище за попередній feedback-directed інструмент Nessie (51,3% і 25,6% відповідно; Рис. 2). Показово, що 92,8% згенерованих тестів мали не більше 50% подібності до наявних тестів, причому жоден не був точною копією. Також додаткові експерименти з іншими LLM (code-cushman-002 — 68,2%; StarCoder — 54,0%) показали, що ефективність підходу залежить від розміру й якості навчання моделі, але не є фундаментально прив'язаною до конкретної LLM.

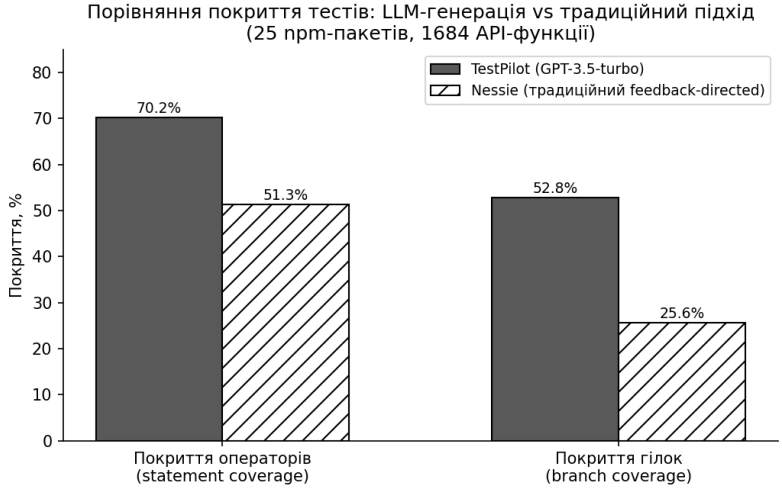

Рис. 2. Порівняння покриття тестів TestPilot (LLM) та Nessie (традиційний підхід) [7]

«TestART: Improving LLM-based Unit Test via Co-evolution of Automated Generation and Repair» [8].У статті розглядають систему, що впроваджує коеволюційний цикл автоматизованої генерації та виправлення тестів. Подібно до TestPilot, у промпт включаються текст тесту й повідомлення про помилку для усунення несправностей. Але процес виправлення побудований як

ітеративна коеволюція, а не як одноразовий запит. У порівнянні з базовим підходом ChatUniTest TestART демонструє ефективніше використання сильних сторін LLM. Разом з тим, він пом'якшує їхні слабкі сторони, що в результаті дає надійніші й якісніші тести.

Дослідження «An Empirical Study of Unit Test Generation with Large Language Models» [9] систематизує й порівнює кілька конкуруючих підходів до LLM-генерації тестів. Порівнюються ChatTester, який застосовує ланцюжок міркувань для генерації тестів, ChatUniTest , який використовує процес генерації на основі зворотного зв'язку, підхід Siddiq, TestPilot Schäfer та роботу Dakhel. Порівняльний аналіз цих підходів у єдиних умовах дозволяє виявити, які компоненти промпту роблять найбільший внесок у якість згенерованих тестів. Серед таких можна виділити документацію, приклади використання, зворотний зв'язок про помилки.

«A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites (AgoneTest)»[10]. На відміну від TestPilot, обмеженого 25 репозиторіями, AgoneTest забезпечує значно ширшу застосовність завдяки використанню вибірки з 9410 GitHub-репозиторіїв. Ці репозиторії мають автоматичну інтеграцію тестових бібліотек у кожен з них. Система пропонує структурований механізм для оцінки й порівняння кількох LLM та технік промпт-інжинірингу в межах єдиного уніфікованого фреймворку. На відміну від підходу Cedar, що спирається на одну стратегію few-shot-навчання з моделлю Codex і не передбачає порівняння кількох технік. Окремо в межах цього дослідження проаналізовано вплив варіювання гіперпараметрів GPT-3.5-turbo на якість згенерованих тестів.

«Parameter-Efficient Fine-Tuning of Large Language Models for Unit Test Generation: An Empirical Study» [11]. Дослідження проводить масштабну емпіричну оцінку ефективності LLM для автоматичної генерації unit-тестів без додаткового навчання чи ручних зусиль. Порівнюючи Codex, GPT-3.5-turbo та StarCoder на датасетах HumanEval і SF110 за показниками частоти успішної компіляції, коректності тестів, покриття та наявності test smells. Окремо досліджується параметро-ефективне донавчання (PEFT) як альтернатива zero-shot промптингу. Результати показують, що навіть невелике донавчання здатне суттєво покращити якість згенерованих тестів. Це є важливим практичним висновком для команд з обмеженими обчислювальними ресурсами.

Автори статті «PR-Aware Automated Unit Test Generation: Challenges and Opportunities» [12] документують ключову проблему одноразового промпту до LLM для генерації тестів. Значна частка згенерованих тестів виявляється некоректно сформованою або взагалі не компілюється. Це виникає переважно через галюциновані залежності моделі та некоректно сформульовані твердження. Такі помилки в тестових налаштуваннях і результатах суттєво обмежують практичну корисність тестів, згенерованих без додаткового циклу виправлення. Саме ця проблема мотивує розробку вдосконалених інструментів на кшталт TestART і ChatUniTest, що включають зворотний зв'язок та ітеративність.

«Test smells in LLM-Generated Unit Tests»[13] є одним з найбільш ретельних емпіричних досліджень якості згенерованих тестів. Було проаналізовано бенчмарк із 20 500 тестових наборів, згенерованих чотирма моделями (GPT-3.5, GPT-4, Mistral 7B, Mixtral 8x7B) за п'ятьма техніками промпт-інжинірингу. Також проаналізовано 780 144 тестових наборів, написаних людьми, для 34 637 проєктів. За допомогою інструмента TsDetect, що виявляє test smells, встановлено, що LLM систематично відтворюють типові антипатерни тестування. Найчастіше такими патернами є Magic Number Test і Assertion Roulette. Причому окремі антипатерни мають тенденцію співіснувати в одному тестовому наборі залежно від застосованої техніки промптингу. В роботі роблять висновок, що високе покриття коду саме по собі не гарантує якості автоматично згенерованого тесту, і оцінку варто доповнювати аналізом test smells.

«A Survey on Large Language Models for Software Engineering (розділ 4.3.4 Unit Test Generation)» [14]. У межах цього масштабного огляду систематизовано підходи до LLM-генерації unit-тестів за двома напрямками. Підходи на основі донавчання: AthenaTest — перша робота із застосуванням LLM для генерації тестів; A3Test — попередньо навчає PLBART розпізнавати твердження в режимі самонавчання; CAT-LM — модель у стилі GPT, спеціально навчена картографувати відповідність між кодом і його тестами. Підходи на основі промптингу: TestRefineGen — генерує тести за текстовими описами з урахуванням конфіденційності даних; ChatUniTest — реалізує схему «генерація – валідація – виправлення; CasModaTest — розбиває задачу генерації тесту на два каскадні підзавдання: генерацію тестового префіксу та генерацію тестового оракула; CODAMOSA — інтегрує Codex із класичним пошуковим тестуванням (SBST) для генерації Python-тестів.

4. Етап розробки: гібридизація LLM із пошуковим тестуванням (SBST)

«EvoSuite: automatic test suite generation for object-oriented software»[18]. EvoSuite є одним з найкращих інструментів пошукового тестування (SBST) для Java. Він приймає на вхід клас або метод, застосовує еволюційні алгоритми для генерації тестового набору, що відповідає заданим критеріям покриття, наприклад, лінійним, гіллястим тощо. У останніх версіях за замовчуванням застосовується алгоритм DynaMOSA. Він формулює задачу пошуку як багатоцільову оптимізацію одночасно за всіма цілями покриття, а не як послідовну оптимізацію окремих цілей. Головним практичним недоліком EvoSuite, неодноразово підтвердженим іншими дослідженнями, є низька читабельність згенерованих тестів та обмежена підтримка сучасних версій Java.

У статті «Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests (UTGen)» [15] автори описують поєднання SBST і LLM для розв'язання проблеми читабельності. Пошуковий алгоритм в такому випадку відповідає за досягнення покриття. LLM, своєю чергою, контекстуалізує тестові дані, покращує іменування ідентифікаторів і додає описові коментарі. Такий поділ відповідальності дозволяє зберегти сильну сторону SBST (систематичне досягнення покриття) і водночас усунути його головний недолік (незрозумілість тесту для людини).

Автори «EvoGPT: Leveraging LLM-Driven Seed Diversity to Improve Search-Based Test Suite Generation» [16] розвивають ідею гібридизації в іншому напрямку. LLM генерує початкову різноманітну популяцію тестів, яку потім оптимізує еволюційний алгоритм. Кожен LLM-згенерований тест додатково вдосконалюється через цикл генерації-виправлення із покриттям генерацією тверджень. Особливістю EvoGPT є здатність автоматично виявляти «плато» покриття та додавати додаткові LLM-згенеровані тести, спрямовані саме на непокриті гілки коду, для подолання цього застою.

«Improving the Readability of Automatically Generated Tests using Large Language Models.» [17]. Огляд систематизує SBST-інструменти для різних типів тестування. EvoSuite залишається провідним інструментом для Java unit-тестів. Randoop є сильною базовою лінією на основі feedback-directed випадкового тестування. EvoMaster застосовується для системного тестування REST API. Sapienz для генерації Android GUI-тестів, а Pynquin для unit-тестів Python. Автори також наводять кількісний показник збільшення інтересу до теми: з січня 2019 року по жовтень 2023 року опубліковано понад 100 робіт про застосування LLM у тестуванні ПЗ, причому понад 80% лише за 2023 рік. Це ілюструє різке прискорення досліджень саме в цій вузькій темі.

Дослідження «Exploring Test Suite Coverage of Large Language Model–Enhanced Test Generation» [19] безпосередньо порівнює покриття UTGen і базового EvoSuite. Воно фіксує, що покращення читабельності через залучення LLM супроводжується лише незначною (близько 1%) зміною середнього покриття гілок. Тобто, підвищення зрозумілості тестів практично не впливає на втрати покриття. Водночас автори звертають увагу на суперечливість наявних даних щодо ефективності LLM-підходів без SBST. В одних дослідженнях зафіксовано приріст покриття до 80%, тоді як інші показують, що ті самі LLM без пошукового компонента досягають лише близько 2% покриття на еталонному датасеті EvoSuite SF110. Ця розбіжність підкреслює критичну залежність результатів від конкретного бенчмарку й методології оцінювання.

Робота «Automated Test Suite Enhancement Using Large Language Models with Few-shot Prompting» [20] спирається на класичні теоретичні засади пошукового тестування. Зокрема, дослідження щодо локального, глобального та гібридного пошуку, а також спостереження про те, що метрика покриття не завжди сильно корелює з реальною ефективністю тестового набору у виявленні дефектів. Вона поєднує їх із технікою few-shot-промптингу LLM і принципом «нульового пострілу» для вдосконалення й виправлення вже згенерованих тестових наборів. Це дослідження ілюструє загальну тенденцію, що сучасні роботи дедалі частіше не протиставляють SBST і LLM, а свідомо комбінують перевірені десятиліттями теоретичні засади пошукового тестування з новими генеративними можливостями мовних моделей.

5. Етап розробки: AI для рев'ю коду та передбачення дефектів

Робота «AI-Assisted Bug Prediction and Code Review Automation using Machine Learning» [21] описує типовий пайплайн Just-In-Time Software Defect Prediction (JIT-SDP). Починаючи з видобування даних з історії комітів, попередню обробку та закінчуючи ML/DL. Останній етап поєднує класичні алгоритми (Random Forest, XGBoost) з моделями глибокого навчання (CNN, LSTM) та трансформерними представленнями коду (CodeBERT, GraphCodeBERT). Це дозволяє одночасно вловлювати процесні та семантичні патерни. Автори наголошують на важливості боротьби з дисбалансом класів і на валідації через стратифіковану k-кратну крос-валідацію. Також

вони пропонують інтеграцію такого пайплайну безпосередньо в CI/CD для безперервного передбачення дефектів і автоматизованого рев'ю коду.

«A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning» [22]. Огляд узагальнює широкий пласт досліджень з виявлення вразливостей, що використовують ML/DL. Розглядаються моделі передбачення розвитку функціональних експлойтів і огляди прогнозування інцидентів кібербезпеки на основі даних до передбачення тяжкості багів на основі текстів запитань-відповідей зі StackOverflow. Окремо розглянуто підходи до виявлення вразливостей у бінарному коді на основі програмного зрізу (program slicing) та поєднання графового навчання з автоматизованим збором даних для виявлення вразливостей у коді. Також роботи з навчання семантичних ознак дефектів безпосередньо із вихідного коду без ручного інжинірингу.

У огляді «A Survey on Software Defect Prediction Using Deep Learning. MDPI Mathematics» [23] простежено еволюцію представлень коду для задачі передбачення дефектів. Розглядають послідовності токенів з one-hot-кодуванням, що подаються в двонаправлену LSTM-мережу для навчання нейронного пошуку багів на прикладах дефектного й коректного коду. А також деревоподібні представлення на основі абстрактного синтаксичного дерева (Tree-LSTM) та графових нейронних мереж (GNN). Такі мережі здатні видобувати інформацію про дефекти шляхом побудови й порівняння AST дефектної та виправленої версій фрагмента коду з подальшим виділенням «дефектного піддерева» алгоритмами виявлення спільнот у графі.

«AI-Powered Defect Prediction: From Code Smells to Failure Forecasting» [24]. Це систематичне дослідження охоплює 500 рецензованих статей і простежує застосування ШІ на різних стадіях існування дефекту програмного забезпечення. Починають огляд з ранніх непрямих ознак у вигляді code smells до прогнозування збоїв на рівні цілої системи. Робота охоплює як статичний, так і динамічний аналіз, окремо наголошуючи на зростаючій ролі пояснюваного ШІ. Зокрема, у задачах передбачення дефектів розробникам важливо не лише отримати прогноз, а й зрозуміти, чому конкретна ділянка коду вважається ризикованою.

Огляд «Automated Bug Detection and Program Repair Using Deep Learning: A Comprehensive Review» [25] систематизує напрямок навчально-орієнтованого автоматичного виправлення програм. Також розглядає ітеративні методи нейронного ремонту для патчів із кількома локаціями змін, підходи до підвищення ефективності виправлень через greybox-аналіз, а також задачу передбачення кількості дефектів у порівнянні контрольованих і неконтрольованих методів. Окремо описано гібридну архітектуру CBIL (комбінація CNN і Bi-LSTM) для передбачення дефектів, LLM-орієнтованих агентів для навчання використанню інструментів, а також відтворювані бенчмарки виправлення багів на основі GitHub Actions. Крім того, описано дослідження впливу вибору вхідних параметрів для виправлень на продуктивність систем автоматичного виправлення програм на еталонному датасеті Defects4J.

«AI-Based Code Review and Bug Detection» [26]. Дослідження проводить порівняльний аналіз кількох архітектур глибокого навчання для пошуку багів за критеріями точності, кількості хибнопозитивних спрацювань та обчислювальної ефективності. Після цього пропонує гібридний підхід, що поєднує кілька технік для максимізації продуктивності разом з усуненням недоліків окремих моделей. Розглянуто застосування графових нейронних мереж для представлення коду у вигляді графа з метою виявлення патернів, пов'язаних з минулими багами. Також описується використання рекурентних мереж для аналізу виконання та журналів на предмет аномалій. Згадують і трансформерні моделі для вловлювання довгих залежностей у коді для подальшої локалізації дефекту та переходу до виправлення.

6. Етап CI/CD: інтелектуальна пріоритизація та відбір тестів

«Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review» [27]. Це систематичний огляд літератури, що узагальнює методи пріоритизації та відбору тестових випадків на основі машинного навчання. Автори сформулювали пошуковий запит, що поєднує загальні терміни тестування («test», «selection», «prioritization», «rank») зі специфічними ML-термінами («learning», «clustering», «logistic», «support vector machine», «neural network», «bayes», «natural language processing», «k nearest neighbors», «reinforcement learning») Огляд дає поглиблений аналіз стану справ у ML-орієнтованій TSP та окреслює перспективні напрями подальших досліджень.

«Revisiting Machine Learning based Test Case Prioritization for Continuous Integration» [28]. Це перше комплексне порівняльне дослідження, що безпосередньо зіставляє 11 репрезентативних ML-технік пріоритизації тестів на 11 відкритих проєктах в однакових експериментальних умовах. В

попередніх схожих роботах різні техніки оцінювалися на різних датасетах з різними метриками, що унеможливлювало пряме порівняння. Ключовий висновок дослідження: продуктивність технік суттєво змінюється між циклами безперервної інтеграції переважно через зміну обсягу навчальних даних, а не через еволюцію коду чи додавання/видалення тестів. Це методологічно важливий результат, що вказує на необхідність постійного перенавчання ML-моделей пріоритизації в міру накопичення нових даних CI.

«Data-Driven Test Case Prioritization (DD-TCP): A Machine Learning Framework for Intelligent Software Quality Assurance» [29]. Запропонований фреймворк Data-Driven Test Case Prioritization реалізує адаптивний, навчально-орієнтований підхід до оптимізації регресійного тестування на основі даних, зібраних безпосередньо з CI. На відміну від традиційних підходів на основі покриття чи фіксованих евристик ризику, DD-TCP моделює задачу пріоритизації як завдання навчання з учителем, здатне вловлювати нелінійні взаємодії між ознаками і використовує проєкт-специфічні історичні дані для ухвалення рішень. Додавання ймовірнісного ранжування забезпечує стійкість до шуму та невизначеності, притаманних великомасштабним промисловим даним тестування. Ефективність підходу перевірена через масштабні симуляційні експерименти на синтетичних та відкритих регресійних датасетах, що імітують реалістичні CI-пайплайни.

Дослідження «Test Case Prioritization for Regression Testing Using Machine Learning» [30] прямо зосереджується на класичній проблемі регресійного тестування, пов'язаній із зростанням і ускладненням систем. Через це прогін усього набору тестів стає практично нездійсненним у межах часових і ресурсних обмежень CI. Техніки пріоритизації тестових випадків (TCP) впорядковують тести так, щоб максимізувати ймовірність раннього виявлення помилок. Робота демонструє, що інтеграція сучасних ML-технік із класичними TCP-підходами дає суттєвий приріст як ефективності, так і результативності процесу пріоритизації.

Огляд «The Application of ML in Test Case Prioritization – Review» [31] систематизує три альтернативні стратегії регресійного тестування: повторне виконання всіх тестів, відбір тестових випадків та власне пріоритизацію. Також докладно розглядає, чи здатні ML-моделі ефективно пріоритизувати тестові сценарії саме в контексті регресійного тестування. Серед розглянутих підходів були методологія предиктивної пріоритизації тестів (PTP) для CI-середовищ та дослідження пріоритизації регресійних тестових випадків на основі різних методів машинного навчання. У висновку підтверджують стійке зростання дослідницького інтересу до цього напрямку протягом останніх років.

7. Етап підтримки: self-healing тестування та GUI/visual-тестування

«Self-Healing Test Automation Framework using AI and ML» [32]. В роботі запропонована архітектура поєднує кілька AI/ML-компонентів. Першим компонентом є агент навчання з підкріпленням (RL), що навчається на історії попередніх виконань для покращення майбутніх адаптацій. Далі йде NLP-процесор, здатний генерувати й підтримувати тестові випадки безпосередньо з природномовних вимог. Наступним є модуль предиктивної аналітики, що прогнозує ймовірні майбутні збої тестів. Останній компонент це модуль розпізнавання зображень для візуальної взаємодії з елементами графічного інтерфейсу. Робочий процес системи охоплює автоматичне оновлення локаторів елементів, пропозиції щодо модифікацій тестових скриптів та рекомендації щодо зміни конфігурації тестового середовища, реалізовані поверх типового стеку інструментів (TensorFlow, scikit-learn, Selenium WebDriver, Appium).

Матеріал «Tricentis. What is self-healing test automation? A guide.» [33] визначає self-healing тести як такі, що використовують ШІ й машинне навчання для ідентифікації елементів інтерфейсу одразу за кількома атрибутами. Серед таких є ідентифікатор, назва класу, XPath, текстовий вміст, позиція та візуальний вигляд. У разі зміни одного атрибута система автоматично перемикається на інший і продовжує виконання тесту без переривання. За даними джерела, такий підхід застосовується у регресійному, наскрізному, кросбраузерному тестуванні та безпосередньо в CI/CD-пайплайнах. Такий підхід дозволяє суттєво знизити витрати на підтримку тестів без втрати покриття чи точності.

«Ranorex. Self-Healing Test Automation: Benefits and How It Works.»[34]. Джерело деталізує конкретні AI/ML-техніки, застосовані в self-healing автоматизації. Розглядають згорткові нейронні мережі (CNN) для комп'ютерного зору й розпізнавання елементів UI за їхнім візуальним виглядом, а не лише за ідентифікатором чи атрибутом. Також досліджують обробку природної мови (NLP) для розуміння семантичного призначення елемента, що допомагає системі адаптуватися до оновлень інтерфейсу. Описано навчання з учителем на історичних тестових даних для передбачення

ймовірного нового розташування елемента навіть у разі зміни його властивостей. І не менш важливим розглянутим елементом є навчання без учителя для виявлення прихованих патернів, що вказують на проблемні ділянки коду.

Робота «Functionize. Self-Healing Test Automation: Smarter Testing for Modern Apps.» [35] описує еволюцію self-healing від суто виправлення локаторів до повноцінного навчання моделі поведінки. Механізми самовідновлення навчаються на попередніх успішних збігах і патернах змін інтерфейсу, а ML-моделі уточнюють свої передбачення з кожним новим прогоном тесту. Це особливо актуально для сучасних динамічних фронтенд-фреймворків (React, Vue, Svelte, мікрофронтенди), швидких CI/CD-циклів із багаторазовими розгортаннями на день та великих корпоративних тестових наборів. Останньою тенденцією є «генеративне самовідновлення». LLM використовуються не лише для виправлення зламаного селектора елемента, а й для автоматичного написання абсолютно нових кроків тесту у разі принципової зміни бізнес-процесу, який цей тест перевіряє. Індустріальні джерела повідомляють про суттєве зниження витрат на підтримку тестів. За оцінками одних платформ, зниження сягає близько 70%, за оцінками інших — близько 85% (Рис. 3).

Рис. 3. Орієнтовне зниження витрат на підтримку тестових наборів завдяки self-healing test automation (ілюстративні дані за оцінками галузевих постачальників) [33, 35]

8. Індустріальний контекст: комерційні платформи з ШІ

Огляд ринку «Gartner Peer Insights. Best AI-Augmented Software Testing Tools.» [36] фіксує перехід індустрії від окремих точкових AI-функцій до повноцінних «агентних» платформ забезпечення якості (agentic software quality assurance). Розглянуто, зокрема, ACCELQ Unified, яка є платформою наскрізної автоматизації й управління тестами з AI-орієнтованим дизайном тестів через візуальне моделювання робочих процесів. Katalon Platform — рішення, що об'єднує ручне тестування, автоматизацію, виконання, аналітику й продакшн-інсайти в єдиній системі обліку для управління якістю. Також описано BrowserStack, яка є платформою для кросбраузерного тестування, тестування на реальних пристроях, перевірки доступності й візуального тестування. TestMu AI — «повністю агентна» платформа інженерії якості. І останньою платформою є UiPath Test Suite — рішення для тестування RPA-робочих процесів, інтегроване безпосередньо в автоматизацію бізнес-процесів.

«SmartBear Delivers AI-Enhancements Across Entire Software Application Testing Lifecycle» [37]. Стаття ілюструє практичне охоплення індустріальними рішеннями одразу двох етапів SDLC: розробки та підтримки. Це все доповнене корпоративними механізмами управління для дотримання вимог відповідності. Огляд спирається на дані того самого опитування 273 фахівців [2], що зафіксувало занепокоєння 70% респондентів щодо вже наявного погіршення якості й 68% респондентів щодо ризику вузьких місць у тестуванні. Стаття демонструє одночасний рух постачальників у двох напрямках: до повністю автономних рішень та до AI-інструментів, що доповнюють тестування, кероване людиною.

9. Порівняльний аналіз та обґрунтування отриманих результатів

Узагальнення розглянутих джерел дозволяє обґрунтувати два ключові наукові результати огляду. Перший — порівняльна характеристика інструментів LLM-орієнтованої генерації unit-тестів із зазначенням базової моделі, метрик ефективності та основних обмежень (Таблиця 1).

Таблиця 1. Порівняння інструментів LLM-орієнтованої генерації unit-тестів

Інструмент	Базова LLM / метод	Метрика ефективності	Основне обмеження
TestPilot [7]	GPT-3.5-turbo, без донавчання	70,2% statement / 52,8% branch coverage (25 npm-пакетів)	Обмежено JavaScript API; залежність від якості документації
TestART [8]	LLM + ітеративне виправлення	Вища частка валідних (компільованих) тестів vs простим LLM	Додаткові обчислювальні витрати на цикли виправлення
ChatUniTest [9, 14]	ChatGPT, схема generation-validation-repair	Покращена компільованість тестів	Залежність від якості зворотного зв'язку компілятора
AgoneTest [10]	Кілька LLM + few-shot технік	Оцінка на 9410 GitHub-репозиторіїв	Складність налаштування для нових мов/фреймворків
CODAMOSA [14]	Codex + SBST (гібрид)	Покращення покриття на «плато» SBST	Обмежено Python; потребує доступу до LLM під час пошуку
EvoGPT [16]	LLM (посів популяції) + еволюційний алгоритм	Подолання «плато» покриття SBST	Складність калібрування температури/різноманітності
UTGen [15]	EvoSuite (SBST) + LLM (читабельність)	Покращена зрозумілість тестів при збереженні покриття	Незначне зниження покриття порівняно з чистим SBST

Другий результат — узагальнений розподіл підходів ШІ за етапами SDLC, що є основою запропонованої в статті класифікаційної моделі (Таблиця 2). Розподіл підтверджує, що дослідницька активність концентрується на етапі розробки (три з шести категорій), тоді як етапи CI/CD і підтримки охоплені відповідно ML-пріоритизацією та self-healing автоматизацією.

Таблиця 2. Розподіл AI-підходів до автоматизації тестування за етапами SDLC

Етап SDLC	AI-підхід	Приклади інструментів/робіт	Основна задача
Розробка	LLM-генерація unit-тестів	TestPilot, TestART, ChatUniTest, AgoneTest [7–14]	Автоматичне створення тестів для нового/зміненого коду
Розробка	Гібрид SBST + LLM	EvoSuite, UTGen, EvoGPT, CODAMOSA [15–20]	Поєднання високого покриття та читабельності тестів
Розробка	AI-рев'ю коду / defect prediction	CodeBERT, GraphCodeBERT, DeepJIT-подібні системи [21–26]	Раннє виявлення дефектів і вразливостей у коді
CI/CD	ML-пріоритизація тестів	ML-based TSP, DD-TCP [27–31]	Оптимізація порядку виконання регресійних тестів
Підтримка	Self-healing тестування	Testim, Mabl, Applitools, Functionize [32–35]	Автоматичне виправлення тестів при зміні UI
Наскрізно	Індустріальні AI-QA платформи	ACCELQ, Katalon, ReadyAPI, TestComplete [36, 37]	Комплексна підтримка AI на всіх етапах тестування

Проведений аналіз також дозволяє виділити наскрізні проблеми, що залишаються відкритими для напрямку AI-driven тестування:

- Надійність згенерованого коду тестів. Значна частка тестів, згенерованих простими LLM-підходами без циклу виправлення, є некомпільованою або містить галюциновані залежності й некоректні твердження [12, 13]. Це обмежує пряме впровадження таких тестів у продакшн без додаткової верифікації.
- Проблема тестового оракула зберігає актуальність і для AI-згенерованих тестів. LLM здатні згенерувати правдоподібний тестовий код, проте питання коректності самого очікуваного результату часто лишається невирішеним автоматично й потребує залучення розробника [14].

- Довіра розробників до AI-згенерованих артефактів. Попри високі кількісні метрики покриття, якісні дослідження test smells [13] показують, що AI-згенеровані тести можуть відтворювати чи навіть підсилювати типові антипатерни тестування, що ставить питання про необхідність автоматизованого контролю якості поверх самої генерації.
- Брак довгострокових незалежних досліджень. Більшість кількісних показників ефективності self-healing тестування [33, 35] походить від постачальників комерційних рішень і потребує незалежної академічної верифікації на репрезентативних вибірках проєктів.
- Суперечливість оцінок ефективності. Дослідження [19] прямо демонструє, що оцінки приросту покриття від LLM-генерації в різних роботах відрізняються на порядок (від ~2% до ~80%) залежно від бенчмарку й методології. Це ускладнює узагальнення результатів окремих досліджень на практиці.
- Гібридизація підходів як основний напрямок. Тенденція до поєднання класичних SBST-технік із генеративними можливостями LLM [15, 16, 19, 20] свідчить, що майбутні дослідження, ймовірно, зосередяться не на протиставленні цих підходів, а на оптимальних стратегіях їх комбінування залежно від мови програмування, типу застосунку та стадії SDLC.

Висновки та перспективи подальших досліджень. У статті оглянуто та систематизовано сучасний стан застосування штучного інтелекту для автоматизації тестування програмного забезпечення на різних етапах життєвого циклу розробки. Для кожного з 37 розглянутих джерел наведено окремий розбір матеріалу, результатів та обмежень.

Наукова новизна дослідження полягає в наступному: запропоновано lifecycle-орієнтовану класифікаційну модель, що в межах єдиного огляду узгоджує етапи SDLC з відповідними AI-підходами та інструментами, на основі порівняльного аналізу інструментів LLM-генерації тестів обґрунтовано, що кількісні метрики покриття недостатні для оцінки практичної цінності AI-згенерованих тестів і мають доповнюватися якісними показниками, виявлено й систематизовано шість наскрізних відкритих проблем напрямку, зокрема суперечливість оцінок ефективності та відсутність незалежної верифікації заявок щодо self-healing рішень.

Показано, що найбільш активно розвиваються методи LLM-орієнтованої генерації unit-тестів на етапі розробки з чіткою тенденцією до гібридизації з класичними пошуковими техніками тестування для компенсації взаємних недоліків обох підходів. На етапах CI/CD та підтримки ключову роль відіграють методи ML-пріоритизації тестів та self-healing test automation відповідно, які дозволяють суттєво скоротити витрати на підтримку тестових наборів у динамічних середовищах розробки.

Подальші дослідження доцільно спрямувати на розробку стандартизованих бенчмарків якості AI-згенерованих тестів та на верифікацію заявок щодо ефективності self-healing рішень у довгостроковій перспективі.

Список бібліографічного опису

1. Naqvi S., Baqar M. Breaking Barriers in Software Testing: The Power of AI-Driven Automation. arXiv : препринт. 2025. arXiv:2508.16025. URL: https://arxiv.org/pdf/2508.16025 (дата звернення: 16.08.2026).
2. AI Software Quality Gap Report / SmartBear Software. 2026. URL: https://smartbear.com/ai-software-quality-gap-report/ (дата звернення: 16.08.2026).
3. Faraji A., Pombo N. AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges. IEEE Access. 2025. Vol. 13. P. 183296–183313. DOI: 10.1109/ACCESS.2025.3623944.
4. Baqar M., Khanda R. The Future of Software Testing: AI-Powered Test Case Generation and Validation. Intelligent Computing : Proceedings of the 2025 Computing Conference. Cham : Springer, 2025. Vol. 2. P. 276–300. DOI: 10.1007/978-3-031-92605-1.
5. Hymel C. The AI-Native Software Development Lifecycle: A Theoretical and Practical New Methodology. arXiv : препринт. 2024. arXiv:2408.03416. URL: https://arxiv.org/pdf/2408.03416 (дата звернення: 16.08.2026).
6. Vizard M. Survey Sees AI Being Applied to Improve Software Quality Testing. DevOps.com. 2024. 22 жовт. URL: https://devops.com/survey-sees-ai-being-applied-to-improve-software-quality-testing/ (дата звернення: 16.08.2026).
7. Schäfer M., Nadi S., Eghbali A., Tip F. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering. 2024. Vol. 50, No. 1. P. 85–105. DOI: 10.1109/TSE.2023.3334955.
8. Gu S., Fang C., Zhang Q., Tian F., Zhou J., Chen Z. TestART: Improving LLM-based Unit Test via Co-evolution of Automated Generation and Repair Iteration. arXiv : препринт. 2024. arXiv:2408.03095. URL: https://arxiv.org/html/2408.03095v3 (дата звернення: 16.08.2026).
9. Yang L., Yang C., Gao S., Wang W., Wang B., Zhu Q., Chu X., Zhou J., Liang G., Wang Q., Chen J. An Empirical Study of Unit Test Generation with Large Language Models. arXiv : препринт. 2024. arXiv:2406.18181. URL: https://arxiv.org/html/2406.18181v1 (дата звернення: 16.08.2026).

10. Lops A., Narducci F., Ragone A., Trizio M., Bartolini C. A System for Automated Unit Test Generation Using Large Language Models and Assessment of Generated Test Suites. 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). 2025. P. 29–36. DOI: 10.1109/ICSTW64639.2025.10962454.
11. Storhaug A., Li J. Parameter-Efficient Fine-Tuning of Large Language Models for Unit Test Generation: An Empirical Study. arXiv : препринт. 2024. arXiv:2411.02462. DOI: 10.48550/arXiv.2411.02462.
12. Haratian V., Akar A., Çakar B., Tüzün E. PR-Aware Automated Unit Test Generation: Challenges and Opportunities. arXiv : препринт. 2026. arXiv:2605.25285. URL: https://arxiv.org/pdf/2605.25285 (дата звернення: 16.08.2026).
13. Ouédraogo W. C., Li Y., Kaboré A. K., Tang X., Koyuncu A., Klein J., Lo D., Bissyandé T. F. Test Smells in LLM-Generated Unit Tests. arXiv : препринт. 2024. arXiv:2410.10628. DOI: 10.48550/arXiv.2410.10628.
14. Zhang Q., Fang C., Xie Y., Zhang Y., Yang Y., Sun W., Yu S., Chen Z. A Survey on Large Language Models for Software Engineering. Science China Information Sciences. 2026. Vol. 69. Art. 141102. DOI: 10.1007/s11432-025-4670-0.
15. Deljouyi A., Koohestani R., Izadi M., Zaidman A. Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests. 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). 2025. P. 1449–1461. DOI: 10.1109/ICSE55347.2025.00032.
16. Broide L., Stern R., Mordoch A. EvoGPT: Leveraging LLM-Driven Seed Diversity to Improve Search-Based Test Suite Generation. arXiv : препринт. 2025. arXiv:2505.12424. URL: https://arxiv.org/abs/2505.12424 (дата звернення: 16.08.2026).
17. Biagiola M., Ghislotti G., Tonella P. Improving the Readability of Automatically Generated Tests Using Large Language Models. 2025 IEEE Conference on Software Testing, Verification and Validation (ICST). 2025. P. 162–173. DOI: 10.1109/ICST62969.2025.10989020.
18. Fraser G., Arcuri A. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference (ESEC/FSE 2011). New York : ACM, 2011. P. 416–419. DOI: 10.1145/2025113.2025179.
19. Drăgoi A. Exploring Test Suite Coverage of Large Language Model-Enhanced Test Generation : bachelor thesis. Delft : Delft University of Technology, 2024. URL: https://repository.tudelft.nl/file/File_51e9ae71-24e0-4eed-be48-0b7728d17d71 (дата звернення: 16.08.2026).
20. Chudic A., Çalıklı G. Automated Test Suite Enhancement Using Large Language Models with Few-shot Prompting. arXiv : препринт. 2026. arXiv:2602.12256. URL: https://arxiv.org/pdf/2602.12256 (дата звернення: 16.08.2026).
21. Prasad S. AI-Assisted Bug Prediction and Code Review Automation Using Machine Learning. SSRN Electronic Journal. 2026. URL: https://www.researchgate.net/publication/401660442 (дата звернення: 16.08.2026).
22. Harzevili N. S., Belle A. B., Wang J., Wang S., Jiang Z. M., Nagappan N. A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning. arXiv : препринт. 2023. arXiv:2306.11673. DOI: 10.48550/arXiv.2306.11673.
23. Akimova E. N., Bersenev A. Yu., Deikov A. A., Kobylkin K. S., Konygin A. V., Mezentsev I. P., Misilov V. E. A Survey on Software Defect Prediction Using Deep Learning. Mathematics. 2021. Vol. 9, No. 11. Art. 1180. DOI: 10.3390/math9111180.
24. Rahman M. M., Md N. G., Shuvra S. D. M. K., Rahman M. M., Hossain M. S. AI-Powered Defect Prediction: From Code Smells to Failure Forecasting. International Journal of Communication Networks and Information Security. 2026. Vol. 18, No. 1. P. 95–118. URL: https://www.ijcnis.org/index.php/ijcnis/article/view/8763 (дата звернення: 16.08.2026).
25. Ali R. H., Chyad A. M. Automated Bug Detection and Program Repair Using Deep Learning: A Comprehensive Review. Cybernetics and Information Technologies. 2026. Vol. 26, No. 1. P. 93–121. DOI: 10.2478/cait-2026-0006.
26. Anand A., Choudhary S., Verma H. AI-Based Code Review and Bug Detection: Towards Smarter Software Development. International Journal of Progressive Research in Engineering Management and Science. 2025. Vol. 5, No. 5. P. 1017–1027. URL: https://www.ijprems.com/uploadedfiles/paper/issue_5_may_2025/41352/final/fin_ijprems1748285482.pdf (дата звернення: 16.08.2026).
27. Pan R., Bagherzadeh M., Ghaleb T. A., Briand L. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. Empirical Software Engineering. 2022. Vol. 27, No. 2. Art. 29. DOI: 10.1007/s10664-021-10066-6.
28. Zhao Y., Hao D., Zhang L. Revisiting Machine Learning Based Test Case Prioritization for Continuous Integration. 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2023. P. 232–244. DOI: 10.1109/ICSME58846.2023.00032.
29. Ramzan H. A., Islam K., Hussain M. A., Monim R. M., Asad S. M., Ramzan S. Data-Driven Test Case Prioritization (DD-TCP): A Machine Learning Framework for Intelligent Software Quality Assurance. Computers, Materials & Continua. 2026. Vol. 88, No. 1. Art. 57. DOI: 10.32604/cmc.2026.077782.
30. Sawant P. D. Test Case Prioritization for Regression Testing Using Machine Learning. 2024 IEEE International Conference on Artificial Intelligence Testing (AITest). Shanghai, 2024. P. 152–153. DOI: 10.1109/AITest62860.2024.00027.
31. Meçe E. K., Binjaku K., Paci H. The Application of Machine Learning in Test Case Prioritization – A Review. European Journal of Electrical Engineering and Computer Science. 2020. Vol. 4, No. 1. DOI: 10.24018/ejece.2020.4.1.128.
32. Saarathy S. C. P., Bathrachalam S., Rajendran B. K. Self-Healing Test Automation Framework Using AI and ML. International Journal of Strategic Management. 2024. Vol. 3, No. 3. P. 45–77. DOI: 10.47604/ijsm.2843.
33. Reyes J. What Is Self-Healing Test Automation? A Guide. Tricentis. 2026. 9 квіт. URL: https://www.tricentis.com/learn/self-healing-test-automation (дата звернення: 16.08.2026).

34. Pruitt M. Self-Healing Test Automation: Benefits and How It Works. Ranorex. 2025. 2 груд. URL: https://www.ranorex.com/blog/self-healing-test-automation/ (дата звернення: 16.08.2026).
35. Cser T. Self-Healing Test Automation: Smarter Testing for Modern Apps. Functionize. 2025. 17 груд. URL: https://www.functionize.com/automated-testing/self-healing-test-automation (дата звернення: 16.08.2026).
36. Best AI-Augmented Software Testing Tools: Reviews and Ratings / Gartner Peer Insights. 2026. URL: https://www.gartner.com/reviews/market/ai-augmented-software-testing-tools (дата звернення: 16.08.2026).
37. SmartBear Delivers AI-Enhancements Across Entire Software Application Testing Lifecycle / SmartBear Software. Business Wire. 2026. 31 берез. URL: https://www.businesswire.com/news/home/20260331994897/en/ (дата звернення: 16.08.2026).

References

1. Naqvi, S., & Baqar, M. (2025). Breaking barriers in software testing: The power of AI-driven automation. arXiv. https://arxiv.org/abs/2508.16025
2. SmartBear Software. (2026). AI software quality gap report. https://smartbear.com/ai-software-quality-gap-report/
3. Faraji, A., & Pombo, N. (2025). AI-driven software test automation: An AI4SE-oriented survey of techniques, tools, and challenges. IEEE Access, 13, 183296–183313. https://doi.org/10.1109/ACCESS.2025.3623944
4. Baqar, M., & Khanda, R. (2025). The future of software testing: AI-powered test case generation and validation. In Intelligent computing: Proceedings of the 2025 Computing Conference (Vol. 2, pp. 276–300). Springer. https://doi.org/10.1007/978-3-031-92605-1
5. Hymel, C. (2024). The AI-native software development lifecycle: A theoretical and practical new methodology. arXiv. https://arxiv.org/abs/2408.03416
6. Vizard, M. (2024, October 22). Survey sees AI being applied to improve software quality testing. DevOps.com. https://devops.com/survey-sees-ai-being-applied-to-improve-software-quality-testing/
7. Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering, 50(1), 85–105. https://doi.org/10.1109/TSE.2023.3334955
8. Gu, S., Fang, C., Zhang, Q., Tian, F., Zhou, J., & Chen, Z. (2024). TestART: Improving LLM-based unit test via co-evolution of automated generation and repair iteration. arXiv. https://arxiv.org/abs/2408.03095
9. Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q., Chu, X., Zhou, J., Liang, G., Wang, Q., & Chen, J. (2024). An empirical study of unit test generation with large language models. arXiv. https://arxiv.org/abs/2406.18181
10. Lops, A., Narducci, F., Ragone, A., Trizio, M., & Bartolini, C. (2025). A system for automated unit test generation using large language models and assessment of generated test suites. In 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 29–36). IEEE. https://doi.org/10.1109/ICSTW64639.2025.10962454
11. Storhaug, A., & Li, J. (2024). Parameter-efficient fine-tuning of large language models for unit test generation: An empirical study. arXiv. https://doi.org/10.48550/arXiv.2411.02462
12. Haratian, V., Akar, A., Çakar, B., & Tüzün, E. (2026). PR-aware automated unit test generation: Challenges and opportunities. arXiv. https://arxiv.org/abs/2605.25285
13. Ouédraogo, W. C., Li, Y., Kaboré, A. K., Tang, X., Koyuncu, A., Klein, J., Lo, D., & Bissyandé, T. F. (2024). Test smells in LLM-generated unit tests. arXiv. https://doi.org/10.48550/arXiv.2410.10628
14. Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., & Chen, Z. (2026). A survey on large language models for software engineering. Science China Information Sciences, 69, Article 141102. https://doi.org/10.1007/s11432-025-4670-0
15. Deljouyi, A., Koohestani, R., Izadi, M., & Zaidman, A. (2025). Leveraging large language models for enhancing the understandability of generated unit tests. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (pp. 1449–1461). IEEE. https://doi.org/10.1109/ICSE55347.2025.00032
16. Broide, L., Stern, R., & Mordoch, A. (2025). EvoGPT: Leveraging LLM-driven seed diversity to improve search-based test suite generation. arXiv. https://arxiv.org/abs/2505.12424
17. Biagiola, M., Ghislotti, G., & Tonella, P. (2025). Improving the readability of automatically generated tests using large language models. In 2025 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 162–173). IEEE. https://doi.org/10.1109/ICST62969.2025.10989020
18. Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference (ESEC/FSE 2011) (pp. 416–419). ACM. https://doi.org/10.1145/2025113.2025179
19. Drăgoi, A. (2024). Exploring test suite coverage of large language model-enhanced test generation [Bachelor thesis, Delft University of Technology]. TU Delft Repository. https://repository.tudelft.nl/file/File_51e9ae71-24e0-4eed-be48-0b7728d17d71
20. Chudic, A., & Çalıklı, G. (2026). Automated test suite enhancement using large language models with few-shot prompting. arXiv. https://arxiv.org/abs/2602.12256
21. Prasad, S. (2026). AI-assisted bug prediction and code review automation using machine learning. SSRN Electronic Journal. https://www.researchgate.net/publication/401660442
22. Harzevili, N. S., Belle, A. B., Wang, J., Wang, S., Jiang, Z. M., & Nagappan, N. (2023). A survey on automated software vulnerability detection using machine learning and deep learning. arXiv. https://doi.org/10.48550/arXiv.2306.11673
23. Akimova, E. N., Bersenev, A. Yu., Deikov, A. A., Kobylkin, K. S., Konygin, A. V., Mezentsev, I. P., & Misilov, V. E. (2021). A survey on software defect prediction using deep learning. Mathematics, 9(11), Article 1180. https://doi.org/10.3390/math9111180

24. Rahman, M. M., Md, N. G., Shuvra, S. D. M. K., Rahman, M. M., & Hossain, M. S. (2026). AI-powered defect prediction: From code smells to failure forecasting. International Journal of Communication Networks and Information Security, 18(1), 95–118. https://www.ijcnis.org/index.php/ijcnis/article/view/8763
25. Ali, R. H., & Chyad, A. M. (2026). Automated bug detection and program repair using deep learning: A comprehensive review. Cybernetics and Information Technologies, 26(1), 93–121. https://doi.org/10.2478/cait-2026-0006
26. Anand, A., Choudhary, S., & Verma, H. (2025). AI-based code review and bug detection: Towards smarter software development. International Journal of Progressive Research in Engineering Management and Science, 5(5), 1017–1027. https://www.ijprems.com/uploadedfiles/paper/issue_5_may_2025/41352/final/fin_ijprems1748285482.pdf
27. Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test case selection and prioritization using machine learning: A systematic literature review. Empirical Software Engineering, 27(2), Article 29. https://doi.org/10.1007/s10664-021-10066-6
28. Zhao, Y., Hao, D., & Zhang, L. (2023). Revisiting machine learning based test case prioritization for continuous integration. In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 232–244). IEEE. https://doi.org/10.1109/ICSME58846.2023.00032
29. Ramzan, H. A., Islam, K., Hussain, M. A., Monim, R. M., Asad, S. M., & Ramzan, S. (2026). Data-driven test case prioritization (DD-TCP): A machine learning framework for intelligent software quality assurance. Computers, Materials & Continua, 88(1), Article 57. https://doi.org/10.32604/cmc.2026.077782
30. Sawant, P. D. (2024). Test case prioritization for regression testing using machine learning. In 2024 IEEE International Conference on Artificial Intelligence Testing (AITest) (pp. 152–153). IEEE. https://doi.org/10.1109/AITest62860.2024.00027
31. Meçe, E. K., Binjaku, K., & Paci, H. (2020). The application of machine learning in test case prioritization – A review. European Journal of Electrical Engineering and Computer Science, 4(1). https://doi.org/10.24018/ejece.2020.4.1.128
32. Saarathy, S. C. P., Bathrachalam, S., & Rajendran, B. K. (2024). Self-healing test automation framework using AI and ML. International Journal of Strategic Management, 3(3), 45–77. https://doi.org/10.47604/ijsm.2843
33. Reyes, J. (2026, April 9). What is self-healing test automation? A guide. Tricentis. https://www.tricentis.com/learn/self-healing-test-automation
34. Pruitt, M. (2025, December 2). Self-healing test automation: Benefits and how it works. Ranorex. https://www.ranorex.com/blog/self-healing-test-automation/
35. Cser, T. (2025, December 17). Self-healing test automation: Smarter testing for modern apps. Functionize. https://www.functionize.com/automated-testing/self-healing-test-automation
36. Gartner Peer Insights. (2026). Best AI-augmented software testing tools: Reviews and ratings. https://www.gartner.com/reviews/market/ai-augmented-software-testing-tools
37. SmartBear Software. (2026, March 31). SmartBear delivers AI-enhancements across entire software application testing lifecycle. Business Wire. https://www.businesswire.com/news/home/20260331994897/en/

Історія статті:
Отримано: 17.05.2026 Доопрацьовано: 20.08.2026 Прийнято до друку: 23.09.2026 Опубліковано: 29.09.2026