Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра математичного моделювання та аналізу даних

*До захисту допущено*
*Кафедрою ММтаАД протокол №___ від_____*

*завідувач кафедри _____ Володимир СТРУКОВ*
*(підпис) (ім'я, прізвище)*

*«___»_____ 2026 р.*

Кваліфікаційна робота
здобувача першого (бакалаврського) рівня вищої освіти

Розробка засобу для генерації автоматизованих тестів з використанням штучного інтелекту
(назва роботи)

Спеціальність (спеціалізація) 122 Комп'ютерні науки

Освітня програма Комп'ютерні науки

Виконавець _____ Іван ЗАЄЦЬ
*(підпис) (ім'я, прізвище)*

Науковий керівник _____ *Олександр Мішин*
*( підпис) (ім'я, прізвище)*

Харків – 2026

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра математичного моделювання та аналізу даних

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПСіТ

_____ Володимир СТРУКОВ
підпис ім'я, прізвище

"____" _____2026 року

**З А В Д А Н Н Я**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ**

здобувача першого (бакалаврського) рівня вищої освіти
(першого (бакалаврського) / другого (магістерського))

Заєць Іван Миколайович
(прізвище, ім'я, по батькові здобувача)

Спеціальність (спеціалізація) 122 Комп'ютерні науки
(код та найменування спеціальності; спеціалізації спеціальності)

Освітня програма Комп'ютерні науки
(назва освітньої програми)

1. Тема роботи: Розробка засобу для генерації автоматизованих тестів з використанням штучного інтелекту

керівник роботи Мішин Олександр Вікторович, старший викладач зво
( прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по Університету від _____2026 року №_____

2. Строк здобувачем роботи "____" _____2026 року

3. Вихідні дані до роботи: Технічне завдання на розробку програмного засобу для генерації автоматизованих тестів вебзастосунків класу Single Page Application (SPA); вимоги до архітектури системи (патерн Page Object Model) та базового синхронного технологічного стека (Python, Playwright Python API, фреймворк PyTest, структуровані конфігураційні файли JSON); локальні бази знань та інструкції з проектування тестів; вимоги щодо інтеграції великих мовних

моделей (Google Gemini API, модель gemini-2.5-flash) для контекстного синтезу тестових сценаріїв та автоматичного розрахунку аналітичних метрик покриття інтерфейсу.

4. Перелік питань, які потрібно розробити:

1) ознайомитись з особливостями проектування та автоматизації E2E-тестів для SPA-додатків за допомогою патерну Page Object Model (POM), а також вивчити теоретичні засади та виклики класичного тестування інтерфейсів.

2) провести огляд існуючих методів та спеціалізованих хмарних ШІ-інструментів автоматизації тестування;

3) познайомитися з алгоритмом роботи і методиками динамічного неблокуючого сканування DOM-дерева, створити модель та програмну архітектуру автоматичного аналізатора структури вебресурсів;

4) провести програмну реалізацію та експериментальне дослідження інтелектуального оркестратора на базі Google Gemini API для генерації стабільних, параметризованих PyTest сценаріїв у синхронному синтаксисі Python із автоматичним розрахунком аналітичних метрик покриття.

5. План роботи

| № з/п | Назви етапів роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР). | 12.02 – 27.02 | |
| 2 | Пошук та аналіз інформаційних джерел за темою КР. | 28.02– 14.03 | |
| 3 | Визначення особливостей функціонування динамічних інтерфейсів SPA-додатків та аналізу існуючих хмарних ШІ-інструментів тестування. | 15.03 – 31.03 | |
| 4 | Розробка моделей архітектурної взаємодії та алгоритмів модулів сканування DOM-структури та оркестрації ШІ-генерації. | 01.04 – 15.04 | |
| 5 | Програмна реалізація динамічного аналізатора коду, підсистеми інженерії промптів та інтеграції з хмарним інтерфейсом Google Gemini API. | 16.04 – 05.05 | |
| 6 | Експериментальні дослідження розробленого засобу ШІ-генерації на базі цільових вебресурсів та оцінка метрик покриття інтерфейсу. | 06.05 – 18.05 | |
| 7 | Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ШІ. | 18.05 – 01.06 | |

6. Дата видачі завдання ____ _____ 2026 р.

Здобувач вищої освіти _____ І. М. Заєць
(підпис) (ініціали, прізвище)

Керівник роботи _____ О. В. Мішин
( підпис) (ініціали, прізвище)

# АНОТАЦІЯ

Пояснювальна записка містить 55 сторінок, 3 розділи, 7 рисунків, 1 додаток, 18 джерел за переліком посилань.

Мета роботи – проектування та програмної реалізації автоматизованого засобу генерації наскрізних тестів динамічних вебзастосунків.

Методи дослідження і перелік використаних програмно-апаратних рішень: методи динамічного аналізу об'єктної моделі документа (DOM), алгоритми семантичної редукції контексту, інженерія промптів та еквівалентне розбиття тестових даних. Програмна реалізація виконана мовою Python із застосуванням фреймворків PyTest, Playwright та хмарного інтерфейсу Google Gemini API.

Результати та їх новизна: розроблено засіб, який автоматично сканує SPA-додатки, фільтрує візуальний шум та генерує тестові класи патерну POM. Новизна полягає у застосуванні архітектурного підходу редукції контексту (конвертації HTML у JSON) перед передачею даних до великої мовної моделі, що усуває проблему галюцинацій нейромережі та гарантує синтез синтаксично коректного коду.

Рекомендації щодо використання результатів роботи: засіб рекомендовано до використання командами забезпечення якості (QA) як AI-асистент, а також для інтеграції в конвеєри CI/CD з метою прискорення регресійного тестування.

АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, ВЕБЗАСТОСУНОК, ВЕЛИКА МОВНА МОДЕЛЬ, ПАТЕРН, ФРЕЙМВОРК.

# ABSTRACT

The explanatory note contains 55 pages, 3 chapters, 7 figures, 1 appendix, and 18 sources according to the list of references.

The purpose of the work is to increase the efficiency of development and reduce the time spent on maintaining end-to-end tests for dynamic web applications by creating an autonomous intelligent automated test generation tool.

Research methods and hardware-software solutions: methods of dynamic Document Object Model (DOM) analysis, semantic context reduction algorithms, prompt engineering, and test data equivalence partitioning. The software implementation is performed in Python using PyTest, Playwright frameworks, and the Google Gemini API.

Results and their novelty: a tool was developed that autonomously scans SPA applications, filters visual noise, and generates POM test classes. The novelty lies in the application of the context reduction architectural approach (HTML to JSON conversion) before transmitting data to the large language model, which eliminates neural network hallucinations and guarantees the synthesis of syntactically correct code.

Recommendations for use: the tool is recommended for use by QA teams as an AI assistant, and for integration into CI/CD pipelines to accelerate regression testing.

AUTOMATED TESTING, ARTIFICIAL INTELLIGENCE, WB APPLICATION, LARGE LANGUAGE MODEL, PATTERN, FRAMEWORK.

# ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ .....8
ВСТУП.....9
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ МЕТОДІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ.....13
1.1. Теоретичні основи та архітектурні виклики наскрізного тестування динамічних вебзастосунків.....13
1.2 Огляд існуючих ШІ-інструментів тестування.....15
1.3. Принципи застосування генеративного штучного інтелекту для створення програмного коду.....19
Висновки до розділу 1.....22
РОЗДІЛ 2. ПРОЄКТУВАННЯ ЗАСОБУ ГЕНЕРАЦІЇ ТЕСТІВ З ВИКОРИСТАННЯМ ШІ.....24
2.1 Декомпозиція задачі генерації автоматизованих E2E-тестів.....24
2.2. Математичне обґрунтування метрик покриття інтерфейсу.....26
2.3 Алгоритмічна модель динамічного сканування вебсторінок.....28
2.4. Архітектура інтелектуального оркестратора та підсистема інженерії промптів.....30
2.5. Механізми валідації, ізоляції та структурування артефактів тестування..32
Висновки до розділу 2.....36
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ, АНАЛІЗ РЕЗУЛЬТАТІВ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА.....38
3.1 Програмна реалізація.....38
3.2. Експериментальне дослідження на базі цільового вебресурсу.....41
3.3. Оцінка одержаних результатів та аналіз недоліків.....48
Висновок до розділу 3.....49
ВИСНОВКИ.....51
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ.....53
ДОДАТОК А. Вихідний код.....55

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ШІ – штучний інтелект

API – Application Programming Interface (інтерфейс програмування додатків)

AST – Abstract Syntax Tree (абстрактне синтаксичне дерево)

CI/CD – Continuous Integration / Continuous Delivery (безперервна інтеграція та доставка)

DOM – Document Object Model (об'єктна модель документа)

E2E – End-to-End (наскрізне тестування)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

LLM – Large Language Model (велика мовна модель)

POM – Page Object Model (патерн проектування тестових моделей)

QA – Quality Assurance (забезпечення якості)

SPA – Single Page Application (односторінковий вебзастосунок)

UI – User Interface (інтерфейс користувача)

# ВСТУП

**Актуальність теми.** Актуальність теми зумовлена стрімким розвитком методів машинного навчання та його використання на різних етапах розробки програмного забезпечення. Не виключенням є використання штучного інтелекту для генерації автоматизованих тестів, що є дуже доцільним у життєвому циклі програми, бо може допомогти скоротити час та зберегти ресурси на забезпечення якості програмного продукту. Генеративні моделі дозволяють не лише зменшити витрати на тестування, але й підвищити його ефективність, зокрема за рахунок автоматичної генерації тест-кейсів, оптимізації сценаріїв тестування та передбачення дефектів.

Розвиток архітектурних шаблонів розподілених програмних систем у бік клієнт-орієнтованих високодинамічних вебзастосунків класу Single Page Application (SPA) зумовив докорінну перебудову методологій інженерії якості. Для забезпечення функціонування таких систем потребує впровадження ефективних методологій контролю якості, де провідне місце посідає наскрізне (E2E) тестування. Декларативне виконання тестових сценаріїв дозволяє мінімізувати вплив людського фактора, прискорити цикли регресійного аналізу та оптимізувати конвеєри безперервної інтеграції та доставки (CI/CD). Проте традиційні підходи до автоматизації інтерфейсів стикаються з критичними викликами. Асинхронний рендеринг та постійна мутація об'єктної моделі документа (DOM) призводять до колосальної трудомісткості проектування скриптів та високої крихкості локаторів. Найменші зміни в ієрархічній структурі DOM-дерева спричиняють десинхронізацію сценаріїв та виникнення ефекту нестабільних результатів (flaky tests). Внаслідок цього QA-інженери змушені витрачати до половини робочого часу не на проектування нових перевірок, а на перманентний рефакторинг та відновлення «зламаних» локаторів, що нівелює концепцію безперервної інтеграції та доставки (CI/CD).

Перспективним напрямком подолання проблем автоматизованого тестування є використання методів машинного навчання та великих мовних моделей. Існують багато сучасних засобів, які використовують різні методи та впроваджують різні алгоритми для аналізу та реалізують механізми самовідновлення локаторів UI-елементів. Існуючі рішення мають низку суттєвих проблем, що не заперечують їх переваги у швидкості розгортання. Все ж більшість із них унеможливлює проведення глибокого інженерного аудиту генерованого коду. Жорстка прив'язка до пропрієтарної хмарної інфраструктури створює ризики інформаційної безпеки і потребує значних ліцензійних витрат. Основною проблемою залишається неспроможність таких систем генерувати прозорий, повністю контрольований тестовий фреймворк у вигляді відкритого синхронного коду (наприклад, мовою Python), який інженери могли б безперешкодно інтегрувати в локальні репозиторії.

Створення відкритого, автоматизованого програмного інструменту забезпечує потребу у швидкій генерації інтелектуальних тестів і вирішує проблему з архітектурними обмеженнями закритих платформ. Цей засіб має забезпечувати сканування структури веб-інтерфейсу, надійно ізолювати інтерактивні елементи та делегувати логіку побудови тестів генеративним моделям, надаючи результат у вигляді стандартизованої кодової бази.

Об'єкт дослідження − процес забезпечення якості та автоматизованого наскрізного (End-to-End) тестування користувацьких інтерфейсів сучасних динамічних вебзастосунків.

Предмет дослідження − методи, алгоритми та програмні засоби на базі великих мовних моделей та інтелектуального парсингу DOM-дерева, призначені для автоматичного синтезу, оптимізації та стабілізації наскрізних тестових сценаріїв.

Мета дослідження − проектування та програмної реалізації автоматизованого засобу генерації наскрізних тестів динамічних вебзастосунків,

здатного генерувати архітектурно коректні тестові фреймворки у стеку Python, Pytest та Playwright із використанням генеративного штучного інтелекту.

Для досягнення поставленої мети визначено та послідовно вирішено такі завдання дослідження:

1. Дослідити теоретичні засади автоматизації тестування інтерфейсів вебзастосунків класу SPA та особливості застосування архітектурного патерну Page Object Model (POM).

2. Провести порівняльний аналіз існуючих інтелектуальних інструментів забезпечення якості програмного забезпечення.

3. Розробити концептуальну архітектуру та математичні метрики оцінки повноти сканування та комбінаторного покриття сценаріїв.

4. Здійснити програмну реалізацію алгоритмічної моделі динамічного аналізу для неблокуючого сканування вебсторінок та екстракції стійких селекторів.

5. Спроектувати архітектуру оркестрації, яка реалізує взаємодію з великою мовною моделлю (Google Gemini API) через механізми інжинірингу промптів та ізольованих баз правил, для синтезу параметризованих тест-кейсів у синхронному синтаксисі Python.

6. Виконати експериментальне дослідження розробленого програмного засобу на базі цільових вебресурсів, проаналізувати стабільність генерованого коду та сформувати рекомендації щодо його впровадження.

Наукова новизна одержаних результатів полягає у подальшому розвитку методів взаємодії детермінованих тестових систем із стохастичними мовними моделями. Вперше запропоновано та програмно реалізовано підхід семантичної редукції контексту (математичної конвертації «сирої» HTML-структури у цільовий JSON-зліпок із пріоритетизацією стійких локаторів) перед передачею даних до LLM. Це дозволило принципово знизити рівень логічних галюцинацій нейромережі та гарантувати генерацію синтаксично коректного, лінійного коду.

Практичне значення одержаних результатів визначається створенням функціонуючого програмного засобу, готового до інтеграції у виробничі процеси. Розроблена система автоматично сканує об'єктну модель сторінки та ініціює миттєвий синтез чистого, структурованого коду мовою Python. Згенерований фреймворк повністю відповідає індустріальним вимогам патерну POM, підтримує параметризацію синтетичних тестових даних і легко інтегрується у локальні системи контролю версій. Запровадження даного рішення усуває залежність від комерційних SaaS-платформ та дозволяє скоротити витрати робочого часу QA-інженерів на підтримку селекторів.

# РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ МЕТОДІВ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

## 1.1. Теоретичні основи та архітектурні виклики наскрізного тестування динамічних вебзастосунків

Розробка програмного забезпечення за останні роки відчутно змінилася, розвинувшись від монолітних архітектур до мікросервісних розподілених систем та високонавантажених клієнт-орієнтованих інтерфейсів. Тестування програмного забезпечення – вид діяльності в процесі розробки, який пов'язаний з виконанням процедур, спрямованих на виявлення (доказ наявності) помилок (невідповідностей, неповноти, двозначностей і т.д.) в поточному визначенні програмної системи, що розробляється [1]. У контексті сучасних підходів до забезпечення якості (QA) найбільш ресурсомістким, але водночас найважливішим етапом є наскрізне тестування (E2E). Цей вид тестування передбачає перевірку системи в цілому з точки зору кінцевого користувача, імітуючи його реальну поведінку в браузері від моменту авторизації до завершення бізнес-сценарію.

Головним каталізатором ускладнення процесу E2E-тестування став масовий перехід індустрії до архітектури односторінкових вебзастосунків (Single Page Application, SPA). SPA – це вебзастосунки, що взаємодіють з користувачем через динамічне переписування поточної сторінки (DOM) замість повного завантаження нових сторінок із сервера. Використання таких фреймворків, як React, Angular та Vue.js, дозволяє реалізувати складне управління клієнтським станом (client-side state management) та маршрутизацію [2]. Але саме ці переваги створюють так звану проблему крихкості локаторів (locator fragility).

Крихкість локаторів виникає тому, що сучасні SPA-фреймворки генерують DOM-дерево динамічно. Збирачі модулів (наприклад, Webpack або Vite) в процесі компіляції мініфікують та обфускують вихідний код, хешуючи назви

CSS-класів та ідентифікаторів. Відповідно, традиційні методи пошуку елементів (селектори XPath або CSS) стають ненадійними. Будь-яка зміна у візуальній розмітці або ієрархії компонентів призводить до того, що автоматизований тест не може знайти цільовий елемент і завершується помилкою (наприклад, TimeoutError). Дослідження показують, що в сучасних SPA-системах значна частина E2E-тестів стає нестабільною (flaky tests) саме через асинхронний рендеринг та мутацію DOM-об'єктів, а не через реальні дефекти бізнес-логіки.

Ефект нестабільних результатів вимагає постійного ручного втручання QA-інженерів для рефакторингу кодової бази, що нівелює головну перевагу автоматизації – швидкість зворотного зв'язку в конвеєрах безперервної інтеграції та доставки (CI/CD). Стандартом, який частково пом’якшив цю проблему став архітектурний патерн Page Object Model (POM). POM – це поширений шаблон проєктування для автоматизованих UI-тестів, що допомагає організувати код і зменшити дублювання. Ідея POM полягає в тому, щоб представити кожну веб сторінку (або значущу частину інтерфейсу) у вигляді окремого класу – так званого Page Object. Використання POM дозволяє відокремити тестові сценарії від технічних деталей UI-розмітки: якщо структура DOM змінюється, оновлення локатора відбувається локалізовано в одному класі, не зачіпаючи логіку десятків залежних тестів. Проте важливо зазначити, що POM лише оптимізує підтримку коду, але не вирішує першопричину проблеми крихкості.

Для стабілізації тестових фреймворків перейшли від класичних інструментів до нових. Фреймворк Selenium WebDriver, що був ключовим у сфері UI-автоматизації тривалий час, працює а принципом синхронних HTTP-запитів до браузерного драйвера. Через це розробники вимушені імплементувати складні конструкції явних очікувань (Explicit Waits) для обробки асинхронності SPA, що робить код громіздким та складним. Вирішує цю проблему більш сучасний інструмент Playwright, який маж значну архітектурну перевагу. Працюючи через протокол WebSocket безпосередньо з Chrome DevTools Protocol (CDP), Playwright

реалізує механізм автоматичного очікування (auto-waiting), забезпечує нативну підтримку ізольованих браузерних контекстів (Browser Contexts) та ефективно взаємодіє із тіньовим DOM (Shadow DOM). Порівняльний аналіз наглядно демонструє, що Playwright виконує скрипти значно швидше за Selenium і суттєво зменшує кількість хибних спрацювань [4].

З огляду на всі фактори, що зазначені вище, було прийнято архітектурне рішення, яке зменшить кількість недоліків у розроблюваному засобі генерації автоматизованих тестів. По-перше, для подолання крихкості локаторів вирішено налаштувати на пошук виключно стабільних атрибутів ідентифікації елементів (таких як data-test, data-testid, data-qa) модуль аналізу вебзастосунку. Атрибутів ідентифікації елементів розробниками інтегруються спеціально для тестування та вони на підпадають мутаціям під час рендерингу чи стилізації, що звичайно є дуже зручним для автоматизованого тестування. По-друге, генерація виконуваного коду спрямована на екосистему Playwright у поєднанні з фреймворком PyTest, що дозволяє згенерувати максимально стабільні та стійкі до коливань інтерфейсу тестові скрипти.

### 1.2 Огляд існуючих ШІ-інструментів тестування

Колосальні часові витрати на підтримку E2E-тестів стимулювали появу нового покоління програмних продуктів на перетині технологій забезпечення якості та штучного інтелекту. Впровадження ШІ в тестування програмного забезпечення – це практика застосування машинного навчання, великих даних та відповідної аналітики для створення, виконання та оновлення тестів без постійного втручання людини, що має певні переваги. Застосування ШІ в тестуванні не замінює повністю тестувальників, але трансформує їхню роль, дозволяючи зосередитися на більш складних завданнях, таких як дослідницьке тестування, аналіз ризиків та стратегічне планування. Це підвищує загальну якість програмного забезпечення

Саме усвідомлення обмежень використання виключно скриптової автоматизації та високої вартості підтримки POM-архітектур призвело до стрімкої появи різноманіття інструментів нового покоління. Сучасні ШІ-інструменти в QA застосовують підходи машинного навчання, глибокого навчання, комп'ютерного зору та обробки природної мови для імітації функцій тестувальника, розпізнавання патернів у DOM-дереві, виявлення аномалій та автоматичного прийняття рішень під час прогону тестів.

Серед інструментів, що фокусуються на використанні алгоритмів комп'ютерного зору, безперечними лідером є Applitools [5]. На відміну від традиційного попіксельного порівняння, Applitools використовує комп'ютерний зір та машинне навчання, щоб розуміти візуальний контекст інтерфейсу користувача, виявляючи лише значущі зміни та регресії. Цей інструмент інтегрується з існуючими фреймворками автоматизації, такими як Selenium, Cypress, Playwright, та Appium, дозволяючи додавати візуальну перевірку до вже існуючих функціональних тестів. Але цей інструмент має значні недоліки: він не є повноцінним інструментом для функціональної автоматизації; фокусується лише на візуальному аспекті; також це комерційний інструмент, що може бути дорогим; і потребує деяких навичок кодування для інтеграції в існуючі тестові фреймворки.

Найбільший розвиток мають ШІ-інструменти, що в тестуванні використовують комплексні SaaS-платформами (Software as a Service), які пропонують підхід мінімального кодування ("Low-code") або його повної відсутності ("No-code"). До цієї групи належать такі потужні інструменти, як mabl [6], Functionize [7], та Testim [8].

Головною інноваційною особливістю цих платформ є механізм «самовідновлення» (self-healing) локаторів. Замість того, щоб жорстко прив'язуватися до єдиного селектора (як це робить класичний підхід), платформа збирає десятки атрибутів про кожен взаємодіючий елемент: його просторові

координати, текстовий вміст, тип тегу, сусідні вузли в дереві DOM, CSS-властивості та поведінкову історію. Коли розробники випускають оновлення, яке змінює ID або CSS-клас цільової кнопки, традиційний скрипт Selenium або Playwright неминуче зазнав би краху з виключенням TimeoutError або NoSuchElementException. Натомість алгоритми машинного навчання платформ mabl чи Functionize оцінюють ймовірність збігу об'єкта на оновленій сторінці за сукупністю інших збережених ознак. Якщо ШІ знаходить елемент, що має високий показник подібності, він автоматично оновлює еталонний локатор "на льоту", дозволяючи тесту успішно продовжити виконання і повідомляючи інженера про внесені зміни постфактум. Testim (нині частина Tricentis) базується на аналогічних принципах "розумних локаторів", проте пропонує більш гнучку архітектуру для технічних спеціалістів, дозволяючи впроваджувати кастомні блоки коду JavaScript на окремих етапах виконання тестів для обробки нестандартної логіки.

Новітня хвиля інструментів використовує парадигму NLP та генеративного ШІ. Платформа testRigor [9] – це інструмент на основі ШІ, який дозволяє користувачам писати тести простою англійською мовою. Розробникам і тестувальникам стає легко оптимізувати процес розробки тестів. Наявність ШІ також автоматизує процес обслуговування, забезпечуючи кращу точність [10]. Тестування вебзастосунків завдяки цьому ШІ-інструменту стає зручнішим та плавнішим, бо інструмент має функції: самовідновлення, тестування реальних пристроїв, також ідеально інтегрується з CI/CD конвеєрами.

Більш просунуті системи, такі як Relicx[11], ContextQA[12] та Momentic[13]., інтегрують можливості великих мовних моделей у форматі інтерактивних агентів (AI agents) або "копілотів" (Copilots). Вони здатні аналізувати реальні користувацькі сесії, генерувати тестові плани та самостійно адаптуватися до складних змін в інтерфейсі, виконуючи рутинну роботу автономних тестувальників. Платформа Roost.ai[14] зосереджена на

використанні LLM (таких як Vertex AI або GPT-4) для автоматичного оновлення тестів при змінах у вихідному коді та управління покриттям API.

Впровадження описаних вище систем дійсно здатне значно знизити поріг входження в автоматизацію (дозволяючи залучати спеціалістів без глибоких навичок програмування) та радикально пришвидшити початковий етап створення тестових наборів. Однак аналіз інженерного, архітектурного та економічного аспектів виявляє низку критичних проблем та недоліків, які роблять ці рішення неприйнятними або фінансово необґрунтованими для значної частини команд розробки програмного забезпечення.

Абсолютна більшість згаданих рішень (mabl, Testim, testRigor, Applitools) є закритими, пропрієтарними комерційними продуктами. Тести, створені в цих платформах, зберігаються у внутрішньому форматі вендора (часто у вигляді візуальних блоків або пропрієтарних JSON-структур). Це повністю унеможливлює проведення класичного перехресного огляду коду (Code Review) через системи контролю версій (наприклад, Git) за звичними для розробників правилами інтеграції коду (Pull/Merge Requests). Практично неможливо мігрувати накопичену базу з тисяч тестів на інший інструмент без повної втрати результатів багатомісячної праці. Також більшість платформ не здатні згенерувати повноцінний, об'єктно-орієнтований, стандартизований програмний код (наприклад, у стеку Python + Pytest + Playwright з використанням POM), який інженер міг би інтегрувати в локальний репозиторій проєкту, налаштувати під специфічні потреби безпеки та вільно модифікувати без огляду на обмеження платформи-генератора.

Проблемою є неможливість достеменно зрозуміти алгоритм, за яким штучний інтелект приймає рішення про ідентифікацію елемента під час самовідновлення, що знижує рівень довіри до результатів тестування з боку розробників. Тест, який формально пройшов успішно завдяки самовідновленню

локатора, може насправді клікнути на хибний елемент, що візуально або семантично схожий на цільовий пропускаючи таким чином критичний дефект.

Важливим негативним фактором також є те, що комерційні ШІ-інструменти вимагають значних ліцензійних витрат, які часто масштабуються залежно від кількості виконань тестів (test runs), кількості паралельних потоків або кількості зареєстрованих користувачів. Більше того, виконання тестів у сторонніх хмарних інфраструктурах створює серйозні ризики корпоративної інформаційної безпеки, оскільки вимагає надання доступу до внутрішніх, закритих тестових середовищ (які можуть містити конфіденційні дані або комерційну таємницю) зовнішнім SaaS-провайдерам.

На підставі проведеного огляду випливає висновок: на ринку існує нагальна та незадоволена потреба у розробці автоматизованого інструменту, який би поєднував потужність алгоритмів штучного інтелекту зі здатністю генерувати відкритий, прозорий та архітектурно правильний програмний код. Цей інструмент не повинен підміняти собою кодову базу закритими абстракціями, а навпаки – має виступати в ролі інтелектуального помічника (генератора), який створює готовий, надійний фундамент (фреймворк) для подальшої роботи інженерів-автоматизаторів безпосередньо в їхньому локальному робочому середовищі.

### 1.3. Принципи застосування генеративного штучного інтелекту для створення програмного коду

Необхідним кроком, для забезпечення здатності розроблюваного інструменту автоматично генерувати архітектурно коректний код, є аналіз теоретичних засад застосування великих мовник моделей в інженерії програмного забезпечення.

Впровадження генеративного ШІ у процеси розробки та тестування формує новий вид взаємодії в програмному середовищі – Prompt Engineering. На відміну

від традиційного програмного забезпечення, яке спирається на формальні мови зі строгим синтаксисом та детермінованими середовищами виконання, prompt-програмування базується на неструктурованій природній мові, де середовищем виконання виступає ймовірнісна та недетермінована мовна модель [15]. Великі мовні моделі (наприклад, архітектури Transformer) навчаються на терабайтах коду та тексту, формуючи глибоке статистичне розуміння мовних закономірностей. Проте, їхня ймовірнісна природа породжує проблему «галюцинацій» – генерації синтаксично правильного, але логічно хибного або неіснуючого коду.

Для мінімізації галюцинацій і забезпечення стабільного синтезу E2E-тестів критично важливим є розуміння двох аспектів: редукції контексту (Context Reduction) та оптимізації промптів (Few-Shot Prompting).

Кожна велика мовна модель має обмеження на розмір контекстного вікна (Context Window) – максимальну кількість токенів, яку вона здатна обробити за один запит. Хоча сучасні моделі, такі як gemini-2.5-flash, підтримують величезні вікна (до 1 мільйона токенів) [16], передача повного вихідного HTML-коду сучасного SPA-додатка до API моделі є неефективною і хибною практикою.

HTML-код містить величезну кількість візуального шуму: вбудовані SVG-іконки, довгі масиви CSS-класів, невидимі контейнери та технічні скрипти. Недоцільно перевантажувати моделі таким неструктурованої інформації – це призводить до ефекту деградації механізму уваги. Модель починає губитися в контексті, втрачаючи здатність ідентифікувати семантично значущі інтерактивні елементи, що призводить до генерації хибних локаторів та нестабільних тестів (flaky tests). Без доступу до конкретного, очищеного контексту інтерфейсу підходи на базі виключно LLM страждають від галюцинацій[3].

У розроблюваному інструменті ця теоретична проблема вирішується архітектурно через модуль динамічного аналізу. Скрипт виконує роль інтелектуального препроцесора: він сканує DOM-дерево вебсторінки та за

допомогою алгоритмів парсингу відкидає весь нерелевантний візуальний шум. Екстракції підлягають лише значущі вузли взаємодії (кнопки, поля вводу, посилання, форми), причому пріоритет віддається стабільним локаторам data-test. Результатом роботи аналізатора є стислий, структурований JSON-сніппет. Таким чином, мовна модель отримує не сирий HTML, а очищену математичну репрезентацію об'єктної моделі сторінки, що радикально знижує використання токенів, прискорює генерацію та майже повністю усуває ризик галюцинацій при написанні селекторів.

Другим ключовим елементом є методика формування запиту до моделі. Традиційний підхід Zero-Shot Prompting (запит без прикладів) часто призводить до помилок компіляції (TypeError, ImportError), оскільки модель може ігнорувати специфічні вимоги до структури коду або використовувати застарілі бібліотеки[17].

Для забезпечення синтезу архітектурно коректного коду застосовується передова методика Few-Shot Prompting (навчання на кількох прикладах у контексті). Ця техніка передбачає включення до промпту кількох еталонних пар "вхідні дані – вихідний код". Дослідження підтверджують, що додавання структурованих прикладів значно підвищує точність згенерованого коду та гарантує дотримання заданих синтаксичних парадигм. Завдяки In-Context Learning (навчанню в контексті), LLM здатна вловити необхідний патерн програмування без додаткового дороговартісного донавчання (fine-tuning) своїх параметрів.

Описані принципи формують наукове підґрунтя для роботи центрального генератора розроблюваного засобу та бази ізольованих правил. Використання моделі gemini-2.5-flash через Google Gemini API є оптимальним вибором завдяки її здатності швидко обробляти структурні завдання. Програмний оркестратор об'єднує очищений JSON-сніппет (отриманий від DOM-аналізатора) та системний промпт із бази ai-rules, який реалізує методику Few-Shot Prompting,

задаючи моделі чіткі інструкції та еталонні приклади архітектури Page Object Model.

Відтак, доведено, що генеративний штучний інтелект здатний генерувати абсолютно стабільний, синхронний та безпечний PyTest/Playwright код за умови, якщо алгоритмічно обмежити його свободу дій шляхом семантичної редукції вхідного контексту (HTML → JSON) та строгої типізації промптів (Few-Shot). Цей підхід нівелює недоліки інструментів класу «чорна скринька», надаючи інженерам прозорий, керований та високоякісний програмний продукт для локального використання.

**Висновки до розділу 1**

Проведений детальний аналіз предметної області засвідчив, що стрімка еволюція вебтехнологій та масовий перехід до розподілених архітектур Single Page Application (SPA) зумовили кризу традиційних підходів до забезпечення якості програмного забезпечення. Попри те, що використання сучасних інструментів автоматизації (Playwright) та впровадження архітектурного патерну Page Object Model значно покращує структуру тестових фреймворків, процес E2E-тестування залишається критично вразливим до проблеми крихкості локаторів. Нестабільність DOM-дерева породжує феномен "flaky tests", що вимагає постійного ручного втручання та рефакторингу коду.

Огляд та класифікація існуючих комерційних ШІ-інструментів тестування (Mabl, Testim, Functionize, testRigor тощо) показали, що індустрія успішно застосовує методи машинного навчання для алгоритмічного самовідновлення локаторів та обробки природної мови. Проте ці платформи мають фундаментальні недоліки корпоративного характеру: вони створюють жорстку залежність від хмарної інфраструктури постачальника (vendor lock-in), функціонують за принципом «чорної скриньки», що унеможливлює технічний аудит, і головне – не дозволяють експортувати згенеровані тести у вигляді

чистого, відкритого та підтримуваного коду (наприклад, на мові Python). Це позбавляє інженерні команди контролю над власними процесами тестування.

Вирішення цієї проблематики полягає у використанні потужностей LLM для автоматизовного синтезу програмного коду на базі принципів Prompt Engineering. Теоретичний аналіз довів, що пряма передача сирого HTML-коду моделям призводить до перевантаження контексту та галюцинацій. Натомість, застосування механізмів інтелектуальної редукції контексту (екстракція лише значущих data-атрибутів з DOM у формат JSON) у поєднанні з методикою інженерії промптів (Few-Shot Prompting) дозволяє досягти високої детермінованості результату.

Отже, створення відкритого, автомтизованого програмного інструменту, здатного алгоритмічно парсити структуру вебресурсу та використовувати API генеративної моделі (наприклад, gemini-2.5-flash) для миттєвого синтезу параметризованого коду у стеку Python, PyTest та Playwright, є практично обґрунтованим завданням. Таке рішення усуне недоліки закритих платформ і забезпечить інженерів надійним механізмом для швидкого створення стійких E2E-фреймворків. Наведене теоретичне підґрунтя створюють базу для проєктування архітектури та алгоритмів функціонування розроблюваного засобу, що детально розглядається у наступному розділі.

# РОЗДІЛ 2. ПРОЄКТУВАННЯ ЗАСОБУ ГЕНЕРАЦІЇ ТЕСТІВ З ВИКОРИСТАННЯМ ШІ

## 2.1 Декомпозиція задачі генерації автоматизованих E2E-тестів

Проблема автоматичного генерування стабільних E2E-тестів для динамічних вебзастосунків класу SPA належить до класу слабоструктурованих задач інженерії програмного забезпечення. Як було встановлено під час аналізу предметної області у першому розділі, традиційні методи часто дають збій через постійну зміну структури сторінок. Вирішення цієї задачі за допомогою ШІ має значну складність через глибоке протиріччя між імовірнісною природою LLM та абсолютною детермінованістю, якої вимагають архітектурні патерни тестових фреймворків.

Фронтенд-фреймворки, такі як React і Angular, рендерять об'єктну модель документа асинхронно. Вони постійно змінюють ідентифікатори вузлів, генерують динамічні класи та перемальовують компоненти без перезавантаження сторінки. атомість генеративний ШІ дуже чутливий до формату вхідних даних і схильний до галюцинацій при обробці надлишкового контекстного шуму. Якщо передати мовній моделі стандартний, HTML-код SPA-додатка, масиви несемантичних CSS-класів, вбудованих SVG-графік або прихованих скриптів перевантажать контекстне вікно моделі. Це з високою ймовірністю призведе до генерації неробочих локаторів та хибної логіки взаємодії.

Було виконано декомпозицію загального процесу генерації тестів для ефективного вирішення цього архітектурного конфлікту та забезпечення стабільності результату. Замість монолітного підходу, розроблений програмний продукт базується на концепції розділення обов'язків (Separation of Concerns). Систему спроєктовано на основі дворівневої ізольованої архітектури, що складається з двох взаємопов'язаних, але незалежних модулів:

1. Модуль детермінованого динамічного аналізу (Фронтенд-сканер `SiteAnalyzer`) – він відповідає за фізичне виконання JavaScript-ін'єкцій у контексті браузера, фільтрацію візуального шуму та отримання виключно семантично значущих атрибутів DOM-дерева у стандартизований формат JSON. Завдяки цьому модулю ШІ не взаємодіє зі складним веб-середовищем безпосередньо, а отримує лише чисті, структуровані дані про інтерактивні елементи (кнопки, поля вводу, посилання, навігацію).

2. Модуль інтелектуальної оркестрації (Бекграунд-генератор `AITestGenerator`) – цей модуль відповідає за управління контекстним вікном мовної моделі, агрегацію локальних баз правил (ai-rules) та взаємодію з Google Gemini API. Його головна задача полягає у передачі отриманого JSON-знімка сторінки та системних інструкцій до LLM, а також у кінцевому отриманні, обробці та збереженні виконуваного коду мовою Python.

Важливим інженерним рішенням на етапі проектування стала відмова від повністю асинхронного стека (async/await) на користь строгого синхронного синтаксису Python Playwright (бібліотека `sync_playwright`), що позитивно вплинуло на надійність продукту. Незважаючи на загальноприйняту практику використання асинхронності для оптимізації мережевих запитів, в умовах створення коду машинним інтелектом застосування асинхронних патернів суттєво збільшує ризик виникнення станів гонитви (race conditions) всередині згенерованих сценаріїв. ля LLM генерувати складний ланцюжок асинхронних викликів і керувати циклом подій (event loop) значно важче, що створює ризик логічних помилок для нашого засобу.

Синхронний API примусово серіалізує потік виконання команд браузера. Це рішення дозволило значно стабілізувати середовище та спростити AST-дерево цільового коду, знизивши когнітивне навантаження на LLM під час побудови класів POM. Завдяки цьому модель генерує лінійний, передбачуваний та зрозумілий код, який легко читати, підтримувати та масштабувати QA-інженеру.

Відмова від використання закритих хмарних SaaS-платформ, недоліки яких були описані в огляді існуючих рішень, на користь локальної оркестрації гарантує повну ізоляцію артефактів тестування та відсутність ризиків витоку інформації про структуру цільових вебресурсів, завдяки надсиланню виключно JSON-каркас інтерфейсу на сервери Gemini API, о робить розроблений засіб не лише ефективним, але й безпечним для впровадження у реальні комерційні CI/CD конвеєри.

### 2.2. Математичне обґрунтування метрик покриття інтерфейсу

Для об’єктивної оцінки ефективності розробленого засобу та якості згенерованої кодової бази було впроваджено систему автоматичного розрахунку аналітичних метрик. Оскільки класичні метрики (наприклад, покриття рядків коду – Line Coverage) нерелевантні для оцінки E2E-сценаріїв, було розділено процес валідації на два етапи. Перший етап передбачає збір статичних метрик під час парсингу абстрактних синтаксичних дерев (AST) згенерованого коду. Другий етап базується на зборі динамічних метрик із виводу терміналу під час фізичного виконання тестів.

Першою серед статичних метрик є Коефіцієнт покриття об'єктів інтерфейсу (DOM Element Coverage) рахується за формулою 2.1. Дана метрика визначає повноту інтеграції інтерактивних елементів інтерфейсу у класи POM. Пріоритет віддається стійким локаторам, що містять атрибути `data-test`. Формально коефіцієнт розраховується як відношення кардинальності множини унікальних селекторів, що використані в згенерованому коді, до загальної множини інтерактивних вузлів, ідентифікованих сканером `SiteAnalyzer` у знімку DOM-дерева. Такий аналітичний підхід гарантує об'єктивне вимірювання частки корисної взаємодії ШІ-засобу з доступним веб-середовищем, що математично виражається за допомогою такого рівняння:

$$DEC = \frac{E_{POM}}{E_{DOM}} \times 100\%, \quad (2.1)$$

де $E_{POM}$ – множина ідентифікаторів `data-test`, інтегрованих ШІ-агентом у класи POM,

$E_{DOM}$ – множина унікальних ідентифікаторів `data-test`, виявлених під час динамічного сканування веб-сторінки модулем `SiteAnalyzer`.

Високе значення (вище 80%) математично підтверджує мінімізацію крихкості тестів (flaky tests), оскільки доводить, що генератор успішно ізолював прив'язку сценаріїв до надійних data-атрибутів замість вразливих CSS-шляхів.

Другим статичним індикатором є Рівень комбінаторного покриття тест-дизайну (Combinatorial Coverage). Метрика характеризує здатність інтелектуального оркестратора генерувати не лінійні, а багатовимірні набори тестових даних, використовуючи техніки еквівалентного розділення та аналізу граничних значень. Розрахунок здійснюється шляхом парсингу AST-дерева та підрахунку функцій, задекорованих параметризатором `@pytest.mark.parametrize` відносно загальної кількості згенерованих методів тестування за формулою 2.2:

$$CC = \frac{T_{param}}{T_{total}} \times 100\%, \quad (2.2)$$

де $T_{param}$ – кількість тестових методів, що використовують параметризацію,

$T_{total}$ – загальна кількість функцій.

Цей показник відображає ступінь покриття бізнес-логіки додатка негативними та граничними сценаріями, демонструючи рівень автономності ШІ під час синтезу вхідних векторів даних без втручання QA-інженера.

### 2.3 Алгоритмічна модель динамічного сканування вебсторінок

Формування первинного набору даних, що адекватно відображає поточний стан клієнтського інтерфейсу, покладено на модуль `SiteAnalyzer`. Відмінною рисою запропонованого підходу є категорична відмова від класичного лексичного розбору вихідного HTML-коду (що традиційно застосовується в бібліотеках штибу BeautifulSoup). Сучасні вебзастосунки будують ієрархію DOM-дерева динамічно в процесі виконання клієнтських скриптів. Саме тому розроблений засіб функціонує в режимі ізольованого фонового браузера (headless Chromium). Це дозволяє системі безпосередньо виконувати JavaScript-інструкції та перехоплювати реальні стани елементів виключно після завершення циклів рендерингу.

Для компенсації асинхронної природи SPA-архітектур (де візуальні компоненти з'являються на екрані із затримкою через мережеві запити до бекенду), реалізовано спеціалізований механізм неблокуючого очікування `_wait_for_spa_ready`. Процедура ініціює опитування DOM-дерева з використанням композитного CSS-селектора, що включає базові теги взаємодії (`input, button, form, a, select`) та пріоритетні тестові маркери (`[data-test]`). Якщо протягом визначеного тайм-ауту (12000 мілісекунд) цільові вузли залишаються недоступними, безпечно перехоплює виняток TimeoutError (з модуля `playwright.sync_api`), не перериваючи загальний процес сканування. Важливим інженерним компромісом у цьому методі стала імплементація подальшої примусової затримки потоку виконання на 2.5 секунди. Такий крок є критично необхідним для стабілізації CSS-мікроанімацій та завершення фонових AJAX-запитів, які здатні змінювати геометрію або перекриття елементів перед їх безпосереднім аналізом.

Екстракція цільових локаторів здійснюється шляхом ін'єкції кастомного JavaScript-коду в контекст виконання сторінки за допомогою інтерфейсу

`page.evaluate()`. Внутрішній скрипт методу `_extract_elements` виконує обхід колекції вузлів, сформованої викликом `document.querySelectorAll`. При цьому алгоритм свідомо оминає суто візуальні або контейнерні теги (наприклад, `div`, `span`, `svg`), фокусуючись лише на інтерактивних елементах. Цей скрипт дозволяє сильно знизити рівень інформаційного шуму. Збираються виключно ті властивості, які мають семантичну цінність для розробки тестових сценаріїв: назва тегу, текстовий вміст (обмежений першими 50 символами), значення плейсхолдерів, абсолютні URI-шляхи (`href`) та ідентифікатори автоматизації (`data-test`, `data-qa`, `data-testid`). Завершальним кроком роботи інжектованого скрипта є процедура очищення отриманих JavaScript-об'єктів від порожніх властивостей (значень `null`) за допомогою оператора `delete`. Таке фільтрування гарантує максимальну компактність результуючого DOM-знімка перед його відправкою до мовної моделі.

Процес формування цілісної топологічної моделі ресурсу (`site_structure`) побудований за принципом направленого обходу графа сторінок. Сесія сканування починається з кореневого URL (`home_page`). Наступним кроком парсер застосовує метод `_find_link`, який використовує заздалегідь скомпільовані регулярні вирази (зокрема, паттерн `r"login|auth/login|sign-in"`) для ідентифікації елементів навігації, що ведуть до модулів автентифікації. Завдяки стандартній функції `urljoin` виявлені відносні посилання безпечно перетворюються на абсолютні. Після успішної ідентифікації цільового URL браузер здійснює перехід, повторює процедуру очікування готовності фреймворку та знімає зліпок сторінки авторизації (`login_page`). Аналогічний рекурсивний підхід застосовується для пошуку та сканування форми реєстрації (`register_page`). Зібраний у результаті проходження цих кроків багатовимірний словник серіалізується у

формат JSON, формуючи оптимізований та ідеально структурований вхідний вектор для бекграунд-генератора коду.

### 2.4. Архітектура інтелектуального оркестратора та підсистема інженерії промптів

Проєктування програмного модуля `AITestGenerator`, який виступає сполучною ланкою між локальним файловим середовищем розробника та хмарним інтерфейсом Google Gemini API (модель `gemini-2.5-flash`), вимагало розв'язання фундаментального інженерного протиріччя. Це протиріччя полягало у необхідності трансляції варіативного, неструктурованого вебсередовища у строгу детерміновану ієрархію об'єктно-орієнтованого програмного коду.

Головним технічним викликом на етапі інтеграції великої мовної моделі (LLM) стало подолання ефекту деградації контексту (Context Degradation). При спробах подачі повних масивів сирих HTML-даних нейромережа стикалася з перевантаженням вікна уваги (Attention Mechanism) трансформерної архітектури. Це призводило до втрати фокусу, генерації неіснуючих селекторів (галюцинацій) та порушення логіки побудови тестових класів. Для подолання цієї перешкоди було розроблено алгоритм редукції контексту (Context Reduction). Замість передачі повного дерева документа, оркестратор інкапсулює у запит виключно очищений JSON-зліпок, попередньо відфільтрований сканером `SiteAnalyzer`. Така математична мінімізація вхідного вектора радикально знижує об'єм токенів, примусово концентруючи аналітичний апарат моделі виключно на бізнес-логіці елементів (атрибути `data-test`, абсолютні шляхи `href`).

Додатковою задачею стало забезпечення абсолютної відповідності генерованого коду актуальним індустріальним стандартам інженерії якості. Для її вирішення імплементовано механізм динамічної агрегації знань. Перед

формуванням основного запиту, метод `_load_downloaded_skills()` ініціює рекурсивний обхід системної директорії `.idea/ai-rules`. Цей алгоритм вилучає суворо формалізовані правила проектування (skills)[18] з існуючих текстових артефактів, що базуються на передових галузевих практиках. Зібраний масив інструкцій об’єднується у єдиний керуючий блок, формуючи базу правил для майбутнього кодогенератора.

Формування завдання для мовної моделі реалізовано за допомогою методики Few-Shot Prompting. У тіло системного запиту інжектовано жорсткі архітектурні обмеження, які унеможливлюють відхилення моделі від стандарту POM:

1. Впроваджено заборону використання асинхронних примітивів (async/await) та класів обгорток типу `BasePage`. Кожен згенерований клас POM має бути синтаксично незалежним і безпосередньо приймати об'єкт `page: Page` у конструкторі `__init__`. Це ліквідує проблему перехресних залежностей при масштабуванні фреймворку.

2. Реалізовано суворе розділення бізнес-логіки та етапів верифікації. Оркестратор блокує можливість інтеграції функцій перевірки (наприклад, методів `expect()`) усередину класів сторінок. Вся логіка підтвердження результатів інкапсулюється виключно в тіло виконуваних тест-кейсів .

3. Щоб усунути проблеми хардкодингу тестових констант, алгоритм інструктує ШІ впроваджувати сесійну фікстуру `config`. Вона автоматично здійснює парсинг локального файлу `config.json` на етапі розгортання сесії, забезпечуючи динамічну передачу критичних параметрів.

Окремою проблемою, яку вдалося ефективно розв'язати засобами промпт-інжинірингу, стала алгоритмізація автоматизованого синтезу тестових даних. Оркестратор примушує генеративну модель виходити за межі лінійних позитивних сценаріїв. За допомогою прямих інструкцій ШІ застосовує

математичні техніки еквівалентного розбиття (Equivalence Partitioning). Для цільових модулів (наприклад, автентифікації `test_auth.py`) генеруються багатовимірні масиви некоректних або граничних значень через декоратор `@pytest.mark.parametrize`.

Зважаючи на жорсткі політики безпеки сучасних цільових серверів (які автоматично відхиляють словникові паролі), в систему вбудовано логіку генерації високорандомізованих рядків із максимальною ентропією. Модель самостійно конструює криптографічно стійкі послідовності (на кшталт Kp9$fX2!wM4_vR7), що гарантує успішне проходження процедури реєстрації штучних користувачів без помилок валідації на стороні бекенду.

Завершальним інженерним рішенням у даному модулі є строга типізація вихідного потоку. Завдяки конфігурації `response_mime_type="application/json"`, хмарний інтерфейс переводиться у режим генерації суворого JSON-об'єкта за наперед визначеною схемою. Це виключає наявність неструктурованого супровідного тексту у відповіді моделі, дозволяючи локальному парсеру безпечно десеріалізувати результат і фізично розгорнути його у вигляді готової файлової ієрархії фреймворку тестування.

### 2.5. Механізми валідації, ізоляції та структурування артефактів тестування

Завершальний етап життєвого циклу функціонування розробленого програмного засобу ШІ-генерації пов'язаний із безпосереднім фізичним розгортанням створеної кодової бази, забезпеченням її структурної цілісності та верифікацією на відповідність синтаксичним стандартам мови Python. Оскільки надійна робота E2E-сценаріїв критично залежить від стабільності файлового оточення, проектування цього етапу вимагало впровадження суворих механізмів

ізоляції, автоматизованого статичного аналізу та інтелектуального перехоплення виняткових ситуацій під час виконання тестів.

Для запобігання взаємного перекриття або випадкового перезапису результатів різних ітерацій генерації коду, у системі реалізовано концепцію ізольованого пісочного середовища (Sandboxing). На практиці цей підхід забезпечується роботою внутрішнього методу `_create_run_directory()` бекграунд-генератора `AITestGenerator`. Під час кожного нового аналізу цільового SPA-додатка алгоритм динамічно зчитує поточну системну дату та точний час, після чого створює унікальний каталог запуску за уніфікованим шаблоном `runs/run_[timestamp]`.

Усередині сформованої директорії автоматично розгортається канонічна ієрархія сучасного індустріального тестового фреймворку:

- pages/ – спеціалізована підпапка, призначена для збереження згенерованих об'єктно-орієнтованих класів Page Object Model, які містять інкапсульовані локатори елементів та методи взаємодії з конкретними сторінками;
- tests/ – ізольований каталог для розміщення безпосередніх виконуваних функцій тестування бізнес-логіки (тест-кейсів), ідентифікованих обов'язковим для фреймворку префіксом test_;
- screenshots/ – цільова директорія для автоматичного логування та збереження графічних артефактів відмов, що фіксуються в процесі виконання сценаріїв;
- pytest.ini – конфігураційний файл ініціалізації середовища, який примусово задає глобальні параметри логування, правила реєстрації кастомних маркерів та мінімально необхідні прапорці запуску командного рядка (наприклад, -v, --tb=short).

Імплементація такої структури гарантує автономність кожного окремого прогону та забезпечує повну готовність згенерованого пакета до негайного перенесення в будь-яке інше робоче середовище або систему безперервної інтеграції.

Специфіка взаємодії з великими мовними моделями (LLM) через зовнішні інтерфейси Gemini API накладає додаткові ризики, пов'язані з потенційним виникненням синтаксичних дефектів у результуючому текстовому потоці (наприклад, випадкові пропуски дужок, незакриті лапки у строкових літералах локаторів чи порушення правил блокових відступів PEP 8). Спроба фізичного виконання такого невалідного скрипта призвела б до аварійної зупинки інтерпретатора PyTest ще на етапі збору тестів (test collection phase).

Щоб нівелювати цей деструктивний чинник, у логіку роботи системи перед безпосереднім збереженням файлів на диск було інтегровано фазу попереднього синтаксичного контролю. Реалізація цього механізму базується на застосуванні апарату AST, доступного через вбудований стандартний модуль `ast` мови Python.

Оркестратор передає сирий текстовий вміст кожної згенерованої сторінки чи тесту у функцію `ast.parse(text_content)`. Цей метод виконує компіляцію коду в деревоподібну структуру об'єктів Python, не запускаючи сам скрипт на фізичне виконання. Якщо у тексті наявні помилки граматики, функція негайно генерує стандартне виключення `SyntaxError`. Локальний блок обробки виняткових ситуацій системи перехоплює цю помилку, блокує фізичний запис пошкодженого модуля на диск і переводить службовий прапорець `has_syntax_errors` у стан `True`. Завдяки цьому засіб убезпечує локальний репозиторій від забруднення неробочим кодом, детально логує характер помилки в консоль для інженера та продовжує обробку інших валідних файлів, зберігаючи загальну працездатність сесії розгортання.

Особливе архітектурне значення в контексті забезпечення стабільності та прозорості виконання тестів має файл конфігурації оточення `conftest.py`, що автоматично синтезується в корені робочої директорії кожного запуску. Головною проблемою традиційних фреймворків E2E-автоматизації під час фіксації помилок є потреба у створенні складних кастомних фікстур (custom fixtures) для явного керування життєвим циклом інстансу браузера та створення скріншотів у блоках `try/except`. Такий підхід перевантажує код і часто призводить до конфліктів очікувань, якщо в середовищі розгортання одночасно активні декілька паралельних потоків виконання.

Для усунення цієї вразливості в архітектуру `conftest.py` було інтегровано низькорівневий перехоплювач подій тестового ядра – офіційний хук `pytest_runtest_makereport`. Оскільки система оперує в межах синхронного стека Playwright, хук реалізовано із застосуванням патерну проектування «Генератор-обгортка» (wrapper pattern) за допомогою мовного примітиву `yield`. Це дозволяє перехоплювачу прозоро інтегруватися безпосередньо всередину внутрішнього циклу виконання кожного тест-кейсу. На етапі завершення тесту хук отримує об'єкт звіту (`report`) та перевіряє його фінальний статус

Якщо тест завершується в стані відмови (`rep.failed`), хук динамічно вилучає чинний інстанс об'єкта `page` із контексту фікстур поточного тестового методу та ініціює асинхронний знімок екрана за допомогою команди `page.screenshot()`. Назва графічного файлу автоматично нормалізується під ASCII-стандарт на основі імені функції (наприклад, `test_invalid_login.png`) і зберігається в каталог `screenshots/`.

Таке інженерне рішення повністю ізолює логіку створення артефактів від бізнес-коду самих сценаріїв чи класів сторінок, усуває потребу в custom-фікстурах браузера, виключає виникнення станів гонитви з глобальним плагіном

`pytest-playwright` та гарантує максимальну стабільність і надійність інфраструктури безперервного тестування у виробничих конвеєрах.

**Висновки до розділу 2**

У другому розділі кваліфікаційної роботи виконано комплексне проектування архітектури, алгоритмічних моделей та математичного апарату автоматищовного засобу інтелектуальної генерації тестових сценаріїв. На основі проведеного аналізу предметної галузі та виявлених проблем існуючих рішень було одержано такі науково-практичні результати:

1. Здійснено системну декомпозицію процесу автоматизованого тестування високодинамічних вебзастосунків класу SPA. Запропоновано й обґрунтовано дворівневу ізольовану архітектуру засобу, що розділяє збір структурних даних та аналіз контексту з наступним створенням коду. Зазначено, що відмова від асинхронного стека на користь строгого синхронного синтаксису Python Playwright дозволяє ліквідувати стани гонитви всередині згенерованого коду.

2. Формалізовано математичний критерій оцінки якості роботи розробленого засобу. Впроваджено дві статичні метрики: коефіцієнт покриття об'єктів інтерфейсу, який визначає повноту інтеграції інтерактивних елементів у класи POM із пріоритетом на стійкі атрибути data-test, та рівень комбінаторного покриття тест-дизайну, що оцінює здатність ШІ-агента застосовувати техніки еквівалентного розбиття та аналізу граничних значень через механізми параметризації PyTest.

3. Розроблено та деталізовано алгоритмічну модель неблокуючого динамічного сканування вебсторінок. Описано логіку ін'єкції кастомного JavaScript-коду через інтерфейс page.evaluate(), який забезпечує фільтрацію інформаційного шуму DOM-дерева та серіалізацію семантичних атрибутів у формат JSON. Спроектовано алгоритм спрямованого обходу графа сторінок за

допомогою регулярних виразів для автоматичного виявлення ізольованих критичних шляхів додатка.

4. Спроектовано підсистему інженерії промптів на базі методики Few-Shot Prompting для взаємодії з моделлю gemini-2.5-flash. Досліджено архітектурну концепцію редукції контексту, яка шляхом подачі очищеного JSON-знімка сторінки мінімізує обсяг токенів, фокусує механізм уваги нейромережі та повністю усуває логічні галюцинації ШІ. Впроваджено механізм динамічної агрегації інженерних стандартів із локальної бази правил.

5. Визначено та програмно закріплено механізми валідації, ізоляції, та структурування вихідних артефактів. Імплементовано фазу попереднього синтаксичного контролю згенерованих скриптів за допомогою аналізу абстрактних синтаксичних дерев, що унеможливлює забруднення репозиторію невалідним кодом.

Проектування, виконане у цьому розділі, слугує безпосереднім технологічним і нормативним базисом для подальшої програмної реалізації та експериментального дослідження розробленого інтелектуального засобу.

# РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ, АНАЛІЗ РЕЗУЛЬТАТІВ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА

## 3.1 Програмна реалізація

Перехід від етапу теоретичного та алгоритмічного проектування до безпосередньої програмної реалізації вимагав вибору інструментарію, здатного забезпечити високу швидкість виконання, стабільність роботи з абстрактними синтаксичними деревами та мережеву взаємодію з хмарними API. Програмний засіб успішно реалізовано мовою програмування Python, що зумовлено її нативною підтримкою необхідних бібліотек, а також наявністю офіційних оптимізованих клієнтів для обраної мовної моделі. Повний вихідний код програмного засобу розміщений у відкритому репозиторії на платформі GitHub(див. Додаток А).

З інженерної точки зору, код проекту спроектовано з жорстким дотриманням принципів об’єктно-орієнтованого програмування, зокрема принципу єдиної відповідальності. Увесь робочий код фізично та логічно розділений на незалежні скрипти, що дозволяє уникнути монолітної структури та значно спрощує подальше масштабування чи рефакторинг засобу. Для забезпечення надійності інтеграції модулів між собою повсюдно застосовано строгу статичну типізацію параметрів та результатів функцій за допомогою модуля typing.

Основним інструментом збору метрик інтерфейсу виступає програмно реалізований клас `SiteAnalyzer`. Він інкапсулює в собі всі необхідні параметри ініціалізації: цільовий маршрут, композитний селектор готовності сторінки та заздалегідь скомпільовані регулярні вирази для маршрутизації. Методи цього класу реалізовані як класичні методи екземпляра, що дозволяє їм динамічно звертатися до внутрішнього стану об'єкта (`self`) під час обробки поточного інстансу сторінки фреймворку Playwright.

Головний оркеструючий вузол системи реалізовано у вигляді класу `AITestGenerator`. Цей об'єкт виступає фасадом програмного засобу. Під час ініціалізації об'єкт динамічно зчитує системні змінні оточення, зокрема, API-ключі, виконує безпечну конфігурацію клієнта `google.generativeai` та створює екземпляр моделі `gemini-2.5-flash`. Додатково клас містить внутрішні методи для роботи з файловою системою: парсинг конфігураційного масиву `config.json` та рекурсивне зчитування інженерних правил із локальної директорії. Фізичне створення артефактів тестування виконується методом `_create_run_directory`, який програмно формує ієрархію папок, перевіряє синтаксис згенерованого результату за допомогою вбудованого модуля `ast` та зберігає виконувані скрипти.

Зв'язуючою ланкою між програмними модулями виступає файл `config.json`, який усуває проблему хардкодингу в кодовій базі. Він десеріалізується в об'єкт словника і передається як вхідний аргумент для ініціалізації екземпляра `SiteAnalyzer`, а згодом – як частина системного промпту для об'єкта `AITestGenerator`.

Для формалізації динамічної поведінки розробленого засобу та візуалізації хронології обміну повідомленнями між усіма підсистемами було побудовано діаграму послідовності. Життєвий цикл виконання програми розпочинається з ініціалізації конфігурації, після чого фокус управління передається аналізатору для запуску headless-сесії браузера та ін'єкції JavaScript-коду. Отриманий масив відфільтрованих вузлів DOM повертається до генератора, де відбувається семантична редукція контексту. Фінальним етапом циклу є відправка сформованого запиту до Google Gemini API, отримання відповіді у форматі JSON та валідація абстрактного синтаксичного дерева з подальшим збереженням файлів. Логічна послідовність описаних вище викликів та мережевих транзакцій наведена на діаграмі послідовності (рис. 3.1).

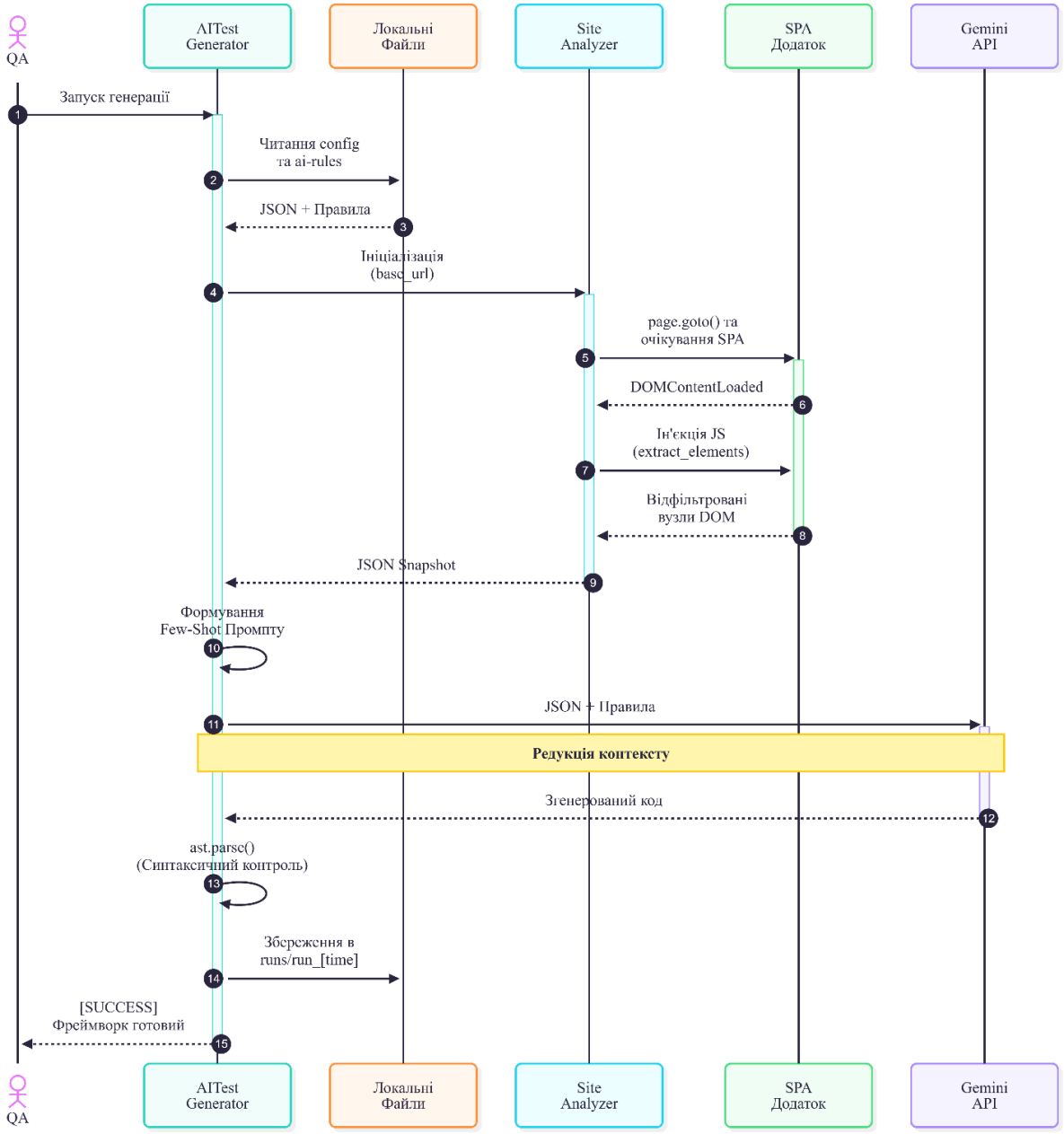


Рисунок 3.1 – UML-діаграма послідовності генерації тестів

Наведена діаграма демонструє строгу послідовність виконання та підтверджує синхронний характер обміну даними між локальними процесами та зовнішнім хмарним інтерфейсом. Завдяки ізоляції функцій збору даних від функцій їх обробки гарантується безперебійна робота засобу навіть у випадках тимчасових мережевих затримок на боці цільового вебзастосунку.

### 3.2. Експериментальне дослідження на базі цільового вебресурсу

Для перевірки працездатності розробленого програмного засобу та оцінки якості згенерованої кодової бази було проведено експериментальне дослідження. Як цільовий об'єкт тестування обрано відкритий демонстраційний вебресурс електронної комерції `https://practicesoftwaretesting.com`, базовий URL якого, разом із тестовими обліковими даними, було попередньо зафіксовано у глобальному конфігураційному файлі `config.json`. Вибір даного ресурсу зумовлений його архітектурою SPA, що робить його релевантним середовищем для перевірки здатності системи обробляти динамічний асинхронний рендеринг.

Експеримент було ініційовано запуском головного модуля генерації. На першому етапі підсистема динамічного аналізу (`SiteAnalyzer`) успішно ініціалізувала безголову сесію браузера, дочекалася завантаження DOM-дерева та виконала ін'єкцію JavaScript-коду для екстракції цільових елементів. Результатом цього етапу стало формування очищеного JSON-знімка цільових сторінок. Аналіз отриманого масиву даних підтвердив, що сканер успішно відфільтрував візуальний шум та вилучив виключно семантично значущі вузли, надавши пріоритет стійким тестовим маркерам. Застосування такого підходу дозволило повністю ізолювати процес алгоритмічної генерації тестів від нестабільності візуального шару вебдодатка, сконцентрувавши обчислювальні ресурси штучного інтелекту виключно на аналізі бізнес-логіки.

На другому етапі експерименту отриманий JSON-об'єкт було передано до оркестратора `AITestGenerator` для семантичної редукції контексту та формування запиту до API `gemini-2.5-flash`. На основі інтегрованих локальних правил мовна модель синтезувала структуру класів Page Object Model (POM) та виконувані тест-кейси. Процес автоматичного формування структури каталогів, валідації синтаксису (через модуль `ast`) та збереження артефактів тестування зафіксовано в системному лозі терміналу, наведено на рисунку 3.2.

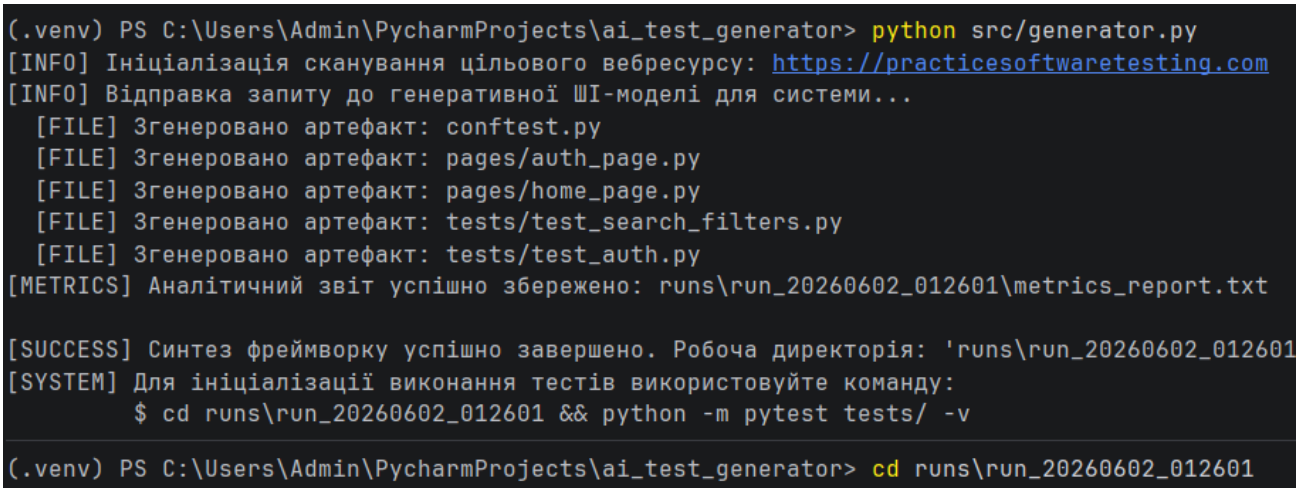


Рисунок 3.2 – Візуалізація процесу генерації тестового фреймворку та збереження артефактів

Відразу після успішної генерації POM-класів та тестових сценаріїв система здійснила автоматичний розрахунок статичних метрик якості, математичне обґрунтування яких було наведено у другому розділі. Коефіцієнт покриття інтерфейсу склав 100%, що означає безвтратну трансляцію всіх 75 виявлених інтерактивних елементів у відповідні класи фреймворку. Водночас рівень комбінаторного покриття сягнув 88.9%, оскільки мовна модель успішно імплементувала техніки тест-дизайну для 8 з 9 згенерованих сценаріїв, відмовившись від параметризації лише у тих випадках, де перевірка мала строгий лінійний характер.

Аналіз згенерованих файлів `auth_page.py` та `home_page.py` продемонстрував високу точність виконання системних інструкцій. Генеративна модель суворо дотрималася архітектурних обмежень: код реалізовано у строгому синхронному синтаксисі, класи є синтаксично незалежними та приймають об'єкт `page: Page` безпосередньо в конструкторі `__init__`. Штучний інтелект коректно ініціалізував локатори, прив'язавши їх до атрибутів `data-test` (наприклад, `self.email_input = page.locator("[data-`

`test='email']")`), та інкапсулював бізнес-дії без змішування з логікою перевірок.

Паралельно оркестратор згенерував виконувані тест-кейси `test_auth.py` та `test_search_filters.py`. Модель застосувала техніки еквівалентного розбиття для забезпечення комбінаторного покриття, імплементувавши декоратор `@pytest.mark.parametrize`. Фінальна стадія експерименту передбачала фізичне виконання згенерованого фреймворку. Візуалізацію логування результатів прогону наведено на рисунку 3.3.

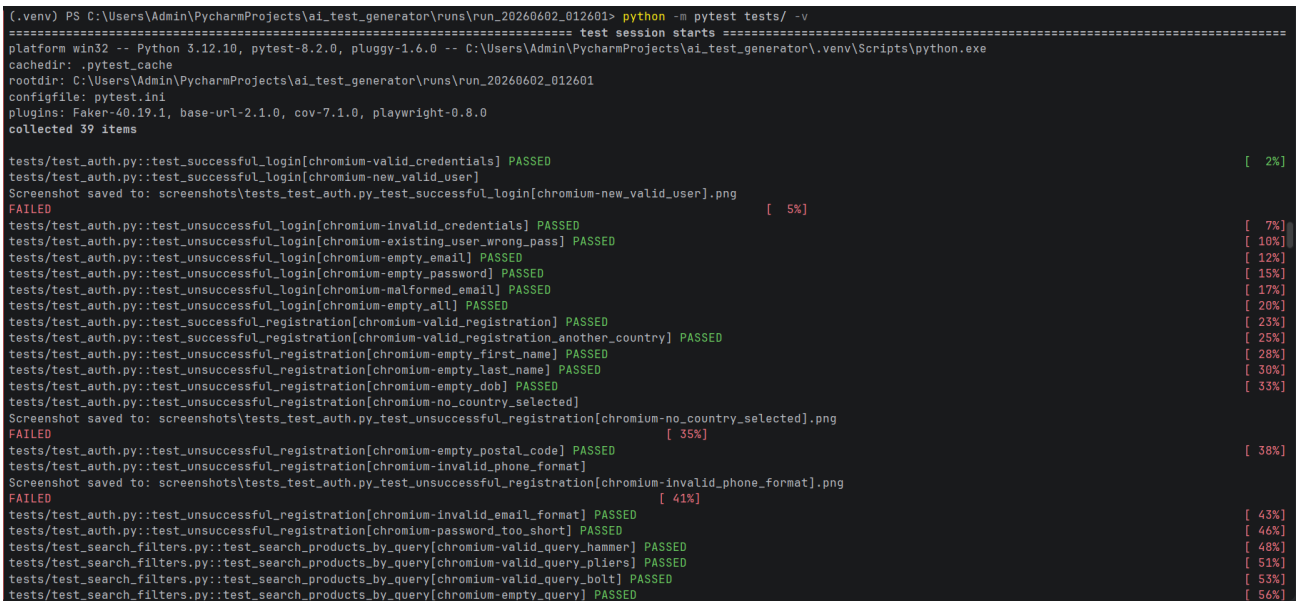


Рисунок 3.3 – Скріншот результатів фізичного виконання згенерованих сценаріїв у консолі PyTest

Аналіз отриманих результатів експерименту виявив важливу науково-практичну закономірність. Як видно з результатів запуску (рис. 3.3), система успішно виконала 39 згенерованих тестових сценаріїв. Більшість із них (35 тестів) завершилася зі статусом PASSED, що підтверджує абсолютну синтаксичну та базову логічну коректність синтезованого коду. Проте 4 тести завершилися зі статусом FAILED. Наявність таких негативних результатів є

закономірною і слугує цінним матеріалом для виявлення обмежень взаємодії LLM із непередбачуваним клієнтським інтерфейсом.

Детальний аналіз відмов (автоматично зафіксованих системою у вигляді скріншотів за допомогою хука `pytest_runtest_makereport`) виявив чотири патерни логічних розбіжностей між очікуваннями ШІ та реальною поведінкою додатка `practicesoftwaretesting.com`.

Нестандартна локалізація елементів помилки (рис. 3.4). В одному з тестів авторизації (test_successful_login[chromium-new_valid_user]) очікувалося, що після введення даних з'явиться стандартний банер про успішний вхід. Натомість цільовий додаток вивів повідомлення про помилку у нестандартному місці інтерфейсу (через проблеми з валідацією нового користувача), яку згенерований селектор [data-test='login-error'] не зміг однозначно ідентифікувати.

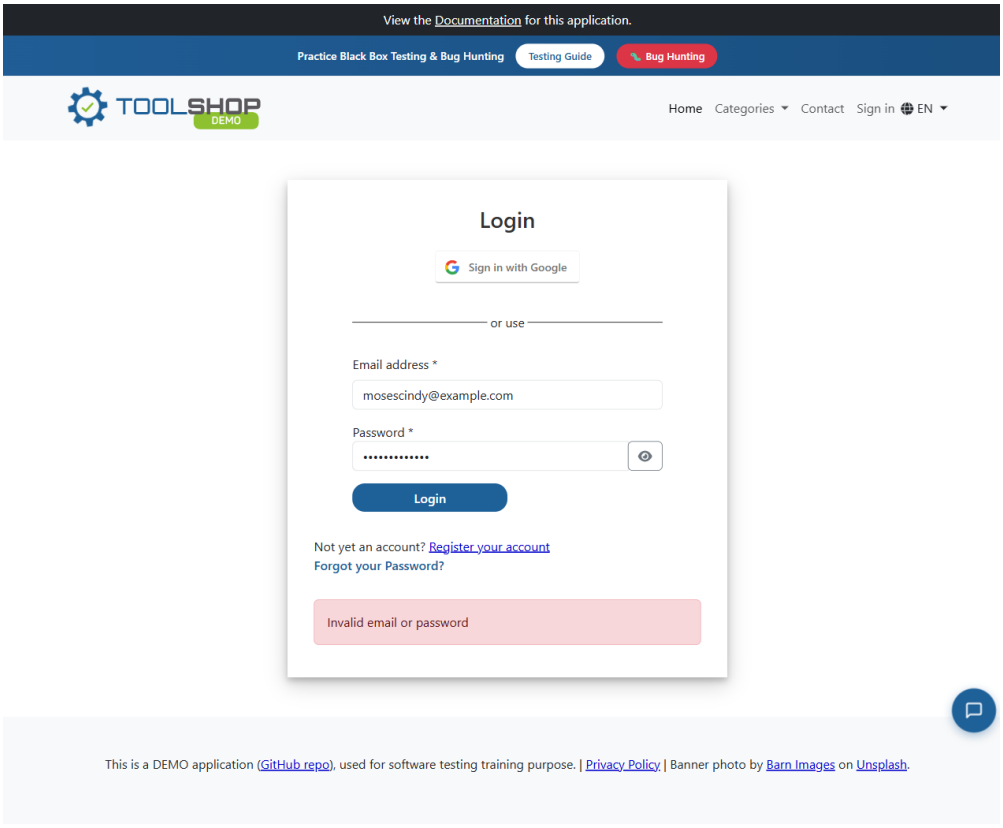


Рисунок 3.4 – Перехоплена відмова тесту: нестандартне місце виведення помилки авторизації

Непередбачена логіка фронтенд-валідації (рис. 3.5). Під час перевірки реєстрації (test_unsuccessful_registration[chromium-invalid_phone_format]) ШІ-оркестратор згенерував перевірку на появу попередження про "невалідний формат". Однак додаток не вивів червоного попередження, а натомість просто заблокував ввід літер і вивів системну підказку "дозволені лише цифри". Модель, спираючись на загальні патерни e-commerce, очікувала жорсткої появи елемента помилки.

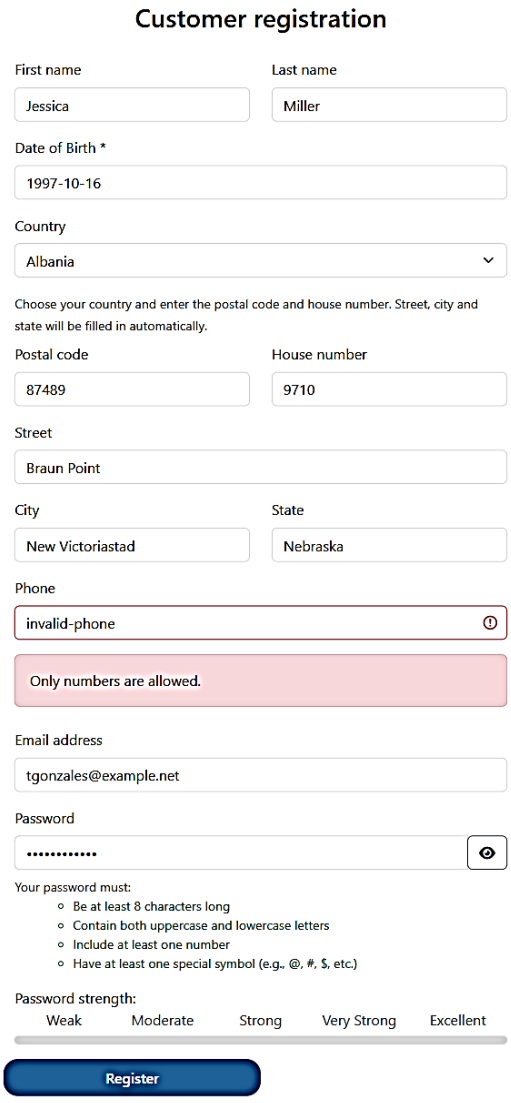


Рисунок 3.5 – Перехоплена відмова тесту: конфлікт очікувань з логікою валідації поля вводу

Блокування потоку виконання обов'язковими списками (рис. 3.6). У сценарії реєстрації (test_unsuccessful_registration[chromium-no_country_selected]) процес виконання завис на обов'язковому випадаючому списку (select) вибору країни. ШІ-генератор не передбачив необхідності явної взаємодії з цим елементом для активації наступних кроків форми реєстрації, що призвело до спрацювання TimeoutError.

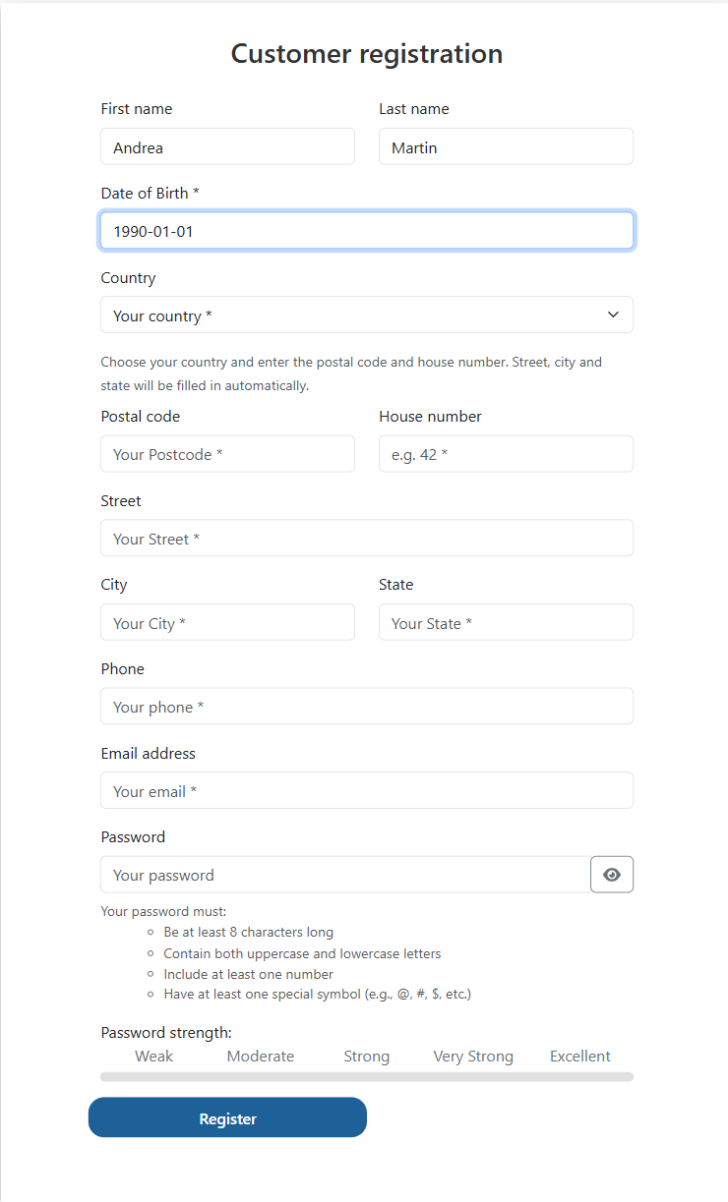


Рисунок 3.6 – Перехоплена відмова тесту: блокування сценарію обов'язковим елементом "select"

Створення неповних ланцюжків взаємодії з фільтрами (рис. 3.7). Під час тестування модуля пошуку та фільтрації (test_filter_products_by_category) генеративна модель згенерувала пошуковий запит для перевірки реального товару («Grinder»). Проте на тестовому сайті цей інструмент має базову вартість, що перевищує 100 доларів, і за замовчуванням приховується фронтенд-налаштуваннями цінового діапазону (price range slider). ШІ-оркестратор не врахував необхідність попередньої взаємодії з повзунком ціни для цієї конкретної категорії інструментів, що логічно призвело до порожньої видачі товарів на екрані та падіння функцій expect(). Цей випадок яскраво демонструє нездатність сучасних LLM передбачати складні, багатокомпонентні стани інтерфейсу без жорстких попередніх інструкцій.

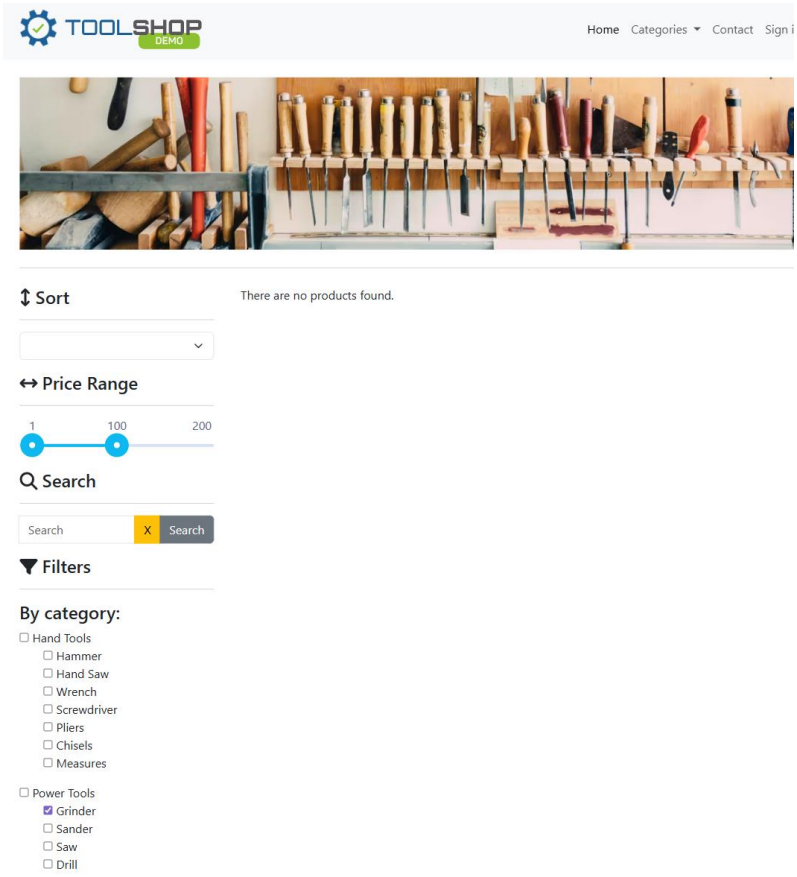


Рисунок 3.7 – Перехоплена відмова тесту: конфлікт базових налаштувань цінового фільтра та згенерованих тестових даних

Відмови тестових сценаріїв емпірично пояснюються кількома фундаментальними факторами. По-перше, системний промпт та база правил (ai-rules) розроблялися як універсальні інструкції. Оскільки кожен конкретний вебресурс має унікальні ідіосинкразії в побудові DOM-дерева, універсальний промпт не завжди здатен передбачити ці мікростани без додаткового налаштування (fine-tuning). По-друге, LLM мають ймовірнісну природу: під час генерації вхідних векторів даних для `@pytest.mark.parametrize` модель іноді синтезує лексично правильні, але контекстуально нерелевантні значення (як у випадку з пошуком категорій).

Отримані результати експерименту підтверджують світову тенденцію: на сучасному етапі розвитку штучний інтелект виступає високоефективним асистентом (Copilot). Високий відсоток успішних прогонів (35 з 39) свідчить, що інструмент здатен взяти на себе великий відсоток рутинної роботи зі створення базового фреймворку. Проте наявність 4-х обґрунтованих логічних відмов доводить, що ШІ не є повноцінною заміною QA-інженера. Згенерована кодова база є потужним фундаментом, який вимагає фінального інженерного рефакторингу (наприклад, уточнення локаторів помилок або додавання кроку вибору зі списку), що повністю відповідає концепції "Human-in-the-loop" у сучасній індустрії.

### 3.3. Оцінка одержаних результатів та аналіз недоліків

Проведений експеримент дозволив кількісно та якісно оцінити ефективність розробленого інтелектуального оркестратора. Головний висновок дослідження полягає у тому, що поставлену мету роботи досягнуто в повному обсязі: створено функціонуючий автоматизовний програмний засіб, який автоматизує синтез E2E-тестів на базі патерну POM.

Серед беззаперечно позитивних результатів варто відзначити 100% покриття виявлених інтерактивних елементів, високий рівень комбінаторного

покриття (88.9%) та здатність генератора самостійно синтезувати рандомізовані тестові дані для перевірок безпеки. Використання синхронної архітектури Playwright дозволило повністю уникнути проблем зі станами гонитви (race conditions) під час виконання коду.

Водночас, як було детально проаналізовано у підрозділі 3.2, емпірична перевірка виявила специфічні недоліки. Випадкові падіння сценаріїв (4 з 39) зумовлені стохастичною природою великих мовних моделей, які за певних умов генерують неоптимальні селектори очікувань, якщо фронтенд-елементи мають складну динамічну мутацію або приховані стилі (як у випадку з повзунком цінового діапазону).

Світові тенденції розробки засобів тестування на основі штучного інтелекту підтверджують, що дана проблема є фундаментальною для всієї галузі генеративного програмування. Повна довіра до ШІ-агентів (Zero-Touch Automation) наразі є неможливою. Найбільш раціональним рішенням, що домінує в сучасній науковій та інженерній спільноті, є використання таких систем виключно як високопродуктивних асистентів, що вимагає фінального рефакторингу коду інженером. Перспективним напрямком подальшого вдосконалення розробленого засобу є доповнення алгоритму генерації евристичними механізмами самовідновлення (self-healing) локаторів безпосередньо під час прогону тестів.

**Висновок до розділу 3**

У третьому розділі кваліфікаційної роботи здійснено практичну програмну реалізацію спроектованого інтелектуального засобу генерації автоматизованих тестів та проведено його комплексне експериментальне дослідження. Аналіз одержаних результатів дозволяє зробити наступні висновки:

1. Успішно розроблено програмний засіб мовою Python із використанням фреймворків Playwright та PyTest, а також хмарного інтерфейсу

Google Gemini API. Імплементована об'єктно-орієнтована архітектура з розподілом відповідальності між модулем збору даних (SiteAnalyzer) та інтелектуальним оркестратором (AITestGenerator) довела свою надійність і стійкість до мережевих затримок.

2. Тестування засобу на базі високодинамічного SPA-додатка електронної комерції підтвердило досягнення мети дослідження. Засіб продемонстрував 100% коефіцієнт покриття об'єктів інтерфейсу (DOM Element Coverage), безвтратно перетворивши 75 виявлених інтерактивних елементів у класи POM. Завдяки автоматичному застосуванню технік еквівалентного розбиття, рівень комбінаторного покриття тест-дизайну склав 88.9%.

3. Під час фізичного виконання 39 згенерованих сценаріїв успішно завершилися 35 тестів. Детальний аналіз 4-х відмов дозволив виявити ключові обмеження сучасних великих мовних моделей. Встановлено, що ймовірнісна природа ШІ стикається з труднощами при обробці нестандартної локалізації помилок, прихованих фронтенд-валідацій та складних багатокомпонентних фільтрів. Ці результати емпірично доводять, що концепція повної автоматизації наразі є недосяжною.

Результати дослідження мають значний потенціал для впровадження у виробничі цикли розробки програмного забезпечення. Передбачувані області використання створеного засобу включають:

- автоматизацію регресійного тестування у CI/CD конвеєрах для швидкого отримання зворотного зв'язку при релізах нових версій вебдодатків;
- прискорення процесу підготовки початкових тестових наборів (Test Suite Bootstrapping) для стартап-проектів та команд середнього бізнесу;
- використання засобу як навчального інструменту (тренажера) для вивчення принципів інженерії промптів та архітектурних патернів забезпечення якості.

# ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-прикладне завдання щодо автоматизації процесу забезпечення якості програмного забезпечення шляхом інтеграції генеративного штучного інтелекту в конвеєр тестування вебдодатків. За результатами проведених досліджень, архітектурного проектування та програмної розробки сформульовано наступні підсумки:

1. Аналіз сучасних підходів до автоматизації засвідчив, що головною перепоною для ефективного використання LLM є надмірний візуальний шум DOM-дерев та обмеження контекстного вікна нейромереж. Пряма передача сирої HTML-розмітки призводить до високого рівня «галюцинацій» та крихкості згенерованих локаторів.

2. Найбільш цінним концептуальним здобутком роботи автор вважає розробку та обґрунтування методу семантичної редукції контексту. Перетворення складного графа сторінки на ізольований JSON-зліпок із жорсткою пріоритетизацією атрибутів дозволило відсікти візуальний шар додатка, сконцентрувавши обчислювальні ресурси штучного інтелекту виключно на бізнес-логіці.

3. Теоретичні моделі успішно реалізовано шляхом створення автоматизованого програмного засобу. Інтеграція алгоритмів динамічного неблокуючого сканування з хмарним API генеративної моделі забезпечила можливість синтезу синтаксично коректного коду, що суворо відповідає індустріальному патерну Page Object Model.

4. Експериментальна перевірка на базі динамічного SPA-додатка підтвердила високу результативність запропонованої архітектури. Система продемонструвала абсолютне покриття ідентифікованих цільових елементів та здатність до самостійного конструювання параметризованих матриць тестових

даних, що доводить життєздатність концепції делегування рутинного тест-дизайну ШІ-агентам.

5. Аналіз результатів фізичного виконання згенерованого фреймворку дозволив виявити фундаментальні обмеження сучасних LLM. Емпірично встановлено, що генеративні моделі все ще мають складнощі з інтерпретацією нетипових фронтенд-валідацій та прихованих мікростанів інтерфейсу. Це концептуально спростовує можливість повної автоматизації (Zero-Touch) на сучасному етапі розвитку технологій і підтверджує раціональність використання створеного засобу виключно як високоефективного AI-Copilot у парадигмі «Human-in-the-loop», де інженер виконує роль фінального валідатора.

Оцінюючи перспективи розвитку розробленого засобу, можна виокремити декілька ключових векторів його подальшого вдосконалення для впровадження у складні enterprise-середовища. По-перше, критично важливим кроком є розробка евристичних механізмів самовідновлення локаторів (self-healing) на етапі виконання: у разі падіння тесту через зміну верстки, система повинна вміти автоматично надсилати стек помилки до LLM для корекції селектора без переривання всього прогону.

По-друге, значний приріст якості логічних перевірок (assertions) може дати перехід від універсальної моделі (gemini-2.5-flash) до використання вузькоспеціалізованих нейромереж, донавчених (fine-tuned) виключно на еталонних репозиторіях автотестів. Нарешті, архітектуру доцільно розширити за рахунок інтеграції технології RAG (Retrieval-Augmented Generation), що дозволить системі аналізувати не лише поточний стан вебінтерфейсу, але й супровідну технічну документацію продукту (наприклад, Swagger API), генеруючи більш комплексні наскрізні сценарії взаємодії.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Смагіна О.О. Якість програмного забезпечення та тестування : навч. посіб. Луганськ: ДЗ «ЛНУ імені Тараса Шевченка», 2021. – 286 с. URL: https://dspace.luguniv.edu.ua/xmlui/bitstream/handle/123456789/7510/2021.pdf?sequence=4&isAllowed=y (дата звернення: 25.04.2026).

2. Integrated Graph-Based Testing Pipeline for Modern Single-Page Applications // ResearchGate, 2024. URL: https://www.researchgate.net/publication/393621893_INTEGRATED_GRAPH-BASED_TESTING_PIPELINE_FOR_MODERN_SINGLE-PAGE_APPLICATION (дата звернення: 25.04.2026).

3. Vallu H. V. M. K., Venkata S. S. K. TeleTester: An Industrial Evaluation of LLM-Based RAG Cypress End-to-End Test Generation. Karlskrona: Blekinge Institute of Technology, 2026.

4. Мороз Б.І., Сироткина О.І., Кострицька С.І., Ларикова М.В. Selenium vs. playwright: exploring end-to-end testing in multi-threaded environment, 2023

5. Applitools | AI-Automated Compliance Testing URL: https://applitools.com/ (дата звернення: 26.04.2026).

6. Mabl URL: https://help.mabl.com/hc/en-us (дата звернення: 26.04.2026).

7. Functionize URL: https://www.functionize.com/resources (дата звернення: 26.05.2026).

8. Testim URL: https://www.testim.io/ (дата звернення: 26.04.2026).

9. testRigor URL: https://testrigor.com/docs/ (дата звернення: 26.04.2026).

10. Guru99. (б.д.). Best AI Testing Tools. URL: https://www.guru99.com/uk/best-ai-testing-tools.html (дата звернення: 26.04.2026).

11. Relicx URL: https://www.relicx.ai/ (дата звернення: 26.04.2026).

12. ContextQA URL: https://contextqa.com/book-a-demo/ (дата звернення: 26.04.2026).

13. Momentic URL: https://momentic.ai/ (дата звернення: 26.04.2026).

14. Roost.ai URL: https://www.roost.ai/ (дата звернення: 26.04.2026).

15. Promptware Engineering: Software Engineering for LLM Prompt Development // arXiv preprint arXiv:2503.02400v2. – 2025.

16. Google Gemini 2.5 Flash Model Overview. OpenRouter. URL: https://openrouter.ai/google/gemini-2.5-flash (дата звернення: 27.04.2026).

17. The Impact of Prompt Programming on Function-Level Code Generation – arXiv URL: https://arxiv.org/html/2412.20545v1 (дата звернення: 27.04.2026).

18. Top 10 QA Skills Every Developer Should Install in 2026. *QASkills.sh*. 2026. URL: https://qaskills.sh/blog/top-10-qa-skills-developers-2026 (дата звернення: 25.05.2026).

# ДОДАТОК А. Вихідний код

Посилання на код, у відкритому репозиторії на платформі GitHub, знаходиться за адресою: https://github.com/Ivan1336/ai_test_generator