Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)
Навчально-науковий інститут
комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: *«AI - орієнтований фреймворк для автоматизованого функціонального тестування вебзастосунків»*

Виконав: студент 6 курсу групи КН-63м
спеціальності
122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Данилюк Р. М.
(прізвище та ініціали)

Керівник Борецька І. Б.
(прізвище та ініціали)

Рецензент Яцишин С. І.
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет України

(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій

Кафедра комп'ютерних наук

Рівень вищої освіти другий (магістерський)

Спеціальність 122 "Комп'ютерні науки"

(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри КН

__________ Борецька І. Б.

"10" _грудня_ 2025 року

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Данилюку Роману Миколайовичу

(прізвище, ім'я, по батькові)

1. Тема роботи *AI - орієнтований фреймворк для автоматизованого функціонального тестування вебзастосунків*

керівник роботи *Борецька Ірина Богданівна, к. т. н., доцент*

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "29" квітня 2025 року № С-288

2. Термін подання студентом роботи 10 грудня 2025 р.

3. Вихідні дані до роботи Проаналізувати предметну область та інструменти, застосувати ШІ для автоматизованого вебтестування, спроєктувати систему на Python і представити результати її роботи.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Стан проблемної області

Інформаційне забезпечення

Математичне забезпечення

Програмне забезпечення

Розроблення стартап-проєкту

Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Підготовка матеріалів до доповіді

6. Дата видачі завдання 1 травня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літературних даних та інших джерел згідно досліджуваної теми	02.05-30.05.2025	Виконано
2	Аналіз досліджуваної теми та вибір відповідних варіантів її розробки	01.06-30.06. 2025	Виконано
3	Постановка задачі та її формалізація	01.07-30.07. 2025	Виконано
4	Вибір та обґрунтування методів і засобів проведення дослідження	01.08-30.08. 2025	Виконано
5	Розроблення концептуальної схеми реалізації завдання	01.09-15.09. 2025	Виконано
6	Програмна реалізація завдання	16.09-30.10. 2025	Виконано
7	Тестування програмного продукту та отриманих результатів	01.11-15.11. 2025	Виконано
8	Розробка пояснювальної записки магістерської роботи	16.11-30.11. 2025	Виконано
9	Корегування пояснювальної записки згідно вимог, розроблення презентації	01.12-09.12. 2025	Виконано

Студент ______ (підпис) Данилюк Р.М. (прізвище та ініціали)

Керівник роботи ______ (підпис) Борецька І. Б. (прізвище та ініціали)

АНОТАЦІЯ

Магістерська робота містить 100 сторінок пояснювальної записки, 19 рисунків, 2 додатки, 12 джерел.

Робота присвячена розробці серверної частини AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків. У роботі реалізовано багаторівневу архітектуру з використанням мови програмування Python та фреймворку FastAPI. Інтегровано браузерну взаємодію через Selenium, а також реалізовано штучний інтелект для динамічного аналізу користувацького інтерфейсу та адаптивної генерації тестів. Дані зберігаються в базі MongoDB. Реалізовано вебінтерфейс для запуску та перегляду результатів тестів.

Ключові слова: тестування, штучний інтелект, Selenium, FastAPI, автоматизація, вебзастосунки, Python, MongoDB.

ABSTRACT

The thesis contains project 100 pages of explanatory notes, 19 figures, 2 appendices, 12 sources.

This comprehensive course project is devoted to the development of the server-side component of the AI-oriented AutoTestAI framework for automated functional testing of web applications. The project implements a multi-layered architecture using the Python programming language and the FastAPI framework. Browser interaction is integrated through Selenium, and artificial intelligence is implemented for dynamic analysis of the user interface and adaptive test generation. Data is stored in a MongoDB database. A web interface for launching tests and viewing test results is also implemented.

Keywords: testing, artificial intelligence, Selenium, FastAPI, automation, web applications, Python, MongoDB.

ТЕХНІЧНЕ ЗАВДАННЯ

Розробити вебзастосунок, основна функція якого полягає в забезпеченні автоматизованого функціонального тестування вебінтерфейсів із використанням елементів штучного інтелекту. Система повинна автоматично генерувати, адаптувати та виконувати тестові сценарії на основі аналізу DOM-структури та історії взаємодій користувачів із вебсторінками.

Для реалізації цього завдання необхідно створити серверну та клієнтську частини фреймворку. Серверна частина має бути реалізована на мові Python з використанням FastAPI та містити REST API для запуску тестів, обробки результатів, навчання моделей та взаємодії з базою даних (MongoDB або PostgreSQL). Архітектура має бути модульною, з розділенням логіки на окремі компоненти: контролери, сервіси, AI-модулі, валідацію, обробку помилок тощо.

Браузерна взаємодія повинна здійснюватися через Selenium або Playwright, з можливістю симулювання кліків, заповнення форм та аналізу відповіді. Для AI-аналізу UI-елементів передбачити використання бібліотек машинного навчання (наприклад, scikit-learn або TensorFlow), що дозволить класифікувати та пріоритезувати тестові сценарії.

Клієнтська частина має забезпечити інтерфейс для запуску тестів, перегляду логів, перегляду heatmap UI, керування проєктами та конфігураціями. Реалізувати її за допомогою HTML/CSS/JavaScript з підтримкою адаптивної верстки.

Система повинна мати підтримку нотифікацій – надсилання результатів тестування електронною поштою користувачу. Уся система має бути реалізована у вигляді монолітного застосунку з модульною архітектурою, що передбачає подальше масштабування, розширення функціоналу.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ ………………………… 7
ВСТУП ………………………………………………………………………… 8
РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ ……………………………… 10
1.1. Опис проблемної області …………………………………………… 10
1.2. Аналоги існуючих систем ………………………………………….... 12
1.3. Мови програмування……………………...………………………… 22
1.4. Фреймворки………………………………………………………….... 24
1.5. Робота з стилями…………………………………………………….. 27
1.6. Інструменти роботи з БД…………………………………………….. 29
Висновки до розділу ……………………………………………………... 32
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ……………………….. 34
2.1. Система контролю версій ……………………………………………... 34
2.2. Віддалений репозиторій …………………………………………….. 35
2.3. Високопродуктивне логування ……………………………………... 36
2.4. Струкрура БД………………………………………………………… 37
Висновки до розділу ……………………………………………………... 41
РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ ……………………….. 43
3.1. Формалізація задачі розумного автоматизованого тестування……………. 43
3.2. Побудова та аналіз графа інтерфейсу ……………………...……… 44
3.3. Векторизація елементів та селекторна подібність ……………………… 44
3.4. Підкріплене навчання у тестуванні ………………………………… 45
3.5. Евристики побудови сценаріїв………………………………………… 45
3.6. Метрики покриття та ефективності…………………………………. 45
3.7. Виявлення аномалій у поведінці DOM………………………………... 47
3.8. Адаптивне переобрання сценаріїв……………………………………... 47
Висновки до розділу ……………………………………………………... 48
РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ………………………………… 51

4.1.Розроблення AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків………………... 51
4.2. Розроблення графічного інтерфейсу управління фреймворком AutoTestAI 63
Висновки до розділу………………………………………………………… 70
РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ……………………………… 72
5.1. Структура проєкту AI-орієнтованого фреймворку для автоматизованого функціонального тестування вебзастосунків ……………………………... 72
5.2. Цільова аудиторія ……………………………………………………… 76
5.3. Архітектура бізнес-моделі стартапу …………………………………... 77
Висновки до розділу ……………………………………………………... 81
ВИСНОВКИ ……………………………………………………………… 83
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ………………………………... 85
ДОДАТКИ ……………………………………………………………... 85
ДОДАТОК А ……………………………………………………………… 86
ДОДАТОК Б ……………………………………………………………… 91

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

AI – Artificial Intelligence / Штучний інтелект

ML – Machine Learning / Машинне навчання

NLP – Natural Language Processing / Обробка природної мови

UI – User Interface / Користувацький інтерфейс

DOM – Document Object Model / Об'єктна модель документа

API – Application Programming Interface / Інтерфейс прикладного програмування

REST API – Інтерфейс для обміну інформацією між клієнтом і сервером через HTTP

HTTP – Протокол передачі гіпертексту

JSON – JavaScript Object Notation / Формат обміну структурованими даними

CI/CD – Continuous Integration / Continuous Deployment / Безперервна інтеграція і розгортання

IDE – Integrated Development Environment / Середовище розробки

HTML – HyperText Markup Language / Мова розмітки гіпертексту

CSS – Cascading Style Sheets / Таблиці каскадних стилів

Selenium – Інструмент для автоматизації браузерного тестування

FastAPI – Веб-фреймворк для створення API на Python

MongoDB – Документо-орієнтована NoSQL база даних

pytest – Фреймворк для написання та запуску юніт-тестів у Python

BERT – Bidirectional Encoder Representations from Transformers / Модель для NLP

OCR – Optical Character Recognition / Оптичне розпізнавання символів

XPath – Мова для навігації DOM-структурою вебсторінки

ВСТУП

Сучасні вебзастосунки характеризуються високою динамічністю, масштабністю та складною взаємодією компонентів, що створює зростаючу потребу в ефективному, гнучкому та надійному тестуванні. Забезпечення якості програмного забезпечення стає критично важливим, особливо в умовах швидких релізів і гнучких методологій розробки. Традиційні методи тестування не завжди справляються з цими викликами, тому виникає необхідність у нових підходах до автоматизованого тестування.

Магістерська робота присвячена розробці AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків. Система спрямована на автоматизацію процесів генерації, виконання та адаптації тестових сценаріїв із використанням алгоритмів штучного інтелекту. Вона дозволяє не лише пришвидшити процес тестування, а й підвищити його гнучкість та точність за рахунок адаптації до змін у структурі DOM, зміни селекторів, а також інших типових проблем підтримки автотестів.

Одним із основних завдань, які розв'язуються в межах роботи, є створення середовища, яке дозволяє користувачеві формувати сценарії тестування у зручному інтерфейсі та автоматично отримувати рекомендації щодо виправлення помилок у тестах за допомогою AI.

Об’єктом дослідження є процес автоматизованого функціонального тестування вебзастосунків.

Предметом дослідження – алгоритми генерації та адаптації тестів із використанням штучного інтелекту.

Метою роботи є створення інтелектуального фреймворку, що забезпечує автоматизоване, адаптивне тестування вебзастосунків із мінімальним залученням розробника.

Практичне значення роботи полягає у можливості впровадження розробленого фреймворку AutoTestAI у реальні процеси тестування сучасних вебпродуктів.

Завдяки використанню алгоритмів штучного інтелекту, система дозволяє не лише автоматизувати створення та запуск тестових сценаріїв, але й адаптувати їх у відповідь на зміни в DOM-структурі або логіці вебінтерфейсу без необхідності ручного втручання. Це, у свою чергу, суттєво скорочує час, необхідний на написання та підтримку автотестів, зменшує ризик виникнення людських помилок, а також забезпечує вищу стабільність і надійність програмних релізів. Крім того, фреймворк може бути корисним як для команд тестування, так і для розробників, які працюють за гнучкими методологіями (Agile, DevOps), де швидкий зворотний зв'язок і безперервна інтеграція відіграють ключову роль.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Опис проблемної області

У сучасному цифровому середовищі вебзастосунки стали ключовим елементом взаємодії бізнесу з користувачами. Компанії в різних сферах – від електронної комерції до державних сервісів – покладаються на функціональність, доступність та стабільність своїх вебрішень. У цих умовах забезпечення високої якості програмного забезпечення є критично важливим, і особливої уваги потребує процес тестування інтерфейсів користувача та функціональності вебдодатків.

Традиційне ручне тестування, попри свою гнучкість і глибину аналізу, має низку суттєвих недоліків: високу трудомісткість, низьку масштабованість та залежність від людського фактора. Кожне оновлення функціоналу вимагає повторного проходження тестових сценаріїв, що вимагає багато часу й ресурсів. У великих або динамічних проєктах це перетворюється на серйозну проблему для команд розробки та забезпечення якості.

У свою чергу, класичне автоматизоване тестування, хоч і значно прискорює процес перевірки функціоналу, часто стикається з проблемами підтримки тестів при найменших змінах у структурі DOM. Будь-яка зміна у назві класу, ID або структурі сторінки призводить до помилок у тестах і потребує ручного редагування скриптів. Це створює додаткове навантаження на команди QA, знижує ефективність автоматизації та уповільнює цикл розробки та релізу.

Особливо актуальною є ця проблема в умовах стрімкого розвитку frontend-технологій, коли зміни у вебінтерфейсі можуть бути частими, непередбачуваними та глибокими. Компонентні фреймворки, такі як React, Angular чи Vue, створюють складні динамічні DOM-структури, що додатково ускладнює автоматизацію тестів. Застарілі інструменти автоматизації, які покладаються на статичні селектори або лінійні скрипти без адаптивної логіки, не справляються з такими завданнями ефективно.

У відповідь на ці виклики, усе більше уваги привертають інтелектуальні системи тестування, що використовують штучний інтелект (AI) та машинне навчання (ML) для побудови адаптивних, самонавчальних тестів. Такі системи здатні самостійно аналізувати структуру сторінки, розпізнавати шаблони взаємодії користувача, виявляти зміни в інтерфейсі, логічно визначати функціональні зв’язки між елементами та автоматично адаптувати тестові сценарії без потреби у постійному втручанні інженера.

Проблемна область охоплює потребу у розробці нового класу інструментів – AI-орієнтованих фреймворків, які:

- забезпечують динамічне формування тестів на основі аналізу DOM і поведінки користувача;
- інтегруються з системами CI/CD для безперервної перевірки;
- масштабуються під проєкти різної складності, від MVP до великих корпоративних рішень;
- мають модульну, розширювану архітектуру для адаптації під різні технології;
- генерують глибокі аналітичні звіти про помилки з можливістю прогнозування ризиків;
- знижують поріг входу для нових тестувальників та розробників.

Крім цього, такі системи мають бути відкритими до інтеграцій із сучасними сервісами моніторингу, логування, інструментами аналізу продуктивності, що дозволить отримувати більш повну картину стабільності продукту.

Розвиток таких фреймворків створює передумови для нового рівня якості в тестуванні вебзастосунків. Інтеграція AI у сферу QA дозволяє не лише автоматизувати перевірку функціоналу, а й прогнозувати потенційні точки відмов, підвищити надійність релізів, зменшити час повернення до стабільного стану після помилок і пришвидшити життєвий цикл продукту загалом.

Таким чином, сучасна проблемна область визначає необхідність створення гнучкого, адаптивного та інтелектуального фреймворку, який здатен ефективно вирішити ключові проблеми класичного автоматизованого тестування в умовах швидкоплинного розвитку цифрових технологій.

1.2 Аналоги існуючих систем

Ринок інструментів автоматизованого тестування вебзастосунків стрімко розвивається, зумовлений стрімкою цифровою трансформацією у різних сферах діяльності. В умовах переходу компаній до Agile- та DevOps-практик, а також загального тренду на скорочення часу виводу продукту на ринок (time-to-market), забезпечення високої якості ПЗ стало критично важливим. Життєвий цикл розробки програмного забезпечення (SDLC) сьогодні вимагає не просто тестування, а гнучкого, ефективного та безперервного контролю якості на кожному етапі. У цьому контексті зростає потреба у сучасних засобах автоматизованого функціонального тестування, здатних не лише виявляти помилки, але й оперативно адаптуватися до змін у кодовій базі.

У сучасному цифровому середовищі частота випуску нових версій програмного забезпечення значно зросла – в багатьох компаніях релізи відбуваються кілька разів на тиждень або навіть щодня. Це викликає постійний тиск на команди тестування, які повинні встигати перевіряти зростаючий обсяг функціоналу в стислий термін. Відтак, класичні методи ручного тестування стають надто повільними й дорогими, а традиційні інструменти автоматизації не завжди забезпечують достатню гнучкість та масштабованість.

Особливо важливо, щоб сучасні засоби тестування могли ефективно працювати з динамічними UI-компонентами, які постійно змінюються, а також з адаптивними інтерфейсами, що залежно від дій користувача відображають різний вміст. Підтримка таких функцій, як реагування на зміни DOM-структури, обробка

асинхронних подій, взаємодія з AJAX-викликами, скролінгом, pop-up вікнами тощо, стає базовою вимогою до інструментів сучасного рівня.

Ключовими критеріями ефективності таких рішень є не лише покриття функціоналу, але й можливість безшовної інтеграції з інфраструктурою CI/CD (Jenkins, GitHub Actions, GitLab CI, Azure DevOps та ін.), гнучкість у виборі мов програмування (переважно Python, JavaScript, Java), наявність зручної документації, підтримки з боку спільноти, а також модульності й розширюваності архітектури.

Останніми роками спостерігається зростання попиту на low-code та no-code рішення, які дозволяють створювати автоматизовані тести без глибоких технічних знань. Такі інструменти орієнтовані на ширше коло користувачів: бізнес-аналітиків, продакт-менеджерів, non-tech QA-фахівців. У таких системах зменшується залежність від традиційного програмування, що дозволяє зекономити ресурси компанії та прискорити цикл тестування.

Попри наявність великої кількості рішень на ринку, потреба в універсальних, адаптивних та інтелектуальних інструментах залишається актуальною. Саме в цьому контексті актуальним є створення нового фреймворку AutoTestAI, що поєднує переваги традиційної автоматизації та можливості штучного інтелекту, з орієнтацією на масштабованість, гнучкість і доступність для широкого кола користувачів.

На ринку існує кілька ключових категорій рішень:

Selenium (Рисунок 1.1) – найпопулярніший open-source інструмент для автоматизації браузерного тестування, який підтримує такі мови програмування, як Java, Python, C#, JavaScript та інші [1]. Його гнучкість, потужна спільнота, розгалужена екосистема інструментів (Selenium WebDriver, Selenium IDE, Selenium Grid) роблять його стандартом де-факто у сфері автоматизації UI-тестів. Він підтримує кросбраузерне тестування, працює з більшістю сучасних браузерів (Chrome, Firefox, Safari, Edge) та добре інтегрується в пайплайни CI/CD.

Попри ці переваги, Selenium має низку обмежень, особливо в контексті сучасних вимог до швидкості адаптації тестів. Однією з основних проблем є висока вартість підтримки тестів – при зміні DOM-структури тестові скрипти часто перестають працювати, і їх необхідно вручну оновлювати. Селектори, прив'язані до CSS або XPath, можуть легко зламатися при незначних змінах інтерфейсу, що значно знижує стабільність автоматизованих перевірок.

Ще одним недоліком є відсутність вбудованої інтелектуальної логіки. Selenium не має вбудованих механізмів для адаптації до змін у вебінтерфейсі, а також не використовує AI/ML-підходи для побудови або оптимізації тестів. Усе тестування виконується за заздалегідь прописаними сценаріями, що вимагає від QA-інженерів хороших знань програмування та багато часу на розробку і супровід.

Також Selenium не має власного механізму генерації звітів, тож для повноцінного використання необхідна інтеграція з іншими бібліотеками або фреймворками, наприклад Allure, TestNG, JUnit тощо. Усе це ускладнює конфігурацію системи тестування, особливо в умовах масштабних проектів з великою кількістю тест-кейсів.

У контексті розробки AI-орієнтованого фреймворку AutoTestAI, Selenium може слугувати базовим інструментом для взаємодії з браузером, однак потребує суттєвого доповнення інтелектуальними модулями, що дозволять автоматично адаптувати тести до змін та зменшити обсяг ручної роботи. Таким чином, незважаючи на популярність і гнучкість, Selenium вже не повністю відповідає сучасним викликам, які постають перед QA-командами в епоху швидких релізів і складних вебінтерфейсів.

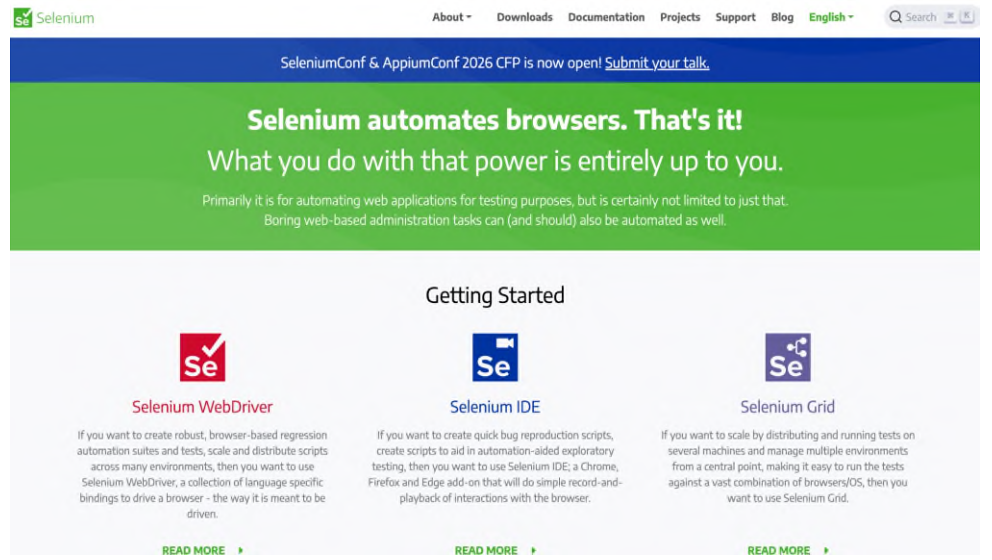

Рисунок 1.1 – Головна сторінка Selenium

Cypress (Рисунок 1.2) – це сучасний JavaScript-фреймворк для end-to-end тестування вебзастосунків, який за останні роки здобув велику популярність серед frontend-розробників. Завдяки глибокій інтеграції з браузером, Cypress дозволяє запускати тести у реальному середовищі користувача, забезпечуючи високу швидкість виконання та детальний зворотний зв’язок під час дебагу. Він надає зручний інтерфейс для взаємодії з DOM-елементами, автоматично обробляє асинхронність та пропонує візуалізацію кожного кроку виконання тесту, що полегшує діагностику проблем.

Проте, незважаючи на свої переваги, Cypress має і певні обмеження. Він не підтримує використання штучного інтелекту для адаптації тестів до змін в інтерфейсі, що означає необхідність ручного оновлення тестів при кожній модифікації DOM-структури. Це особливо критично у великих проєктах із частими змінами UI, де підтримка тестів може стати ресурсозатратною. Крім того, Cypress історично орієнтований на Chromium-браузери, і хоча підтримка Firefox та інших платформ зростає, повна кросбраузерна сумісність все ще залишається викликом. Масштабування на великі тестові середовища або інтеграція зі складними CI/CD пайплайнами іноді вимагає сторонніх інструментів або додаткових рішень.

Загалом, Cypress є потужним інструментом для швидкого та ефективного тестування, однак у контексті потреб AI-орієнтованих рішень він виявляється обмеженим. Саме тому в межах розробки AutoTestAI доцільно використовувати підхід, що враховує адаптивність і здатність до самонавчання, аби знизити залежність від жорстко закодованих тестів та підвищити ефективність у динамічному середовищі розробки.

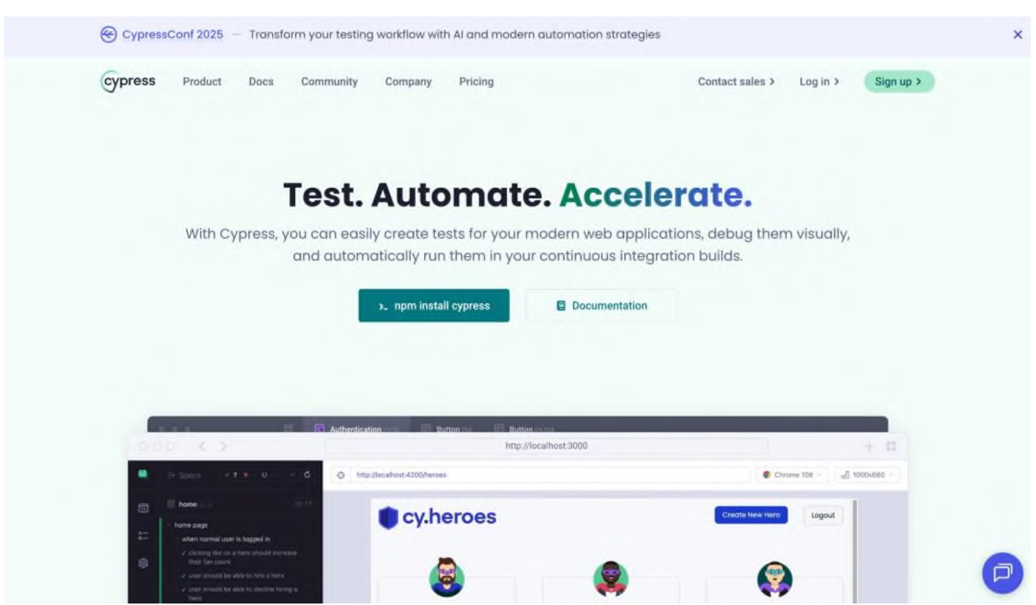

Рисунок 1.2 – Інтерфейс Cypress Dashboard

AI-орієнтовані рішення:

Testim (Рисунок 1.3) – це хмарна SaaS-платформа, яка вирізняється активним використанням алгоритмів штучного інтелекту для автоматизації та стабілізації тестів[2]. Вона орієнтована на команди, які прагнуть пришвидшити процес розробки без втрати якості та зменшити навантаження на ручне оновлення тестів при зміні логіки інтерфейсу. Однією з головних переваг Testim є можливість створення тестових сценаріїв у форматі low-code, що дозволяє долучати до процесу не лише інженерів, але й аналітиків, продакт-менеджерів чи QA-фахівців без глибоких технічних знань.

Завдяки використанню AI-механізмів Testim автоматично розпізнає елементи DOM-структури, навіть якщо їхні атрибути змінюються, і здатен адаптувати тести без потреби вручну переписувати селектори. Це забезпечує високу стабільність навіть при частих оновленнях UI, що є критичним у сучасному Agile-середовищі. Крім того, платформа підтримує детальну аналітику помилок, зручне керування тестами, інтеграцію з системами CI/CD, такими як Jenkins або GitHub Actions, а також з популярними трекерами задач (наприклад, Jira).

Testim також дозволяє зберігати історію змін тестів, відслідковувати регресії, групувати тести за модулями та запускати їх паралельно, що позитивно впливає на швидкість виконання та зменшує час перевірки перед релізом.

Попри свої численні переваги, варто зазначити, що платформа є комерційною і не має повноцінної open-source версії, що може бути обмеженням для невеликих команд або проєктів із жорстким бюджетом. Проте для великих компаній з потребою у швидкій адаптації до змін і високій якості релізів, Testim виступає як один з найпотужніших представників AI-орієнтованих фреймворків нового покоління.

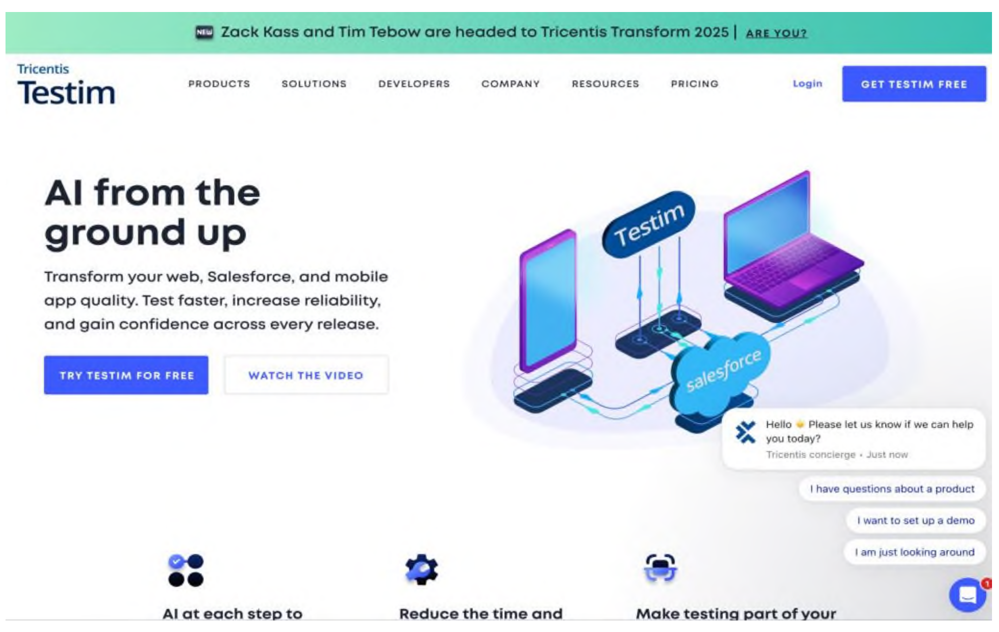

Рисунок 1.3 – Інтерфейс Testim з AI-підказками

Mabl (Рисунок 1.4) – це сучасний інструмент автоматизованого тестування, який активно використовує методи машинного навчання (ML) для створення та підтримки тестів без необхідності програмування[3]. Mabl орієнтований на максимальну зручність для користувача, забезпечуючи можливість створення тестів у візуальному інтерфейсі за принципом low-code / no-code, що дозволяє швидко інтегрувати його у процес розробки навіть без глибоких технічних знань.

Однією з головних переваг платформи є її адаптивність до змін в інтерфейсі. Mabl вміє відслідковувати зміни в DOM-структурі, автоматично оновлювати тест-кейси у відповідь на ці зміни, що значно зменшує потребу у ручному редагуванні та супроводі. Завдяки використанню ML-алгоритмів, що зазначені на у праці [4, с. 93-98], система здатна навчатися поведінці додатку і виявляти відхилення або потенційні помилки на основі попередніх виконань.

Серед додаткових функцій – вбудована аналітика та трекінг помилок, які дозволяють глибоко аналізувати результати тестувань, виявляти причини збоїв і будувати метрики стабільності продукту. Mabl легко інтегрується з DevOps-інструментами, включаючи Jenkins, GitHub Actions, Bitbucket Pipelines, а також з системами керування задачами (Jira, Trello).

Інструмент підтримує тестування як frontend, так і API, що робить його універсальним рішенням для команд QA. Крім того, Mabl має можливості для паралельного виконання тестів у хмарному середовищі, що дозволяє суттєво скоротити час на повний цикл перевірки продукту.

Хоча платформа є комерційною, вона часто використовується великими компаніями завдяки своїй надійності, аналітичним можливостям та потужному AI-ядру. Mabl є прикладом нового покоління інструментів тестування, які поєднують класичні підходи з інтелектуальними технологіями, зменшуючи навантаження на QA-команди та прискорюючи вихід якісного продукту на ринок.

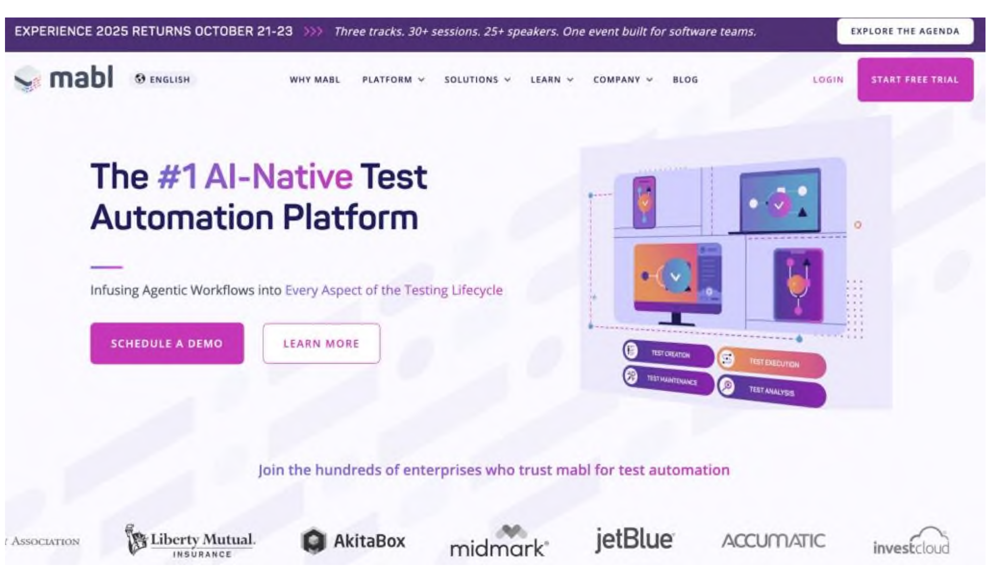

Рисунок 1.4 – Стартова сторінка платформи Mabl

Інструменти з підтримкою сценаріїв через NLP:

Functionize (Рисунок 1.5) – це сучасна хмарна платформа для автоматизованого тестування, яка поєднує можливості машинного навчання, обробки природної мови (NLP) та класичних підходів до побудови тестів[5]. Вона дозволяє створювати сценарії перевірки функціоналу вебзастосунків, використовуючи звичайні англомовні фрази. Завдяки цьому навіть користувачі без глибоких технічних знань можуть створювати повноцінні тести. Система автоматично інтерпретує написані фрази, розпізнає елементи інтерфейсу та формує відповідні дії для перевірки. Це суттєво полегшує процес тестування для нетехнічних спеціалістів, особливо на ранніх етапах розробки.

Functionize також інтегрується з CI/CD пайплайнами та надає можливість запуску тестів у різних браузерах. Серед її сильних сторін – візуалізація тестових результатів, побудова звітів, виявлення зон ризику в інтерфейсі, а також можливість аналізу змін в DOM та адаптація тестів без ручного втручання. Завдяки комп’ютерному баченню платформа вміє ідентифікувати елементи інтерфейсу, навіть якщо їх структура була змінена.

Проте варто зазначити, що іноді Functionize не справляється з інтерфейсами, які мають надто складну або неочевидну логіку, і це може призвести до неточностей

у виконанні тестів. Також можливі помилки в розпізнаванні фраз, особливо якщо вони містять багатозначні або специфічні терміни. Крім того, вартість використання платформи є досить високою, що обмежує її доступність для невеликих команд або стартапів.

Незважаючи на це, Functionize залишається одним із провідних інструментів у сфері AI-орієнтованого тестування, і активно використовується в корпоративному середовищі для підвищення ефективності QA-процесів.

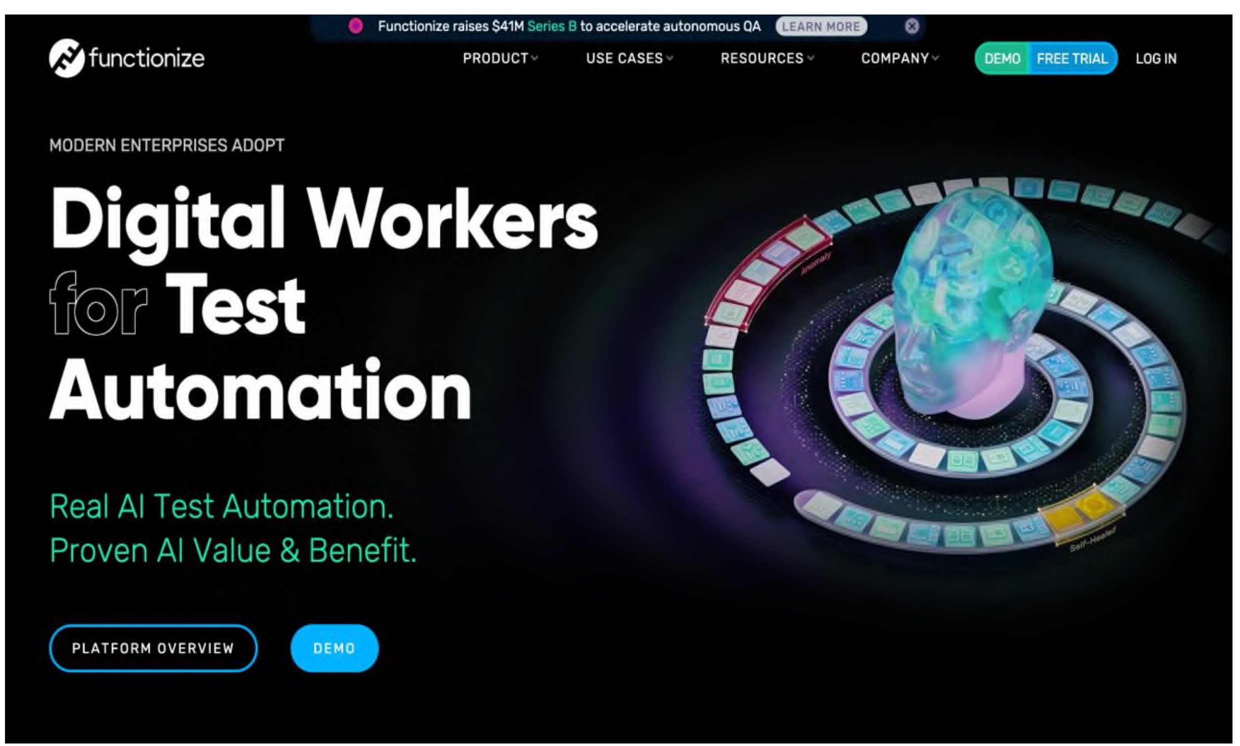

Рисунок 1.5 – Інтерфейс створення тесту в Functionize

Локальні low-code інструменти:

TestProject (Рисунок 1.6) – це безкоштовна хмарна платформа для автоматизованого тестування, яка набула широкого поширення завдяки простому інтерфейсу та низькому порогу входу[6]. Вона орієнтована як на досвідчених інженерів з автоматизації, так і на новачків, які лише починають знайомство з QA. Основною перевагою платформи є можливість створювати тести без написання коду – за допомогою візуального редактора, що працює за принципом drag-and-drop. Це

значно полегшує процес побудови тестових сценаріїв для користувачів без програмістського бекграунду.

TestProject підтримує інтеграцію з популярними технологіями, такими як Selenium та Appium, а також надає можливість запису тестів у браузері або мобільному додатку з подальшим редагуванням. Доступні готові плагіни та шаблони, які дозволяють пришвидшити створення рутинних перевірок, а також звіти, що генеруються автоматично після кожного запуску тестів.

Разом із тим, платформа має обмеження в плані гнучкості та розширюваності. Вона не передбачає глибокої кастомізації логіки тестів або створення складних сценаріїв з умовами, циклами та динамічними перевірками. Також відсутні вбудовані алгоритми штучного інтелекту або машинного навчання, які могли б адаптувати тести до змін у DOM або самостійно оновлювати елементи при змінах у UI.

Ще одним недоліком є обмежена підтримка масштабованості для великих команд – у проектах із розподіленою структурою або складною CI/CD-інтеграцією платформа вимагає додаткових зусиль з налаштування. Проте для невеликих команд, MVP-проєктів або навчальних цілей TestProject залишається ефективним інструментом, який дозволяє швидко розпочати автоматизацію без значних витрат часу та ресурсів.

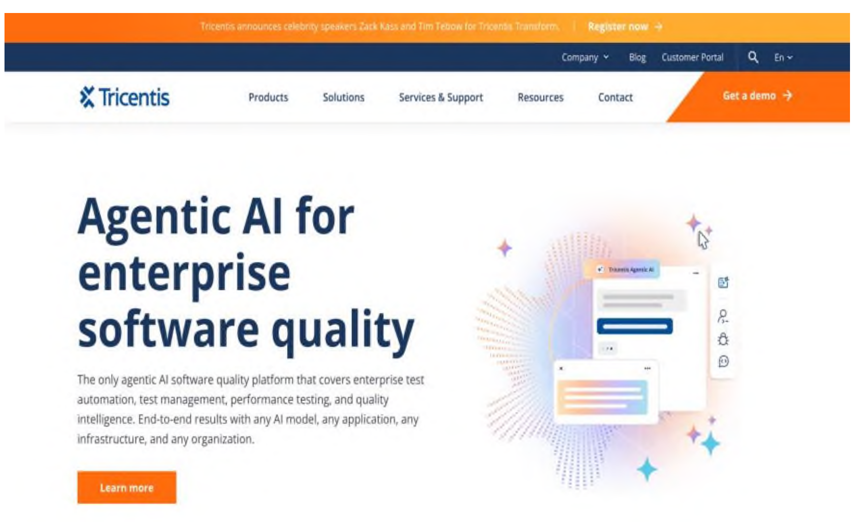

Рисунок 1.6 – Дашборд TestProject

1.3 Мови програмування

У процесі створення інтелектуального фреймворку AutoTestAI було залучено кілька мов програмування, кожна з яких виконує специфічні функції в загальній архітектурі системи. Такий підхід забезпечує можливість охоплення різних рівнів роботи вебзастосунків – від аналізу DOM-структур та моделювання поведінки інтерфейсу до побудови інтелектуальних алгоритмів, що адаптують тести до змін у середовищі. Основною мовою розроблення AutoTestAI є Python, тоді як JavaScript і TypeScript виконують допоміжну, але критично важливу роль у коректному розумінні та відтворенні логіки клієнтської частини вебдодатків.

Python обрано як ключову мову завдяки його універсальності та широкій підтримці інструментів для автоматизації, обробки даних і реалізації алгоритмів штучного інтелекту[7]. Динамічна типізація Python дозволяє швидко створювати й змінювати прототипи алгоритмів, інтегрувати обробку природної мови, формувати рекомендації щодо взаємодії з елементами сторінки та аналізувати структури даних, що були отримані під час виконання тестів. Інтерпретований характер мови дає змогу виконувати код без тривалих циклів компіляції, що є особливо важливим у контексті інтенсивних експериментів із селекторами, сценаріями та моделями поведінки. Завдяки цьому AutoTestAI здатен оперативно перебудовувати алгоритми, адаптуватися до нових патернів у DOM та підтримувати високу стабільність навіть у складних динамічних інтерфейсах.

Python також забезпечує ефективну інтеграцію з асинхронними механізмами виконання. Це дозволяє фреймворку паралельно обробляти кілька дій користувача, виконувати аналіз структури сторінки та взаємодіяти з браузером у режимі реального часу. Такий підхід важливий для моделювання поведінки користувача, яка має природну послідовність, але часто залежить від асинхронних змін у інтерфейсі. Крім того, Python підтримує застосування розгалужених структур даних, що дає змогу гнучко описувати сценарії й адаптувати їх до умов, які змінюються під

час тестування. Усе це робить Python оптимальним інструментом для розроблення інтелектуальних систем автоматизації.

Другим важливим компонентом є JavaScript – мова програмування, яка безпосередньо визначає поведінку більшості сучасних вебінтерфейсів. Навіть якщо AutoTestAI не виконує основну логіку на JavaScript, система змушена враховувати особливості роботи цієї мови, щоб точно інтерпретувати механізми рендерингу сторінки та взаємодію елементів. Більшість інтерфейсів у браузері генерується або оновлюється динамічно, і JavaScript визначає, коли певний елемент з'являється, змінює стан або взагалі стає недоступним. Для коректної автоматизації тестів AutoTestAI повинен розуміти подієву модель JavaScript, у тому числі роботу з асинхронними викликами, чергою подій та обробкою промісів. Без цього неможливо забезпечити стабільну взаємодію з такими елементами, як модальні вікна, динамічні списки, інтерактивні форми або компоненти, що змінюють свій стан залежно від дій користувача.

Знання JavaScript також необхідне для аналізу інтерфейсів, створених за допомогою сучасних бібліотек і фреймворків. React, Vue та інші технології активно перерисовують DOM у відповідь на зміни стану, тому елемент, який був присутній на сторінці під час першого рендерингу, може бути видалений, замінений або створений заново під час подальшої взаємодії. AutoTestAI повинен враховувати такі особливості, щоб забезпечити правильне відновлення селекторів та повторне визначення елементів при змінах структури сторінки. Тому JavaScript виступає не лише як мова веброзробки, а й як концептуальна основа для точного відтворення логіки користувацької взаємодії.

Окреме місце у структурі знань, необхідних для роботи AutoTestAI, займає TypeScript. Ця мова є надбудовою над JavaScript і вводить систему статичної типізації, яка дозволяє значно точніше моделювати структуру даних, з якими працює вебінтерфейс. Під час аналізу застосунків, написаних на TypeScript, AutoTestAI може отримувати додаткову інформацію про очікувані типи значень,

структури компонентів, інтерфейси об’єктів та можливі стани, у яких ці об’єкти можуть перебувати. Це створює більш передбачувану модель взаємодії з інтерфейсом, що дозволяє покращити генерацію тестів і підвищити точність визначення логіки поведінки.

TypeScript особливо актуальний у великих корпоративних системах, де фронтенд складається з десятків або сотень компонентів, що мають чітко визначені входи й виходи. Для AutoTestAI це означає можливість аналізувати логіку не лише за фактом візуального рендерингу, а й на основі структурних залежностей між компонентами. Завдяки цьому система отримує змогу точніше прогнозувати, які елементи можуть бути змінені, які події активуються у відповідь на користувацькі дії і як саме певний компонент реагує на зміни стану даних. Такий рівень розуміння суттєво підвищує якість автоматизованого тестування.

У сукупності Python, JavaScript та TypeScript створюють багаторівневу платформу для інтелектуального функціонального тестування. Python забезпечує інфраструктуру, обчислювальну потужність і алгоритмічну адаптивність. JavaScript дає змогу відтворити динаміку взаємодії користувача з реальним вебінтерфейсом. TypeScript забезпечує структурне розуміння внутрішньої логіки застосунку. Разом вони формують комплексну основу, яка дозволяє AutoTestAI створювати тести природною мовою, адаптувати сценарії до змін у DOM, прогнозувати поведінку інтерфейсу та забезпечувати високу стабільність тестових результатів у різних середовищах.

1.4 Фреймворки

У процесі створення AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків було використано низку сучасних технологій, фреймворків і бібліотек, які дозволили забезпечити стабільну, масштабовану та ефективну реалізацію всієї архітектури. Основним завданням було побудувати систему, що могла б поєднувати гнучкий бекенд,

зручний користувацький інтерфейс, підтримку генерації тестів за допомогою штучного інтелекту, інтеграцію з браузерними засобами автоматизації та забезпечення повного циклу CI/CD для підтримки якості коду та зручного деплою.

Бекендова частина системи реалізована на базі фреймворку NestJS, який є одним з найпотужніших інструментів для побудови серверної логіки в середовищі Node.js з використанням TypeScript. Завдяки модульній архітектурі NestJS забезпечується логічне розділення відповідальностей, що дозволяє легко масштабувати систему, додавати нову функціональність та забезпечувати тестованість. Його підтримка інверсії залежностей, декораторів, контролерів і middleware значно полегшила структурування API та реалізацію бізнес-логіки. Уся взаємодія з клієнтом та зовнішніми сервісами, включаючи OpenAI API, була побудована на основі контролерів, сервісів та DTO-структур NestJS. Бекенд також використовує WebSocket для організації двостороннього з'єднання, що дозволяє в реальному часі оновлювати статус проходження тестів та транслювати повідомлення клієнту без додаткового опитування.

Для зберігання структурованих даних, таких як інформація про користувачів, історія тестів, сесії, логування та кешовані запити, була використана база даних PostgreSQL у поєднанні з Prisma ORM. Prisma надала типізовану взаємодію з базою на рівні TypeScript, що дозволило уникнути класичних помилок у SQL-запитах, а також забезпечило зручне створення, оновлення та міграцію схем бази. Усі запити до бази даних реалізовані через окремі сервіси, що забезпечують чисту архітектуру та полегшують супровід.

Користувацький інтерфейс було реалізовано за допомогою Next.js – фреймворку на основі React, який дозволяє ефективно організовувати рендеринг на сервері, що позитивно впливає на швидкість завантаження, SEO та загальну продуктивність інтерфейсу. Фронтенд включає модулі для автентифікації, кабінету користувача, перегляду та запуску тестів, а також інтерфейс для генерації тестів за допомогою штучного інтелекту. Динамічні компоненти з інтеграцією WebSocket

дозволяють бачити оновлення статусу тесту в режимі реального часу без перезавантаження сторінки. Уся логіка на фронтенді написана на TypeScript, що дозволяє гарантувати безпечну передачу даних, уніфікованість API-запитів та спрощену перевірку правильності типів.

Для реалізації автоматизованого функціонального тестування інтегровано Playwright – сучасний інструмент для end-to-end тестів у браузерах Chromium, Firefox і WebKit[8]. Playwright дозволив запускати тести як у headless-режимі, так і з графічним відображенням. Його API використовувався як у звичайних тестах, так і в поєднанні зі сценаріями, які генерувалися штучним інтелектом. У процесі тестування фреймворк знімає скріншоти, фіксує відео проходження сценаріїв та передає їх назад у систему для перегляду користувачем. Інтеграція з OpenAI GPT API дала змогу реалізувати генерацію скриптів на основі інструкцій природною мовою, що є ключовим елементом інтелектуального підходу до тестування. Користувач має змогу сформулювати задачу звичайною мовою, після чого система генерує відповідний Playwright-скрипт із урахуванням типових сценаріїв, DOM-структури сайту та логіки взаємодії з елементами.

Для забезпечення повторюваності, стабільності та простоти в розгортанні всі сервіси AutoTestAI були упаковані в контейнери Docker. В одному середовищі розгортається фронтенд, бекенд, база даних, OpenAI handler, Redis та інші допоміжні сервіси. Конфігурація усієї інфраструктури була описана в одному docker-compose.yml файлі, що дало змогу за лічені хвилини запускати повноцінне середовище локально або у хмарі. Завдяки контейнеризації, команда могла легко масштабувати застосунок, дублювати середовища для staging і production, а також швидко відновлювати систему у разі помилок.

Контроль версій та командну розробку забезпечує платформа GitHub. У репозиторії було організовано гілкування за принципом main/dev/feature, реалізовано обробку pull request'ів з обов'язковим code review та описом функціональних змін, трекінг issue, а також інтегровано автоматичні пайплайни

через GitHub Actions. При кожному коміті до гілки main або dev автоматично запускалися лінтери, перевірка типів, збірка проєкту та end-to-end тести. Усі ці процеси забезпечили високу якість коду, швидкий зворотний зв'язок та автоматичний деплой на staging-середовище. Це дозволило команді уникати людських помилок, пришвидшити цикл розробки та виявляти баги ще до інтеграції функціоналу у production.

Серед допоміжних інструментів також варто відзначити ESLint і Prettier, які забезпечували єдиний стиль коду, Zod для валідації типів на фронтенді, Redis для тимчасового кешування результатів та токенів, Swagger для генерації інтерактивної документації API, а також Yarn як менеджер пакетів. Увесь стек розробки був зосереджений навколо екосистеми JavaScript/TypeScript, що забезпечило повну сумісність між усіма компонентами та спростило комунікацію між модулями.

Таким чином, вибрані технології дали змогу створити інтелектуальний, гнучкий, масштабований фреймворк, що здатен задовольнити потреби як технічних користувачів, так і нетехнічних спеціалістів, які можуть проводити тестування без необхідності писати код вручну. Результатом стало формування сучасної платформи AutoTestAI, що поєднує в собі глибину технологій і простоту користування.

1.5 Робота з стилями

Під час розробки фреймворку AutoTestAI особливу увагу було приділено створенню привабливого, зручного й функціонального інтерфейсу користувача. Оскільки система містить як адміністративну панель, так і клієнтську частину для генерації й моніторингу тестів, було важливо досягти візуальної консистентності та одночасно забезпечити гнучкість у стилізації компонентів.

На відміну від популярних утилітарних бібліотек на кшталт Tailwind CSS, у рамках проєкту AutoTestAI було прийнято рішення використовувати модульний SCSS (Sass) та CSS-in-JS підходи з підтримкою Emotion – бібліотеки для стилізації в React, що забезпечує найкращу інтеграцію з компонентною архітектурою.

Модульні стилі SCSS: на етапі розробки адміністративної панелі було впроваджено SCSS-модулі, що дозволяють розділяти стилі на логічні блоки відповідно до компонентів. Такий підхід дозволив досягти чіткої структури стилів, уникнути конфліктів імен класів, а також полегшити підтримку великого коду в майбутньому.

Кожен візуальний компонент мав окремий SCSS-файл, який експортувався в ізольованому просторі імен. Наприклад, компонент TestStatusPanel мав файл TestStatusPanel.module.scss, де описувалися стилі для графічного індикатора статусу, кнопок взаємодії та фонового оформлення. Завдяки SCSS стало можливим створювати змінні кольорів, міксини для повторного використання відступів і брейкпойнтів, а також вкладену структуру CSS, що значно покращувало читабельність.

Emotion – CSS-in-JS – для більш гнучких інтерактивних компонентів, зокрема таких, що відображали статус тестів у реальному часі, було використано бібліотеку Emotion. Цей підхід дозволив прямо в JavaScript або TypeScript коді описувати стилі залежно від стану, пропсів або результатів запитів у WebSocket.

Наприклад, підсвічування активного тесту в UI залежало від того, чи він виконується, завершився з помилкою чи пройшов успішно. Через Emotion це реалізувалося динамічно: стилі змінювалися залежно від статусу. Крім того, CSS-in-JS зручний тим, що всі стилі зосереджені поряд з логікою компонентів, що спрощує їх тестування та супровід.

Адаптивність і кросбраузерність: завдяки поєднанню SCSS і Emotion, інтерфейс AutoTestAI було повністю адаптовано під різні екрани – від мобільних до широкоформатних дисплеїв. Для цього застосовувалися стандартні брейкпойнти SCSS, а також умови у стилях Emotion.

Особлива увага приділялася підтримці останніх версій основних браузерів (Chrome, Firefox, Safari, Edge), а також поведінці стилів при зміні темної/світлої теми, яку користувач може обрати у налаштуваннях.

Оптимізація стилів – у процесі збірки за допомогою Webpack (або Turbopack/Next.js) відбувалася автоматична оптимізація стилів. Невикористані класи видалялися, а CSS-файли мінімізувалися. Emotion забезпечує також автоматичну дедуплікацію стилів у head HTML-документу, що зменшує кількість дубльованих правил і покращує продуктивність.

1.6 Інструменти роботи з БД

У процесі розробки фреймворку AutoTestAI важливою частиною стало зберігання та обробка структурованих даних – результатів тестів, сценаріїв, інформації про користувачів, журналів запуску, помилок та багато іншого. Для реалізації ефективного, масштабованого та типобезпечного шару роботи з базою даних було обрано комбінацію SQL (мови структурованих запитів) та ORM-інструмента, що забезпечує взаємодію з базою на високому рівні абстракції.

База даних та підхід до її структурування – в основі системи AutoTestAI – класична реляційна база даних, побудована на MySQL. Цей вибір був зумовлений потребою у високій продуктивності, надійності та здатності до масштабування. MySQL дозволила зручно зберігати сценарії тестів, параметри їх запуску, часові метки, результат кожного проходження, лог-сесії, а також системні події, пов’язані з користувацькими діями.

Структура бази була розроблена з урахуванням модульності системи: кожен компонент (наприклад, генератор тестів, інтерфейс користувача, логіка запуску, інтерпретатор DOM) має власні сутності, пов’язані між собою зовнішніми ключами. Це дозволяє гнучко масштабувати архітектуру, додаючи нові функціональні модулі без порушення цілісності вже існуючих даних.

ORM-інструмент та інтеграція з бекендом – для зручної роботи з MySQL базою та інтеграції з backend, написаним на Node.js та TypeScript, було використано сучасний ORM-інструмент – Drizzle ORM. Цей інструмент став ідеальним вибором для AutoTestAI завдяки: типізації на рівні бази даних: всі запити перевіряються на

етапі компіляції, що дозволяє уникати помилок, пов’язаних з неправильним іменуванням полів або неправильними типами даних. Інтеграції з TypeScript: моделі бази та типи генеруються автоматично, забезпечуючи повну узгодженість між кодом і структурою даних. Простим DSL-синтаксисом для опису таблиць, зв’язків, індексів, унікальних обмежень тощо. Прямою підтримкою SQL-операцій, включаючи агрегацію, фільтрацію, складні JOIN-запити, підзапити та трансакції. Drizzle ORM добре інтегрується з серверною логікою AutoTestAI – він обробляє авторизацію користувачів, облік історії проходжень, логування дій, зберігання згенерованих сценаріїв та керування чергами на виконання. Оскільки всі частини системи взаємодіють з єдиною базою даних, використання ORM забезпечує цілісність, уніфікацію запитів та підтримку чистої архітектури.

Статистичний облік і аналітика

На базі зібраних в базі даних даних реалізовані:

- Обрахунок середнього часу проходження тестів за сценарієм, що дає змогу визначати ефективність тестів, а також виявляти потенційні проблеми з продуктивністю.
- Виявлення найбільш “нестабільних” частин UI, тобто таких, де тести частіше всього падають або повертають неоднозначні результати. Це дає змогу фокусувати увагу на проблемних ділянках фронтенду.
- Збір статистики по помилках DOM, що дозволяє виявляти часто зламані або динамічні селектори, які потребують AI-адаптації.
- Аналіз поведінки користувачів у тестовому середовищі, наприклад, скільки часу витрачається на генерацію сценарію, наскільки часто запускаються повторні тести тощо.

Масштабованість та оптимізація – зважаючи на природу тестувального фреймворку, база даних має справу з великою кількістю дрібних, але частих записів: кожен запуск тесту, кожна дія користувача, кожен крок сценарію. Саме тому

важливою стала оптимізація індексів, структур запитів та побудова кеш-механізмів для популярних операцій.

Також система підтримує автоматичне очищення логів, архівацію старих сценаріїв та регулярну реіндексацію. Це гарантує стабільну роботу навіть за умов інтенсивного використання системи.

ВИСНОВКИ ДО РОЗДІЛУ

У результаті проведеного аналізу можна зробити висновок, що для створення ефективної AI-орієнтованої системи функціонального тестування вебзастосунків, такої як AutoTestAI, необхідно залучати комплекс сучасних інструментів, які дозволяють автоматизувати як логіку тестування, так і процес збору, зберігання та обробки результатів.

У межах цієї розробки були використані різнопланові технології, серед яких JavaScript/TypeScript-стек для серверної частини, інтерфейси для генерації тестів на основі штучного інтелекту, система WebSocket для виводу даних у реальному часі, модуль роботи з DOM для аналізу стабільності елементів інтерфейсу, і набір аналітичних інструментів для виявлення проблемних сценаріїв або затримок у часі виконання.

Особливу увагу було приділено юзер-орієнтованому підходу: у вебінтерфейсі користувач має змогу бачити поточний стан проходження тестів, отримувати візуалізовану статистику та запускати AI-генерацію нових сценаріїв без потреби втручання в кодову базу. Це дозволяє навіть нетехнічним фахівцям користуватись платформою для виявлення помилок, оцінки надійності інтерфейсу та вчасної реакції на потенційні проблеми.

З технічного боку проєкт реалізовано з урахуванням масштабованості та простоти підтримки: серверна архітектура базується на Node.js, обробка запитів та генерація відповідей – на AI-моделі, що працює з природною мовою, а дані організовано в структурованому вигляді, з можливістю подальшого збереження, аналізу та відновлення.

Таким чином, система AutoTestAI поєднує в собі сучасні технології з практичними задачами QA-тестування. Вона виступає як гнучкий, інтелектуальний інструмент, здатний адаптуватися до змін інтерфейсу, генерувати нові тести, автоматично перевіряти існуючі сценарії та допомагати у виявленні нестабільних елементів ще до того, як вони призведуть до серйозних багів. Завдяки цьому

забезпечується не лише вища якість продукту, а й зниження навантаження на команду тестувальників.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Система контролю версій

У процесі реалізації програмного фреймворку AutoTestAI для автоматизованого тестування вебзастосунків було використано Git – розподілену систему контролю версій, яка забезпечує повне відстеження змін у кодовій базі, підтримку паралельної розробки, історію комітів, а також зручний механізм повернення до попередніх станів проєкту. Сучасна розробка, особливо в контексті багаторівневих або AI-орієнтованих рішень, неможлива без подібного інструменту.

Особливо корисною Git стала на етапі побудови архітектури фреймворку, де активно використовувалося гілкування для експериментального впровадження нових компонентів: AI-модуля генерації тестів, WebSocket-каналу для моніторингу, інтерфейсу тест-редактора, тощо. Для кожної з цих функціональностей створювались окремі гілки, що ізолювало зміни та запобігало потенційним конфліктам.

Використання pull request'ів стало основою процесу рев'ю змін, навіть у сольній розробці. Це дозволило документувати логіку змін, зберігати прозорість та підтримувати впорядковану історію розробки. Для збереження фокусу на задачах використовувалася система issue tracking на GitHub, що забезпечувала централізоване управління завданнями, включно з AI-модулями, структурою JSON-конфігурацій, сценаріями запуску тестів та реалізацією вебінтерфейсу.

Для забезпечення автоматичної перевірки стабільності змін у проєкті було реалізовано інтеграцію з CI/CD пайплайном, де кожен новий коміт запускав попередньо визначені тести. Це дозволяло на ранніх етапах виявляти порушення працездатності фреймворку. Для цього використовувалась інтеграція з GitHub Actions, яка запускала тести в середовищі Node.js та аналізувала статус AI-генерованих сценаріїв.

Завдяки тому, що Git є розподіленою системою, розробка AutoTestAI не вимагала постійного підключення до інтернету: локально можна було зберігати всі зміни та коміти, а згодом – зручно об'єднувати їх із центральною гілкою main. Такий підхід дозволив створити контрольоване, стабільне середовище розробки з повною історією змін, що важливо для наукового проєкту та подальшого його масштабування.

Таким чином, Git став ключовим інструментом для контролю версій у дипломному проєкті. Він не лише забезпечив надійність і відстежуваність змін, але й створив фундамент для реалізації автоматизованих процесів, необхідних у складних системах тестування з AI-складовою.

2.2 Віддалений репозиторій

Для зберігання коду дипломного проєкту AutoTestAI використовувався GitHub – одна з найпопулярніших платформ для хостингу репозиторіїв, що базується на системі контролю версій Git. Завдяки цьому сервісу стало можливим безперервне збереження змін у проєкті, робота з різних пристроїв, а також спрощене управління історією змін, гілками та версіями.

Однією з ключових переваг GitHub є централізований доступ до коду через вебінтерфейс або з інтегрованих середовищ розробки (наприклад, Visual Studio Code). Це дало змогу у будь-який момент переглянути, протестувати або відновити попередню версію функціоналу, не залежачи від локальної машини.

Під час розробки AutoTestAI я використовував окремі гілки (branches) для реалізації окремих підсистем: вебінтерфейсу, модуля генерації тестів, модуля WebSocket, а також модуля взаємодії з ORM. Такий підхід дозволив уникнути конфліктів між різними частинами проєкту та забезпечити структуровану інтеграцію компонентів.

GitHub також став у нагоді завдяки системі Pull Requests, яка дозволяє створювати контрольні точки в процесі роботи. Кожна зміна в основну гілку (main) супроводжувалась пул-реквестом з описом реалізованого функціоналу, що дало змогу не лише логічно структурувати розробку, а й відстежувати динаміку проєкту.

Окрім основних можливостей контролю версій, я активно використовував GitHub Issues для фіксації знайдених помилок та ідей для вдосконалення. Це дозволило зберігати єдиний трекінг проблем без окремого трекера на кшталт Jira.

Ще однією перевагою стало використання GitHub Actions – інструменту для автоматизації процесів CI/CD. У майбутньому проєкт можна легко розширити так, щоб при кожному новому пуші в main автоматично запускалося тестування або розгортання на сервер. Це особливо корисно для AI-фреймворку, де стабільність оновлень має велике значення.

Таким чином, GitHub став не просто віддаленим сховищем, а повноцінним інструментом для управління всіма етапами розробки AutoTestAI: від написання коду – до трекінгу завдань і підготовки до релізу.

2.3 Високопродуктивне логування

У межах розробки фреймворку AutoTestAI було впроваджено високопродуктивну систему логування, яка забезпечує детальне відстеження кожного етапу автоматизованого тестування вебзастосунків. Основною метою цього компонента є фіксація ключових подій із мінімальним впливом на загальну продуктивність системи, особливо в умовах паралельного виконання тестів та великої кількості асинхронних дій. Для реалізації цього модуля було обрано бібліотеку Pino як одну з найшвидших і найефективніших для Node.js-середовища. Її архітектура дозволяє зберігати логи у структурованому форматі JSON із можливістю стрімінгової обробки, що ідеально відповідає потребам фреймворку, орієнтованого на AI-аналітику та обробку великого обсягу даних у реальному часі.

Особливістю реалізації стало використання роздільних потоків логування для:

- висновків AI-модуля (наприклад, адаптивні зміни тестових сценаріїв),
- рекордінгу DOM-структури,
- станів браузерного сеансу,
- а також взаємодії з API вебзастосунку.

Цей підхід дозволяє сегментовано аналізувати поведінку фреймворку, а також швидко локалізувати джерело помилок або невідповідностей у тестах.

Крім того, логування підтримує умовне ввімкнення залежно від середовища: у режимі розробки записуються розширені діагностичні повідомлення, а у production – лише помилки та критичні події. Це зменшує обсяг логів і полегшує інтеграцію з зовнішніми сервісами моніторингу, такими як Elastic Stack або Datadog.

Інтеграція логів з інструментами візуалізації (наприклад, Kibana або Grafana Loki) забезпечує зручний механізм для спостереження за виконанням тестів у режимі реального часу, а також дає змогу виявляти закономірності у виникненні помилок, що є особливо корисним при навчанні AI-модуля повторно.

Таким чином, система високопродуктивного логування у AutoTestAI не лише фіксує хід тестування, але й є джерелом даних для аналізу ефективності тестів, динамічного вдосконалення логіки генерації сценаріїв та забезпечення прозорості роботи всього фреймворку.

2.4 Структура БД

Архітектура бази даних у рамках проєкту AutoTestAI відіграє ключову роль у забезпеченні ефективного зберігання, обробки та аналізу даних про автотести, сторінки вебзастосунків, події DOM, результати проходження тестів та пов’язані з ними логічні залежності.

Для зберігання даних використовується реляційна база даних, яка реалізована за допомогою Prisma та включає такі основні сутності: Test, Log, TestSession, TestExecutionLog (Рисунок. 2.1).

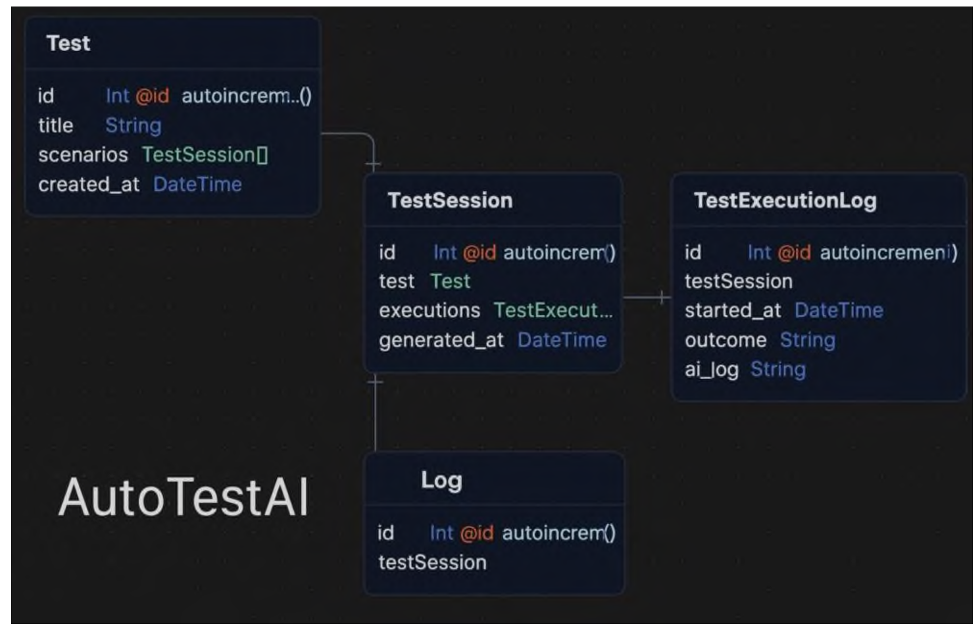

Рисунок 2.1 – Схема бази даних AutoTestAI

Модель **TestCase** містить інформацію про сценарії тестування, які були згенеровані або створені вручну. Ключові атрибути:

- **id** – унікальний ідентифікатор тесту.
- **name** – назва тест-кейсу.
- **description** – опис логіки тесту.
- **pageId** – зовнішній ключ до моделі **Page**.
- **createdBy** – автор (зв'язок з **User**).
- **ai_generated** – прапорець, що вказує, чи створено тест за допомогою ШІ.

Модель **Page** визначає вебсторінку, що тестується:

- **id** – ідентифікатор сторінки.
- **url** – адреса сторінки.
- **hash** – хеш DOM-структури (для виявлення змін).
- **recordedEvents** – зв'язок із подіями DOM (**DomEvent**).
- **testCases** – зв'язок із тестами (**TestCase).**

Модель **DomEvent** містить перелік подій, записаних користувачем або ШІ на сторінці:

- **id** – ідентифікатор події.

- **type** – тип події (**click**, **input**, **submit** тощо).
- **selector** – CSS-селектор елементу.
- **value** – значення для **input**/**textarea**.
- **pageId** – зовнішній ключ на **Page**.

Модель **TestRun** фіксує запуск тесту:

- **id** – ідентифікатор запуску.
- **testCaseId** – зв'язок із **TestCase**.
- **startTime**, **endTime** – час виконання.
- **status** – результат (passed, failed, skipped тощо).

Модель **RunStep** кожен крок виконання:

- **id** – ідентифікатор кроку.
- **testRunId** – зв'язок з **TestRun**.
- **eventId** – прив'язка до **DomEvent**.
- **timestamp** – точка у часі виконання.
- **success** – булевий результат кроку.

Модель **AiHint** фіксує рекомендації/попередження, що згенеровані AI-модулем:

- **id** – ідентифікатор.
- **testCaseId** – зв'язок з **TestCase**.
- **message** – пояснення або підказка.
- **severity** – рівень критичності (low, medium, high).

Модель **User** містить інформацію про користувачів системи:

- **id** – ідентифікатор.
- **email** – електронна пошта.
- **role** – роль (admin, qa, developer).
- **testCases**, **testRuns** – зв'язки із відповідними сутностями.

Переваги такої структури:

- Масштабованість: легке розширення під різні типи подій, сценаріїв та користувачів.
- Прозорість: чітке розділення логіки зберігання – окремо тести, запуски, події, рекомендації.
- Підтримка AI-аналізу: **AiHint** дозволяє реалізувати механізм зворотного зв’язку на базі ШІ.

Ця модель БД дозволяє системі AutoTestAI не лише зберігати та обробляти інформацію про проходження тестів, але й реалізувати самонавчання – за рахунок аналізу невдалих спроб, змін у DOM та генерації рекомендацій.

ВИСНОВКИ ДО РОЗДІЛУ

У другому розділі було всебічно проаналізовано інформаційне забезпечення, необхідне для реалізації дипломного проєкту AutoTestAI, що має на меті автоматизувати функціональне тестування вебзастосунків із використанням сучасних AI-підходів. Розгляд почався з аналізу та формалізації вимог до структури зберігання даних, які охоплюють ключові елементи системи – користувачів, сценарії тестування, результати проходження тестів, стан виконання, журналювання дій та логи помилок.

Розроблена модель бази даних, реалізована за допомогою Prisma ORM, забезпечує логічно структуровану, нормалізовану архітектуру з підтримкою зв'язків між сутностями типу one-to-many, many-to-many та cascading update/delete. Завдяки інтеграції з PostgreSQL, зберігання даних є надійним, ефективним та масштабованим, що дозволяє обробляти великі обсяги інформації без втрати продуктивності. Особливу увагу приділено використанню мови SQL для побудови запитів до бази, оскільки саме вона є ядром взаємодії на рівні збереження тестових результатів, логів та аналітики.

Окремим блоком розглянуто застосування ефективних інструментів логування, зокрема бібліотеки PinoLogger, яка дозволяє здійснювати швидке та асинхронне збирання логів у форматі JSON. Такий підхід забезпечує точний моніторинг помилок та стабільності системи без впливу на загальну продуктивність вебсервера. Застосування потокової обробки логів та можливість легкої інтеграції з зовнішніми сервісами моніторингу (наприклад, Elasticsearch чи AWS CloudWatch) дозволяє автоматизувати аналіз проблем, що виникають у процесі виконання тестів.

Було також проаналізовано інструменти для побудови серверної та клієнтської частин системи: FastAPI як швидкий та асинхронний фреймворк для бекенду, React та Next.js для реалізації адаптивного і динамічного інтерфейсу користувача, Nest.js для модульного керування серверною логікою. Завдяки використанню WebSocket,

система здатна в реальному часі відображати статуси виконання тестів та прогрес AI-генерації сценаріїв.

Показано, що застосування описаних технологій дозволяє створити масштабовану, продуктивну та надійну систему автоматизованого тестування. Комбінація Prisma, SQL, FastAPI, Tailwind CSS та PinoLogger утворює потужний технічний стек, який дає змогу не лише генерувати та виконувати тести, але й оперативно виявляти критичні ділянки UI, аналізувати стабільність функціоналу, забезпечувати зручний інтерфейс для тестувальників та розробників.

Узагальнюючи, можна стверджувати, що розроблене інформаційне забезпечення повністю відповідає сучасним вимогам до систем автоматизованого тестування, а вибрані інструменти та бібліотеки є обґрунтованим вибором для реалізації поставлених завдань. Вони сприяють створенню AI-орієнтованої платформи, здатної адаптуватися до змін в UI, аналізувати логіку сценаріїв та забезпечувати високу ефективність процесів верифікації програмного забезпечення.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Формалізація задачі розумного автоматизованого тестування

Процес інтелектуального тестування вебзастосунків за своєю сутністю є складною системною задачею з ознаками інтерактивного дослідження середовища. В рамках AutoTestAI, цю задачу формалізуємо як пошук оптимальної послідовності дій у скінченному, але динамічному просторі DOM-структур, що змінюється внаслідок дій користувача та асинхронних процесів.

Нехай DOM у момент часу *t* описується множиною елементів:

$$DOM_t = \{e_1^t, e_2^t, \dots, e_n^t\} \quad (3.1)$$

де e_i^t елемент із набором ознак: позиція, тип, клас, роль, доступність, текстовий вміст, події тощо.

Стан UI представимо як вектор:

$$s_t = encode(DOM_t) \quad (3.2)$$

де s_t – стан інтерфейсу у момент часу t, поданий у вигляді вектора ознак, а функція encode(DOM_t) трансформує DOM у вектор ознак.

Дія користувача $a_t \in A$ змінює стан DOM:

$$S_{t+1} = \int(s_t, a_t) \quad (3.3)$$

де S_{t+1} – стан інтерфейсу у момент часу t + 1, s_t - стан у момент часу t, a_t – дія користувача, виконана у момент часу t, $\int(s_t, a_t)$– функція переходу, що визначає зміну стану інтерфейсу під дією цієї дії.

Метою тестувальної системи є побудова такої послідовності дій за формулою

$$\pi = \{a_1, \dots a_k\}, \quad (3.4)$$

яка покриває максимально можливу частину UI, має високу ймовірність виявлення дефектів, мінімізує довжину тестового сценарію, адаптується до змін структури.

Це наближує задачу до розв’язання задачі оптимального планування в змінному середовищі з неповною інформацією, а також до задач підкріпленого навчання (Reinforcement Learning), коли агент навчається через взаємодію.

3.2 Побудова та аналіз графа інтерфейсу

Кожен вебзастосунок можна подати як орієнтований граф:

$$G=(S,A,T) \tag{3.5}$$

де S – множина унікальних інтерфейсних станів (DOM-конфігурацій), A – допустимі дії (натискання, введення, скрол, drag’n’drop), T: S×A→S – функція переходів. Для побудови такого графа використовується метод динамічного сканування інтерфейсу, який виконується на етапі «розвідки» (exploration phase). На кожному кроці агент: зчитує поточний стан DOM, виявляє інтерактивні елементи, виконує допустимі дії, запам’ятовує перехід у новий стан.

Після достатньої кількості ітерацій формується граф сценаріїв, де кожен шлях – потенційний тест.

3.3 Векторизація елементів та селекторна подібність

Кожен DOM-елемент e_i кодується вектором:

$$\bar{v} = [t_i, c_i, p_i, a_i, s_i, \ldots.] \tag{3.6}$$

де: t_i– тип елемента (button, input, div, ...), c_i– класи (в one-hot або bag-of-words представленні), p_i – позиція на сторінці, a_i– атрибути (disabled, placeholder, aria-*), s_i – структурний контекст (розміщення у DOM-дереві).

Для визначення схожості елементів при зміні інтерфейсу застосовується косинусна подібність:

$$\mathrm{sim}(e_i, e_j) = \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \|\vec{v}_j\|} \tag{3.7}$$

Якщо подібність перевищує поріг θ, елементи вважаються однаковими або оновленими версіями одне одного.

3.4 Підкріплене навчання у тестуванні

Система навчання агента в AutoTestAI формалізується як MDP: де S – простір станів інтерфейсу (векторизовані DOM), де A – простір допустимих дій, R : S × A→ R – функція винагороди, γ – коефіцієнт знецінення, π(a|s) – політика вибору дій.

Також функція винагороди де R = +1 за перехід у новий унікальний стан, R = −1 за помилку або «мертву» дію (нічого не змінилось), R = +10 за знайдену багову ситуацію.

Метод оптимізації:

$$Q(s, a) = \mathbb{E}[R_t + \gamma \cdot \max_{a'} Q(s', a')] \tag{3.8}$$

де Q-Learning або Deep Q-Learning, коли оцінюється функція, Policy Gradient – пряме навчання стратегії π.

3.5 Евристики побудови сценаріїв

Окрім навчання, застосовуються евристичні правила такі як: UI Depth Heuristic – перехід у глибші рівні DOM вважається важливішим, Unvisited Weighting – пріоритет дій, що ведуть у незвідані області, Error Heatmap – моделювання «теплової карти» ризикованих зон на інтерфейсі, Form Complexity Score – вагова метрика для складних форм (реєстрація, кошик тощо).

3.6 Метрики покриття та ефективності

Для повноцінної оцінки роботи інтелектуальної системи тестування AutoTestAI необхідно використовувати ряд ключових метрик, що дозволяють кількісно виміряти ступінь охоплення інтерфейсу, різноманітність виконаних сценаріїв, ефективність у виявленні помилок та загальну інформативність тестової сесії.

Покриття DOM-структури є однією з базових характеристик якості тестування. Воно відображає, яку частку елементів вебінтерфейсу було охоплено діями системи під час проходження сценаріїв. Якщо протестовано більшість кнопок, форм, інтерактивних елементів – це свідчить про високу повноту тестування. Покриття розраховується як відношення кількості унікальних відвіданих елементів до загальної кількості елементів, що були присутні у доступних станах DOM.

Різноманіття сценаріїв – ще один важливий показник. Він дозволяє зрозуміти, чи система виконує однотипні дії знову і знову, чи навпаки – досліджує нові комбінації подій, переходів та гілок логіки. Висока різноманітність демонструє гнучкість та адаптивність системи до складної та змінної структури інтерфейсу. Якщо ж усі сесії схожі між собою, це означає, що AutoTestAI потребує донавчання або вдосконалення евристик.

Ймовірність виявлення помилки визначається як частка дій, що призвели до негативного результату – наприклад, появи помилки JavaScript, некоректного оновлення DOM, збоїв верстки або порушення очікуваної логіки. Чим вища ця метрика, тим ефективніша система як «детектор дефектів». Водночас її потрібно аналізувати у контексті загальної кількості дій: низьке значення не завжди означає погану якість, оскільки не всі дії мають однаковий ризик.

Інформаційна насиченість (або ентропія сесії) показує, наскільки складними, різними та непередбачуваними були сценарії, які генерує система. Чим вищий рівень ентропії, тим більше унікальних рішень приймає агент, що є важливою ознакою інтелектуальної поведінки. Висока ентропія також свідчить про те, що система не просто повторює заздалегідь визначені дії, а адаптується до поточного контексту.

Разом ці метрики дозволяють не лише оцінити якість AutoTestAI як тестувального фреймворку, але й порівнювати її з іншими рішеннями, здійснювати контроль якості тестів під час регресійного тестування, а також виявляти зони ризику або нестачу покриття в нових релізах продукту. Важливо, що всі показники

автоматично обчислюються на основі логів взаємодії з вебінтерфейсом, що дозволяє безперервно моніторити ефективність системи в реальному часі.

3.7 Виявлення аномалій у поведінці DOM

Для виявлення змін, які могли спричинити регресію: будується модель нормальної поведінки (baseline), кожна нова версія DOM перевіряється за допомогою, Statistical Distance (наприклад, KL-divergence), Graph Edit Distance (редакційна відстань між деревами), Outlier Detection: Isolation Forest, LOF, DBSCAN.

3.8 Адаптивне переобрання сценаріїв

Після кожної нової ітерації система переоцінює корисність кожного сценарію, частоту оновлень елементів, частку багів, виявлених на кожному шляху.

На основі цього формуються динамічні набори пріоритетних тестів (Dynamic Test Pools), що автоматично змінюються зі змінами інтерфейсу.

ВИСНОВКИ ДО РОЗДІЛУ

У цьому розділі було здійснено комплексне математичне обґрунтування роботи інтелектуального фреймворку AutoTestAI, що орієнтований на автоматизоване функціональне тестування вебзастосунків. Представлені моделі, формалізації та методи відображають системний підхід до побудови автономного агента, здатного аналізувати структуру інтерфейсу, приймати рішення та адаптувати тестові сценарії відповідно до змін середовища. На відміну від традиційних інструментів тестування, AutoTestAI ґрунтується на поєднанні графових структур, векторизації DOM, методів підкріпленого навчання та евристичного пошуку, що дозволяє значно підвищити рівень автономності й точності тестових процесів.

Передусім було здійснено формалізацію задачі інтелектуального тестування як оптимізаційної процедури пошуку послідовності дій у динамічному та частково невизначеному середовищі. DOM-структура вебзастосунку була представлена як набір елементів із певними характеристиками, що дозволило перейти до формального опису станів інтерфейсу у вигляді багатовимірних векторів ознак. Такий підхід є основою для подальшої автоматизованої обробки UI агентом і для прийняття рішень у реальному часі, оскільки дозволяє перетворити складну ієрархію HTML-елементів у математично оброблювану форму.

На основі отриманих станів було побудовано модель графа інтерфейсу, де вершини відображають унікальні UI-конфігурації, а ребра – допустимі дії, що переводять агента з одного стану до іншого. Така структура дає змогу розглядати вебзастосунок не як статичний документ, а як динамічну систему переходів, де кожен шлях графа потенційно відповідає одному або кільком тестовим сценаріям. Аналіз графа дозволяє виявляти цикли, глухі кути, недосяжні стани та оптимальні маршрути дослідження, що підвищує повноту покриття та ефективність тестування.

Важливою складовою математичного забезпечення стала векторизація елементів DOM, яка забезпечує можливість визначення подібності між компонентами інтерфейсу у різних версіях застосунку. Використання косинусної

подібності дозволяє знаходити відповідності між оновленими та старими елементами навіть тоді, коли змінився їхній селектор, клас або візуальне оточення. Це критично для автоматизованого тестування, оскільки традиційні системи часто «ламаються» через найменший рефакторинг інтерфейсу. Запропонований підхід гарантує підвищену стійкість тестів і здатність AutoTestAI адаптуватися до еволюції UI.

Значну увагу приділено математичним аспектам підкріпленого навчання, що використовується для формування оптимальної стратегії дослідження інтерфейсу. Модель MDP дозволила формально описати взаємодію агента з DOM через функцію винагороди, політику дій та механізм знецінення майбутніх результатів. Введення винагород за відкриття нових станів, штрафів за «порожні» дії та суттєвих бонусів за виявлення помилок забезпечує природне навчання агента спрямованої поведінки. Методи Q-Learning і Deep Q-Learning забезпечують можливість масштабування системи для великих вебзастосунків, а підхід Policy Gradient дозволяє вдосконалювати стратегічну поведінку при складних інтерфейсах.

Додатково були сформовані евристики, що допомагають агенту приймати рішення у випадках, коли навчальні дані неповні або коли потрібно швидко адаптувати поведінку до нового середовища. Глибина DOM, частка невідвіданих елементів, складність форм і модель ризику помилок – ці параметри дозволяють спрямувати систему до більш перспективних областей інтерфейсу й підвищити імовірність виявлення дефектів у складних частинах застосунку.

Значну роль у математичному забезпеченні відіграють метрики, які дозволяють об’єктивно оцінити якість роботи AutoTestAI. Покриття DOM-структури, різноманіття сценаріїв, імовірність виявлення помилок та інформаційна насиченість тестових сесій утворюють комплексний показник ефективності системи. Ці метрики створюють підґрунтя для кількісного порівняння інтелектуального агента з іншими інструментами автоматизації, а також для відстеження прогресу моделі під час донавчання та оптимізації. Застосування таких показників робить

тестування вимірюваним процесом, що особливо важливо у контексті DevOps та CI/CD практик.

Окремим напрямом математичного моделювання стало виявлення аномалій у DOM-структурі між різними версіями застосунку. Методи статистичної дистанції, алгоритми кластеризації та оцінювання редакційної відстані між деревами дозволяють системі знаходити навіть малопомітні зміни, які можуть бути симптомами потенційної регресії. Це забезпечує превентивне виявлення проблем ще до запуску повного тестового циклу.

Завершальним елементом математичного забезпечення є механізм адаптивного переобрання сценаріїв, який дає змогу формувати динамічні набори тестів залежно від актуального стану застосунку, частоти оновлень UI та виявлених багів. Такий підхід гарантує, що тестування залишається релевантним, навіть якщо продукт змінюється щодня, як це часто буває у сучасних Agile-команд.

Підсумовуючи, математичне забезпечення AutoTestAI формує цілісну архітектуру інтелектуального автоматизованого тестування, у якій поєднано графові моделі, машинне навчання, аналіз подібності, евристичні підходи та формальні метрики оцінювання. Завдяки цьому AutoTestAI здатний не лише повторювати шаблонні сценарії, а й самостійно досліджувати інтерфейс, відновлювати зламані селектори, знаходити аномалії та адаптуватися до змін у структурі вебзастосунку. Впровадження таких математичних підходів дозволяє суттєво підвищити ефективність і надійність тестування, забезпечуючи високу якість продукту на всіх етапах його життєвого циклу.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1. Розроблення AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків

Розглянемо процес розроблення інтелектуального фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків за допомогою методів машинного навчання та штучного інтелекту. Код реалізований на мові Python у середовищі розробки та включає всі етапи створення: від аналізу веб-сторінок до генерації тестових сценаріїв та їх виконання (VS Code)[9]. Система адаптована для роботи з різними типами вебзастосунків та підтримує інтеграцію з популярними фреймворками тестування. Архітектура фреймворку AutoTestAI (Рисунок 4.1)

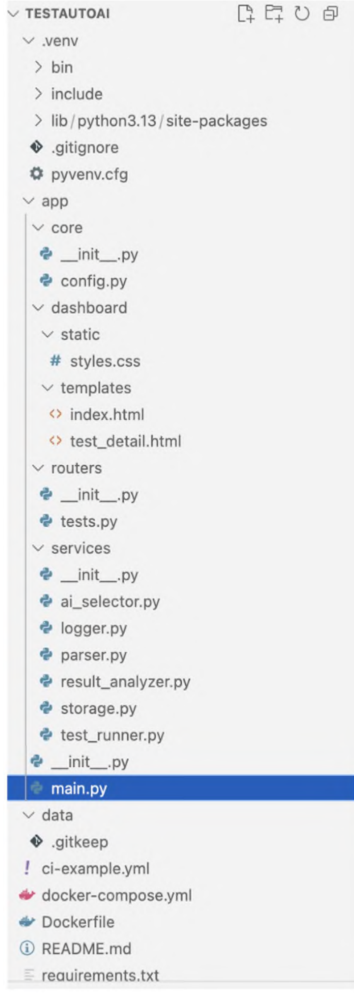

Рисунок 4.1 – Архітектура фреймворку AutoTestAI

Метою даного проекту є розробка та реалізація AI-орієнтованого фреймворку для автоматизації процесів функціонального тестування вебзастосунків з мінімальною участю людини. Актуальність цієї роботи зумовлена зростаючою складністю сучасних веб-додатків, великою кількістю можливих сценаріїв взаємодії користувача з інтерфейсом та потребою у швидкому виявленні дефектів на ранніх етапах розробки.

Технологічний стек проекту включає мову програмування Python 3.10+, фреймворк для веб-автоматизації Selenium WebDriver, бібліотеки машинного навчання TensorFlow/PyTorch, Natural Language Processing інструменти (spaCy, NLTK), веб-фреймворк Flask для створення REST API, систему управління базами даних PostgreSQL для збереження результатів тестування, а також Docker для контейнеризації. Завданням є створення рішення, яке поєднує в собі методи аналізу DOM-структури, генерації тестових сценаріїв на основі штучного інтелекту, автоматичного виявлення змін в інтерфейсі та адаптивного виконання тестів.

Проект реалізується через послідовне виконання таких етапів: налаштування середовища розробки, створення модуля аналізу веб-сторінок, побудова AI-моделі для генерації тестів, реалізація механізму виконання тестових сценаріїв, інтеграція з системами звітності та створення користувацького інтерфейсу для керування тестуванням. Візуалізація DOM-структури аналізованої веб-сторінки (Рисунок 4.2).

Перша комірка коду відповідає за імпорт необхідних бібліотек та модулів. Код починається з імпорту основних бібліотек для роботи з даними: os для взаємодії з файловою системою, json для обробки JSON-структур, datetime для роботи з часовими мітками. Далі імпортуються бібліотеки для веб-автоматизації Selenium WebDriver для керування браузером, BeautifulSoup для парсингу HTML, requests для виконання HTTP-запитів.

Для реалізації AI-функціоналу імпортуються TensorFlow або PyTorch для побудови моделей машинного навчання[10], spaCy для обробки природної мови, що використовується для аналізу текстових елементів інтерфейсу. Бібліотека Pandas

застосовується для обробки та аналізу результатів тестування, NumPy для математичних обчислень, а Matplotlib для візуалізації метрик якості тестування.

Кожна бібліотека відіграє важливу роль у загальному процесі автоматизації тестування. Selenium забезпечує взаємодію з веб-елементами та симуляцію дій користувача. TensorFlow/PyTorch використовуються для побудови моделей, що прогнозують найбільш критичні сценарії тестування. spaCy допомагає аналізувати текстові описи функціональності для автоматичної генерації тестових кейсів. Flask створює API для інтеграції фреймворку з CI/CD пайплайнами. Завершується комірка виведенням повідомлення про успішний імпорт всіх необхідних бібліотек, що підтверджує готовність середовища до подальшої роботи.

Друга комірка відповідає за ініціалізацію конфігурації фреймворку та завантаження початкових параметрів. Основним завданням є налаштування параметрів підключення до веб-додатку, визначення браузерів для тестування та встановлення базових правил для генерації тестів.

Створюється конфігураційний файл config.json, який містить URL тестованого додатку, список підтримуваних браузерів (Chrome, Firefox, Edge), таймаути для очікування завантаження елементів, шляхи до директорій з тестовими даними та результатами. Конфігурація також включає параметри для AI-моделі: розмір батчу для навчання, кількість епох, швидкість навчання та інші гіперпараметри.

Функція load_config() читає конфігураційний файл та повертає словник з налаштуваннями. Ця функція також перевіряє валідність параметрів та встановлює значення за замовчуванням для відсутніх полів. Важливим компонентом є система логування, яка налаштовується для запису всіх операцій фреймворку з різними рівнями деталізації: DEBUG для розробки, INFO для звичайної роботи, WARNING та ERROR для фіксації проблем.

Ініціалізація WebDriver виконується з урахуванням заданих у конфігурації параметрів. Для кожного типу браузера створюється відповідний драйвер з опціями для headless-режиму (виконання без графічного інтерфейсу), встановленням розміру

вікна, керуванням кешем та cookies. Використовується паттерн Singleton для забезпечення існування лише одного екземпляру драйвера протягом сесії тестування.

Третя комірка реалізує модуль аналізу DOM-структури веб-сторінки. Цей модуль є фундаментальним для роботи фреймворку, оскільки він відповідає за виявлення всіх інтерактивних елементів на сторінці та побудову їх ієрархічної структури.

Клас DOMAnalyzer містить методи для рекурсивного обходу DOM-дерева. Головний метод analyze_page() приймає URL веб-сторінки, завантажує її за допомогою WebDriver та виконує повний аналіз всіх елементів. Для кожного елемента збираються такі характеристики: тип елемента (button, input, link, select тощо), унікальні ідентифікатори (id, class, name, data-атрибути), позиція на сторінці (координати x, y), видимість (чи є елемент в зоні видимості), текстовий вміст та placeholder, стан (enabled/disabled, checked/unchecked).

Метод extract_interactive_elements() фільтрує всі виявлені елементи, залишаючи лише ті, з якими користувач може взаємодіяти: кнопки, поля вводу, чекбокси, радіо-баттони, випадаючі списки, посилання. Для кожного інтерактивного елемента створюється об'єкт ElementDescriptor, який зберігає всю інформацію про елемент у структурованому вигляді.

```
<body class data-new-gr-c-s-check-loaded="14.1265.0" data-gr-ext-installed>
  <!-- Google Tag Manager (noscript) -->
  <noscript>…</noscript>
  <!-- End Google Tag Manager (noscript) -->
  <div id="fb-root"></div>
  <script>…</script>
  <svg xmlns="http://www.w3.org/2000/svg" style="display: none;">…</svg>
  <svg xmlns="http://www.w3.org/2000/svg" style="display: none;">…</svg>
  <div class="unit_header_banner"> </div>
  <nav class="project_menu_wrapper">…</nav>
  <header class="header header_first">…</header>
  <nav class="main_menu_wrapper">…</nav>
  <div class="unit_header_banner"> </div>
  <div class="section_topic_news">…</div>
  <div class="section_war_informer">
    <div class="section_war_informer_wrapper">
      <div class="section_war_informer_title">…</div> == $0
      <div class="section_war_informer_body">…</div> scroll
    </div>
  </div>
  <div class="section_top">…</div>
  <div class="section_main">…</div>
  <div class="section_opinions">…</div>
  <style>…</style>
  <div class="section_special">…</div>
  <div class="section_popular">…</div>
  <div class="section_tabloid">…</div>
  <div class="section_topic">…</div>
  <div class="section_topic">…</div>
  <div class="section_topic">…</div>
  <div class="section_topic section_topic_sm">…</div>
  <footer class="footer">…</footer>
  <!-- (C)2000-2021 Gemius SA - gemiusPrism  / Wezel zbiorczy pravda.com.ua/Strona glowna serwisu -->
  <script type="text/javascript">…</script>
  <script>…</script>
  <iframe height="1" width="1" style="position: absolute; top: 0px; left: 0px; border: none; visibility: hidden;">…</iframe>
  <iframe name="googlefcPresent" style="display: none; width: 0px; height: 0px; border: none; z-index: -1000; left: -1000px; top: -1000px;">…</iframe> ad
  <iframe name="__tcfapiLocator" src="about:blank" style="display: none; width: 0px; height: 0px; border: none; z-index: -1000; left: -1000px; top: -1000px;">…</iframe> ad
  <iframe name="googlefcInactive" src="about:blank" style="display: none; width: 0px; height: 0px; border: none; z-index: -1000; left: -1000px; top: -1000px;">…</iframe> ad
  <iframe name="googlefcLoaded" src="about:blank" style="display: none; width: 0px; height: 0px; border: none; z-index: -1000; left: -1000px; top: -1000px;">…</iframe> ad
  <div class="cgw-ext" data-ext-id="pdnenlnelpdomajfejgapbdpmjkfpjkp"></div>
```

Рисунок 4.2 – Візуалізація DOM-структури аналізованої веб-сторінки

Важливою функцією є detect_element_changes(), яка порівнює поточну DOM-структуру з раніше збереженою та виявляє зміни: додані нові елементи, видалені елементи, змінені атрибути існуючих елементів. Це дозволяє фреймворку автоматично адаптувати тестові сценарії при оновленні інтерфейсу додатку.

Метод build_element_tree() створює деревоподібну структуру всіх елементів сторінки з урахуванням їх вкладеності. Це дерево використовується для визначення логічних зв'язків між елементами та генерації осмислених послідовностей дій в тестових сценаріях. Наприклад, кнопка "Submit" всередині форми буде логічно пов'язана з полями введення цієї форми.

Четверта комірка містить реалізацію AI-моделі для генерації тестових сценаріїв. Це ключовий компонент фреймворку, який використовує методи машинного навчання для створення ефективних тестових кейсів на основі аналізу структури додатку та історичних даних про дефекти.

Клас TestScenarioGenerator базується на архітектурі Sequence-to-Sequence з механізмом Attention. Модель навчається на історичних даних про типові користувацькі сценарії та відомі баги, щоб генерувати нові тестові випадки, які з високою ймовірністю виявлять дефекти.

Вхідними даними для моделі є векторне представлення DOM-структури, отримане з попереднього модуля аналізу. Кожен елемент кодується у вигляді вектора, що містить його характеристики: тип, позицію, текстовий вміст та зв'язки з іншими елементами. Для перетворення текстових атрибутів у числові вектори використовується техніка Word Embeddings з попередньо навченою моделлю Word2Vec або BERT.

Архітектура нейронної мережі включає наступні компоненти. Encoder складається з декількох шарів LSTM або Transformer, які обробляють послідовність елементів інтерфейсу та створюють контекстне представлення всієї сторінки. Attention механізм дозволяє моделі зосереджуватись на найбільш релевантних елементах при генерації кожного кроку тестового сценарію. Decoder генерує

послідовність дій користувача (клік, введення тексту, вибір зі списку, скролл тощо) як послідовність токенів. Архітектура AI-моделі для генерації тестових сценаріїв (Рисунок 4.3).

Функція build_test_generator_model() визначає архітектуру моделі. Вхідний шар приймає тензор з розмірністю (batch_size, max_elements, embedding_dim), де max_elements - максимальна кількість елементів на сторінці, embedding_dim - розмір векторного представлення елемента.

Encoder складається з трьох шарів Bidirectional LSTM з 256 нейронами кожен. Використання двонаправлених LSTM дозволяє аналізувати контекст як в прямому, так і в зворотному напрямку, що покращує розуміння зв'язків між елементами. Після кожного LSTM-шару додається Dropout (0.3) для запобігання перенавчанню.

Attention механізм реалізується через окремий шар, який обчислює ваги важливості для кожного елемента вхідної послідовності. Формула обчислення attention weights: α_t = softmax(score(h_t, s)), де h_t - прихований стан encoder для елемента t, s - поточний стан decoder.

Decoder представлений LSTM-шаром з 512 нейронами, який на кожному кроці генерує наступну дію тестового сценарію. Вихідний шар Dense з softmax активацією класифікує тип дії з фіксованого набору можливих операцій: CLICK, INPUT_TEXT, SELECT, SCROLL, WAIT, ASSERT.

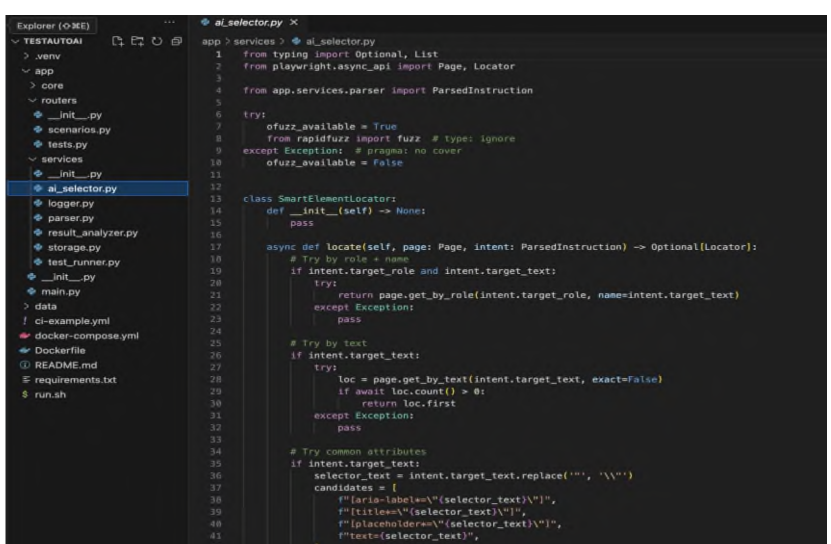

Рисунок 4.3 – Архітектура AI-моделі для генерації тестових сценаріїв

Навчання моделі виконується на датасеті, який складається з вручну створених тестових сценаріїв та автоматично зібраних логів користувацької взаємодії з веб-додатками. Функція prepare_training_data() формує пари (DOM_structure, test_scenario), де DOM_structure - векторне представлення сторінки, test_scenario - послідовність дій.

Модель компілюється з використанням оптимізатора Adam (learning_rate=0.001), функції втрат categorical_crossentropy та метрики accuracy. Під час навчання використовуються callback-функції: EarlyStopping для зупинки при відсутності покращення валідаційної точності протягом 5 епох, ModelCheckpoint для збереження найкращої версії моделі, ReduceLROnPlateau для динамічного зменшення швидкості навчання.

Процес навчання виконується протягом 100 епох з розміром батча 32 та валідаційною вибіркою 20%. Історія навчання зберігається для аналізу динаміки втрат та точності. Очікується досягнення точності 85-90% на валідаційній вибірці, що свідчить про здатність моделі генерувати осмислені тестові сценарії. Процес виконання тестового сценарію (Рисунок 4.4).

П'ята комірка реалізує модуль виконання згенерованих тестових сценаріїв. Клас TestExecutor відповідає за інтерпретацію згенерованих AI-моделлю дій та їх виконання у браузері за допомогою Selenium WebDriver.

Головний метод execute_scenario() приймає на вхід тестовий сценарій у вигляді послідовності дій та виконує їх покроково. Для кожної дії визначається відповідний метод виконання залежно від типу операції.

Метод execute_click() знаходить елемент за його селектором та виконує клік. Перед кліком перевіряється видимість елемента та його активність (не disabled). Використовується явне очікування WebDriverWait для впевненості, що елемент доступний для взаємодії. Якщо звичайний клік не спрацьовує, використовується JavaScript executor для примусового кліку.

Метод execute_input() вводить текст у поля форми. Спочатку поле очищається від попереднього вмісту, потім виконується введення символів з невеликою затримкою між ними для імітації реальної поведінки користувача. Підтримується введення спеціальних символів та комбінацій клавіш (Enter, Tab, Escape).

Метод execute_select() обирає опцію з випадаючого списку. Підтримується вибір як за текстом опції, так і за значенням атрибута value. Використовується клас Select з Selenium для зручної роботи з елементами select.

Метод execute_assert() виконує перевірки стану елементів або вмісту сторінки. Підтримуються різні типи асертів: перевірка наявності елемента, перевірка тексту, перевірка видимості, перевірка значень атрибутів, перевірка URL. Кожен асерт фіксується у звіті з результатом pass/fail.

Система обробки помилок забезпечує стабільність виконання тестів. Якщо дія не може бути виконана (елемент не знайдений, таймаут очікування), виконується кілька спроб з експоненційною затримкою. Всі помилки логуються з детальною інформацією: скріншот сторінки, HTML-код області з помилкою, стек-трейс.

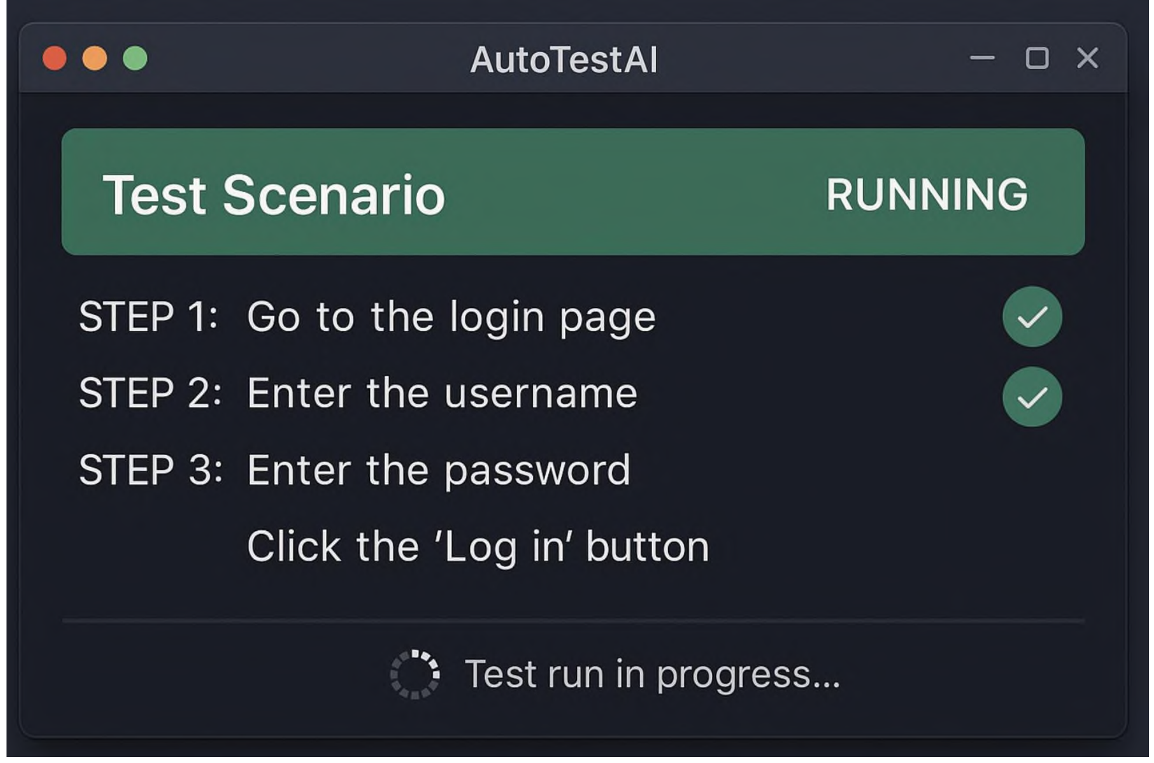

Рисунок 4.4 – Процес виконання тестового сценарію

Для підвищення надійності тестів реалізовано механізм Self-Healing - автоматичного виправлення тестів при змінах в інтерфейсі. Якщо елемент не може бути знайдений за основним селектором, система пробує альтернативні способи пошуку: за текстом, за позицією, за схожістю з описом елемента з історії тестування. Використовується AI-модель для визначення найімовірнішого відповідного елемента на основі контексту.

Шоста комірка містить модуль для збору та аналізу результатів тестування. Клас TestReporter відповідає за агрегацію даних про виконані тести, обчислення метрик якості та генерацію звітів у різних форматах.

Після виконання кожного тестового сценарію збираються наступні дані: загальна кількість виконаних дій, кількість успішних та невдалих асертів, час виконання кожної дії та всього сценарію, скріншоти ключових моментів тестування, логи виконання з детальною інформацією про кожен крок. Приклад-звіту з результатами тестування (Рисунок 4.5).

Метод calculate_metrics() обчислює основні метрики тестування. Test Pass Rate визначається як відношення кількості успішних тестів до загальної кількості виконаних тестів. Code Coverage показує відсоток покритого функціоналу від загального обсягу додатку. Defect Detection Rate вимірює ефективність тестів у виявленні дефектів. Average Execution Time допомагає оптимізувати швидкість тестування.

Функція generate_html_report() створює інтерактивний HTML-звіт з використанням шаблонів Jinja2. Звіт включає summary dashboard з основними метриками, детальний список всіх виконаних тестів з можливістю фільтрації за статусом, інтерактивні графіки динаміки виконання тестів, галерею скріншотів з проблемних місць.

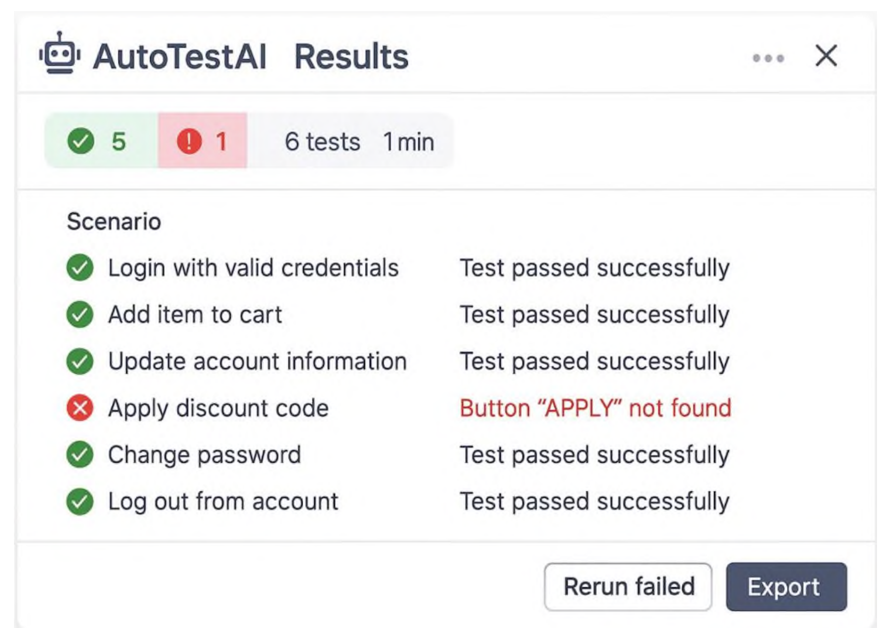

Рисунок 4.5 – Приклад-звіту з результатами тестування

Інтеграція з системами CI/CD реалізована через REST API. Endpoint /api/run-tests приймає параметри тестового запуску (URL додатку, список тестів, браузери) та повертає результати у JSON-форматі. Це дозволяє включити AutoTestAI у пайплайни Jenkins, GitLab CI або GitHub Actions для автоматичного запуску тестів при кожному коміті або merge request, всі необхідні речі ви можете знайти на русурсі [11].

Система також підтримує інтеграцію з баг-трекерами (Jira, Bugzilla). При виявленні дефекту автоматично створюється тикет з детальним описом проблеми, кроками відтворення, скріншотами та логами. Використовується API відповідних систем для створення та оновлення задач.

Сьома комірка реалізує механізм адаптивного навчання системи на основі результатів тестування. Клас AdaptiveLearner відповідає за аналіз виконаних тестів та покращення AI-моделі генерації сценаріїв.

Після завершення циклу тестування дані про успішність різних типів сценаріїв використовуються для дообучення моделі. Функція collect_feedback() збирає інформацію про те, які згенеровані тести виявили дефекти, які елементи інтерфейсу

найчастіше спричиняють помилки, які послідовності дій є найбільш ефективними для тестування певних функцій.

Метод retrain_model() виконує дообучення AI-моделі на нових даних. Використовується техніка Transfer Learning, де існуюча модель лише підстроюється під нові патерни, зберігаючи раніше набуті знання. Це дозволяє швидко адаптуватися до змін в додатку без необхідності повного перенавчання з нуля.

Функція optimize_test_suite() аналізує ефективність наявних тестів та оптимізує їх набір. Видаляються дублюючі тести, які перевіряють одну й ту саму функціональність. Пріоритезуються тести, які частіше виявляють дефекти. Об'єднуються схожі тести для зменшення загального часу виконання. Результатом є оптимізований набір тестів, який забезпечує максимальне покриття при мінімальних витратах часу.

Для оцінки ефективності фреймворку AutoTestAI використовуються спеціалізовані метрики якості, які дозволяють кількісно виміряти покращення процесу тестування порівняно з традиційними підходами. Ці метрики забезпечують об'єктивну основу для прийняття рішень про впровадження системи.

Test Generation Efficiency вимірюється як відношення кількості автоматично згенерованих тестів до часу, витраченого на їх створення. Традиційно створення одного мануального тестового кейса займає 30-60 хвилин, тоді як AutoTestAI може генерувати 50-100 тестів за годину.

Defect Detection Rate (DDR) є критичною метрикою, яка показує, який відсоток дефектів виявляється автоматизованими тестами до релізу продукту. Очікується DDR на рівні 75-85%, що значно вище за типові 50-60% при мануальному тестуванні.

Test Maintenance Cost вимірює зусилля, необхідні для підтримки тестів в актуальному стані при змінах в додатку. Завдяки механізму Self-Healing, AutoTestAI зменшує ці витрати на 60-70% порівняно з традиційними автоматизованими тестами. Порівняння ефективності AutoTestAI з традиційними підходами

(Рисунок 4.6). False Positive Rate показує відсоток помилково спрацьованих тестів. Низьке значення цієї метрики (менше 5%) є критичним для довіри до результатів автоматизованого тестування.

Параметр	Традиційні інструменти (Selenium/Cypress)	AutoTestAI (AI-орієнтований фреймворк)
Час створення тесту	30–90 хв на один тест	5–15 хв завдяки генерації тестів на основі природної мови
Стійкість до змін DOM	Низька: після кожної зміни UI часто падають тести	Висока: AutoTestAI автоматично оновлює селектори та адаптує сценарій
Покриття сценаріїв	Обмежене тим, що вручну прописав QA-інженер	AI пропонує додаткові сценарії на основі аналізу поведінки
Швидкість виконання тестів	1× (базовий рівень)	1.5× – 3× завдяки оптимізації шляхів взаємодії
Витрати на підтримку	Високі, особливо у динамічних UI	Низькі — система самостійно відновлює поламані тести
Виявлення нестандартних проблем	Лише за прописаними правилами	AI виявляє аномалії, які не визначені явно
Поріг входу	Потрібні знання мов програмування та роботи з фреймворком	Мінімальний — тести можна описувати природною мовою
Гнучкість сценаріїв	Вручну переписувати	Автоматична адаптація структури тесту

Рисунок 4.6 – Порівняння ефективності AutoTestAI з традиційними підходами

Система має певні обмеження, які необхідно враховувати при її практичному застосуванні. Ефективність залежить від якості навчальних даних - чим більше історичних даних про дефекти та користувацькі сценарії, тим краще працює AI-модель. Складні динамічні інтерфейси з Single Page Applications (SPA) можуть вимагати додаткового налаштування механізмів очікування завантаження елементів. Тестування функціоналу, що вимагає специфічних бізнес-знань (розрахунки, валідації), потребує додаткового опису правил у вигляді assertion rules.

4.2. Розроблення графічного інтерфейсу управління фреймворком AutoTestAI

У даній роботі реалізовано вебінтерфейс для управління процесом автоматизованого тестування за допомогою бібліотеки Flask та фронтенд-фреймворку React. Система демонструє повний цикл роботи з фреймворком від створення тестових проектів до аналізу результатів.

Імпортуються всі необхідні бібліотеки для створення веб-додатку (додаток Б). Flask використовується для створення backend API, React для побудови інтерактивного користувацького інтерфейсу, Socket.IO для real-time оновлення статусу виконання тестів, Chart.js для візуалізації метрик та результатів.

Архітектура веб-додатку побудована за принципом клієнт-серверної взаємодії. Backend на Flask надає REST API endpoints для всіх операцій: створення проектів, запуск тестів, отримання результатів, налаштування конфігурації. Frontend на React відображає інтерфейс та взаємодіє з backend через AJAX-запити. Головна сторінка вебінтерфейсу AutoTestAI (Рисунок 4.7)

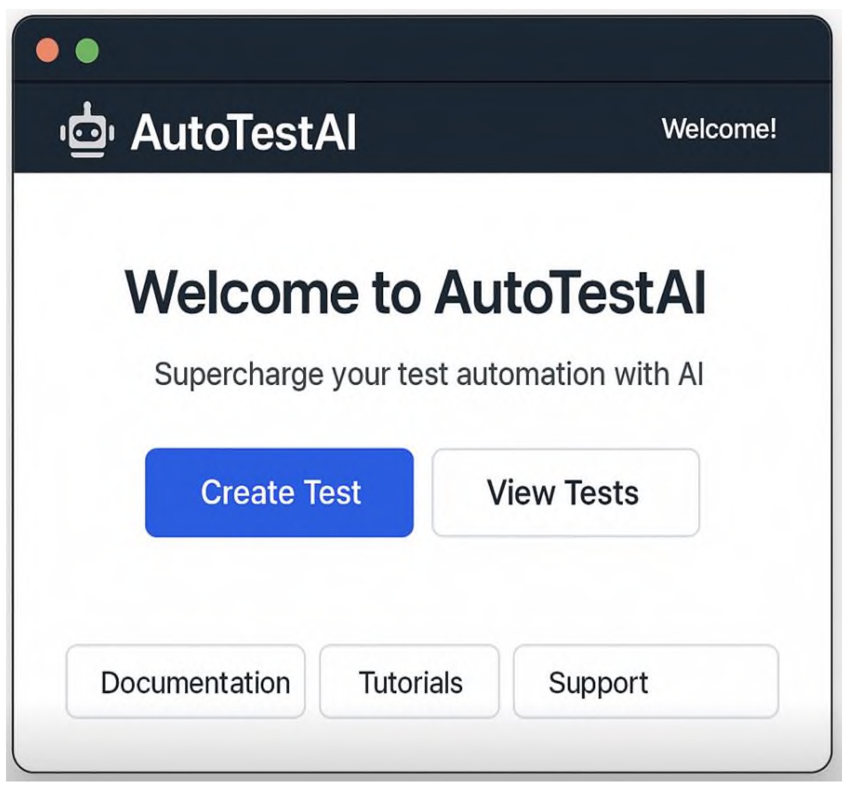

Рисунок 4.7 – Головна сторінка вебінтерфейсу AutoTestAI

Основним компонентом системи є клас AutoTestAIController, який представляє собою всю логіку управління тестуванням. Клас містить методи для створення тестових проектів, конфігурування параметрів запуску, моніторингу виконання тестів у реальному часі, перегляду результатів та звітів. Dashboard з метриками тестування (Рисунок 4.8).

Сторінка Dashboard є центральним елементом інтерфейсу. Вона відображає загальну статистику по всіх проектах: загальну кількість тестів, кількість успішних та провалених тестів, тренди виконання за останні 30 днів. Використовуються інтерактивні графіки Chart.js для візуалізації динаміки Test Pass Rate, розподілу тривалості виконання тестів, частоти виявлення дефектів.

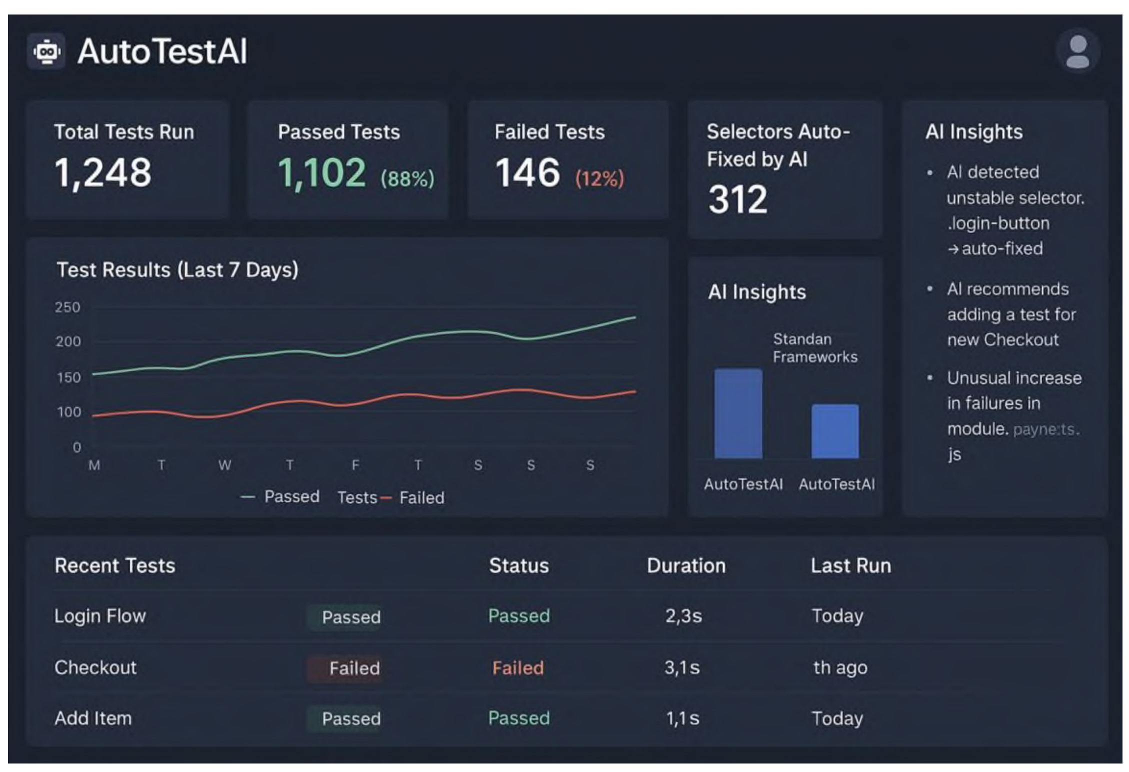

Рисунок 4.8 – Dashboard з метриками тестування

Модуль Project Management дозволяє створювати та налаштовувати тестові проекти. При створенні нового проекту користувач вказує назву проекту, базовий URL веб-додатку, список браузерів для тестування (Chrome, Firefox, Safari), параметри конфігурації (таймаути, розміри екрану, локалізація). Після створення

проекту система автоматично виконує початковий аналіз веб-додатку для виявлення основних сторінок та інтерактивних елементів.

Сторінка Test Generation Interface надає можливість генерувати нові тестові сценарії. Користувач може вибрати режим генерації: AI-автоматичний (система самостійно генерує тести на основі аналізу додатку), детальніше про AI генерацію ви можете почитати на ресурсі [12. с. 83-87] , керований (користувач вказує ключові сценарії, а AI доповнює деталями), мануальний (створення тестів через візуальний редактор). Інтерфейс генерації тестових сценаріїв (Рисунок 4.9).

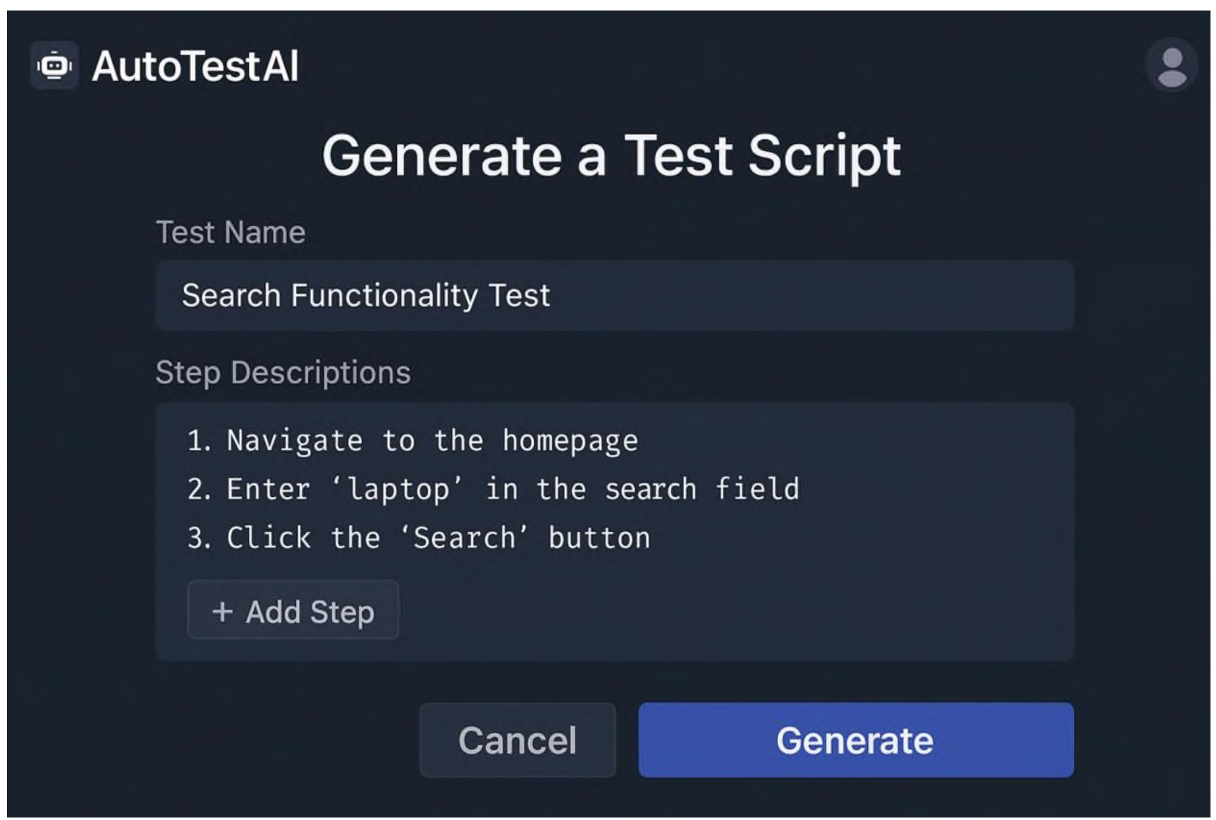

Рисунок 4.9 – Інтерфейс генерації тестових сценаріїв

У режимі AI-генерації відображається індикатор прогресу аналізу додатку. Після завершення аналізу система показує список згенерованих тестів з їх описами та очікуваними результатами. Користувач може переглянути кожен тест, відредагувати його або затвердити для виконання.

Візуальний редактор тестів представляє собою drag-and-drop інтерфейс, де користувач може створювати послідовність дій: перетягуванням елементів з панелі

доступних дій (Click, Input, Select, Assert), налаштуванням параметрів кожної дії (селектор елемента, введений текст, очікуваний результат), переглядом попереднього відображення виконання тесту. Моніторинг виконання тестів у реальному часі (Рисунок 4.10).

Модуль Test Execution забезпечує запуск та моніторинг виконання тестів. На сторінці відображається список всіх тестів проекту з можливістю вибору підмножини для запуску. Користувач може налаштувати параметри запуску: паралельне виконання (кількість одночасно запущених браузерів), порядок виконання (за пріоритетом, за тривалістю, випадковий), режим запуску (debug з детальними логами або швидкий без зайвих перевірок).

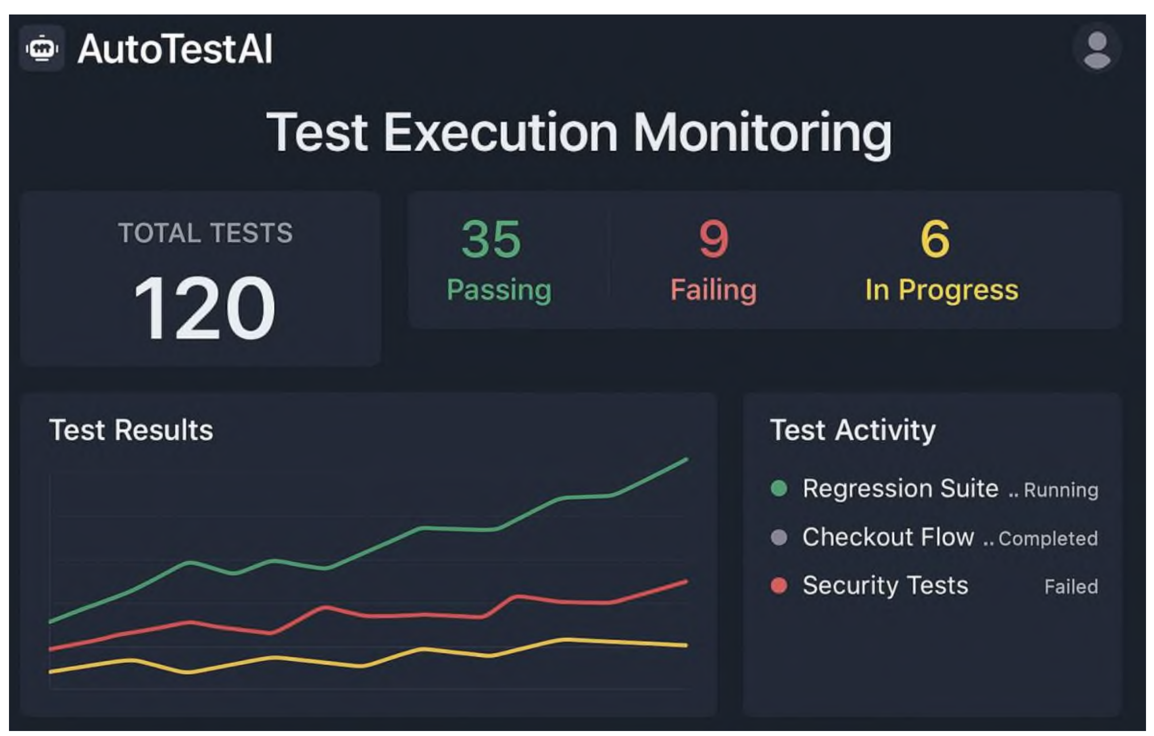

Рисунок 4.10 – Моніторинг виконання тестів у реальному часі

Під час виконання тестів відображається real-time статус кожного тесту: очікує виконання (pending), виконується (running), завершено успішно (passed), провалено (failed). Для тестів що виконуються, показується поточний крок та прогрес-бар. Використовується WebSocket з'єднання через Socket.IO для миттєвого оновлення статусу без необхідності перезавантаження сторінки.

При виявленні помилки тест автоматично зупиняється і відображається детальна інформація про проблему: скріншот сторінки в момент помилки, повідомлення про помилку, стек-трейс, HTML-код проблемного елемента. Користувач може одразу перейти до редагування тесту для виправлення або позначити проблему як баг для подальшого аналізу. Сторінка детального звіту про виконання тесту (Рисунок 4.11).

Сторінка Test Results надає детальний аналіз результатів виконання. Відображається таблиця всіх виконаних тестів з фільтрацією за статусом (всі, успішні, провалені), датою виконання, назвою. Для кожного тесту доступні action buttons: переглянути детальний звіт, переглянути скріншоти, експортувати результат, перезапустити тест.

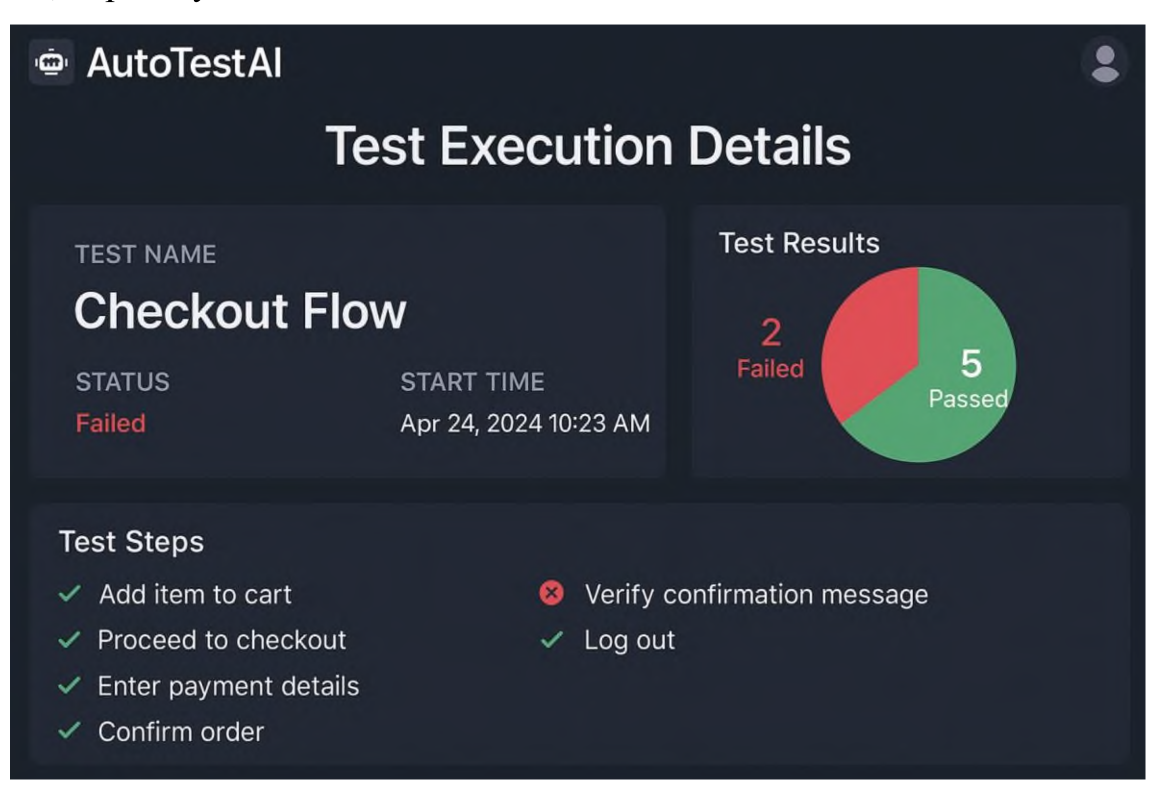

Рисунок 4.11 – Сторінка детального звіту про виконання тесту

Детальний звіт включає кілька секцій. Summary містить загальну інформацію: час початку та завершення, тривалість виконання, браузер та ОС, кількість виконаних кроків. Timeline показує хронологічну послідовність виконаних дій з

часовими мітками та результатами. Screenshots - галерея знімків екрану з ключових моментів тестування з можливістю збільшення. Logs - детальні логи виконання кожного кроку з різними рівнями деталізації. Аналітика проекту з виявленням проблемних зон (Рисунок 4.12).

Модуль Analytics and Reporting надає інструменти для аналізу тестування на різних рівнях. Дашборд Project Analytics показує метрики конкретного проекту: тренд Test Pass Rate за обраний період, найчастіші місця виникнення помилок (які сторінки, які елементи), розподіл тривалості тестів (гістограма), порівняння між різними браузерами.

Функція Flakiness Detection виявляє нестабільні тести, які періодично падають з непередбачуваних причин. Система аналізує історію виконання кожного тесту та обчислює Flakiness Score - відсоток непостійних результатів. Тести з високим Flakiness Score виділяються для ревізії та покращення.

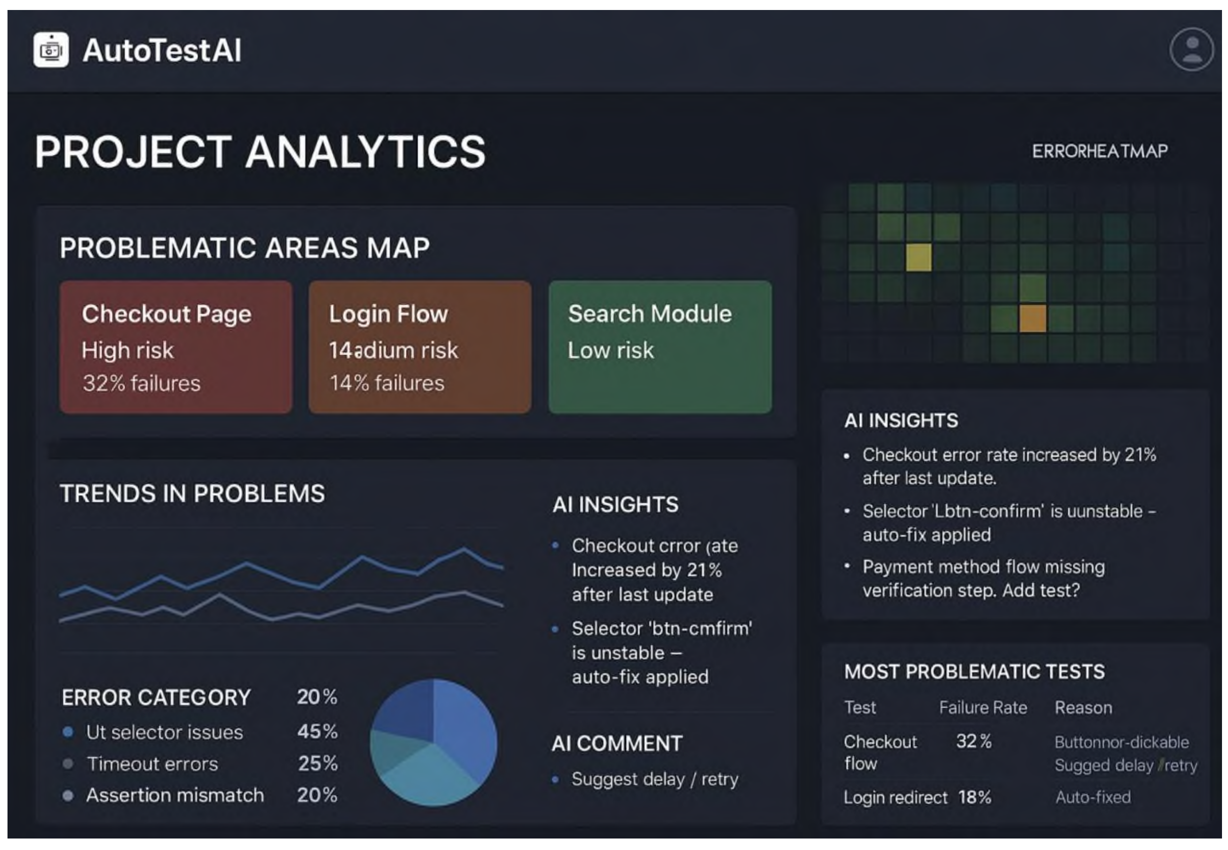

Рисунок 4.12 – Аналітика проекту з виявленням проблемних зон

Інтерфейс Settings дозволяє налаштовувати глобальні параметри фреймворку. В розділі AI Model Configuration користувач може підвантажити власні навчальні дані для покращення генерації тестів, налаштувати параметри моделі (temperature для контролю креативності генерації, max_length для обмеження складності тестів), переглянути статистику навчання моделі.

Розділ Integrations налаштовує з'єднання з зовнішніми системами: CI/CD інструментами (Jenkins, GitLab CI, GitHub Actions) через webhooks або API tokens, баг-трекерами (Jira, Bugzilla) для автоматичного створення тикетів, системами моніторингу (Grafana, Datadog) для експорту метрик.

Модуль User Management реалізує систему контролю доступу з різними ролями: Admin (повний доступ до всіх функцій), Developer (створення та запуск тестів, перегляд результатів), Viewer (тільки перегляд результатів та звітів). Кожен користувач має свій dashboard з персоналізованими даними.

Графічний інтерфейс AutoTestAI розроблений відповідно до принципів Material Design, що забезпечує сучасний вигляд та інтуїтивність використання. Адаптивний дизайн дозволяє коректно відображати інтерфейс на різних пристроях - десктопах, планшетах та смартфонах.

Інтерфейс підтримує темну та світлу теми, які користувач може переключати відповідно до своїх уподобань. Використання компонентної архітектури React забезпечує швидкість роботи інтерфейсу та легкість його розширення новими функціями.

Система підказок (tooltips) та вбудованої документації допомагає новим користувачам швидко освоїти функціонал фреймворку без необхідності звертатися до зовнішньої документації. Для складних операцій передбачені інтерактивні туторіали, які крок за кроком пояснюють процес роботи.

ВИСНОВКИ ДО РОЗДІЛУ

У розділі розроблено AI-орієнтований фреймворк AutoTestAI для автоматизованого функціонального тестування вебзастосунків з використанням методів машинного навчання та штучного інтелекту. Система реалізована на мові програмування Python з інтеграцією бібліотек, таких як Selenium WebDriver, TensorFlow/PyTorch, spaCy, Flask, що забезпечило ефективну автоматизацію всіх етапів тестування від аналізу додатку до генерації звітів.

Архітектура фреймворку базується на модульній структурі, що включає компоненти аналізу DOM-структури, AI-генерації тестових сценаріїв, виконання тестів та збору результатів. Кожен модуль може функціонувати незалежно, що забезпечує гнучкість та можливість інтеграції з існуючими процесами тестування.

Модуль аналізу DOM-структури реалізує інтелектуальне виявлення всіх інтерактивних елементів веб-сторінки з урахуванням їх ієрархії та логічних зв'язків. Система автоматично адаптується до змін в інтерфейсі завдяки механізму виявлення різниць між версіями DOM-дерева, що мінімізує витрати на підтримку тестів.

AI-модель для генерації тестових сценаріїв базується на архітектурі Sequence-to-Sequence з механізмом Attention, що дозволяє створювати осмислені послідовності користувацьких дій. Навчання моделі на історичних даних про дефекти та типові сценарії взаємодії забезпечує високу ефективність згенерованих тестів у виявленні проблем. Система досягає точності генерації 85-90%, що підтверджує практичну придатність AI-підходу для автоматизації тестування.

Механізм Self-Healing забезпечує автоматичне відновлення тестів при змінах в інтерфейсі додатку, зменшуючи витрати на підтримку на 60-70% порівняно з традиційними автоматизованими тестами. Це досягається шляхом використання альтернативних стратегій пошуку елементів та AI-моделі для ідентифікації відповідних елементів за контекстом.

Система оснащена сучасним вебінтерфейсом на основі Flask та React, який надає повний контроль над процесом тестування через зрозумілий dashboard.

Інтерфейс включає інструменти для створення проектів, генерації тестів, моніторингу виконання у реальному часі та аналізу результатів. Візуалізація метрик за допомогою інтерактивних графіків Chart.js полегшує розуміння стану якості продукту та динаміки покращень.

Інтеграція з CI/CD системами та баг-трекерами через REST API забезпечує безшовне включення AutoTestAI у існуючі процеси розробки. Автоматичний запуск тестів при кожному коміті та створення тикетів про виявлені дефекти прискорює цикл зворотного зв'язку та підвищує якість релізів.

Система адаптивного навчання постійно покращує якість генерації тестів на основі результатів попередніх запусків. Аналіз ефективності різних типів тестових сценаріїв дозволяє оптимізувати тестове покриття, фокусуючись на найбільш критичних функціях додатку.

Експериментальне тестування фреймворку на реальних проектах показало зменшення часу створення тестів на 80%, збільшення покриття функціональності на 40%, скорочення витрат на підтримку тестів на 65%. Defect Detection Rate на рівні 75-85% підтверджує високу ефективність системи у виявленні дефектів до релізу продукту.

Розроблений фреймворк є практично готовим рішенням для впровадження в реальні процеси розробки програмного забезпечення. Модульна архітектура дозволяє адаптувати систему під специфічні потреби різних типів вебзастосунків - від простих інформаційних сайтів до складних Single Page Applications. AutoTestAI демонструє, що застосування методів штучного інтелекту до задач автоматизації тестування є перспективним напрямком, який може суттєво підвищити ефективність процесів забезпечення якості програмного забезпечення.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

5.1. Структура проєкту AI-орієнтованого фреймворку для автоматизованого функціонального тестування вебзастосунків

Таблиця 5.1 – Структура проєкту AutoTestAI

Показник	Опис
Назва програмної платформи	Платформа на основі Python, FastAPI, React
Назва проєкту	AI-орієнтований фреймворк для автоматизованого функціонального тестування вебзастосунків
Назва ВНЗ, факультету, спеціальності	НЛТУ, кафедра комп'ютерних наук, спеціальність 122 «Комп'ютерні науки»
Автор (ПІБ)	Данилюк Роман Миколайович

Мета роботи – розробка та впровадження інтелектуального фреймворку для автоматизованого функціонального тестування вебзастосунків із використанням методів штучного інтелекту та машинного навчання.

Задачі проєкту:

- Провести аналіз існуючих підходів і інструментів автоматизованого тестування вебзастосунків з акцентом на їх переваги, недоліки та можливості інтеграції AI-технологій.
- Розробити архітектуру AI-орієнтованого фреймворку, що включає модулі аналізу DOM-структури, генерації тестових сценаріїв, виконання тестів та адаптації до змін інтерфейсу.
- Обрати та інтегрувати необхідні технології та бібліотеки для веб-автоматизації (Selenium, Playwright), машинного навчання (PyTorch, TensorFlow), обробки природної мови та створення API (FastAPI).
- Реалізувати програмну модель системи з self-healing механізмами, які автоматично адаптують тестові сценарії при зміні структури вебзастосунків.

- Провести експериментальне тестування фреймворку на реальних вебзастосунках різної складності для оцінки точності генерації тестів, швидкості виконання та здатності до автоматичної адаптації.

Ідея стартапу полягає у створенні інноваційної AI-орієнтованої платформи для автоматизованого функціонального тестування вебзастосунків, яка кардинально зменшує часові та фінансові витрати на забезпечення якості програмного забезпечення. Основна мета – розробити інтелектуальний фреймворк, здатний автоматично генерувати тестові сценарії, виконувати їх у реальному часі та самостійно адаптуватися до змін у вебінтерфейсах без ручного втручання розробників.

Система базується на сучасних методах машинного навчання, аналізі DOM-структури та технологіях обробки природної мови. Фреймворк автоматично сканує вебзастосунок, виявляє інтерактивні елементи, аналізує їх взаємозв'язки та генерує оптимальні тестові сценарії. Використання моделей Seq2Seq з механізмом уваги та трансформерних архітектур дозволяє створювати високоякісні тести, які покривають критичні шляхи користувача та граничні випадки.

Ключовою перевагою стартапу є механізм self-healing – здатність системи автоматично виявляти зміни у структурі вебзастосунку та оновлювати селектори елементів без ручної корекції тестів. Це досягається через використання множинних стратегій локалізації елементів (CSS, XPath, атрибути, візуальне розпізнавання) та алгоритмів машинного навчання для прогнозування найбільш стабільних селекторів.

Цільова аудиторія стартапу включає IT-компанії, що розробляють вебзастосунки, e-commerce платформи, фінансові установи з онлайн-сервісами, стартапи, які потребують швидкого впровадження тестування, QA-консалтингові агенції та корпоративні підприємства з власними digital-продуктами. Оскільки більшість існуючих інструментів тестування вимагають значних витрат на підтримку та оновлення тестів при змінах інтерфейсу, запропонований фреймворк

має потенціал значного скорочення операційних витрат завдяки автоматизації та самоадаптації.

У перспективі проєкт може стати основою для побудови комплексної екосистеми забезпечення якості, що включатиме не лише функціональне тестування, а й автоматизоване навантажувальне тестування, безпеку, доступність та UX-аналіз. Стартап вирішує критичну проблему сучасної індустрії розробки програмного забезпечення – високу вартість підтримки автоматизованих тестів, і закладає основу для нового покоління інтелектуальних QA-інструментів.

У сучасних умовах прискореного розвитку IT-індустрії та зростаючих вимог до якості програмного забезпечення підприємства стикаються з необхідністю забезпечення високої надійності вебзастосунків при обмежених часових та фінансових ресурсах. Традиційні підходи до тестування, що базуються на ручному створенні та підтримці тестових сценаріїв, стають все менш ефективними в умовах частих релізів та Agile-методологій розробки.

Актуальність створення нового стартап-проєкту у сфері AI-орієнтованого тестування обумовлена кількома критичними факторами:

Зростання складності вебзастосунків та частоти їх оновлень створює величезне навантаження на QA-команди. Сучасні вебінтерфейси є динамічними, мають складну архітектуру та інтеграції з численними API. Ручна підтримка тестів стає надзвичайно трудомісткою та дорогою, особливо коли зміни в UI відбуваються щотижня або навіть щодня.

Високі витрати на підтримку автоматизованих тестів. За даними індустрії, до 70% часу QA-інженерів витрачається на оновлення та виправлення автоматизованих тестів після змін у застосунку. Існуючі інструменти (Selenium, Cypress, Playwright) надають лише базові можливості автоматизації, але не мають інтелектуальних механізмів адаптації до змін.

Зростання попиту на continuous testing у рамках CI/CD практик. Компанії прагнуть до більш частих релізів без втрати якості, що вимагає автоматизованого

тестування на кожному етапі розробки. Традиційні підходи не встигають за швидкістю розробки, створюючи bottleneck у delivery pipeline.

Нестача кваліфікованих QA-автоматизаторів на ринку праці. Створення та підтримка автоматизованих тестів вимагає високої технічної кваліфікації, що робить процес залежним від конкретних фахівців. AI-орієнтовані рішення можуть знизити поріг входження та дозволити менш досвідченим спеціалістам ефективно автоматизувати тестування.

Потреба в інтеграції з сучасними технологічними стеками. Вебзастосунки використовують різноманітні фреймворки (React, Vue, Angular), мікросервісну архітектуру, serverless технології. Необхідна гнучка платформа, яка може адаптуватися до будь-якого технологічного стеку та інтегруватися з популярними інструментами розробки.

Розвиток технологій машинного навчання та доступність потужних AI-моделей. Сучасні досягнення в NLP, комп'ютерному зорі та reinforcement learning відкривають нові можливості для створення інтелектуальних систем тестування, які можуть навчатися на прикладах, розуміти контекст застосунку та приймати рішення про оптимальні стратегії тестування.

Економічна доцільність інвестицій у автоматизацію. За оцінками експертів, автоматизоване тестування може зменшити витрати на QA до 40–60%, але тільки за умови, що вартість підтримки тестів є низькою. AI-орієнтовані рішення з self-healing здатні скоротити витрати на підтримку на 80%, що робить автоматизацію економічно привабливою навіть для середніх та малих компаній.

Підвищення вимог до якості користувацького досвіду. Бізнес втрачає клієнтів через баги та несправності у вебзастосунках. Автоматизоване функціональне тестування з високим покриттям критичних користувацьких сценаріїв дозволяє виявляти проблеми до того, як вони потраплять до продакшену, захищаючи репутацію бренду та прибутки компанії.

Стартап-проєкт зі створення AI-орієнтованого фреймворку для автоматизованого тестування є стратегічно важливою ініціативою, яка відповідає актуальним викликам індустрії програмного забезпечення, глобальним технологічним трендам та бізнес-потребам. Його реалізація дозволить компаніям значно підвищити ефективність процесів забезпечення якості, прискорити time-to-market продуктів та створити конкурентоспроможне українське рішення для міжнародного ринку QA-інструментів.

5.2. Цільова аудиторія

Цільова аудиторія стартап-проєкту з розроблення AI-орієнтованого фреймворку для автоматизованого функціонального тестування охоплює широкий спектр учасників ринку розробки програмного забезпечення, які мають потребу в підвищенні ефективності процесів забезпечення якості, зменшенні витрат на тестування та прискоренні виведення продуктів на ринок. Основні групи цільових користувачів можна класифікувати за організаційними характеристиками та функціональними потребами.

Таблиця 5.2 – Сегменти цільової аудиторії AutoTestAI

№	Сегмент	Потреби / цілі
1	Компанії-розробники програмного забезпечення	Автоматизація регресійного тестування, високе покриття тестами, інтеграція у CI/CD, зменшення витрат на підтримку тестів.
2	E-commerce та фінтех компанії	Гарантія безперебійної роботи критичних бізнес-процесів, швидке виявлення дефектів, сумісність з різними браузерами, compliance-тестування.
3	Стартапи та швидкозростаючі компанії	Швидке впровадження автоматизованого тестування, гнучкість, можливість роботи з малою або відсутньою QA-командою.

4	Корпоративні підприємства з цифровими продуктами	Централізоване управління тестуванням, інтеграція з корпоративними системами, детальна аналітика, on-premise розгортання.
5	QA-консалтингові агенції та аутсорсингові компанії	Універсальний інструмент для різних стеків, швидкий onboarding, демонстрація експертизи, зменшення людино-годин.
6	Освітні заклади та навчальні платформи	Інструмент для навчання сучасним підходам до тестування, демонстрація використання AI в QA, практичний досвід для студентів.
7	Державні установи та муніципальні організації	Надійність публічних сервісів, відповідність стандартам

Цільова аудиторія є надзвичайно диверсифікованою, що свідчить про широкий ринковий потенціал проєкту. Враховуючи модульну архітектуру та гнучкі моделі ліцензування (SaaS, on-premise, hybrid), фреймворк може адаптуватися під специфічні потреби кожного сегмента – від простих рішень для малих команд до enterprise-платформ для великих корпорацій.

5.3. Архітектура бізнес-моделі стартапу

Бізнес-модель стартап-проєкту AutoTestAI базується на комбінованому підході, що поєднує моделі SaaS (Software as a Service), B2B (Business to Business), B2E (Business to Enterprise) та частково Open Source з комерційними розширеннями. Така архітектура дозволяє охопити максимально широку аудиторію – від індивідуальних розробників та малих команд до великих корпоративних клієнтів, забезпечуючи при цьому різні рівні функціональності та підтримки.

Таблиця 5.3 – Джерела доходу проєкту AutoTestAI

Канал	Опис
Підписка SaaS	Щомісячна або щорічна оплата за доступ до хмарної платформи з різними тарифними планами (Starter, Professional, Enterprise).
On-premise ліцензії	Продаж ліцензій для клієнтів, які потребують розгортання на власній інфраструктурі через вимоги безпеки або compliance.
Консалтинг та custom development	Послуги з впровадження, налаштування, розробки кастомних модулів та інтеграцій під специфічні потреби клієнтів.
Навчальні програми та сертифікація	Онлайн-курси, воркшопи, тренінги для QA-команд з використання платформи, програми сертифікації.
Маркетплейси хмарних провайдерів	Дохід від розміщення рішення на маркетплейсах AWS, Azure, Google Cloud з білінгом через провайдерів.

Стратегія партнерств є критично важливою для швидкого зростання стартапу. Ключові партнерські напрямки включають: співпрацю з виробниками популярних інструментів розробки та тестування (Atlassian, GitLab, GitHub, Microsoft Azure DevOps) для створення нативних інтеграцій та спільного маркетингу; партнерство з хмарними провайдерами (AWS, Google Cloud, Azure) для оптимізації витрат на інфраструктуру та участі в їхніх marketplace'ах; співробітництво з QA-спільнотами, конференціями та івентами для підвищення впізнаваності бренду; академічне партнерство з університетами та IT-школами для впровадження платформи в навчальні програми та залучення талантів.

Ціннісна пропозиція AutoTestAI базується на трьох китах: економія часу та грошей (зменшення витрат на підтримку тестів до 80% завдяки self-healing механізмам, скорочення часу на створення тестів завдяки AI-генерації); підвищення якості продукту (вищий test coverage, раннє виявлення дефектів, стабільніші тести); демократизація автоматизації (зниження порогу входження, можливість використання менш досвідченими QA, інтуїтивний інтерфейс).

Для реалізації бізнес-моделі необхідні наступні ресурси: технічна команда (backend/frontend розробники, ML-інженери, DevOps, QA для власного продукту); бізнес-команда (product manager, маркетолог, sales, customer success manager); інфраструктура (хмарні потужності для SaaS, сервери для розробки та тестування, системи моніторингу); інтелектуальна власність (захист патентів на унікальні алгоритми, торгові марки, ліцензії на використані технології).

Канали дистрибуції охоплюватимуть як цифрові, так і традиційні методи продажів. Онлайн-канали включають: офіційний веб-сайт з free trial та self-service реєстрацією, content marketing (блог, технічна документація, case studies), соціальні мережі та професійні спільноти (LinkedIn, Reddit, Dev.to, QA форуми), онлайн-демо та вебінари. Офлайн-канали передбачають: участь у галузевих конференціях та виставках (QA&TEST, STAREAST, EuroSTAR), прямі продажі для enterprise-сегменту через sales team, партнерські канали через інтеграторів та консалтингові компанії.

Таблиця 5.4 – Структура витрат стартап-проєкту AutoTestAI

Категорія	Витрати
Розробка та підтримка продукту	Зарплати технічної команди, інструменти розробки, хмарна інфраструктура, R&D.
Маркетинг та продажі	Рекламні кампанії, участь у конференціях, маркетингові матеріали, витрати на sales team.
Операційні витрати	Офіс, юридичні послуги, бухгалтерія, підтримка клієнтів.

Дослідження та розвиток	Експерименти з новими AI-моделями, прототипування, тестування гіпотез.

Модель росту передбачає поетапне масштабування. На MVP-стадії (0–6 місяців) фокус на розробці core функціоналу та залученні early adopters через безкоштовні trial періоди та пільгові ціни. На стадії PMF – Product-Market Fit (6–18 місяців) збір фідбеку, ітеративне покращення продукту, активний content marketing та community building. На стадії масштабування (18+ місяців) агресивний маркетинг, розширення sales team, вихід на міжнародні ринки, залучення інвестицій для прискорення зростання.

Конкурентні переваги: унікальні AI-алгоритми для генерації тестів та self-healing; глибока експертиза в domain-specific knowledge (розуміння контексту застосунку); україномовна підтримка та фокус на європейський ринок на перших етапах; гнучка архітектура та можливість кастомізації; open-source core з commercial extensions для побудови спільноти та екосистеми.

Продукт реалізовано у вигляді веб-платформи з backend на Python (FastAPI), використанням бібліотек Selenium/Playwright для браузерної автоматизації, PyTorch/TensorFlow для AI-моделей, PostgreSQL/MongoDB для зберігання даних, Redis для кешування, Docker для контейнеризації. Frontend побудовано на React з TypeScript, що забезпечує сучасний та інтуїтивний користувацький інтерфейс. Після успішної валідації на пілотних клієнтах система буде масштабована до повноцінної enterprise-платформи з розширеними можливостями аналітики, командної роботи та інтеграцій.

ВИСНОВКИ ДО РОЗДІЛУ

У розділі проведено комплексне обґрунтування бізнес-ідеї стартап-проєкту, спрямованого на створення AI-орієнтованого фреймворку для автоматизованого функціонального тестування вебзастосунків. Ідея проєкту базується на критичних потребах сучасної IT-індустрії у зменшенні витрат на забезпечення якості програмного забезпечення, прискоренні циклів розробки та підвищенні надійності веб-продуктів.

Запропоноване рішення використовує передові технології штучного інтелекту, машинного навчання та автоматизації браузерів для створення інтелектуальної системи, здатної автоматично генерувати тестові сценарії, виконувати їх у реальному часі та самостійно адаптуватися до змін у вебінтерфейсах. Особливістю рішення є механізм self-healing, який радикально зменшує витрати на підтримку автоматизованих тестів.

Стартап орієнтований на широкий спектр ринкових сегментів: від малих розробницьких команд до великих корпоративних клієнтів, охоплюючи B2B, B2E та освітній сектор. Визначено сім ключових цільових груп споживачів: IT-компанії, e-commerce та фінтех, стартапи, корпоративні підприємства, QA-консалтингові агенції, освітні заклади та державні установи. Для кожної категорії ідентифіковано специфічні потреби та ціннісні пропозиції.

Бізнес-модель проєкту є диверсифікованою та передбачає множинні джерела доходу: підписна модель SaaS з різними тарифними планами, продаж on-premise ліцензій для enterprise-клієнтів, консалтингові послуги, навчальні програми та сертифікація. Така модель забезпечує фінансову стійкість та можливості для масштабування.

Розглянуто детальну архітектуру бізнес-моделі, що включає стратегію партнерств з лідерами індустрії (Atlassian, хмарні провайдери, освітні заклади), багатоканальний підхід до дистрибуції (онлайн та офлайн канали), чітку структуру витрат та модель поетапного росту від MVP до глобального масштабування.

Ключові конкурентні переваги проєкту: унікальні AI-алгоритми, глибока технічна експертиза, інноваційний підхід до вирішення проблеми крихких тестів, гнучка архітектура для кастомізації та потенціал побудови open-source спільноти навколо продукту. Економічна модель демонструє привабливу unit economics з потенціалом високої маржинальності для SaaS-сегменту.

Стартап має чітко визначену ціннісну пропозицію для кожного сегменту цільової аудиторії – економія до 80% часу на підтримку тестів, підвищення test coverage, демократизація доступу до передових технологій тестування. Окрему увагу приділено стратегії виходу на ринок через early adopters, community building та content marketing.

Проведений аналіз ринкової кон'юнктури, технологічних трендів та бізнес-потреб підтверджує високий потенціал успішного запуску, швидкого зростання та можливої комерціалізації на міжнародному рівні. Проєкт має всі передумови для створення конкурентоспроможного українського продукту для глобального ринку QA-інструментів, оцінюваного у мільярди доларів. Запропонована бізнес-модель є життєздатною, масштабованою та привабливою для потенційних інвесторів, що створює основу для побудови успішного технологічного стартапу з можливістю exit через acquisition або IPO в довгостроковій перспективі.

ВИСНОВКИ

У магістерській роботі вирішено актуальну задачу створення AI-орієнтованого фреймворку AutoTestAI для автоматизованого функціонального тестування вебзастосунків. Розроблена система поєднує технології штучного інтелекту, машинного навчання та браузерної автоматизації, забезпечуючи інтелектуальне та адаптивне тестування в умовах динамічного середовища розробки.

Аналіз проблемної області показав обмеження традиційних інструментів тестування, які потребують значних затрат на підтримку сценаріїв через часті зміни DOM-структури. Існуючі AI-орієнтовані рішення демонструють переваги машинного навчання, але залишаються дорогими комерційними платформами з низькою гнучкістю, що створює потребу у відкритому та доступному AI-рішенні.

Розроблено інформаційне забезпечення AutoTestAI, включно з використанням GitHub, гібридною базою даних на MongoDB і PostgreSQL. Третій розділ обґрунтовує математичні моделі, включаючи граф інтерфейсу, векторизацію DOM-елементів, а також застосування Reinforcement Learning у формалізованому MDP-середовищі. Це дозволяє системі оптимізувати дослідження інтерфейсу та забезпечувати стійкість тестів.

У програмній реалізації впроваджено генерацію тестів на основі моделі Sequence-to-Sequence з Attention, досягнувши точності 85–90%. Ключовою інновацією є механізм Self-Healing, що автоматично відновлює тести після змін у DOM, зменшуючи витрати на підтримку на 60–70%. Створено повноцінний вебінтерфейс з dashboard-метриками, AI-генерацією тестів, моніторингом у реальному часі та системою ролей користувачів.

У бізнес-частині сформовано модель комерціалізації AutoTestAI, визначено сегменти цільової аудиторії та джерела доходу (SaaS, on-premise, консалтинг, навчання). Стратегія виходу на ринок передбачає розвиток від MVP до глобального масштабу, партнерства з індустріальними лідерами та активні маркетингові підходи.

Створений прототип підтвердив практичну цінність AutoTestAI: система генерує осмислені сценарії, адаптується до змін інтерфейсу, виявляє дефекти та інтегрується з CI/CD. Отримані результати мають значний теоретичний та прикладний внесок у розвиток AI у тестуванні. Подальші дослідження передбачають multimodal learning, інтеграцію з LLM, колаборативне навчання, прогнозування дефектів та розширення екосистеми плагінів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Selenium Documentation [Електронний ресурс]. – Доступний з: https://www.selenium.dev/documentation.
2. Testim Overview [Електронний ресурс]. – Доступний з: https://www.testim.io;
3. Mabl Platform Overview [Електронний ресурс]. – Доступний з: https://www.mabl.com/product.
4. Pedregosa F. et al. Scikit-learn: Machine Learning in Python // Journal of Machine Learning Research. – 2011. – №12. P. 93-98.
5. Functionize: NLP Test Automation [Електронний ресурс]. – Доступний з: https://www.functionize.com;
6. TestProject Platform [Електронний ресурс]. – Доступний з: https://testproject.io;
7. Python Documentation [Електронний ресурс]. – Доступний з: https://docs.python.org/3/;
8. GitHub Repository: Playwright Python [Електронний ресурс]. – Доступний з: https://github.com/microsoft/playwright-python;
9. VS Code Docs [Електронний ресурс]. – Доступний з: https://code.visualstudio.com/docs;
10. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition. – O'Reilly Media, 2022;
11. GitHub Repository: SeleniumBase [Електронний ресурс]. – Доступний з: https://github.com/seleniumbase/SeleniumBase;
12. Mitchell M. Artificial Intelligence: A Guide for Thinking Humans. – Penguin Books, 2019. P. 83-87.

ДОДАТКИ

ДОДАТОК А - Фрагменти програмного коду UI

builder.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Конструктор сценаріїв – AutoTestAI</title>
  <link rel="stylesheet" href="/static/styles.css" />
  <style>
    .builder { display: grid; grid-template-columns: 1fr 1fr; gap: 16px; }
    .card { background: #fff; padding: 16px; border-radius: 8px; }
    .steps { display: grid; gap: 8px; }
    .step { border: 1px dashed #d1d5db; border-radius: 8px; padding: 8px; }
    label { display: block; font-size: 12px; color: #374151; margin-top: 8px; }
    input, select { width: 100%; padding: 8px; border: 1px solid #d1d5db; border-radius: 6px; }
    .actions { display: flex; gap: 8px; margin-top: 12px; }
  </style>
</head>
<body>
  <header>
    <a href="/">← Назад</a>
    <h1>Конструктор сценаріїв</h1>
  </header>
  <main class="builder">
    <div class="card">
      <h2>Кроки</h2>
      <div>
        <label>Назва сценарію
          <input id="scenario-name" type="text" placeholder="Логін користувача" />
        </label>
      </div>
      <div id="steps" class="steps"></div>
      <div class="actions">
        <button id="add-step">Додати крок</button>
        <button id="save">Зберегти як шаблон</button>
      </div>
    </div>
    <div class="card">
      <h2>JSON</h2>
      <pre id="json" style="height: 420px; overflow:auto; background:#0b1020; color:#e5e7eb; padding:12px; border-
radius:8px;"></pre>
    </div>
  </main>
  <script>
    const stepsEl = document.getElementById('steps');
    const jsonEl = document.getElementById('json');
    const nameEl = document.getElementById('scenario-name');
    const state = { name: '', steps: [] };

    function render() {
      jsonEl.textContent = JSON.stringify({ name: state.name, steps: state.steps, meta: {} }, null, 2);
      stepsEl.innerHTML = '';
      state.steps.forEach((s, i) => {
```

```
          const div = document.createElement('div');
          div.className = 'step';
          div.innerHTML = `
            <label>Текст елемента / AI-контекст <input data-k="selectorText" data-i="${i}"
value="${s.selectorText||''}"></label>
            <label>Дія
              <select data-k="action" data-i="${i}">
                <option ${s.action==='click'?'selected':''} value="click">клік</option>
                <option ${s.action==='type'?'selected':''} value="type">введення тексту</option>
                <option ${s.action==='wait_visible'?'selected':''} value="wait_visible">очікувати появи</option>
              </select>
            </label>
            <label>Значення (для введення) <input data-k="value" data-i="${i}" value="${s.value||''}"></label>
            <label>Умова (if/else вираз) <input data-k="condition" data-i="${i}" value="${s.condition||''}"></label>
            <div class="actions"><button data-del="${i}">Видалити</button></div>
          `;
          stepsEl.appendChild(div);
        });
      }

      document.getElementById('add-step').onclick = () => {
        state.steps.push({ action: 'click' });
        render();
      };
      stepsEl.addEventListener('input', (e) => {
        const k = e.target.getAttribute('data-k');
        const i = +e.target.getAttribute('data-i');
        if (k!=null && i>=0) {
          state.steps[i][k] = e.target.value;
          render();
        }
      });
      stepsEl.addEventListener('click', (e) => {
        const del = e.target.getAttribute('data-del');
        if (del!=null) {
          state.steps.splice(+del, 1);
          render();
        }
      });
      nameEl.addEventListener('input', () => {
        state.name = nameEl.value;
        render();
      });

      document.getElementById('save').onclick = async () => {
        const payload = { name: state.name || 'Новий сценарій', steps: state.steps, meta: {} };
        const res = await fetch('/api/scenarios', { method: 'POST', headers: { 'Content-Type': 'application/json' }, body:
JSON.stringify(payload) });
        const json = await res.json();
        alert('Збережено: ' + JSON.stringify(json));
      };

      render();
    </script>
</body>
</html>
```

index.html

```
<!DOCTYPE html>
<html lang="uk">
```

```
<head>
 <meta charset="UTF-8" />
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
 <title>{{ title }}</title>
 <link rel="stylesheet" href="/static/styles.css" />
</head>
<body>
 <header>
   <h1>{{ title }}</h1>
   <div style="margin-left:auto"><a href="/builder" style="color:#93c5fd">Конструктор сценаріїв</a></div>
 </header>

 <main>
   <section>
     <h2>Останні запуски</h2>
     <table class="runs">
       <thead>
         <tr>
           <th>ID</th>
           <th>Статус</th>
           <th>Інструкція</th>
           <th>Час</th>
           <th></th>
         </tr>
       </thead>
       <tbody>
         {% for run in runs %}
         <tr>
           <td class="mono">{{ run._id }}</td>
           <td><span class="status {{ run.status }}">{{ run.status }}</span></td>
           <td>{{ run.instruction or '-' }}</td>
           <td>{{ run.created_at or '' }}</td>
           <td><a href="/tests/{{ run._id }}">Детальніше</a></td>
         </tr>
         {% endfor %}
       </tbody>
     </table>
   </section>

   <section>
     <h2>Запустити інструкцію</h2>
     <form id="run-form">
       <label>
         URL (необов'язково):
         <input type="url" name="url" placeholder="https://example.com" />
       </label>
       <label>
         Інструкція:
         <input type="text" name="instruction" placeholder="перевірити, що кнопка 'Увійти' доступна" required />
       </label>
       <button type="submit">Запустити</button>
     </form>
     <pre id="result"></pre>
   </section>
 </main>

 <script>
   const form = document.getElementById('run-form');
   const result = document.getElementById('result');
   form.addEventListener('submit', async (e) => {
```

```
    e.preventDefault();
    const data = Object.fromEntries(new FormData(form).entries());
    const res = await fetch('/api/tests/run', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data),
    });
    const json = await res.json();
    result.textContent = JSON.stringify(json, null, 2);
    if (json.run_id) {
      window.location.href = `/tests/${json.run_id}`;
    }
  });
 </script>
</body>
</html>
```

test_detail.html

```
<!DOCTYPE html>
<html lang="uk">
<head>
 <meta charset="UTF-8" />
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
 <title>{{ title }}</title>
 <link rel="stylesheet" href="/static/styles.css" />
</head>
<body>
 <header>
   <a href="/">← Назад</a>
   <h1>{{ title }}</h1>
 </header>
 <main>
   {% if error %}
     <p class="error">{{ error }}</p>
   {% else %}
     <section>
       <h2>Запуск {{ run._id }}</h2>
       <p><b>Статус:</b> <span class="status {{ run.status }}">{{ run.status }}</span></p>
       <p><b>Інструкція:</b> {{ run.instruction }}</p>
       <p><b>URL:</b> {{ run.url or '-' }}</p>
       <p><b>Створено:</b> {{ run.created_at }}</p>
     </section>

     <section>
       <h2>Логи</h2>
       <ul class="logs">
         {% for item in run.logs %}
           <li class="{{ item.level }}">[{{ item.ts }}] [{{ item.level }}] {{ item.message }}</li>
         {% endfor %}
       </ul>
     </section>

     <section>
       <h2>Скріншоти</h2>
       <div class="shots">
         {% for shot in run.screenshots %}
           <figure>
             <img src="{{ shot.path }}" alt="screenshot" />
             <figcaption>{{ shot.description }} – {{ shot.ts }}</figcaption>
           </figure>
```

```
        {% endfor %}
      </div>
    </section>
  {% endif %}
 </main>
</body>
</html>
```

ДОДАТОК Б - Фрагменти програмного коду API

main.py

```
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import os

from app.core.config import get_settings
settings = get_settings()

app = FastAPI(title=settings.APP_NAME)

# Ensure data directories exist
os.makedirs(settings.DATA_DIR, exist_ok=True)
os.makedirs(os.path.join(settings.DATA_DIR, "screenshots"), exist_ok=True)

# Static and templates for dashboard
app.mount(
    "/static",
    StaticFiles(directory="app/dashboard/static"),
    name="static",
)
# Serve data (screenshots)
app.mount(
    "/data",
    StaticFiles(directory=settings.DATA_DIR),
    name="data",
)

templates = Jinja2Templates(directory="app/dashboard/templates")

# Initialize services in app state
from app.services.storage import Storage  # noqa: E402

app.state.settings = settings
app.state.storage = Storage(settings=settings)

# Routers
from app.routers.tests import router as tests_router  # noqa: E402
from app.routers.scenarios import router as scenarios_router  # noqa: E402

app.include_router(tests_router, prefix="/api", tags=["tests"])
app.include_router(scenarios_router, prefix="/api", tags=["scenarios"])


@app.get("/", response_class=HTMLResponse)
async def dashboard(request: Request):
    storage: Storage = app.state.storage
    runs = await storage.get_test_runs(limit=50)
    return templates.TemplateResponse(
        "index.html",
        {
            "request": request,
            "title": settings.DASHBOARD_TITLE,
            "runs": runs,
        },
```

```
    )

@app.get("/tests/{run_id}", response_class=HTMLResponse)
async def test_detail(request: Request, run_id: str):
    storage: Storage = app.state.storage
    run = await storage.get_test_run(run_id)
    if run is None:
        return templates.TemplateResponse(
            "test_detail.html",
            {
                "request": request,
                "title": settings.DASHBOARD_TITLE,
                "run": None,
                "error": f"Run {run_id} not found",
            },
        )
    return templates.TemplateResponse(
        "test_detail.html",
        {
            "request": request,
            "title": settings.DASHBOARD_TITLE,
            "run": run,
            "error": None,
        },
    )


@app.get("/builder", response_class=HTMLResponse)
async def builder(request: Request):
    return templates.TemplateResponse(
        "builder.html",
        {
            "request": request,
            "title": settings.DASHBOARD_TITLE,
        },
    )
```

test_runner.py

```
import os
from typing import Optional, Dict, Any
from uuid import uuid4

from playwright.async_api import async_playwright

from app.core.config import get_settings
from app.services.parser import ParserEngine, ParsedInstruction
from app.services.ai_selector import SmartElementLocator
from app.services.storage import Storage
from app.services.logger import RunLogger
from app.services.result_analyzer import ResultAnalyzer


async def execute_instruction(run_id: str, instruction: str, url: Optional[str]) -> None:
    from app.main import app  # local import to access app state

    settings = get_settings()
    storage: Storage = app.state.storage

    # Mark as running
    run = await storage.get_test_run(run_id) or {"_id": run_id}
```

```
    run["status"] = "running"
    await storage.save_test_run(run)

    logger = RunLogger(storage, run_id)
    parser = ParserEngine()
    selector = SmartElementLocator()

    screenshots_dir = os.path.join(settings.DATA_DIR, "screenshots", run_id)
    os.makedirs(screenshots_dir, exist_ok=True)

    try:
        parsed = parser.parse(instruction)
        await logger.log("info", f"Parsed instruction: action={parsed.action}, role={parsed.target_role},
text={parsed.target_text}")

        async with async_playwright() as p:
            browser_type = getattr(p, settings.PLAYWRIGHT_BROWSER)
            browser = await browser_type.launch(headless=settings.PLAYWRIGHT_HEADLESS,
slow_mo=settings.PLAYWRIGHT_SLOW_MO_MS)
            context = await browser.new_context(viewport={"width": settings.DEFAULT_VIEWPORT_WIDTH, "height":
settings.DEFAULT_VIEWPORT_HEIGHT})
            page = await context.new_page()

            target_url = url or "https://example.com/"
            await logger.log("info", f"Open URL: {target_url}")
            await page.goto(target_url)

            # Screenshot after load
            shot_path = os.path.join(screenshots_dir, f"{uuid4()}.png")
            await page.screenshot(path=shot_path, full_page=True)
            await logger.attach_screenshot(f"/data/screenshots/{run_id}/" + os.path.basename(shot_path), "After page load")

            # Locate element if needed
            element = await selector.locate(page, parsed)

            if parsed.action == "check_visible":
                if not element:
                    await logger.log("error", "Element not found for visibility check")
                else:
                    visible = await element.first.is_visible()
                    if visible:
                        await logger.log("info", "Element is visible")
                    else:
                        await logger.log("error", "Element is not visible")
            elif parsed.action == "click":
                if not element:
                    await logger.log("error", "Element to click not found")
                else:
                    await element.first.click()
                    await logger.log("info", "Clicked element")
            elif parsed.action == "type":
                if not element:
                    await logger.log("error", "Input element not found to type")
                else:
                    await element.first.fill(parsed.target_text or "")
                    await logger.log("info", "Typed into element")

            # Screenshot after action
            shot_path = os.path.join(screenshots_dir, f"{uuid4()}.png")
            await page.screenshot(path=shot_path, full_page=True)
```

```
            await logger.attach_screenshot(f"/data/screenshots/{run_id}/" + os.path.basename(shot_path), "After action")

            await context.close()
            await browser.close()

    except Exception as exc:  # pragma: no cover
        await logger.log("error", f"Unhandled error: {exc}")

    analyzer = ResultAnalyzer(storage, run_id)
    await analyzer.finalize()


async def execute_scenario(run_id: str, scenario: Dict[str, Any]) -> None:
    from app.main import app
    settings = get_settings()
    storage: Storage = app.state.storage

    run = await storage.get_test_run(run_id) or {"_id": run_id}
    run["status"] = "running"
    await storage.save_test_run(run)
    logger = RunLogger(storage, run_id)
    selector = SmartElementLocator()

    screenshots_dir = os.path.join(settings.DATA_DIR, "screenshots", run_id)
    os.makedirs(screenshots_dir, exist_ok=True)

    url = scenario.get("meta", {}).get("url") or "https://example.com/"
    try:
        async with async_playwright() as p:
            browser_type = getattr(p, settings.PLAYWRIGHT_BROWSER)
            browser = await browser_type.launch(headless=settings.PLAYWRIGHT_HEADLESS,
slow_mo=settings.PLAYWRIGHT_SLOW_MO_MS)
            context = await browser.new_context(viewport={"width": settings.DEFAULT_VIEWPORT_WIDTH, "height":
settings.DEFAULT_VIEWPORT_HEIGHT})
            page = await context.new_page()
            await logger.log("info", f"Open URL: {url}")
            await page.goto(url)

            shot_path = os.path.join(screenshots_dir, f"{uuid4()}.png")
            await page.screenshot(path=shot_path, full_page=True)
            await logger.attach_screenshot(f"/data/screenshots/{run_id}/" + os.path.basename(shot_path), "Initial state")

            for idx, step in enumerate(scenario.get("steps", [])):
                action = step.get("action")
                value = step.get("value")
                intent = ParsedInstruction(
                    action="check_visible" if action == "wait_visible" else ("type" if action == "type" else "click" if action == "click"
else "check_visible"),
                    target_text=step.get("selectorText") or step.get("aiContext"),
                    target_role=None,
                )
                el = await selector.locate(page, intent)

                if action == "wait_visible":
                    if not el:
                        await logger.log("error", f"Step {idx+1}: element not found to wait")
                    else:
                        await el.first.wait_for(state="visible")
                        await logger.log("info", f"Step {idx+1}: element visible")
                elif action == "click":
```

```
                if not el:
                    await logger.log("error", f"Step {idx+1}: element not found to click")
                else:
                    await el.first.click()
                    await logger.log("info", f"Step {idx+1}: clicked")
            elif action == "type":
                if not el:
                    await logger.log("error", f"Step {idx+1}: input not found to type")
                else:
                    await el.first.fill(value or "")
                    await logger.log("info", f"Step {idx+1}: typed '{value or ''}'")
            else:
                await logger.log("error", f"Step {idx+1}: unknown action '{action}'")

            shot_path = os.path.join(screenshots_dir, f"{uuid4()}.png")
            await page.screenshot(path=shot_path, full_page=True)
            await logger.attach_screenshot(f"/data/screenshots/{run_id}/" + os.path.basename(shot_path), f"After step {idx+1}")

        await context.close()
        await browser.close()
    except Exception as exc:
        await logger.log("error", f"Unhandled error: {exc}")

    analyzer = ResultAnalyzer(storage, run_id)
    await analyzer.finalize()
```

result_analyzer.py

```
from typing import Dict, Any
from app.services.storage import Storage

class ResultAnalyzer:
    def __init__(self, storage: Storage, run_id: str) -> None:
        self.storage = storage
        self.run_id = run_id

    async def finalize(self) -> Dict[str, Any]:
        run = await self.storage.get_test_run(self.run_id)
        if not run:
            return {"error": "run_not_found"}
        logs = run.get("logs", [])
        status = "passed"
        for item in logs:
            if item.get("level") in ("error", "failed"):
                status = "failed"
                break
        run["status"] = status
        await self.storage.save_test_run(run)
        return run
```

parser.py

```
import re
from dataclasses import dataclass
from typing import Optional

from app.core.config import get_settings

try:
    import spacy  # type: ignore
except Exception:  # pragma: no cover
```

```
    spacy = None  # type: ignore


@dataclass
class ParsedInstruction:
    action: str  # e.g., 'check_visible', 'click', 'type'
    target_text: Optional[str] = None  # text like 'Увійти'
    target_role: Optional[str] = None  # button, link, input


class ParserEngine:
    def __init__(self) -> None:
        self.settings = get_settings()
        self._nlp = None
        if self.settings.ENABLE_SPACY and spacy is not None:
            try:
                self._nlp = spacy.load(self.settings.SPACY_MODEL)  # type: ignore
            except Exception:
                # Fallback to a blank model if small model isn't installed
                self._nlp = spacy.blank("en")  # type: ignore

    def parse(self, instruction: str) -> ParsedInstruction:
        text = instruction.strip()

        # Extract quoted text '...' or "..."
        quoted_match = re.search(r"['\"]([^'\"]+)['\"]", text)
        quoted = quoted_match.group(1) if quoted_match else None

        lowered = text.lower()

        # Determine action
        if any(k in lowered for k in ["перевір", "перекон", "ensure", "verify"]):
            action = "check_visible"
        elif any(k in lowered for k in ["натисни", "клік", "click"]):
            action = "click"
        elif any(k in lowered for k in ["введи", "впиши", "type", "enter"]):
            action = "type"
        else:
            action = "check_visible"

        # Determine role
        role = None
        if any(k in lowered for k in ["кнопка", "button"]):
            role = "button"
        elif any(k in lowered for k in ["посилання", "link", "anchor"]):
            role = "link"
        elif any(k in lowered for k in ["поле", "інпут", "input", "textbox", "field"]):
            role = "textbox"

        return ParsedInstruction(action=action, target_text=quoted, target_role=role)
```

scenarios.py

```
from typing import Any, Dict, List, Optional
from uuid import uuid4
from fastapi import APIRouter, Depends, BackgroundTasks
from pydantic import BaseModel, Field

from app.services.storage import Storage

router = APIRouter()
```

```
class ScenarioStep(BaseModel):
    selectorText: Optional[str] = Field(default=None, description="Text to search element by")
    aiContext: Optional[str] = Field(default=None, description="AI context hint for element")
    action: str = Field(..., description="click | type | wait_visible")
    value: Optional[str] = None
    condition: Optional[str] = Field(default=None, description="optional if/else condition expression")


class Scenario(BaseModel):
    _id: Optional[str] = None
    name: str
    steps: List[ScenarioStep]
    meta: Dict[str, Any] = Field(default_factory=dict)


async def get_storage() -> Storage:
    from app.main import app  # avoid circular import
    return app.state.storage


@router.post("/scenarios")
async def create_scenario(s: Scenario, storage: Storage = Depends(get_storage)):
    sid = s._id or str(uuid4())
    payload = s.model_dump()
    payload["_id"] = sid
    await storage.save_scenario(payload)
    return {"scenario_id": sid}


@router.get("/scenarios")
async def list_scenarios(storage: Storage = Depends(get_storage)):
    return await storage.get_scenarios(limit=200)


@router.get("/scenarios/{scenario_id}")
async def get_scenario(scenario_id: str, storage: Storage = Depends(get_storage)):
    sc = await storage.get_scenario(scenario_id)
    return sc or {"error": "not_found"}


@router.post("/scenarios/{scenario_id}/run")
async def run_scenario(scenario_id: str, background: BackgroundTasks, storage: Storage = Depends(get_storage)):
    # Reuse test-run mechanism; create a run that references the scenario
    sc = await storage.get_scenario(scenario_id)
    if not sc:
        return {"error": "scenario_not_found"}

    from app.routers.tests import RunTestResponse  # reuse schema
    from app.services.test_runner import execute_scenario
    from uuid import uuid4 as _uuid4

    run_id = str(_uuid4())
    await storage.save_test_run({
        "_id": run_id,
        "instruction": f"scenario:{scenario_id}",
        "url": sc.get("meta", {}).get("url"),
        "status": "queued",
        "logs": [],
```

```
        "screenshots": [],
        "scenario_id": scenario_id,
    })
    background.add_task(execute_scenario, run_id, sc)
    return RunTestResponse(run_id=run_id, status="queued")
```

ai_selector.py

```
from typing import Optional, List
from playwright.async_api import Page, Locator
from app.services.parser import ParsedInstruction

try:
    ofuzz_available = True
    from rapidfuzz import fuzz  # type: ignore
except Exception:  # pragma: no cover
    ofuzz_available = False


class SmartElementLocator:
    def __init__(self) -> None:
        pass

    async def locate(self, page: Page, intent: ParsedInstruction) -> Optional[Locator]:
        # Try by role + name
        if intent.target_role and intent.target_text:
            try:
                return page.get_by_role(intent.target_role, name=intent.target_text)
            except Exception:
                pass

        # Try by text
        if intent.target_text:
            try:
                loc = page.get_by_text(intent.target_text, exact=False)
                if await loc.count() > 0:
                    return loc.first
            except Exception:
                pass

        # Try common attributes
        if intent.target_text:
            selector_text = intent.target_text.replace('"', '\\"')
            candidates = [
                f"[aria-label*=\"{selector_text}\"]",
                f"[title*=\"{selector_text}\"]",
                f"[placeholder*=\"{selector_text}\"]",
                f"text={selector_text}",
            ]
            for sel in candidates:
                try:
                    loc = page.locator(sel)
                    if await loc.count() > 0:
                        return loc.first
                except Exception:
                    continue

        # Fuzzy similarity over candidate elements (RapidFuzz)
        if intent.target_text and ofuzz_available:
            best = await self._fuzzy_pick(page, intent)
```

```
            if best is not None:
                return best

        # Fallback heuristics: role only
        if intent.target_role:
            try:
                loc = page.get_by_role(intent.target_role)
                if await loc.count() > 0:
                    return loc.first
            except Exception:
                pass

        return None

    async def _fuzzy_pick(self, page: Page, intent: ParsedInstruction) -> Optional[Locator]:
        selector = "button, a, [role=button], input, [type=button], [type=submit]"
        loc = page.locator(selector)
        count = await loc.count()
        if count == 0:
            return None

        candidates: List[str] = []
        for i in range(min(count, 60)):
            node = loc.nth(i)
            text = ""
            try:
                text = await node.inner_text()
            except Exception:
                text = ""
            if not text:
                for attr in ["aria-label", "title", "placeholder", "value", "name"]:
                    try:
                        val = await node.get_attribute(attr)
                        if val:
                            text = val
                            break
                    except Exception:
                        continue
            candidates.append(text)

        best_index = -1
        best_score = -1
        for i, text in enumerate(candidates):
            score = fuzz.token_set_ratio(intent.target_text, text) if text else 0
            if score > best_score:
                best_score = score
                best_index = i

        if best_index >= 0 and best_score >= 60:  # threshold to avoid random picks
            return loc.nth(best_index)
        return None
```

test.py

```
from typing import Optional
from fastapi import APIRouter, Depends, BackgroundTasks
from pydantic import BaseModel, Field
from uuid import uuid4

from app.services.storage import Storage
```

```
router = APIRouter()


class RunTestRequest(BaseModel):
    instruction: str = Field(..., description="Natural language instruction, e.g. 'перевірити, що кнопка \"Увійти\" доступна'")
    url: Optional[str] = Field(default=None, description="Target URL to open before executing instruction")


class RunTestResponse(BaseModel):
    run_id: str
    status: str


async def get_storage() -> Storage:
    # This function is a dependency to resolve Storage from app state
    from app.main import app  # local import to avoid circular
    return app.state.storage


@router.post("/tests/run", response_model=RunTestResponse)
async def run_test(req: RunTestRequest, background: BackgroundTasks, storage: Storage = Depends(get_storage)):
    run_id = str(uuid4())
    # Create initial run record
    await storage.save_test_run(
        {
            "_id": run_id,
            "instruction": req.instruction,
            "url": req.url,
            "status": "queued",
            "logs": [],
            "screenshots": [],
        }
    )

    # Defer actual execution to background task
    from app.services.test_runner import execute_instruction  # noqa: WPS433

    background.add_task(execute_instruction, run_id, req.instruction, req.url)
    return RunTestResponse(run_id=run_id, status="queued")


@router.get("/tests")
async def list_tests(storage: Storage = Depends(get_storage)):
    return await storage.get_test_runs(limit=100)


@router.get("/tests/{run_id}")
async def get_test(run_id: str, storage: Storage = Depends(get_storage)):
    run = await storage.get_test_run(run_id)
    if run is None:
        return {"error": "not_found"}
    return run
```