Міністерство освіти і науки України

Луцький національний технічний університет
(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)

Кафедра комп'ютерної інженерії та кібербезпеки
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ЗАСОБИ ТЕСТУВАННЯ ІТ-ПРОДУКТУ ПОБУДОВАНОГО НА МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ
IT PRODUCT TESTING TOOLS ON MICROSERVICE ARCHITECTURE

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІз-41
Каламарчук Віктор Сергійович

(підпис)

Керівник:
д.е.н., професор
Ляшенко Оксана Миколаївна

(підпис)

Кваліфікаційну роботу
допущено до захисту
« _____ » червня 2023 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2023 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та кібербезпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ проф. Н.Чернящук
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Каламарчуку Віктору Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Засоби тестування IT-продукту побудованого на мікросервісній архітектурі*

Керівник роботи *д.е.н., професор Ляшенко Оксана Миколаївна*

затверджені наказом закладу вищої освіти від «28» грудня 2022 року № 982/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 01.06.2023р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області та різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Теоретична частина
Аналітична частина
Рекомендаційна частина
Висновки

5. Перелік графічного (ілюстративного) матеріалу:
Практична цінність розроблена система може використовуватись при вирішенні різних завдань відеоаналітики, і в першу чергу має безпосереднє застосування в системах Контролю доступу і ідентифікації особистості

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Ляшенко О.М.*		
Теоретичне дослідження та практична реалізація	*Ляшенко О.М.*		
Практична реалізація об'єкта проектування	*Ляшенко О.М.*		
Висновки	*Ляшенко О.М.*		

7. Дата видачі завдання 01.11.2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Обгрунтування теми*	До 15.11.2022 р.	Виконано
2.	*Огляд літератури із досліджуваної проблеми*	До 15.12.2022 р.	Виконано
3.	*Розділ 1*	До 02.02.2023 р.	Виконано
4.	*Розділ 2*	До 02.03.2023 р.	Виконано
5.	*Висновки та пропозиції*	До 02.04.2023 р.	Виконано
6.	*Формування списку використаних джерел*	До 15.04.2023 р.	Виконано
7.	*Формування додатків*	До 02.05.2023 р.	Виконано
8.	*Оформлення ілюстративного матеріалу*	До 15.05.2023 р.	Виконано
9.	*Нормоконтроль*	До 25.05.2023 р.	Виконано
10.	*Інструментальна перевірка на академічний плагіат*	До 01.6.2023 р	Виконано
11.	*Представлення кваліфікаційної роботи бакалавра до захисту*	До 07.06.2023 р.	Виконано

Здобувач вищої освіти _____ Каламарчук В.С.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Ляшенко О.М.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Каламарчук В. С. Засоби тестування ІТ–продукту побудованого на мікросервісній архітектурі. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2023.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі розглянуто теоретичні підходи до тестування мікросервісних програмних продуктів. Проаналізовано характеристики та моделі хмарних обчислень, що визначають вимоги до тестового середовища. Описано архітектурні особливості мікросервісних систем (декомпозиція, незалежне розгортання, масштабування, взаємодія через API), які впливають на стратегії перевірки.

У другому розділі виконано дослідження та оцінювання підходів до формування і відбору тестових випадків для автоматизованого тестування. Розкрито показники та критерії результативності тестування програмного забезпечення. Розглянуто техніки проєктування тестів: поділ на еквівалентні класи, аналіз граничних значень, аналіз причинно-наслідкових зв’язків, булеві моделі тестування та підходи до побудови оптимальної моделі тестування.

У третьому розділі здійснено перевірку ефективності обраних методів на прикладі мікросервісного ІТ-продукту. Сформовано інформаційні моделі, необхідні для організації автоматизованого тестування. Запропоновано процедуру тестування з використанням підходу BDD: визначено загальну концепцію алгоритму, описано структуру BDD-тестування мікросервісів та подано формалізований алгоритм виконання тестів.

Ключові слова: мікросервісна архітектура, хмарні обчислення, автоматизоване тестування, BDD, інтеграційне тестування, критерії тестування, якість програмного забезпечення.

ABSTRACT

Kalamarchuk V. S. Tools for testing an IT product built on a microservice architecture. Manuscript.

Bachelor's qualification work OP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2023.

The qualification work consists of an introduction, three sections, conclusions, and a list of sources used.

The first section considers theoretical approaches to testing microservice software products. The characteristics and models of cloud computing that determine the requirements for the test environment are analyzed. The architectural features of microservice systems (decomposition, independent deployment, scaling, interaction via API) that affect testing strategies are described.

The second section studies and evaluates approaches to the formation and selection of test cases for automated testing. The indicators and criteria for the effectiveness of software testing are disclosed. Test design techniques are considered: division into equivalent classes, boundary value analysis, cause-and-effect analysis, Boolean testing models and approaches to building an optimal testing model.

In the third section, the effectiveness of the selected methods is verified using the example of a microservice IT product. Information models necessary for organizing automated testing are formed. A testing procedure using the BDD approach is proposed: the general concept of the algorithm is defined, the structure of BDD testing of microservices is described, and a formalized test execution algorithm is presented.

Keywords: microservice architecture, cloud computing, automated testing, BDD, integration testing, testing criteria, software quality.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасних умовах стрімкого розвитку інформаційних технологій та зростання кількості програмних рішень особливої ваги набуває питання забезпечення якості програмного продукту. Високий рівень конкуренції серед компаній-розробників IT-рішень змушує приділяти підвищену увагу стабільності, надійності та відповідності програмного забезпечення вимогам замовників і кінцевих користувачів.

Однією з ключових стадій життєвого циклу програмного продукту є етап формування та уточнення вимог, параметрів і критеріїв якості, що безпосередньо впливають на функціональні можливості системи. Важливим інструментом управління якістю у сфері розробки програмного забезпечення є тестування, яке дозволяє виявити дефекти та недоліки на ранніх етапах створення продукту.

Практика показує, що усунення помилок на етапі супроводу програмного забезпечення потребує значно більших фінансових і часових витрат порівняно з їх виправленням під час тестування. У разі пізнього виявлення дефектів загальний бюджет проєкту може зрости на десятки відсотків, що негативно впливає на економічну ефективність розробки.

Для перевірки відповідності програмного продукту задекларованим вимогам і характеристикам необхідно застосовувати комплексний підхід до тестування з використанням різних методів. Це дає змогу своєчасно виявити помилки, усунення яких сприяє підвищенню загальної якості програмного забезпечення.

Водночас традиційне ручне тестування не здатне повною мірою забезпечити необхідний рівень якості сучасних складних програмних систем. До його основних недоліків належать вплив людського фактору, значна трудомісткість повторного тестування після внесення змін, а також обмежені можливості перевірки продуктивності та навантаження.

У зв’язку з цим дедалі більшого поширення набуває автоматизоване тестування. Його ключовою перевагою є суттєве скорочення часу перевірки

програмних модулів, оскільки виконання тестових сценаріїв автоматизованими засобами є значно ефективнішим, ніж багаторазове ручне тестування. Крім того, один раз створені тестові сценарії можуть бути повторно використані в подальших циклах розробки, що сприяє оптимізації процесу тестування.

Автоматизоване тестування також дозволяє моделювати різні рівні навантаження на систему, забезпечуючи всебічну перевірку її працездатності та стабільності. Це, у свою чергу, підвищує надійність і якість програмного продукту.

Сучасні тенденції розвитку програмних систем передбачають використання нових архітектурних підходів, здатних задовольнити вимоги до масштабованості, доступності та гнучкості. Користувачі очікують отримати інтерактивний і динамічний досвід роботи з додатками на різних платформах, що робить хмарні технології та мікросервісну архітектуру особливо актуальними.

Мікросервісна архітектура передбачає побудову програмного продукту у вигляді набору незалежних сервісів, кожен з яких виконує окрему функцію та може розвиватися автономно. Такий підхід значно спрощує масштабування системи, прискорює процес розробки та оновлення, а також є особливо ефективним у хмарних середовищах.

Разом із тим, збільшення кількості мікросервісів призводить до ускладнення процесів інтеграції та взаємодії між ними. Помилки узгодження API можуть призвести як до затримок у впровадженні нового функціоналу, так і до повної непрацездатності програмного продукту. У деяких випадках такі дефекти виявляються лише під час експлуатації, що зумовлює необхідність повторного тривалого й дорогого циклу тестування.

Об’єкт дослідження – методи автоматизованого тестування програмного забезпечення, реалізованого з використанням мікросервісної архітектури.

Предмет дослідження – процес розробки сценаріїв автоматизованого тестування програмного продукту, побудованого на основі мікросервісної архітектури.

Мета роботи – розроблення ефективного алгоритму автоматизованого тестування програмного продукту з мікросервісною архітектурою, спрямованого на підвищення якості програмних систем і скорочення термінів постачання оновлень замовнику.

Для досягнення поставленої мети в роботі передбачається виконання таких завдань:

– дослідити теоретичні основи та сучасні підходи до автоматизованого тестування програмних продуктів, побудованих на мікросервісній архітектурі;

– проаналізувати існуючі методики автоматизованого тестування хмарних ІТ-рішень;

– розробити алгоритм автоматизованого тестування хмарного програмного продукту на основі мікросервісної архітектури, який забезпечить підвищення якості та швидкості доставки оновлень.

Теоретична значущість роботи полягає у формуванні вдосконаленого підходу до автоматизованого тестування хмарних ІТ-продуктів, побудованих на мікросервісній архітектурі, із застосуванням тестового дублера. Отримані результати можуть бути використані в подальших дослідженнях, спрямованих на підвищення якості програмного забезпечення.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ПІДХОДИ ДО ТЕСТУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

1.1 Характеристики та моделі хмарних обчислень

Хмарні обчислення доцільно трактувати як підхід, що забезпечує зручний і повсюдний мережевий доступ до спільного пулу обчислювальних ресурсів. До таких ресурсів належать мережі, серверні потужності, сховища, прикладні сервіси та програмні компоненти, які надаються користувачу «на вимогу». Їх можна швидко отримувати та масштабувати з мінімальними витратами часу на адміністрування й без необхідності постійної взаємодії з провайдером.

У загальноприйнятому підході виділяють п’ять ключових характеристик хмарних рішень та три базові сервісні моделі.

Основні характеристики хмарних обчислень:

1) Самообслуговування на вимогу. Споживач самостійно замовляє й налаштовує потрібні ресурси (наприклад, обчислювальний час, дисковий простір, мережеві сховища) без прямого залучення працівників провайдера.

2) Широкий мережевий доступ. Послуги доступні через стандартні мережеві механізми та можуть використовуватися на різних типах пристроїв і платформ – від смартфонів і планшетів до ноутбуків та робочих станцій.

3) Об’єднання ресурсів (пулінг). Провайдер застосовує багатокористувацьку модель, у межах якої фізичні й віртуальні ресурси спільно використовуються різними клієнтами та перерозподіляються динамічно залежно від актуального попиту. Зазвичай клієнт не знає точного фізичного розташування ресурсів, однак може задавати його на узагальненому рівні (наприклад, країна, регіон або конкретний дата-центр). До ресурсів належать зберігання, обробка, оперативна пам’ять і пропускна здатність мережі.

4) Еластичність і масштабованість. Хмарна інфраструктура дозволяє швидко збільшувати або зменшувати потужності, інколи – автоматично, відповідно до навантаження. Для користувача ресурси виглядають практично необмеженими та можуть надаватися у потрібний момент.

5) Вимірюваність і облік споживання. Системи хмарного провайдера здійснюють моніторинг, контроль і оптимізацію використання ресурсів на відповідному рівні абстракції (наприклад, обсяг сховища, обчислювальні ресурси, пропускна здатність, активні облікові записи). Це підвищує прозорість для обох сторін, оскільки використання послуг можна відстежувати, а звіти — формувати автоматично.

Сервісні моделі за NIST визначає три типові моделі надання хмарних сервісів:

– IaaS (Infrastructure as a Service) – інфраструктура як сервіс;

– PaaS (Platform as a Service) – платформа як сервіс;

– SaaS (Software as a Service) – програмне забезпечення як сервіс.

Кожна з моделей спирається на набір базових складових:

1) Апаратна частина: сервери, мережеве обладнання, системи зберігання даних;

2) Віртуалізація: програмно-апаратний комплекс, що дозволяє запускати різні процеси та сервіси на одному фізичному ресурсі 2];

3) Платформа: програмне середовище для виконання застосунків (наприклад, Apache, PHP, .NET, JRE);

4) Додаток: прикладний рівень, створений для розв’язання конкретних бізнес-завдань.

Конфігурацію IT-продукту до використання хмарних сервісів наведено на рисунку 1.1.

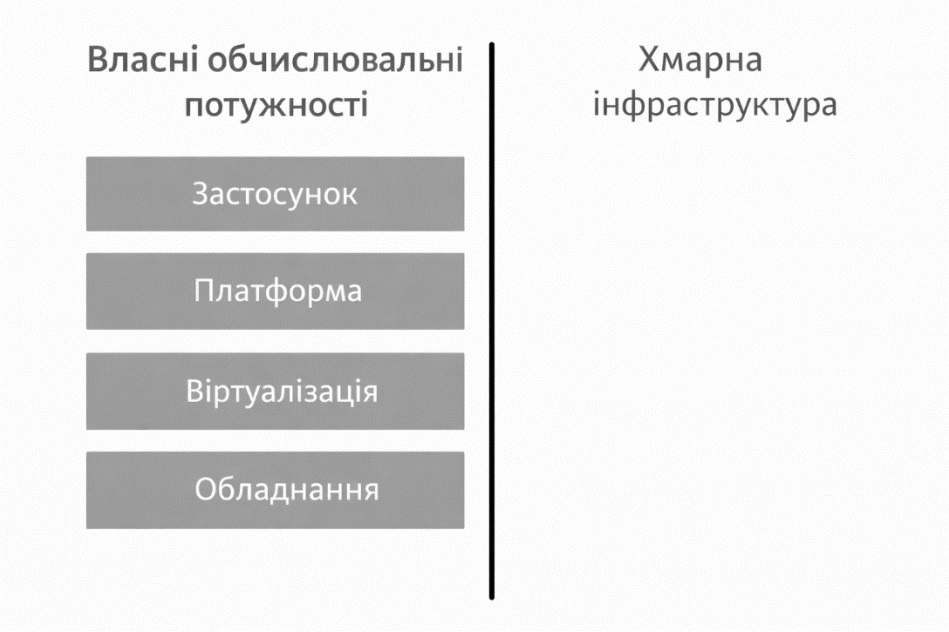

Рисунок 1.1 – Конфігурацію IT-продукту до використання хмарних сервісів

Згідно з рисунком 1.1, усі складові рішення розміщені на власних обчислювальних ресурсах організації [2].

Конфігурації при використанні хмарних сервісів:

1) IaaS. Споживач отримує в оренду обчислювальні потужності, дискові сховища та мережеві ресурси, не керуючи безпосередньо фізичною інфраструктурою провайдера. Водночас він може обирати операційні системи, необхідний обсяг диску та прикладні компоненти, які встановлюються. Також можливі налаштування окремих мережевих елементів (наприклад, правил брандмауера). У цій моделі віртуалізація входить до хмарного шару. Прикладом є підхід оплати «pay-as-you-go», коли послуги оплачуються за фактичне використання (як у великих провайдерів на кшталт Amazon із широким набором сервісів).

2) PaaS. Користувач отримує середовище для розгортання та виконання застосунків із підтримкою певних мов програмування та інструментів, які визначає провайдер. Контроль над інфраструктурою відсутній, проте клієнт налаштовує параметри застосунку під власні потреби, а сам додаток залишається у його повному керуванні. Типовий приклад — веб-хостинг/платформні сервіси, що дають змогу швидко розгорнути сайт та

користуватися базами даних без самостійного встановлення й обслуговування.

3) SaaS. Кінцевий користувач працює з готовим програмним продуктом, який функціонує у хмарній інфраструктурі провайдера і доступний через браузер незалежно від місця, часу та типу пристрою. Зазвичай можливості налаштування обмежуються параметрами інтерфейсу (наприклад, локалізацією та візуальними налаштуваннями). Класичний приклад — хмарні поштові сервіси: користувачеві доступний функціонал, але він не взаємодіє з питаннями зберігання даних чи фізичним розміщенням серверів.

Конфігурацію рішення на базі IaaS подано на рисунку 1.2: у хмарі розміщуються апаратний шар і віртуалізація [28]. Такий підхід дозволяє організаціям оперативно збільшувати або зменшувати орендовані ресурси, покриваючи пікові навантаження без значних капітальних витрат.

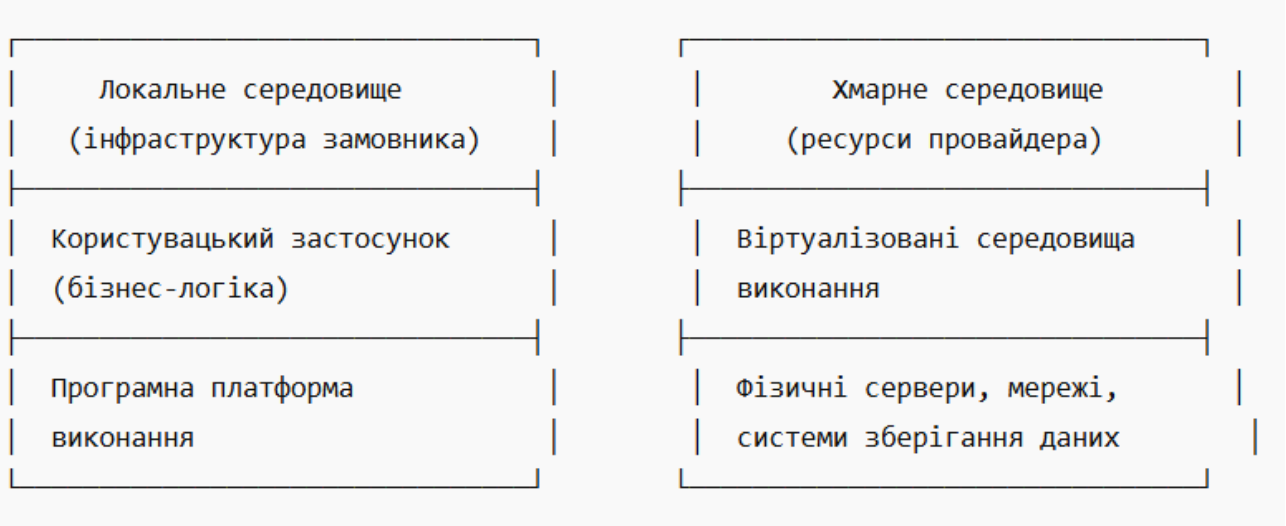

Рисунок 1.2 – Конфігурація рішення із сервісним рівнем IaaS

Конфігурацію рішення на рівні PaaS наведено на рисунку 1.3.

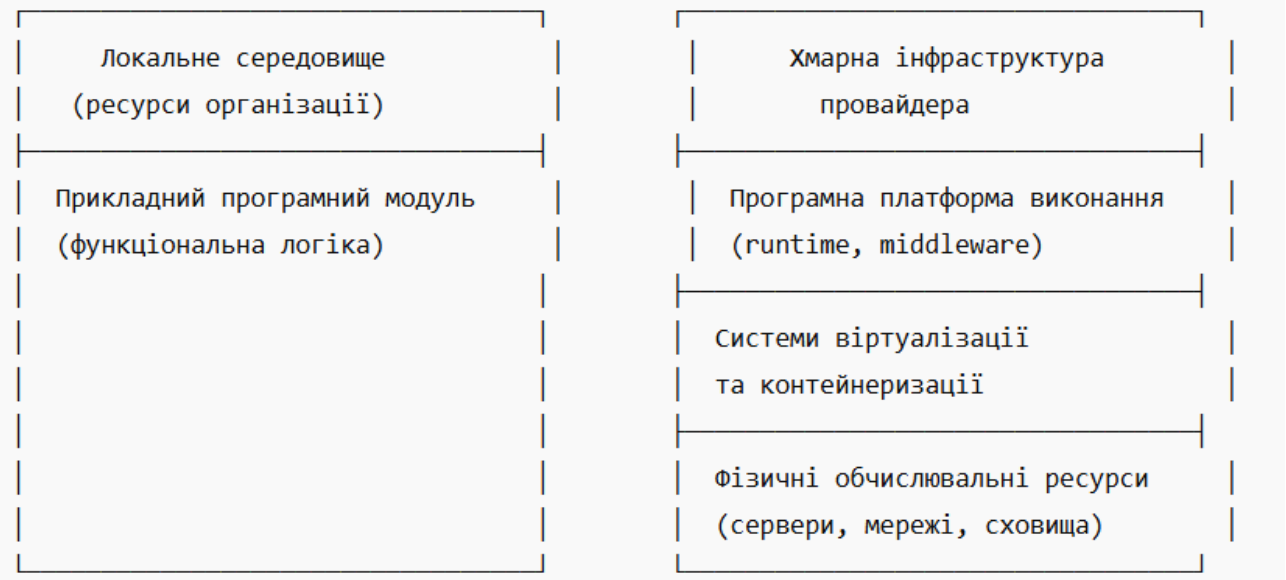

Рисунок 1.3 – Конфігурація рішення із сервісним рівнем PaaS

Відповідно до рисунка 1.3, у власності замовника залишається переважно прикладний компонент, тоді як платформа, віртуалізація й апаратна частина переходять у зону відповідальності провайдера.

Конфігурацію рішення для SaaS показано на рисунку 1.4.

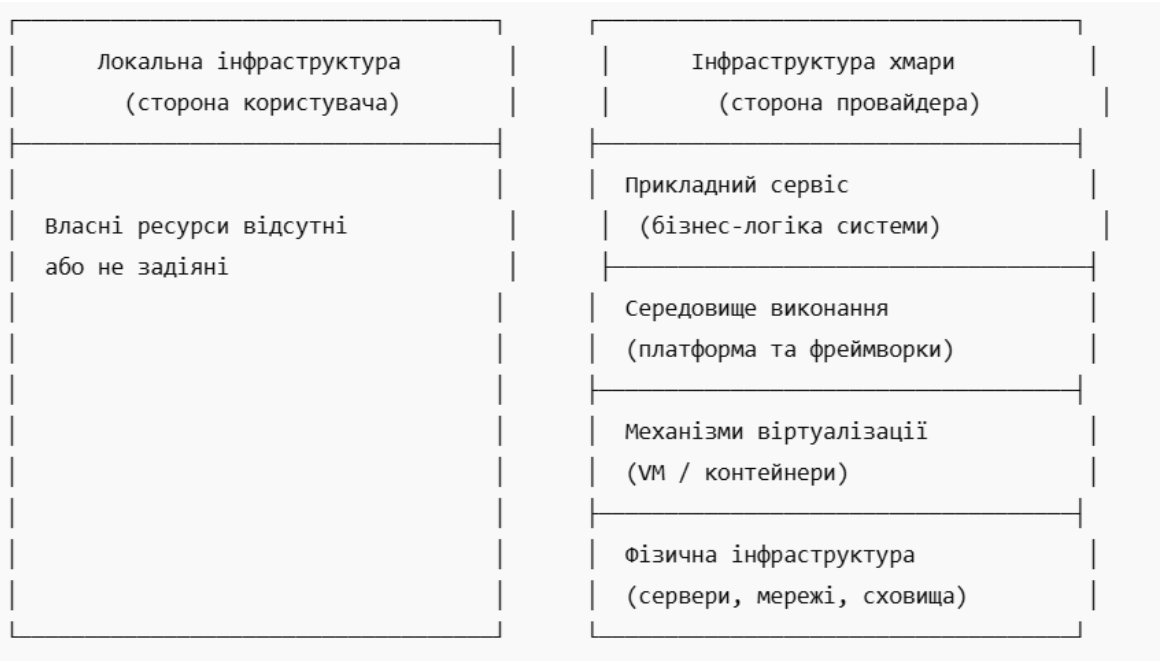

Рисунок1.4 – Конфігурація рішення з сервісним рівнем SaaS

Як видно з рисунка 1.4, у цій моделі всі рівні (додаток, платформа, віртуалізація й апаратна інфраструктура) знаходяться у хмарному середовищі

та доступні користувачу в режимі сервісу.

Отже, порівняння моделей IaaS, PaaS і SaaS дозволяє зробити висновок: залежно від вимог замовника та ресурсних можливостей IT-компанії можна обрати найбільш придатний формат використання хмарних сервісів. Оперативний доступ до інфраструктури та гнучке масштабування спрощують розробку й тестування IT-продукту, зменшуючи адміністративні витрати та потребу в додаткових погодженнях із постачальниками.

1.2 Архітектурні особливості мікросервісних програмних систем

Стрімкий розвиток інформаційних технологій та постійна поява нових цифрових рішень формують підвищені вимоги до архітектури сучасних програмних систем. Від IT-продуктів очікується здатність стабільно працювати на різних платформах і пристроях, швидко адаптуватися до змін, підтримувати регулярні оновлення та забезпечувати мінімальний часовий розрив між ідеєю й виходом першої робочої версії на ринок. Окрім цього, важливим є раціональний вибір технологій для розв’язання конкретних задач і можливість багаторазового використання окремих функціональних компонентів.

Одним із підходів, що відповідає зазначеним вимогам, є мікросервісна архітектура. Вона ґрунтується на побудові програмного забезпечення у вигляді набору автономних сервісів, які слабо залежать один від одного та взаємодіють через чітко визначені інтерфейси. Кожен мікросервіс реалізує окрему бізнес-функцію та може розроблятися, тестуватися й розгортатися незалежно від інших компонентів системи.

До характерних рис мікросервісної архітектури належать:

1) функціональна зосередженість – кожен сервіс має обмежену відповідальність і призначений для виконання однієї логічно завершеної задачі, що спрощує його супровід і модифікацію;

2) бізнес-орієнтовані межі – межі між сервісами визначаються не технічними, а бізнес-вимогами, що дозволяє чітко відокремити функціональні області системи;

3) ізольованість компонентів – мікросервіси можуть функціонувати як окремі процеси або розгортатися у хмарному середовищі (зокрема, у моделі PaaS), що зменшує вплив змін в одному сервісі на роботу всієї системи;

4) гнучкість технологічних рішень – для кожного сервісу можна обирати найбільш доцільні мови програмування, платформи та сховища даних, виходячи з вимог до продуктивності та надійності;

5) підвищена відмовостійкість – збій одного сервісу не призводить до повної зупинки системи, оскільки проблемний компонент можна локалізувати, зберігши працездатність інших частин;

6) вибіркове масштабування – на відміну від монолітних систем, у мікросервісній архітектурі масштабуються лише ті компоненти, які цього реально потребують, що дозволяє ефективніше використовувати ресурси та зменшувати витрати;

7) спрощене розгортання – автоматизовані процеси доставки дозволяють оновлювати окремі сервіси без зупинки всієї системи, швидко повертатися до попередніх версій і оперативно надавати новий функціонал користувачам.

Завдяки цим властивостям мікросервісна архітектура широко застосовується у великих розподілених системах та хмарних середовищах, що підтверджується практикою таких компаній, як Amazon і Netflix. Важливою перевагою цього підходу є також можливість повторного використання окремих сервісів у різних програмних продуктах або сценаріях застосування.

Разом із перевагами мікросервісний підхід створює додаткові виклики, зокрема під час розробки та тестування. Системи часто взаємодіють із зовнішніми сервісами, які можуть бути недоступними або ще не реалізованими на етапі тестування. Це ускладнює перевірку коректності роботи програмного забезпечення, особливо в автоматизованих тестах, де необхідно враховувати як стандартні, так і аварійні сценарії взаємодії.

Навіть у межах однієї системи окремі мікросервіси можуть перебувати на різних стадіях готовності, що створює потребу в ізоляції тестованого компонента від його залежностей. Для вирішення цієї проблеми застосовують

тестові дублери – спеціальні об’єкти, які імітують поведінку реальних сервісів або модулів під час тестування.

Поняття тестового дублера було узагальнене Джерардом Мезаросом як колективний термін для різних типів замінників реальних компонентів системи. Залежно від рівня складності та призначення виділяють декілька основних різновидів таких об’єктів:

1) фіктивні об’єкти (fake) – спрощені реалізації інтерфейсів без внутрішньої логіки, що використовуються для передачі необхідних параметрів;

2) заглушки (stub) – компоненти, які повертають заздалегідь визначені значення у відповідь на виклики та дозволяють ізолювати тестований модуль від зовнішніх залежностей;

3) імітації (mock) – більш складні дублери, здатні перевіряти правильність викликів, порядок взаємодії та коректність переданих параметрів;

4) макети – динамічно створювані об’єкти, що конфігуруються під час виконання тесту та можуть поводитися як різні типи дублерів;

5) тестові шпигуни – об’єкти, які, окрім повернення даних, фіксують інформацію про виклики методів для подальшого аналізу в тестах.

Застосування тестових дублерів у процесі автоматизованого тестування мікросервісних систем дозволяє проводити перевірки поетапно, ізольовано та незалежно від стану інших компонентів або зовнішніх сервісів. Такий підхід суттєво знижує складність тестування, підвищує його надійність і сприяє загальному покращенню якості IT-продукту.

Логіку роботи програми, у якій передбачено перевірку облікових даних користувача, із застосуванням фіктивного модуля наведено на рисунку 1.5.

Згідно з рисунком 1.5, фіктивні модулі не реалізують повноцінної бізнес-логіки й використовуються переважно у випадках, коли для виконання тесту потрібна лише наявність параметрів або формальне дотримання інтерфейсу. У ролі fake можуть виступати й “порожні” значення, однак коректніше розглядати фіктивний модуль як спрощену реалізацію інтерфейсу чи базового класу, що не містить реальних обчислень.

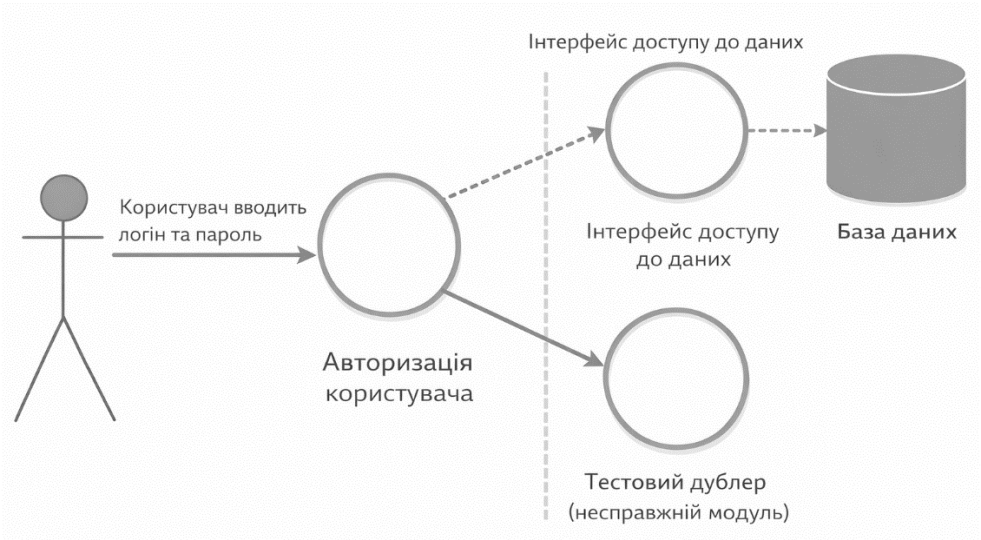

Рисунок 1.5 – Логічна схема функціонування програми із застосуванням фіктивного модуля

Схему роботи програми для обчислення середніх оцінок студентів із використанням заглушки (stub) подано на рисунку 1.6.

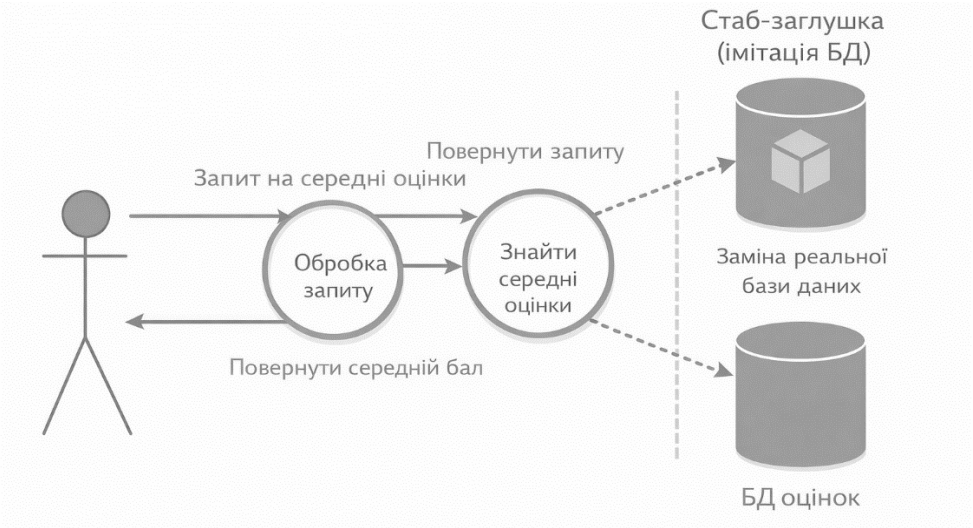

Рисунок 1.6 – Логічна схема роботи програми із застосуванням заглушки

Відповідно до рисунка 1.6, головне призначення заглушки полягає в тому, щоб повертати тестованому модулю наперед визначений (детермінований) результат, необхідний для перевірки конкретного сценарію без залучення зовнішніх залежностей.

Логічну схему роботи програми, що виконує надсилання повідомлень, за умови використання імітації (mock) наведено на рисунку 1.7.

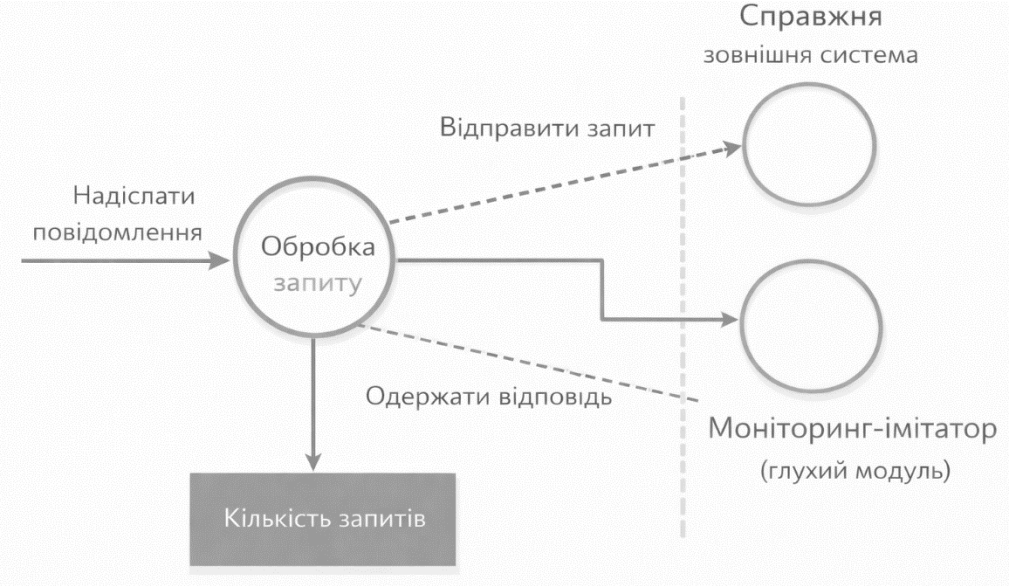

Рисунок 1.7 – Логічна схема роботи програми із застосуванням імітації

Як видно з рисунка 1.7, імітація не лише повертає значення, а й може контролювати правильність переданих параметрів, відстежувати послідовність викликів та формувати відповідь за заданими правилами. При цьому mock не є повноцінним робочим компонентом системи, хоча частково відтворює його поведінку в тестовому середовищі [4].

Окрім наведених типів, на практиці також застосовують:

1) Макет (mock object / mock generated) – об’єкт, який динамічно генерується спеціальною бібліотекою. Тестувальник не взаємодіє з «реальним» кодом реалізації інтерфейсу чи базового класу, натомість задає правила: які дані повертати, які виклики очікувати, які перевірки виконувати тощо. Залежно від налаштувань такий макет може поводитися як fake, stub або spy.

2) Тестовий шпигун (spy) – об’єкт, подібний до заглушки, але додатково фіксує факти взаємодії: які методи викликались, скільки разів, з якими аргументами. Це дозволяє модульним тестам перевіряти, чи відповідає реальна поведінка системи очікуванням (наприклад, сервіс «пошти», який зберігає інформацію про кількість надісланих повідомлень) [5].

Отже, під час автоматизованого тестування IT-продуктів, побудованих на мікросервісній архітектурі, доцільно використовувати тестові дублери, які зовні відтворюють взаємодію з мікросервісами, але є значно простішими за внутрішньою реалізацією. Їх основна перевага полягає в можливості ізолювати компоненти, виконувати поетапне тестування кожного мікросервісу, а надалі – перевіряти взаємодію сервісів між собою. У результаті зменшується складність тестової інфраструктури та підвищується загальна якість тестування програмного продукту.

1.3 Методи автоматизованої перевірки програмних систем

Автоматизоване тестування програмного забезпечення є складовою процесу забезпечення якості та здійснюється на етапі розроблення програмних продуктів із використанням спеціалізованих програмних інструментів. Такі інструменти забезпечують виконання заздалегідь визначених тестових сценаріїв та автоматичну перевірку отриманих результатів. Комплексне застосування автоматизації дозволяє істотно зменшити часові витрати на тестування та підвищити керованість цього процесу.

Основними причинами впровадження автоматизованого тестування є необхідність підвищення якості програмного продукту та оптимізація ресурсів, що витрачаються на перевірку його працездатності. Застосування автоматизованих засобів дає змогу охопити значно ширший набір тестових випадків, у тому числі таких, які складно або неможливо виконати вручну. Окрім цього, автоматизація мінімізує вплив людського фактору, що знижує ймовірність помилок у результатах тестування. Оскільки виконання автоматичних тестів потребує значно менше часу, ніж аналогічні ручні перевірки, це сприяє суттєвій економії часу.

До ключових переваг автоматизованого тестування належать:

– істотне скорочення тривалості виконання тестового набору порівняно з ручним тестуванням;

– можливість моделювання навантажень і сценаріїв, які не піддаються ручній перевірці;

– зменшення залежності результатів тестування від дій людини;

– можливість запуску тестів у будь-який час, зокрема без участі тестувальника та поза робочими годинами.

Водночас автоматизоване тестування має й низку обмежень. Зокрема, його впровадження потребує значних початкових витрат часу та ресурсів, пов’язаних із підготовкою тестової інфраструктури й навчанням персоналу. Для ефективної роботи з такими інструментами необхідні фахівці з відповідним рівнем кваліфікації. Після виконання тестів обов’язковим є детальний аналіз отриманих результатів. Будь-які зміни в програмному коді можуть вимагати оновлення автоматичних тестів, а помилка в одному тесті здатна призвести до спотворення результатів наступних перевірок. Крім того, не всі функціональні вимоги можливо коректно перевірити автоматизованими засобами в межах обраного інструментарію.

Таким чином, доцільність використання автоматизованого тестування не є універсальною та має визначатися з урахуванням конкретних умов проєкту. Автоматизація є обґрунтованою у випадках повторюваних тестів, навантажувального тестування, перевірки продуктивності або ситуацій, де ручне створення необхідних умов є складним чи неможливим. Водночас автоматизація перевірки функціональних вимог часто потребує додаткового аналізу та може бути менш ефективною.

Важливу роль у побудові тестового середовища відіграє віртуалізація, яка забезпечує гнучке керування конфігураціями середовища тестування. Використання віртуальних машин спрощує налаштування операційних систем і програмних компонентів, зменшує витрати на інфраструктуру та дозволяє швидко відтворювати необхідні умови для тестування. Зазвичай вимоги до тестового середовища фіксуються в окремих документах, що дає змогу планувати його використання або створення нового середовища.

Ліза Кріспен і Джанет Грегорі запропонували модель тестової піраміди у вигляді матриці, яка дозволяє систематизувати всі види тестування, необхідні

для забезпечення якості програмного продукту (рис. 1.8). Відповідно до цієї моделі, перший сектор (Q1) охоплює модульні тести, призначені для перевірки окремих функцій або методів. До цієї категорії належать також тести, реалізовані відповідно до підходу Test Driven Development (TDD), а також property-based testing. Такі тести виконуються без запуску сервісів та без взаємодії із зовнішніми системами, що дозволяє запускати їх у великій кількості за мінімальний час.

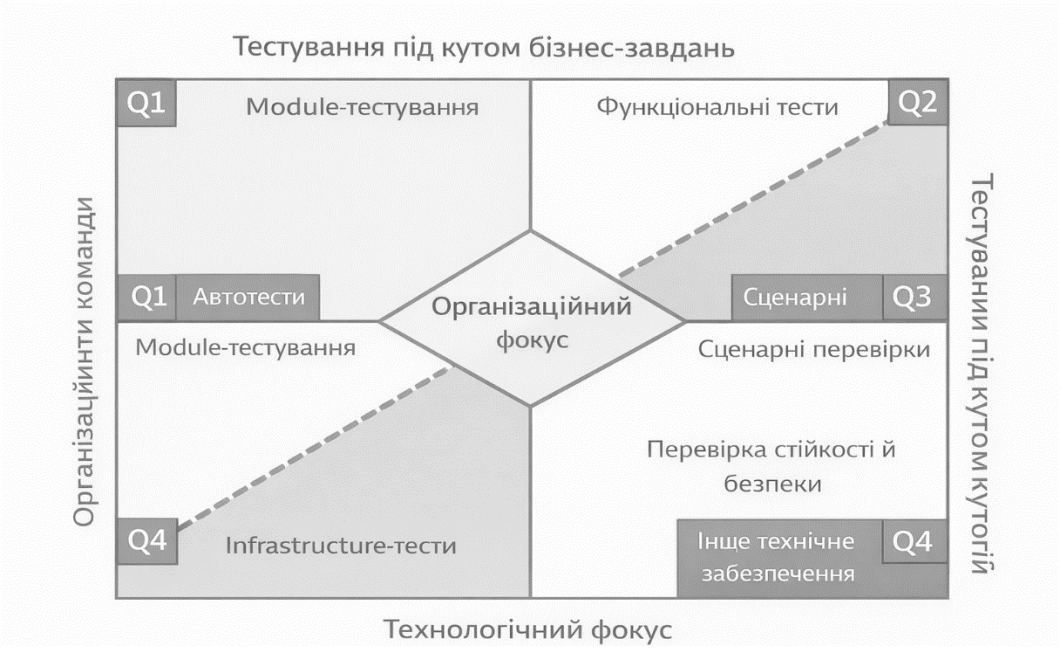

Рисунок 1.8 – Сектори тестування за Лізою Кріспен та Джанет Грегорі

Другий сектор (Q2) представлений функціональними тестами, які перевіряють відповідність реалізованого функціоналу вимогам замовника. Вони виконуються на рівні всієї системи та часто реалізуються через користувацький інтерфейс, зокрема веб-інтерфейс. Результатом такого тестування є підтвердження готовності продукту до використання в реальних умовах.

Третій сектор (Q3) включає дослідницьке тестування, спрямоване на оцінку зручності та логічності роботи системи з точки зору кінцевого користувача. Цей тип тестування виконується виключно вручну та зазвичай проводиться спільно з представниками замовника на етапі приймального тестування (User Acceptance Testing). Такий підхід дозволяє не лише виявити недоліки, а й сформувати вимоги до подальшого розвитку системи.

Четвертий сектор (Q4) охоплює нефункціональні перевірки, зокрема навантажувальне тестування. Для їх реалізації застосовуються спеціалізовані інструменти, що дозволяють оцінити продуктивність, стабільність, масштабованість, надійність та інші характеристики системи. Нефункціональні вимоги є критично важливими для бізнесу, оскільки низька швидкодія або порушення безпеки можуть призвести до фінансових та репутаційних втрат.

Отже, тестування програмного забезпечення виконується з метою виявлення помилок, підтвердження надійності та перевірки відповідності продукту поставленим завданням. Чітке розуміння функціональних і нефункціональних вимог, інтеграційних зв’язків і аспектів безпеки значно полегшує процес проєктування та дозволяє обґрунтовано обирати архітектурні рішення й інструменти тестування.

Проведений аналіз теоретичних засад і методів автоматизованого тестування хмарних програмних продуктів, побудованих на мікросервісній архітектурі, свідчить про необхідність комплексного підходу до вибору технологій і ресурсів. Використання оптимальної моделі хмарних сервісів забезпечує оперативний доступ до обчислювальних ресурсів без суттєвих додаткових витрат.

Мікросервісна архітектура створює умови для незалежної розробки та тестування окремих компонентів, кожен з яких виконує чітко визначену функцію. У таких умовах ефективним є застосування автоматизованого тестування із використанням тестових дублерів, що імітують взаємодію з іншими підсистемами. Аналіз типів тестових дублерів і принципів їх роботи дозволяє обрати оптимальний варіант для конкретного мікросервісу.

Разом з тим слід зазначити, що впровадження автоматизованого тестування потребує обґрунтованого планування, наявності кваліфікованих спеціалістів, а також додаткових часових і фінансових витрат.

РОЗДІЛ 2
ДОСЛІДЖЕННЯ ТА ОЦІНЮВАННЯ ПІДХОДІВ ДО АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ІТ-ПРОДУКТІВ, РЕАЛІЗОВАНИХ НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Показники та критерії результативності тестування програмного забезпечення

Систематизацію ключових підходів до формування тестових наборів під час перевірки програмного забезпечення наведено на рисунку 2.1.

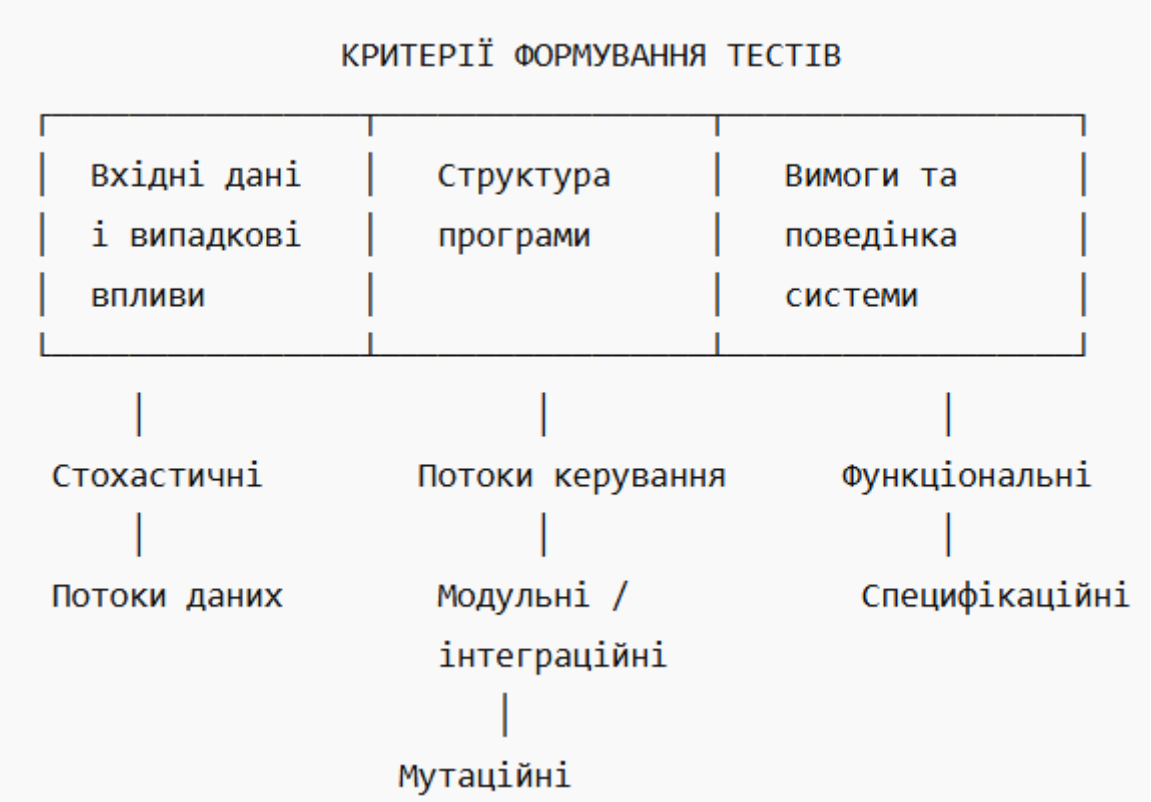

Рисунок 2.1 – Класифікація критеріїв тестування

Характеристика основних критеріїв:

1) Імовірнісні (стохастичні) критерії. Даний підхід застосовується для перевірки складних програмних компонентів у випадках, коли повний перелік детермінованих тестів є надмірно великим або практично недосяжним. Для цього використовуються генератори випадкових вхідних послідовностей, на основі яких формуються пари значень вхідних і вихідних параметрів. Отримані

комбінації дозволяють оцінити поведінку системи в умовах невизначеності та нестандартних сценаріїв.

2) Критерії, орієнтовані на потоки керування. Цей клас критеріїв базується на аналізі внутрішньої логіки програми та передбачає доступ до її вихідного коду. Тестування виконується відповідно до структури графа керування, що описує переходи між операторами та блоками. Такий підхід доцільно застосовувати під час модульного та інтеграційного тестування, коли необхідно перевірити всі можливі шляхи виконання коду.

3) Критерії аналізу потоків даних. У межах цього підходу формуються тестові сценарії, що забезпечують перевірку всіх класів вхідних даних та їх впливу на роботу окремих компонентів або мікросервісів. Значна увага приділяється граничним значенням, комбінаціям параметрів і некоректним структурам даних. Хоча такий метод дозволяє суттєво скоротити кількість необхідних перевірок, його реалізація часто потребує значних часових ресурсів.

4) Функціональні критерії тестування. Функціональне тестування є ключовим інструментом перевірки відповідності програмного продукту вимогам замовника. Воно здійснюється без урахування внутрішньої реалізації системи, тобто за принципом «чорної скриньки».

Найпоширенішими різновидами є:

– тестування функцій, яке передбачає перевірку коректності роботи окремих можливостей системи;

– тестування відповідності специфікації, що забезпечує перевірку виконання кожного пункту технічних вимог.

У практиці такі критерії часто поєднують риси структурного та функціонального аналізу, що дозволяє досягти більш повного тестового покриття.

5) Мутаційні критерії. Мутаційне тестування ґрунтується на штучному внесенні незначних змін (мутацій) у програмний код з метою створення альтернативних версій програми — мутантів. Основна ідея полягає в тому, що ефективний тестовий набір повинен виявляти всі такі штучно створені помилки. Якщо тестові сценарії успішно ідентифікують більшість мутантів як

некоректні, можна зробити висновок про достатню якість тестування та надійність системи.

Отже, аналіз підходів до формування тестових наборів свідчить, що ключовою проблемою тестування є пошук оптимального балансу між повнотою перевірки та кількістю тестів. Надмірна кількість сценаріїв ускладнює процес тестування, тоді як недостатнє покриття підвищує ризик пропуску критичних дефектів. Раціональне поєднання різних критеріїв дозволяє забезпечити комплексну перевірку програмного продукту без необґрунтованих витрат ресурсів.

2.2 Автоматизоване тестування хмарних програмних систем на основі мікросервісної архітектури

Одним із ключових викликів у процесі тестування програмного забезпечення є визначення обґрунтованого обсягу тестових сценаріїв, достатнього для підтвердження коректності роботи системи. Надмірна кількість тестів призводить до перевитрат ресурсів, тоді як недостатнє покриття підвищує ризик невиявлених дефектів.

Для розв’язання цієї проблеми в практиці автоматизованого тестування застосовують низку перевірених методик, серед яких найбільш поширеними є:

– поділ вхідних даних на еквівалентні множини (Equivalence Partitioning);

– аналіз граничних значень параметрів (Boundary Value Analysis);

– побудова моделей причинно-наслідкових залежностей (Cause-Effect Analysis) [6].

2.2.1 Поділ на еквівалентні класи

Метод еквівалентного поділу ґрунтується на припущенні, що вхідні дані можна згрупувати у множини, в межах яких система поводиться однаково. Для кожного такого класу достатньо перевірити лише один репрезентативний набір значень. Це дозволяє суттєво скоротити кількість тестів, виключивши сценарії, які не несуть додаткової діагностичної цінності та призводять до однакових результатів виконання програми.

2.2.2 Аналіз граничних значень

На рисунку 2.2 наведено приклад поділу простору вхідних параметрів на допустимі та недопустимі області, де кожна з них представлена відповідним класом еквівалентності.

Рисунок 2.2 – Виділення класів еквівалентності

Метод перевірки граничних значень доповнює попередній підхід і передбачає тестування не лише самих меж допустимих діапазонів, але й значень, що знаходяться безпосередньо поблизу цих меж. Практика показує, що саме на кордонах діапазонів найчастіше виникають помилки реалізації.

Зазвичай аналіз граничних значень застосовується разом із поділом на еквівалентні класи. Водночас обмеженням цього методу є неможливість охоплення всіх можливих комбінацій багатовимірних вхідних даних, що створює ризик пропуску окремих критичних сценаріїв (рис. 2.2).

2.2.3 Аналіз причинно-наслідкових зв’язків

Метод причинно-наслідкових діаграм спрямований на формування ефективних тестових наборів шляхом формалізації логічних залежностей між умовами та результатами роботи системи. Процес побудови тестів за цим підходом включає кілька послідовних етапів:

– аналіз вимог і специфікації з метою визначення вхідних умов (причин) та очікуваних реакцій системи (наслідків);

– побудову логічної (булевої) схеми, що відображає взаємозв’язки між виявленими причинами та наслідками;

– трансформацію цієї схеми у таблицю рішень шляхом системного перебору можливих комбінацій умов;

– формування набору тест-кейсів на основі стовпців отриманої таблиці рішень.

2.2.4 Булева модель тестування

Приклад булевої моделі, побудованої для тестування, подано на рисунку 2.3.

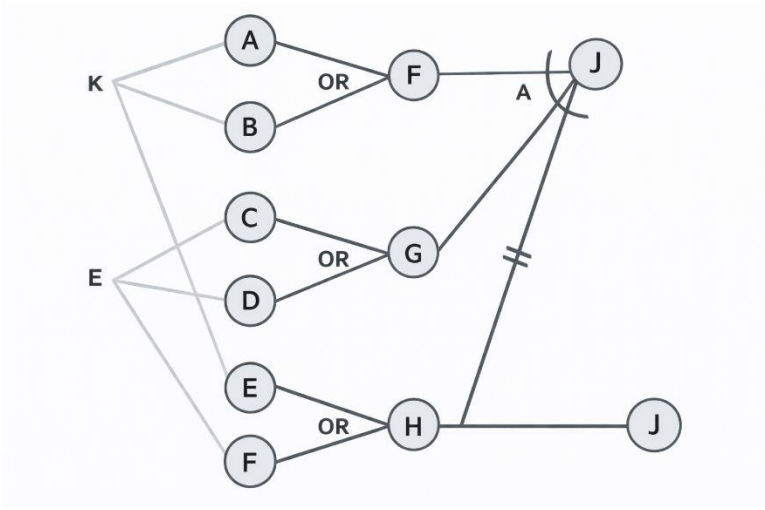

Рисунок 2.3 – Приклад булевого графа тестування

Зазначений метод забезпечує високу інформативність сформованих тестових сценаріїв та дозволяє виявляти значну кількість логічних помилок. Разом з тим, процес створення причинно-наслідкових моделей є складним і трудомістким, особливо у випадку ручної реалізації. Частину операцій, зокрема перетворення логічних схем у таблиці рішень та генерацію тестів, доцільно автоматизувати за допомогою спеціалізованих інструментів.

2.2.5 Оптимальна модель тестування

Для вибору оптимальної методики тестування необхідно попередньо розробити специфікацію тестового процесу, яка визначає обсяг перевірок, використовувані підходи та інструменти. Узагальнену послідовність проєктування і впровадження автоматизованого тестування програмного

продукту наведено на рисунку 2.4.

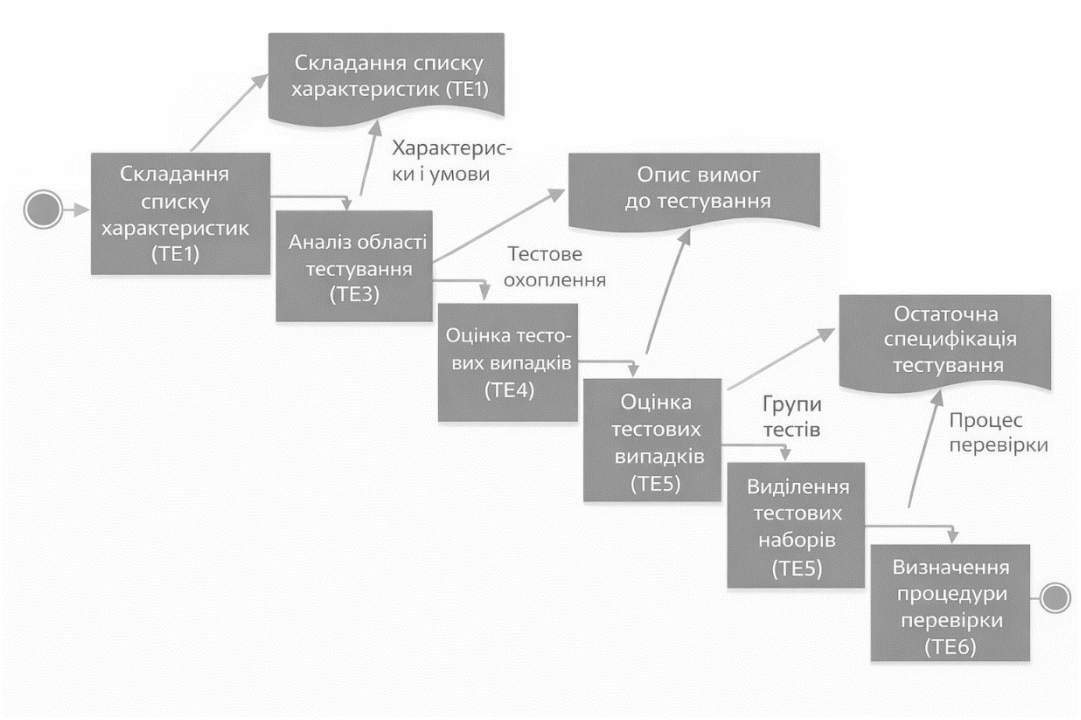

Рисунок 2.4 –Реалізація процесу тестування

Відповідно до рисунка 2.4, процес автоматизованого тестування розпочинається з формування переліку характеристик системи та визначення умов, за яких виконуватиметься перевірка. Після цього аналізується область тестового покриття, а тестові сценарії впорядковуються за пріоритетністю. Наступним кроком є підготовка специфікації тестування, яка містить опис методики проведення перевірок та узгоджений фінальний набір тестів. З практичної точки зору доцільно ділити тестові набори на менші логічні компоненти, що можуть бути використані повторно в різних сценаріях тестування.

Ключовим чинником ефективності автоматизованого тестування є можливість багаторазового застосування створених тестів. Саме повторне використання тестових сценаріїв дозволяє суттєво скоротити витрати часу та ресурсів у процесі контролю якості. Життєвий цикл тесту подано на рисунку 2.5.

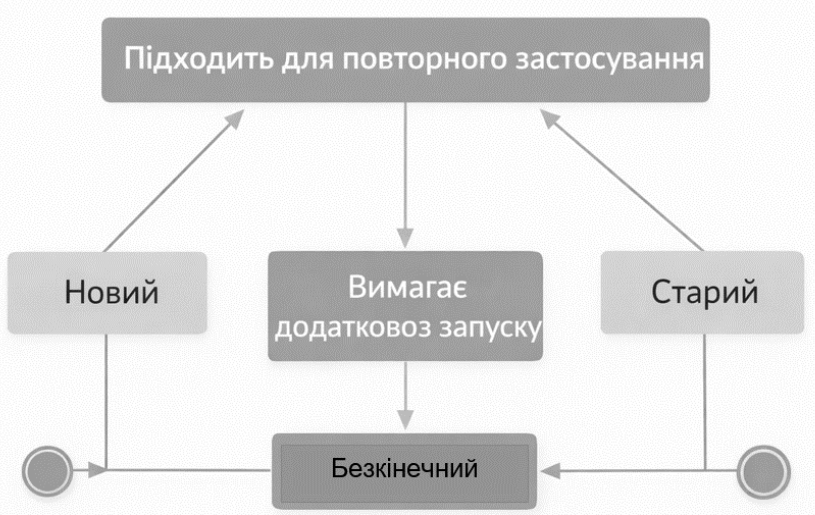

Рисунок 2.5 – Життєвий цикл тесту

Згідно з рисунком 2.5, кожен тест проходить низку послідовних стадій: від створення нового тесту до його повторного використання або втрати актуальності. У процесі експлуатації тест може неодноразово запускатися та залишатися придатним для подальшого застосування або ж з часом стати застарілим і бути виведеним з використання, що означає завершення його життєвого циклу.

Існує декілька типових ситуацій, за яких виникає потреба у коригуванні або оновленні тестових наборів:

– розробка нових, актуальних тестових сценаріїв;

– повторне виконання наявних тестів;

– внесення змін до програмного коду.

Тест втрачає актуальність і підлягає заміні у випадку, якщо змінюються вхідні дані або умови виконання. Коригування вхідної інформації призводить до зміни логіки виконання та, відповідно, кінцевих результатів. У зв’язку з цим доцільно розглядати різні рівні повторного використання тестів.

На першому рівні повторне застосування тестів не передбачається – кожен тест використовується лише один раз.

Другий рівень допускає повторне використання лише вхідних даних. Така ситуація можлива, коли частина програмного коду залишається незмінною або

використовується в нових модулях. Проте після змін у системі або специфікації результати виконання можуть істотно відрізнятися від попередніх запусків.

На третьому рівні повторно використовуються як вхідні, так і вихідні дані тестів. До цього рівня належить функціональне тестування. Якщо в програмному продукті змінюється реалізація, але зберігається загальна функціональність, уже створені функціональні тести можуть застосовуватися багаторазово для перевірки коректності роботи системи. При цьому очікується, що результати повторних запусків збігатимуться з попередніми [27].

Четвертий рівень є найвищим рівнем повторного використання і передбачає повторне застосування не лише вхідних і вихідних даних, а й усієї траєкторії виконання тесту. Якщо логіка проходження тесту не зазнала змін, повторний запуск такого тесту є недоцільним, оскільки він не дасть нової інформації про стан системи [11]. Загальна ідея впровадження автоматизованого тестування полягає в тому, що з кожним завершеним модулем кількість перевірюваних сценаріїв поступово зростає.

Таким чином, аналіз методів автоматизованого тестування хмарного програмного продукту, реалізованого на основі мікросервісної архітектури, дозволяє зробити висновок, що вибір конкретного підходу визначається особливостями системи, рівнем автоматизації та критеріями ефективності її функціонування. Планування тестування доцільно розглядати як задачу оптимізації витрат під час розробки програмного забезпечення.

Варто також враховувати, що результати попередніх тестових запусків можуть впливати на виконання нових тестів, особливо в межах інтеграційного тестування, де сценарії мають підвищений рівень взаємозалежності.

2.3 Автоматизація тестування хмарних рішень з мікросервісною архітектурою

Мікросервісний підхід розглядають як одну з найбільш придатних архітектурних альтернатив монолітним системам, які зазвичай характеризуються обмеженою гнучкістю еволюції, складністю масштабування

та високою «вартістю змін». У моноліті навіть локальна модифікація окремої процедури або підсистеми часто потребує повторного збирання, розгортання й перевірки всієї програми, що збільшує ризики регресій та уповільнює доставку оновлень.

Під час тестування мікросервісного програмного продукту доцільно розглядати систему як сукупність автономних сервісів, які взаємодіють через узгоджені інтерфейси та протоколи. Така декомпозиція спрощує розробку й валідацію окремих компонентів, пришвидшує випуск змін та створює передумови для гнучкого масштабування, що особливо важливо для хмарних розгортань на різних платформах і в неоднорідних середовищах виконання.

До ключових особливостей автоматизованого тестування мікросервісних хмарних рішень належать:

– потреба в автоматизованій інфраструктурі, здатній керувати життєвим циклом сервісів (розгортання, реєстрація/іменування, маршрутизація, масштабування, моніторинг стану) з урахуванням поточного навантаження;

– необхідність безперервної інтеграції нових або змінених мікросервісів у вже функціонуючу систему, що зумовлює регулярну перевірку як ізольованої логіки сервісів, так і коректності їх взаємодії в складі загального рішення.

У межах життєвого циклу мікросервісів акцент доцільно робити на трьох взаємодоповнювальних напрямах: компонентному тестуванні, тестуванні контейнера (або артефакту розгортання) та інтеграційному тестуванні. Така послідовність дозволяє знизити складність пошуку дефектів: спочатку виявляються помилки в межах сервісу, далі – проблеми середовища виконання, і лише після цього – помилки міжсервісної взаємодії. Весь процес розробки та тестування мікросервісів представлений на рисунку 2.6.

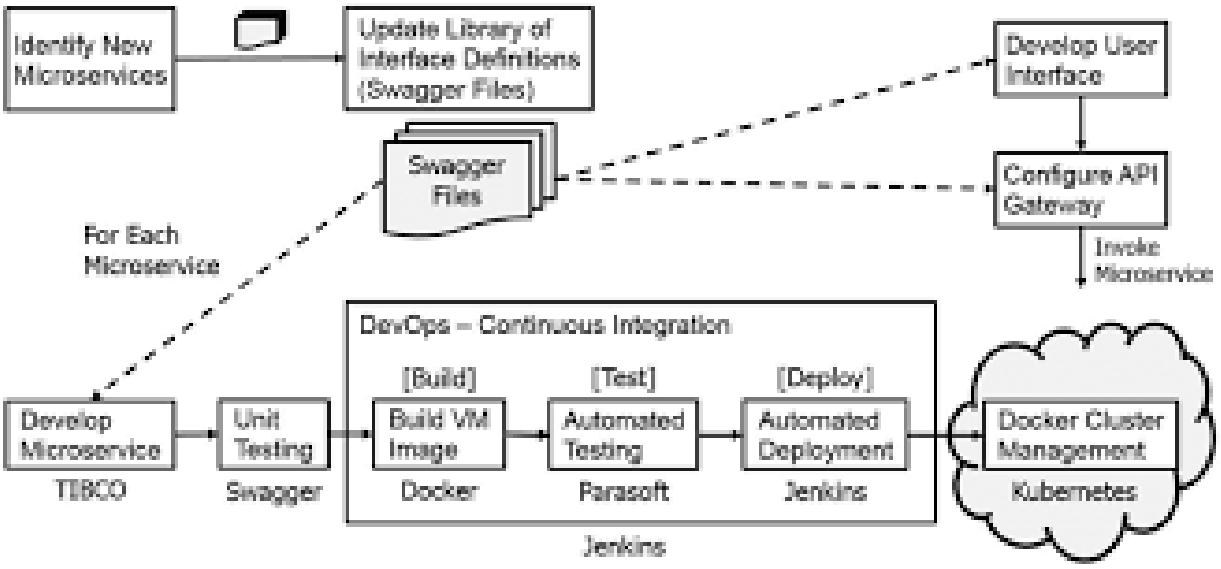

Рисунок 2.6 – Розробка та тестування мікросервісів

Компонентне тестування передбачає перевірку кожного мікросервісу ізольовано від інших сервісів (рис. 2.7). Фактично воно спрямоване на самоперевірку інтерфейсу сервісу та його внутрішньої логіки в межах «базового рівня» (власного контексту виконання). У практиці автоматизації компонентне тестування доцільно деталізувати за такими підвидами:

– функціональне компонентне тестування – валідує поведінку сервісу щодо визначених функціональних вимог (контракти API, правила обробки запитів, коректність відповідей);

– навантажувальне компонентне тестування – оцінює продуктивність та межі стійкості окремого сервісу за заданих профілів навантаження;

– компонентне тестування безпеки – перевіряє ізольованість, коректність контролю доступу, стійкість до типових класів зловживань на рівні конкретного сервісу.

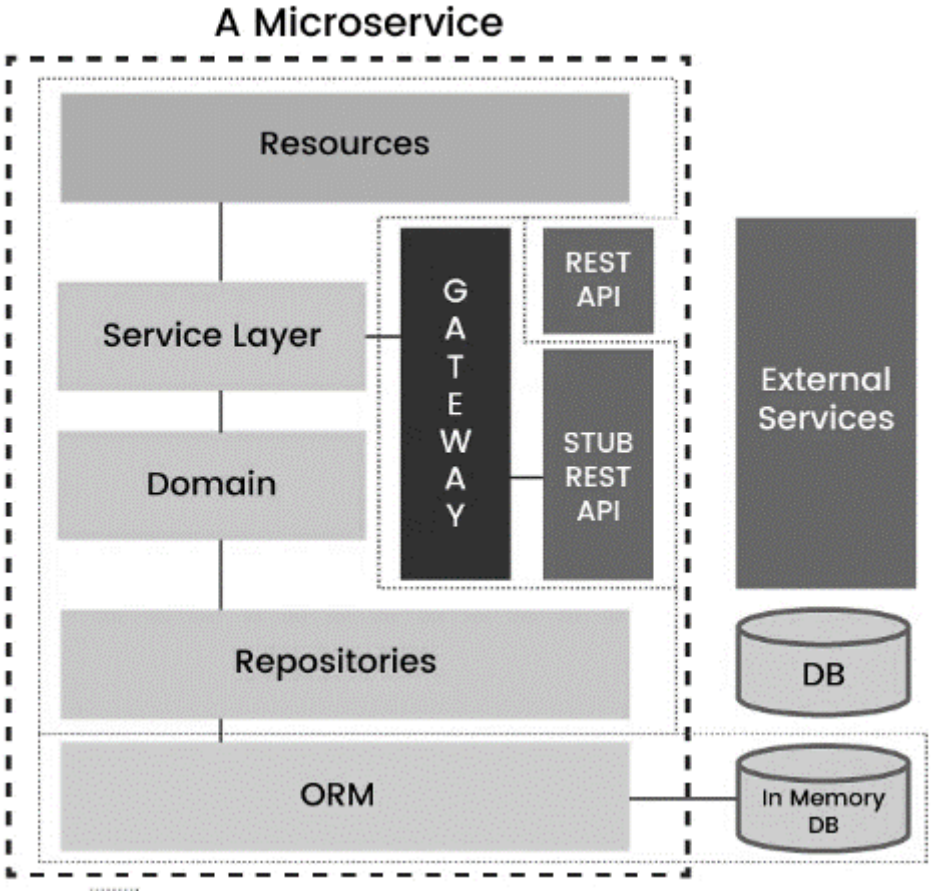

Рисунок 2.7 – Компонентне тестування мікросервісних систем

Тестування контейнера з мікросервісом (container-level testing) орієнтоване на перевірку сервісу у вигляді розгорнутого артефакту: у контейнері або іншому середовищі виконання, максимально наближеному до продуктивного. У цьому сценарії кожен мікросервіс викликається незалежно, а оцінюються його відповіді та побічні ефекти відповідно до заданих вхідних даних. Критичною вимогою є мінімізація небажаної взаємодії з іншими сервісами під час такої перевірки: залежності мають бути «під'єднані» так, щоб забезпечити відтворюваність, але не спричиняти непередбачуваної поведінки через зовнішні компоненти. Важливо також контролювати стабільність контрактів: споживачі повинні отримувати узгоджений результат навіть у випадку внутрішніх змін реалізації сервісу, за умови що зовнішній контракт не змінювався. Окрему увагу слід приділяти тому, що в хмарному середовищі сервіс може мати кілька екземплярів часу виконання, отже під час тестування потрібно враховувати налаштування, розгортання, масштабування та

спостереження за кожним екземпляром [10].

Інтеграційне тестування застосовують для систем, що складаються щонайменше з двох модулів, і воно спрямоване на виявлення помилок в реалізації або інтерпретації інтерфейсів взаємодії між компонентами (рис. 2.8). Для мікросервісних продуктів інтеграційний рівень зазвичай охоплює:

– функціональне інтеграційне тестування – перевіряє коректність міжсервісних обмінів (протоколи, формати повідомлень, узгодженість контрактів), а також коректність взаємного використання спільних ресурсів і уникнення взаємних блокувань;

– інтеграційне тестування продуктивності/масштабування – оцінює поведінку системи під час автоматичного розгортання, горизонтального/вертикального масштабування, а також за сценаріїв пікових навантажень;

– інтеграційне тестування безпеки – аналізує захищеність комунікацій між сервісами, контроль доступу між доменами, а також ризики перехоплення, підміни чи несанкціонованого доступу до повідомлень ззовні або зсередини системи.

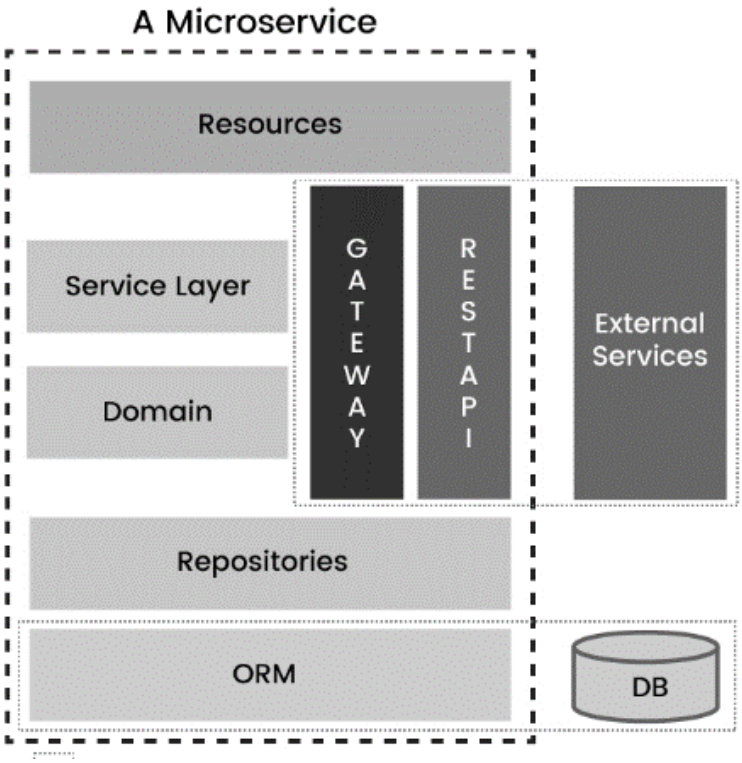

Рисунок 2.8 – Інтеграційне тестування

У межах інтеграційного тестування використовують різні сценарії «збирання» системи. Поширеними є [14]:

– одночасна (монолітна) інтеграція всіх сервісів у єдине тестове середовище з подальшою перевіркою цілісної системи; за відсутності окремих модулів або недоступності залежностей доцільно застосовувати тестові дублери;

– інкрементальна інтеграція, коли система нарощується поетапно: спочатку перевіряють взаємодію двох або логічно пов’язаних сервісів, а потім поступово під’єднують інші до моменту повної інтеграції, після чого виконують комплексне тестування.

Для інкрементальної інтеграції зазвичай застосовують дві базові стратегії:

– висхідну («знизу догори»), коли спершу перевіряють нижчі за ієрархією модулі, поступово підключаючи вищі компоненти; відсутні залежності часто замінюють заглушками/дублерами;

– низхідну («згори донизу»), коли інтеграція починається з головного (керувального) модуля, а підлеглі компоненти під’єднуються послідовно, також із можливим застосуванням заглушок для непідключених елементів.

У практиці часто використовують змішану інтеграцію, коли окремі модулі спочатку агрегуються у функціональні блоки (елементи «низового» рівня), а далі ці блоки приєднуються до керувальних компонентів (елементи «верхового» рівня). Така комбінація дозволяє збалансувати швидкість збирання та керованість діагностики дефектів. Окремо виділяють підхід «Big Bang», за якого всі модулі інтегруються одномоментно; він може скорочувати календарний час інтеграції, однак підвищує ризик одночасного прояву великої кількості дефектів і, відповідно, ускладнює локалізацію причин помилок [8].

Отже, незалежно від обраного сценарію, для мікросервісного хмарного продукту необхідно поєднувати ізольовану перевірку сервісів із валідацією їхньої взаємодії в складі системи. При цьому тестові дублери залишаються одним із ключових інструментів забезпечення відтворюваності та керованості тестування, оскільки вони дозволяють тимчасово заміщати недоступні або ще

не реалізовані модулі та стабілізувати тестове середовище.

2.4 Методології створення ПЗ з інтегрованим процесом тестування

Сучасна індустрія програмного забезпечення використовує різні методологічні підходи до організації розробки та контролю якості. Серед найбільш відомих моделей традиційно виокремлюють каскадну (Waterfall) та гнучкі підходи (Agile і похідні). Відмінність між ними полягає передусім у способі планування робіт, управлінні змінами вимог і моменті, коли тестування стає «системною» активністю проєкту.

Каскадна модель (Waterfall) базується на суворій послідовності етапів, які виконуються за попередньо затвердженим планом. Перехід до наступної стадії відбувається лише після завершення попередньої, що обмежує можливість оперативного перегляду вимог і швидкого внесення змін. У сучасних проєктах цей підхід застосовують переважно тоді, коли функціональні та нефункціональні вимоги чітко визначені наперед і очікується їхня стабільність. За каскадної організації робіт інтенсивне тестування нерідко концентрується наприкінці циклу, через що дефекти, виявлені на пізніх стадіях, спричиняють повторні повернення до етапів розробки та повторні перевірки, збільшуючи витрати та ризики зриву термінів.

Гнучкі підходи (Agile) орієнтовані на ітеративний розвиток продукту: функціональність постачається невеликими інкрементами, а зворотний зв’язок від замовника інтегрується у планування наступних ітерацій. Така організація робіт створює передумови для частих оновлень і швидшого отримання бізнес-цінності, проте одночасно підвищує значущість автоматизації тестування, оскільки частота змін вимагає регулярної та відтворюваної регресійної перевірки.

З метою посилення принципу «якість як частина проєктування» сформувалися підходи, у яких тестування інтегрується безпосередньо в цикл розробки:

– TDD (Test Driven Development) – передбачає створення автоматизованих тестів до написання реалізації: спершу формулюється перевірка, потім пишеться мінімально достатній код для її проходження, після чого виконується рефакторинг з контролем незмінності зовнішньої поведінки;

– BDD (Behavior Driven Development) – фокусується на описі поведінки системи з позиції користувача та бізнес-вимог, використовуючи сценарії природною мовою або наближеними до неї формалізмами; це полегшує узгодження очікувань між стейкхолдерами та знижує ризики різночитань вимог;

– ATDD (Acceptance Test–Driven Development) – орієнтує команду на попереднє визначення приймальних критеріїв і сценаріїв, за якими результат буде визнано коректним; у межах підходу сценарії одночасно виконують роль тестів та артефактів для документування очікуваної поведінки системи.

У BDD та ATDD тестові дублери часто застосовують для моделювання великих зовнішніх залежностей (суміжних підсистем, сторонніх сервісів, тимчасово недоступних компонентів) або для відтворення спеціальних сценаріїв (негативні умови, деградації, пікові навантаження). Це дає змогу перевіряти ключові бізнес-сценарії без затримок, пов’язаних із готовністю всіх інтеграцій, і підвищує стабільність тестового середовища під час приймальних випробувань.

Узагальнюючи, ефективність автоматизованого тестування мікросервісного хмарного продукту значною мірою визначається тим, наскільки раціонально обрано критерії відбору й пріоритизації тестових випадків, а також наскільки процес тестування адаптований до динамічних властивостей хмари (адресація, масштабування, зміна кількості екземплярів сервісів). Оцінювання властивостей тестових сценаріїв (критичність, частота повторного виконання, вартість запуску, чутливість до змін коду) дає підстави для оптимізації використання ресурсів тестування та підвищення практичної результативності автоматизації. Додатково слід враховувати, що накопичені артефакти попередніх прогонів (логи, стани тестових середовищ, залишкові дані) можуть впливати на результати наступних запусків, особливо на рівні

інтеграційних перевірок, де взаємозалежність сценаріїв є суттєво вищою.

РОЗДІЛ 3
ПЕРЕВІРКА ЕФЕКТИВНОСТІ МЕТОДІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ІТ-ПРОДУКТУ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Формування інформаційних моделей для автоматизованого тестування хмарного програмного продукту з мікросервісною архітектурою

На етапі проєктування програмного продукту доцільно формувати інформаційні моделі, які відображають загальні принципи функціонування системи та логіку виконання її алгоритмів. Однією з поширених таких моделей є діаграма варіантів використання (Use Case Diagram), що подає узагальнену концептуальну структуру майбутнього рішення.

Основні цілі побудови діаграми варіантів використання полягають у тому, щоб:

– на ранніх стадіях визначити межі системи відповідно до її призначення та області застосування;

– зафіксувати функціональні вимоги до програмного продукту;

– сформувати початкову концептуальну модель для подальшого уточнення з урахуванням логічних і фізичних взаємозв’язків;

– підготувати елементи технічної документації, які спрощують предметну комунікацію між розробниками та замовником.

Побудована діаграма демонструє, які саме сценарії використання повинні бути доступними в системі, а також окреслює зовнішні взаємодії із програмним продуктом. Суб’єкти цієї взаємодії можуть бути представлені користувачами, зовнішніми сервісами або іншими системами. На діаграмі вони відображаються як актори – зовнішні сутності, що ініціюють певні дії та запускають відповідну поведінку розроблюваної системи.

Діаграма варіантів використання для тестування мікросервісу із залученням тестового дублера наведена на рис. 3.1. Для мікросервісу передбачено такі варіанти використання: формування запиту на читання даних, формування запиту на зміну (модифікацію) даних та обробка отриманих відповідей. Водночас тестовий дублер реалізує варіанти використання, пов’язані з обробкою запиту та генерацією відповіді на запит.

Рисунок 3.1 – Використання процесу тестування мікросервісу

На рисунку 3.1 зображено взаємодію мікросервісу з тестовим дублером (test double), який відтворює поведінку зовнішньої залежності під час перевірок.

Формування запиту на читання даних або запиту на зміну даних у будь-якому випадку передбачає виконання кроку «Обробка запиту». Аналогічно, «Формування відповіді на запит» логічно завершується етапом «Обробка відповіді». Оскільки ці дії є обов’язковими складовими відповідних сценаріїв, зв’язки між ними доцільно трактувати як відношення включення (<<include>>).

Під час побудови відповіді може виникати додаткова дія «Пошук шаблону відповіді». Водночас наявність шаблону не гарантується, тобто цей крок виконується лише за певної умови. Тому взаємозв’язок має характер розширення (<<extend>>), адже додаткова поведінка активується лише в

окремих випадках.

Діаграма класів є графічним поданням структури системи у вигляді множини сутностей (класів) та зв’язків між ними. Вона належить до статичних UML-моделей і відображає склад класів, їх атрибути, операції та типи відношень, не описуючи динаміку виконання або послідовність викликів у часі.

У межах моделі виокремлено три ключові класи: Requests, Responses і Templates. Кожен із них описується набором властивостей (атрибутів) та операцій (методів). Зв’язки між класами мають характер асоціації, оскільки жоден клас не є структурною частиною іншого. Клас Responses пов’язаний із Requests та Templates, зокрема через ідентифікатори запиту й відповіді (наприклад, request_id, response_id), що забезпечує трасування: яка відповідь сформована для конкретного запиту. З огляду на це, тестовий дублер при надходженні запиту виконує пошук відповідного шаблону; якщо потрібний шаблон відсутній, система має повернути відповідь з ознакою помилки.

Отже, аналіз діаграми варіантів використання та діаграми класів показує, що в процесі тестування мікросервіс взаємодіє із зовнішнім компонентом, поведінку якого моделює тестовий дублер. Для кожного вхідного запиту дублер підбирає шаблон відповіді, а за його відсутності ініціює повернення помилкового результату, що дозволяє перевіряти як штатні, так і виняткові сценарії.

3.2 Процедура тестування хмарного програмного продукту з використанням BDD та мікросервісів

3.2.1 Загальна концепція алгоритму тестування

Сучасні хмарні програмні продукти, побудовані на мікросервісній архітектурі, характеризуються високим рівнем динамічності, масштабованості та складною взаємодією між окремими сервісами. За таких умов ефективність тестування значною мірою залежить від здатності тестових сценаріїв відображати реальну поведінку системи з точки зору кінцевого користувача та

бізнес-процесів. Саме тому доцільним є застосування підходу Behavior-Driven Development (BDD), який орієнтує процес тестування на перевірку очікуваної поведінки системи.

Алгоритм тестування, розроблений на основі BDD, базується на формалізації вимог у вигляді поведінкових сценаріїв, які є зрозумілими як для технічних спеціалістів, так і для бізнес-стейкхолдерів. У межах мікросервісної архітектури такий алгоритм забезпечує узгоджене тестування окремих сервісів і їхньої взаємодії в хмарному середовищі з урахуванням масштабування, асинхронної комунікації та використання тестових дублерів.

3.2.2 Структура алгоритму BDD-тестування мікросервісів

Алгоритм автоматизованого тестування хмарного мікросервісного продукту на основі BDD включає послідовність логічно пов’язаних етапів, які охоплюють повний життєвий цикл тестування – від формування сценаріїв до аналізу результатів.

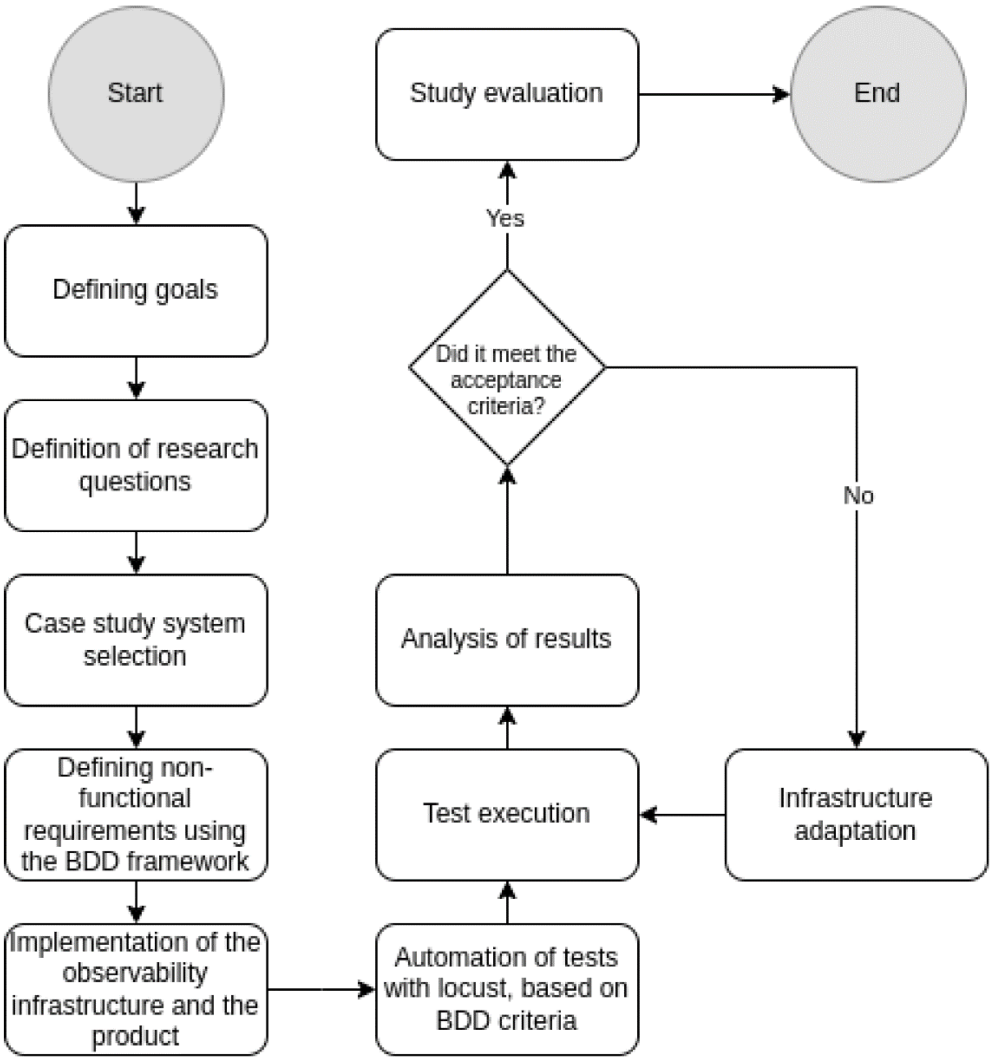

Рисунок 3.2 – Узагальнена схема алгоритму BDD-тестування мікросервісного хмарного продукту

Алгоритм складається з таких основних етапів:

1) Аналіз бізнес-вимог, на початковому етапі здійснюється аналіз функціональних і нефункціональних вимог до програмного продукту. Вимоги формулюються з точки зору очікуваної поведінки системи у реальних сценаріях використання.

2) Формування BDD-сценаріїв, на основі вимог створюються сценарії у форматі *Given–When–Then*, які описують початковий стан системи, подію та очікуваний результат. Такі сценарії є основою для подальшої автоматизації.

3) Побудова тестового середовища, налаштовується хмарне тестове середовище з урахуванням контейнеризації, оркестрації мікросервісів та підключення тестових дублерів для зовнішніх залежностей.

4) Автоматизація сценаріїв, BDD-сценарії трансформуються у виконувані автотести з використанням відповідних фреймворків. На цьому етапі забезпечується зв’язок між текстовими сценаріями та тестовим кодом.

5) Виконання тестів, автоматизовані тести запускаються в межах CI/CD-конвеєра з урахуванням різних конфігурацій розгортання та навантаження.

6) Аналіз результатів і зворотний зв’язок, результати тестування аналізуються, формується звіт про відповідність поведінки системи очікуванням. За необхідності сценарії коригуються або розширюються.

3.2.3 Формалізований алгоритм тестування

Для наочного подання логіки роботи алгоритму доцільно використовувати формалізований опис у вигляді таблиці 3.2.

Особливості застосування алгоритму в хмарному середовищі, яка реалізує алгоритму BDD-тестування в умовах хмарної інфраструктури має низку специфічних особливостей:

– необхідність адаптації тестів до динамічної адресації та масштабування мікросервісів;

– використання тестових дублерів для імітації нестабільних або недоступних зовнішніх сервісів;

– інтеграція тестування у безперервний цикл розгортання (CI/CD);

– забезпечення ізольованого виконання сценаріїв у контейнеризованому середовищі.

Таблиця 3.2 – Етапи алгоритму BDD-тестування мікросервісного хмарного продукту

Етап алгоритму	Вхідні дані	Результат
Аналіз вимог	Бізнес-вимоги, специфікація	Набір поведінкових вимог
Формування сценаріїв	Поведінкові вимоги	BDD-сценарії (Given–When–Then)
Підготовка середовища	Архітектура мікросервісів	Тестове хмарне середовище
Автоматизація	BDD-сценарії	Виконувані автотести
Запуск тестів	Автотести, CI/CD	Результати тестування
Аналіз	Логи, звіти	Оцінка якості та стабільності

Особливості застосування алгоритму в хмарному середовищі, яка реалізує алгоритму BDD-тестування в умовах хмарної інфраструктури має низку специфічних особливостей:

– необхідність адаптації тестів до динамічної адресації та масштабування мікросервісів;

– використання тестових дублерів для імітації нестабільних або недоступних зовнішніх сервісів;

– інтеграція тестування у безперервний цикл розгортання (CI/CD);

– забезпечення ізольованого виконання сценаріїв у контейнеризованому середовищі.

Використання тестових дублерів у BDD-алгоритмі тестування мікросервісів представлено на рисунку 3.3.

Сценарії взаємодії із системою, сформульовані за підходом BDD, є прозорими як для інженерної команди, так і для замовника або кінцевих користувачів, оскільки описують очікувану поведінку у прикладному (предметному) контексті.

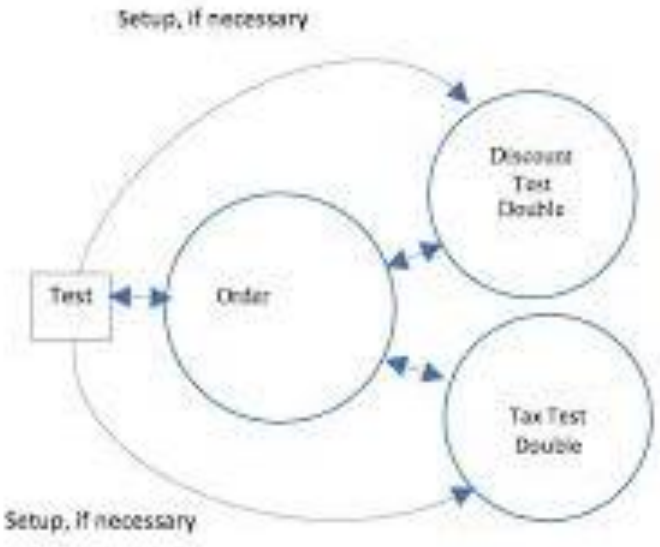

Рисунок 3.3 – Використання тестових дублерів у BDD-алгоритмі тестування мікросервісів

Інструментальні засоби BDD-тестування хмарного мікросервісного продукту

Специфікація, орієнтована на поведінку користувача, зазвичай фіксується в обмеженій формі природної мови з чітко визначеним синтаксисом. Тому програмні засоби, що підтримують BDD, зазвичай реалізують такі принципи роботи:

– Парсер розділяє текст специфікації на структурні елементи (наприклад, за ключовими словами мови Gherkin). У результаті формується послідовність речень, що відповідають BDD-формату.

– Кожне таке речення інтерпретується як окремий крок тестового сценарію.

– Механізм зіставлення шаблонів (часто на основі регулярних виразів) виділяє фрагменти речення, що містять параметри (вхідні дані), необхідні для виконання дії. Інша частина тексту використовується здебільшого для читабельності та однозначного опису події.

Після виділення параметри перетворюються у потрібні типи й передаються у відповідну операцію тестового коду. На цих підходах базується робота таких інструментів, як Cucumber, JBehave та JGiven.

Cucumber – кросплатформний інструмент автоматизації тестування, орієнтований на BDD. Він дозволяє описувати сценарії природною мовою у вигляді файлів формату .feature, де структура визначається мовою Gherkin. Типовий шаблон сценарію формулюється через ключові конструкції Given / When / Then, які задають передумови, дію та очікуваний результат.

JBehave – Java-орієнтований фреймворк, який підтримує автоматизацію приймальних перевірок і процес розробки у BDD-стилі. Бізнес-вимоги та критерії приймання задаються як користувацькі сценарії у текстовому вигляді. Зіставлення кроків сценарію з тестовою реалізацією виконується за допомогою Java-анотацій, що зв’язують фрази зі сценарію з методами тестового коду.

JGiven – інструмент BDD для Java, у якому сценарії описуються безпосередньо мовою Java через доменно-орієнтований API. Однією з ключових переваг є генерація звітів, придатних для читання фахівцями предметної області, що спрощує комунікацію між технічними та нетехнічними учасниками проєкту.

Отже, розглянуті засоби реалізують BDD-підхід і підтримують модель кроків Given / When / Then. Водночас для практичного застосування у мікросервісних хмарних системах часто обирають Cucumber-JVM завдяки широкій екосистемі, великій кількості прикладів та зрозумілій звітності. У межах Cucumber-JVM кожен крок сценарію пов’язується з конкретною інструкцією (методом) через шаблон зіставлення, найчастіше – регулярний вираз, а сценарії (Scenario) групуються у функціональні описи (Feature), що відображають вимоги до поведінки системи.

Запропонований алгоритм тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі та розробленого на основі BDD, забезпечує системний і масштабований підхід до перевірки коректності поведінки програмного продукту. Орієнтація на поведінкові сценарії дозволяє узгодити очікування бізнесу та реалізацію програмного забезпечення, а інтеграція автоматизованого тестування у хмарне середовище сприяє підвищенню надійності, стабільності та якості готового ІТ-рішення.

3.2 Переваги інтеграції тестових дублерів до складу програмного продукту

Практика автоматизованого тестування хмарних програмних рішень, реалізованих за мікросервісною архітектурою, демонструє, що застосування тестових дублерів (зокрема сервісів-заглушок) підвищує повноту та керованість перевірок. Завдяки дублерам стає можливим відтворювати необхідні сценарії взаємодії із зовнішніми підсистемами без залежності від їх фактичної доступності, стану або готовності. Водночас потреба у комплексному тестуванні виникає не лише під час розроблення, а й на етапах

впровадження, встановлення, приймання та подальшого супроводу рішення у середовищі замовника.

Включення тестових дублерів до постачання програмного продукту забезпечує низку практично значущих переваг.

1. Верифікація коректності інсталяції та конфігурації у середовищі замовника. Тестова інфраструктура, у якій виконувались розроблення та перевірки, часто істотно відрізняється від продуктивного середовища замовника за моделями серверного й мережевого обладнання, параметрами мережі та наявністю додаткових засобів захисту. Використання BDD-тестів, сформованих під час розроблення, у поєднанні з дублерами дозволяє підтвердити, що критичні підсистеми в межах зони відповідальності постачальника функціонують відповідно до вимог, а інтеграційні взаємодії здійснюються згідно з узгодженою специфікацією інтерфейсів.

2. Перевірка працездатності в ускладнених режимах експлуатації. У межах приймальних випробувань зазвичай необхідно підтвердити відповідність як функціональним, так і нефункціональним вимогам, включно з навантажувальними тестами та перевірками граничних станів. У реальних умовах взаємодія із зовнішніми сервісами може бути обмеженою (через неготовність підсистем, регламентні обмеження, відсутність доступів) або економічно невигідною (наприклад, при інтеграціях із платіжними сервісами чи каналами сповіщення). Тестові дублери дають змогу відтворити такі сценарії без залучення «дорогих» або недоступних зовнішніх компонентів, зберігаючи репрезентативність перевірки.

3. Створення доданої цінності для замовника. Замовники, як правило, зацікавлені у можливості самостійно виконувати регресійні та приймальні перевірки, проте стикаються з тими самими труднощами, що і команда тестування: залежності від сторонніх систем, дефіцит часу на підготовку стенда, потреба в автоматизації при частих релізах. Наявність готового комплексу сценаріїв і дублерів може бути використана як складова комерційної пропозиції, оскільки скорочує витрати замовника на побудову власного контуру перевірок, що є критично важливим у практиках Agile та

при регулярному оновленні продуктивних середовищ. Навіть за відсутності наміру використовувати підготовлені сценарії повністю, дублери можуть стати корисним компонентом внутрішнього процесу контролю якості замовника.

4. Спрощення робіт з впровадження та підтримки. Під час експлуатації можливі інциденти, пов’язані не лише з самим продуктом, а й з порушенням контрактів інтеграції з боку суміжних систем (наприклад, зміни формату повідомлень або некоректна реалізація API). Наявність попередньо сконфігурованих тестових дублерів дає змогу оперативно локалізувати джерело проблеми, швидше відтворити дефект, зменшити тривалість розслідування та прискорити виправлення. Це, у свою чергу, позитивно впливає на стабільність сервісу та рівень задоволеності користувачів.

Отже, інтеграція тестових дублерів до складу програмного продукту підвищує керованість тестування на всьому життєвому циклі рішення — від розроблення до підтримки у замовника. Водночас слід враховувати обмеження: створення, розгортання та супровід дублерів потребують додаткових трудовитрат і часу, що має бути враховано під час планування процесів розроблення та експлуатації.

3.4 Приклад BDD-сценарію (Gherkin) для мікросервісного продукту

У якості прикладу розглянемо типовий мікросервіс Auth Service**,** що видає токен доступу, та сервіс Profile Service, який повертає профіль користувача за токеном. Сценарій перевіряє поведінку системи для коректної авторизації та доступу до ресурсу (лістинг 3.1).

Лістинг 3.1 – BDD-сценарій у форматі Gherkin (feature-файл)

```
—
Feature: Доступ до профілю користувача через токен
  Для забезпечення безпечного доступу
  Як зареєстрований користувач
  Я хочу отримувати свій профіль після успішної авторизації

  Background:
    Given у системі існує користувач з логіном "student01" та паролем
"P@ssw0rd!"
    And сервіс авторизації доступний

  Scenario: Успішна авторизація і читання профілю
    When я надсилаю запит на авторизацію з логіном "student01" та
паролем "P@ssw0rd!"
    Then я отримую відповідь зі статусом 200
    And відповідь містить поле "access_token"

    When я надсилаю запит на отримання профілю з отриманим
"access_token"
    Then я отримую відповідь зі статусом 200
    And у відповіді є поле "username" зі значенням "student01"

  Scenario: Відмова при неправильному паролі
    When я надсилаю запит на авторизацію з логіном "student01" та
паролем "wrong"
    Then я отримую відповідь зі статусом 401
    And відповідь містить повідомлення "Invalid credentials"
```

—

Кінець Лістингу 3.1.

У реальній мікросервісній системі **Background** часто реалізують через *seed-дані* тестового середовища або через API/БД-скрипти перед запуском сценаріїв.

Інтеграція BDD-тестів у CI/CD забезпечує:

– автоматичний запуск сценаріїв на кожен push/merge request;

– раннє виявлення регресій (особливо критично для мікросервісів);

– формування артефактів: звітів, логів, скриншотів (за потреби).

В лістингу 3.2 та 3.3 наведено практичні варіанти: **GitLab CI** та **GitHub Actions**. Варіант А. GitLab CI: запуск BDD після збірки та розгортання тестового стенду.

Лістинг 3.2 – Приклад .gitlab-ci.yml (узагальнений шаблон)

```
__
stages:
  - build
  - test
  - report

variables:
  BDD_REPORT_DIR: "bdd-reports"

build_services:
  stage: build
  image: docker:27
  services:
    - docker:27-dind
  script:
    - docker compose -f docker-compose.test.yml build
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" || $CI_PIPELINE_SOURCE ==
"merge_request_event"

bdd_tests:
  stage: test
  image: node:20
  services:
    - name: docker:27-dind
      command: ["--tls=false"]
  before_script:
    - apt-get update && apt-get install -y docker-compose
  script:
    # 1) Підняти тестове середовище (мікросервіси + тестові дублери)
    - docker compose -f docker-compose.test.yml up -d
    # 2) Дочекатися готовності (healthcheck або простий sleep)
    - node ./scripts/wait-for-health.js
    # 3) Запустити BDD (наприклад, cucumber-js)
    - npm ci
    - npm run bdd -- --format json:$BDD_REPORT_DIR/report.json
  after_script:
    - docker compose -f docker-compose.test.yml down
  artifacts:
    when: always
    paths:
      - bdd-reports/
    expire_in: 7 days
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" || $CI_PIPELINE_SOURCE ==
"merge_request_event"
```

Кінець лістингу 3.2

Важливо в контексті алгоритму:

– CI-пайплайн піднімає ізольоване середовище (мікросервіси + дублери).

– BDD запускається як етап перевірки поведінки (Given–When–Then).

– Зберігаються артефакти звіту, що дозволяє порівнювати результати між збірками.

Лістинг 3.2– Приклад workflow .github/workflows/bdd.yml

—

```
name: BDD tests

on:
  push:
    branches: ["main", "develop"]
  pull_request:
    branches: ["main", "develop"]

jobs:
  bdd:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: "20"

      - name: Build and start test environment
        run: |
          docker compose -f docker-compose.test.yml up -d --build

      - name: Wait for services
        run: node ./scripts/wait-for-health.js

      - name: Install dependencies
        run: npm ci
      - name: Run BDD scenarios
        run: npm run bdd -- --format json:bdd-reports/report.json

      - name: Upload BDD report
```

```
      if: always()
      uses: actions/upload-artifact@v4
      with:
        name: bdd-reports
        path: bdd-reports/

    - name: Shutdown environment
      if: always()
      run: docker compose -f docker-compose.test.yml down
```

Кінець лістингу – 3.3

Переваги GitHub Actions для BDD:

– сценарії автоматично виконуються при pull_request;

– артефакти зі звітом доступні у вкладці *Artifacts*;

– однаковий підхід для різних середовищ завдяки Docker Compose.

Вбудування BDD у CI/CD логічно відповідає етапам алгоритму:

– формування BDD-сценаріїв → feature-файли (Gherkin);

– підготовка тестового середовища → docker-compose.test.yml (мікросервіси + дублери);

– запуск тестів → job bdd_tests / Run BDD scenarios;

– аналіз результатів → JSON/HTML звіти, збережені як artifacts.

ВИСНОВКИ

У межах виконання кваліфікаційної роботи проаналізовано базові підходи та характерні риси автоматизованого тестування програмного продукту, реалізованого за мікросервісною архітектурою. Встановлено, що застосування хмарної інфраструктури забезпечує оперативний і зручний доступ до обчислювальних ресурсів, що дає змогу зменшити організаційні витрати та ефективніше організувати розроблення і перевірку працездатності програмного рішення.

На основі опрацювання теоретичних засад і наявних методів автоматизованого тестування визначено їх ключові переваги та обмеження, а також окреслено специфіку тестування хмарного програмного продукту, побудованого на мікросервісах. Систематизовано критерії оцінювання ефективності автоматизації та підходи до формування тестових наборів.

У результаті аналізу методів автоматизованого тестування виокремлено найбільш придатні методики та інструментальні засоби, що забезпечують повнішу і швидшу верифікацію хмарного програмного продукту.

Доведено, що використання тестових дублерів, які відтворюють виклики взаємодіючих підсистем, підвищує комплексність і оперативність перевірок, дозволяючи ізолювати тестовані компоненти від зовнішніх залежностей.

За підсумками розгляду сучасних підходів до розроблення із вбудованим процесом тестування визначено методологію, яка дає змогу найшвидше сформувати якісний план тестування, зрозумілий як замовнику, так і команді розробників та тестувальників.

Розглянуто інструменти, що забезпечують автоматизацію тестування хмарного програмного продукту, та проаналізовано їх сильні й слабкі сторони з позиції практичної зручності застосування. На цій основі сформовано сценарій автоматизованого тестування мікросервісного рішення із використанням тестових дублерів.

У межах апробації обраних методів, сценарію та інструментарію підготовлено набір тест-кейсів для перевірки змодельованого хмарного програмного продукту на мікросервісній архітектурі, а також описано послідовність виконання та отримані результати тестування.

На підставі оцінювання практики застосування тестових дублерів визначено переваги підходу, виконано оцінку трудовитрат на розгортання й налаштування дублера в інфраструктурі замовника та сформульовано рекомендації щодо підвищення якості, надійності й швидкості доставки оновлень.

Таким чином, запропонований сценарій (алгоритм) автоматизованого тестування хмарного програмного продукту, побудованого на мікросервісній архітектурі, сприяє зниженню сукупних витрат, підвищенню якості програмного забезпечення та скороченню термінів його виходу на ринок.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dragoni N., Lanese I., Larsen S. T., et al. Microservices: Yesterday, Today, and Tomorrow // Present and Ulterior Software Engineering. – Cham : Springer, 2021. – P. 195–216.

2. Pahl C., Jamshidi P. Microservices: A Systematic Mapping Study // Proceedings of the 6th International Conference on Cloud Computing and Services Science. – 2021. – P. 137–146

3. Soldani J., Tamburri D. A., Van Den Heuvel W.-J. The pains and gains of microservices: A Systematic Grey Literature Review // Journal of Systems and Software. – 2021. – Vol. 146. – P. 215–232.

4. Crispen L., Gregory J. Agile Testing: A Practical Guide for Testers and Agile Teams. – 2nd ed. – Boston : Addison-Wesley, 2021. – 384 p.

5. Fowler M. Microservices Testing Strategy. URL: https://martinfowler.com/articles/microservice-testing/ (дата звернення: 15.01.2023).

6. Wynne M., Hellesøy A. The Cucumber Book: Behaviour-Driven Development for Testers and Developers. – Updated edition. – Dallas : Pragmatic Bookshelf, 2022. – 312 p.

7. Newman S. Building Microservices: Designing Fine-Grained Systems. – 2nd ed. – Sebastopol : O'Reilly Media, 2021. – 582 p.

8. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. – Updated edition. – Boston : Addison-Wesley, 2021. – 512 p. Beck K. Test Driven Development: By Example. Addison–Wesley Professional,2021. 242 p.

9. Що потрібно знати про онлайн-освіту в Україні URL: https://techno.nv.ua/ukr/technoblogs/onlayn-osvita-v-ukrajini-50128010.html (дата звернення: 15.02.2023 р.)

10. Платформи для організації дистанційного навчання. URL: https://buki.com.ua/news/5-platform-dlya-orhanizatsiyi-dystantsiynoho-navchannya/ (дата звернення: 18.02.2023 р.)

11. ТОП-3 системи для онлайн навчання. URL: https://osvita.ua/news/ 71748/ (дата звернення: 22.02.2023 р.)

12. Додатки і платформи для змішаного і дистанційного навчання. URL: https://osvitanova.com.ua/posts/4264-dodatky-i-platformy-dlia-zmishanoho-ta-dystantsiinoho-navchannia (дата звернення: 19.03.2023 р.)

13. Освітні платформи URL: https://adl.nuou.org.ua/platforms (дата звернення: 18.03.2023 р.)

14. EdEra Book URL: https://www.ed-era.com/books/ (дата звернення: 15.03.2023 р.)

15. Інформація про проєкт ВУМ URL: https://vumonline.ua/about-project/ (дата звернення: 15.03.2023 р.)

16. Сервіси та платформи для дистанційного навчання. URL: https://tech.24tv.ua/navchannya-vdoma-spisok-program-platform-dlya-distantsiynogo-navchannya_n1416110 (дата звернення: 18.03.2023 р.)

17. Moodle в Україні. Що таке Moodle URL: https://moodle.org/mod/page/view.php?id=8174 (дата звернення: 18.03.2023 р.)

18. Що таке CMS Wordpress? Система управління контентом Вашого сайту URL: https://websoft.biz.ua/article_chto_takoe_wordpress_chto_za_systema. html (дата звернення: 18.03.2023 р.)

19. Drupal URL: https://drupal.ua/ (дата звернення: 18.03.2023 р.)

20. XMIND URL: https://xmind.app/ (дата звернення: 25.03.2023 р.)