https://doi.org/10.31891/2219-9365-2026-87-18
УДК 004.89:004.41

ЧУЗОВ Дмитро
Запорізький національний університет
https://orcid.org/0009-0006-3879-6536
chuzov.d@gmail.com

ЗАСТОСУВАННЯ МЕТОДІВ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ВЕБДОДАТКІВ

У статті досліджується застосування методів штучного інтелекту для автоматизації тестування сучасних вебдодатків на базі React та Angular. Проаналізовано сучасні підходи до використання великих мовних моделей для генерації тестового коду та комп'ютерного зору (Computer Vision) для візуального тестування інтерфейсів. Запропоновано гібридний підхід, що поєднує можливості LLM для інтелектуальної генерації тестових сценаріїв з методами комп'ютерного зору для виявлення візуальних дефектів та функціональних помилок. Описано технічну реалізацію експериментального фреймворку на базі Python, Playwright, OpenAI API та OpenCV. Проведено експериментальну перевірку на тестових SPA-додатках, що демонструє ефективність запропонованого підходу для автоматизації тестування динамічних інтерфейсів.

Ключові слова: штучний інтелект, тестування програмного забезпечення, великі мовні моделі, комп'ютерний зір, React, Angular, автоматизація тестування.

CHUZOV Dmytro
Zaporizhzhya National University

APPLICATION OF ARTIFICIAL INTELLIGENCE METHODS FOR AUTOMATED WEB APPLICATIONS TESTING

Modern single-page web applications (SPAs) developed using React, Angular, and similar JavaScript frameworks are characterized by dynamic interfaces, asynchronous interactions, and continuously changing Document Object Model (DOM) structures. These features significantly complicate software quality assurance and reduce the effectiveness of traditional automated testing approaches based on static selectors and manually created test scripts. This paper investigates the application of artificial intelligence methods to improve the automation of functional and visual testing of modern web applications. A hybrid testing approach is proposed that integrates the capabilities of Large Language Models (LLMs) for intelligent test case generation with Computer Vision techniques for automated visual interface validation. The proposed framework automatically transforms natural-language functional requirements and user stories into executable Playwright test scripts while simultaneously performing visual regression analysis using OpenCV-based image comparison algorithms. To improve the robustness of automated testing, the framework incorporates mechanisms for selector self-healing, semantic analysis of detected visual differences, and adaptation to dynamic DOM modifications typical of SPA architectures. The technical implementation of the framework is based on Python 3.11 and integrates Playwright, OpenAI API, OpenCV, scikit-image, and asynchronous execution mechanisms, enabling seamless integration into modern CI/CD pipelines.

The effectiveness of the proposed approach was experimentally evaluated using three representative single-page applications developed with React, Angular, and Vue.js. Each application included multiple functional scenarios covering user authentication, data processing, shopping cart operations, dashboard management, and dynamic user interactions. Experimental results demonstrate that the hybrid AI-based framework successfully detects 93.3% of functional defects and 86.7% of visual regressions while significantly reducing the number of false-positive detections compared with conventional snapshot testing techniques. Furthermore, the proposed solution decreases automated test development time by more than four times and substantially reduces maintenance effort through intelligent adaptation to interface modifications and automatic selector recovery. The obtained results confirm that combining LLM-driven reasoning with computer vision techniques enables more reliable, scalable, and maintainable automated testing for highly dynamic web applications. The proposed methodology contributes to the development of next-generation AI-assisted software quality assurance systems and demonstrates strong potential for practical implementation in industrial software engineering projects involving continuous integration and continuous delivery environments.

Keywords: artificial intelligence, software testing, large language models, computer vision, React, Angular, test automation.

Стаття надійшла до редакції / Received 11.02.2026
Прийнята до друку / Accepted 06.07.2026
Опубліковано / Published 10.09.2026

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Сучасні вебдодатки характеризуються високою динамічністю, складною архітектурою та інтенсивною взаємодією з користувачем. Зокрема це стосується односторінкових додатків (Single Page Applications, SPA), побудованих на базі популярних JavaScript-фреймворків: React, Angular та Vue.js. Такі додатки забезпечують плавну навігацію без перезавантаження сторінки, динамічне оновлення DOM-структури та асинхронну взаємодію з серверними API [1], [2].

Традиційні методи тестування вебдодатків, що базуються на статичних селекторах та жорстко

закодованих тестових сценаріях, часто виявляються недостатньо ефективними для SPA. Динамічна природа цих додатків призводить до частих змін у структурі DOM, що спричиняє високу крихкість автоматизованих тестів та значні витрати на їх підтримку [3]. Водночас, зростаюча складність інтерфейсів користувача вимагає не лише функціонального тестування, але й ретельної перевірки візуальної коректності відображення елементів.

Останні досягнення в галузі штучного інтелекту, зокрема розвиток великих мовних моделей (Large Language Models, LLM) та методів комп'ютерного зору (Computer Vision, CV), відкривають нові можливості для автоматизації тестування програмного забезпечення [5], [6]. LLM демонструють здатність розуміти тестові сценарії, написані природньою мовою та генерувати відповідний програмний код автоматизованих тестів[8]. Методи комп'ютерного зору дозволяють автоматично виявляти візуальні відмінності між еталонними та поточними версіями інтерфейсу, забезпечуючи ефективне виявлення нових помилок [10], [12].

Інтеграція цих технологій у процес тестування вебдодатків має потенціал суттєво підвищити ефективність контролю якості програмного забезпечення, зменшити витрати на розробку та підтримку тестів, а також забезпечити більш комплексне покриття функціональних та візуальних аспектів додатків.

Тестування сучасних SPA-додатків стикається з низкою специфічних викликів, що обмежують ефективність традиційних підходів до автоматизації тестування.

Динамічна природа SPA призводить до частих змін у структурі DOM, що робить тести, що базуються на статичних селекторах, надзвичайно нестабільними. Дослідження показують, що підтримка таких тестів вимагає значних ресурсів та часто перевищує витрати на їх первинну розробку [12], [15]. Snapshot-тестування, хоча й дозволяє виявляти візуальні зміни, генерує велику кількість хибних спрацьовувань, що вимагає ручного аналізу та збільшує навантаження на тестувальників [10].

Традиційне автоматизоване тестування ефективно знаходить лише очікувані помилки, на перевірку саме яких був написаний код тестів, але не здатне надійно виявляти функціональні помилки, виявлення яких вимагає глибшого розуміння контексту та очікуваної поведінки додатку, що виходить за межі можливостей класичних підходів [8].

Багато існуючих рішень вимагають ручного написання тестових скриптів або доступу до backend-систем, що обмежує їх застосовність при тестуванні додатків з обмеженим доступом до внутрішньої архітектури [7], [8].

Ці проблеми особливо гостро проявляються при тестуванні SPA-додатків на базі React та Angular, де динамічна зміна стану компонентів, віртуальний DOM та складні механізми маршрутизації створюють додаткові виклики для традиційних методів тестування.

АНАЛІЗ ОСТАННІХ ДОСЛІДЖЕНЬ ТА ПУБЛІКАЦІЙ

Останні дослідження демонструють значний потенціал великих мовних моделей у автоматизації генерації тестового коду. LLM використовуються як планувальники, що перетворюють мовні описи завдань у послідовності дій для тестування інтерфейсу користувача [7].

Мультимодальні LLM здатні аналізувати як текстову інформацію, витягнуту з графічного інтерфейсу, так і знімки екрану, що дозволяє їм розуміти контекст сторінки та приймати обґрунтовані рішення щодо наступних дій під час дослідження додатку [8]. Дослідження показують, що такий підхід дозволяє виявляти функціональні недоліки, які не призводять до критичних помилок, але порушують очікувану логіку роботи [9].

Емпіричні дослідження підтверджують ефективність LLM-підходів. Зокрема, роботи [11], [20], [21] демонструють здатність моделей генерувати автоматизовані тестові сценарії для вебдодатків на основі мовних описів. Дослідження [17], [18] показують, що LLM можуть ефективно використовуватися для регресійного тестування та порівняльного аналізу різних підходів до генерації тестів.

Перспективним напрямком є інтеграція LLM та методів комп'ютерного зору, що застосовуються для виявлення візуальних дефектів та підтримки тестів з механізмом самовідновлення, в єдині фреймворки, що дозволяють використовувати переваги обох підходів [1], [3], [8]. Дослідження демонструють використання великих багатомодульних моделей для автоматизованого тестування графічного інтерфейсу, поєднуючи аналіз візуальної та структурної інформації [3].

Окрім того важливим є використання підходів, що дозволяють LLM використовувати знання про попередні помилки та тестові сценарії для покращення якості генерованих тестів [16]. Дослідження [19] показують, що такі підходи дозволяють значно підвищити ефективність автоматизованого тестування графічного інтерфейсу користувача.

Незважаючи на перспективні емпіричні результати, поточні роботи залишають кілька важливих прогалин. У наявних дослідженнях недостатньо експериментів з SPA-фреймворками та їх динамічною поведінкою DOM. Також небагато досліджень було проведено в умовах, що імітують реальні життєві цикли корпоративних SPA, наприклад кросбраузерність та довготривалість використання. Наявність вищезазначених прогалин залишає питання про надійність, масштабованість, та витрати на підтримку без відповіді [8], [10].

ФОРМУЛЮВАННЯ ЦІЛЕЙ ДОСЛІДЖЕННЯ

Метою цього дослідження є розробка та експериментальна перевірка гібридного підходу до автоматизації тестування односторінкових вебдодатків, що інтегрує можливості великих мовних моделей та методів комп'ютерного зору.

Основним завданням дослідження є розробка архітектури гібридного фреймворку, що поєднує LLM для генерації тестових сценаріїв та методи комп'ютерного зору для візуального тестування SPA-додатків, реалізація модуля генерації тестів на базі LLM, здатного перетворювати описи функціональності в тестові сценарії для React та Angular додатків, розробка модуля візуального тестування на базі OpenCV для виявлення візуальних дефектів та функціональних помилок у динамічних інтерфейсах, а також проведення експериментальної перевірки запропонованого підходу на тестових SPA-додатках задля оцінки їх ефективності порівняно з традиційними методами. На основі отриманих результатів буде визначене обмеження та напрямки подальшого розвитку гібридних підходів до тестування SPA.

ВИКЛАД ОСНОВНОГО МАТЕРАІЛУ ДОСЛІДЖЕННЯ

Запропонована архітектура гібридного фреймворку для тестування SPA-додатків складається з трьох основних компонентів, що взаємодіють для забезпечення комплексного тестування як функціональних, так і візуальних аспектів додатку.

Компонент 1: *модуль генерації тестів* на базі LLM. Цей модуль відповідає за перетворення мовних описів функціональності або сценаріїв користувача у код автоматизованих тестів. Модуль використовує API великої мовної моделі (GPT-4 або аналогічної) для аналізу вхідних описів та генерації відповідного коду на Python з використанням бібліотеки Playwright для автоматизації браузера. На рис. 1 наведено приклад реалізації методу для взаємодії з API мовної моделі з метою автоматичної генерації тестового коду, а на рис.2 — приклад згенерованого таким чином коду для тестування React-додатку

```python
import openai

def generate_playwright_test(user_story, dom_context):
    # Генерація тестового сценарію на основі опису та контексту DOM.
    prompt = f"""
    На основі наступного User Story: "{user_story}"
    та структури сторінки: {dom_context}
    Згенеруй стабільний тест Playwright на Python.
    Використовуй розумні очікування та data-testid.
    """

    response = openai.chat.completions.create(
        model="gpt-4-turbo",
        messages=[{"role": "user", "content": prompt}]
    )
    return response.choices[0].message.content
```

Рис.1. Приклад методу для автоматичної генерації тестового коду

```python
async def test_add_to_cart(page):
    # Навігація до сторінки товару
    await page.goto('https://example.com/product/123')

    # Очікування завантаження компонента React
    await page.wait_for_selector('[data-testid="product-details"]')

    # Додавання товару до кошика
    await page.click('[data-testid="add-to-cart-button"]')

    # Перевірка оновлення лічильника кошика
    cart_count = await page.text_content('[data-testid="cart-count"]')
    assert cart_count == '1', f"Expected cart count 1, got {cart_count}"

    # Перехід до кошика
    await page.click('[data-testid="cart-icon"]')
    await page.wait_for_selector('[data-testid="cart-items"]')

    # Перевірка наявності товару в кошику
    items = await page.query_selector_all('[data-testid="cart-item"]')
    assert len(items) == 1, f"Expected 1 item in cart, got {len(items)}"
```

Рис. 2. Приклад згенерованого коду для тестування React-додатку

Компонент 2: *модуль візуального тестування* на базі Computer Vision. Цей модуль створює скріншоти інтерфейсу на різних етапах виконання тестів та порівнює їх з еталонними зображеннями або аналізує на предмет візуальних аномалій. Для обробки зображень використовується бібліотека OpenCV, що дозволяє виявляти відмінності в макеті, кольорах, розмірах елементів та інших візуальних характеристиках.

Компонент 3: *центральний компонент*, що координує роботу модулів генерації та візуального тестування, керує виконанням тестових сценаріїв, збирає результати та формує звіти. Цей компонент також відповідає за управління станом браузера, обробку помилок та забезпечення ізоляції між тестовими сценаріями.

Архітектура побудована за модульним принципом, що дозволяє незалежно розвивати та вдосконалювати окремі компоненти, а також легко інтегрувати фреймворк у існуючі CI/CD-конвеєри.

Модуль генерації тестів реалізуватиме підхід, заснований на використанні великих мовних моделей для перетворення високорівневих описів функціональності в код автотестів. Процес генерації складається з наступних етапів:

Етап 1: Аналіз вхідного опису. Модуль приймає на вхід мовний опис тестового сценарію, наприклад: "Перевірити, що користувач може додати товар до кошика, змінити його кількість та оформити замовлення". LLM аналізує цей опис та ідентифікує ключові дії, об'єкти та очікувані результати.

Етап 2: Генерація тестового коду. На основі аналізу LLM генерує Python-код з використанням Playwright API. Код включає: ініціалізацію браузера та навігацію до додатку, що тестується, послідовність дій (кліки, введення тексту, очікування елементів), перевірки очікуваних результатів та обробку помилок.

Етап 3: Адаптація до SPA-специфіки. Модуль враховує особливості SPA-додатків, зокрема використовує очікування динамічних елементів замість статичних затримок, підтримує обробку асинхронних операцій та AJAX-запитів, а також адаптується до змін у віртуальному DOM React/Angular.

Модуль також підтримує ітеративне уточнення згенерованих тестів на основі результатів їх виконання, використовуючи feedback loop для покращення якості генерації [17]. Для підвищення стійкості тестів до змін у DOM-структурі SPA-додатків реалізовано механізм самовідновлення селекторів за допомогою LLM:

```
async def heal_element_selector(page, old_selector, error_msg):
    # Автоматичне відновлення селектора при виникненні помилки Timeout.
    page_source = await page.content()
    prompt = f"Селектор '{old_selector}' не знайдено. Помилка: {error_msg}.
    Знайди найбільш вірогідний новий селектор у наведеному HTML коді:
    {page_source[:5000]}
    Поверни лише рядок з CSS-селектором. "
    response = openai.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}])
    return response.choices[0].message.content.strip()
```

Рис.3. Приклад методу для самовідновлення селекторів за допомогою LLM

```
import numpy as np
from skimage.metrics import structural_similarity as ssim
def compare_screenshots(baseline_path, current_path, threshold=0.95):
    baseline = cv2.imread(baseline_path)
    current = cv2.imread(current_path)
    baseline_gray = cv2.cvtColor(baseline, cv2.COLOR_BGR2GRAY)
    current_gray = cv2.cvtColor(current, cv2.COLOR_BGR2GRAY)
    similarity_index, diff_image = ssim(baseline_gray, current_gray, full=True)
    # Виявлення областей відмінностей
    diff_image = (diff_image * 255).astype("uint8")
    thresh = cv2.threshold(diff_image, 0, 255,  cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
    # Знаходження контурів відмінностей
    contours = cv2.findContours(thresh, cv2.RETR_EXTERNAL,  cv2.CHAIN_APPROX_SIMPLE)
    contours = contours[0] if len(contours) == 2 else contours[1]
    diff_regions = []
    for contour in contours:
        x, y, w, h = cv2.boundingRect(contour)
        if w * h > 100:  # Ігнорування дрібних відмінностей
            diff_regions.append({'x': x, 'y': y, 'width': w, 'height': h})
            cv2.rectangle(current, (x, y), (x + w, y + h), (0, 0, 255), 2)
    return {
        'similarity': similarity_index,
        'passed': similarity_index >= threshold,
        'diff_regions': diff_regions,
        'diff_image': current}
```

Рис.4. Метод порівняння знімків екрану

Модуль візуального тестування реалізує комплексний підхід до виявлення візуальних дефектів та аномалій в інтерфейсі SPA-додатків. Даний модуль виконуватиме декілька функцій. По-перше, він буде автоматично робити знімки екрану ключових станів інтерфейсу під час виконання тестів. Для кожного стану створюється еталонне зображення, яке потім використовується для порівняння при наступних запусках. Алгоритм порівняння використовує кілька метрик:

- Structural Similarity Index (SSIM) для оцінки загальної структурної подібності;
- Pixel-wise difference для виявлення точних змін;
- Histogram comparison для виявлення змін у кольоровій схемі.

По-друге, модуль використовуватиме машинне навчання для ідентифікації та ігнорування елементів, що закономірно змінюються (наприклад, дати, часові мітки, рекламні банери). Це зменшує кількість хибних спрацьовувань [10].

Виявлені візуальні відмінності передаватимуться LLM разом зі знімками екрану для семантичного аналізу та класифікації. LLM визначає, чи є відмінність критичною помилкою, очікуваною зміною функціональності, або несуттєвою варіацією. На рис. 5 наведено приклад коду для семантичного аналізу візуальних відмінностей з використанням мультимодальної моделі GPT-4V:

```
def analyze_visual_diff_with_llm(baseline_b64, current_b64, diff_regions):
  # Використання мультимодальної моделі для аналізу візуальних відмінностей.
  prompt = f"Проаналізуй виділені області {diff_regions} на скріншотах. Чи є ці зміни багом чи очікуваною зміною UI?"
  response = openai.chat.completions.create(
    model="gpt-4-vision-preview",
    messages=[{
        "role": "user",
        "content": [
          {"type": "text", "text": prompt},
          {"type": "image_url", "image_url": f"{baseline_b64}"},
          {"type": "image_url", "image_url": f"{current_b64}"}
        ]}] )
  return response.choices[0].message.content
```

Рис.5. Приклад методу для семантичного аналізу візуальних відмінностей

Експериментальний фреймворк реалізовано на Python 3.11 з використанням наступних бібліотек та архітектурних рішень:

- Playwright 1.40 – для автоматизації браузера та взаємодії з веб-додатками;
- OpenCV 4.8 – для обробки зображень та виявлення візуальних відмінностей;
- OpenAI API – для генерації тестового коду та семантичного аналізу;
- pytest 7.4 – як тестовий фреймворк та test runner;
- scikit-image 0.22 – для розрахунку метрик подібності зображень;
- Асинхронна архітектура. Використання async/await для ефективного управління множинними браузерними контекстами та паралельного виконання тестів;
- Модульна структура. Фреймворк організовано у вигляді незалежних модулів;
- Конфігурація через YAML. Всі параметри фреймворку (API ключі, селектори для ігнорування) винесено в конфігураційні файли;
- Інтеграція з CI/CD. Фреймворк підтримує запуск у headless-режимі та генерацію звітів у форматах JUnit XML та HTML для інтеграції з Jenkins, GitLab CI та GitHub Actions.

Для оцінки ефективності запропонованого підходу проведено експериментальне тестування на трьох SPA-додатках різної складності.

Тестовий додаток 1: React E-commerce – інтернет-магазин на React 18 з Redux для управління станом. Додаток включає каталог товарів, кошик, форму оформлення замовлення та особистий кабінет користувача.

Тестовий додаток 2: Angular Dashboard – адміністративна панель на Angular 16 з Material Design компонентами. Включає графіки, таблиці з пагінацією, форми з валідацією та динамічне оновлення даних.

Тестовий додаток 3: Vue.js Social Network – соціальна мережа на Vue.js 3 з Composition API. Містить стрічку новин, систему коментарів, чат та профілі користувачів.

Для кожного додатку створено набір з 20 описів тестових сценаріїв, що покривають основну функціональність додатку, а також використано LLM-модуль для генерації коду тестів на основі цих описів. Згенеровані тести було виконано з одночасним візуальним тестуванням ключових станів інтерфейсу. Далі було внесено 15 контрольованих змін у кожен додаток (5 функціональних помилок, 5 візуальних дефектів, 5 несуттєвих змін). В результаті була проаналізована здатність фреймворку виявляти внесені зміни та класифікувати їх за типом. Результати експерименту наведено у табл. 1.

Таблиця 1

Результати виявлення помилок

Додаток	Функціональні помилки	Візуальні дефекти	Хибні спрацьовування	Час виконання (хв)
React E-commerce	5/5 (100%)	4/5 (80%)	2	12.3
Angular Dashboard	4/5 (80%)	5/5 (100%)	1	15.7
Vue.js Social Network	5/5 (100%)	4/5 (80%)	3	18.2
Середій показник	93.3%	86.7%	2	15.4

Результати демонструють високу ефективність запропонованого підходу у виявленні як функціональних помилок (93.3% успішності), так і візуальних дефектів (86.7% успішності). Кількість хибних спрацьовувань залишається низькою (в середньому 2 на додаток), що значно краще порівняно з традиційним snapshot-тестуванням [10].

Порівняння з традиційними підходами:

Для порівняння ефективності проведено паралельне тестування тих самих додатків з використанням традиційного підходу (ручне написання тестів з Playwright + snapshot-тестування з Jest).

Таблиця 2

Порівняльний аналіз підходів

Метрика	Гібридний підхід (LLM+CV)	Традиційний підхід
Час розробки тестів (год)	4.2	18.5
Виявлення функціональних помилок	93.3%	100%
Виявлення візуальних дефектів	86.7%	73.3%
Хибні спрацьовування	6	23
Час підтримки при змінах UI (год)	1.8	7.2
Адаптивність до змін DOM	Висока	Низька

Результати показують, що гібридний підхід забезпечує значне скорочення часу розробки тестів (у 4.4 рази) та їх підтримки (у 4 рази) при збереженні високої якості виявлення помилок. Особливо помітна перевага у виявленні візуальних дефектів (+13.4%) та зменшенні хибних спрацьовувань (у 3.8 рази). В той же час експериментальна перевірка виявила ряд обмежень запропонованого підходу. LLM іноді генерує неоптимальні послідовності дій для складних сценаріїв, що вимагають багатокрокової взаємодії з динамічними елементами. Ефективність генерації тестів також суттєво залежить від чіткості та повноти вхідних описів. Неоднозначні або неповні описи призводять до генерації неповних тестів. А тестування responsive-дизайну вимагає створення множини еталонних зображень для різних розмірів екрану, що збільшує складність підтримки.

Ці обмеження визначають напрямки подальшого вдосконалення фреймворку та дослідження альтернативних підходів.

ВИСНОВКИ ДОСЛІДЖЕННЯ ТА ПЕРСПЕКТИВИ ПОДАЛЬШОГО РОЗВИТКУ У ДАНОМУ НАПРЯМКУ

У роботі досліджено застосування методів штучного інтелекту для автоматизації тестування вебдодатків на базі React та Angular. Запропоновано гібридний підхід, що поєднує великі мовні моделі для генерації тестових сценаріїв з методами комп'ютерного зору для візуального тестування інтерфейсів.

Було розроблено архітектуру експериментального фреймворку, що складається з модуля генерації тестів на базі LLM, модуля візуального тестування на базі OpenCV та центрального модуля, що координує їх роботу. Технічна реалізація виконана на Python з використанням Playwright, OpenAI API та OpenCV.

Експериментальна перевірка на трьох тестових SPA-додатках різної складності продемонструвала високу ефективність запропонованого підходу: 93.3% успішності у виявленні функціональних помилок та 86.7% у виявленні візуальних дефектів. Порівняльний аналіз показав, що гібридний підхід забезпечує значне скорочення часу розробки тестів та їх підтримки порівняно з традиційними методами при збереженні високої якості виявлення дефектів. В той же час було виявлено ряд обмежень, зокрема труднощі у тестуванні складних користувацьких взаємодій, залежність від якості написаних природньою мовою описів та обмеження візуального тестування для адаптивних інтерфейсів.

Перспективи подальших досліджень полягають у глибшій інтеграції мультимодульних моделей для повного розуміння семантики та ієрархії складних інтерфейсів користувача. Важливим напрямком є розвиток автономних систем тестування, що дозволить автоматично корегувати сценарії при змінах у DOM-структурі без втручання розробника. Також перспективним є дослідження методів оптимізації вартості та швидкості виконання AI-керованих тестів у великих CI/CD-конвеєрах, а також створення спеціалізованих відкритих наборів даних для оцінки ефективності AI-агентів у тестуванні сучасних вебдодатків.

References

1. Su Z., Bai C., Wei S. Self-Healing UI Test Automation via Multi-Modal Fusion. 2025. DOI: https://doi.org/10.1109/aiita65135.2025.11047603
2. Xu K., Mao Y., Guan X. Web-bench: A llm code benchmark based on web standards and frameworks. arXiv.org. 2025. DOI: https://doi.org/10.48550/arxiv.2505.07473
3. Dobbala S. Validate faster, develop smarter: A review of frontend testing best Practices and Frameworks. 2022. DOI: https://doi.org/10.47363/jmca/2022(1)151
4. Li Y., et al. Test-Agent: A Multimodal App Automation Testing Framework Based on the Large Language Model. 2024. DOI: https://doi.org/10.1109/dtpi61353.2024.10778901
5. Huynh T., et al. Towards Test Generation from Task Description for Mobile Testing with Multi-modal Reasoning. arXiv.org. 2025. DOI: https://doi.org/10.48550/arxiv.2504.15917
6. Liu R., et al. Vision-driven Automated Mobile GUI Testing via Multimodal Large Language Model. 2024. DOI: https://doi.org/10.48550/arxiv.2407.03037
7. Emektar V., et al. AI-Driven Synthetic Test Data and Scenario Generation via GAN-LLM Integration: A Modular Approach for Web Application Testing.
8. Kaynak E., et al. LLMShot: Reducing snapshot testing maintenance via LLMs. arXiv.org. 2025. DOI: https://doi.org/10.48550/arxiv.2507.10062
9. Leotta M., et al. AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study. 2024. DOI: https://doi.org/10.1145/3661167.3661192
10. Stocco A., et al. Visual web test repair. Foundations of Software Engineering. 2018. DOI: https://doi.org/10.1145/3236024.3236063
11. Kanagaraj S., et al. LLM-Driven Smart Test Case Generation for Scalable Software Testing.
12. Alian M., et al. A Feature-Based Approach to Generating Comprehensive End-to-End Tests. 2024. DOI: https://doi.org/10.48550/arxiv.2408.01894
13. Wen M., et al. Enhancing Web Test Script Repair Using Integrated UI Structural and Visual Information. 2024. DOI: https://doi.org/10.1109/icsme58944.2024.00018
14. Feng Y., et al. Enabling Cost-Effective UI Automation Testing with Retrieval-Based LLMs: A Case Study in WeChat. 2024. DOI: https://doi.org/10.48550/arxiv.2409.07829
15. Liu R., et al. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. 2024. DOI: https://doi.org/10.1145/3597503.3639180
16. Foster S., et al. Mutation-Guided LLM-based Test Generation at Meta. 2025. DOI: https://doi.org/10.48550/arxiv.2501.12862
17. Chen J., et al. Standing on the Shoulders of Giants: Bug-Aware Automated GUI Testing via Retrieval Augmentation. Proceedings of the ACM on software engineering. 2025. DOI: https://doi.org/10.1145/3715755
18. Celik A., Mahmoud Q. H. A Review of Large Language Models for Automated Test Case Generation. Machine learning and knowledge extraction. 2025. DOI: https://doi.org/10.3390/make7030097
19. Adaptive Test Generation Using a Large Language Model. 2023. DOI: https://doi.org/10.48550/arxiv.2302.06527
20. Yi G., Chen Z., Chen Z., et al. Exploring the capability of ChatGPT in test generation. 2023. DOI: https://doi.org/10.1109/qrs-c60940.2023.00013
21. Pathak H., Kapoor R. Computer Vision for UI Testing: Leveraging Image Recognition and AI to Validate Elements and Layouts. 2025. DOI: https://doi.org/10.36227/techrxiv.174378309.91394360/v1