Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
https://doi.org/10.36930/40340612
Article received 29.05.2024
Article accepted 05.09.2024
UDC 004.052
Correspondence author: V. Ya. Lakhai vladyslav.y.lakhai@lpnu.ua
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)

В. Я. Лахай, М. М. Сенів
Національний університет "Львівська політехніка", м. Львів, Україна

АНАЛІЗ НАЯВНИХ МЕТОДІВ І ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ ЗРУЧНОСТІ СУПРОВОДУ МІКРОСЕРВІСНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Проаналізовано літературні джерела, що досліджують методи та засоби забезпечення зручності супроводу мікросервісного програмного забезпечення (ПЗ). Виявлено, що наявні підходи, хоч і корисні, не повністю вирішують проблеми, пов'язані зі складністю мікросервісної архітектури. Особливу увагу приділено складності управління великою кількістю незалежних компонент, які потребують узгодженості для забезпечення надійності та ефективності системи. Дослідження виявило, що забезпечення зручності супроводу мікросервісів є складним завданням через їхню розподілену природу, високий ступінь автономії компонент та потребу підтримання узгодженості між ними. Зручність супроводу мікросервісів розглянуто як комплексну характеристику, що містить модульність, аналізованість (англ. *Analyzability*), можливість повторного використання, змінюваність та тестованість (англ. *Testability*). Визначено ключові проблеми, що ускладнюють супровід мікросервісів, та запропоновано шляхи їх вирішення. Запропоновано розширити підхід чистої архітектури (англ. *Clean Architecture*) шляхом додавання шару доменних сервісів, що сприятиме кращій організації бізнес-логіки та зниженню складності в разі внесення змін. Удосконалення структури компонент містить розроблення нового методу управління залежностями, який поєднує ін'єкцію залежностей з локатором служб, що забезпечить гнучкіше та контрольованіше середовище для управління залежностями. Розроблено спеціалізований метод тестування ініціалізації залежностей, що дасть змогу зменшити кількість помилок на ранніх етапах розроблення та підвищити стабільність системи. Окрім цього, розглянуто подальший розвиток використання моків (англ. *Mocks*) для спрощення процесу тестування, що дасть змогу знизити залежність від зовнішніх сервісів під час перевірки компонент. Результати дослідження засвідчують важливість подальшого вивчення та вдосконалення методів забезпечення зручності супроводу мікросервісів, що має вирішальне значення для підтримання високої якості програмного забезпечення та зниження витрат на його підтримку. На підставі запропонованих шляхів вирішення можна розробити нові та вдосконалити наявні методи, що дасть змогу підвищити показники зручності супроводу мікросервісного ПЗ, зменшити ризики, пов'язані з інтеграцією нових компонент, та забезпечити ефективніше управління складними мікросервісними системами.

Ключові слова: зручність супроводу; мікросервіси; Clean Architecture; тестування; ін'єкція залежностей.

Вступ / Introduction

У сучасному інформаційному світі, де обсяги даних стрімко збільшуються, організації постійно шукають шляхи оптимізації обчислювальних потужностей та підвищення ефективності програмного забезпечення (ПЗ). Традиційні підходи, такі як власні центри оброблення даних, стикаються з проблемами масштабованості та гнучкості. У відповідь на ці виклики з'явилися хмарні технології та нові архітектурні парадигми, такі як мікросервісна архітектура. Хоча ці інновації значно розширили можливості розроблення та розгортання ПЗ, вони також внесли свої складнощі. Однією з ключових проблем є зниження зручності супроводу (maintainability) – критичної характеристики якості ПЗ, що визначає легкість його модифікації, виправлення помилок та адаптації до нових вимог [16]. Зниження зручності супроводу мікросервісного ПЗ пов'язане з розподіленістю мікросервісного ПЗ та складністю цієї технологій.

Наявні методології та інструменти розроблення ПЗ, такі як Clean Architecture, розроблення через тестування (англ. *Test-Driven Development*, TDD), предметно-орієнтоване проєктування (англ. *Domain Driven Design*, DDD), розроблення на підставі поведінки (англ. *Behavior-Driven Development*, BDD), розроблення на підставі контрактів (англ. *Designe by contract*, DbC), інверсія управління (англ. *Inversion of Control*, IoC) і локатор служб (англ. *Service Locator*), хоча й ефективні у традиційних умовах, не завжди повністю відповідають вимогам нових архітектур. Тому виникає нагальна потреба у розробленні вдосконалених підходів, спеціально адаптованих для мікросервісної архітектури, з метою підвищення зручності супроводу мікросервісного ПЗ. У цьому дослідженні проаналізовано наявні методи і засоби забезпечення зручності супроводу мікросервісного ПЗ, а також визначено проблеми забезпечення зручності супроводу, які не повністю вирішуються наявними методами та засобами.

Інформація про авторів:
Лахай Владислав Ярославович, аспірант, кафедра програмного забезпечення.
Email: vlad.luckhi@gmail.com; https://orcid.org/0000-0003-1346-5731
Сенів Максим Михайлович, канд. техн. наук, доцент, кафедра програмного забезпечення.
Email: maksym.m.seniv@lpnu.ua; https://orcid.org/0000-0003-1044-4628

Цитування за ДСТУ: Лахай В. Я., Сенів М. М. Аналіз наявних методів і засобів забезпечення зручності супроводу мікросервісного програмного забезпечення. Науковий вісник НЛТУ України. 2024, т. 34, № 6. С. 87–92.
Citation APA: Lakhai, V. Ya., & Seniv, M. M. (2024). Analysis of existing methods and tools for ensuring the maintainability of microservice software. *Scientific Bulletin of UNFU*, 34(6), 87–92. https://doi.org/10.36930/40340612

Об'єкт дослідження – забезпечення зручності супроводу мікросервісного програмного забезпечення.

Предмет дослідження – методи та засоби забезпечення зручності супроводу мікросервісного програмного забезпечення, що дасть змогу підвищити якість програмного забезпечення внаслідок зниження ризику виникнення помилок, покращення надійності та масштабованості системи, а також забезпечення більш швидкого реагування на зміни бізнес-вимог.

Мета роботи – проаналізувати наявні публікації, які стосуються методів і засобів підвищення зручності супроводу мікросервісного ПЗ, визначити їх переваги та недоліки, виявити проблеми, які не повністю вирішуються наявними методами, та запропонувати шляхи вирішення, що дасть змогу розробити нові та вдосконалити наявні методи для підвищення показників зручності супроводу.

Для досягнення зазначеної мети визначено такі основні *завдання дослідження*:

1. Описати та охарактеризувати особливості забезпечення зручності супроводу в мікросервісному програмному забезпеченні, зокрема специфічні виклики та проблеми, пов'язані з розподіленою архітектурою та незалежним розгортанням компонент.
2. Проаналізувати наявні методи та засоби забезпечення зручності супроводу, визначити їх переваги та недоліки, що дасть змогу визначити проблеми забезпечення зручності супроводу мікросервісів, які не вирішуються повністю наявними методами.
3. Запропонувати шляхи вирішення визначених проблем, а саме запропонувати розроблення нових і вдосконалення наявних методів, які дали б змогу підвищити показники зручності супроводу мікросервісів.

Аналіз останніх досліджень та публікацій. Зручність супроводу ПЗ є критично важливою особливістю його життєвого циклу, особливо в умовах мікросервісної архітектури. Останні дослідження та публікації підтверджують, що зручність супроводу мікросервісів є багатоаспектною проблемою, яка залежить від численних чинників, зокрема технічних, організаційних та специфічних для мікросервісів особливостей.

На зручність супроводу ПЗ впливають такі чинники, як:

1) Технічна якість коду: структурованість коду [1], відсутність технічного боргу [25], використання статичного аналізу коду для виявлення потенційних проблем [12], а також використання найкращих практик розроблення та тестування [28] є ключовими для забезпечення легкості внесення змін та виправлення помилок.
2) Розмір та складність проєкту: більші та складніші проєкти потребують більше зусиль для супроводу [24].
3) Архітектурні рішення: вибір правильної архітектури, такої як Clean Architecture [21], має вирішальний вплив на зручність супроводу системи [14].
4) Застосування інтелектуального аналізу: використання методів інтелектуального аналізу даних [7] та тексту [18] може допомогти виявити та виправити дефекти, спрощуючи процес супроводу та підвищуючи зручність внесення змін.

Дослідження, зосереджені на розробленні мікросервісів [2, 10, 36], засвідчують як їх потенціал для вдосконалення зручності супроводу мікросервісів, так і пов'язані з ними виклики:

1) Потенційні переваги: модульність, незалежне розгортання та масштабування сервісів сприяють удосконаленню зручності супроводу мікросервісів [36].
2) Виклики: Управління складністю розподіленої системи, забезпечення надійної взаємодії між сервісами та моніторинг їх роботи є основними проблемами, що впливають на зручність супроводу мікросервісів [2, 10].

В індустрії вже існують різноманітні методи та інструменти, що можуть бути застосовані для підвищення зручності супроводу мікросервісів:

1) Архітектурні підходи: Clean Architecture [21, 22], DDD [17, 30, 37] та ін'єкція залежностей (англ. *Dependency Injection*, DI) [32, 37] сприяють створенню гнучких та легко супроводжуваних систем.
2) Практики розроблення та тестування: (TDD) [6, 29], рефакторинг [34], безперервна інтеграція та доставка CI/CD (англ. *Continuous Integration/Continuous Delivery*) [33] покращують якість коду та спрощують його модифікацію, підвищуючи цим самим зручність супроводу мікросервісів.
3) Інші особливості: написання читабельного коду [8] та ефективне управління технічним боргом [27] також відіграють важливу роль у забезпеченні зручності супроводу мікросервісів.

Деякі практики розроблення та проєктування ПЗ особливо корисні для забезпечення зручності супроводу мікросервісів:

1) Розроблення на підставі поведінки (BDD): BDD допомагає забезпечити відповідність коду очікуванням користувачів і зацікавлених сторін, що полегшує внесення змін та оновлень у майбутньому [9].
2) Розроблення на підставі контрактів (DbC): DbC визначає чіткі контракти між компонентами системи, що допомагає запобігти помилкам та забезпечити коректну роботу системи навіть після внесення змін [23].
3) Інверсія управління (IoC): IoC та її реалізація у вигляді впровадження залежностей (DI) роблять систему більш гнучкою та модульною, спрощуючи заміну та оновлення окремих компонент [26, 35].
4) Використання моків: моки дають можливість ізолювати компоненти системи під час тестування, що полегшує виявлення та виправлення помилок, а також сприяє розробленню більш модульного та супроводжуваного коду [38].
5) Service Locator: Хоча Service Locator може бути корисним у деяких випадках, його надмірне використання може призвести до проблем зі зручністю супроводу. Важливо ретельно зважувати переваги та недоліки цього патерну та використовувати його з обережністю [20].

Проведений аналіз останніх досліджень та публікацій підтверджує, що зручність супроводу мікросервісного ПЗ є актуальною та складною проблемою. Для її вирішення необхідно застосовувати комплексний підхід, що містить використання відповідних архітектурних підходів, загальних практик розроблення та тестування, а також специфічних практик та інструментів, призначених для роботи з мікросервісами.

Результати дослідження та їх обговорення / Research results and their discussion

Зручність супроводу мікросервісного ПЗ є критично важливою, оскільки вона безпосередньо впливає на здатність виправляти помилки та впровадження нового функціоналу, що, водночас, визначає конкурентоспроможність та успіх продажу продукту.

Забезпечення високого рівня зручності супроводу в мікросервісній архітектурі є складним завданням через розподілену природу системи, велику кількість незалежних компонент та їхніх взаємодій, а також потребу забезпечення узгодженості та синхронізації між ними.

Наявні підходи та методи забезпечення зручності супроводу мікросервісного ПЗ можна класифікувати за етапами життєвого циклу ПЗ.

Таблиця. Переваги та недоліки наявних підходів до забезпечення зручності супроводу мікросервісів / Advantages and disadvantages of existing maintainability approaches

Публікація	Назва підходу	Переваги	Недоліки
1	2	3	4
[21, 22, 31]	Чиста архітектура (Clean Architecture)	Незалежність від фреймворків: код не прив'язаний до конкретних бібліотек чи фреймворків, що полегшує їх заміну або оновлення. Тестованість: архітектура сприяє створенню легко тестованого коду, що підвищує його якість та надійність. Гнучкість: легше вносити зміни до бізнес-логіки, не зачіпаючи інші шари системи. Довговічність: система стає більш стійкою до змін у вимогах та технологіях.	Складність: вимагає глибшого розуміння принципів проєктування та може бути складнішою для реалізації, ніж традиційні архітектури. Більше коду: іноді може знадобитися написати більше коду для дотримання принципів чистої архітектури.
[4, 6, 29]	Кероване тестами розроблення (Test-Driven Development, TDD)	Якість коду: TDD сприяє створенню більш чистого, модульного та надійного коду. Менше помилок: тести допомагають виявляти помилки на ранніх етапах розроблення. Впевненість у змінах: легше вносити зміни до коду, знаючи, що тести підтвердять їх коректність. Жива документація: тести слугують як документація до коду, показуючи, як він повинен працювати.	Початкові витрати часу: підготовка тестів потребує додаткового часу на початку розроблення ПЗ. Складність підтримки тестів: тести також потрібно підтримувати та оновлювати разом зі змінами в коді.
[17, 30, 37]	Предметно-орієнтоване проєктування (Domain-Driven Design, DDD)	Краще розуміння предметної області: DDD допомагає розробникам глибше зрозуміти предметну область, що сприяє створенню більш ефективних та адекватних рішень. Спільна мова: DDD встановлює спільну мову між розробниками та експертами предметної області, що покращує комунікацію та співпрацю. Гнучкість та адаптивність: DDD робить систему більш гнучкою та адаптивною до змін у вимогах.	Високий поріг входу: DDD вимагає певного досвіду та знань для ефективного застосування. Не підходить для всіх проєктів: DDD може бути надмірним для простих проєктів з обмеженою предметною областю.
[8, 11]	Ін'єкція залежностей (Dependency Injection, DI)	Зменшення зв'язності: DI зменшує зв'язність між компонентами, роблячи систему більш модульною та гнучкою. Удосконалення тестованості: DI спрощує написання юніт-тестів, оскільки залежності можна легко замінити на моки або стаби. Підвищення повторного використання коду: компоненти з DI легше використовувати повторно в інших контекстах.	Складність: DI може зробити код більш складним для розуміння, особливо для початківців. Налаштування: DI вимагає додаткового налаштування для визначення залежностей між компонентами.
[23]	Кероване поведінкою розроблення (Behavior-Driven Development, BDD)	Краща комунікація: BDD сприяє кращій комунікації між зацікавленими сторонами (розробниками, тестувальниками, бізнес-аналітиками тощо) завдяки використанню спільної мови для опису поведінки системи. Фокус на бізнес-цінності: BDD допомагає зосередитися на розробленні функціональності, яка має найбільшу цінність для бізнесу. Автоматизовані тести: BDD сценарії можуть бути використані для створення автоматизованих тестів, що забезпечує швидкий зворотний зв'язок та підвищує надійність системи.	Вимагає додаткових зусиль: Написання BDD сценаріїв вимагає додаткового часу та зусиль. Складність підтримки: BDD сценарії також потрібно підтримувати та оновлювати разом зі змінами у вимогах.
[19]	Локатор служб (Service Locator)	Простота використання: Service Locator – шаблон проєктування, який інкапсулює логіку отримання сервісів. У деяких випадках цей шаблон є антипатерном. Можна простим способом отримати залежності.	Приховування залежностей: Service Locator приховує залежності між компонентами, що ускладнює розуміння коду та його тестування. Ускладнення рефакторингу: зміни в залежностях можуть вимагати змін у багатьох місцях коду.
[20]	Моки (Mocks)	Ізоляція компонент: моки дають можливість ізолювати компоненти системи під час тестування, що спрощує написання та підтримку тестів. Тестування в ізоляції: можна тестувати компонент незалежно від його залежностей, що дає змогу виявляти помилки на ранніх етапах розроблення.	Складність: написання та підтримка моків може бути складним і трудомістким процесом. Ризик перетестування: можна випадково протестувати реалізацію моку, а не реальну поведінку компоненти.
[38]	Статичний аналіз (Static analysis)	Раннє виявлення помилок: статичний аналіз може виявити потенційні помилки та вразливості ще до запуску коду. Удосконалення якості коду: статичний аналіз може допомогти виявити порушення стандартів кодування та запропонувати рекомендації щодо їх виправлення. Автоматизація: статичний аналіз може бути легко інтегрований у процес безперервної інтеграції, що забезпечує автоматичну перевірку коду.	Хибнопозитивні результати: статичний аналіз може видавати хибнопозитивні результати, тобто повідомляти про помилки там, де їх насправді немає. Обмежені можливості: статичний аналіз не може виявити всі типи помилок, особливо ті, що пов'язані з логікою бізнесу.

Методи проєктування: на цьому етапі застосовуються архітектурні патерни, такі як чиста архітектура, що сприяють модульності, полегшуючи їх заміну та модифікацію. Окрім цього, використовуються принципи (DDD) для створення моделі предметної області, що відображає бізнес-логіку та полегшує розуміння системи.

Наведемо переваги та недоліки кожного з цих методів у таблиці.Методи розроблення: під час розроблення застосовуються практики, такі як (TDD), (CI/CD), використання статичного аналізу коду та інструментів для рефакторингу. Ці методи допомагають виявляти та виправляти помилки на ранніх етапах, а також автоматизувати процеси розроблення та розгортання, що пришвидшує внесення змін та зменшує ризики.

Методи тестування: на цьому етапі застосовуються різні види тестування, такі як модульне, інтеграційне, компонентне та контрактне. Вони допомагають перевірити коректність роботи окремих компонент, їхню взаємодію та відповідність вимогам. Окрім цього, використовуються інструменти для моніторингу та логування, що дають змогу відстежувати роботу системи в реальному часі та швидко виявляти та виправляти проблеми.

Проаналізувавши близько 35 наукових джерел та визначивши переваги та недоліки наявних методів забезпечення зручності супроводу мікросервісів, виявлено кілька ключових проблем, які не повністю вирішуються наявними методами:

1. *Складність розуміння бізнес-логіки*: Бізнес-логіка мікросервісу може бути складною та заплутаною, особливо якщо вона розподілена по різних класах та методах. Це ускладнює розуміння того, як мікросервіс працює та як він взаємодіє з іншими компонентами системи.
2. *Складність управління залежностями*: Мікросервісна архітектура призводить до великої кількості залежностей між сервісами, що ускладнює їх відстеження, оновлення та управління версіями.
3. *Складність тестування мікросервісних систем*: Тестування мікросервісів вимагає перевірки не тільки окремих компонент, а й їхньої взаємодії в розподіленому середовищі. Це ускладнюється асинхронною комунікацією, мережевими затримками, потенційними збоями та великою кількістю різноманітних залежностей.
4. *Складність оброблення помилок та винятків*: У розподілених системах, таких як мікросервіси, оброблення помилок і винятків може бути складною через асинхронну комунікацію та відсутність централізованого механізму оброблення помилок.

На підставі проведеного аналізу запропоновано можливі шляхи вирішення виявленої проблеми:

1. Подальший розвиток підходу Clean Architecture шляхом додавання шару доменних сервісів, що має містити надто складну доменну логіку для шару сутностей, проте надто багатофункціональну для шару сервісів додатку. Це має забезпечити спрощення розуміння бізнес-логіки шляхом підвищення згуртованості та зниження зв'язності функціоналу.
2. Подальший розвиток методу формування мінімальної структури компонента мікросервісного ПЗ, який дає змогу розбити функціонал так, щоб забезпечити підвищення згуртованості та зниження зв'язності. Це має забезпечити спрощення розуміння кодової бази.
3. Розроблення методу управління залежностями для модулів мікросервісного ПЗ шляхом поєднання методу ін'єкції залежностей з методом локатора служб, який, на відміну від наявних методів, дає змогу впроваджувати через спільний інтерфейс залежності зі змінного оточення, баз даних та інших сервісів.
4. Розроблення методу тестування ініціалізації залежностей для модулів мікросервісного ПЗ, який, на відміну від наявних методів, дасть змогу впевнитись в правильній ініціалізації усіх очікуваних модулем залежностей.
5. Подальший розвиток використання моків задля спрощення процесу тестування.

Обговорення результатів дослідження. Упродовж останніх років мікросервісна архітектура ПЗ набула значного поширення завдяки своїй гнучкості та масштабованості. Проте, як зазначено в роботах [3, 36], зручність супроводу мікросервісних систем залишається великим викликом через складність управління незалежними компонентами та їх взаємодією.

Автори досліджень [1, 3, 25] визначили, що на зручність супроводу ПЗ впливають такі чинники, як: технічна якість коду, розмір та складність проєкту, архітектурні рішення, застосування інтелектуального аналізу. На ці чинники неабиякий вплив має архітектура ПЗ. Автори досліджень [2, 10, 36] зазначають, що є і потенційні переваги в мікросервісної архітектури, але викликів, пов'язаних з чинниками, більше.

Наявні роботи, які стосуються методів для забезпечення зручності, здебільшого зосереджуються на одному методі. Наприклад, автор роботи [21] розглядає тільки підхід Clean Architecture, а автор роботи [6] – тільки підхід TDD. Проте, за результатами їхнього аналізу, використання одного підходу має обмежену ефективність у вирішенні комплексних проблем, та забезпеченні якості всіх чинників, які впливають на зручність супроводу мікросервісів.

Останніми роками почали з'являтися роботи, які пропонують комплексні підходи до вирішення цих проблем. Наприклад, автори дослідження [19] розробили метод поєднання управління залежностями з тестуванням для вдосконалення зручності супроводу мікросервісів, модульності та масштабованості безсерверних систем. Проте, цей підхід потребує значних ресурсів для впровадження та адаптації до конкретних умов, а також не є адаптованим до мікросервісної архітектури.

Аналіз наявних підходів свідчить про те, що в процесі забезпечення зручності супроводу мікросервісів досі залишаються проблеми, а саме: складність розуміння бізнес-логіки, складність управління залежностями, складність тестування мікросервісних систем, складність оброблення помилок та винятків. Запропоновані потенційні шляхи вирішення покликані усунути ці проблеми.

Отже, за результатами виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – набула подальшого розвитку методика аналізу наявних методів щодо забезпечення зручності супроводу мікросервісного ПЗ, що уможливило розроблення інтегрованих підходів, які, на відміну від наявних, враховують складність управління залежностями та тестування у розподілених системах і дають змогу покращити модульність і масштабованість мікросервісних архітектур.

Практична значущість результатів дослідження – розроблену методику аналізу наявних підходів до забезпечення зручності супроводу мікросервісного ПЗ можна використати для створення більш гнучкої та легко підтримуваної архітектури програмних систем у таких галузях, як інформаційні технології, телекомунікації, банківська справа тощо. Це дасть змогу підвищити стабільність та надійність ПЗ, знизити витрати на його

підтримку та модернізацію, а також покращити загальну продуктивність мікросервісних застосунків шляхом впровадження ефективніших методів управління залежностями та тестування.

Висновки / Conclusions

Проаналізовано останні публікації, які стосуються підвищення зручності супроводу мікросервісного ПЗ, визначено їх переваги та недоліки, виявлено проблеми, які не повністю вирішуються наявними методами, та запропоновано шляхи вирішення, щоб розробити нові та вдосконалити наявні методи, які дали б змогу підвищити показники зручності супроводу мікросервісів. За результатами проведеного дослідження можна зробити такі основні висновки:

1. Проаналізувавши літературу, виявлено основні методи забезпечення зручності супроводу мікросервісів, їх класифікацію та особливості застосування. Встановлено, що наявні підходи, такі як Clean Architecture, TDD та DDD мають свої переваги, зокрема спрощують внесення змін, є незалежними від фреймворків, та забезпечують документованість, проте не вирішують проблеми зі складністю розуміння бізнес-логіки, управління залежностями, тестування мікросервісних систем та оброблянням помилок.
2. Виявлено кілька ключових проблем у забезпеченні зручності супроводу мікросервісного ПЗ, які не повністю вирішуються наявними методами. Існує потреба в удосконаленні розуміння бізнес-логіки, оскільки її розподіленість по різних компонентах ускладнює внесення змін та підвищує ризик помилок. Недостатня автоматизація управління численними залежностями між мікросервісами та всередині них призводить до складнощів з їх оновленням та версіонуванням. Труднощі тестування, спричинені розподіленістю системи, асинхронністю та потенційними збоями, вимагають нових підходів для забезпечення надійності. Недостатня централізація механізмів оброблення помилок та винятків може призвести до непередбачуваної поведінки системи та втрати даних.
3. Запропоновано вирішення виявлених проблем, а саме подальший розвиток та вдосконалення методів забезпечення зручності супроводу мікросервісного ПЗ, зокрема: розширення підходу Clean Architecture шляхом додавання шару доменних сервісів, удосконалення структури компонент, розроблення нового методу управління залежностями, що поєднує ін'єкцію залежностей з локатором служб, створення спеціалізованого методу тестування ініціалізації залежностей та подальший розвиток використання моків для спрощення процесу тестування.
4. Виявлено ключові проблеми та запропоновано шляхи їх вирішення, на підставі яких можна розробити нові та вдосконалити наявні методи, що дасть змогу підвищити показники зручності супроводу мікросервісного ПЗ.

References

1. Abeyrathne, T. & Gayal, B. & Jayathilake, R. & Weththasinghe, W. (2021). Increase Maintainability of a Software Product using Structuredness. In *University of Kelaniya*, 1–16. URL: https://www.researchgate.net/publication/354363138_Increase_Maintainability_of_a_Software_Product_using_Structuredness
2. Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., & Clarke, P. (2023). Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. *IEEE Transactions on Software Engineering*, 1–32. https://doi.org/10.1109/tse.2023.3287297
3. Alenezi, M., & Zarour, M. (2020). On the relationship between software complexity and security. *International Journal of Software Engineering & Applications*, 11(1), 51–60. https://doi.org/10.5121/ijsea.2020.11104
4. Anwer, F., Asif, M., Ahmad, M. O., & Usman, M. (2017). Agile Software Development Models TDD, FDD, DSDM, and Crystal Methods: A Survey. *International Journal of Multidisciplinary Sciences and Engineering*, 8(2), 1–10. URL: https://www.researchgate.net/publication/316273992_Agile_Software_Development_Models_TDD_FDD_DSDM_and_Crystal_Methods_A_Survey
5. Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (3rd ed.). *Addison-Wesley Professional*. URL: https://edisciplinas.usp.br/pluginfile.php/5922722/mod_resource/content/1/2013%20-%20Book%20-%20Bass%20%20Kazman-Software%20Architecture%20in%20Practice%20%281%29.pdf
6. Bissi, W., Serra Seca Neto, A. G., & Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74, 45–54. https://doi.org/10.1016/j.infsof.2016.02.004
7. Bondyopadhyay, A., & Mandal, D. A. (2022). Improvement of Software Reliability using Data Mining Technique. *International Journal of Scientific and Research Publications*, 12(6), 183–187. URL: https:www.ijsrp.org/research-paper-0622/ijsrp-p12620.pdf
8. Boswell, D., & Foucher, T. (2011). Art of Readable Code. *O'Reilly Media, Incorporated*. URL: https://www.oreilly.com/library/view/the-art-of/9781449318482/
9. Farooq, M. S., Omer, U., Ramzan, A., Rasheed, M. A., & Atal, Z. (2023). Behavior Driven Development: A Systematic Literature Review, 1. https://doi.org/10.1109/access.2023.3302356
10. Ghofrani, J., & Lübke, D. (2018). Challenges of Microservices Architecture: A Survey on the State of the Practice. *10th Central European Workshop on Services and their Composition*. URL: https://www.researchgate.net/publication/328216639_Challenges_of_Microservices_Architecture_A_Survey_on_the_State_of_the_Practice
11. Gillis, A., & Ferguson, K. (2023). What is dependency injection in object-oriented programming (OOP). *TechTarget*. URL: https://www.techtarget.com/searchapparchitecture/definition/dependency-injection
12. Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68, 18–33. https://doi.org/10.1016/j.infsof.2015.08.002
13. Grytsiuk, P. Y., Ivanyshyn, A. V., & Hrytsiuk, Y. I. (2023). Quality assurance of software products in accordance with IEEE 730-2014 standard within the project implementation lifecycle. *Scientific Bulletin of UNFU*, 33(2), 101–117. https://doi.org/10.36930/40330214
14. Hasselbring, W. (2018). Software Architecture: Past, Present, Future. *The Essence of Software Engineering*, 169–184. Springer International Publishing. https://doi.org/10.1007/978-3-319-73897-0_10
15. Hrytsiuk, Y. I., & Mukha, T. O. (2020). Methods of determination of quality of software. *Scientific Bulletin of UNFU*, 30(1), 158-167. https://doi.org/10.36930/40300127
16. ISO/IEC. (2023). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model (ISO/IEC 25010: 2023). URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en
17. Kapferer, S., & Zimmermann, O. (2020). Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling. *8th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS – Science and Technology Publications. https://doi.org/10.5220/0008910502990306
18. Khaire, U. M., & Dhanalakshmi, R. (2019). Stability of feature selection algorithm: A review. *Journal of King Saud University – Computer and Information Sciences*. https://doi.org/10.1016/j.jksuci.2019.06.012
19. Lakhai, V. Y., Kuzmych, O. M., & Seniv, M. M. (2023). An improved method for increasing maintainability in terms of serverless architecture application. *Ukrainian Journal of Information Technology*, 5(1), 09–16. https://doi.org/10.23939/ujit2023.01.009

20. Martin Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. *martinfowler.com*. URL: https://martinfowler.com/articles/injection.html
21. Martin, R. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson Education, Limited. URL: https://audiobookstore.com/audiobooks/clean-architecture-1.aspx?campaign=dynamic_ads_international&source=google&iwAoRO5rMPrL9ChfBa4kd7AyaKboNqE90ns23s4ns33scvEnD_RLxxrSVRwUBoCm50QAvD_BwE
22. Martin, R. C. (2012). The Clean Architecture. *Clean Coder Blog*. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
23. Meyer, B. (1992). Applying 'design by contract'. *Computer*, 25(10), 40–51. https://doi.org/10.1109/2.161279
24. Molnar, A.-J., & Motogna, S. (2021). A Study of Maintainability in Evolving Open-Source Software. У Communications in Computer and Information Science, 261–282. Springer International Publishing. https://doi.org/10.1007/978-3-030-70006-5_11
25. Molnar, A.-J., & Motogna, S. (2021). A study of maintainability in evolving open-source software. *Communications in Computer and Information Science*, 261–282. Springer International Publishing. https://doi.org/10.1007/978-3-030-70006-5_11
26. Nageshkumar, M., & Mandavkar, S. (2021). Handling Unhandled Exceptions in Python 3 using Dependency Injection (DI) or Inversion of Control (IoC). *International Journal of Engineering Research & Technology* (*IJERT*), 10(7), 29–33. URL: https: www.ijert.org/research/handling-unhandled-exceptions-in-python-3-using-dependency-injection-di-or-inversion-of-control-ioc-IJERTV10IS070030.pdf
27. Nielsen, M. E., Ostergaard Madsen, C., & Lungu, M. F. (2020). Technical Debt Management: A Systematic Literature Review and Research Agenda for Digital Government. *Lecture Notes in Computer Science*, 121–137. Springer International Publishing. https://doi.org/10.1007/978-3-030-57599-1_10
28. Paasonen, T. (2011). *Methods for Improving the Maintainability of Application Software*. [Master's thesis, Aalto University]. URL: https:aaltodoc.aalto.fi/server/api/core/bitstreams/48383ab2- 54b5-4987-b30e-da2a7e1821a1/content
29. Rafique, Y., & Misic, V. B. (2013). The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis. *Transactions on Software Engineering*, 39(6), 835–856. https://doi.org/10.1109/tse.2012.28
30. Rogers, D. S. (2022). Implementing Domain-Driven Design (by V. Vernon). *ACM SIGSOFT Software Engineering Notes*, 47(3), 24. https://doi.org/10.1145/3539814.3539822
31. Sardar, M. (2023). Exploring the Impact of Software Architecture on System Performance and Maintainability: A Comprehensive Study. *ResearchGate*. URL: https://www.researchgate.net/publication/372231890_Exploring_the_Impact_of_Software_Architecture_on_System_Performance_and_Maintainability_A_Comprehensive_Study
32. Seemann, M., & Deursen, S. V. (2019). Dependency Injection Principles, Practices, and Patterns. *Manning Publications*. URL: https://www.manning.com/books/dependency-injection-principles-practices-patterns
33. Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. URL: https://arxiv.org/pdf/1703.07019
34. Sharma, T., & Singh, Y. (2022).A Systematic Literature Review on Software- refactoring Techniques, Challenges, and Practices. The University of Lahore. URL: https://www.researchgate.net/publication/360310217_A_Systematic_Literature_Review_on_Software-_refactoring_Techniques_Challenges_and_Practices
35. Späth, P., Cosmina, I., Harrop, R., & Schaefer, C. (2023). Introducing IoC and DI in Spring. *Pro Spring 6 with Kotlin*, 41–101. https://doi.org/10.1007/978-1-4842-9557-1_3
36. Thatikonda, V. K. (2023). Assessing the Impact of Microservices Architecture on Software Maintainability and Scalability. *European Journal of Theoretical and Applied Sciences*, 1(4), 782–787. https://doi.org/10.59324/ejtas.2023.1(4).71
37. Uludağ, Ö., Hauder, M., Kleehaus, M., Schimpfle, C., & Matthes, F. (2018). Supporting Large-Scale Agile Development with Domain-Driven Design. *Lecture Notes in Business Information Processing*, 232–247. Springer International Publishing. https://doi.org/10.1007/978-3-319-91602-6_16
38. Veng, M. (2014). Dependency Injection and Mock on Software and Testing [Master's thesis, Uppsala University]. URL: https://www.diva-portal.org/smash/get/diva2:726176/FULLTEXT01.pdf

V. Ya. Lakhai, M. M. Seniv

Lviv Polytechnic National University, Lviv, Ukraine

ANALYSIS OF EXISTING METHODS AND TOOLS FOR ENSURING THE MAINTAINABILITY OF MICROSERVICE SOFTWARE

This study conducts a comprehensive review of methods and tools aimed at enhancing the maintainability of microservice software systems. Although existing approaches provide valuable benefits, they do not fully address the inherent challenges of microservice architectures, particularly in understanding complex business logic, managing dependencies, executing comprehensive testing, and handling errors. The distributed nature of microservices, along with the autonomy of numerous independent components, makes maintaining these systems a complex task. Maintainability is viewed as a multifaceted characteristic that includes modularity, analyzability, reusability, modifiability, and testability. Key challenges identified include the difficulty of integrating new functionalities without disrupting existing systems, effectively managing the vast network of dependencies, and ensuring consistent testing in asynchronous communication environments. The study proposes several solutions, such as extending the Clean Architecture by introducing a domain services layer to improve the organization of business logic and reduce complexity during changes. Additionally, a new dependency management method is suggested, which combines dependency injection with a service locator pattern to create a more flexible and controlled environment. A specialized method for testing dependency initialization is also introduced, aimed at reducing early-stage development errors and enhancing system stability. The research findings emphasize the importance of continuous refinement of methodologies to enhance microservice maintainability. The proposed solutions have the potential to significantly improve maintainability, reduce risks associated with the integration of new components, and lead to more efficient management of complex microservice systems, ultimately contributing to higher software quality and lower maintenance costs.

Keywords: maintainability; microservices; Clean Architecture; testing; dependency injection.