БАКАЛАВРСЬКА РОБОТА

БР.ІП – 14.00.00.000 ІПЗ

Група ІП-22-1

Лазечко Артур

2026

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Лазечко Артур Русланович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Інструменти автоматизованого тестування в CI/CD-конвеєрах
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Лазечко А. Р._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Бандура Вікторія Валеріївна, к.т.н., доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ БАКАЛАВРА СТУДЕНТОВІ

Лазечко Артур Русланович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Інструменти автоматизованого тестування в CI/CD-конвеєрах"

керівник проекту (роботи) Бандура Вікторія Валеріївна, к.т.н., доцент

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1 - Безперервне інтегрування (рис. 1.2, ст. 19)
2 Безперервна інтеграція та безперервна доставка (рис. 1.4, ст. 22)
3 Динамічне тестування безпеки застосунків (DAST (рис. 2.3, ст. 41)
4 Приклад файлу package.json (рис. 3.2, ст. 56)
5 Візуалізація профілів навантажувальних тестів61 (рис. 3.3, ст. 63)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Дипломна робота містить 78 сторінок, 26 рисунків, 2 таблиці, список використаних джерел із 40 найменувань.

Метою роботи є дослідження інструментів автоматизованого тестування в CI/CD-конвеєрах та аналіз підходів до забезпечення якості, надійності та безпеки програмного забезпечення в процесі безперервної інтеграції.

Об'єкт дослідження: процеси автоматизованого тестування в CI/CD-конвеєрах програмного забезпечення.

Предмет дослідження: методи, технології та інструменти автоматизованого тестування, а також засоби інтеграції тестування в DevOps.

Результати дослідження: проведено аналіз сучасних підходів до автоматизації тестування, досліджено принципи побудови CI/CD-конвеєрів, розглянуто популярні інструменти автоматизованого тестування та інтеграції, а також реалізовано приклад використання інструментів тестування.

У першому розділі розглянуто теоретичні основи автоматизованого тестування, сучасні підходи та концепції CI/CD, а також їх роль у забезпеченні якості програмного забезпечення.

У другому розділі досліджено методи автоматизації тестування, інструменти для різних рівнів тестування та принципи інтеграції безпеки.

У третьому розділі представлено реалізацію автоматизованого тестування в CI/CD-конвеєрі із використанням сучасних інструментів.

Висновок: використання автоматизованого тестування в CI/CD-конвеєрах дозволяє підвищити якість програмного забезпечення, скоротити час розробки, забезпечити безперервний контроль якості та ефективно інтегрувати процеси тестування в життєвий цикл розробки.

КЛЮЧОВІ СЛОВА: АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, CI/CD, DEVOPS, DEVSECOPS, UNIT-ТЕСТУВАННЯ, ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ, E2E-ТЕСТУВАННЯ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ANNOTATION

The thesis consists of 78 pages, 26 figures, 2 tables, and a reference list with 40 sources.

The aim of the work is to study automated testing tools in CI/CD pipelines and to analyze approaches for ensuring software quality, reliability, and security within continuous integration and delivery processes.

Research object: processes of automated testing in CI/CD pipelines.

Research subject: methods, technologies, and tools of automated testing, including unit, integration, and end-to-end testing, as well as approaches to integrating testing into DevOps and DevSecOps processes.

Research results: modern approaches to automated testing were analyzed, principles of CI/CD pipeline design were studied, popular testing and integration tools were reviewed, and a practical implementation of automated testing in a real development environment was carried out.

The first section describes the theoretical foundations of automated testing, modern approaches, and CI/CD concepts, as well as their role in ensuring software quality.

The second section analyzes methods of test automation, tools for different testing levels, and principles of integrating security into DevSecOps processes.

The third section covers the implementation of automated testing within a CI/CD pipeline using modern tools and evaluates the effectiveness of the proposed solutions.

Conclusion: the use of automated testing in CI/CD pipelines improves software quality, reduces development time, ensures continuous quality control, and effectively integrates testing processes into the software development lifecycle.

KEY WORDS: AUTOMATED TESTING, CI/CD, DEVOPS, DEVSECOPS, UNIT TESTING, INTEGRATION TESTING, END-TO-END TESTING, SOFTWARE QUALITY.

ЗМІСТ

ВСТУП

У сучасному світі розробка програмного забезпечення характеризується високою динамікою, складністю систем та постійним зростанням вимог до якості продуктів. Швидке впровадження нових функціональних можливостей, часті оновлення та необхідність забезпечення безперебійної роботи систем зумовлюють потребу у використанні ефективних підходів до автоматизації процесів розробки та тестування. Одним із ключових напрямів у цьому контексті є впровадження практик безперервної інтеграції та безперервної доставки (CI/CD), що дозволяють автоматизувати життєвий цикл програмного забезпечення та забезпечити його стабільність і надійність.

Актуальність теми зумовлена необхідністю використання інструментів автоматизованого тестування в CI/CD-конвеєрах, які дозволяють своєчасно виявляти помилки, зменшувати ризики при розгортанні програмних продуктів і підвищувати ефективність процесів розробки. Існуючі підходи до тестування, що базуються на ручних методах, не забезпечують необхідної швидкості та точності перевірки в умовах сучасних вимог до безперервної доставки. У зв’язку з цим впровадження автоматизованих інструментів тестування, а також інтеграція їх у DevOps та DevSecOps-процеси є важливим завданням для підвищення якості програмного забезпечення.

Метою роботи є дослідження та аналіз інструментів автоматизованого тестування в CI/CD-конвеєрах, а також розробка та оцінка ефективності їх використання для забезпечення якості програмного забезпечення.

Завданнями дослідження є: аналіз теоретичних основ автоматизованого тестування та CI/CD-підходів; дослідження сучасних методів і інструментів автоматизації тестування; вивчення принципів інтеграції тестування безпеки в DevSecOps; розробка та реалізація прикладу автоматизованого тестування в CI/CD-конвеєрі; оцінка ефективності впроваджених рішень.

Об’єктом дослідження є процеси автоматизованого тестування програмного забезпечення в умовах CI/CD-конвеєрів.

Предметом дослідження є методи, моделі та інструменти автоматизованого тестування, зокрема unit-, integration- та end-to-end тестування, а також засоби інтеграції тестування в процеси безперервної інтеграції та доставки.

Методи дослідження включають аналіз наукових та технічних джерел для визначення сучасного стану проблеми, порівняльний аналіз інструментів автоматизованого тестування, моделювання процесів CI/CD, експериментальний підхід для реалізації та тестування системи, а також аналіз отриманих результатів.

У роботі передбачається розробка та впровадження рішення для автоматизованого тестування в CI/CD-конвеєрі із використанням сучасних інструментів, що забезпечують підвищення якості програмного забезпечення, зменшення часу на тестування та оптимізацію процесів розгортання. Особлива увага приділяється інтеграції тестування безпеки та забезпеченню надійності програмних систем.

Очікується, що результати дослідження сприятимуть підвищенню ефективності процесів розробки програмного забезпечення, забезпеченню безперервного контролю якості та зниженню витрат на підтримку та супровід програмних продуктів.

Бакалаврська робота містить 78 сторінок, 26 рисунків, 2 таблиці, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ В CI/CD-КОНВЕЄРАХ

1.1 Основи автоматизованого тестування програмного забезпечення

Оскільки розробка програмного забезпечення прискорюється завдяки впровадженню методологій Agile та DevOps, організації все більше покладаються на конвеєри безперервної інтеграції та безперервного розгортання (CI/CD) для забезпечення швидкої та надійної доставки. Хоча ці практики підвищують ефективність та швидкість, вони часто поступаються пріоритетом безпеки на користь гнучкості. Цей компроміс став суттєвою проблемою, оскільки вразливості з'являються на ранніх етапах життєвого циклу розробки програмного забезпечення та залишаються непоміченими до моменту виробництва. У відповідь на це виникла концепція DevSecOps, спрямована на вбудовування практик безпеки безпосередньо в робочий процес CI/CD. DevSecOps розширює принципи DevOps, інтегруючи тестування безпеки, забезпечення відповідності та зниження ризиків з найперших етапів розробки. Він наголошує на автоматизації, співпраці та постійному зворотному зв'язку між командами розробки, експлуатації та безпеки. Мета полягає в тому, щоб змістити безпеку "вліво" - виявляти та вирішувати проблеми якомога раніше, коли їх дешевше та легше виправити [1]. У цій роботі досліджується, як автоматизовані інструменти тестування безпеки можуть бути ефективно інтегровані в конвеєри CI/CD як частину стратегії DevSecOps. У ньому досліджуються принципи роботи ключових методів тестування безпеки, таких як статичне тестування безпеки додатків (SAST), динамічне тестування безпеки додатків (DAST), аналіз складу програмного забезпечення (SCA) та сканування інфраструктури як коду (IaC). Впровадження та впровадження цих інструментів мають вирішальне значення для досягнення безперервної безпеки без перешкоджання процесу розробки. Крім того, це дослідження розглядає сучасний ландшафт інструментів DevSecOps, порівнюючи їхні можливості, функції інтеграції та ефективність у реальному світі. У ньому

також представлені стратегії практичного впровадження, спираючись на висновки галузевих тематичних досліджень. Зрештою, робота має на меті надати дорожню карту для організацій, які прагнуть побудувати безпечні конвеєри CI/CD, збалансувавши швидкість та безпеку в процесі розробки програмного забезпечення. Вбудовуючи безпеку в життєвий цикл розробки, організації можуть проактивно керувати ризиками, зберігаючи гнучкість та інновації.

Індустрія розробки програмного забезпечення зазнала значних трансформацій за останні роки завдяки широкому впровадженню гнучких методологій та практик DevOps. Ці парадигми наголошують на безперервній розробці, інтеграції та доставці, що дозволяє командам випускати функції швидше та ефективніше. Однак, такі швидкі темпи часто відсувають на другий план безпеку, яка традиційно входить у процес на завершальних етапах. Такий реактивний підхід виявився неадекватним з огляду на зростання кіберзагроз та нормативних вимог. В результаті багато організацій опиняються вразливими до порушень безпеки, яким можна було б запобігти за допомогою раннього втручання [2]. Мотивацією для цього дослідження є зростаюча потреба вбудовувати безпеку в кожен етап життєвого циклу розробки, забезпечуючи не лише швидку, але й безпечну доставку програмного забезпечення. DevSecOps постає як стратегічний підхід, який об'єднує розробку, операції та безпеку в єдиний, автоматизований конвеєр.

У середовищі CI/CD новий код інтегрується, тестується та розгортається у виробничих середовищах дуже швидко - іноді кілька разів на день. Така частота, хоча й корисна для інновацій та швидкого реагування на потреби клієнтів, залишає мало місця для ручної оцінки безпеки. Вразливості, які прослизають, можуть бути використані невдовзі після розгортання, що призводить до втрати даних, компрометації системи та шкоди репутації. Тому інтеграція перевірок безпеки безпосередньо в конвеєр CI/CD стає важливою. Автоматизуючи тестування безпеки та забезпечуючи дотримання політик безпеки на кожному етапі - фіксації коду, збірці, тестуванні та розгортанні - організації можуть виявляти та пом'якшувати загрози на ранній стадії. Такий проактивний підхід

гарантує, що безпека не стане вузьким місцем, а натомість функціонуватиме як вбудований компонент якості процесу доставки.

Основною метою цього дослідження є аналіз ролі та впровадження автоматизованих інструментів тестування безпеки в конвеєрах CI/CD, керуючись принципами DevSecOps. Його метою є визначення та оцінка ефективності різних методів тестування безпеки, таких як SAST, DAST, SCA та сканування IaC. Ще однією ключовою метою є порівняння стандартних галузевих інструментів, що сприяють цій автоматизації, та надання практичного розуміння їх інтеграції, масштабованості та практичного застосування. Дослідження також має на меті запропонувати організаціям основу для впровадження або вдосконалення практик DevSecOps, тим самим покращуючи загальний стан безпеки без шкоди для швидкості розробки.

Тестування продуктивності є критичним етапом життєвого циклу сучасної розробки програмного забезпечення. Воно гарантує, що програми можуть витримувати очікувані робочі навантаження та залишатися стійкими під час піків трафіку або операційних аномалій. Для великомасштабних систем навіть незначне зниження продуктивності може призвести до значних фінансових втрат. Як широко наведений приклад, повідомляється, що збільшення часу завантаження однієї з веб-сторінок Amazon на 1 секунду може коштувати компанії до 1,6 мільярда доларів щорічно у втрачених продажах [9].

Паралельно, зростання популярності DevOps змінило те, як програмне забезпечення створюється, тестується та постачається. DevOps наголошує на безперервній інтеграції та безперервній доставці (CI/CD), що дозволяє швидке, автоматизоване розгортання через тісно інтегровані конвеєри [3]. Хоча функціональне тестування зараз рутинно вбудоване в робочі процеси CI/CD, тестування продуктивності створює унікальну проблему через його трудомісткий характер, складність та вимоги до ресурсів. Ручне проектування тестів, тривалий час виконання та часте обслуговування тестових скриптів часто перешкоджають безперешкодній інтеграції тестування продуктивності в процеси CI/CD.

Хоча попередні дослідження розглядали аспекти автоматизації тестування

продуктивності в середовищах CI/CD, більшість досліджень спираються на фіксовані інструментальні ланцюжки та зосереджуються переважно на самому процесі. Вплив на продуктивність інструментів навантажувального тестування, зокрема їх споживання ресурсів, поведінка виконання та масштабованість, отримав порівняно мало уваги. Проте вибір інструменту може мати прямий вплив на продуктивність конвеєра, його ресурсоефективність і зрештою, швидкість доставки програмного забезпечення [4].

У цьому дослідженні ми усуваємо цю прогалину, проводячи порівняльну оцінку п'яти широко використовуваних інструментів для тестування навантаження з відкритим кодом: Apache JMeter , Grafana K6 , Gatling , Locust та Artillery . Ці інструменти були обрані на основі попереднього аналізу, який визначив їх як такі, що мають найвищу активність розробки між 2017 і 2020 роками. Ми інтегрували кожен інструмент у конвеєр CI/CD на базі Jenkins, налаштований для тестування програми Spring Boot Java. У наших експериментах використовувалися три широко застосовні методології тестування: репрезентативне тестування навантаження, тестування максимального (стресового) навантаження та тестування пікового навантаження. Крім того, ми автоматизували генерацію профілю навантаження за допомогою Python та включили моніторинг системи в режимі реального часу за допомогою стеку Prometheus+Grafana+cAdvisor .

Ми виміряли показники системного рівня (використання процесора та оперативної пам'яті) та бізнес-рівня (середній час відгуку), а потім застосували U-тести Краскела-Уолліса та Манна-Вітні для визначення статистичної значущості відмінностей у продуктивності [5]. Дані показують, що вибір інструменту для тестування продуктивності помітно впливає як на ефективність системи, так і на якість тестування в конвеєрах CI/CD.

Наші внески такі:

1. Ми пропонуємо емпіричну оцінку п'яти інструментів для тестування продуктивності з відкритим кодом у повністю автоматизованому середовищі CI/CD.

2. Ми представляємо багаторазово використовуваний, розширюваний проект конвеєра для інтеграції автоматизованих тестів продуктивності з генерацією динамічного профілю навантаження та моніторингом у режимі реального часу.

3. Ми статистично перевіряємо наші висновки та пропонуємо практичні поради щодо того, як команди DevOps можуть оптимізувати вибір інструментів для підвищення ефективності, зменшення витрат на ресурси та збільшення швидкості виконання.

Ця робота пропонує підхід до вибору інструментів для тестування продуктивності CI/CD, що базується на даних, і сприяє більш ресурсоорієнтованим та автоматизованим практикам у сучасних робочих процесах DevSecOps. З практичної точки зору, це дослідження має на меті відповісти на такі питання, які може поставити собі організація-розробник програмного забезпечення, розглядаючи використання автоматизованого конвеєра тестування продуктивності:

1. Яка роль та застосування тестування продуктивності як частини концепції DevOps CI/CD?

2. Які інструменти, методи, процеси та метрики складають ефективний автоматизований конвеєр тестування продуктивності для Java-застосунків у середовищі CI/CD DevOps ?

3. Як можна інтегрувати автоматизовану генерацію профілів навантаження в конвеєри тестування продуктивності в середовищі CI/CD?

4. Які категорії тестування можна ефективно інтегрувати в конвеєр?

У сучасному середовищі розробки програмного забезпечення важливо мати ефективний та надійний конвеєр безперервної інтеграції/безперервного розгортання (CI/CD), який допомагає командам швидко створювати високоякісні програмні продукти [6]. Одним із ключових компонентів такого конвеєра є наскрізне тестування, яке гарантує, що всі функції програми працюють належним чином та відповідають вимогам користувача.

Комплексне приймальне тестування може бути трудомістким і дорогим,

якщо проводити його вручну, що може призвести до помилок і затримок. Однак, автоматизувавши процес, команди можуть ефективніше розповсюджувати програмне забезпечення, зменшити ризик помилок і заощадити час.

У цій роботі розглядається, як автоматизувати наскрізне приймальне тестування в React-застосунку як частини конвеєра CI/CD, використовуючи програму для підтвердження концепції в Azure Cloud [7]. Будуть розглянуті основні процедури та інструменти для налаштування автоматизованого тестування. Крім того, у роботі буде досліджено, як інтегрувати автоматизоване тестування з іншими компонентами конвеєра CI/CD, такими як контроль версій, збірка та розгортання.

Ось сформульовані дослідницькі питання:

- Які існують різні способи реалізації наскрізних тестів?
- Як автоматизувати наскрізні приймальні випробування як частину конвеєрів CI/CD?

До кінця цієї роботи буде сформовано чітке розуміння того, як побудувати автоматизований процес комплексного приймального тестування як частину вашого конвеєра CI/CD, використовуючи сервіси Azure та інші інструменти. Мета полягає в тому, щоб чітко зрозуміти переваги автоматизації цього процесу та те, як це може допомогти команді швидко та ефективніше створювати високоякісні програмні продукти.

1.2 Огляд сучасних підходів та досліджень у сфері автоматизації тестування

Еволюція методологій розробки програмного забезпечення від традиційних каскадних моделей до Agile та DevOps суттєво вплинула на те, як створюються та розгортаються додатки. Хоча DevOps зосереджується на автоматизації та співпраці між командами розробки та експлуатації, спочатку йому бракувало чіткого акценту на безпеці. Ця прогалина призвела до появи DevSecOps, яка інтегрує безпеку в кожен етап життєвого циклу розробки

програмного забезпечення (SDLC). Кілька досліджень підкреслили критичну необхідність «зсуву безпеки ліворуч» у конвеєрі CI/CD. За даними, раннє виявлення вразливостей може зменшити витрати на їх усунення до 30 разів порівняно з їх виправленням після розгортання. Це спонукало дослідників та практиків зосередитися на вбудовуванні автоматизованих перевірок безпеки в робочі процеси CI/CD. Традиційні підходи до безпеки часто включають ручний огляд коду та періодичне тестування на проникнення, що неможливо у швидкозмінних середовищах CI/CD.

Такі інструменти, як SonarQube, Checkmarx та Fortify, були перевірені на предмет їхніх можливостей у статичному аналізі коду (SAST), що дозволяє розробникам виявляти вразливості на етапі кодування. Аналогічно, інструменти динамічного тестування, такі як OWASP ZAP та Burp Suite, пропонують аналіз часу виконання для виявлення недоліків під час виконання програми.

Інструменти аналізу складу програмного забезпечення (SCA), включаючи Snyk та WhiteSource , здобули популярність у виявленні вразливостей у залежностях сторонніх розробників, які складають значну частину сучасних програм. Інструменти сканування інфраструктури як коду (IaC), такі як Terraform Sentinel та Checkov, також досліджувалися через їхню здатність виявляти неправильні конфігурації та проблеми безпеки в інфраструктурних скриптах. Дослідження, такі як дослідження Раджа та Арори (2021), підкреслюють необхідність збалансованого підходу, де автоматизація безпеки доповнює, але не замінює людський нагляд. У літературі також вказується на відсутність стандартизованих практик ефективної інтеграції цих інструментів у різноманітні середовища CI/CD. Незважаючи на зростаючий інтерес, залишаються прогалини в досягненні послідовного впровадження практик DevSecOps у всій організації. Існує потреба в більш емпіричних дослідженнях та фреймворках, які б керували вибором інструментів, стратегіями інтеграції та оцінкою продуктивності в реальних випадках використання [8]. Це дослідження має на меті усунути ці прогалини, надаючи детальний аналіз автоматизованих інструментів тестування безпеки та їх практичного впровадження в конвеєрах CI/CD.

DevOps революціонізував розробку програмного забезпечення, сприяючи тісній співпраці між командами розробників та операцій, прагнучи підвищити швидкість розробки, надійність системи та оперативність реагування на потреби клієнтів. Однак безпека часто розглядалася як окремий етап, який вирішувався після завершення розробки та тестування. Це створювало вузькі місця та наражало програми на вразливості, виявлені наприкінці життєвого циклу. DevSecOps, з іншого боку, розширює DevOps, вбудовуючи безпеку як спільну відповідальність на всіх етапах процесу розробки програмного забезпечення. Він інтегрує автоматизовані інструменти та практики безпеки безпосередньо в конвеєр CI/CD, що дозволяє раннє виявлення вразливостей без порушення процесу розробки. Перехід від DevOps до DevSecOps являє собою культурну та технічну зміну, де безпека є безперервною, автоматизованою та інтегрованою, а не реактивною та ізольованою [9].

Безперервне розгортання (CD) робить акцент на швидкості та автоматизації, дозволяючи часто впроваджувати новий код у продакшн — навіть кілька разів на день. Хоча це підвищує швидкість реагування, це також створює значні ризики для безпеки. Основні проблеми включають відсутність видимості вразливостей у коді або залежностях від сторонніх розробників, невідповідність політик безпеки в різних середовищах та недостатнє тестування конфігурацій у хмарних та контейнерних додатках. Більше того, традиційна модель безпеки, яка спирається на ручні перевірки та тестування на пізніх стадіях, не може встигати за швидкими циклами розгортання CD. Ця невідповідність призводить до потрапляння невиявлених недоліків у продакшн, створюючи потенційні точки входу для зловмисників. Ці проблеми вимагають автоматизованого підходу до безпеки в режимі реального часу, безшовно інтегрованого в конвеєри CI/CD.

З'явилося безліч інструментів для підтримки автоматизованого тестування безпеки в робочих процесах CI/CD. Інструменти статичного тестування безпеки додатків (SAST), такі як SonarQube та Checkmarx, аналізують вихідний код на наявність вразливостей без запуску додатка. Інструменти динамічного тестування безпеки додатків (DAST), такі як OWASP ZAP та Burp Suite, моделюють атаки на

запущені додатки для виявлення вразливостей під час виконання.

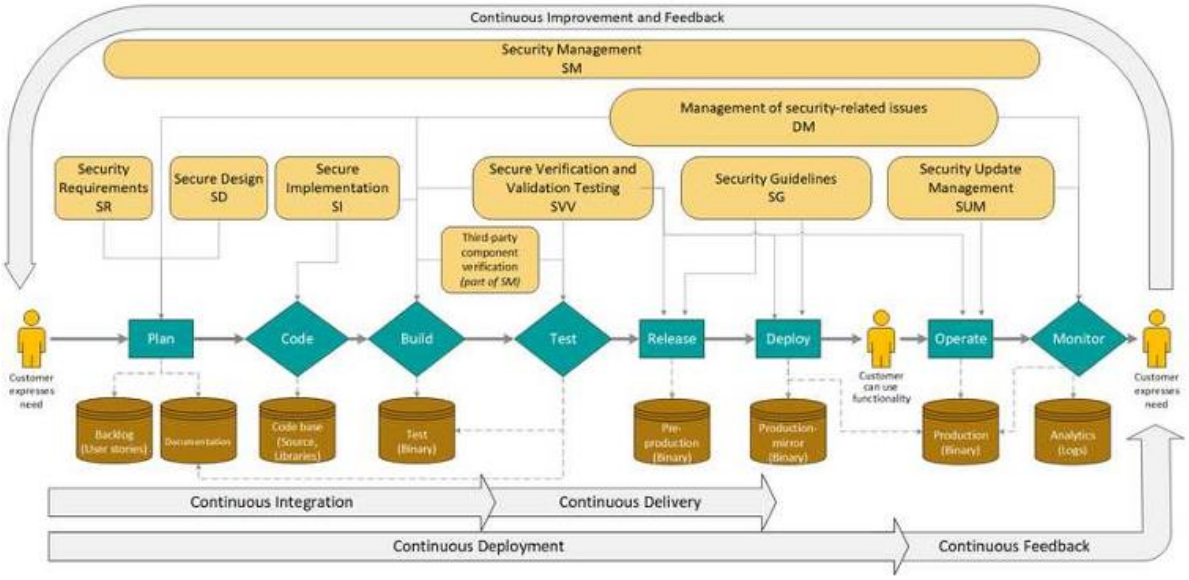


Рисунок 1.1 - Проблеми безпеки під час безперервного розгортання

Інструменти аналізу складу програмного забезпечення (SCA), включаючи Snyk та Black Duck, виявляють ризики в бібліотеках з відкритим кодом та залежностях. Крім того, інструменти безпеки «Інфраструктура як код» (IaC), такі як Checkov та TerraScan, оцінюють файли конфігурації хмари на наявність недоліків безпеки. Ці інструменти можна інтегрувати в різні етапи конвеєра CI/CD, пропонуючи зворотний зв'язок у режимі реального часу та автоматизуючи перевірки відповідності [10]. Однак кожен інструмент має свої обмеження та має бути обраний на основі вимог конкретного проекту, середовища розробки та зрілості команди.

Незважаючи на зростаючий інтерес та прогрес у сфері інструментів і практик DevSecOps, залишається кілька прогалин у дослідженнях. По-перше, бракує стандартизованих фреймворків для інтеграції інструментів тестування безпеки в гетерогенних середовищах CI/CD. Багато організацій стикаються з проблемами сумісності інструментів, масштабованості та підтримки узгоджених політик безпеки в командах розробників. По-друге, хоча окремі інструменти добре задокументовані, порівняльні дослідження, що аналізують їхню

ефективність у практичних масштабних впровадженнях, обмежені. Крім того, більшість існуючої літератури зосереджена на теоретичних перевагах, тоді як менше емпіричних досліджень досліджують довгострокові результати, такі як економія коштів, рівень зниження вразливостей або продуктивність розробників [11]. Нарешті, інтеграція штучного інтелекту та машинного навчання для інтелектуального виявлення та визначення пріоритетів загроз залишається недостатньо дослідженою галуззю. Це дослідження має на меті сприяти заповненню цих прогалин, представивши орієнтований на впровадження погляд на автоматизацію DevSecOps.

1.3 Концепції безперервної інтеграції та безперервної доставки (CI/CD)

У цьому підрозділі розглядається концепція безперервної інтеграції, безперервної доставки та те, як вони поєднуються в конвеєрі CI/CD. Безперервна інтеграція (CI) та безперервна доставка (CD) є частиною процесу безперервної інтеграції/безперервної доставки (CI/CD) та використовуються в сучасній розробці програмного забезпечення.

Безперервна інтеграція (БІ) є частиною процесу розробки програмного забезпечення DevOps, де розробники часто об'єднують зміни коду з основним репозиторієм, після чого проводяться тести та автоматичне виконання збірки, що означає, що тести запускаються та збірка створюється автоматично. (AWS, nd). Неперервна інтеграція (CI) позначає фазу збірки або інтеграції процесу випуску програмного забезпечення, що включає компонент автоматизації, такий як CI або інструмент збірки, та культурний компонент, який стосується навчання тому, як регулярно інтегрувати такий робочий процес у реальну практику [12]. Основними цілями CI є пришвидшення процесу виявлення та виправлення помилок коду, підвищення рівня якості програмного забезпечення та скорочення часу проведення тестів та публікації нових оновлень програмного забезпечення. (AWS, nd)

Неперервна інтеграція (CI) – це крок, який надає розробникам початковий

зворотний зв'язок. Він перевіряє код з репозиторію, збирає його, виконує модульні тести та оцінює якість коду. Якщо будь-яка частина процесу дає збій, конвеєр зупиняється, і розробники зобов'язані негайно виправити збірку CI. Час є вирішальним фактором на цьому кроці. Наприклад, якщо цей етап займе одну годину, розробники мають намір швидше виконувати коміти, що призведе до постійного збою конвеєра.

Як видно з рисунка 1.2, безперервна інтеграція має чотири етапи, перш ніж код, який розробник закомітив, може бути випущений. CI забезпечує безперервне тестування, збірку та звітування про помилки, щоб забезпечити початковий зворотний зв'язок з інженером. Щоразу, коли виникають помилки чи помилки, збірки необхідно швидко виправляти.

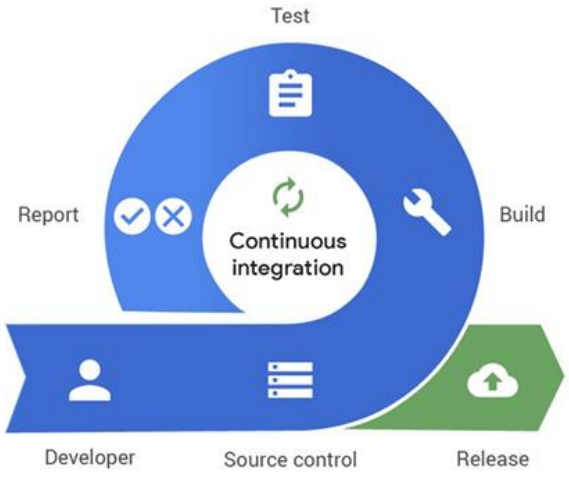


Рисунок 1.2 - Безперервне інтегрування

На рисунку 1.2 показано основні етапи, які зазвичай є частиною налаштування безперервної інтеграції. Ці кроки гарантують, що кожна зміна в коді буде перевірена та перевірена перед подальшим внесенням змін. Автоматизуючи цей процес, команди можуть виявляти проблеми на ранній стадії та підтримувати високу якість свого програмного забезпечення, навіть у міру зростання проекту.

Більшість розробників часто стикаються з проблемами швидкого та безпечного випуску реалізованого коду. Традиційний процес доставки

пов'язаний з небезпеками, що призводить до невдоволення як клієнтів, так і команд розробників.

«Безперервна доставка - це здатність безпечно, швидко та стабільно впроваджувати зміни всіх типів, включаючи нові функції, зміни конфігурації, виправлення помилок та експерименти, у виробництво або в руки користувачів».

Наведене вище визначення походить від гуру безперервної доставки Джеза Хамбла та Дейва Фарлі [13]. Воно пропонує базову концепцію конвеєра безперервної доставки (CD), що означає, що будь-який розробник може вносити зміни у продакшн і мати можливість виправляти їх пізніше.

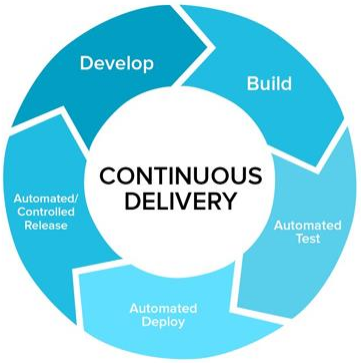


Рисунок 1.3 Безперервна доставка

Безперервна доставка (БД) – це процес розробки програмного забезпечення, який автоматично готує модифікації коду для випуску в робочу версію. Безперервна доставка (БД) – це важливий компонент сучасної розробки додатків, який базується на безперервній інтеграції шляхом розгортання змін коду в середовищі тестування або на етапі збірки, як видно з рисунка 2. На заключному етапі перевірки розгортання людина, автоматизований тест або бізнес-правило вирішують, чи слід перенести додаток на завершальний етап. Якщо ці кроки виконані правильно, розробники створили артефакт, підготовлений до випуску та пройшов набір стандартизованих процесів тестування.

CD дозволяє розробникам перевіряти оновлення програм у різних вимірах перед розгортанням для клієнтів, використовуючи автоматизацію тестування, яка виходить за рамки модульних тестів. Такі тести можуть включати, серед іншого,

тести надійності інтерфейсу користувача, навантаження та API (інтерфейсу прикладного програмування). Цей процес допомагає розробникам ефективно виявляти проблеми та перевіряти оновлення. Автоматизація налаштування та реплікації численних середовищ тестування є простою та доступною за допомогою хмари, тоді як раніше це було складно робити локально.

Безперервна інтеграція та безперервна доставка (CI/CD) – це метод постійної доставки додатків клієнтам шляхом додавання автоматизації до різних фаз розробки додатків. Безперервна інтеграція, безперервна доставка та безперервне розгортання – це три ідеї, пов’язані з CI/CD [14]. Це відповідь на проблеми, з якими можуть зіткнутися команди розробки та експлуатації під час об’єднання нового коду.

Зокрема, CI/CD забезпечує постійне відстеження та безперервну автоматизацію всього життєвого циклу програми, від етапів інтеграції та тестування до етапів доставки та випуску. Зрештою, команди розробки та експлуатації співпрацюють гнучко, використовуючи стратегію DevOps або SRE (Site Reliability Engineering). Ці взаємопов'язані практики називаються «конвеєром CI/CD».

Як показано на рисунку 1.4, конвеєр може бути використаний для представлення безперервної інтеграції та безперервної доставки (CI/CD), де новий код знаходиться на одному боці та оцінюється протягом кількох фаз, таких як вихідний код, збірка, тестування, проміжна розробка та виробництво. Після цього він випускається як код, готовий до виробництва, на іншому.

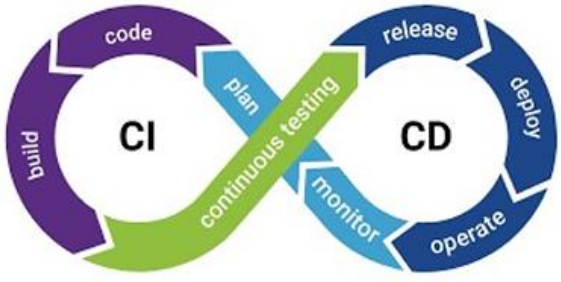


Рисунок 1.4 - Безперервна інтеграція та безперервна доставка

Метод доставки організований таким чином, що кожен крок конвеєра CI/CD є логічною сутністю. Кожен етап служить бар'єром, який перевіряє певний розділ коду. Мета полягає в тому, щоб - як

Коли код рухається по конвеєру, якість зросте, оскільки більше фрагментів коду буде перевірено, як видно на малюнку

Рисунок 1.4. Виявлення проблем на ранній стадії перешкоджає просуванню коду в конвеєрі. Результати тестів миттєво повідомляються команді, і якщо програмне забезпечення не проходить цей етап, усі наступні збірки та оновлення зупиняються.

1.4 Висновки по розділу

У першому розділі було розглянуто теоретичні основи автоматизованого тестування в CI/CD-конвеєрах. Проаналізовано принципи автоматизованого тестування програмного забезпечення, його роль у забезпеченні якості та стабільності програмних продуктів. Виконано огляд сучасних підходів і досліджень у сфері автоматизації тестування, що дозволило визначити основні тенденції розвитку цієї галузі. Також досліджено концепції безперервної інтеграції та безперервної доставки (CI/CD), їх значення для прискорення процесів розробки та підвищення ефективності тестування. Отримані результати формують теоретичну основу для подальшого аналізу методів та інструментів автоматизації тестування.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ В CI/CD

2.1 Методи та підходи до автоматизації тестування в CI/CD-конвеєрах

Це дослідження використовує експериментальну методологію для оцінки інтеграції та характеристик продуктивності п'яти інструментів тестування продуктивності з відкритим кодом - Apache JMeter , Grafana k6 , Gatling , Locust та Artillery - в рамках конвеєра CI/CD. Наш підхід включає проектування повністю автоматизованого конвеєра тестування, розробку реалістичних синтетичних профілів навантаження, систематичний моніторинг продуктивності та ретельну статистичну валідацію.

Для оцінки інструментів за різних умов навантаження ми реалізуємо три сценарії тестування продуктивності на основі класифікації Паргаонкара [16]:

— Типове навантажувальне тестування: імітує нормальне щоденне використання для встановлення базової продуктивності.

— Стрес-тестування : поступове збільшення навантаження на систему для виявлення точок зниження продуктивності або порогів збоїв.

— Пікове тестування : Застосовує раптове, екстремальне збільшення навантаження для оцінки стійкості та відновлення системи.

Кожен тип тестування застосовується однаково до всіх інструментів, збираючи такі основні показники:

— Середній час відгуку

— Пропускна здатність

— Коефіцієнт помилок

— Використання процесора

— Використання оперативної пам'яті

Ці показники дозволяють нам оцінювати продуктивність як на бізнес-рівні, так і на системному рівні стандартизованим способом.

Архітектура системи та інтеграція інструментів

На рисунку 2.1 показано архітектуру експериментальної системи. Конвеєр CI/CD організовано за допомогою Jenkins , обраного через його домінуюче використання (77% частки ринку) [15]. Тести продуктивності запускаються як автоматизовані етапи в рамках конвеєра.

Усі компоненти контейнеризовані за допомогою Docker для забезпечення узгодженості та відтворюваності середовища. cAdvisor використовується для моніторингу метрик на рівні контейнера, тоді як метрики на рівні хоста збираються за допомогою Windows Exporter . Стек моніторингу побудовано на Prometheus для збору та зберігання метрик, а також Grafana для візуалізації в режимі реального часу.

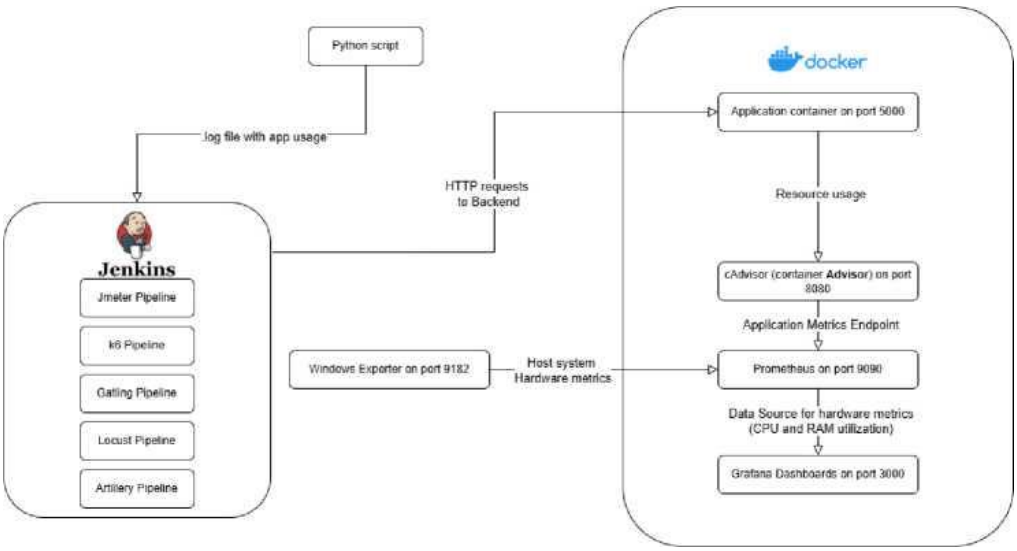


Рисунок 2.1 - Системна архітектура експериментального тестового стенду

Тестований застосунок — це система електронної комерції Spring Boot з відкритим кодом , обрана завдяки своїй реалістичній бізнес-логіці та реалізації серверної частини на Java. Системі виділено 4 ядра процесора та 4 ГБ оперативної пам'яті, а автоматичні перезапуски контейнерів запускаються у відповідь на витоки пам'яті, виявлені під час попередніх тестів.

Генерація синтетичного профілю навантаження: Оскільки тестований додаток Spring Boot не має доступу до реальних даних про використання у виробничому середовищі. Натхненні підходом, ми моделюємо поведінку

користувачів, генеруючи синтетичні журнали транзакцій за допомогою скриптів Python. Ці журнали моделюють поширені взаємодії в електронній комерції та використовуються для отримання профілів навантаження для тестування. Ми надаємо статистичні ваги для кожного виклику API, що використовується в профілі, генеруємо файл .log з викликами API протягом заданого періоду часу, зазвичай одного дня. Інший скрипт Python аналізує результуючий файл і надає ваги для транзакцій, що складаються з використаних викликів API [16]. Цей підхід забезпечує узгодженість і повторюваність між тестовими запусками та інструментами. Ці скрипти можна знайти в публічному репозиторії GitHub разом з усім вихідним кодом, використаним у цій роботі.

Деталі виконання інструменту: Незважаючи на відмінності в мовах сценаріїв та форматах конфігурації, кожен інструмент інтегровано в конвеєр для виконання тих самих тестових сценаріїв:

— Apache JMeter : плани тестування на основі XML виконуються через CLI.

— Grafana k6 : Тестові скрипти на основі JavaScript.

— Гатлінг : Кодові бази Scala/Java.

— Locust : Сценарії на основі Python.

— Артилерія : конфігурації на основі YAML.

Для забезпечення справедливості, тестові навантаження та рівні інтенсивності є однаковими для всіх інструментів. Інфраструктура моніторингу реєструє однакові показники для кожного запуску, що дозволяє здійснювати пряме порівняння.

Статистична валідація. Щоб оцінити значущість спостережуваних відмінностей у продуктивності, ми застосовуємо непараметричні статистичні тести через ненормальний розподіл зібраних даних:

— Ми перевіряємо нульову гіпотезу *H0* : медіани всіх груп рівні, проти альтернативної гіпотези Ні: принаймні одна медіана відрізняється .

— Для виявлення загальних відмінностей між інструментами для кожного показника використовується тест Краскела-Уолліса (рівняння 1).

— Якщо виявлено суттєві відмінності, ми застосовуємо U-критерій Манна-Вітні (рівняння 2.1) для попарних порівнянь, застосовуючи поправку Бонферроні для корекції на множинні порівняння.

Ми дотримуємося суворих порогів значущості для забезпечення надійності:

$p < 0{,}05$ для тесту Краскела-Уолліса

$$П = \frac{12}{N(N+1)} \sum_{i=1}^{k} \frac{R_i^2}{n_i} - 3(N+1) \quad (2.1)$$

— p < 0,005 для критерію Манна-Вітні (з поправкою Бонферроні)

Де:

— N: загальна кількість спостережень,

— k: кількість груп,

— n_i : спостереження в групі i,

— R_i : сума рангів у групі i.

$$\mathbf{U = n_1 n_2 + \frac{\left(n_{1(n_1-1)}\right)}{2} - R_1} \quad (2.2)$$

Де:

— *n1* , n2 : розміри вибірки двох груп ,

— Ri: сума рангів для групи 1.

Результати цих випробувань наведено в таблицях, що відповідають кожному сценарію випробування.

Наскрізне тестування. Комплексне (E2E) тестування – це форма тестування, яка перевіряє весь робочий процес програмного застосунку. E2E-тестування гарантує, що кожен компонент застосунку працює належним чином у реальному сценарії. Програмне забезпечення тестується шляхом відтворення сценарію кінцевого користувача, такого як інтерфейс користувача, серверні служби, бази даних та мережеві підключення. Типові дії в E2E-тестуванні – це

натискання кнопки, прочитання тексту та надсилання форми. Мета полягає в тому, щоб гарантувати загальну поведінку застосунку, таку як його функціональність, надійність, продуктивність та безпека [17].

Мета E2E-тестування полягає у виявленні проблем або недоліків, які можуть виникнути під час взаємодії різних компонентів програми один з одним. Воно охоплює всю систему, впроваджуючи наскрізне тестування кожного компонента, щоб визначити, чи всі вони працюють належним чином, а очікування користувачів відповідають дійсності. Наскрізне тестування часто проводиться після інтеграційного тестування, яке оцінює окремі модулі, та перед приймальним тестуванням користувача, яке підтверджує вимоги користувача.

Комплексне тестування має охоплювати всі процеси, включаючи комп'ютерну систему та бізнес-процеси, як видно на рисунку 2.2. Наприклад, тест процесу замовлення в системі онлайн-торгівлі має гарантувати прийняття покупки клієнта, а також запускати ідентифікацію, пакування та відправлення потрібних товарів клієнту. Однак, щоб транзакція була завершена, ці процеси повинні запускати інші процеси. Замовлення має згенерувати та надіслати правильний рахунок-фактуру, а платіж має точно відповідати рахунку-фактурі, щоб закрити транзакцію. Незважаючи на те, що сценарій простий, це досить складний набір взаємопов'язаних процесів, і існує затримка між розміщенням замовлення та оплатою рахунку-фактури.

Для відстеження замовлення в системі та понаднормової роботи буде потрібно повне тестування транзакції від початку до кінця, щоб переконатися, що всі необхідні взаємодії відбуваються правильно та призводять до успішного результату [18]. Хоча сценарій простий, це досить складний набір взаємозалежних процесів, а також існує часовий лаг між початковим замовленням та оплатою рахунку-фактури.

Цей тип тестування важливий, оскільки навіть невеликі зміни в одній системі можуть спричинити збої в іншій. E2E-тестування допомагає зменшити ризик того, що ці проблеми досягнуть кінцевих користувачів, перевіряючи всю систему з точки зору користувача.

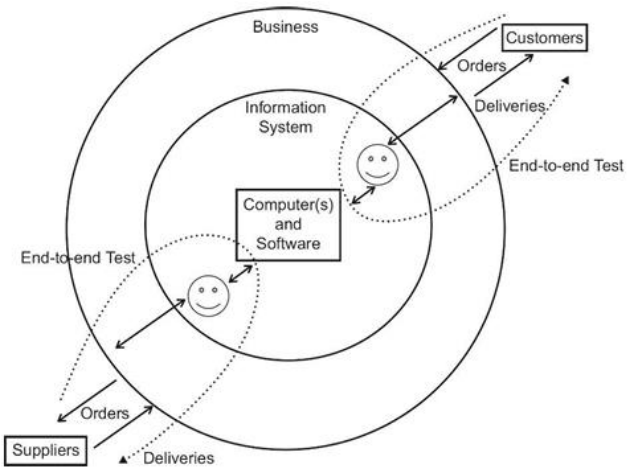


Рисунок 2.2 - Комплексне тестування транзакції в онлайн- магазині

Методи наскрізного тестування. Існує два основних способи проведення наскрізного тестування: горизонтальний та вертикальний. Обидва вони намагаються протестувати всю систему, але по-різному. Важливо з'ясувати, який з них найкраще працює для проекту, перш ніж розпочинати етап тестування. Хоча вони мають однакову мету - переконатися, що все працює від початку до кінця, вони по-різному підходять до цього з точки зору того, як тестування налаштовано та проведено.

Горизонтальне тестування

Горизонтальне наскрізне тестування передбачає комплексну оцінку застосунку шляхом оцінки його функціональності в усіх аспектах. Перед проведенням горизонтального тестування важливо розуміти, що весь робочий процес тестується в численних застосунках. Системи та підсистеми є частиною тестування, тому їх необхідно протестувати незалежно заздалегідь [19].

Існує безліч переваг використання горизонтального наскрізного тестування, включаючи його комплексний та широкий характер. Воно в першу чергу наголошує на зручності використання, забезпечуючи точне відображення досвіду кінцевого користувача. Крім того, якщо програмне забезпечення вимагає взаємодії з іншими програмами, цю інтерактивність також необхідно протестувати.

Більше того, коли підприємства використовують програмне забезпечення,

тестувальники також можуть перевірити взаємодію програмного забезпечення з цільовою базою даних та мережевою інфраструктурою. Горизонтальне тестування зазвичай проводиться як завершальний етап циклу випуску, що гарантує ретельну оцінку всіх аспектів програми.

Вертикальне тестування

Вертикальне наскрізне тестування розділяє застосунок на складові шари та продовжує незалежне тестування кожного шару в ієрархічному порядку [20]. Ці шари можуть включати, серед іншого, виклики бази даних інтерфейсу користувача та запити API.

Вертикальне тестування пропонує різні переваги, включаючи придатність для критично важливого для безпеки програмного забезпечення та здатність досягати широкого охоплення коду. Крім того, вертикальне тестування часто швидше, ніж горизонтальне наскрізне тестування.

Під час процесу тестування програмне забезпечення зазвичай оцінюється разом із рівнями інтерфейсу користувача, інтеграцією API та мережевою інфраструктурою. Хоча як вертикальний, так і горизонтальний підходи до тестування повинні враховувати всі відповідні деталі, вертикальне тестування використовує більш детальний підхід.

Істотна відмінність між вертикальним та горизонтальним тестуванням полягає в тому, що горизонтальне тестування може проводитися більш незалежно, без необхідності залучення інших зацікавлених сторін. Натомість, вертикальне тестування вимагає співпраці з різними зацікавленими сторонами, такими як потенційні власники продукту.

Дизайн комплексного тестування. Проектування комплексного тестування містить три категорії, такі як функції користувача, умова та тестові випадки. Ці категорії будуть пояснені більш детально.

Існує кілька кроків, які необхідно виконати для побудови користувацьких функцій як частини процесу наскрізного тестування. Необхідно перерахувати користувацькі функції програмних систем та взаємопов'язаних систем. Важливо відстежувати кожну операцію функції, а також вхідні та вихідні дані та розуміти

різні функції між різними з'єднаннями. Необхідно визначити природу кожної користувацької функції, незалежно від того, чи є вони незалежними чи придатними для повторного використання.

Наступний крок – встановити умови побудови на основі функцій користувача. Для кожної функції користувача слід підготувати набір умов, таких як час, стан даних та інші фактори, які можуть впливати на функцію користувача.

Зрештою, для створення тестового випадку необхідно створити один або декілька тестових випадків для тестування кожної функціональності. Умови слід перераховувати в різних тестових випадках. Ці кроки можуть допомогти тестувальникам визначити ефективнішу стратегію тестування та забезпечити ретельне тестування функцій користувача.

2.2 Інструменти автоматизованого тестування

У DevSecOps автоматизація має вирішальне значення для підтримки швидкості та узгодженості, одночасно забезпечуючи безпеку програм. Широкий спектр інструментів, доступних для автоматизованого тестування безпеки, допомагає розробникам виявляти вразливості на кожному етапі життєвого циклу розробки, від написання коду до розгортання. Ці інструменти можна класифікувати залежно від типу тестування, яке вони виконують, наприклад, як статичне тестування безпеки програм (SAST), динамічне тестування безпеки програм (DAST), аналіз складу програмного забезпечення (SCA) та сканування безпеки інфраструктури як коду (IaC).

Інструменти SAST виконують статичний аналіз вихідного коду, байт-коду або бінарних файлів, щоб знайти вразливості на ранніх етапах процесу розробки, без необхідності запуску програми. Ці інструменти перевіряють код на наявність таких проблем, як переповнення буфера, SQL-ін'єкції, міжсайтовий скриптинг (XSS) тощо. Вони надають негайний зворотний зв'язок розробникам, дозволяючи їм виправляти недоліки безпеки, перш ніж вони поширяться на пізніші етапи конвеєра розробки [20]. SonarQube - популярна платформа з відкритим кодом, яка

аналізує код на наявність вразливостей та проблем з якістю коду. Вона добре інтегрується в конвеєри CI/CD та підтримує кілька мов. Checkmarx — комерційний інструмент, розроблений для глибокого аналізу коду та виявлення вразливостей, особливо у складних, великомасштабних програмах. Fortify Static Code Analyzer - широко використовуваний корпоративний інструмент, який сканує як вихідний код, так і бінарні файли, пропонуючи широке охоплення вразливостей безпеки. Veracode - хмарне рішення, яке забезпечує автоматизований SAST для виявлення вразливостей та недоліків безпеки протягом усього життєвого циклу розробки програмного забезпечення. Інструменти SAST інтегровані в ранні етапи конвеєрів CI/CD, зазвичай запускаючись під час фази фіксації або збірки, що робить їх важливими для зміщення безпеки, ліворуч у SDLC.

На відміну від SAST, інструменти динамічного тестування безпеки додатків (DAST) тестують додаток у його запущеному стані, імітуючи реальні атаки, щоб виявити вразливості під час виконання, які можуть бути невидимими у вихідному коді. Інструменти DAST виконують тестування на проникнення, взаємодіючи з інтерфейсом користувача (UI) додатка або API, та виявляючи такі проблеми, як зламана автентифікація, неправильні конфігурації безпеки та витік даних. OWASP ZAP (Zed Attack Proxy) Інструмент тестування на проникнення з відкритим кодом, розроблений для пошуку вразливостей у веб-додатках. Він широко використовується для DAST та легко інтегрується з інструментами CI/CD. Burp Suite Популярна платформа для тестування безпеки веб-додатків, що використовується для ручного та автоматизованого сканування вразливостей веб-додатків. Автоматизований сканер Burp Suite може виявляти такі проблеми, як SQL-ін'єкції, XSS та інші вразливості. Acunetix Автоматизований інструмент сканування вразливостей, розроблений для виявлення та зменшення вразливостей веб-додатків, таких як SQL-ін'єкції та XSS. Його можна інтегрувати з конвеєрами CI/CD для безперервного тестування безпеки. Netsparker Автоматизований сканер безпеки веб-додатків, який може виявляти широкий спектр вразливостей, включаючи SQL-ін'єкції, XSS та інші критичні вразливості веб-додатків.

Інструменти DAST зазвичай використовуються на пізніх етапах розробки, наприклад, під час проміжного або передпродакшн-середовища, оскільки для проведення тестування вони вимагають активного запусків програми.

Інструменти аналізу складу програмного забезпечення (SCA) зосереджені на виявленні та управлінні ризиками безпеки, пов'язаними з компонентами з відкритим кодом та сторонніми бібліотеками, що використовуються в розробці додатків [21]. Ці інструменти автоматично сканують на наявність відомих вразливостей, проблем з ліцензуванням та застарілих версій залежностей. Snyk - інструмент, який сканує залежності з відкритим кодом на наявність відомих вразливостей, допомагаючи розробникам виправляти проблеми, надаючи автоматичні виправлення або оновлення. Black Duck - рішення, яке виявляє вразливості безпеки у програмному забезпеченні з відкритим кодом, відстежує ліцензії та контролює стан безпеки всіх залежностей. OWASP Dependency-Check - інструмент аналізу складу програмного забезпечення, який перевіряє наявність відомих вразливостей у сторонніх бібліотеках та надає детальні звіти. WhiteSource — інструмент, який забезпечує автоматизоване керування компонентами з відкритим кодом, включаючи виявлення вразливостей у режимі реального часу та моніторинг відповідності ліцензіям. Ці інструменти інтегровані в процес збірки та забезпечують видимість безпеки компонентів з відкритим кодом, які часто ігноруються в традиційному тестуванні безпеки.

Інфраструктура як код (IaC) стає дедалі популярнішою для управління хмарною інфраструктурою, але неправильні конфігурації в скриптах IaC можуть призвести до серйозних вразливостей безпеки [22]. Інструменти безпеки IaC допомагають автоматизувати виявлення таких вразливостей шляхом сканування коду інфраструктури (наприклад, Terraform,

CloudFormation або скрипти Ansible) на наявність неправильних конфігурацій, небезпечних налаштувань та потенційних ризиків безпеці. Checkov — інструмент статичного аналізу коду для Terraform, CloudFormation, Kubernetes та Dockerfiles, розроблений для виявлення неправильних конфігурацій у скриптах IaC перед розгортанням. TFLint - інструмент аналізу коду для Terraform, який

сканує код на наявність потенційних вразливостей безпеки та порушень найкращих практик. TerraScan - інструмент з відкритим кодом, який сканує IaC на наявність порушень відповідності та неправильних конфігурацій безпеки в кількох хмарних середовищах. Puppet - інструмент керування конфігурацією, який автоматизує управління інфраструктурою та включає перевірки безпеки для IaC[22]. Інструменти безпеки IaC забезпечують необхідну автоматизацію для перевірки засобів контролю безпеки в хмарних середовищах, забезпечуючи відповідність та запобігаючи неправильним конфігураціям, які можуть призвести до витоків даних або експлуатації ресурсів.

Контейнеризовані додатки потребують іншого підходу до безпеки, враховуючи їхню динамічну та ефемерну природу. Інструменти безпеки контейнерів автоматично сканують образи контейнерів, конфігурації та поведінку під час виконання, щоб виявити вразливості, неправильні конфігурації та потенційні ризики безпеці. Aqua Security Комплексна платформа безпеки контейнерів, яка забезпечує сканування образів контейнерів та моніторинг безпеки під час виконання. Trivy Інструмент з відкритим кодом, який сканує образи контейнерів на наявність вразливостей як у залежностях додатків, так і в операційній системі. Clair Інструмент сканування вразливостей контейнерів, який виявляє відомі вразливості в образах контейнерів, порівнюючи їх із загальнодоступними базами даних вразливостей. Kube-bench Інструмент, який використовується для перевірки конфігурацій безпеки кластерів Kubernetes на відповідність бенчмарку CIS Kubernetes, допомагаючи забезпечити безпечну оркестрацію контейнерів. Ці інструменти інтегровані в конвеєр CI/CD, щоб гарантувати безпеку контейнерів перед розгортанням у робочому середовищі.

Автоматизація перевірок відповідності вимогам безпеки в конвеєрах CI/CD є важливою для забезпечення відповідності всіх практик розробки та розгортання стандартам безпеки та нормативним вимогам. Інструменти «Політика як код» допомагають визначати, застосовувати та автоматизувати перевірки відповідності політик безпеки в інфраструктурі та додатках організації. Open Policy Agent (OPA) - механізм політик, який дозволяє користувачам писати

власні політики декларативною мовою високого рівня, яку можна застосовувати до всіх аспектів розробки додатків, включаючи контроль доступу та управління інфраструктурою. HashiCorp Sentinel - фреймворк «політика як код» для визначення, забезпечення виконання та автоматизації політик безпеки та відповідності в хмарних середовищах, включаючи Terraform та Kubernetes. Cloud Custodian — механізм правил, який використовується для забезпечення виконання політик безпеки та відповідності в хмарних ресурсах, автоматизуючи перевірки та виправлення в AWS, Azure та GCP. Ці інструменти дозволяють автоматизувати безпеку та відповідність у великих масштабах, дозволяючи командам DevSecOps застосовувати політики безпеки без ручного втручання.

Стратегія впровадження інтеграції автоматизації безпеки в конвеєр CI/CD за допомогою використання практик DevSecOps включає кілька ключових кроків, які забезпечують вбудовану безпеку в конвеєр на кожному етапі життєвого циклу розробки програмного забезпечення [23]. Ця стратегія спрямована на безперешкодну інтеграцію безпеки в практики DevOps, зменшення вразливостей, забезпечення відповідності та оптимізацію розробки та розгортання безпечного програмного забезпечення.

Налаштування безпечного конвеєра CI/CD є важливим для інтеграції практик безпеки в життєвий цикл розробки програмного забезпечення. Це гарантує послідовне застосування заходів безпеки, автоматизує тестування безпеки та зменшує ризик проникнення вразливостей у виробництво. Першим критично важливим кроком є зсув безпеки вліво, тобто практики безпеки повинні бути інтегровані на ранніх етапах процесу розробки. Цього можна досягти шляхом включення інструментів статичного тестування безпеки додатків (SAST) , таких як SonarQube , на етапі фіксації коду, щоб якомога раніше виявити потенційні вразливості. Аналогічно, інструменти аналізу складу програмного забезпечення (SCA), такі як Snyk, слід використовувати для сканування відомих вразливостей у бібліотеках з відкритим кодом з самого початку.

Тести безпеки слід автоматизувати на кількох етапах конвеєра, від інтеграції коду до збірки, тестування, проміжної розробки та розгортання. Таке

безперервне тестування гарантує, що проблеми безпеки виявляються на кожному етапі, а не лише в кінці [24]. Крім того, інструменти динамічного тестування безпеки додатків (DAST) , такі як OWASP ZAP , можуть бути інтегровані в проміжне середовище для імітації реальних атак, що додатково захищає додаток до його потрапляння у виробництво. Для захисту конфіденційних даних слід використовувати інструменти керування секретами , такі як HashiCorp Vault, для безпечного зберігання та керування обліковими даними, такими як ключі API, гарантуючи, що вони ніколи не будуть жорстко закодовані в кодовій базі. Крім того, такі інструменти, як Open Policy Agent (OPA), можуть застосовувати політики безпеки, гарантуючи дотримання найкращих практик безпеки, наприклад, забезпечуючи відповідність конфігурацій інфраструктури організаційним та нормативним стандартам. Налаштувавши безпечний конвеєр CI/CD, організації можуть автоматизувати тестування, моніторинг та забезпечення застосування засобів контролю безпеки, зменшуючи ручне втручання та забезпечуючи більш надійний та послідовний підхід до безпеки програмного забезпечення.

Вибір відповідних інструментів та точок їх інтеграції в конвеєр CI/CD є вирішальним аспектом створення ефективного та безпечного конвеєра. Першим кроком у цьому процесі є визначення вимог безпеки, специфічних для середовища організації. Це включає визначення того, чи необхідне статичне тестування безпеки додатків (SAST) для аналізу коду, динамічне тестування безпеки додатків (DAST) для сканування вразливостей під час виконання, чи аналіз складу програмного забезпечення (SCA) для управління залежностями програмного забезпечення з відкритим кодом.

Після того, як вимоги чітко визначені, наступним кроком є вибір правильних інструментів для кожного етапу конвеєра. Наприклад, інструменти SAST , такі як SonarQube або Checkmarx, можуть бути інтегровані в процес фіксації коду , скануючи вихідний код на наявність вразливостей, щойно розробник вносить свої зміни. Інструменти DAST , такі як OWASP ZAP, можуть бути впроваджені в проміжному середовищі для оцінки стану безпеки програми

під час тестування або передпродакційного розгортання. Інструменти SCA , такі як Snyk, можуть бути інтегровані під час фази збірки для моніторингу та постійного оновлення залежностей з відкритим кодом, гарантуючи, що вони не містять відомих вразливостей. Точки інтеграції повинні бути ретельно визначені в конвеєрі CI/CD, щоб забезпечити виконання сканувань безпеки у потрібний час. Ці точки інтеграції зазвичай відбуваються на етапах фіксації , збірки , тестування , проміжної розробки та розгортання . Інструменти повинні бути налаштовані на автоматичний запуск у цих точках, що зменшує ризик людської помилки та забезпечує постійне забезпечення безпеки протягом усього життєвого циклу розробки. Крім того, слід забезпечити сумісність між вибраними інструментами та існуючими платформами CI/CD (такими як Jenkins , GitLab , CircleCI або Azure DevOps). Більшість сучасних інструментів DevSecOps надають плагіни або API для безперешкодної інтеграції з цими системами CI/CD, оптимізуючи процес та підтримуючи ефективність робочого процесу.

Скрипти автоматизації є центральними для впровадження безпечного конвеєра DevSecOps, що дозволяє автоматизувати виконання інструментів безпеки та інтеграцію практик безпеки по всьому конвеєру. Ці скрипти, часто написані мовами сценаріїв, такими як Bash , Python або Groovy (для Jenkins), допомагають оптимізувати процес тестування безпеки, автоматизуючи такі завдання, як запуск сканування, перевірка залежностей та забезпечення дотримання політик безпеки.

Скрипти можуть автоматизувати тести безпеки на різних етапах розробки. Наприклад, можна створити скрипт SAST для автоматичного запуску аналізу SonarQube щоразу, коли розробник додає код, тоді як скрипт DAST може автоматично викликати OWASP ZAP для запуску тестів безпеки програми в проміжному середовищі [25]. Крім того, можна написати скрипти SCA для виклику таких інструментів, як Snyk, під час процесу збірки, щоб перевірити наявність вразливостей у залежностях з відкритим кодом. Для організацій, які використовують інструменти «Інфраструктура як код» (IaC), такі як Terraform або CloudFormation , слід створити скрипти автоматизації для сканування

конфігурацій IaC на наявність неправильних налаштувань. Такі інструменти, як Checkov, можна інтегрувати в ці скрипти, автоматично перевіряючи наявність недоліків безпеки перед внесенням будь-яких змін в інфраструктуру.

Ще одним важливим аспектом автоматизації є політика як код . Скрипти автоматизації також можуть застосовувати політики безпеки та відповідності за допомогою таких інструментів, як Open Policy Agent (OPA) [26]. Ці скрипти гарантують, що будь-які конфігурації або ресурси, розгорнуті через конвеєр, відповідають стандартам безпеки організації, запобігаючи впровадженню будь-яких несанкціонованих або небезпечних конфігурацій у середовище. Автоматизуючи ці процеси, організації гарантують послідовне застосування безпеки на кожному етапі конвеєра, зменшуючи ймовірність помилок, внесених вручну, та покращуючи загальний стан безпеки.

Моніторинг та звітність є критично важливими компонентами безпечного конвеєра DevSecOps, оскільки вони забезпечують постійний нагляд за ризиками безпеки та вразливостями протягом усього життєвого циклу програмного забезпечення. Інструменти постійного моніторингу, такі як Falco та Sysdig, можуть бути інтегровані для моніторингу поведінки під час виконання на предмет підозрілої активності або порушень безпеки. Ці інструменти можуть відстежувати системні виклики, мережеву активність та поведінку контейнерів у режимі реального часу, допомагаючи виявляти потенційні інциденти безпеки, як тільки вони виникають.

Автоматизовані звіти безпеки повинні генеруватися на кожному етапі конвеєра, що підсумовують результати різних сканувань безпеки. Такі інструменти, як SonarQube , OWASP ZAP та Snyk, автоматично генерують детальні звіти про вразливості, проблеми з якістю коду та ризики безпеки відкритого коду. Ці звіти слід надавати відповідним зацікавленим сторонам, таким як розробники, команди безпеки та керівники проектів, щоб забезпечити оперативне вирішення проблем безпеки. Для візуалізації та відстеження стану безпеки програми можна створювати панелі інструментів за допомогою таких платформ, як Prometheus та Grafana . Ці панелі інструментів надають інформацію

про показники безпеки, тенденції та вразливості в режимі реального часу, надаючи командам повне уявлення про загальний стан безпеки програми. Окрім моніторингу та панелей інструментів, автоматичні сповіщення є важливими для своєчасного реагування на критичні проблеми безпеки. Сповіщення повинні бути налаштовані для повідомлення команд про вразливості або інциденти високого пріоритету, і ці сповіщення можуть бути інтегровані в такі платформи, як Slack або PagerDuty , для швидкого сповіщення. Повинен бути розроблений ефективний план реагування на інциденти, що автоматизує процес сортування та вирішення інцидентів безпеки.

Зрештою, для цілей дотримання вимог та регулювання, організації повинні створювати журнали аудиту та звіти про відповідність . Вони можуть створюватися автоматично за допомогою таких інструментів, як Cloud Custodian або OPA , що гарантує відповідність програми та інфраструктури галузевим стандартам, таким як GDPR, HIPAA або PCI-DSS. Автоматизована звітність про відповідність допомагає організації швидко реагувати на аудити та демонструвати наявність засобів контролю безпеки. Встановлюючи надійні механізми моніторингу та звітності, організації можуть забезпечити постійну оцінку безпеки та своєчасне виявлення та зменшення будь-яких загроз безпеці.

2.3 Принципи інтеграції тестування безпеки в DevSecOps-процеси

Автоматизація безпеки DevSecOps побудована на фундаментальному принципі інтеграції безпеки в кожен етап життєвого циклу розробки програмного забезпечення (SDLC) з акцентом на автоматизацію **,** співпрацю та постійний зворотний зв'язок . Вбудовуючи засоби контролю безпеки безпосередньо в конвеєри CI/CD, організації можуть виявляти вразливості на ранній стадії, зменшувати ручні зусилля, забезпечувати дотримання політик та підтримувати швидкість розробки.

Інтеграція безпеки в конвеєри CI/CD є фундаментальною практикою в DevSecOps, яка гарантує виявлення та усунення вразливостей на ранніх етапах

життєвого циклу розробки програмного забезпечення. Ця інтеграція передбачає вбудовування інструментів та перевірок безпеки безпосередньо в автоматизовані робочі процеси, завдяки чому безпека стає невід'ємною та ненав'язливою частиною процесу розробки. Звичайні етапи, на яких вбудовується безпека, включають етап фіксації вихідного коду, фази збірки та тестування, а також процес розгортання. Наприклад, після того, як розробник фіксує код у репозиторії, сервер CI, такий як Jenkins, GitLab CI або GitHub Actions, може автоматично запускати сканери безпеки, такі як SAST, SCA або інструменти виявлення секретів. Роблячи це, розробники отримують негайний зворотний зв'язок, що дозволяє їм вирішувати проблеми безпеки, не чекаючи окремих перевірок безпеки. Крім того, автоматизовані шлюзи безпеки можна налаштувати для блокування збірок або розгортань, які не проходять критичні перевірки безпеки, гарантуючи, що через конвеєр просувається лише безпечний код. Ця безшовна інтеграція знижує вартість та складність ручних перевірок та узгоджує безпеку зі швидкістю сучасних циклів розробки.

Моделювання загроз та оцінка ризиків – це проактивні методи безпеки, спрямовані на виявлення, класифікацію та пом’якшення потенційних загроз безпеці до того, як вони проявляться у виробничому середовищі. У рамках DevSecOps ці методи зазвичай впроваджуються на ранніх етапах проектування або планування та можуть постійно оновлюватися в міру розвитку системи. Автоматизовані інструменти моделювання загроз, такі як

IriusRisk або Microsoft Threat Modeling Tool дозволяє командам відображати компоненти програми, потоки даних та межі довіри, щоб виявляти поверхні атаки та слабкі місця. Після виявлення загроз інструменти або фреймворки оцінки ризиків, такі як STRIDE або DREAD, можуть допомогти визначити пріоритети проблем на основі ймовірності та впливу. Ці дані допомагають інтегрувати цільові засоби контролю безпеки та механізми тестування в конвеєр CI/CD. Автоматизація частин цього процесу допомагає організаціям підтримувати послідовний, повторюваний аналіз ризиків, особливо у великомасштабних середовищах з частими архітектурними змінами. Це також

допомагає командам розробників зосередити свої зусилля щодо безпеки на областях високого ризику, роблячи загальний процес більш ефективним та результативним.

Статичне тестування безпеки додатків (SAST) є однією з основних методик у DevSecOps для аналізу вихідного коду, байт-коду або бінарних файлів додатка з метою виявлення вразливостей безпеки без запуску програми. Інструменти SAST працюють на найраніших етапах життєвого циклу розробки, зазвичай під час перевірки коду або етапів збірки, що робить їх ідеальними для раннього виявлення дефектів. Ці інструменти сканують кодову базу на наявність поширених вразливостей, таких як SQL-ін'єкції, міжсайтовий скриптинг (XSS), небезпечне використання API та жорстко закодовані облікові дані. Популярні інструменти SAST включають SonarQube, Fortify Static Code Analyzer, Checkmarx та Veracode [27]. Основною перевагою SAST є те, що він дозволяє розробникам отримувати зворотний зв'язок у режимі реального часу безпосередньо у своїх IDE або конвеєрах CI/CD, що дозволяє швидко виправляти проблеми, перш ніж код перейде до наступних етапів. Крім того, розширені інструменти SAST підтримують налаштування правил, інтеграцію CI/CD та автоматизовану звітність, що дозволяє командам адаптувати сканування відповідно до політик безпеки, специфічних для проекту. Постійно застосовуючи SAST по всьому конвеєру, організації можуть значно зменшити кількість вразливостей, що впроваджуються у продакшн, покращуючи як якість коду, так і загальний стан безпеки.

Динамічне тестування безпеки застосунків (DAST) включає аналіз запущеного застосунку для виявлення вразливостей безпеки, які можуть бути невидимими за допомогою статичного аналізу. На відміну від SAST, який перевіряє вихідний код, DAST працює за принципом «чорної скриньки», імітуючи атаки з точки зору зовнішнього користувача без знання внутрішньої структури застосунку. Інструменти DAST зазвичай інтегруються на пізніших етапах конвеєра CI/CD, таких як проміжні або передпродакшн-середовища, де застосунок розгортається та функціонує. Ці інструменти виявляють поширені

вразливості під час виконання, такі як пошкоджена автентифікація, неправильна обробка помилок, міжсайтовий скриптинг (XSS) та SQL-ін'єкції, взаємодіючи з застосунком через його інтерфейс користувача або API. Такі інструменти, як OWASP ZAP, Burp Suite та Netsparker, широко використовуються в цій категорії. DAST забезпечує вирішальний рівень безпеки, виявляючи проблеми, що виникають під час виконання, включаючи неправильні конфігурації, помилки сервера та небезпечні інтеграції третіх сторін. Його автоматизація в конвеєрі CI/CD гарантує, що кожна збірка проходить ретельну оцінку під час виконання, що дозволяє організаціям виявляти вразливості, які проявляються лише в реальних умовах. Однак, DAST може вимагати більше часу для виконання, ніж SAST, і потребує добре структурованого середовища для точного сканування, але його переваги у виявленні реальних загроз є безцінними в конвеєрах DevSecOps.

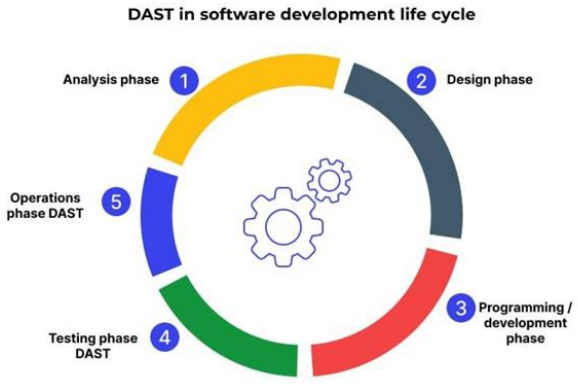


Рисунок 2.3 - Динамічне тестування безпеки застосунків (DAST)

Інструменти аналізу складу програмного забезпечення (SCA) відіграють життєво важливу роль у забезпеченні безпеки програм, створених з використанням бібліотек з відкритим кодом та сторонніх бібліотек. Сучасні розробки програмного забезпечення часто значною мірою залежать від зовнішніх компонентів, які можуть призвести до появи відомих вразливостей, якщо ними не керувати належним чином. Інструменти SCA автоматично сканують дерево залежностей проекту, щоб виявити пакети з відкритим кодом, їх версії та будь-які відомі вразливості, пов'язані з ними. Такі інструменти, як Snyk, Black Duck,

WhiteSource та OWASP Dependency-Check, є популярними рішеннями, які інтегруються безпосередньо в конвеєри збірки або IDE. Ці інструменти порівнюють виявлені пакети з публічними базами даних вразливостей, такими як Національна база даних вразливостей (NVD), або власними джерелами, для створення звітів та оцінок ризиків. Інструменти SCA також можуть відстежувати проблеми відповідності ліцензіям, що є критично важливим для організацій, яким необхідно дотримуватися правових та регуляторних стандартів [28]. Автоматизуючи SCA в конвеєрах CI/CD, розробники отримують ранні попередження про вразливі або застарілі бібліотеки, що дозволяє їм оновлювати або замінювати компоненти перед розгортанням у робочому середовищі. Це забезпечує як безпеку, так і відповідність законодавству в динамічних - середовищах розробки.

Сканування «інфраструктура як код» (IaC) є важливим для захисту базової інфраструктури, яка підтримує сучасні програми, особливо в хмарних та контейнерних середовищах. Інструменти IaC, такі як Terraform, AWS CloudFormation та Ansible, дозволяють визначати та керувати інфраструктурою за допомогою коду з контролем версій. Однак неправильні конфігурації в цих файлах, такі як відкриті порти, надмірно дозвільні ролі IAM або відсутність шифрування, можуть створювати критичні вразливості безпеки. Інструменти сканування IaC, такі як Checkov, TFLint, TerraScan та AWS Config, автоматично аналізують ці скрипти на наявність небезпечних конфігурацій та порушень найкращих практик. Ці інструменти можна інтегрувати в конвеєр CI/CD для сканування визначень інфраструктури на етапах збірки або планування. Коли виявляються проблеми , їх можна позначити як збої збірки або анотувати для розгляду розробниками. Сканування IaC також підтримує фреймворки «політика як код», що дозволяє командам безпеки визначати власні політики управління, які забезпечують дотримання вимог у всіх розгортаннях інфраструктури. Автоматизуючи сканування IaC, організації можуть забезпечити оцінку безпеки кожної зміни в інфраструктурі перед її впровадженням, зменшуючи ризик неправильних конфігурацій та покращуючи загальне управління хмарними

ресурсами.

Контейнери та платформи оркестрації контейнерів, такі як Docker та Kubernetes, стали основою сучасних хмарних застосунків завдяки своїй масштабованості, ефективності та узгодженості в різних середовищах [29]. Однак контейнери створюють унікальні проблеми безпеки, які необхідно вирішувати за допомогою автоматизованих практик безпеки в конвеєрі DevSecOps. Безпека контейнерів та оркестрації включає сканування образів контейнерів, моніторинг поведінки контейнерів під час виконання та забезпечення дотримання політик безпеки в межах рівня оркестрації. Автоматизовані інструменти безпеки, такі як Trivy **,** Clair та Aqua Security , призначені для сканування образів контейнерів на наявність вразливостей, неправильних конфігурацій та застарілих компонентів перед розгортанням. Ці інструменти аналізують образи контейнерів у базах даних вразливостей, щоб виявити відомі CVE (загальні вразливості та експозиції) в операційних системах, бібліотеках та залежностях. Сканування часто інтегровано в конвеєр CI/CD, що гарантує розгортання лише безпечних образів у продакшені. Крім того, такі інструменти, як Kube-bench та Kubesec, використовуються для оцінки конфігурацій безпеки кластерів Kubernetes. Вони перевіряють відповідність найкращим практикам (наприклад, контроль доступу на основі ролей (RBAC), мережеві політики) та виявляють слабкі місця, які можуть призвести до несанкціонованого доступу або витоків даних.

Безпека під час виконання також є критично важливою, і інструменти моніторингу контейнерів, такі як Falco та Sysdig**,** можна використовувати для виявлення аномальної поведінки в контейнерах або кластерах Kubernetes під час роботи. Ці інструменти забезпечують виявлення загроз у режимі реального часу, аналізуючи системні виклики та інші дані під час виконання, щоб виявити шкідливу діяльність, таку як підвищення привілеїв або спроби несанкціонованого доступу. Крім того, такі інструменти, як Twistlock (тепер частина Palo Alto Networks Prisma Cloud), надають більш розширені функції безпеки, включаючи керування вразливостями, захист під час виконання та моніторинг відповідності як для контейнерів, так і для безсерверних додатків. Автоматизуючи перевірки

безпеки контейнерів та оркестрації в рамках конвеєра DevSecOps, організації можуть гарантувати, що їхні контейнеризовані додатки постійно скануються на наявність вразливостей та безпечно налаштовуються, від етапу збірки до розгортання та виконання.

Політика як код (PaC) – це нова практика в DevSecOps, яка дозволяє визначати, версіонувати та застосовувати правила безпеки та відповідності у формі коду. Завдяки PaC політики безпеки більше не є абстрактними або не застосовуються вручну через окремі робочі процеси. Натомість вони вбудовані безпосередньо в конвеєр CI/CD, що гарантує автоматичне спрацьовування та застосування перевірок відповідності щоразу, коли вносяться зміни в інфраструктуру або розгортається код. Такі інструменти, як Open Policy Agent (OPA) та HashiCorp Sentinel, зазвичай використовуються для впровадження політики як коду. Ці інструменти дозволяють командам писати політики, які регулюють усе, від контролю доступу та управління ідентифікацією до шифрування та конфігурації мережі, безпосередньо в конвеєр як виконуваний код. Коли вводиться новий код або зміни в інфраструктуру, ці політики автоматично оцінюються та перевіряються на відповідність набору попередньо визначених правил [30]. Якщо зміна порушує політику, конвеєр зупиняється, а проблема позначається для виправлення. Цей автоматизований підхід забезпечує безперервне застосування політик та зменшує ручний нагляд, необхідний для аудитів відповідності.

PaC також добре інтегрується з інструментами сканування «Інфраструктура як код» (IaC) , оскільки політики безпеки можна застосовувати до визначень інфраструктури та файлів конфігурації перед розгортанням. Наприклад, команди можуть застосовувати політики, які обмежують використання певних сервісів AWS, вимагають шифрування для всіх даних, що зберігаються, або встановлюють суворі налаштування контролю доступу для хмарних ресурсів. Таким чином, PaC допомагає організаціям підтримувати постійну відповідність галузевим стандартам, таким як GDPR, HIPAA, PCI-DSS та SOC 2, без ручного втручання. На додаток до політик безпеки, вимоги до

відповідності, які часто встановлюються регуляторними органами, можна автоматизувати та перевіряти в рамках конвеєрів DevSecOps за допомогою PaC. Це спрощує процес забезпечення відповідності, зменшує ризик помилок та покращує загальний стан безпеки організації. Завдяки постійному забезпеченню дотримання правил безпеки та відповідності, «Політика як код» допомагає організаціям досягти проактивної стратегії безпеки, яка масштабується разом з їхніми процесами розробки.

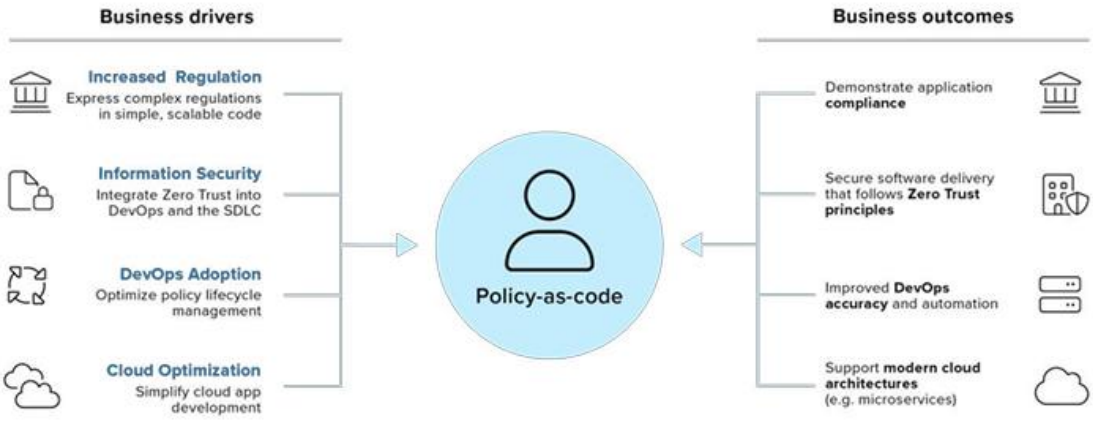


Рисунок 2.4 - Політика як код та автоматизація відповідності

2. 4 Висновки по розділу

У другому розділі було досліджено методи та інструменти автоматизованого тестування в CI/CD-конвеєрах. Розглянуто основні підходи до автоматизації тестування, що застосовуються на різних етапах життєвого циклу програмного забезпечення. Проведено аналіз сучасних інструментів автоматизованого тестування, їх функціональних можливостей та особливостей інтеграції в CI/CD-середовище. Окрему увагу приділено принципам інтеграції тестування безпеки в DevSecOps-процеси, що дозволяє підвищити рівень захищеності програмних систем. У результаті дослідження визначено ефективні підходи до побудови автоматизованих процесів тестування в сучасних програмних проєктах.

РОЗДІЛ 3. РОЗРОБКА, ІНТЕГРАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РІШЕНЬ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

3.1 Реалізація автоматизованого тестування в CI/CD-конвеєрі

Одним із методів тестування програмного забезпечення є автоматизоване тестування, яке використовує спеціалізовані інструменти для запуску тестових скриптів без втручання людини. Початкове створення скрипта для автоматизованого тестування вимагає людської праці, але наступні процеси, такі як порівняння фактичних та очікуваних результатів тестів, виконуються автоматично. Це найбільш підходяща стратегія для підвищення ефективності, продуктивності та охоплення тестування програмного забезпечення. Інструмент автоматизованого тестування спрощує тестування даних, обробку реалізації тестів та порівняння фактичних результатів з очікуваними.

Існує два різних методи виконання завдань або робочих елементів: гнучкий та каскадний. Каскадний підхід у розробці програмного забезпечення можна вважати одноразовою подією. На першому кроці важливо спланувати та спроектувати, потім розробити, і як завершальний крок обов'язково оцінити готовий продукт, щоб забезпечити його якість та внести необхідні корективи. Припускаючи, що якщо планування та інженерія були виконані належним чином, проект повинен мати кінцеву функцію, як задумано. Крім того, незначні помилки коду можна швидко виправити [31]. Цей метод дозволяє перевірити кінцевий продукт лише один раз, щоб побачити, чи поводиться він належним чином. Єдиний раз, коли тест слід повторити для підтвердження ремонту, це якщо він не спрацює, і буде зроблено виправлення. Дешевше та простіше проводити тести безпосередньо, ніж автоматично, якщо вони проводяться лише один або два рази.

Для складних проектів каскадний підхід втратив свою ефективність через зміни в технологіях та вимогах компанії протягом багатьох років. Тому цінніше отримати зворотний зв'язок від клієнта та оперативно реагувати відповідно до нього, ніж дотримуватися початкового плану. У індустрії програмного

забезпечення швидкі цикли поставок стали більш поширеними, коли компанії випускають нові функції та виправлення проблем кілька разів на день.

Гнучкий підхід розподіляє завдання на кілька фаз для управління проєктом. Постійне вдосконалення потрібне на кожному кроці, а також постійна співпраця із зацікавленими сторонами.

Після початку проекту команди циклічно проходять фазу планування, виконання та оцінки процедури. Безперервна співпраця між членами команди та зацікавленими сторонами проекту має бути постійною. Чотири основні принципи гнучкого методології (Agilemanifesto) – це люди та взаємодія, а не процеси та інструменти, робоче програмне забезпечення, а не комплексна документація, співпраця з клієнтами, а не переговори щодо контрактів, та реагування на зміни.

Перехід від традиційного водоспадного підходу до гнучкого підходу до автоматизації тестування був основним фактором у переході від водоспадного до гнучкого методу розробки програмного забезпечення. Це було пов'язано з необхідністю автоматизації тестування, яка вимагає, щоб кінцевий результат відповідав специфікації лише один раз. З часом водоспадний підхід ставав дедалі складнішим і дорожчим, і більшість програмних проектів стали настільки складними, що неможливо було заздалегідь спланувати та завершити всі технічні деталі. В останні роки індустрія програмного забезпечення перейшла від одноразових програмних проектів до швидких циклів доставки, при цьому деякі компанії постачають нові функції.

На передовій зв'язку зі споживачами технології використовуються як конкурентна зброя, що підвищує потребу в швидкості, а також в економії ресурсів і витрат. Автоматизація використовується підприємствами для скорочення витрат на тестування та часу виходу на ринок. Помилки програмного забезпечення можуть коштувати мільйони, якщо не мільярди доларів, а в певних ситуаціях цілі підприємства були знищені. Автоматизація не зменшить нестабільність та збої програмного забезпечення, якщо вам не вистачає персоналу або часу для проведення відповідного тестування. Надійність програмного забезпечення має бути на першому місці, а потім у центрі уваги має бути оптимізація часу та витрат.

Неважливо, як швидко чи недорого ви розгортаєте своє програмне забезпечення, якщо воно не функціонує.

Автоматизоване тестування програмного забезпечення має три основні переваги, такі як повторюваність, яка економить час і гроші, кумулятивне охоплення, яке допомагає знаходити помилки та знижує вартість збоїв, та важіль використання, що підвищує продуктивність ресурсів.

Майте на увазі, що цикли тестування і так будуть короткими. Замість того, щоб розраховувати на автоматизацію для пришвидшення процесу, очікуйте, що вона вчасно надасть надійний продукт. Автоматизація допомагає зменшити витрати на підтримку та переробку, а також потенційно катастрофічні витрати, розширюючи охоплення та знижуючи ймовірність збоїв.

Програми розвиваються та ускладнюються протягом процесу розробки. Якщо розробник змінить 10% коду, 100% необхідних функцій все одно потребуватимуть тестування, тому ручне тестування не може встигати за постійним тестуванням. Однак автоматизація стає тут корисною, дозволяючи збирати тестові випадки протягом життєвого циклу програми, гарантуючи, що можна протестувати як нові, так і існуючі функції. Тестери можуть обрати регресійне тестування для нових функцій, коли час обмежений. Існує більша ймовірність того, що існуючі функції матимуть проблеми, ніж нові, які були додані. Якщо є операція, яка ламається або виконує неправильні дії, тоді необхідно припинити операції.

Автоматизовані тести надають певну перевагу, але її можна втратити, якщо тести не розроблені для зручності обслуговування. Неможливо буде спиратися на попередні зусилля, коли необхідно повністю перебудувати систему або внести суттєві зміни перед використанням. Для забезпечення зручності обслуговування важливо застосувати підхід до проектування бібліотеки тестів, який підтримує зручність обслуговування протягом усього терміну служби програми.

Автоматизоване тестування в процесі розробки програмного забезпечення є важливою частиною доставки продукту клієнту, хоча це не завжди найрозумніше рішення. Використовувати автоматизоване тестування корисно

щоразу, коли відома поведінка програми [32]. Однак, якщо це не так, цінніше використовувати ручне тестування.

По-перше, існують нестабільні розробки програмного забезпечення, наприклад, програмне забезпечення, яке залежить від даних у режимі реального часу, що постійно змінюються. Якщо немає симулятора, який контролює вхідні дані, автоматизоване тестування є складним завданням, оскільки очікуваний результат невідомий.

По-друге, людина, яка пише автоматизовані тести, може не мати достатнього досвіду роботи з програмою. Якщо людина новачок у команді, вона може не мати достатніх знань про те, як має поводитися програма. З цієї причини тести можуть містити невизначені значення, і це одна з причин, чому рекомендується, щоб автоматизовані тести писав експерт.

По-третє, дозволяти тимчасовим тестувальникам писати автоматизовані тести неекономічно через початкові інвестиції в навчання людей користуватися інструментами тестування та дотримуватися дизайну бібліотеки. Ефективніше, якщо вони пишуть ручні тести, оскільки вони можуть робити ті ж помилки, що й кінцевий користувач

Зрештою, якщо часу та ресурсів для автоматизованих тестів недостатньо, розумніше обрати ручні тести. Автоматизовані тести можуть допомогти в довгостроковій перспективі, оскільки це стратегічне рішення.

Регресійне тестування – це метод тестування програмного забезпечення, який використовується для того, щоб переконатися, що зміни, внесені до програмного забезпечення, не матимуть неочікуваних наслідків, і що попередні функції працювали належним чином. Після внесення змін до програмного застосунку або програми все необхідно протестувати ще раз, щоб виявити та виправити будь-які помилки, які могли виникнути під час внесення змін.

Головною метою регресійного тестування, як форми тестування «чорної скриньки», є переконатися, що програмне забезпечення працює належним чином. Воно виявляє нові дефекти або проблеми, які могли бути внесені до програми. Регресійне тестування проводиться після виправлення помилок, оновлень

програмного забезпечення або покращення функцій.

Регресійне тестування можна виконувати вручну або за допомогою автоматизованих інструментів тестування. Його можна виконувати на різних рівнях процесу тестування програмного забезпечення, таких як модульне тестування, інтеграційне тестування та системне тестування. Впровадження регресійного тестування дозволяє розробникам і тестувальникам знаходити та вирішувати будь-які проблеми, перш ніж вони стануть серйозними та негативно вплинуть на продуктивність програмного забезпечення та взаємодію з користувачем.

Тестування методом чорної скриньки – це підхід до тестування, який оцінює поведінку та функціональність програми, не знаючи її основної функціональності, наприклад, внутрішньої структури, дизайну чи реалізації. Тестувальник зосереджується на вхідних та вихідних даних програмних програм, і це базується на критеріях та функціях програмного забезпечення, також відомих як поведінкове тестування [33]. Цей підхід зазвичай використовується для гарантії того, що програмне забезпечення відповідає визначеним вимогам та функціям у різних ситуаціях. Тестування методом чорної скриньки може бути впроваджено на всіх рівнях тестування програмного забезпечення, таких як інтеграційне, системне та приймальне тестування. Метою тестування методом чорної скриньки є виявлення дефектів, помилок або багів у продуктивності програми, які можуть вплинути на функції та враження кінцевого користувача.

Існує загальноприйнята думка, що автоматизація тестування подібна до ручного тестування шляхом визначення етапів процесу ручного тестування. Хоча автоматизація має відмінності порівняно з ручним тестуванням, такі як різні проблеми та можливості, оскільки автоматизація - це передбачуваність, а люди непередбачувані. Таким чином, використання ручного тестування для того, чого не можна очікувати, а автоматизації для перевірки того, що очікується, є необхідним. Необхідно розуміти основи автоматизації тестування.

Важливо пам'ятати, що процеси, які не є чітко визначеними, не можуть бути автоматизовані. Тест повинен мати необхідну документацію, а ручний

процес може бути погано розроблений для бібліотеки автоматизації. Навіть якщо процес тестування може бути чітко визначений, автоматизація все одно може бути складною. Встановлення фундаментальних принципів, які застосовуються та які необхідно зрозуміти, перш ніж досягти успіху. Розуміння того, що тестове програмне забезпечення є програмним забезпеченням, є важлива концепція, яка готує основу для успішного автоматизованого тестування. Розуміючи її, інші принципи випливають природним чином.

3.2 Використання сучасних інструментів

У цьому підрозділі роботи будуть пояснені інструменти, що використовуються для створення застосунку для перевірки концепції. React був обраний фреймворком для розробки веб-застосунків, оскільки це найпопулярніша бібліотека JavaScript для створення користувацьких інтерфейсів. Azure використовується для конвеєра CI/CD. Cypress використовується для написання наскрізного тестування, оскільки його легко зрозуміти, і це фреймворк для наскрізного тестування.

React - це бібліотека JavaScript з відкритим кодом, яка використовується для створення користувацьких інтерфейсів, відомих як UI. React був створений та підтримується Facebook. Це спільнота компаній та розробників. React пропонує декларативний підхід до створення користувацького інтерфейсу, в якому можна визначити, як має виглядати інтерфейс користувача, а React піклується про оновлення та рендеринг інтерфейсу [34]. React дозволяє розробникам створювати компоненти багаторазового використання та ділитися ними в кількох програмах. Цей підхід спрощує життя розробника, оскільки він створює складний користувацький інтерфейс з меншою кількістю коду та помилок.

React використовує віртуальну DOM (модель об'єктів документа), яка дозволяє ефективно змінювати інтерфейс користувача без безпосереднього маніпулювання DOM браузера. Як результат, додатки React швидші та продуктивніші, ніж додатки, створені за допомогою традиційних веб-технологій.

Cypress - це комплексний інструмент для тестування веб-застосунків та їх автоматизації. Це відкритий код на основі JavaScript, який пропонує повноцінне рішення для тестування з потужним набором інструментів для автоматизації тестування та своєчасного надання відгуків розробникам. Cypress спрощує створення тестів, які взаємодіють з фронтендом вашого застосунку, дозволяючи писати тести на JavaScript, і користується популярністю серед розробників і тестувальників. Крім того, він пропонує інструменти, які роблять написання тестів простішим і швидшим, такі як перезавантаження в режимі реального часу, автоматичне очікування та налагодження. Cypress містить потужну панель інструментів, яка надає аналітику результатів ваших тестів, що спрощує відстеження та усунення проблем.

Оскільки Cypress працює всередині браузера та має доступ до того ж DOM, що й ваша програма, це одна з її особливих характеристик. Це дозволяє Cypress імітувати поведінку користувача, таку як натискання кнопок, введення тексту та перегляд сторінок, одночасно взаємодіючи з вашою програмою в режимі реального часу [35]. Завдяки цьому створювати тести, які точно відображають поведінку користувача, набагато простіше.

Оскільки Cypress має широку документацію та велику базу користувачів, розпочати роботу з ним та знайти рішення ваших проблем дуже просто. Загалом, Cypress – це потужний та популярний інструмент для тестування веб-додатків, який допомагає переконатися, що ваш додаток працює належним чином та без помилок.

Microsoft Azure - це публічний хмарний сервіс обчислень від Microsoft. Microsoft Azure надає різноманітні інструменти та послуги. Для цієї роботи було реалізовано конвеєр безперервної інтеграції та безперервної розробки з автоматизованим тестуванням.

Azure DevOps - це платформа, яка пропонує різноманітні інструменти для контролю життєвого циклу розробки програмного забезпечення, від планування до фази кодування та тестування до фази розгортання. Доступні такі функції, як контроль версій, безперервна інтеграція, управління релізами та автоматизоване

тестування.

Частиною Azure DevOps є Azure Pipelines, який пропонує функцію CI/CD для створення, тестування та розгортання програм. Він допомагає розробникам автоматизувати процес створення та випуску, забезпечуючи інтеграцію з такими інструментами, як GitHub, Jenkins та Docker.

Загалом, ці інструменти дозволяють розробникам створювати повністю автоматизований конвеєр CI/CD, який містить автоматизоване тестування на різних етапах циклу розробки програмного забезпечення. Azure Pipelines можна використовувати для автоматичного створення та розгортання додатків. З іншого боку, конвеєри можна використовувати для запуску автоматизованих тестів та надання зворотного зв'язку щодо якості коду. Реалізація цих.

процедури допомагають знизити ймовірність виникнення помилок та помилок під час виробництва, а також підтверджують відповідність коду високим стандартам якості перед його розгортанням.

У цій роботі розроблено простий React-to-do додаток. Веб-додаток інтегровано з автоматизованим наскрізним тестуванням як частину конвеєра CI/CD у хмарних сервісах Azure.

Для цієї роботи було розроблено застосунок для створення списків справ, який дозволяє користувачам створювати та керувати ними. Застосунок було створено за допомогою фреймворку React, а код зберігається в репозиторії GitHub. Для гарантування послідовного та надійного розгортання застосунків було створено конвеєр Azure DevOps CI/CD з автоматизованим наскрізним тестуванням за допомогою Cypress.

Для побудови конвеєра CI/CD важливо сформувати процес, який автоматично збирає, тестує та розгортає програму з необхідними коригуваннями. Перший – це етап збірки, де остання версія коду буде витягнута з репозиторію та створена програма to-do за допомогою React. Цей метод включає встановлення залежностей, збір коду та формування артефакту збірки, готового до роботи. Наступний – це етап тестування, де виконуються автоматизовані тести для забезпечення правильної функціональності програм. Тести створюються за

допомогою Cypress шляхом тестування функціональних можливостей програм, таких як створення та видалення завдань to-do та перевірка виконаних елементів. Далі йде етап розгортання, де після успішного проходження тестів програма запускається в проміжне середовище за допомогою Azure App Services. Ретельне тестування може бути проведене перед розгортанням у продакшені, оскільки проміжне середовище є точною копією продакшену. Заключний етап – це випуск програм після проходження автоматичних тестів [36]. Це розгортає артефакт збірки продакшену в екземплярі Azure App Service та оновлює DNS, щоб він вказував на нове розгортання.

Загалом, ця програма надає огляд того, як автоматизувати наскрізне тестування як частину конвеєра CI/CD за допомогою програми React у хмарних сервісах Azure.

Для написання наскрізних тестів необхідно мати розроблений застосунок та проєкт, взятий з GitHub від користувача rcdexta. Проєкт було розщеплено або клоновано в особистий проєкт, де можна внести зміни. Менеджер пакетів Node та cypress необхідно встановити з терміналу за допомогою наступних команд з коду 3.1.

```
npm install
npm i cypress --save-dev
```

Лістинг 3.1 – Встановлення Node

Встановлення cypress додасть до вашого проєкту папку cypress. Тестові випадки зберігаються в папці e2e під іменем файлу spec.cy.js, як видно з рисунка 3.1.

У цій частині дипломної роботи буде налаштовано середовище Azure. Спочатку потрібно налаштувати обліковий запис Azure з підпискою, у цьому випадку використовується студентська підписка. В Azure є два вебсайти, які будуть використані для налаштування всього проєкту: dev.azure.com та portal.azure.com.

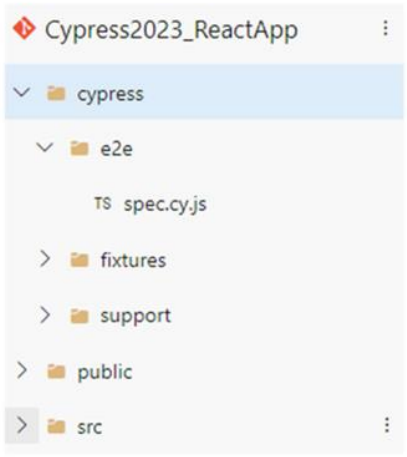


Рисунок 3.1 - Розташування файлу spec.cy.js

На сайті dev.azure.com потрібно створити проект і застосувати налаштування за замовчуванням. Після створення проекту код потрібно імпортувати до репозиторіїв Azure. Це можна зробити в бічному меню Репозиторії > Файли > Імпорт репозиторію та вказати URL-адресу вашого репозиторію GitHub (уніфіковані локатори ресурсів).

Налаштовано два конвеєри. Перший конвеєр запускає статичний веб-застосунок, а YAML буде налаштовано автоматично Azure. Другий конвеєр виконає необхідні встановлення та запустить тести Cypress, виконавши файл azure-pipelines.yml. Цей файл було налаштовано заздалегідь, і під час створення конвеєра необхідно вибрати існуючий YAML-файл Azure Pipelines. Код можна знайти в Додатку 3. Перші десять рядків коду визначають конвеєр та тригер в Azure DevOps, щоб щоразу, коли в гілці master вносяться зміни, запускався конвеєр Deploy to ASP. З дванадцятого рядка - розділ завдань, який буде виконуватися на конвеєрі. Ми налаштовуємо машину з Windows 2019, і кроки - це завдання, які потрібно виконати. Checkout перевіряє код з репозиторію. Для запуску тестів Cypress обов'язково потрібно встановити менеджер пакетів node, а потім запустити скрипт Cypress. Скрипт, який ми запускаємо, визначено у файлі package.json. У нашому випадку ми використовували тестовий скрипт, який запускає тести cypress, як видно на рисунку 6 нижче, у рядку 24. Після цього було виконано збірку, яка створила артефакт. Відтепер, щоразу, коли в гілці master буде

внесено зміни, конвеєри будуть виконуватися автоматично, і тести будуть запускатися.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test2": "react-scripts test --env=jsdom",
  "cy:report": "cypress run",
  "test": "cypress run",
  "ci:unit": "npm run test - - watchAll=false",
  "ci:cypress": "start-server-and-test start http://localhost:3000 cypress:run"
}
```

Рисунок 3.2 - Приклад файлу package.json

Загалом, результати роботи були досягнуті. Основним завданням роботи була розробка застосунку для перевірки концепції, який би давав огляд того, як реалізувати наскрізне приймальне тестування як частину конвеєра CI/CD, щоб зробити розгортання швидшим та ефективнішим.

За допомогою React було розроблено простий у виконанні застосунок. Він мав різні функції, такі як створення та видалення завдання, а також позначка його як виконаного. Після створення застосунку його було розгорнуто в сервісах Azure, які отримали код з репозиторію GitHub та перенесли його в Azure. Для початку тестування застосунку було налаштовано обліковий запис Azure. Для частини тестування наскрізні тести пишуться за допомогою Cypress. Для цілей цієї роботи Cypress має бути автоматизований за допомогою сервісів Azure, щоб забезпечити автоматизоване наскрізне тестування застосунку. Після успішного повернення тестів застосунок розгортається та стає видимим для кінцевого користувача.

На початку процесу написання роботи було вирішено використовувати Amazon Web Services, але під час процесу виникли проблеми з налаштуванням облікового запису. Тому довелося перейти на сервіси Microsoft Azure, щоб створити конвеєр CI/CD з автоматизованим наскрізним тестуванням.

Роботу можна розвинути різними способами, наприклад, порівнявши вартість того самого процесу та час розгортання у різних хмарних провайдерів.

Крім того, можна встановити з'єднання з базою даних та виконати тестування з її допомогою. Також можна реалізувати завершальну частину конвеєра CI/CD, як-от розгортання застосунку.

Загалом, конвеєр CI/CD з автоматизованим наскрізним тестуванням допомагає розробникам і компаніям швидше доставляти своє програмне забезпечення клієнту. Наскрізне тестування може бути дорогим і трудомістким, але автоматизація тестування гарантує повну функціональність програми, її роботу за планом і відповідність вимогам користувача.

Результати та аналіз» стратегії впровадження оцінює результати інтеграції автоматизації безпеки в конвеєр CI/CD. Це включає вимірювання ефективності впроваджених інструментів та процесів, аналіз впливу на загальну продуктивність конвеєра розробки та збір відгуків від команд розробників щодо інтеграції автоматизації безпеки. Цей розділ допомагає визначити, наскільки добре практики DevSecOps досягають своїх цілей щодо покращення безпеки та оптимізації робочого процесу розробки [37].

Після впровадження практик DevSecOps у конвеєр CI/CD вкрай важливо оцінити ефективність інтегрованих заходів безпеки. Це можна зробити, визначивши та вимірявши ключові показники безпеки, які надають уявлення про стан безпеки в конвеєрі. Загальні показники безпеки включають: - Коефіцієнт виявлення вразливостей. Цей показник відстежує кількість вразливостей, виявлених інструментами безпеки на кожному етапі конвеєра (наприклад, під час фіксації коду, збірки, проміжної розробки тощо). Порівнюючи цей показник з часом, організації можуть оцінити ефективність автоматизованих сканувань безпеки у виявленні вразливостей на ранніх етапах процесу розробки. Коефіцієнти хибнопозитивних та хибнонегативних результатів. Ці показники вимірюють точність інструментів безпеки. Хибнопозитивний результат виникає, коли інструмент позначає непроблемну вразливість як вразливість, тоді як хибнонегативний результат виникає, коли інструмент пропускає фактичну вразливість. Відстеження цих показників допомагає гарантувати, що інструменти надають надійні результати. Час виправлення. Цей показник відстежує, скільки

часу потрібно розробникам для усунення вразливостей після їх виявлення. Коротший час виправлення вказує на те, що інструменти безпеки та автоматизовані процеси ефективно допомагають розробникам швидко виправляти проблеми. Дефекти безпеки, що потрапляють у виробниче середовище [38]. Цей показник вимірює, скільки вразливостей проходять через конвеєр CI/CD та присутні у виробничому середовищі. Нижчий показник свідчить про те, що вжиті заходи безпеки ефективні для виявлення проблем до того, як вони потраплять у виробниче середовище. Регулярна оцінка цих показників забезпечує уявлення про те, як автоматизація безпеки сприяє підвищенню рівня безпеки організації, а також допомагає визначити області, які потребують подальшого вдосконалення.

Хоча автоматизація безпеки покращує безпеку застосунку, важливо оцінити вплив інтеграції інструментів безпеки в конвеєр CI/CD на продуктивність. Інструменти безпеки іноді можуть створювати накладні витрати, що може вплинути на швидкість та ефективність конвеєра. Аналіз цих впливів на продуктивність включає - Час збірки та затримку Впровадження інструментів безпеки, таких як SAST , DAST та SCA, може збільшити час, необхідний для процесів збірки та розгортання. Зокрема, інструменти SAST можуть додати значну затримку під час фаз фіксації або збірки, особливо у великих кодових базах. Тому організаціям необхідно оцінити додатковий час, який інструменти безпеки додають до конвеєра, та оцінити, чи є цей час прийнятним у контексті їхніх цілей розробки. Споживання ресурсів Інструменти безпеки, особливо під час фаз DAST або тестування на проникнення , можуть бути ресурсоємними. Вони можуть вимагати значних обчислювальних ресурсів, що може вплинути на продуктивність інфраструктури CI/CD, особливо при масштабуванні на кілька проектів. Використання контейнерів та розподілених методів сканування може допомогти зменшити ці проблеми, але організації повинні уважно стежити за використанням ресурсів. Компроміси між швидкістю та безпекою Часто існує компроміс між швидкістю розробки та рівнем безпеки, який можна застосувати. Інтеграція комплексних інструментів безпеки на кожному етапі розробки може

уповільнити процес, особливо якщо організація обирає вичерпне сканування. Балансування швидкості та безпеки передбачає визначення пріоритетів сканування в яких середовищах та забезпечення постійного виявлення критичних вразливостей, навіть якщо деякі некритичні тести пропускаються або відкладаються. Розуміння та пом'якшення впливу на продуктивність, одночасно підтримуючи надійний рівень безпеки, є важливим для організацій, щоб знайти правильний баланс між безпекою та ефективністю розробки. Це може включати оптимізацію розробки шляхом впровадження паралельних сканувань безпеки, обмеження обсягу сканувань критичними областями або використання хмарних ресурсів для масштабування за потреби.

Зворотній зв'язок від команд розробників має вирішальне значення для розуміння того, наскільки добре працюють інтегровані практики DevSecOps з точки зору тих, хто використовує конвеєр щодня. Успіх DevSecOps значною мірою залежить від його прийняття та впровадження розробниками, тому збір їхніх знань надає цінну інформацію для точного налаштування впровадження. Загальні області зворотного зв'язку включають:

Досвід користувача інструментів безпеки**.** Розробники можуть надавати відгуки про те, наскільки легко чи складно працювати з інструментами безпеки, інтегрованими в конвеєр. Це включає в себе те, наскільки інтуїтивно зрозумілі інструменти, наскільки легко виправляти проблеми, позначені інструментами, та наскільки добре інструменти інтегруються з існуючим середовищем розробки [39]. Якщо розробники виявлять, що інструменти безпеки створюють труднощі в їхньому робочому процесі, вони можуть бути менш схильні ефективно їх впроваджувати. Вплив на продуктивність розробників. Розробники можуть висловлювати занепокоєння, якщо додаткові процеси безпеки уповільнюють їхній робочий процес або створюють вузькі місця в конвеєрі. Наприклад, якщо тести безпеки часто блокують збірки або якщо помилкових спрацьовувань забагато, розробники можуть відчувати розчарування та шукати способи обійти перевірки безпеки. Постійна взаємодія з командами розробників та коригування на основі їхніх відгуків необхідні для безперебійної роботи автоматизації безпеки.

Співпраця між командами безпеки та розробки Однією з цілей DevSecOps є сприяння кращій співпраці між командами безпеки та розробки [40]. Зворотній зв'язок від розробників може показати, чи є внесок команди безпеки корисним та своєчасним, а також чи відчувають розробники підтримку у виправленні вразливостей. Позитивний відгук свідчитиме про те, що інтеграція безпеки працює добре, тоді як негативний відгук може свідчити про те, що практики безпеки необхідно скоригувати, щоб вони тісніше відповідали робочому процесу розробки. Навчання та обмін знаннями**.** Розробники можуть надавати відгуки щодо необхідності додаткового навчання або кращої документації, щоб зрозуміти, як інтерпретувати та діяти на основі висновків щодо безпеки. Надання чітких інструкцій та постійне навчання може підвищити здатність команди розробників ефективно працювати з інструментами автоматизації безпеки. Врахування відгуків команд розробників має вирішальне значення для підвищення ефективності та впровадження практик автоматизації безпеки. Регулярні цикли зворотного зв'язку дозволяють організаціям постійно вдосконалювати процес DevSecOps, роблячи його безпечним та зручним для розробників.

3.3 Оцінювання ефективності автоматизованого тестування та аналіз результатів

Оцінювання включало проведення 20 тестових запусків для кожного інструменту для кожного типу тесту продуктивності. Це забезпечило статистично достовірний збір даних та мінімізувало вплив викидів. Використані профілі навантаження проілюстровано на рисунку 2. Тести були розділені на три типи: репрезентативні (стійке навантаження), максимальні (масштабоване навантаження) та пікові (раптові стрибки). Ключові показники ефективності (KPI), що вимірювалися, включали використання процесора та оперативної пам'яті, середній час відгуку, пропускну здатність (запити за секунду, RPS) та коефіцієнт помилок.

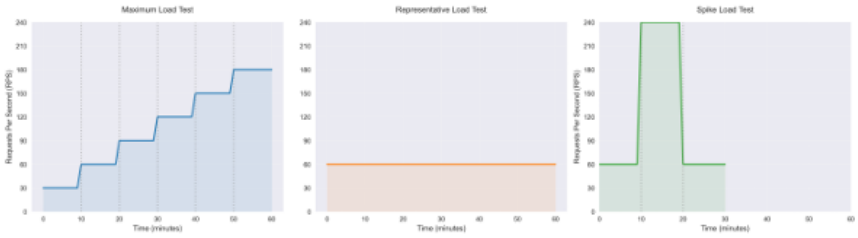
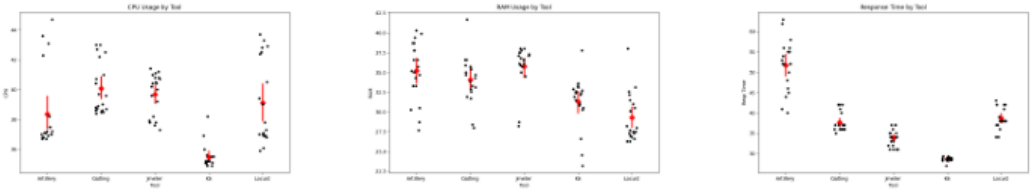


Рисунок 3.3 - Візуалізація профілів навантажувальних тестів

При постійному навантаженні 60 RPS, Grafana k6 виявився найефективнішим інструментом з використанням процесора в середньому 35,5% та найнижчим середнім часом відгуку 28 мс, як показано в Таблиці 3.1. Locust продемонстрував найнижче споживання оперативної пам'яті (29,2%), тоді як Artillery повідомив про найвищий час відгуку (63 мс), що більш ніж удвічі перевищує час відгуку k6. Усі інструменти зафіксували нульовий рівень помилок при такому навантаженні.



(a) CPU Utilization (b) RAM Utilization (c) Response time

Рисунок 3.4 - Типове навантаження: розподіл значень між різними інструментами тестування

Таблиця 3.1

Типові показники навантажувального тесту (60 RPS)

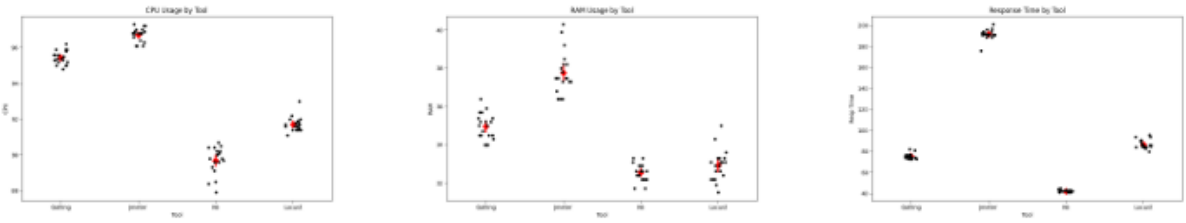
Метрика	**JMeter**	**k6**	**Гатлінг**	**Сарана**	**Артилерія**
Середнє використання процесора (%)	39,7	35,5	40.1	39.1	38,4
Середнє використання оперативної пам'яті (%)	35,8	31.3	34.1	29.2	35.2
Середній час відгуку (мс)	34	28	37	39	63
Пропускна здатність (RPS)	60	60	60	60	60
Коефіцієнт помилок (%)	0	0	0	0	0

Під час стрес-тестів із поступово зростаючим навантаженням (від 30 до 180 RPS), Artillery неодноразово спричиняв збої системи та був виключений з подальшого аналізу. Серед решти інструментів k6 стабільно демонстрував найкращу продуктивність: пікове використання процесора 89,7% та середній час відгуку лише 42 мс (табл. 3.2, рис. 3.5). JMeter споживав найбільше ресурсів процесора (96,7%) та зафіксував найвищу затримку (192 мс), що вказує на обмеження масштабованості. Погіршення стану системи спостерігалося не у вигляді частоти помилок, а у збільшенні часу відгуку.

Таблиця 3.2

Показники тесту максимального навантаження (180 обертів за секунду)

Метрика	**JMeter**	**k6**	**Гатлінг**	**Сарана**
Максимальне використання процесора (%)	96,7	89,7	95,4	91,7
Максимальне використання оперативної пам'яті (%)	37,8	32,5	32,9	35,0
Середній час відгуку (мс)	192	42	76	87
Максимальна пропускна здатність (RPS)	180	180	180	180
Коефіцієнт помилок (%)	0	0	0	0



(a) CPU Utilization (b) RAM Utilization (c) Response time

Рисунок 3.5 - Максимальне навантаження: розподіл значень за різними випробувальними приладами

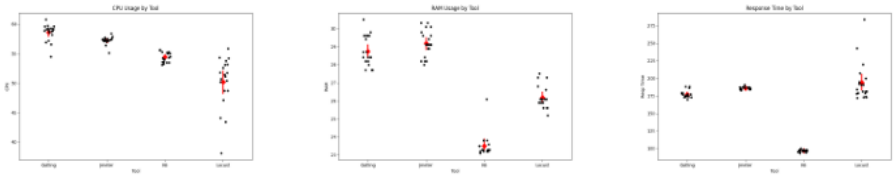
Тестування пікових частот імітувало різке збільшення трафіку від 60 до 240 RPS. Знову ж таки, **k6** лідирував з найнижчим використанням ресурсів (ЦП: 54,4%, ОЗП: 23,5%) та найшвидшим середнім часом відгуку (98 мс). **Gatling** та **Locust** показали порівнянні результати у використанні ресурсів, але продемонстрували вищий час відгуку (177 мс та 194 мс відповідно). **JMeter**

демонстрував нестабільну поведінку під час відновлення навантаження, намагаючись підтримувати стабільний час відгуку (таблиця 3.3, рис. 3.6)

Таблиця 3.3

Показники випробування на пікове навантаження (60-240 обертів за секунду)

Метрика	JMeter	к6	Гатлінг	Сарана
Середнє використання процесора (%)	57.2	54,4	58,6	50.2
Середнє використання оперативної пам'яті (%)	29.2	23,5	28.8	26.2
Середній час відгуку (мс)	186	98	177	194
Максимальна пропускна здатність (RPS)	240	240	240	240
Коефіцієнт помилок (%)	0	0	0	0



(a) CPU Utilization (b) RAM Utilization (c) Response time

Рисунок 3.6 - Розподіл значень між різними інструментами тестування

Статистична валідація

Щоб підтвердити спостережувані відмінності, ми застосували тест Краскела-Уолліса, який дав дуже значущі результати за всіма показниками ефективності ($p < 6{,}3 \times 10^{-9}$). Потім ми провели попарні порівняння за допомогою U-тесту Манна-Вітні, скоригувавши значущість за допомогою корекції Бонферроні

$$\alpha' = \frac{0.05}{\binom{5}{2}} = \frac{0.05}{10} = 0.005 \qquad (3.1)$$

З 66 парних порівнянь 55 виявилися статистично значущими (див. таблиці 3.4–3.6).

Ці результати переконливо підтверджують розбіжності в ефективності цих інструментів.

Таблиця 3.4

P-значення для репрезентативного навантажувального тестування

(а) Використання CPU

	A	**G**	**J**	**K**	**L**
A	—	10^{-3}	0.002	10^{-6}	0.49
G	10^{-3}	—	0.67	10^{-7}	0.11
J	0.002	0.67	—	10^{-7}	0.17
K	10^{-6}	10^{-7}	10^{-7}	—	10^{-6}
L	0.49	0.11	0.17	10^{-6}	—

(б) Використання RAM

	A	**G**	**J**	**K**	**L**
A	—	0.17	0.67	0.001	10^{-5}
G	0.17	—	0.003	0.002	10^{-5}
J	0.67	0.003	—	10^{-4}	10^{-5}
K	0.001	0.002	10^{-4}	—	0.02
L	10^{-5}	10^{-5}	10^{-5}	0.02	—

(в) Час відгуку (Response Time)

	A	**G**	**J**	**K**	**L**
A	—	10^{-7}	10^{-7}	10^{-7}	10^{-7}
G	10^{-7}	—	10^{-5}	10^{-7}	0.04
J	10^{-7}	10^{-5}	—	10^{-7}	10^{-6}
K	10^{-7}	10^{-7}	10^{-7}	—	10^{-7}
L	10^{-7}	0.04	10^{-6}	10^{-7}	—

На узагальнюваність наших висновків можуть вплинути кілька обмежень:

— Оцінювання базувалося на одному застосунку Java через необхідність вкладень часу. Знадобилося понад 250 годин для виконання 20 тестових запусків для кожної комбінації трьох типів тестів та п'яти інструментів. Результати можуть відрізнятися для застосунків на різних мовах програмування або архітектурах.

— Тестування проводилося в середовищі Windows через WSL, що відрізняється від типових робочих систем на базі Linux.

Профілі навантаження були синтетичними та можуть неточно відображати реальну поведінку користувачів або моделі паралельного використання.

Таблиця 3.5

P-значення для тесту максимального навантаження

(а) Використання CPU

	G	**J**	**K**	**L**
G	—	10^{-7}	10^{-7}	10^{-7}
J	10^{-7}	—	10^{-7}	10^{-7}
K	10^{-7}	10^{-7}	—	10^{-7}
L	10^{-7}	10^{-7}	10^{-7}	—
	G	**J**	**K**	**L**

(б) Використання RAM

	G	**J**	**K**	**L**
G	—	10^{-7}	10^{-7}	10^{-7}
J	10^{-7}	—	10^{-7}	10^{-7}
K	10^{-7}	10^{-7}	—	0.0905
L	10^{-7}	10^{-7}	0.0905	—
	G	**J**	**K**	**L**

(в) Час відгуку (Response Time)

	G	**J**	**K**	**L**
G	—	10^{-7}	10^{-7}	10^{-7}
J	10^{-7}	—	10^{-7}	10^{-7}
K	10^{-7}	10^{-7}	—	10^{-7}
L	10^{-7}	10^{-7}	10^{-7}	—
	G	**J**	**K**	**L**

Наші висновки підкреслюють вплив вибору інструменту на результати тестування продуктивності в конвеєрах CI/CD:

— Grafana K6 постійно перевершує інші, що робить його ідеальним для трубопроводів, чутливих до ресурсів.

— JMeter , хоча й зручний у використанні завдяки своєму графічному інтерфейсу, продемонстрував найбільші проблеми з накладними витратами та масштабованістю.

— Locust знаходить баланс між зручністю використання (скрипти на основі Python) та помірною продуктивністю.

— Gatling показав результати нижче середніх у використанні ресурсів, але має переваги у вигляді функцій звітності після тестування та сценаріїв Java.

— артилерії під час масштабування роблять її непридатною для застосування в стрес-тестуванні.

Ця оцінка вирішує критичні прогалини в літературі, надаючи емпіричні - порівняння різних типів тестів та інструментів. Подальша робота повинна дослідити ширші сценарії тестування, області застосування та додаткові інструменти для створення більш комплексної системи тестування продуктивності.

Це дослідження розглядало проблему вибору ресурсоефективного інструменту тестування продуктивності для автоматизованої інтеграції в конвеєри CI/CD DevOps. Наша мета.

полягало у зменшенні витрат на ресурси тестування продуктивності шляхом визначення інструментів, які пропонують хороший баланс між використанням системних ресурсів та ефективністю тестування. Завдяки цьому можна підвищити ефективність конвеєра, і більше конвеєрів можуть працювати в рамках фіксованого бюджету ресурсів.

Таблиця 3.6

P-значення для тесту пікового навантаження

(а) Використання CPU

	G	J	K	L
G	—	0.0004	10^{-6}	10^{-7}
J	0.0004	—	10^{-6}	10^{-7}
K	10^{-6}	10^{-6}	—	10^{-5}
L	10^{-7}	10^{-7}	10^{-5}	—

(б) Використання RAM

	G	**J**	**K**	**L**
G	—	0.1055	10^{-7}	10^{-7}
J	0.1055	—	10^{-7}	10^{-7}
K	10^{-7}	10^{-7}	—	10^{-6}
L	10^{-7}	10^{-7}	10^{-6}	—

(в) Час відгуку (Response Time)

	G	**J**	**K**	**L**
G	—	10^{-5}	10^{-7}	0.0048
J	10^{-5}	—	10^{-7}	0.6445
K	10^{-7}	10^{-7}	—	10^{-7}
L	0.0048	0.6445	10^{-7}	—

Ми вибрали п'ять найпопулярніших інструментів з відкритим кодом – Apache Jmeter , Gatling , Grafana K6 , Locust та Artillery – та оцінили їх у конвеєрах Jenkins за допомогою докеризованого застосунку Spring Boot. Рішення для моніторингу в режимі реального часу було розроблено за допомогою Grafana , Prometheus та cAdvisor . Експерименти проводилися за трьома методологіями тестування: репрезентативне навантаження, максимальне навантаження та пікове навантаження. Щоб імітувати реалістичну поведінку застосунку, ми створили штучні профілі навантаження за допомогою скриптів Python , забезпечивши автоматизацію та повторюваність у конвеєрах.

Кожна методологія тестування була розроблена для перевірки різних характеристик продуктивності. Щоб забезпечити узгодженість навантаження між різними інструментами, виконувалися налагоджувальні запуску, коригуючи внутрішній розподіл навантаження, доки інтенсивність тестування не стала майже однаковою у всіх скриптах. Репрезентативні навантажувальні тести імітували типове використання, тести максимального навантаження збільшували навантаження на систему з часом, а пікові тести імітували раптові сплески попиту. Для кожної комбінації інструменту/типу тесту ми провели 20 тестів та зібрали такі показники, як час відгуку, використання процесора та оперативної пам'яті. Хоча Artillery конкурентоспроможно працював при стабільних навантаженнях, він спричиняв повторні збої програми за різних умов навантаження та був виключений з подальшого аналізу в сценаріях максимального та пікового навантаження.

Для оцінки статистичної значущості спостережуваних відмінностей у продуктивності між протестованими інструментами ми застосували непараметричні статистичні методи через ненормальний розподіл зібраних даних.

Для оцінки статистично значущості відмінностей у показниках продуктивності між усіма інструментами було використано тест Краскела-Уолліса. Паралельно ми використовували U-тест Манна-Вітні для попарних порівнянь, застосовуючи корекцію Бонферроні для контролю помилок I типу, що виникають в результаті множинних порівнянь.

Нульова гіпотеза в обох тестах припускала, що немає суттєвих відмінностей у розподілі контрольованих показників між інструментами. Для тесту Краскела-Уолліса статистична значущість визначалася пороговим значенням p-значення 0,05. Усі обчислені p-значення були значно нижчими за цей поріг, а найвище становило $6,3 \times 10^{-9}$, що свідчить про вагомі докази проти нульової гіпотези. Для U-тесту Манна-Вітні рівень значущості, скоригований за Бонферроні, був встановлений на рівні 0,005. З 66 можливих пар інструментів 55 дали p-значення нижче цього порогу, як детально описано в таблицях 3.4-3.6.

Ці статистичні результати підтверджують, що спостережувані коливання продуктивності не є випадковими. Таким чином, ми робимо висновок, що відмінності у використанні ресурсів, часі відгуку та пропускній здатності між протестованими інструментами є статистично значущими, а набір даних, зібраний під час оцінювання, є надійним та достовірним для порівняльних висновків.

Серед інструментів Grafana K6 послідовно демонстрував найефективнішу продуктивність як у використанні системи, так і в часі відгуку, що робить його найкращим варіантом для високоефективних середовищ CI/CD. Locust також добре показав себе і може бути привабливим для команд з досвідом роботи з Python. Jmeter та Gatling забезпечили цінні функції, такі як графічні інтерфейси та вбудована звітність, що може бути корисним для команд, які надають пріоритет простоті використання або підтримці налагодження. Artillery , хоча й зручний у використанні, виявився непридатним для тестів навантаження з високою дисперсією через проблеми зі стабільністю.

Це дослідження також підкреслює практичну можливість інтеграції тестування продуктивності в конвеєри CI/CD. Наша стратегія автоматизації - з динамічною генерацією профілів, моніторингом у режимі реального часу та

статистичним аналізом - демонструє, що навантажувальне тестування може бути повністю вбудоване в процес розробки, керований конвеєром, з обмеженими накладними витратами. Експерименти також виявили кілька важливих уроків. По-перше, вибір інструменту сильно впливає як на надійність тестування, так і на вартість інфраструктури. По-друге, автоматизовані конвеєри є більш масштабованими та менш схильними до помилок, ніж ручне тестування. По-третє, для підтвердження достовірності відмінностей у продуктивності необхідні статистичні методи. Результати цієї роботи пропонують кілька внесків. Ми надаємо порівняльний аналіз поведінки п'яти сучасних інструментів з відкритим кодом для навантажувального тестування за різних сценаріїв навантаження. Ми представляємо багаторазову конфігурацію конвеєра, яка включає автоматичну генерацію профілів, моніторинг системи та статистичну перевірку. Її можна знайти в публічному репозиторії GitHub разом з метриками з проаналізованих запусків та скриптами для розрахунку використаних статистичних тестів. Ми також виділяємо практичні компроміси, такі як складність скриптів, стабільність під навантаженням та придатність для автоматизації

Оскільки ландшафт розробки програмного забезпечення та кібербезпеки продовжує розвиватися, майбутні вдосконалення DevSecOps та автоматизованого тестування безпеки мають вирішальне значення для того, щоб не відставати від нових викликів та підтримувати надійні методи безпеки. Наступний етап DevSecOps, ймовірно, включатиме передові технології, підвищить масштабованість та забезпечить адаптивність до потреб організацій, що постійно змінюються. Ось ключові сфери, де ці досягнення можуть вплинути.

Штучний інтелект (ШІ) та машинне навчання (МН) готові революціонізувати сферу автоматизованого тестування безпеки. Наразі інструменти безпеки покладаються на попередньо визначені правила та шаблони для виявлення вразливостей. Однак інтеграція ШІ та МН може дозволити інструментам безпеки навчатися на нових моделях атак , прогнозувати потенційні загрози та автоматично адаптуватися до постійно мінливого ландшафту кіберризиків. ШІ може покращити статичне тестування безпеки додатків (SAST)

та динамічне тестування безпеки додатків (DAST) , дозволяючи інструментам виявляти складні вразливості, які традиційні методи можуть пропустити, наприклад, ті, що покладаються на неочевидні шаблони експлуатації. Крім того, моделі машинного навчання можна навчати на великих наборах даних для виявлення аномалій, зменшуючи кількість хибнопозитивних результатів та підвищуючи точність сканування безпеки. Крім того, автоматизоване виправлення на основі ШІ може швидше виявляти та пропонувати виправлення вразливостей, значно скорочуючи час, який розробники витрачають на ручне вирішення проблем безпеки. З часом моделі МН можуть розвиватися, щоб виявляти нові класи вразливостей на основі потоків інформації про загрози, роблячи тестування безпеки більш проактивним та прогнозованим. У міру розвитку цих технологій вони матимуть вирішальне значення для покращення загального стану безпеки, зберігаючи при цьому швидкість та ефективність робочих процесів розробки.

Інтеграція нових технологій, таких як хмарно-орієнтовані архітектури , безсерверні обчислення , блокчейн та квантові обчислення, створить нові виклики та можливості для DevSecOps у майбутньому. Хмарно-орієнтована безпека : Оскільки організації продовжують впроваджувати хмарно-орієнтовані архітектури, в основі яких лежать контейнери та мікросервіси, акцент на безпеці зміститься на управління безпекою контейнерів та оркестрації. Інструменти, що підтримують контейнеризовані середовища, такі як інструменти безпеки Kubernetes та безсерверні рішення безпеки , повинні будуть розвиватися, щоб йти в ногу зі складністю розподілених хмарно-орієнтованих додатків. DevSecOps повинні будуть використовувати інструменти безпеки контейнерів на основі штучного інтелекту та рішення для захисту середовища виконання , які виявляють загрози та реагують на них у режимі реального часу, забезпечуючи безперервний моніторинг у динамічних хмарних середовищах. Технології блокчейну та розподіленого реєстру : Зі зростанням впровадження блокчейну інструменти DevSecOps можуть потребувати інтеграції рішень безпеки на основі блокчейну, таких як незмінні журнали для журналів аудиту або сканування вразливостей

смарт-контрактів. Ці технології вимагатимуть нових типів оцінок безпеки для забезпечення цілісності та безпеки децентралізованих додатків (DApps) та інфраструктур блокчейну. Квантові обчислення : З огляду на появу квантових обчислень, необхідно буде вирішити проблеми безпеки, пов'язані з квантово-стійкими алгоритмами шифрування. DevSecOps повинні будуть впровадити інструменти, здатні оцінювати стандарти постквантової криптографії та забезпечувати квантову стійкість криптографічних ключів, що використовуються в конвеєрі CI/CD, щоб підготуватися до можливої загрози квантових обчислень. Майбутнє DevSecOps вимагатиме безперешкодної інтеграції з цими новими технологіями, забезпечуючи безпеку в усіх сучасних парадигмах розробки програмного забезпечення.

Оскільки організації продовжують зростати та впроваджувати мікросервісні архітектури , багатохмарні середовища та розподілені системи , інструменти DevSecOps повинні розвиватися, щоб враховувати масштабованість та складність великомасштабних середовищ. Масштабування DevSecOps : Великим організаціям з тисячами програм або мікросервісів потрібні інструменти для тестування безпеки, які можуть працювати масштабовано, без надмірних накладних витрат або затримок у конвеєрі. Майбутні рішення DevSecOps повинні зосереджуватися на розподіленому тестуванні безпеки , яке може ефективно сканувати та моніторити кілька сервісів, репозиторіїв та середовищ одночасно. Такі технології, як платформи оркестрації контейнерів (наприклад, Kubernetes), стануть вирішальними для управління безпекою сотень або тисяч контейнерів, зберігаючи при цьому послідовний рівень безпеки. Адаптація до складних архітектур: У великомасштабних середовищах DevSecOps повинні бути адаптивними для задоволення різноманітних потреб безпеки в кількох середовищах. Це може включати багатохмарні архітектури, де політики безпеки повинні застосовуватися однаково в публічних, приватних та гібридних хмарах. Інструменти безпеки майбутнього повинні безперешкодно інтегруватися з кількома постачальниками хмарних послуг та забезпечувати дотримання галузевих норм у різних юрисдикціях. Автоматизоване масштабування

інструментів безпеки: Зі зростанням середовищ автоматизоване масштабування інструментів безпеки стане важливим. Замість виконання сканувань безпеки за фіксованим графіком або коли відбувається певна подія, безперервне сканування безпеки потребуватиме коригування на основі використання ресурсів у режимі реального часу. Це включає динамічне масштабування інструментів безпеки на основі кількості розгортань , розміру кодових баз або складності компонентів інфраструктури. Зворотній зв'язок щодо безпеки в режимі реального часу в масштабі: У великомасштабному середовищі важливо надавати зворотний зв'язок щодо вразливостей у режимі реального часу, не знижуючи продуктивності розробників. Сканування вразливостей у режимі реального часу має масштабуватися на великі команди та великі кодові бази, водночас надаючи розробникам дієвий, пріоритетний зворотний зв'язок. Такі рішення, як хмарні платформи тестування безпеки та децентралізовані команди безпеки, будуть життєво важливими для управління зростаючою складністю. Зі збільшенням розміру та масштабу програмних середовищ інструменти DevSecOps повинні стати більш гнучкими та масштабованими, здатними адаптуватися до зростаючого набору послуг, програм та інфраструктур, зберігаючи при цьому стабільний та високий рівень безпеки.

Це дослідження призначене для тестувальників продуктивності, які оцінюють варіанти інструментів, а також для DevSecOps-інженерів, які прагнуть покращити автоматизацію забезпечення якості в конвеєрах розробки програмного забезпечення. Інтегруючи тестування продуктивності безпосередньо в робочі процеси CI/CD, команди можуть підвищити надійність релізу без шкоди для швидкості. Оскільки ми рухаємося до ширшої автоматизації процесів DevSecOps [17], ця робота демонструє, що тестування продуктивності може - і повинно - бути частиною цих зусиль з автоматизації.

3.4 Висновки по розділу

У третьому розділі було розглянуто практичну реалізацію, інтеграцію та оцінювання ефективності рішень автоматизованого тестування в CI/CD-конвеєрах. Проаналізовано процес впровадження автоматизованого тестування у середовище безперервної інтеграції та доставки, а також особливості використання сучасних інструментів для автоматизації перевірки програмного забезпечення. Проведено оцінювання ефективності автоматизованого тестування та аналіз отриманих результатів, що дозволило визначити переваги застосування CI/CD-підходів для підвищення якості, стабільності та швидкості розробки програмного забезпечення. Отримані результати підтверджують доцільність використання автоматизованого тестування як важливої складової сучасних процесів розробки ПЗ.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено та розроблено підхід до використання інструментів автоматизованого тестування в CI/CD-конвеєрах, що є комплексним процесом, спрямованим на підвищення якості програмного забезпечення, скорочення часу розробки та забезпечення безперервної доставки надійних продуктів. У роботі проаналізовано сучасні підходи до автоматизації тестування, досліджено переваги та обмеження існуючих інструментів, а також реалізовано практичні рішення для інтеграції тестування в CI/CD-процеси.

На початковому етапі було проведено аналіз предметної області, визначено ключові принципи безперервної інтеграції та безперервної доставки, а також сформовано функціональні та нефункціональні вимоги до системи автоматизованого тестування. Особливу увагу приділено ролі автоматизації у підвищенні надійності програмних систем, зменшенні людського фактору та забезпеченні швидкого зворотного зв’язку для розробників.

У процесі моделювання було розглянуто архітектуру CI/CD-конвеєра, що включає етапи збірки, тестування, аналізу якості коду та розгортання. Визначено ключові компоненти системи, такі як інструменти автоматизованого тестування, засоби перевірки безпеки та механізми інтеграції з системами контролю версій. Було також досліджено підходи DevSecOps, що дозволяють інтегрувати перевірки безпеки на всіх етапах життєвого циклу програмного забезпечення.

Практична частина роботи охопила реалізацію автоматизованого тестування з використанням сучасних інструментів, зокрема фреймворків для тестування інтерфейсу та API, а також інтеграцію цих інструментів у CI/CD-конвеєр. Було забезпечено автоматичний запуск тестів при кожній зміні коду, що дозволило оперативно виявляти дефекти та підвищити стабільність програмного продукту. Особливу увагу приділено використанню таких інструментів, як Cypress для тестування веб-інтерфейсів та хмарних платформ для організації процесів безперервної інтеграції.

У ході тестування проведено оцінку ефективності впровадженого рішення,

що включала аналіз швидкості виконання тестів, покриття коду та здатності системи виявляти помилки на ранніх етапах розробки. Результати показали, що використання автоматизованого тестування в CI/CD-конвеєрах значно зменшує кількість дефектів у продуктивному середовищі та підвищує загальну якість програмного забезпечення.

Загалом, розроблене рішення відповідає поставленим цілям і демонструє ефективність використання автоматизованих інструментів тестування в сучасних процесах розробки. Перевагами підходу є підвищення швидкості розробки, покращення якості коду, зменшення витрат на тестування та забезпечення безперервної доставки програмного забезпечення. Водночас існують певні обмеження, зокрема складність налаштування CI/CD-конвеєрів, необхідність підтримки тестів та залежність від обраних інструментів.

У перспективі подальші дослідження можуть бути спрямовані на розширення можливостей автоматизованого тестування, зокрема шляхом використання штучного інтелекту для генерації тестів, підвищення рівня автоматизації процесів DevSecOps та інтеграції з інструментами моніторингу й аналітики, що дозволить ще більше підвищити ефективність і надійність програмних систем.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Azure DevOps Documentation. (2025). https://learn.microsoft.com/en-us/azure/devops/
2. GitHub Actions Documentation. (2025). https://docs.github.com/en/actions
3. GitLab CI/CD Documentation. (2025). https://docs.gitlab.com/ee/ci/
4. Jenkins User Documentation. (2025). https://www.jenkins.io/doc/
5. Atlassian Bamboo CI/CD Guide. (2025). https://www.atlassian.com/software/bamboo
6. Cypress End-to-End Testing. (2025). https://docs.cypress.io/
7. Selenium WebDriver Documentation. (2025). https://www.selenium.dev/documentation/
8. Playwright Testing Framework. (2025). https://playwright.dev/docs/intro
9. Postman API Testing Documentation. (2025). https://learning.postman.com/docs/
10. JUnit Testing Framework Guide. (2024). https://junit.org/junit5/docs/current/user-guide/
11. NUnit Documentation. (2024). https://docs.nunit.org/
12. xUnit.net Testing Framework. (2024). https://xunit.net/
13. Mocha JavaScript Test Framework. (2024). https://mochajs.org/
14. Chai Assertion Library. (2024). https://www.chaijs.com/
15. Jest Testing Framework. (2025). https://jestjs.io/docs/getting-started
16. OWASP Testing Guide. (2024). https://owasp.org/www-project-web-security-testing-guide/
17. OWASP Top 10 Security Risks. (2024). https://owasp.org/www-project-top-ten/
18. DevSecOps Fundamentals. (2024). https://www.redhat.com/en/topics/devops/what-is-devsecops

19. Microsoft DevSecOps Guide. (2025). https://learn.microsoft.com/en-us/devops/devsecops/

20. Google DevOps Research and Assessment (DORA). (2024). https://cloud.google.com/devops

21. Docker Documentation. (2025). https://docs.docker.com/

22. Kubernetes CI/CD Practices. (2025). https://kubernetes.io/docs/concepts/

23. Helm Package Manager Docs. (2024). https://helm.sh/docs/

24. Terraform CI/CD Integration. (2024). https://developer.hashicorp.com/terraform/docs

25. Ansible Automation Documentation. (2024). https://docs.ansible.com/

26. Martin Fowler. Continuous Integration Principles. (2023). https://martinfowler.com/articles/continuousIntegration.html

27. Martin Fowler. Continuous Delivery. (2023). https://martinfowler.com/bliki/ContinuousDelivery.html

28. Accelerate: State of DevOps Report. (2024). https://dora.dev/

29. Google Cloud CI/CD Pipelines. (2025). https://cloud.google.com/docs/devops

30. AWS CodePipeline Documentation. (2025). https://docs.aws.amazon.com/codepipeline/

31. AWS CodeBuild Testing. (2025). https://docs.aws.amazon.com/codebuild/

32. Azure Pipelines Testing Strategy. (2025). https://learn.microsoft.com/en-us/azure/devops/pipelines/

33. IBM DevOps Automation Practices. (2024). https://www.ibm.com/cloud/learn/devops

34. Red Hat OpenShift CI/CD. (2024). https://www.redhat.com/en/technologies/cloud-computing/openshift

35. SonarQube Code Quality & Testing. (2025). https://www.sonarsource.com/products/sonarqube/

36. Checkmarx Application Security Testing. (2024). https://checkmarx.com/

37. Snyk DevSecOps Security Testing. (2025). https://snyk.io/

38. TestNG Framework Documentation. (2024). https://testng.org/doc/
39. Apache Maven CI Integration. (2024). https://maven.apache.org/guides/
40. Gradle Build Automation & Testing. (2025). https://docs.gradle.org/

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Інструменти автоматизованого тестування в CI/CD-конвеєрах"

Обсяг пояснювальної записки: 75 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____