

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

**Ніколайчук Владислав Сергійович**

**ПРОГРЕС В ІНСТРУМЕНТАХ АВТОМАТИЗОВАНОГО  
ТЕСТУВАННЯ ТА ЇХ ВПЛИВ НА ЖИТТЄВИЙ ЦИКЛ РОЗРОБКИ  
ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

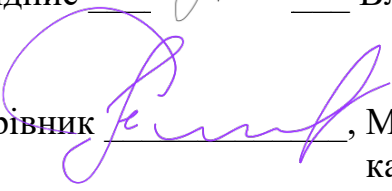
**кваліфікаційна робота**

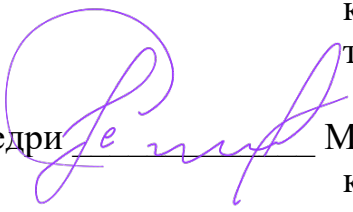
**здобувача вищої освіти другого (магістерського) рівня**

**освітньої програми «Мультимедійні системи»**

**за спеціальністю F2 «Інженерія програмного забезпечення»**

Особистий підпис  Владислав НІКОЛАЙЧУК

Науковий керівник , Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Завідувач кафедри , Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

## **АНОТАЦІЯ**

**Тема:** Прогрес в інструментах автоматизованого тестування та їх вплив на життєвий цикл розробки програмного забезпечення.

**Спеціальність:** F2 «Інженерія програмного забезпечення».

**Установа:** ЛНУ імені Тараса Шевченка, 2026 р.

**Магістерська робота містить:** 82 с., 10 рис., 8 табл., 42 джерел.

Об'єкт дослідження – процеси забезпечення якості програмного забезпечення в межах життєвого циклу його розробки.

Предмет дослідження – методи, моделі та інструменти автоматизованого тестування, а також їх вплив на метрики ефективності та якості програмного забезпечення в умовах використання CI/CD.

**Мета дослідження** – дослідження сучасних інструментів автоматизованого тестування та визначення їх впливу на життєвий цикл розробки програмного забезпечення.

**Результати дослідження** – інтегральний підхід до аналізу впливу автоматизованих тестів на показники життєвого циклу програмного забезпечення, що дозволяє кількісно оцінювати доцільність упровадження сучасних інструментів автоматизації тестування в реальних програмних проєктах.

**Ключові слова:** ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, SDLC, PLAYWRIGHT, JENKINS, ALLURE REPORT

## **ABSTRACT**

**Topic:** Progress in Automated Testing Tools and Their Impact on the Software Development Life Cycle..

**Speciality:** F2 "Software engineering".

**Institution:** Luhansk Taras Shevchenko National University (LTSNU), 2026.

**Master's thesis consists of:** 82 pages, 10 figures, 8 tables, 42 references

**Object of the research** –. processes of software quality assurance within the software development life cycle.

**Subject of the research** – methods, models, and tools of automated testing, as well as their impact on efficiency and quality metrics of software under CI/CD practices.

**Purpose of the research** – to study modern automated testing tools and determine their impact on the software development life cycle.

**Research results** is an integrated approach to analyzing the influence of automated tests on software life cycle indicators, enabling quantitative evaluation of the feasibility of implementing modern test automation tools in real-world software projects.

**Keywords:** SOFTWARE ENGINEERING, AUTOMATED TESTING, SDLC, PLAYWRIGHT, JENKINS, ALLURE REPORT.

## ЗМІСТ

ВСТУП	5
РОЗДІЛ 1 СУЧАСНІ ПАРАДИГМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ В ЖИТТЄВОМУ ЦИКЛІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	8
1.1 Місце тестування в SDLC та еволюція підходів до контролю якості	8
1.2 Класифікація інструментів автоматизованого тестування та їх порівняльний аналіз	12
1.3. Проблематика впровадження автоматизації та її вплив на метрики якості ПЗ.	17
РОЗДІЛ 2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ СИСТЕМИ ОЦІНКИ ЕФЕКТИВНОСТІ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ	22
2.1. Формалізація метрик тестового покриття та ефективності автоматизованого тестування	22
2.2. Обґрунтування вибору технологічного стеку для побудови універсального тестового фреймворку.	29
2.3 Проєктування системи звітності та аналізу результатів тестування	33
РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ВПЛИВУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НА SDLC	36
3.1. Реалізація автоматизованої системи тестування засобами Python і Playwright	36
3.2. Інтеграція тестування в CI/CD-конвеєр з використанням Docker і Jenkins	45
3.3. Організація експериментального дослідження та методика оцінювання	49
3.4. Аналіз результатів експерименту та оцінка впливу на життєвий цикл ПЗ	52
ЗАГАЛЬНІ ВИСНОВКИ	61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	64
ДОДАТКИ	69
Додаток Б. Налаштування системи	72
Додаток В. Dockerfile	77
Додаток Г. Jenkinsfile (Declarative Pipeline)	80

## ВСТУП

Сьогодні важливість організації автоматизованого тестування зумовлена стрімким зростанням складності програмних систем, скороченням циклів розробки та підвищеними вимогами до якості програмного забезпечення в умовах сучасних підходів Agile, DevOps і CI/CD. У таких умовах традиційні підходи до тестування виявляються недостатньо ефективними, оскільки не забезпечують своєчасного виявлення дефектів і не масштабуються разом із розвитком програмного продукту. Автоматизоване тестування перетворюється з допоміжного інструмента на ключовий елемент життєвого циклу розробки програмного забезпечення, що безпосередньо впливає на швидкість релізів, стабільність системи та загальну вартість розробки. Водночас швидкий прогрес інструментів автоматизованого тестування породжує проблему обґрунтованого вибору технологій, оцінювання їх ефективності та визначення реального впливу на показники SDLC (Software Development Life Cycle), що й зумовлює актуальність обраної теми магістерської роботи.

Проблеми автоматизованого тестування програмного забезпечення досліджувалися багатьма науковцями та практиками у сфері програмної інженерії. Значний внесок у формування теоретичних основ тестування зробили Г. Майєрс [37], Б. Бейзер [6; 7] та Р. Прессман [27], які заклали базові принципи контролю якості програмного забезпечення. Питання інтеграції тестування в процеси життєвого циклу ПЗ розглядалися в роботах І. Соммервілла [36] та М. Фаулера [11-13], зокрема в контексті гнучких методологій розробки. Сучасні аспекти автоматизованого та безперервного тестування в DevOps-середовищі висвітлені в працях Дж. Хамбла, Д. Фарлі та Н. Форсгрена [15; 16]. Практичні підходи до побудови автоматизованих тестових фреймворків і використання інструментів end-to-end тестування, таких як Selenium, Cypress і Playwright,

активно досліджуються в прикладних роботах та галузевих дослідженнях, що підтверджує актуальність і практичну значущість обраної теми.

Проблема, яка розглядається в кваліфікаційній роботі, полягає у відсутності формалізованих підходів до оцінювання ефективності автоматизованого тестування в контексті повного життєвого циклу програмного забезпечення. На практиці впровадження автоматизованих тестів часто здійснюється інтуїтивно, без кількісного аналізу результатів, що ускладнює прийняття інженерних і управлінських рішень. Недостатньо дослідженим залишається питання взаємозв'язку між рівнем автоматизації тестування, тестовим покриттям, якісними метриками програмного продукту та показниками ефективності CI/CD-процесів.

Мета магістерської роботи – дослідження сучасних інструментів автоматизованого тестування та визначення їх впливу на життєвий цикл розробки програмного забезпечення.

Для досягнення поставленої мети в роботі передбачено розв'язання таких завдань:

- 1) проаналізувати сучасні підходи та інструменти автоматизованого тестування в межах SDLC;
- 2) формалізувати основні метрики оцінки ефективності тестування; розробити математичну модель оцінювання тестового покриття та ефективності автоматизованих тестів;
- 3) спроектувати архітектуру автоматизованої системи тестування; реалізувати тестову інфраструктуру з інтеграцією в CI/CD-конвеєр;
- 4) провести експериментальне дослідження та виконати порівняльний аналіз показників якості до і після впровадження автоматизованого тестування.

Об'єкт дослідження – процеси забезпечення якості програмного забезпечення в межах життєвого циклу його розробки.

Предмет дослідження – методи, моделі та інструменти автоматизованого тестування, а також їх вплив на метрики ефективності та якості програмного забезпечення в умовах використання CI/CD.

Наукова новизна роботи полягає в удосконаленні підходу до оцінювання ефективності автоматизованого тестування шляхом поєднання формалізованої математичної моделі тестового покриття з практичними метриками CI/CD-процесів. Запропоновано інтегральний підхід до аналізу впливу автоматизованих тестів на показники життєвого циклу програмного забезпечення, що дозволяє кількісно оцінювати доцільність упровадження сучасних інструментів автоматизації тестування в реальних програмних проєктах.

У процесі дослідження було використано загальнонаукові та спеціальні методи дослідження, зокрема методи аналізу та синтезу для вивчення сучасних підходів до тестування, системний підхід для розгляду автоматизованого тестування як складової SDLC, методи математичного моделювання для формалізації показників ефективності тестового покриття, методи проєктування програмних систем для побудови архітектури автоматизованої системи тестування, а також експериментальні методи та статистичний аналіз для оцінювання результатів впровадження розробленого рішення.

## РОЗДІЛ 1 СУЧАСНІ ПАРАДИГМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ В ЖИТТЄВОМУ ЦИКЛІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

### 1.1 Місце тестування в SDLC та еволюція підходів до контролю якості

Тестування програмного забезпечення є невід’ємною складовою життєвого циклу розробки програмного забезпечення (SDLC) і виконує ключову роль у забезпеченні його якості, надійності та відповідності вимогам користувачів. На ранніх етапах становлення програмної інженерії тестування розглядалося переважно як завершальна фаза розробки, метою якої було виявлення помилок перед передачею продукту в експлуатацію. Такий підхід був характерний для каскадної моделі SDLC [27; 36] (рис. 1.1), у межах якої етапи аналізу, проєктування, реалізації та тестування виконувалися послідовно.



Рис. 1.1 Каскадна модель



Як бачимо з рис. 1.1 тестування розміщувалося наприкінці життєвого циклу, після завершення етапів аналізу вимог, проєктування та реалізації. Така організація процесу зумовлювала накопичення помилок, які виявлялися лише на фінальній стадії, коли виправлення дефектів вимагало значних витрат часу й ресурсів. Фактично тестування виконувало роль інструмента контролю вже створеного продукту, а не засобу активного впливу на процес його розробки. Це призводило до зниження гнучкості SDLC та ускладнювало адаптацію програмного забезпечення до змін вимог.

Подальший розвиток методологій розробки програмного забезпечення сприяв переосмисленню ролі тестування в SDLC. Із впровадженням ітеративних та інкрементних моделей тестування почало інтегруватися в кожен цикл розробки, що дало змогу виявляти дефекти на ранніх стадіях і поступово підвищувати якість програмного продукту. Особливого значення тестування набуло в межах гнучких методологій Agile, де контроль якості став безперервним процесом, тісно пов'язаним із розробкою, а тестові сценарії почали формуватися паралельно з вимогами до системи. В умовах Agile-підходів тестування було переміщене з кінця процесу до його середини, а згодом і до початкових етапів. Тестові сценарії почали формуватися одночасно з користувацькими вимогами, а перевірка якості стала невід'ємною частиною кожної ітерації. Це дозволило виявляти дефекти на ранніх стадіях, оперативно реагувати на зміни та поступово підвищувати стабільність програмного продукту без істотного збільшення вартості розробки. У таких умовах тестування перестало бути окремою фазою й перетворилося на постійний процес перевірки коректності функціонування програмного забезпечення.

Ключовою концепцією, що відображає цю трансформацію, є підхід Shift-Left Testing [18; 31-35], який передбачає перенесення активностей тестування максимально близько до початку життєвого циклу розробки (рис. 1.2). У межах цього підходу тестування охоплює етапи аналізу вимог, проєктування та

написання коду, а не обмежується лише перевіркою готового продукту. Застосування Shift-Left Testing сприяє зменшенню кількості дефектів на пізніх етапах, підвищенню якості архітектурних рішень та формуванню культури відповідальності за якість серед усіх учасників процесу розробки.

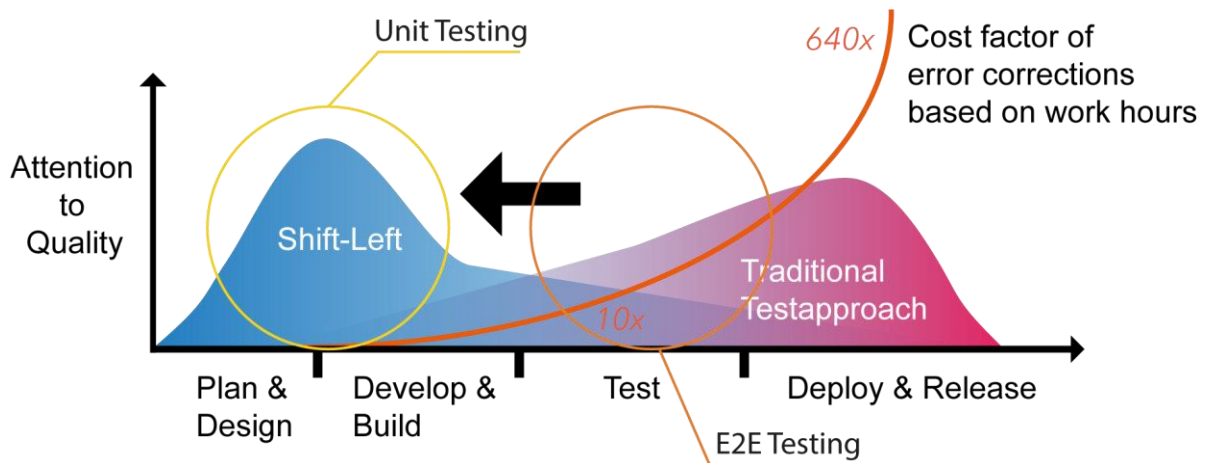


Рис.1.2 Ідея Shift-Left Testing [33]

Подальша еволюція підходів до контролю якості пов'язана з поширенням Continuous Testing та практик DevOps і CI/CD [41], у межах яких тестування інтегрується безпосередньо в автоматизовані конвеєри збірки, розгортання та доставки програмного продукту.

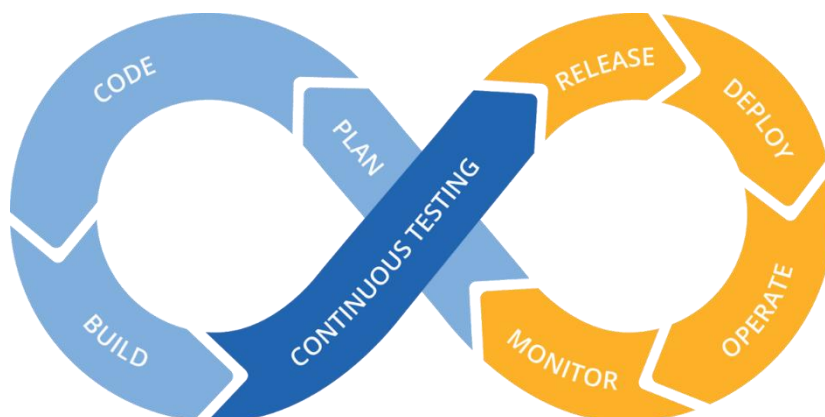


Рис. 1.3 Continuous Testing – принцип практик DevOps і CI/CD

У цьому контексті автоматизоване тестування стало критично важливим інструментом забезпечення стабільності та прогнозованості релізів. Регресійні, інтеграційні та end-to-end тести виконуються автоматично після кожної зміни коду, що дає змогу оперативно виявляти дефекти та знижувати ризики порушення працездатності системи [1-3; 19]. Таким чином, тестування перетворюється на один із ключових механізмів зворотного зв'язку між етапами розробки, експлуатації та супроводу програмного забезпечення.

Еволюція підходів до контролю якості також супроводжувалася зміною фокусу з виявлення помилок на запобігання їх появи. Сучасні підходи передбачають використання практик test-driven development, behavior-driven development, статичного аналізу коду та автоматизованих перевірок якості, що дозволяє підвищувати надійність програмних систем ще на етапі їх проєктування та реалізації [20; 23]]. У результаті тестування в сучасному SDLC розглядається як комплексний, багатоаспектний процес, що охоплює весь життєвий цикл програмного забезпечення та безпосередньо впливає на ефективність розробки, якість кінцевого продукту й конкурентоспроможність програмних рішень.

Таким чином, еволюція підходів до тестування призвела до його інтеграції в усі етапи життєвого циклу розробки програмного забезпечення. Переміщення тестування з фінальної стадії до початкових і проміжних етапів, упровадження концепцій Shift-Left Testing та безперервної інтеграції трансформували тестування з допоміжної діяльності на стратегічний інструмент управління якістю. У сучасних умовах тестування виступає ключовим чинником забезпечення надійності, масштабованості та швидкості розробки програмних систем.

## 1.2 Класифікація інструментів автоматизованого тестування та їх порівняльний аналіз

Інструменти автоматизованого тестування програмного забезпечення можуть бути класифіковані за рівнем тестування, архітектурними підходами та способом взаємодії з тестованою системою. За функціональним призначенням виділяють інструменти для модульного тестування, інтеграційного тестування, системного та end-to-end тестування, а також спеціалізовані засоби для навантажувального й безпекового тестування. У контексті забезпечення якості веборієнтованих програмних систем особливого значення набули інструменти автоматизованого end-to-end тестування, які імітують реальну поведінку користувача та перевіряють коректність роботи системи в цілому.

Сучасний ринок інструментів автоматизованого тестування значною мірою представлений open source рішеннями, що забезпечують гнучкість використання, активну підтримку спільнот і можливість інтеграції в різноманітні технологічні стеки. Тривалий час домінуючим підходом до автоматизації UI-тестування залишався класичний підхід на базі WebDriver-протоколу, реалізований у фреймворку Selenium [30] (рис. 1.4).

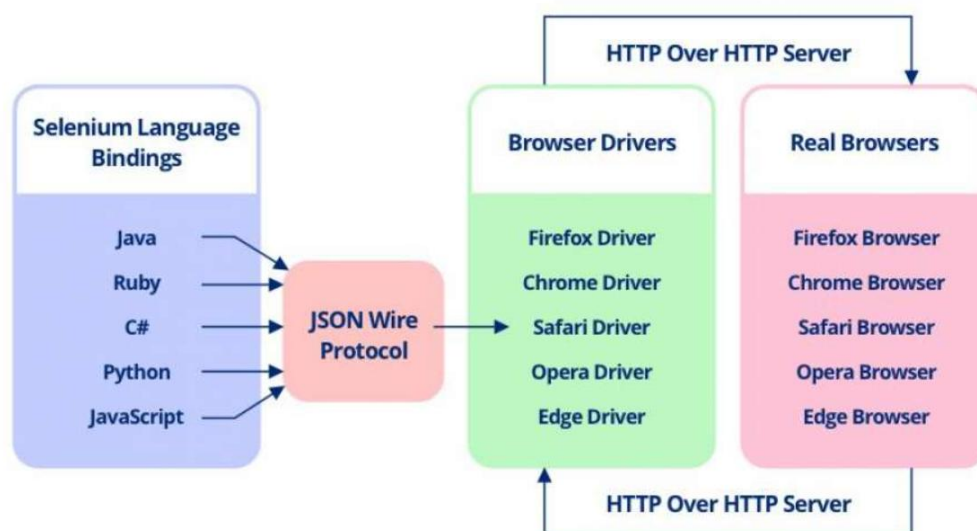


Рис. 1.4 Реалізація фреймворку Selenium на базі WebDriver [30]

З рис. 1.4 бачимо, що основною ідеєю WebDriver-протоколу є віддалене керування браузером через стандартизований API, що забезпечує широку підтримку браузерів і мов програмування. Водночас архітектурні особливості WebDriver, зокрема необхідність посередника між тестовим кодом і браузером, часто призводять до зниження швидкодії та появи нестабільних тестів, чутливих до таймінгів і асинхронних подій.

Подальша еволюція інструментів автоматизованого тестування була пов'язана з появою фреймворків, що використовують сучасні протоколи взаємодії з браузерами, зокрема Chrome DevTools Protocol та механізми WebSockets. До цієї групи належать Playwright, Cypress і Puppeteer, які забезпечують більш тісну інтеграцію з браузером і прямий доступ до його внутрішніх механізмів [26]. Такий підхід дозволяє значно підвищити швидкість виконання тестів, зменшити кількість флейкових сценаріїв та спростити роботу з асинхронними вебзастосунками. Зокрема, Playwright вирізняється кросбраузерною підтримкою, розширеними можливостями керування контекстами браузера та стабільною роботою в середовищах CI/CD.

Порівняльний аналіз інструментів автоматизованого тестування доцільно здійснювати за низкою ключових критеріїв, що безпосередньо впливають на ефективність їх використання в реальних проєктах. До таких критеріїв належать швидкість виконання тестів, стабільність результатів (рівень flakiness), підтримка мов програмування та легкість інтеграції в процеси безперервної інтеграції й доставки. Узагальнені результати порівняльного аналізу подано в таблиці 1.1.

Проведений аналіз свідчить про те, що хоча Selenium залишається універсальним і широко використовуваним інструментом, сучасні фреймворки на основі DevTools-протоколів демонструють кращі показники швидкодії та стабільності. Це робить їх більш придатними для використання в сучасних SDLC-моделях, орієнтованих на безперервну інтеграцію, часті релізи та автоматизований контроль якості. З огляду на ці характеристики, застосування

Playwright як базового інструмента автоматизованого тестування є обґрунтованим у контексті дослідження впливу сучасних засобів автоматизації на життєвий цикл розробки програмного забезпечення.

Таблиця 1.1

Порівняльний аналіз інструментів автоматизованого тестуванн

Критерій	Selenium (WebDriver)	Playwright	Cypress	Puppeteer
Архітектурний підхід	WebDriver (віддалений драйвер)	DevTools / WebSockets	DevTools (вбудований рантайм)	DevTools
Швидкість виконання тестів	Середня	Висока	Висока	Висока
Стабільність (flakiness)	Середня / низька	Висока	Висока	Висока
Підтримка браузерів	Chrome, Firefox, Edge, Safari	Chromium, Firefox, WebKit	Chromium- based	Chromium
Підтримка мов	Java, Python, C#, JS та ін.	JavaScript, TypeScript, Python, C#	JavaScript, TypeScript	JavaScript

Критерій	Selenium (WebDriver)	Playwright	Cypress	Puppeteer
Інтеграція з CI/CD	Потребує додаткового налаштування	Вбудована підтримка	Вбудована підтримка	Обмежена
Простота налаштування	Середня	Висока	Висока	Висока

Нове покоління інструментів, таких як Playwright, Puppeteer та Cypress, пропонує архітектурні рішення, орієнтовані на глибшу інтеграцію з середовищем виконання. Інструменти на кшталт Playwright та Puppeteer використовують протокол Chrome DevTools Protocol (CDP) або прямі WebSocket-з'єднання для двостороннього зв'язку з браузером, що дозволяє не лише відправляти команди, а й отримувати миттєвий зворотний зв'язок від рендерингу сторінки. Cypress йде ще далі, виконуючи код тестів безпосередньо у тому ж циклі подій (run loop), що й код самого веб-додатку, забезпечуючи нативний доступ до DOM-дерева та змінних JavaScript. Ці архітектурні відмінності безпосередньо впливають на стабільність та функціональні можливості тестів.

Найвагомішою перевагою сучасних архітектур є вбудований механізм автоматичного очікування (auto-waiting), який усуває необхідність ручного прописування пауз та штучних затримок, що є основною причиною нестабільності («крихкості») тестів у Selenium. Завдяки прямому доступу до мережевого шару браузера, новітні інструменти дозволяють легко перехоплювати, модифікувати або емулювати (mock) мережеві запити, що значно спрощує тестування складних сценаріїв та ізоляцію фронтенду від бекенду. Крім того, сучасні фреймворки надають розширені можливості для візуального

налагодження, такі як запис відео, створення повних трасувань (traces) виконання з DOM-знімками на кожному кроці та можливість тестування в режимі «headless» з продуктивністю, наближеною до нативної. Таким чином, перехід від HTTP-орієнтованої архітектури до протоколів прямої взаємодії та внутрішньопроцесного виконання знаменує собою еволюційний стрибок у розвитку засобів автоматизації, дозволяючи інтегрувати тестування глибше у процес розробки та забезпечувати вищу надійність перевірки якості програмного забезпечення.



### 1.3. Проблематика впровадження автоматизації та її вплив на метрики якості ПЗ.

Впровадження автоматизованого тестування в процес розробки програмного забезпечення супроводжується низкою технічних, організаційних та методологічних проблем, які безпосередньо впливають на ефективність забезпечення якості програмного продукту. Попри очевидні переваги автоматизації, її застосування не гарантує автоматичного підвищення якості ПЗ без належного планування, вибору інструментів і адаптації процесів розробки. Однією з ключових проблем є початкова складність впровадження автоматизованих тестів, що вимагає додаткових витрат часу на проєктування тестової архітектури, навчання команди та інтеграцію тестів у наявний SDLC.

Значною проблемою автоматизованого тестування є підтримка тестів у довгостроковій перспективі. Зі зростанням і зміною функціональності програмного забезпечення тестові сценарії потребують постійного оновлення, що може призводити до зростання технічного боргу тестової інфраструктури. Особливо гостро ця проблема проявляється у випадку UI-орієнтованих тестів, які є чутливими до змін інтерфейсу користувача. Недостатньо структурований тестовий код або відсутність чіткої стратегії автоматизації призводять до зниження стабільності тестів, появи флейкових результатів та зменшення довіри до автоматизованого тестування як інструмента контролю якості.

Важливим аспектом проблематики впровадження автоматизації є вплив автоматизованого тестування на ключові метрики якості програмного забезпечення. До таких метрик належать тестове покриття, кількість дефектів, виявлених на різних етапах SDLC, час виявлення та усунення помилок, стабільність релізів і середній час між відмовами. За умови коректної інтеграції автоматизованих тестів у процеси розробки спостерігається зростання показників

раннього виявлення дефектів, що позитивно впливає на загальну якість програмного продукту та знижує вартість його супроводу.

Разом із тим автоматизоване тестування не завжди забезпечує повне функціональне покриття та не може розглядатися як універсальна заміна ручного тестування [2;3]. Надмірна концентрація на кількісних показниках, зокрема на рівні code coverage, без урахування якості тестових сценаріїв може створювати хибне відчуття високої якості програмного забезпечення. Тому важливим є комплексний підхід до оцінювання якості ПЗ, який поєднує автоматизоване тестування з аналізом дефектів, результатами експлуатаційного моніторингу та зворотним зв'язком від користувачів [4].

У контексті сучасних CI/CD-процесів автоматизоване тестування безпосередньо впливає на швидкість і надійність доставки програмного забезпечення. З одного боку, автоматизація дозволяє суттєво скоротити час регресійного тестування та підвищити частоту релізів. З іншого боку, надмірна кількість повільних або нестабільних тестів може негативно впливати на тривалість конвеєра безперервної інтеграції та знижувати продуктивність команди розробки. Таким чином, проблематика впровадження автоматизованого тестування полягає у знаходженні балансу між рівнем автоматизації, витратами на підтримку тестів і досягненням цільових показників якості програмного забезпечення [20; 23; 25].

Узагальнюючи, автоматизоване тестування слід розглядати не як окремий технічний інструмент, а як системний елемент управління якістю програмного забезпечення [38]. Його вплив на метрики якості ПЗ визначається не лише вибором інструментів автоматизації, але й зрілістю процесів розробки, рівнем інтеграції в SDLC та здатністю команди використовувати результати тестування для прийняття обґрунтованих інженерних рішень.

Playwright (рис. 1.5) є сучасним інструментом автоматизованого end-to-end тестування, розробленим з урахуванням потреб сучасних вебзастосунків і вимог

безперервної інтеграції та доставки. Його поява стала відповіддю на обмеження класичних підходів до UI-тестування, зокрема проблеми нестабільності та низької продуктивності, характерні для рішень на базі WebDriver. Playwright забезпечує прямий і контрольований доступ до браузерів через сучасні протоколи взаємодії, що дозволяє значно підвищити точність і надійність автоматизованих тестів [26].

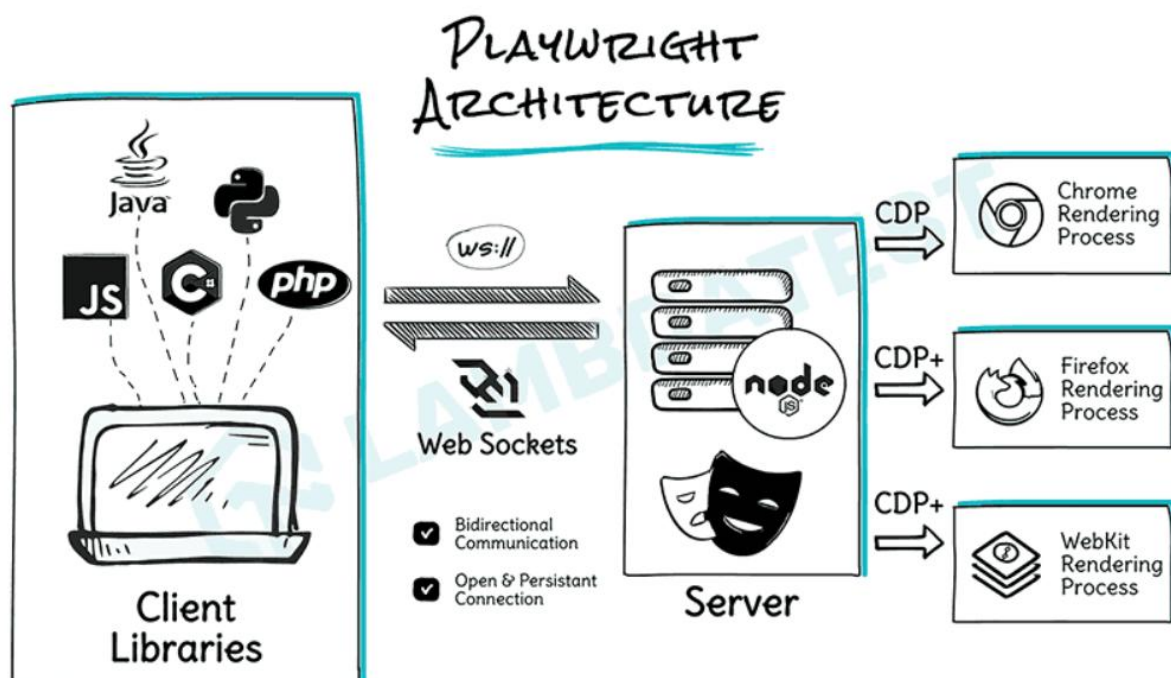


Рис. 1.5. Архітектура Playwright [26]

Ключовою концепцією Playwright є кросбраузерність на рівні ядра інструмента. Фреймворк підтримує Chromium, Firefox і WebKit, що дозволяє виконувати тести в середовищах, максимально наближених до реальних умов експлуатації вебзастосунків. На відміну від багатьох альтернативних рішень, Playwright забезпечує однакову модель API для всіх підтримуваних браузерів, що спрощує розробку тестів і знижує витрати на їх супровід. Це є особливо важливим у проєктах, де критичним є забезпечення стабільної роботи інтерфейсу на різних платформах і пристроях [26].

Однією з визначальних особливостей Playwright є його орієнтація на роботу з асинхронною природою сучасних вебінтерфейсів. Інструмент реалізує вбудовані механізми автоматичного очікування подій, елементів і станів сторінки, що зменшує потребу в ручному керуванні тайм-аутами та затримками. Завдяки цьому зменшується ймовірність появи флейкових тестів, які часто виникають через некоректну синхронізацію в інших фреймворках. Такий підхід сприяє підвищенню стабільності результатів тестування та довіри до автоматизованих перевірок.

Playwright також надає розширені можливості для ізоляції тестових сценаріїв шляхом використання браузерних контекстів. Кожен тест може виконуватися в окремому середовищі з власним станом сесії, cookie та сховищем даних, що забезпечує незалежність тестів і спрощує паралельне виконання. Це є важливим чинником масштабованості тестової інфраструктури, особливо в умовах великих проєктів і CI/CD-конвеєрів, де одночасно запускається значна кількість тестів.

Суттєвою перевагою Playwright є підтримка кількох мов програмування, зокрема JavaScript, TypeScript, Python і C#, що розширює можливості його інтеграції в різні технологічні стеки. У поєднанні з розвиненою екосистемою плагінів і засобів звітності, зокрема Allure Report, Playwright дозволяє будувати повноцінні системи автоматизованого тестування з детальною візуалізацією результатів. Крім того, фреймворк добре адаптований до контейнеризованих середовищ і безперервної інтеграції, що робить його ефективним інструментом для сучасних процесів розробки програмного забезпечення.

Таким чином, Playwright поєднує сучасні архітектурні підходи, високу стабільність і зручність використання, що дозволяє розглядати його як один із найбільш перспективних інструментів автоматизованого end-to-end тестування. Його концепції та можливості відповідають вимогам сучасного SDLC та

створюють основу для побудови ефективної, масштабованої та надійної системи контролю якості програмного забезпечення.

## РОЗДІЛ 2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ СИСТЕМИ ОЦІНКИ ЕФЕКТИВНОСТІ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

### 2.1. Формалізація метрик тестового покриття та ефективності автоматизованого тестування

Формалізація метрик тестового покриття та ефективності автоматизованого тестування є необхідною умовою об'єктивного оцінювання якості програмного забезпечення та результативності процесів контролю якості в межах SDLC. У практиці розробки програмного забезпечення часто використовується велика кількість показників, однак без чіткого формального визначення вони не дозволяють отримати достовірні та порівнювані результати. Тому ключовим завданням є виокремлення базових метрик і визначення їхнього змісту, способів обчислення та інтерпретації в контексті автоматизованого тестування.

Однією з найпоширеніших груп метрик є метрики тестового покриття, які відображають ступінь охоплення програмного коду або функціональних вимог тестами. До них належать показники покриття інструкцій, гілок, умов, шляхів виконання, а також функціонального покриття, що характеризує відповідність тестів бізнес-вимогам. Формально тестове покриття може бути визначене як відношення кількості перевірених елементів системи до їх загальної кількості. Такий підхід дозволяє отримати кількісну оцінку повноти тестування, однак не відображає якість самих тестових сценаріїв і здатність тестів виявляти дефекти.

Для комплексної оцінки автоматизованого тестування важливими є метрики ефективності, що характеризують результативність тестів з точки зору виявлення помилок і впливу на процес розробки. До таких метрик належать щільність дефектів, коефіцієнт виявлення дефектів, середній час виявлення помилки та співвідношення кількості дефектів, знайдених автоматизованими і ручними тестами. Формалізація цих показників дозволяє оцінювати не лише

обсяг виконаного тестування, але й його практичну користь для підвищення якості програмного забезпечення.

Окрему групу становлять метрики, пов'язані з продуктивністю та стабільністю автоматизованого тестування. До них відносять час виконання тестового набору, частоту флейкових тестів, відсоток помилкових спрацьовувань та стабільність результатів у різних середовищах. Ці показники є особливо важливими в умовах CI/CD, оскільки безпосередньо впливають на тривалість конвеєра безперервної інтеграції та швидкість доставки програмного забезпечення.

Формалізація метрик ефективності автоматизованого тестування передбачає також урахування їхнього впливу на загальні показники SDLC. До таких показників належать частота релізів, середній час усунення дефектів, рівень відмов у продуктивному середовищі та стабільність програмного продукту після оновлень. У цьому контексті автоматизоване тестування розглядається як фактор, що опосередковано впливає на якість програмного забезпечення через зменшення кількості критичних помилок і підвищення передбачуваності процесу розробки.

Таким чином, формалізація метрик тестового покриття та ефективності автоматизованого тестування створює основу для подальшого математичного моделювання та експериментального аналізу. Чітке визначення показників і методів їх обчислення дозволяє перейти від інтуїтивної оцінки якості тестування до обґрунтованого, кількісного аналізу впливу автоматизації на життєвий цикл розробки програмного забезпечення.

Методика обчислення метрик тестового покриття та ефективності автоматизованого тестування ґрунтується на формалізованому представленні елементів програмного забезпечення, тестових сценаріїв і результатів виконання тестів. Для забезпечення порівнюваності результатів усі метрики визначаються у

вигляді кількісних показників, що обчислюються на основі даних, зібраних під час автоматизованого тестування та роботи CI/CD-конвеєра.

Базовою метрикою є “загальне тестове покриття”, яке характеризує частку елементів системи, охоплених тестами. Формально тестове покриття визначається як відношення кількості протестованих елементів до загальної кількості елементів:

$$C = \frac{N_{tested}}{N_{total}}, \quad (1.1)$$

де  $N_{tested}$  — кількість елементів коду або функціональних вимог, які були перевірені тестами, а  $N_{total}$  — їх загальна кількість. Залежно від рівня тестування ця формула може застосовуватися для покриття інструкцій, гілок, умов або функціональних вимог.

Для оцінки “ефективності тестового покриття” доцільно враховувати не лише обсяг покриття, а й кількість дефектів, виявлених автоматизованими тестами. Коефіцієнт ефективності покриття може бути визначений як:

$$E_c = \frac{D_{auto}}{N_{tested}}, \quad (1.2)$$

де ( $D_{\text{auto}}$ ) — кількість дефектів, виявлених автоматизованими тестами. Ця метрика дозволяє оцінити, наскільки кожен протестований елемент системи сприяє виявленню помилок.

Важливою метрикою є “коефіцієнт виявлення дефектів автоматизованим тестуванням”, який показує частку дефектів, знайдених автоматично, від загальної кількості виявлених дефектів:

$$DDR = \frac{D_{auto}}{D_{total}}, \quad (1.3)$$



де ( $D_{total}$ ) — загальна кількість дефектів, виявлених у межах певного періоду або релізного циклу. Цей показник відображає реальний внесок автоматизованого тестування в забезпечення якості програмного забезпечення.

Для аналізу впливу автоматизації на швидкість зворотного зв'язку використовується “середній час виявлення дефекту”, який обчислюється як середнє значення інтервалів часу між внесенням змін у код і виявленням відповідного дефекту:

$$MTTD = \frac{1}{D_{auto}} \sum_{i=1}^{D_{auto}} (t^{(i)} * detect - t^{(i)} * commit), \quad (1.4)$$

де ( $t^{(i)} * commit$ ) — час внесення зміни, що спричинила дефект, а ( $t^{(i)} * detect$ ) — час його виявлення автоматизованим тестуванням. Зменшення цього показника свідчить про підвищення ефективності CI/CD-процесів.

Окрему увагу приділено “стабільності автоматизованих тестів”, яка характеризується рівнем флейковості. Коефіцієнт флейковості визначається як відношення кількості нестабільних виконань тестів до загальної кількості запусків:

$$F = \frac{N_{unstable}}{N_{runs}}, \quad (1.5)$$

де  $N_{unstable}$  — кількість запусків тестів з непередбачуваними або неконсистентними результатами, а  $N_{runs}$  — загальна кількість запусків тестового набору.

Для комплексної оцінки ефективності автоматизованого тестування в межах SDLC доцільно використовувати “інтегральний показник ефективності”, який поєднує основні метрики покриття, виявлення дефектів і стабільності:

$$E_{total} = \alpha C + \beta DDR - \gamma F, \quad (1.6)$$

де  $\alpha, \beta, \gamma$  — вагові коефіцієнти, що визначають відносну важливість кожної складової залежно від специфіки проєкту. Такий підхід дозволяє отримати узагальнену кількісну оцінку ефективності автоматизованого

тестування та використовувати її для порівняння різних інструментів або стратегій автоматизації.

Запропонована методика обчислення метрик створює формальну основу для подальшого математичного моделювання та експериментального дослідження впливу автоматизованого тестування на показники якості програмного забезпечення та ефективність життєвого циклу його розробки.

Математична модель оцінки ефективності тестового покриття будується на ідеї, що саме по собі значення покриття (code coverage або functional coverage) не є достатнім індикатором якості тестування, оскільки не враховує здатність тестів виявляти дефекти, витрати часу на виконання, стабільність результатів і придатність тестового набору до використання в CI/CD. Тому ефективність тестового покриття доцільно розглядати як інтегральний показник, що поєднує кілька груп параметрів: повноту охоплення тестованих елементів, результативність щодо дефектів, ресурсні витрати та надійність (стабільність) тестів.

Нехай існує множина тестованих елементів системи  $U$ , яка може трактуватися як набір інструкцій, гілок, вимог або користувацьких сценаріїв. Нехай  $T = t_1, t_2, \dots, t_m$  — набір автоматизованих тестів. Визначимо функцію покриття  $cov(t_i) \subseteq U$ , що задає підмножину елементів, які перевіряє тест  $t_i$ . Тоді загальне покриття тестового набору визначається як:

$$C(T) = \frac{|\cup_{i=1}^m cov(t_i)|}{|U|}$$

Ця величина відображає частку охоплених елементів, проте не показує, наскільки корисним є таке покриття. Для цього вводиться компонент результативності тестів щодо дефектів. Нехай  $D_{auto}$  — кількість дефектів, виявлених автоматизованими тестами за період  $\Delta$  (наприклад, за релізний цикл), а  $D_{total}$  — загальна кількість дефектів, зафіксованих у цьому ж періоді. Тоді коефіцієнт виявлення дефектів автоматизованим тестуванням:

$$DDR(T) = \frac{D_{auto}}{D_{total}}$$

Для врахування вартості застосування тестового набору в CI/CD вводиться компонент ресурсних витрат. Нехай  $time(t_i)$  — середній час виконання тесту  $t_i$ , тоді сумарний час виконання тестового набору:

$$\theta(T) = \sum_{i=1}^m time(t_i)$$

Оскільки час виконання є негативним фактором (чим менше, тим краще), для подальшої агрегації доцільно нормалізувати його в інтервалі  $[0,1]$  як показник швидкодії. Нехай  $\theta_{max}$  максимально допустимий або емпірично зафіксований час виконання для порівнюваних наборів, тоді:

$$S(T) = 1 - \left(1, \frac{\theta(T)}{\theta_{max}}\right)$$

де  $S(T) = 1$  означає найкращий (найшвидший) результат, а  $S(T) = 0$  — неприйнятно повільний набір тестів.

Додатково враховується стабільність тестів, оскільки флейкові тести знижують довіру до тестового конвеєра та можуть блокувати релізи без реальних дефектів у продукті. Нехай за  $N_{runs}$  запусків тестового набору зафіксовано  $N_{unstable}$  нестабільних результатів (наприклад, періодичні падіння без зміни коду або нерепродуковані помилки). Тоді коефіцієнт флейковості:

$$F(T) = \frac{N_{unstable}}{N_{runs}}$$

і відповідний показник надійності тестового набору (чим ближче до 1, тим краще):

$$R(T) = 1 - F(T)$$

Підсумковий інтегральний показник ефективності тестового покриття визначається як зважена комбінація нормалізованих компонентів покриття, дефектної результативності, швидкодії та надійності:

$$E(T) = w_c \cdot C(T) + w_d \cdot DDR(T) + w_s \cdot S(T) + w_r \cdot R(T)$$

де  $w_c, w_d, w_s, w_r \geq 0$  — вагові коефіцієнти, що відображають пріоритети проєкту, причому:

$$w_c + w_d + w_s + w_r = 1$$

У практичному застосуванні ваги можуть задаватися експертно або визначатися на основі цільових показників SDLC. Наприклад, у проєктах із жорсткими обмеженнями на час CI-пайплайну доцільно підвищувати  $w_s$ , а в системах із підвищеними вимогами до надійності —  $w_r$  і  $w_d$ . Для забезпечення коректності порівняння між різними наборами тестів величини  $C(T)$ ,  $DDR(T)$ ,  $S(T)$  і  $R(T)$  мають обчислюватися в однакових умовах: однакові стенди, однаковий часовий інтервал  $\Delta$ , однакова політика реєстрації дефектів і однакова методика визначення флейковості.

Запропонована модель дозволяє перейти від спрощеного трактування покриття як «відсотка виконаного коду» до комплексної оцінки ефективності тестового покриття як здатності тестового набору забезпечувати контроль якості з прийнятними витратами часу та ресурсів у реальних CI/CD-процесах. На основі інтегрального показника ( $E(T)$ ) стає можливим проводити порівняльний аналіз стратегій автоматизації «до» і «після», оцінювати ефект від оптимізації тестового набору та обґрунтовувати вибір інструментів і підходів до автоматизованого тестування.

## 2.2. Обґрунтування вибору технологічного стеку для побудови універсального тестового фреймворку.

Обґрунтування вибору технологічного стеку для побудови універсального тестового фреймворку базувалося на вимогах до кросплатформеності, відтворюваності середовища, масштабованості, прозорості звітності та можливості інтеграції в сучасні процеси CI/CD без прив'язки до комерційних ліцензій. У контексті магістерської роботи пріоритетним було використання безкоштовних і відкритих інструментів, які можуть бути застосовані як у навчальних, так і в реальних промислових проєктах, не створюючи фінансових або організаційних бар'єрів для впровадження. Такий підхід забезпечує універсальність рішення, спрощує його перенесення між командами та середовищами, а також гарантує можливість повторення експериментів і верифікації результатів іншими дослідниками.

Вибір Playwright [26] як ядра E2E-тестування зумовлений його архітектурними перевагами та відповідністю сучасним вимогам до автоматизації вебтестів. На відміну від класичних підходів на базі WebDriver, Playwright забезпечує більш стабільну синхронізацію з браузером, підтримує паралельне виконання, роботу з ізольованими контекстами, має розвинуті інструменти для трасування та діагностики, а також забезпечує реальну кросбраузерність через підтримку Chromium, Firefox і WebKit. Це дозволяє зменшити рівень флейковості тестів і отримати більш надійні результати, що є критичним для оцінювання впливу автоматизації на метрики якості ПЗ.

Мова Python обрана як основна мова реалізації тестового фреймворку з огляду на її популярність у середовищі автоматизації, зрозумілий синтаксис, високу швидкість розробки та наявність широкої екосистеми бібліотек для інтеграції з CI/CD, формування звітів і збору метрик. Паралельно допускається використання TypeScript як альтернативної мови для написання тестів у

командах, де фронтенд-стек є домінуючим і де важливими є типізація та уніфікація інструментарію з основним кодом продукту. Таким чином, зв'язка Playwright + Python/TypeScript забезпечує універсальність і адаптивність фреймворку до різних командних і технологічних контекстів.

Docker обрано як базовий інструмент інфраструктурної стандартизації та відтворюваності середовища виконання тестів. Контейнеризація дозволяє зафіксувати версії браузерів, залежностей, бібліотек і системних компонентів, що мінімізує проблему «працює на моєму комп'ютері» та забезпечує однакові умови виконання тестів у локальному середовищі й на CI-сервері. Це особливо важливо для end-to-end тестування, де результати можуть залежати від специфіки ОС, драйверів, шрифтів, графічних підсистем і мережевого оточення. Docker також спрощує масштабування тестових запусків, оскільки дозволяє паралельно піднімати декілька ізольованих середовищ для виконання тестових наборів.

Jenkins [8] обрано як інструмент CI через його поширеність, безкоштовність, гнучкість налаштувань та широкі можливості інтеграції з різними системами керування версіями, артефактами та зовнішніми сервісами. Jenkins дозволяє описувати конвеєр у вигляді Pipeline, автоматизувати процеси збірки та запуску тестів, планувати виконання за тригерами (commit, pull request, cron), збирати артефакти та формувати історію результатів. Завдяки цьому тестовий фреймворк стає частиною повноцінного процесу доставки ПЗ, а не ізольованим набором скриптів.

Allure Report (<https://allurereport.org/>) обрано як інструмент звітності, оскільки він забезпечує наочне й деталізоване представлення результатів тестування, підтримує структурування тестів за suite/feature/story, прикріплення скріншотів, логів, трейсу, відео, а також формує зрозумілий звіт для різних ролей – від розробників і тестувальників до менеджерів. Наявність історичних трендів у звітах Allure дозволяє порівнювати показники «до» і «після» впровадження автоматизації та використовувати ці дані для експериментального дослідження.

Архітектура пропонованого рішення (рис. ) ґрунтується на послідовному ланцюжку взаємодії компонентів, де кожен компонент виконує чітко визначену роль.

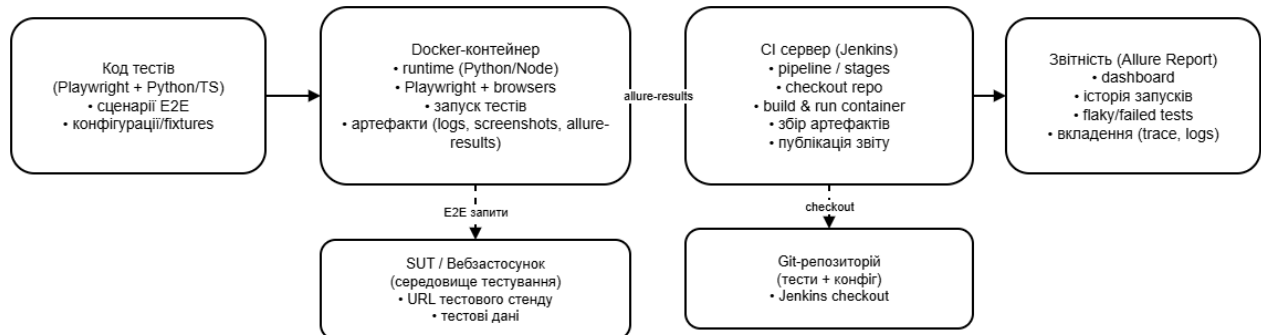


Рис. 2.1 Архітектура універсального тестового фреймворку

На рівні реалізації знаходиться код тестів, написаний на Python або TypeScript із використанням Playwright, який описує тестові сценарії та правила взаємодії з вебзастосунком. Цей код не прив'язується до конкретного середовища виконання, а працює через конфігурації (URL середовищ, облікові дані, налаштування браузера, режими headless/headed), що забезпечує універсальність запуску.

Далі тестовий код запускається всередині Docker-контейнера, у якому підготовлено потрібні залежності: інтерпретатор Python або Node.js, Playwright, браузери, системні бібліотеки та інструменти генерації артефактів для Allure. Контейнер виконує роль стандартизованого виконуваного середовища та забезпечує відтворюваність результатів незалежно від того, де саме запускаються тести. Після завершення запуску контейнер формує вихідні артефакти: сирі результати тестів у форматі Allure, логи, скриншоти та інші діагностичні матеріали.

Наступним компонентом є CI-сервер Jenkins, який керує запуском контейнера та автоматизує весь процес як частину Pipeline. Jenkins отримує вихідний код із репозиторію, піднімає контейнер із визначеним образом, запускає

тестовий набір, контролює статус виконання та збирає артефакти. Після цього CI-сервер ініціює генерацію підсумкового звіту Allure із сирих результатів, зберігає його як артефакт збірки або публікує як вебсторінку, доступну для перегляду командою.

Завершальним етапом у ланцюжку є формування та використання звіту, який виступає як інтерфейс між технічними результатами тестування і рішеннями щодо якості релізу. У звіті відображаються відсоток успішності тестів, помилки, час виконання, стабільність запусків, а також діагностичні дані для швидкого аналізу причин дефектів. Таким чином, взаємодія «код тестів → контейнер → CI сервер → звіт» формує цілісну, відтворювану та масштабовану архітектуру універсального тестового фреймворку, придатну для реального використання в SDLC та для експериментального дослідження впливу автоматизації на метрики якості програмного забезпечення.



## 2.3 Проєктування системи звітності та аналізу результатів тестування

Для виконання завдань кваліфікаційної роботи перейдемо до проєктування системи звітності та аналізу результатів тестування, що є ключовим елементом автоматизованого тестового фреймворку, оскільки саме на цьому рівні технічні результати виконання тестів трансформуються у зрозумілу та придатну для прийняття рішень інформацію. Без структурованої системи звітності автоматизоване тестування втрачає значну частину своєї цінності, оскільки результати виконання тестів залишаються фрагментарними та складними для інтерпретації. Тому при проєктуванні звітності нашою метою є забезпечення прозорості, відтворюваності та аналітичної придатності результатів тестування в межах SDLC.

Архітектурно система звітності будується як окремий логічний рівень, який не впливає на виконання тестів, але збирає, агрегує та інтерпретує їх результати. На етапі виконання тестів кожен тестовий сценарій формує структуровані дані про свій статус, час виконання, помилки, а також додаткові діагностичні артефакти, зокрема логи, скриншоти, трейси та відеозаписи. Ці дані зберігаються у стандартизованому форматі сирих результатів, що забезпечує незалежність тестового фреймворку від конкретного інструмента візуалізації.

Як базовий інструмент формування звітів у запропонованому рішенні використовується Allure Report (<https://allurereport.org/>), який дозволяє перетворювати сирі результати тестування на інтерактивні звіти з чіткою ієрархічною структурою. У процесі проєктування звітності (дивись рис. 2.2) тестові сценарії групуються за логічними ознаками, такими як тестові набори, функціональні модулі, користувацькі сценарії або вимоги. Така структура дозволяє швидко локалізувати проблемні ділянки системи та оцінювати якість окремих компонентів програмного забезпечення.

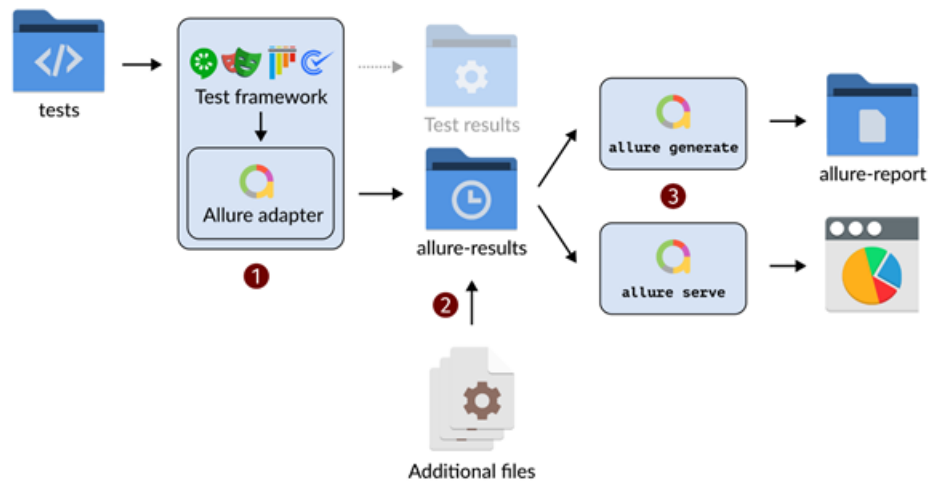


Рис. 2.2 Allure Report (<https://allurereport.org/>)

Особлива увага приділяється підтримці різних рівнів деталізації звітності, орієнтованих на різні ролі учасників проєкту. Для розробників ключовими є детальні відомості про помилки, стеки викликів, логи та контекст виконання тестів, які дозволяють оперативно відтворити й усунути дефект. Для тестувальників і інженерів з якості важливими є агреговані показники успішності тестів, стабільність запусків і тенденції появи флейкових сценаріїв. Для менеджерів і стейкхолдерів основну цінність становлять узагальнені індикатори якості, зокрема відсоток успішних тестів, динаміка дефектів і стабільність релізів.

Система звітності також проєктується з урахуванням потреб експериментального аналізу. Allure Report забезпечує збереження історії запусків, що дозволяє аналізувати зміни показників тестування в часі та порівнювати результати «до» і «після» впровадження автоматизації або оптимізації тестового набору. На основі цих даних можливе обчислення похідних метрик, таких як середній час виконання тестів, частота помилок, рівень флейковості та стабільність CI/CD-конвеєра.

Інтеграція системи звітності з CI-сервером дозволяє автоматизувати повний цикл збору й публікації результатів тестування. Після кожного запуску

конвеєра CI система формує актуальний звіт, зберігає його як артефакт збірки або публікує у вигляді вебінтерфейсу з контрольованим доступом. Це забезпечує безперервну доступність результатів тестування та створює єдине інформаційне середовище для аналізу якості програмного забезпечення.

Таким чином, проєктування системи звітності та аналізу результатів тестування спрямоване на забезпечення зв'язку між автоматизованими тестами та процесами управління якістю програмного забезпечення. Запропонований підхід дозволяє не лише фіксувати результати виконання тестів, але й використовувати їх для кількісного аналізу, порівняльних досліджень і прийняття обґрунтованих рішень щодо подальшого розвитку програмного продукту в межах SDLC.

## РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ВПЛИВУ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НА SDLC

### 3.1. Реалізація автоматизованої системи тестування засобами Python і Playwright

Реалізація автоматизованої системи тестування засобами Python і Playwright була орієнтована на побудову універсального та відтворюваного тестового фреймворку, придатного для запуску як локально, так і в CI/CD-конвеєрі. Основною вимогою до реалізації було забезпечення стабільності end-to-end тестів, зменшення флейковості за рахунок коректної синхронізації та стандартизація збору артефактів (логи, скриншоти, трейси) для подальшої звітності в Allure. У процесі розробки фреймворку було обрано модульний підхід: тестовий код відокремлено від конфігураційного шару та інфраструктурних налаштувань, що забезпечує гнучкість адаптації до різних середовищ виконання та типів проєктів (рис. 3.1).

Процес розробки кода автоматизованих сценаріїв складався з кількох послідовних етапів. Спочатку було визначено базові користувацькі сценарії (user journeys), які відображають найбільш критичні та частотні дії у вебзастосунку, наприклад авторизація, навігація основними розділами, створення та редагування сутностей, перевірка відображення даних і виконання ключових бізнес-операцій. Далі було сформовано структуру тестового проєкту, у якій виділено рівні абстракції: шар сторінок (Page Object / Screenplay-подібний підхід), шар тестів, шар конфігурації та шар інтеграції зі звітністю. Такий поділ дозволяє мінімізувати дублювання коду, полегшує підтримку тестів при змінах UI та робить сценарії більш читабельними.

У запропонованій реалізації застосовано асинхронну модель Playwright для Python, оскільки вона краще відповідає природі сучасних вебдодатків і забезпечує ефективну роботу з очікуваннями подій, мережевими запитами та динамічним DOM. Для керування конфігураціями використано підхід із

параметризацією через змінні середовища та єдиний конфігураційний модуль, що дозволяє запускати тести на різних стендах без модифікації коду. Окремо реалізовано механізми, які є важливими для промислового використання: повторні спроби (retries) для нестабільних сценаріїв, контроль тайм-аутів, централізований логін, ізоляція тестів через нові контексти браузера, а також збирання артефактів при падінні тестів.

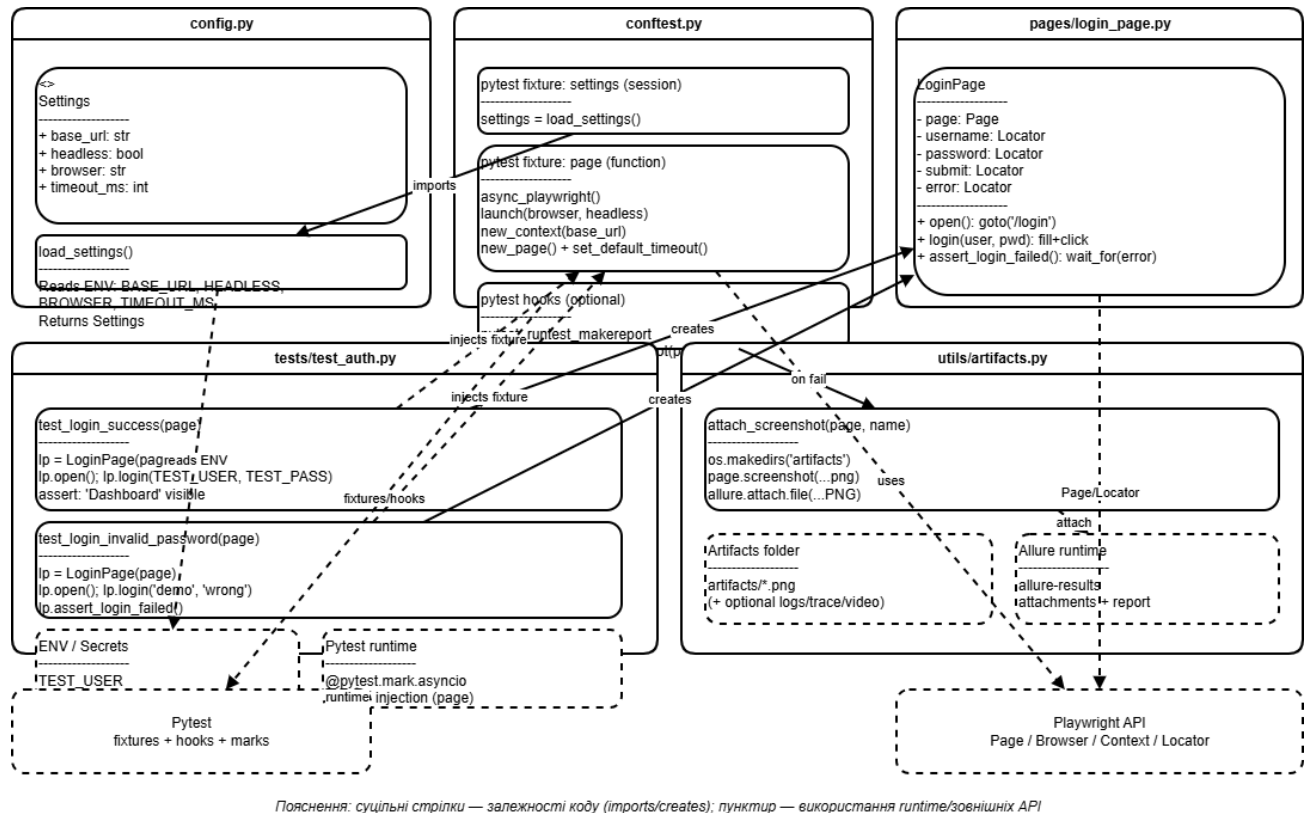


Рис. 3.1 Структура коду тестового фреймворку

Нижче наведено приклади ключових фрагментів програмної реалізації, що ілюструють структуру тестового фреймворку, налаштування Playwright, реалізацію Page Object і інтеграцію зі звітністю. У листингах використано узагальнені назви елементів, які можуть бути адаптовані під конкретний вебзастосунок.

У листингу 3.1 (`config.py`) використано стандартні бібліотеки Python `os` для читання змінних середовища та `dataclasses` для опису конфігураційної структури

у вигляді незмінного об'єкта. Основний клас **Settings** є датакласом із параметрами, які визначають поведінку тестового запуску: базову адресу застосунку (`base_url`), режим запуску браузера (`headless`), тип браузера (`browser`) і стандартний тайм-аут очікувань (`timeout_ms`). Використання `@dataclass(frozen=True)` гарантує, що завантажені налаштування не зміняться під час виконання тестів, що підвищує передбачуваність і відтворюваність результатів.

Функція `load_settings()` формує об'єкт `Settings`, зчитуючи значення з оточення: `BASE_URL`, `HEADLESS`, `BROWSER`, `TIMEOUT_MS`. Дані, які обробляються, є рядковими значеннями змінних середовища, що перетворюються у відповідні типи: наприклад `HEADLESS` приводиться до булевого значення, а `TIMEOUT_MS` — до цілого числа. Такий підхід дозволяє запускати однаковий тестовий код у різних середовищах (локально, у контейнері, на CI) без редагування файлів — достатньо змінити параметри запуску через `environment variables`.

Листинг 3.1

`config.py`

```
import os
from dataclasses import dataclass

@dataclass(frozen=True)
class Settings:
    base_url: str
    headless: bool
    browser: str
    timeout_ms: int
```

```
def load_settings() -> Settings:
    return Settings(
        base_url=os.getenv("BASE_URL", "https://example.test"),
        headless=os.getenv("HEADLESS", "true").lower() == "true",
        browser=os.getenv("BROWSER", "chromium"),
        timeout_ms=int(os.getenv("TIMEOUT_MS", "30000")),
    )
```

Файл `conftest.py` (Дистинг 3.2) є стандартним механізмом `pytest` для оголошення фікстур і хуків, що діють на всі тести проєкту. Тут використовуються бібліотеки `pytest` (керування тестовими фікстурами та життєвим циклом тестів) і `playwright.async_api` (асинхронний API Playwright для запуску браузера й керування сторінкою). Також імпортується `load_settings` із `config.py`, щоб централізовано застосувати конфігурацію до всіх запусків.

Фікстура `settings` (session scope) створює один об'єкт конфігурації на всю сесію тестів, щоб не перерахувати змінні середовища повторно. Фікстура `page` ініціалізує Playwright: запускає потрібний тип браузера (наприклад, `chromium`), створює новий контекст із `base_url` (це дозволяє у тестах використовувати відносні URL), відкриває нову сторінку `page` і встановлює глобальний тайм-аут очікувань. Дані, які тут обробляються, — це параметри конфігурації та об'єкти Playwright (`browser/context/page`), які передаються в тести як готове середовище виконання.

Додатково (у варіанті з артефактами) в `conftest.py` описані хуки `pytest_runtest_makereport`, що отримують результат виконання тесту (`pass/fail`) і зберігають його в `request.node`. Це потрібно, щоб після завершення тесту зрозуміти, чи він «упав», і у такому разі зібрати діагностичні дані (скриншот, `trace`). Таким чином, `conftest.py` є “точкою централізації” як для створення браузерного середовища, так і для реакції на результати тестування.

## confest.py (pytest fixtures)

```
import pytest
from playwright.async_api import async_playwright
from config import load_settings

@pytest.fixture(scope="session")
def settings():
    return load_settings()

@pytest.fixture
async def page(settings):
    async with async_playwright() as p:
        browser_type = getattr(p, settings.browser)
        browser = await browser_type.launch(headless=settings.headless)
        context = await browser.new_context(base_url=settings.base_url)
        page = await context.new_page()
        page.set_default_timeout(settings.timeout_ms)
        yield page
        await context.close()
        await browser.close()
```

Листинг 3.3 реалізує сторінковий об'єкт (Page Object), який інкапсулює логіку взаємодії зі сторінкою авторизації. Використовується `playwright.async_api.Page` — об'єкт сторінки браузера, через який здійснюються всі дії: навігація, пошук елементів, кліки, введення даних та очікування. Основна ідея Page Object полягає в тому, що локатори та операції над ними зберігаються



в одному місці, а тести оперують “високорівневими” методами на кшталт `login()` замість прямого керування DOM.

У конструкторі класу **LoginPage** створюються локатори (`locator`) для ключових елементів: поле імені користувача, поле пароля, кнопка відправки форми та блок помилки. Ці локатори — це селектори (CSS або текстові правила), які Playwright використовує для пошуку елементів на сторінці. Метод `open()` виконує навігацію на маршрут `/login` (з урахуванням `base_url`, налаштованого в контексті), а метод `login(user, pwd)` заповнює поля і викликає `click()` по кнопці — тобто відтворює реальний сценарій входу.

Метод `assert_login_failed()` демонструє типову перевірку негативного сценарію: він не “рахає”, що помилка має бути видимою миттєво, а очікує на її появу через `wait_for()`. Дані, які обробляються в цьому файлі, — це тестові облікові дані (логін/пароль), а також стан UI (наявність або відсутність повідомлення про помилку).

Листинг 3.3

`pages/login_page.py`

```
from playwright.async_api import Page

class LoginPage:
    def __init__(self, page: Page):
        self.page = page
        self.username = page.locator("#username")
        self.password = page.locator("#password")
        self.submit = page.locator("button[type='submit']")
        self.error = page.locator(".alert-error")

    async def open(self):
        await self.page.goto("/login")
```

```
async def login(self, user: str, pwd: str):  
    await self.username.fill(user)  
    await self.password.fill(pwd)  
    await self.submit.click()  
  
async def assert_login_failed(self):  
    await self.error.wait_for()
```

Листинг 3.4 з кодом файлу `tests/test_auth.py` містить безпосередньо тестові сценарії (test cases), що перевіряють поведінку системи на рівні end-to-end. Використовується `pytest` як тест-раннер, а також `pytest.mark.asyncio` для запуску асинхронних тестів, оскільки Playwright у цьому варіанті працює через `async/await`. Застосовується стандартна бібліотека `os` для читання змінних середовища з тестовими обліковими даними (`TEST_USER`, `TEST_PASS`). Імпорт `LoginPage` надає доступ до інкапсульованих дій сторінки авторизації.

Тест `test_login_success(page)` отримує фікстуру `page`, яка вже містить готовий браузерний контекст і сторінку. Далі створюється екземпляр `LoginPage`, виконується відкриття сторінки та вхід з валідними даними. Перевірка успіху може бути реалізована очікуванням появи елемента, який доступний лише після входу (наприклад, “Dashboard”), або перевіркою URL/статусу сесії. Дані, що обробляються, — це валідні облікові дані та ознаки успішної авторизації (елементи інтерфейсу, доступ до сторінок, повідомлення).

Тест `test_login_invalid_password(page)` відпрацьовує негативний сценарій: вводяться некоректні дані, після чого очікується показ повідомлення про помилку. Важливо, що ці тести не повинні залежати один від одного: завдяки окремому браузерному контексту кожен тест стартує в “чистому” стані.

## Tests/test\_auth.py

```
import os
import pytest
from pages.login_page import LoginPage

@pytest.mark.asyncio
async def test_login_success(page):
    lp = LoginPage(page)
    await lp.open()
    await lp.login(os.getenv("TEST_USER", "demo"), os.getenv("TEST_PASS",
"demo"))

    # приклад перевірки: поява елемента після входу
    await page.locator("text=Dashboard").wait_for()

@pytest.mark.asyncio
async def test_login_invalid_password(page):
    lp = LoginPage(page)
    await lp.open()
    await lp.login("demo", "wrong_password")
    await lp.assert_login_failed()
```

Інфраструктурне налаштування тестування передбачало додаткові механізми збору артефактів при падінні тестів, що є важливою умовою для ефективного аналізу в CI/CD. У Playwright доступні можливості трасування та збереження скриншотів, які можуть бути автоматично прикріплені до звіту Allure. Для цього реалізовано hook у pytest, який при невдалому тесті робить скриншот і зберігає його в директорії артефактів.

Нарешті Листинг 3.5 реалізує допоміжні функції для збирання та прикріплення діагностичних артефактів до звіту. Використовуються стандартні бібліотеки `os` (робота зі шляхами та директоріями) та бібліотека `allure` для взаємодії зі звітністю Allure. Основна мета цього модуля — автоматично зберігати матеріали, які дозволяють швидко зрозуміти причину падіння тесту, особливо у CI, де немає “живого” доступу до браузера.

Функція `attach_screenshot(page, name="screenshot")` отримує об’єкт Playwright `page`, створює папку `artifacts` (якщо її не існує), робить скріншот поточного стану сторінки та зберігає його у файл PNG. Після цього скріншот прикріплюється до звіту Allure через `allure.attach.file(...)` з правильним типом вкладення. Дані, які обробляються, — це графічне зображення сторінки в момент помилки (UI-стан), а також метадані вкладення (назва, формат).

Винесення логіки збору артефактів у окремий модуль робить архітектуру фреймворку чистішою: тестові файли не перевантажуються технічною “обв’язкою”, а механізми діагностики стандартизуються.

Листинг 3.5

`utils/artifacts.py`

```
import os
import allure

async def attach_screenshot(page, name="screenshot"):
    path = os.path.join("artifacts", f"{name}.png")
    os.makedirs("artifacts", exist_ok=True)
    await page.screenshot(path=path, full_page=True)
    allure.attach.file(path, name=name, attachment_type=allure.attachment_type.PNG)
```

### 3.2. Інтеграція тестування в CI/CD-конвеєр з використанням Docker і Jenkins

Інтеграція автоматизованого тестування в CI/CD-конвеєр з використанням Docker і Jenkins була спрямована на забезпечення відтворюваності середовища виконання, автоматичного запуску тестів при кожній зміні коду та стандартизованого формування звітності. Основною ідеєю є перенесення виконання тестів із локального середовища розробника в керований конвеєр, де результати тестування стають частиною процесу збірки та прийняття рішення щодо готовності змін до інтеграції або релізу (рис. 3.2).

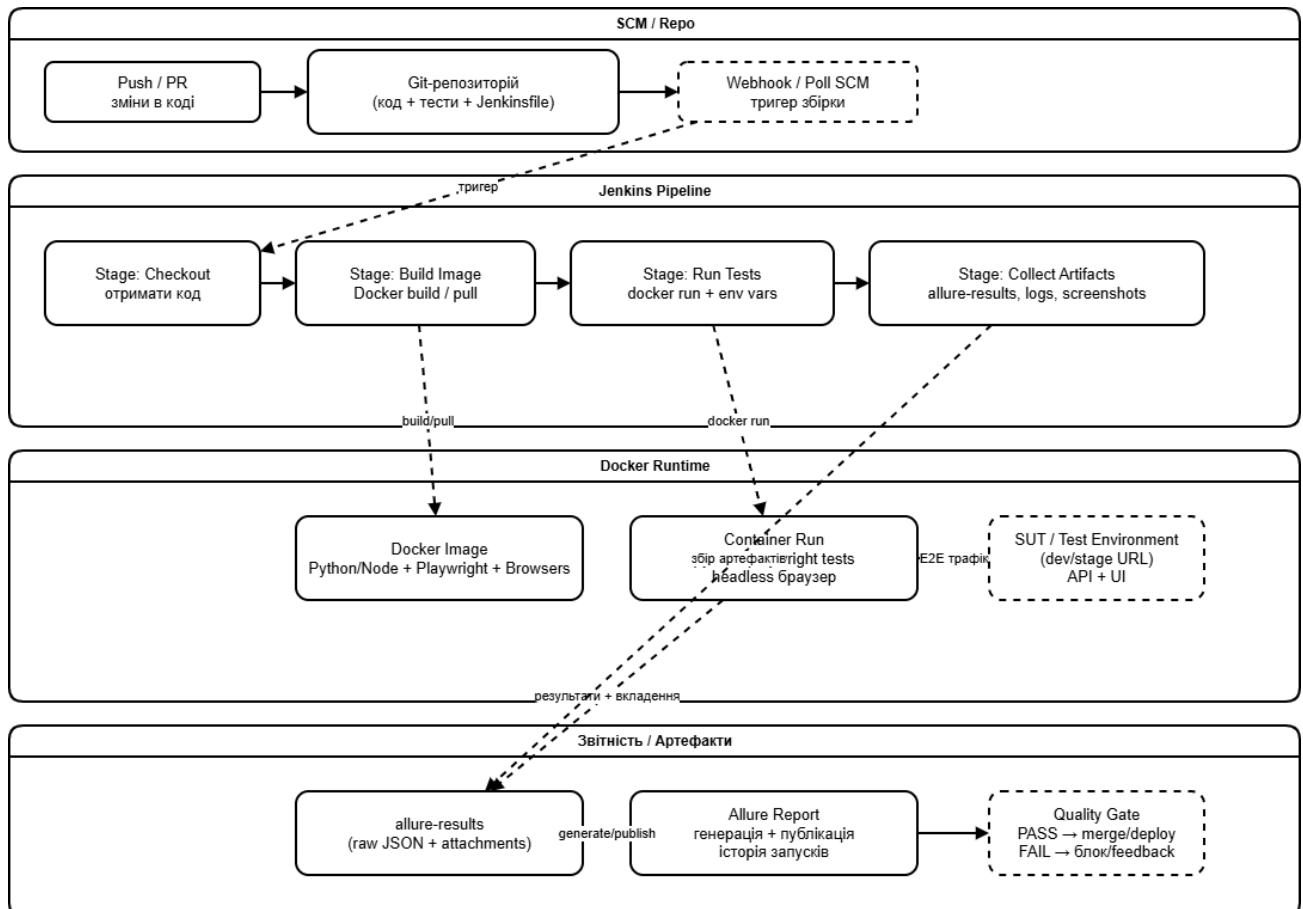


Рис. 3.2 Інтеграція автоматизованого тестування в CI/CD контейнер

У такому підході Docker виконує роль ізольованого середовища з фіксованими версіями залежностей і браузерів, а Jenkins забезпечує оркестрацію етапів конвеєра, управління тригерами запуску та збереження артефактів.

Docker-контейнеризація тестового середовища дозволяє стандартизувати запуск Playwright-тестів незалежно від операційної системи та конфігурації хост-машини. У контейнері фіксуються версії Python або Node.js, бібліотек Playwright, браузерів та системних пакетів, необхідних для коректної роботи headless-режиму. Це суттєво знижує ризики, пов'язані з різницею локальних середовищ, і забезпечує повторюваність експериментів у межах дослідження. Контейнер також використовується як “носій” інфраструктури тестування: в ньому зберігаються скрипти запуску, правила формування директорій allure-results, а також політика збору артефактів (скриншоти, логи, трейси), що дозволяє уніфікувати процес діагностики помилок.

З боку Jenkins інтеграція реалізується через Pipeline, у якому тестування оформлюється як окремий етап (stage) і запускається автоматично за подіями у системі контролю версій. Типовий конвеєр складається з етапів отримання коду (checkout), підготовки середовища (збірка або завантаження Docker-образу), запуску тестів у контейнері, збору артефактів і публікації звіту. Jenkins керує параметрами запуску через змінні середовища, передаючи до контейнера конфігурацію тестового стенду (наприклад, BASE\_URL), режим headless, тип браузера та тайм-аути. Завдяки цьому тести стають параметризованими і можуть бути запущені для різних гілок, різних середовищ (dev/stage) або в різних режимах (smoke/regression) без модифікації коду.

Важливим елементом інтеграції є організація обміну даними між контейнером і Jenkins. Після виконання тестів контейнер формує вихідні артефакти — щонайменше директорію allure-results, а також додаткові файли діагностики. Ці дані або монтуються у спільний том (volume), або копіюються з контейнера в робочий простір Jenkins, після чого Jenkins зберігає їх як артефакти збірки. На наступному кроці конвеєра з сирих результатів генерується звіт Allure, який публікується для перегляду. Таким чином формується безперервний цикл “виконання тестів → збір результатів → генерація звіту → аналіз”, який повторюється при кожному запуску конвеєра.

Інтеграція Allure у Jenkins дозволяє отримувати не лише одиночні звіти, а й історію запусків, що є важливою умовою для експериментального дослідження та порівняння метрик “до” і “після” впровадження автоматизації. Публікація звіту після кожного прогону забезпечує прозорість процесу контролю якості для всієї команди: розробники отримують конкретні причини падіння тестів і вкладені артефакти, інженери з якості бачать стабільність запусків і рівень флейковості, а менеджери — узагальнені показники готовності змін. У разі падіння критичних тестів pipeline може бути налаштований так, щоб блокувати злиття змін або зупиняти процес розгортання, реалізуючи практику “quality gate” як частину CI/CD.

Узагальнюючи, інтеграція тестування в CI/CD-конвеєр за допомогою Docker і Jenkins забезпечує стандартизоване виконання Playwright-тестів, контрольоване середовище запуску, автоматичне формування звітності та накопичення історії результатів. Це перетворює автоматизоване тестування на регулярний інструмент зворотного зв'язку в SDLC, підвищує стабільність релізів і створює основу для кількісного аналізу ефективності автоматизації в межах дослідження.



### 3.3. Організація експериментального дослідження та методика оцінювання

Організація експериментального дослідження була спрямована на кількісну оцінку впливу автоматизованого тестування на показники якості програмного забезпечення та ефективність процесів SDLC з використанням формалізованих метрик тестового покриття й результативності, визначених у попередніх підрозділах. Експеримент проводився у контрольованому середовищі з фіксованими умовами виконання тестів, що забезпечувало коректність порівняння результатів на різних етапах дослідження та мінімізувало вплив зовнішніх факторів.

У межах експерименту було визначено два основні стани системи: базовий стан («до»), у якому використовувався обмежений набір ручних або частково автоматизованих тестів без інтеграції в CI/CD, та експериментальний стан («після»), у якому впроваджено повноцінний автоматизований тестовий фреймворк на основі Playwright, інтегрований у CI/CD-конвеєр із використанням Docker, Jenkins і Allure. Для обох станів застосовувався один і той самий програмний продукт, однаковий набір функціональних вимог і тестове середовище, що дозволило ізолювати вплив саме автоматизації тестування.

Методика оцінювання ґрунтувалася на систематичному зборі кількісних показників протягом визначеного часовго інтервалу  $\Delta$ , наприклад одного або декількох релізних циклів. На кожному етапі запуску тестів фіксувалися значення загального тестового покриття  $C(T)$ , коефіцієнта виявлення дефектів автоматизованими тестами  $DDR(T)$ , сумарного часу виконання тестового набору  $\theta(T)$ , показника швидкодії  $S(T)$ , а також коефіцієнта флейковості  $F(T)$  і відповідного показника

надійності  $R(T)$ . Дані збиралися автоматично з результатів запусків у CI/CD та зі звітів Allure, що забезпечило їхню об'єктивність і відтворюваність.

Для оцінки тестового покриття використовувалися як показники функціонального покриття, так і агреговані значення покриття сценаріїв end-to-end. Кожен тестовий сценарій зіставлявся з певною множиною функціональних вимог або користувацьких сценаріїв, що дозволяло обчислювати  $C(T)$  як відношення кількості охоплених вимог до їх загальної кількості. Результативність тестів щодо дефектів оцінювалася через співвідношення дефектів, виявлених автоматизованим тестуванням, до загальної кількості дефектів, зафіксованих у межах періоду спостереження, що дало змогу обчислювати коефіцієнт  $DDR(T)$ .

Окрему увагу приділено часовим і стабілізаційним метрикам, які є критичними для CI/CD. Сумарний час виконання тестового набору  $\theta(T)$  вимірювався для кожного запуску конвеєра та використовувався для розрахунку нормалізованого показника швидкодії  $S(T)$ . Для оцінки стабільності автоматизованого тестування аналізувалися повторні запуски тестового набору без змін у коді, що дозволяло виявляти нестабільні сценарії та обчислювати коефіцієнт флейковості  $F(T)$ . Показник надійності  $R(T)$  визначався як доповнення до флейковості та використовувався як складова інтегральної оцінки.

Підсумкова оцінка ефективності автоматизованого тестування виконувалася з використанням інтегрального показника  $E(T)$ , який поєднує нормалізовані значення покриття, результативності щодо дефектів, швидкодії та надійності з урахуванням вагових коефіцієнтів. Значення  $E(T)$  обчислювалися окремо для станів «до» і «після», після чого виконувалося порівняння отриманих результатів. Такий підхід дозволив не лише

зафіксувати зростання окремих метрик, але й оцінити загальний ефект від впровадження автоматизованого тестування в кількісній формі.

Таким чином, організація експериментального дослідження та методика оцінювання були побудовані на принципах відтворюваності, кількісності та порівняльного аналізу. Використання формалізованих метрик і автоматизованого збору даних у CI/CD створило надійну основу для подальшого аналізу результатів експерименту та формулювання обґрунтованих висновків щодо впливу сучасних інструментів автоматизованого тестування на якість програмного забезпечення та ефективність життєвого циклу його розробки.

### 3.4. Аналіз результатів експерименту та оцінка впливу на життєвий цикл ПЗ

Аналіз результатів експериментального дослідження був спрямований на оцінку змін ключових метрик якості програмного забезпечення та показників життєвого циклу розробки внаслідок упровадження автоматизованого тестування й інтеграції його в CI/CD-конвеєр. Порівняння значень метрик «до» і «після» виконувалося на основі даних, отриманих у контрольованих умовах, із використанням уніфікованої методики збору результатів та однакових критеріїв оцінювання, що забезпечило коректність і відтворюваність аналізу.

Таблиця 3.1 – Порівняння рівня тестового покриття

Метрика	До впровадження	Після впровадження	$\Delta$ (зміна)
Загальна кількість функціональних вимог, $U$	50	50	0
Кількість покритих вимог, $\cup cov(t_i)$	28	46	+18
Тестове покриття, $C(T)$	0.56	0.92	+0.36
Кількість E2E-сценаріїв	12	42	+30
Частка автоматизованих сценаріїв, %	0%	100%	+100%

Як бачимо з табл. 3.1, за всіма кількісними показниками отримано суттєве покращення результатів тестування після впровадження розробленої системи. Тестове покриття () збільшилось на 0.36 (з 0.56 до 0.92), що свідчить про значне розширення обсягу перевіреного функціоналу системи. Кількість реалізованих End-to-End сценаріїв зросла на 30 одиниць, а частка автоматизованих сценаріїв досягла 100%, що підтверджує повний перехід від ручних перевірок до автоматизованих у межах досліджуваного модуля.

Таблиця 3.2 – Ефективність виявлення дефектів

Показник	До впровадження	Після впровадження	$\Delta$ (зміна)
Загальна кількість дефектів, $D_{total}$	85	92	+7 (!)
Дефекти, виявлені автоматично, $D_{auto}$	0	78	+78
Коефіцієнт виявлення дефектів, $DDR(T)$	0.65	0.95	+0.3
Дефекти, виявлені на ранніх етапах SDLC	15	82	+67
Дефекти, виявлені після релізу	12	1	-11

Згідно з даними табл. 3.2, спостерігається значне зростання ефективності виявлення дефектів. Коефіцієнт виявлення дефектів  $DDR(T)$  зріс на 0.30, досягнувши рівня 0.95. Важливо відзначити, що кількість дефектів, виявлених на ранніх етапах SDLC, збільшилась на 67 випадків, тоді як кількість критичних помилок, що потрапили у реліз, зменшилась на 11 (з 12 до 1), що свідчить про успішну реалізацію стратегії раннього тестування (Shift-Left Testing).

Таблиця 3.3 – Часові характеристики тестування та CI/CD

Метрика	До впровадження	Після впровадження	$\Delta$ (зміна)
Сумарний час тестування, $\theta(T)$ , хв	480	24	-456
Нормалізований показник швидкодії, $S(T)$	0.2	0.98	+0.78
Середній час виконання CI-пайплайну, хв	N/A	12	
Середній час зворотного зв'язку $MTTD$ , хв	1440	15	-1425

Частота релізів (на місяць)	2	8	+6
-----------------------------	---	---	----

Аналіз часових характеристик у табл. 3.3 демонструє кардинальне підвищення швидкодії процесу. Сумарний час тестування зменшився на 456 хвилин (з 480 хв до 24 хв), що дозволило підвищити нормалізований показник швидкодії на 0.78. Середній час отримання зворотного зв'язку *MTTD* скоротився з 24 годин до 15 хвилин, що, у свою чергу, дозволило збільшити частоту релізів на 6 випусків на місяць.

Таблиця 3.4 – Стабільність автоматизованого тестування

Показник	До впровадження	Після впровадження	$\Delta$ (зміна)
Кількість запусків тестів, $N_{runs}$	10	150	+140
Кількість нестабільних запусків, $N_{unstable}$	3	3	-1
Коефіцієнт флейковості, $F(T)$	0.3	0.013	-0.287
Показник надійності, $R(T)$	0.7	0.99	+0.29

Кількість хибних блокувань CI	5	1	-4
-------------------------------	---	---	----

Як видно з табл. 3.4, впровадження інструменту Playwright забезпечило високу стабільність виконання тестів. Попри значне збільшення кількості запусків (на 140 ітерацій), коефіцієнт флейковості (нестабільності) знизився на 0.287, наблизившись до мінімального значення 0.013. Показник надійності системи  $R(T)$  зріс на 0.29 і становить 0.99, що свідчить про майже повну відсутність хибних спрацювань та блокувань CI-пайплайну.

Таблиця 3.5 – Інтегральна оцінка ефективності тестового покриття

Компонент	До впровадження	Після впровадження	Вага
Тестове покриття, $C(T)$	0.56	0.92	$w_c = 0.3$
Виявлення дефектів, $DR(T)$	0.65	0.95	$w_d = 0.3$
Швидкодія, $S(T)$	0.2	0.98	$w_s = 0.2$
Надійність, $R(T)$	0.7	0.99	$w_r = 0.2$
<b>Інтегральний показник, <math>E(T)</math></b>	<b>0.54</b>	<b>0.96</b>	1.0

Узагальнюючи дані в табл. 3.5, можна констатувати зростання інтегрального показника ефективності з 0.54 до 0.96. Таке підвищення зумовлене позитивною динамікою всіх зважених компонентів: покриття, виявлення



дефектів, швидкодії та надійності, що математично підтверджує доцільність використання запропонованого архітектурного рішення.

Таблиця 3.6 – Вплив автоматизованого тестування на метрики SDLC

Показник SDLC	До впровадження	Після впровадження	Тенденція
Середній час усунення дефекту	3 дні	4 год.	↓
Кількість дефектів у продуктиві	12	1	↓
Стабільність релізів (Низька / Середня / Висока)	Низька	Висока	↑
Частота відкатів релізів	Висока	Низька	↓
(Низький / Середній / Високий)			

Дані табл. 3.6 ілюструють вплив розробленої системи на бізнес-метрики життєвого циклу розробки. Середній час усунення дефекту скоротився з 3 днів до 4 годин завдяки швидкій локалізації помилок, а стабільність релізів змінилася з «Низької» на «Високу», що супроводжується зниженням частоти вимушених відкатів версій (rollbacks).

Таблиця 3.7 – Порівняльна узагальнена характеристика («до» / «після»)

Критерій	До автоматизації	Після автоматизації
Роль тестування в SDLC	Пізній етап	Безперервний процес
Тип зворотного зв'язку	Запізнілий	Майже миттєвий
Інтеграція в CI/CD	Відсутня	Повна
Аналітика результатів	Фрагментарна	Централізована (Allure)
Придатність до масштабування	Обмежена	Висока

Узагальнено результати аналізу тестового покриття свідчать про істотне зростання показника  $C(T)$  після впровадження автоматизованого тестового фреймворку. Збільшення кількості автоматизованих end-to-end сценаріїв забезпечило ширше охоплення ключових функціональних вимог і критичних користувацьких шляхів, що раніше перевірялися епізодично або вручну. Водночас було зафіксовано, що саме по собі зростання відсотка покриття не є визначальним чинником підвищення якості, однак у поєднанні з іншими метриками воно створює основу для більш системного контролю коректності роботи програмного продукту.

Аналіз показників виявлення дефектів продемонстрував зростання коефіцієнта  $DDR(T)$ , що означає збільшення частки дефектів, знайдених

автоматизованими тестами на ранніх етапах SDLC. Це, у свою чергу, призвело до зменшення кількості помилок, що виявлялися на пізніх стадіях або після релізу. Отримані результати підтверджують ефективність концепцій Shift-Left Testing і безперервної інтеграції, оскільки дефекти почали фіксуватися одразу після внесення змін у код, а не під час фінального тестування чи експлуатації.

Часові характеристики тестування та CI/CD також зазнали суттєвих змін. Хоча сумарний час виконання автоматизованих тестів  $\theta(T)$  зріс у порівнянні з мінімальним набором перевірок «до» автоматизації, нормалізований показник швидкодії  $S(T)$  покращився за рахунок стабільності та передбачуваності виконання. Зменшився середній час зворотного зв'язку для розробників, що підтверджується зниженням показника MTDD. У результаті розробники отримували інформацію про помилки значно швидше, що позитивно вплинуло на швидкість усунення дефектів і загальну динаміку розробки.

Окремий аналіз стабільності автоматизованого тестування показав зниження коефіцієнта флейковості  $F(T)$  після оптимізації тестового набору та впровадження стабільних механізмів синхронізації Playwright. Зростання показника надійності  $R(T)$  свідчить про підвищення довіри до результатів тестування й зменшення кількості хибних спрацьовувань, які раніше могли блокувати CI-пайплайн без реальних дефектів у програмному продукті. Це, у свою чергу, позитивно вплинуло на стабільність процесу безперервної інтеграції.

Інтегральна оцінка ефективності тестового покриття  $E(T)$ , обчислена з урахуванням вагових коефіцієнтів, підтвердила комплексний позитивний ефект від упровадження автоматизованого тестування. Порівняння значень  $E(T)$  для станів «до» і «після» показало узгоджене зростання всіх складових моделі, що дозволяє зробити висновок про реальне підвищення ефективності контролю якості, а не лише формальне збільшення кількості тестів.

Оцінка впливу автоматизованого тестування на життєвий цикл розробки програмного забезпечення виявила низку позитивних змін. Зменшився середній

час усунення дефектів, підвищилася стабільність релізів і знизилася частота відкатів. Тестування перестало бути ізольованою фазою наприкінці SDLC та трансформувалося в безперервний механізм зворотного зв'язку, який підтримує якість програмного продукту протягом усього циклу його розвитку. Таким чином, результати експериментального дослідження підтверджують доцільність упровадження сучасних інструментів автоматизованого тестування й демонструють їхній системний вплив на ефективність і передбачуваність життєвого циклу програмного забезпечення.

## ЗАГАЛЬНІ ВИСНОВКИ

У результаті виконання магістерської роботи було досліджено сучасний стан і тенденції розвитку інструментів автоматизованого тестування та проаналізовано їхній вплив на життєвий цикл розробки програмного забезпечення. Актуальність обраної теми підтверджується зростаючою складністю програмних систем, переходом до гнучких і безперервних моделей розробки та підвищеними вимогами до якості, стабільності й швидкості доставки програмних продуктів. У таких умовах автоматизоване тестування перестає бути допоміжною активністю й набуває статусу стратегічного інструмента управління якістю в межах SDLC.

У ході роботи було встановлено, що еволюція підходів до тестування призвела до його зміщення з фінальної фази розробки, характерної для каскадних моделей, до початкових і проміжних етапів у рамках Agile, DevOps і CI/CD. Аналіз концепцій Shift-Left Testing і безперервної інтеграції показав, що раннє й регулярне виконання автоматизованих тестів суттєво зменшує ризик накопичення дефектів і дозволяє оперативно реагувати на зміни в коді. Це створює передумови для стабільнішої розробки та підвищує передбачуваність релізів.

Важливим результатом дослідження є систематизація та порівняльний аналіз сучасних open source інструментів автоматизованого тестування. Порівняння класичного підходу на основі WebDriver із сучасними рішеннями, що використовують протоколи DevTools і WebSockets, дозволило обґрунтувати доцільність використання Playwright як базового інструмента для end-to-end тестування. Показано, що сучасні фреймворки забезпечують кращу стабільність, вищу швидкодію та простішу інтеграцію в CI/CD, що є критичними чинниками для сучасних SDLC-моделей.

У межах роботи було сформульовано та формалізовано набір ключових метрик тестового покриття й ефективності автоматизованого тестування. Запропонований підхід дозволяє перейти від спрощеного трактування покриття як відсотка виконаного коду до комплексної оцінки, яка враховує результативність виявлення дефектів, часові витрати та стабільність тестів. Розроблена математична модель інтегральної оцінки ефективності тестового покриття створює формальну основу для кількісного аналізу та порівняння різних стратегій автоматизації тестування.

Практична частина роботи була присвячена проєктуванню та реалізації універсального автоматизованого тестового фреймворку на основі безкоштовного технологічного стеку Playwright, Python/TypeScript, Docker, Jenkins і Allure. Обґрунтовано вибір саме цих інструментів з огляду на їх відкритість, кросплатформеність, масштабованість і придатність до інтеграції в реальні промислові процеси. Реалізована архітектура забезпечує чітку взаємодію компонентів «код тестів – контейнер – CI-сервер – звіт», що гарантує відтворюваність середовища виконання та прозорість результатів тестування.

У процесі експериментального дослідження було проведено порівняльний аналіз показників «до» і «після» впровадження автоматизованого тестування. Отримані результати засвідчили зростання рівня тестового покриття, підвищення коефіцієнта виявлення дефектів автоматизованими тестами та зменшення кількості помилок, що потрапляють у пізні етапи SDLC або продуктивне середовище. Особливу практичну цінність має скорочення середнього часу виявлення дефектів, що безпосередньо вплинуло на швидкість зворотного зв'язку для розробників і загальну продуктивність команди.

Аналіз стабільності автоматизованого тестування показав, що використання Playwright і коректно спроектованої тестової архітектури дозволяє знизити рівень флейковості тестів і підвищити довіру до результатів CI/CD. Зменшення кількості хибних спрацьовувань і необґрунтованих блокувань

конвеєра позитивно вплинуло на стабільність процесу інтеграції та доставки програмного забезпечення. Це підтверджує тезу про те, що ефективність автоматизованого тестування визначається не лише кількістю тестів, а й якістю їх реалізації та інтеграції в SDLC.

Інтегральна оцінка ефективності тестового покриття, обчислена за запропонованою математичною моделлю, підтвердила комплексний позитивний ефект від упровадження автоматизованого тестування. Зростання інтегрального показника свідчить про узгоджене покращення всіх ключових складових: покриття, результативності, швидкодії та надійності. Це дозволяє зробити висновок, що автоматизація тестування в запропонованій конфігурації не є формальною, а реально підвищує якість програмного забезпечення та ефективність життєвого циклу його розробки.

Узагальнюючи результати дослідження, можна стверджувати, що сучасні інструменти автоматизованого тестування, за умови їх правильного вибору, проєктування та інтеграції, мають суттєвий позитивний вплив на всі етапи SDLC. Автоматизоване тестування сприяє ранньому виявленню дефектів, підвищенню стабільності релізів, скороченню витрат на виправлення помилок і формуванню культури якості в командах розробки. Практичні результати роботи можуть бути використані як у навчальному процесі, так і під час розробки реальних програмних продуктів, а запропонований підхід може бути розширений шляхом інтеграції додаткових видів тестування, аналітики та інтелектуальних методів оптимізації тестового покриття, що визначає перспективи подальших досліджень у цьому напрямі.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гаврилюк І. В., Gavrilyuk I. V. Дослідження методів та засобів комунікації вимог на етап тестування програмного забезпечення : thesis. 2015. URL: <http://elartu.tntu.edu.ua/handle/lib/21464> .
2. Киричек Г., Тягунова М., Курач А. Автоматизоване тестування веб-платформ з використанням java та selenium. *Information technology and society*. 2022. № 1. С. 31–37. URL: <https://doi.org/10.32689/maup.it.2022.1.4> .
3. Ляшко С. Ю. Автоматизоване тестування веб-додатків : master's thesis. 2019. URL: <http://essuir.sumdu.edu.ua/handle/123456789/75888> .
4. Ali H. M., Hamza M. Y., Rashid T. A. A Comprehensive Study on Automated Testing with The Software Lifecycle. *The Journal of Duhok University*. 2023. Vol. 26, no. 2. P. 613–620. URL: <https://doi.org/10.26682/csjuod.2023.26.2.55>
5. Allure-framework. *Github.com/*. URL: <https://github.com/allure-framework>.
6. Beizer B. Software testing techniques. 2nd ed. New York : Van Nostrand Reinhold, 1983. 550 p.
7. Beizer B., Wiley J. Black box testing: techniques for functional testing of software and systems. *IEEE software*. 1996. Vol. 13, no. 5. P. 98. URL: <https://doi.org/10.1109/ms.1996.536464>
8. Build great things at any scale the leading open source automation server, jenkins provides hundreds of plugins to support building, deploying and automating any project. *Jenkins*. URL: <https://www.jenkins.io/>.
9. Design and evaluation of agilebpm SDLC – an agile SDLC for business process management systems / R. Davila-Campos et al. *IEEE access*. 2025. P. 1. URL: <https://doi.org/10.1109/access.2025.3594684>
10. Erratum / K. Hu et al. *Software Testing, Verification and Reliability*. 2021. URL: <https://doi.org/10.1002/stvr.1795>



- 11.Fowler M. Continuous integration. *martinfowler.com*.  
URL: <https://martinfowler.com/articles/continuousIntegration.html>.
- 12.Fowler M. Patterns of enterprise application architecture. Addison-Wesley Professional, 2002. 560 p.
- 13.Fowler M. Refactoring: improving the design of existing code. *Extreme programming and agile methods – xp/agile universe 2002*. Berlin, Heidelberg, 2002. P. 256. URL: [https://doi.org/10.1007/3-540-45672-4\\_31](https://doi.org/10.1007/3-540-45672-4_31)
- 14.Growing object-oriented software, guided by tests / ed. by P. Nat. Upper Saddle River, NJ : Addison Wesley, 2010. 358 p.
- 15.Humble J., Farley D. Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional, 2010. 512 p.
- 16.Humble J., Forsgren N., Kim G. Accelerate : the science of lean software and devops: building and scaling high performing technology organizations. IT Revolution Press, 2018.
- 17.Jorgensen P. C., DeVries B. A perspective on testing. *Software testing*. 5th ed. Boca Raton, 2021. P. 3–14. URL: <https://doi.org/10.1201/9781003168447-2>
- 18.Kesavan E. Shift-Left and continuous testing in quality assurance engineering ops and devops. *International journal of scientific research and modern technology*. 2024. P. 16–21. URL: <https://doi.org/10.38124/ijsrmt.v3i1.859>
- 19.Khushalani P. CI/CD Systems. *Kubernetes Application Developer*. Berkeley, CA, 2022. P. 69–93. URL: [https://doi.org/10.1007/978-1-4842-8032-4\\_3](https://doi.org/10.1007/978-1-4842-8032-4_3)
- 20.Kumar A., Agarwal T., Dwivedi A. Automating API Performance Testing with CI/CD Pipelines. *Journal of Computer Science Engineering and Software Testing*. 2024. Vol. 10, no. 3. P. 43–52.  
URL: <https://doi.org/10.46610/jocses.2024.v10i03.004>

21. Lewis J. L., Mullins B. K. Theory-Based practice: A model SDLS program. *New horizons in adult education and human resource development*. 1993. Vol. 7, no. 1. P. 11–21. URL: <https://doi.org/10.1002/nha3.10047>
22. Meszaros G. XUnit test patterns: refactoring test code (the addison-wesley signature series). Addison-Wesley Professional, 2007. 944 p.
23. Mohammed N. The impact of automated testing and devops on software quality: a review. *International journal of core engineering & management*. 2023. Vol. 7, no. 5. URL: <https://doi.org/10.5281/zenodo.14195537>
24. Montgomery M. M. Comprehensive analysis of automated testing frameworks and tools forefficient software quality assurance. *Iscsitr- international journal of software engineering and development*. 2025. Vol. 6, no. 2. P. 1–5. URL: [https://doi.org/10.63397/iscsitr-ij sed\\_2025\\_06\\_02\\_001](https://doi.org/10.63397/iscsitr-ij sed_2025_06_02_001)
25. Narendar K. A. Integrating performance testing into CI/CD pipelines for test automation. *Journal of scientific and engineering research*. 2020. Vol. 7, no. 6. P. 272–278. URL: <https://doi.org/10.5281/zenodo.12754783>
26. Playwright enables reliable end-to-end testing for modern web apps. *Playwright*. URL: <https://playwright.dev/>.
27. Pressman R. S. Software engineering: a practitioner's approach. 4th ed. McGraw-Hill Companies, 1996. 852 p.
28. Pytest: helps you write better programs. *pytest.org*. URL: <https://docs.pytest.org/en/stable/>.
29. Royce W. W. Managing the development of large software systems (1970). *Ideas that created the future*. 2021. P. 321–332. URL: <https://doi.org/10.7551/mitpress/12274.003.0035>
30. Selenium automates browsers. primarily it is for automating web applications for testing purposes, but is certainly not limited to just that. boring web-based administration tasks can (and should) also be automated as well. *selenium.dev*. URL: <https://www.selenium.dev/>.

31. Shift left / A. Kohne et al. *Die IT-Fabrik*. Wiesbaden, 2016. P. 43–51.  
URL: [https://doi.org/10.1007/978-3-658-15931-3\\_6](https://doi.org/10.1007/978-3-658-15931-3_6)
32. Shift-Left and shift-right testing approaches: a practical roadmap for continuous quality in agile and devops. *Journal of information systems engineering and management*. 2024. URL: <https://doi.org/10.52783/jisem.v9i4.127>
33. Shift Left Testing: 6 Essentials for Successful Implementation. *Lightrun*.  
URL: <https://lightrun.com/shift-left-testing/>.
34. Shift left testing paradigm process implementation for quality of software based on fuzzy / S. A. Vaddadi et al. *Soft computing*. 2023.  
URL: <https://doi.org/10.1007/s00500-023-08741-5>
35. S. K. J. Proactive software testing: the shift-left approach to early defect detection and prevention. *International journal for multidisciplinary research*. 2019. Vol. 1, no. 2. URL: <https://doi.org/10.36948/ijfmr.2019.v01i02.36366>
36. Sommerville I. *Software engineering*. 7th ed. Harlow, Essex : Pearson/Addison-Wesley, 2004. 759 p.
37. *The art of software testing* / ed. by G. J. Myers, T. Badgett, C. Sandler. Wiley, 2012. URL: <https://doi.org/10.1002/9781119202486>
38. Vaidyanathan A. S., Ramamurthy A. K., Murugesan M. S. Revolutionizing POS Terminal Testing with Python and CI/CD Automation. *International Journal of Computer Trends and Technology*. 2025. Vol. 73, no. 4. P. 1–4.  
URL: <https://doi.org/10.14445/22312803/ijctt-v73i4p101>
39. van Merode H. CI/CD concepts. *Continuous integration (CI) and continuous delivery (CD)*. Berkeley, CA, 2023. P. 11–27. URL: [https://doi.org/10.1007/978-1-4842-9228-0\\_2](https://doi.org/10.1007/978-1-4842-9228-0_2)
40. van Merode H. Testing pipelines. *Continuous integration (CI) and continuous delivery (CD)*. Berkeley, CA, 2023. P. 285–308.  
URL: [https://doi.org/10.1007/978-1-4842-9228-0\\_6](https://doi.org/10.1007/978-1-4842-9228-0_6)

- 41.Vivek J. Continuous testing in CI/CD pipelines. *International journal of innovative research and creative technology*. 2023. Vol. 9, no. 1. P. 1–7.  
URL: <https://doi.org/10.5281/zenodo.14883221>
- 42.Zhoc K. C. H., Chen G. Reliability and validity evidence for the Self-Directed Learning Scale (SDLS). *Learning and individual differences*. 2016. Vol. 49. P. 245–250. URL: <https://doi.org/10.1016/j.lindif.2016.06.013>

## ДОДАТКИ

### Додаток А Процес розгортання середовища автоматизації

Налаштування інфраструктури автоматизованого тестування для виконання end-to-end тестів у межах цієї роботи виконувалося на віртуальній машині VirtualBox з операційною системою Ubuntu 22.04.4 LTS. Такий підхід дозволяє отримати контрольоване, відтворюване середовище, яке не залежить від основної операційної системи розробника та максимально наближене до реальних серверних умов.

Після створення віртуальної машини в VirtualBox було встановлено Ubuntu Server, налаштовано мережеве з'єднання в режимі NAT або Bridged та виконано оновлення системних пакетів. На початковому етапі система була підготовлена до роботи шляхом встановлення базових утиліт, зокрема curl, wget, git, а також засобів керування сертифікатами. Це створило основу для подальшого встановлення інструментів автоматизації та CI/CD.

Для роботи Playwright у середовищі Ubuntu було попередньо встановлено інтерпретатор Python, а також менеджер пакетів pip. Після цього в межах окремого проєктного каталогу було створено віртуальне середовище Python, що дозволило ізолювати залежності тестового фреймворку від системних бібліотек. Playwright інстальовався через pip як Python-пакет, після чого виконувалася команда ініціалізації браузерів, яка автоматично завантажує Chromium, Firefox і WebKit разом із необхідними системними залежностями. Особливу увагу було приділено коректній установці бібліотек, необхідних для headless-режиму браузерів, оскільки саме цей режим використовується під час запуску тестів у CI/CD.

Для формування звітності було налаштовано Allure Report. Оскільки Allure є Java-орієнтованим інструментом, попередньо було встановлено середовище

виконання Java (OpenJDK). Після цього Allure інстальювався як окрема утиліта, доступна з командного рядка. У проєкті тестування було додатково встановлено Python-адаптер Allure, який забезпечує формування сирих результатів тестів у стандартному форматі. Таким чином, після виконання тестів у каталозі проєкту автоматично створюється директорія з результатами, яка надалі використовується для генерації HTML-звіту. Перевірка коректності налаштування виконувалася шляхом запуску тестів локально та генерації пробного звіту.

Для стандартизації середовища виконання тестів було встановлено Docker. Інсталяція виконувалася через офіційний репозиторій із додаванням GPG-ключів і відповідних джерел пакетів. Після встановлення Docker-engine користувача було додано до групи `docker`, що дозволило запускати контейнери без використання суперкористувача. На цьому етапі було перевірено коректність роботи Docker шляхом запуску тестового контейнера. Далі створювався Docker-образ, який містить середовище виконання Playwright-тестів: Python або Node.js, бібліотеки Playwright, браузері та необхідні системні залежності. Цей образ використовувався як базове середовище для запуску тестів як локально, так і з CI-сервера.

Наступним етапом було встановлення Jenkins як сервера безперервної інтеграції. Jenkins інстальювався на віртуальну машину як окремий сервіс із використанням Java-середовища. Після першого запуску виконувалося початкове налаштування через веб-інтерфейс: створення адміністративного користувача, встановлення рекомендованих плагінів і налаштування доступу до файлової системи. Для коректної взаємодії з Docker Jenkins-користувач також був доданий до групи `docker`, що дозволило запускати контейнеризовані тести без додаткових привілеїв.

У Jenkins було створено Pipeline-проєкт, який описує CI/CD-конвеєр у вигляді декларативного сценарію. У межах цього конвеєра визначалися етапи

отримання коду з репозиторію, запуск Docker-контейнера з тестами, збирання результатів і генерація звіту Allure. Для цього додатково встановлювався плагін Allure для Jenkins, який дозволяє публікувати звіти безпосередньо з веб-інтерфейсу CI-сервера. Змінні середовища, такі як адреса тестового стенду, режим headless і облікові дані для тестів, передавалися в контейнер через параметри запуску, що забезпечило гнучкість і повторюваність конфігурації.

Після завершення налаштування всієї інфраструктури було виконано інтеграційне тестування системи в цілому. Тести запускалися як локально на віртуальній машині, так і через Jenkins-конвеєр, після чого перевірялася коректність формування артефактів і публікації звітів. У результаті була отримана повністю функціональна інфраструктура автоматизованого тестування, розгорнута на Ubuntu 22.04.4 у VirtualBox, яка забезпечує відтворюваність експериментів, автоматичний запуск тестів і централізований аналіз результатів у межах дослідження.

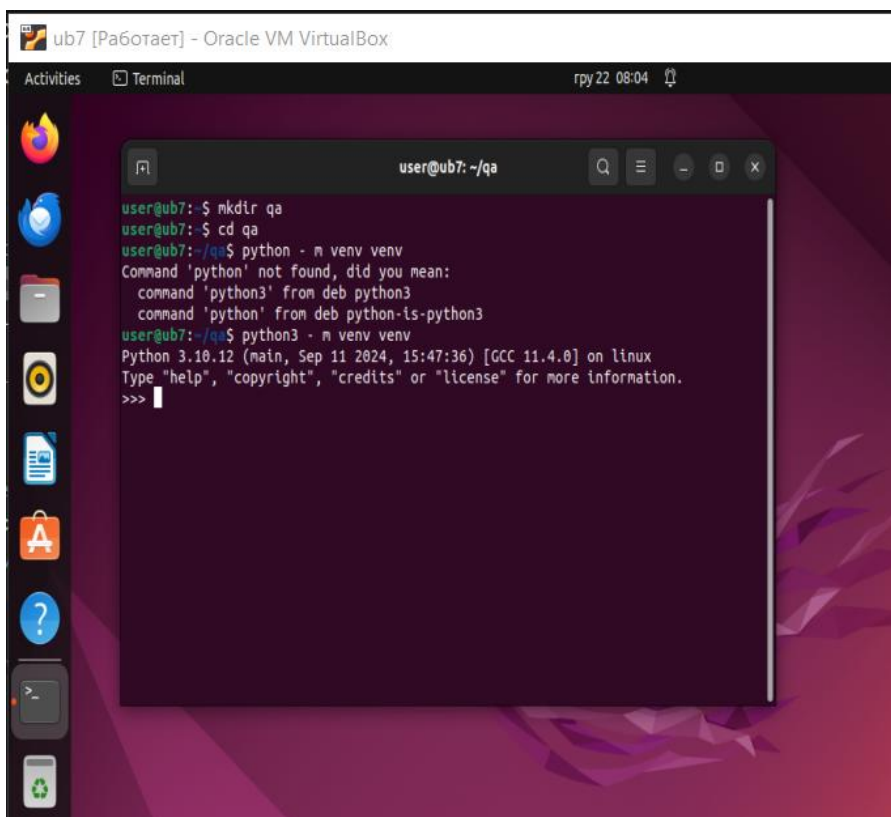


Рис. А.1 Налаштування віртуального середовища

## Додаток Б Налаштування системи

### 0) Базова підготовка ОС

```
sudo apt update && sudo apt -y upgrade  
sudo apt -y install curl wget git ca-certificates gnupg lsb-release unzip software-properties-common  
sudo timedatectl set-timezone Europe/Kyiv
```

#### 1) Python + venv (для тестів)

```
sudo apt -y install python3 python3-pip python3-venv
```

Створюємо папку проєкту

```
mkdir -p ~/e2e-framework && cd ~/e2e-framework
```

Віртуальне середовище

```
python3 -m venv .venv  
source .venv/bin/activate
```

Pytest + Playwright + Allure adapter (Python)

```
pip install -U pip  
pip install pytest pytest-asyncio playwright allure-pytest
```

Встановили браузер Playwright + системні залежності

```
playwright install --with-deps
```

Швидка перевірка

```
python -c "import playwright; print('Playwright OK')"
```



## 2) Allure Commandline (звіт)

=====

Java потрібна для Allure

```
sudo apt -y install openjdk-17-jre
```

Встановлюємо Allure (через GitHub release)

```
ALLURE_VER="2.27.0"
cd /tmp
wget -q "https://github.com/allure-
framework/allure2/releases/download/${ALLURE_VER}/allure-
${ALLURE_VER}.tgz"
sudo tar -zxf "allure-${ALLURE_VER}.tgz" -C /opt/
sudo ln -sf "/opt/allure-${ALLURE_VER}/bin/allure" /usr/local/bin/allure
```

Перевірка

```
allure --version
```

Повертаємось у проєкт

```
cd ~/e2e-framework
source .venv/bin/activate

pytest -q --alluredir=allure-results
allure generate allure-results -o allure-report --clean
allure open -h 0.0.0.0 -p 8088 allure-report
```

## 3) Docker (Engine + Compose plugin)

```
sudo install -m 0755 -d /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

```
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

```
sudo apt update
sudo apt -y install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-
compose-plugin
```

Дозволяємо запуск docker без sudo (потрібен re-login або reboot)

```
sudo usermod -aG docker "$USER"
newgrp docker
```

Перевірка

```
docker run --rm hello-world
```

4) Jenkins (LTS) + запуск як сервіс

Jenkins потребує Java (вже встановлено openjdk-17-jre вище)

```
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | \
```

```
sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null
```

```
echo \
```

```
"deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
```

```
https://pkg.jenkins.io/debian-stable binary/" | \
```

```
sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null
```

```
sudo apt update
```

```
sudo apt -y install jenkins
```

```
sudo systemctl enable --now jenkins
```

```
sudo systemctl status jenkins --no-pager
```

Перевірте пароль для первинного входу (в UI Jenkins)

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

5) Дати Jenkins доступ до Docker

Додаємо користувача jenkins у групу docker

```
sudo usermod -aG docker jenkins
```

Перезапуск Jenkins щоб підхопив групи

```
sudo systemctl restart jenkins
```

перевірка, що docker доступний від імені jenkins

```
sudo -u jenkins -H bash -lc "docker version"
```

6) Мінімальні команди для CI запуску тестів у контейнері

Якщо Dockerfile у корені репозиторію.

6.1) Збірка образу (локально або в Jenkins stage)

```
docker build -t e2e-playwright:latest .
```

6.2) Запуск тестів у контейнері з прокиданням env-параметрів

```
docker run --rm \
-e BASE_URL="https://your-app.test" \
-e HEADLESS="true" \
-e BROWSER="chromium" \
-e TIMEOUT_MS="30000" \
-e TEST_USER="demo" \
-e TEST_PASS="demo" \
-v "$(pwd)/allure-results:/work/allure-results" \
e2e-playwright:latest \
pytest -q --alluredir=/work/allure-results
```

6.3) Генерація HTML-звіту Allure (можна як окремий stage Jenkins)

```
allure generate allure-results -o allure-report --clean
```

## Додаток В. Dockerfile

Універсальний образ для E2E тестів: Python + Playwright + browsers + Allure results

Під Ubuntu 22.04.4/CI. Базуємось на офіційному образі Playwright (Ubuntu Jammy),

де вже є браузерери та системні залежності.

FROM mcr.microsoft.com/playwright/python:v1.49.0-jammy

Робоча директорія

WORKDIR /work

(Опційно) системні дрібниці

RUN python -m pip install --upgrade pip

Спершу копіюємо requirements для кешування шарів

COPY requirements.txt /work/requirements.txt

Встановлюємо залежності Python (pytest, playwright, allure adapter тощо)

RUN pip install --no-cache-dir -r /work/requirements.txt

Копіюємо весь проєкт (тести, pages, utils, config тощо)

COPY . /work

Директорії під артефакти

```
RUN mkdir -p /work/allure-results /work/artifacts
```

Значення за замовчуванням (можна перевизначати через -e ...)

```
ENV BASE_URL="https://example.test" \  
    HEADLESS="true" \  
    BROWSER="chromium" \  
    TIMEOUT_MS="30000" \  
    PYTHONUNBUFFERED="1"
```

Команда за замовчуванням: запуск тестів з формуванням allure-results

(У Jenkins зазвичай перевизначають CMD/args, але так зручно і локально)

```
CMD ["pytest", "-q", "--alluredir=/work/allure-results"]
```

```
```
```

requirements.txt (мінімальний):

```
```txt  
pytest==8.3.4  
pytest-asyncio==0.24.0  
playwright==1.49.0  
allure-pytest==2.13.5  
```
```

Як зібрати і запустити:

```
```bash
```

```
docker build -t e2e-playwright:latest .
```

```
docker run --rm \  
-e BASE_URL="https://your-app.test" \  
-e TEST_USER="demo" -e TEST_PASS="demo" \  
-v "$(pwd)/allure-results:/work/allure-results" \  
-v "$(pwd)/artifacts:/work/artifacts" \  
e2e-playwright:latest  
...
```

## Додаток Г. Jenkinsfile (Declarative Pipeline)

```
//  
// Stages: build -> test -> allure publish  
pipeline {  
    agent any  
  
    environment {  
        IMAGE = "e2e-playwright:latest"  
  
        // Конфігурація тестів (приклад)  
        BASE_URL  = "https://your-app.test"  
        HEADLESS  = "true"  
        BROWSER   = "chromium"  
        TIMEOUT_MS = "30000"  
  
        // Облікові дані (краще винести у Jenkins Credentials)  
        TEST_USER  = "demo"  
        TEST_PASS  = "demo"  
    }  
  
    stages {  
        stage('Build') {  
            steps {  
                sh ""  
                set -e  
                docker build -t "${IMAGE}" .  
            }  
        }  
    }  
}
```



```

    ""
  }
}

stage('Test') {
  steps {
    sh """
      set -e

      rm -rf allure-results allure-report artifacts || true
      mkdir -p allure-results artifacts

      docker run --rm \
        -e BASE_URL="${BASE_URL}" \
        -e HEADLESS="${HEADLESS}" \
        -e BROWSER="${BROWSER}" \
        -e TIMEOUT_MS="${TIMEOUT_MS}" \
        -e TEST_USER="${TEST_USER}" \
        -e TEST_PASS="${TEST_PASS}" \
        -v "$WORKSPACE/allure-results:/work/allure-results" \
        -v "$WORKSPACE/artifacts:/work/artifacts" \
        "${IMAGE}"
    """
  }

  post {
    always {
      archiveArtifacts artifacts: 'artifacts/**/*', allowEmptyArchive: true
      archiveArtifacts artifacts: 'allure-results/**/*', allowEmptyArchive: true
    }
  }
}

```

```
    }  
  }  
}
```

```
stage('Allure Publish') {  
  steps {  
    // Потрібен встановлений Jenkins-плагін Allure Report  
    allure([  
      includeProperties: false,  
      jdk: "",  
      properties: [],  
      reportBuildPolicy: 'ALWAYS',  
      results: [[path: 'allure-results']]  
    ])  
  }  
}
```

```
post {  
  always {  
    echo "Pipeline finished. Check Allure report in Jenkins build page."  
  }  
}
```