Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)
https://doi.org/10.36930/40340511
Article received 17.05.2024
Article accepted 23.05.2024
UDC 004.052
Correspondence author: O. M. Kuzmych oleh.m.kuzmych@lpnu.ua

О. М. Кузьмич, М. М. Сенів
Національний університет "Львівська політехніка", м. Львів, Україна

АНАЛІЗ НАЯВНИХ МЕТОДІВ І ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ МІКРОСЕРВІСНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Проаналізовано літературні джерела, в яких досліджено методи та засоби забезпечення відмовостійкості мікросервісного програмного забезпечення. З'ясовано, що мікросервісна архітектура пропонує переваги у масштабованості та гнучкості, але водночас створює нові виклики у забезпеченні безперебійної роботи через розподілений характер та численні потенційні точки відмови. Оцінено вплив різних класифікацій відмов (миттєві, періодичні, тривалі) на вибір та ефективність методів забезпечення відмовостійкості (англ. *Fault Tolerance*). Охарактеризовано закономірності розвитку методів забезпечення відмовостійкості, в т.ч. й реактивні методи (повторні спроби, вимикачі ланцюгів), які спрямовані на відновлення після збою, проактивні методи (балансування навантаження, резервування), що прагнуть запобігти збоям шляхом розподілу навантаження та створення резервних копій, а також методи тестування (функціональне тестування, інженерія хаосу), які дають змогу виявляти потенційні проблеми до їх виникнення в робочому середовищі. Докладно проаналізовано переваги та недоліки кожного з цих методів, виявлено їх особливості, а також сфери застосування. Досліджено, що хоча ці методи ефективні в певних ситуаціях, жоден з них не є універсальним рішенням для всіх типів відмов. Це вказує на потребу у комплексному підході, який би враховував специфіку мікросервісної архітектури, вимоги до системи та типи можливих відмов. Визначено ключові проблеми, що потребують подальшого дослідження, такі як розроблення комплексних підходів, що враховують специфіку мікросервісної архітектури, автоматизація процесів виявлення та виправлення помилок, а також зниження вартості запровадження методів забезпечення відмовостійкості. Запропоновано перспективні напрями досліджень, зокрема розроблення нових методів і інструментів, які враховують особливості мікросервісної архітектури та забезпечують високий рівень відмовостійкості за оптимальної вартості, а також дослідження методів раннього виявлення потенційних проблем, таких як безперервне функціональне з використанням інтеграції помилок.

Ключові слова: мікросервісна архітектура; надійність програмного забезпечення; розподілені системи; тестування; методи забезпечення відмовостійкості.

Вступ / Introduction

Сучасна індустрія програмного забезпечення стрімко еволюціонує, реагуючи на експоненціальне зростання обсягів інформації та вимог до функціональності, масштабованості та відмовостійкості систем. Мікросервісна архітектура стала відповіддю на ці виклики, дала змогу створювати складні програми як сукупність незалежних сервісів. Однак розподілений характер мікросервісів і збільшення кількості потенційних точок відмови істотно ускладнюють забезпечення їхньої безперебійної роботи. У науковій літературі активно досліджують різноманітні підходи до забезпечення відмовостійкості (англ. *Fault Tolerance*) програмного забезпечення [10]. Проте більшість наявних методів і інструментів не враховують специфіку мікросервісної архітектури, а ті, що адаптовані, часто виявляються надмірно ресурсоємними. Це створює актуальну проблему пошуку ефективних, зокрема з економічної точки зору, рішень для забезпечення відмовостійкості мікросервісів. Актуальність дослідження зумовлена критичною важливістю безперебійної роботи програмного забезпечення в сучасному цифровому світі. Простої систем можуть призводити до значних фінансових втрат, втрати даних, порушення роботи підприємств і навіть загрожувати життю людей. Тому розроблення нових підходів і удосконалення наявних до забезпечення відмовостійкості мікросервісів має не тільки наукову, а й важливу практичну цінність.

Об'єкт дослідження – забезпечення відмовостійкості мікросервісного програмного забезпечення.

Предмет дослідження – методи та засоби забезпечення відмовостійкості мікросервісного програмного забезпечення, які дають змогу знизити накладні витрати на підвищення ефективності процесу його тестування.

Мета роботи – провести аналітичний аналіз останніх публікацій, які мають стосунок до методів забезпечення відмовостійкості мікросервісного програмного забезпечення, визначити їх переваги та недоліки, що дасть змогу розробити нові чи вдосконалити наявні ме-

Інформація про авторів:
Кузьмич Олег Миколайович, магістр, аспірант, кафедра програмного забезпечення.
Email: oleg.kuzmuch@gmail.com; https://orcid.org/0000-0002-9749-6385
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: Кузьмич О. М., Сенів М. М. Аналіз наявних методів і засобів забезпечення відмовостійкості мікросервісного програмного забезпечення. Науковий вісник НЛТУ України. 2024, т. 34, № 5. С. 84–89.
Citation APA: Kuzmych, O. M., & Seniv, M. M. (2024). Analysis of existing methods and tools for ensuring fault tolerance of microservice software. *Scientific Bulletin of UNFU*, 34(5), 84–89. https://doi.org/10.36930/40340511

тоди, які б сприяли зниженню накладних витрат та підвищили ефективність процесу тестування.

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

- провести аналіз останніх публікацій, що описують особливості забезпечення відмовостійкості в мікросервісному програмному забезпеченні, що дасть змогу виявити проблеми в процесі забезпечення відмовостійкості мікросервісного програмного забезпечення;
- проаналізувати та визначити переваги і недоліки наявних методів забезпечення відмовостійкості, що дасть змогу виявити проблеми в методах забезпечення відмовостійкості мікросервісного програмного забезпечення;
- сформувати потенційні шляхи вирішення цих проблем, які уможливлять розроблення нових чи вдосконалення наявних методів, що дадуть змогу знизити накладні витрати та підвищити ефективність процесу тестування.

Аналіз останніх досліджень та публікацій. Розподілений характер мікросервісної архітектури та велика кількість потенційних точок відмови значно ускладнюють забезпечення безперебійної роботи програмного забезпечення, адже відомо, що чим більше в системі елементів, тим більше можливих комбінацій дефектів, які спричиняють виникнення помилок, які своєю чергою призводять до відмов [23].

У роботах [4, 16] запропоновано та описано концепцію реактивних мікросервісів, які використовують асинхронну комунікацію та неблокуючий ввід/вивід для підвищення масштабованості та відмовостійкості. Хоча цей підхід демонструє потенціал у підвищенні продуктивності та стійкості до збоїв, його запровадження потребує ретельного проектування та використання спеціалізованих інструментів.

Автори в роботі [14] стверджують, що методи забезпечення відмовостійкості можна розділити на: реактивні та проактивні. Реактивні методи, такі як повторні спроби [3] та вимикачі ланцюгів [20, 7], спрямовані на відновлення після виникнення збою. Проактивні методи, такі як балансування навантаження [19] та резервування [5, 21], прагнуть запобігти збоям шляхом розподілу навантаження та створення резервних копій.

Робота [21] пропонує більш докладну класифікацію механізмів відмовостійкості, виокремлюючи чотири ключові області: надлишковість даних та захист, стійкість системи і сервісу, моніторинг і відновлення, а також експлуатаційні та проектні практики. Ця класифікація дає змогу систематизувати різноманітні підходи та інструменти, які використовують для забезпечення відмовостійкості мікросервісів.

У роботах [9, 15] розглянуто питання тестування мікросервісів. Автори [9] пропонують фреймворк MicroHive для автоматизованого функціонального тестування та тестування надійності, який дає змогу виявляти помилки та причинно-наслідкові зв'язки у ланцюгах збоїв. У роботі [15] досліджено проблему інтеграційного тестування безсерверних систем та запропоновано удосконалений підхід, який дає змогу значно скоротити час тестування.

Інженерія хаосу (англ. *Chaos Engineering*) [1] є перспективним підходом до забезпечення відмовостійкості, який передбачає навмисне внесення помилок у систему для перевірки її здатності до відновлення. Цей підхід дає змогу виявити приховані проблеми та покращити стійкість системи до збоїв, проте потребує ретельного планування та контролю.

У дослідженнях [13, 12] йдеться про використання платформи Kubernetes та методу вимикачів для підвищення надійності мікросервісів. Автори вказують на потребу розроблення моделі, яка б дала змогу визначати оптимальні методи забезпечення відмовостійкості для конкретних мікросервісних застосунків.

У роботі [11] порівняно два підходи до внесення помилок у систему: підхід посіву помилок (модель Міллса) та підхід ін'єкції помилок. Автори дослідження [2] пропонують набір інструментів FaultSee для відтворюваного внесення помилок у розподілені системи, що дає змогу краще оцінити вплив несправностей. У роботі [8] вказано на переваги та недоліки використання контрольних точок в розподілених обчисленнях, а також проведено порівняння наявних алгоритмів контрольних точок.

Аналіз літератури показав, що проблема відмовостійкості мікросервісного програмного забезпечення є комплексною та багатогранною. Незважаючи на значний прогрес у цій сфері, залишається багато відкритих питань, зокрема щодо розроблення комплексних підходів, які враховують специфіку мікросервісної архітектури та вимоги до системи, а також щодо створення інструментів для автоматизованого виявлення та виправлення помилок для мікросервісів. Комплексний підхід має надати можливість досягти високого рівня відмовостійкості мікросервісної системи за рахунок застосування кількох методів, які враховують різні типи відмов і особливості конкретної системи.

Результати дослідження та їх обговорення / Research results and their discussion

Визначено [16], що мікросервісна архітектура, незважаючи на свої переваги, ускладнює забезпечення відмовостійкості через розподілений характер та велику кількість потенційних точок відмови. Це призводить до виникнення різноманітних відмов, що можуть істотно вплинути на роботу системи.

Відмовостійкість (англ. *Fault Tolerance*) – здатність програмного продукту або компонента працювати, як передбачено, незважаючи на наявність апаратних або програмних несправностей [10].

Забезпечення відмовостійкості є критично важливим завданням, оскільки відмови можуть призвести до значних фінансових втрат, втрати даних, порушення роботи підприємств і навіть загрожувати життю людей. Проте робити це складно, оскільки відмов є велика кількість і вони всі різні. Їх можна прокласифікувати за тривалістю на [6]:

- *миттєві відмови* (англ. *Transient faults*) – це короткочасні збої, які швидко зникають окремо. Приклади: тимчасова втрата мережевого з'єднання, короткочасне перевантаження сервера, збій в роботі кешу. Такі відмови зазвичай можна усунути за допомогою повторних спроб (retries) або механізмів автоматичного відновлення.
- *періодичні відмови* (англ. *Intermittent faults*) – це збої, які виникають періодично, але не постійно. Причини можуть бути різними: програмні помилки, неправильна конфігурація, апаратні проблеми. Виявлення та усунення таких відмов може бути складнішим, оскільки вони не завжди легко відтворюються.
- *тривалі відмови* (англ. *Permanent faults*) – це збої, які зберігаються протягом тривалого часу і не зникають окремо. Приклади: апаратний збій, критична помилка в коді, некоректні дані. Такі відмови зазвичай потребують

втручання людини для усунення, наприклад, перезапуску сервісу, відновлення даних з резервної копії або виправлення коду.

Вирішити проблему забезпечення відмовостійкості в мікросервісних системах мали наявні методи забезпечення відмовостійкості. Їх поділяють на:

Реактивні методи: Спрямовані на відновлення системи після виникнення збою. До цієї категорії належать такі методи, як повторні спроби (retries) та вимикачі ланцюгів (англ. *Circuit Breakers*). Повторні спроби дають змогу обробляти тимчасові збої, автоматично повторюючи запит до сервісу, який не відповів. Вимикачі ланцюгів запобігають каскадним збоям, тимчасово відключаючи доступ до проблемного сервісу [14].

Проактивні методи: прагнуть запобігти збоям шляхом розподілу навантаження та створення резервних копій. До цієї категорії належать такі методи, як балансування навантаження (англ. *Load Balancing*) та резервування. Балансування навантаження розподіляє запити між кількома екземплярами сервісу, запобігаючи перевантаженню окремих екземплярів. Резервування створює резервні копії даних та сервісів, що дає змогу швидко відновити роботу системи у разі збою [14].

*Методи тестування: д*ають змогу виявляти потенційні проблеми та вразливість до їх виникнення в робочому середовищі. До цієї категорії належать такі методи, як функціональне тестування, тестування навантаження та інженерія хаосу (англ. *Chaos Engineering*). Функціональне тестування перевіряє коректність роботи окремих функцій та сервісів. Тестування навантаження дає змогу оцінити продуктивність системи під високим навантаженням. Інженерія хаосу передбачає навмисне внесення помилок у систему для перевірки її здатності до відновлення.

Наведемо переваги та недоліки кожного з цих методів у таблиці.

Таблиця. Переваги та недоліки наявних підходів забезпечення відмовостійкості / Advantages and disadvantages of existing fault tolerance approaches

Публікація	Назва підходу	Переваги	Недоліки
6	Повторні спроби (англ. Retries)	Простота: легко реалізувати та налаштувати. Гнучкість: можна використовувати з будь-якими типами мікросервісів.	Реактивність: проблеми виявляються тільки після їх виникнення, що може призвести до простою. Непередбачуваність: неможливо гарантувати, що відновлення буде успішним
8	Вимикачі ланцюгів (англ. *Circuit Breakers*)	Локалізація: обмежує вплив збою на інші мікросервіси. Захист даних: запобігає поширенню проблем на інші частини системи	Складність: потребує додаткової логіки та інфраструктури. Можливість втрати даних: якщо не синхронізовані дані, може призвести до втрати даних
4	Заслони (англ. *Bulkheads*)	Локалізація: обмежує вплив збою на інші мікросервіси. Захист даних: запобігає поширенню проблем на інші частини системи	Складність: потребує додаткової логіки та інфраструктури. Можливість втрати даних: якщо не синхронізовані дані, це може призвести до їх втрати
9	Балансування навантаження (англ. *Load Balancing*)	Покращена доступність: шляхом розподілу трафіку балансування навантаження запобігає єдиній точці збою. Якщо екземпляр мікросервісу виходить з ладу, інші можуть впоратися з навантаженням, мінімізуючи час простою. Масштабованість: балансування навантаження дає змогу легко горизонтально масштабувати. У міру збільшення трафіку можна додавати додаткові екземпляри мікросервісу, розподіляючи навантаження та зберігаючи продуктивність. Підвищена продуктивність: розподіляючи трафік, балансувальники навантаження можуть запобігти перевантаженню окремих екземплярів, що призводить до швидшого часу відповіді та більш плавної взаємодії з користувачем. Моніторинг працездатності: балансувальники навантаження можуть контролювати працездатність екземплярів мікросервісу. Несправні екземпляри можна видалити з пулу, щоб вони не вплинули на загальну продуктивність.	Додаткова складність: впровадження та підтримка балансувальника навантаження ускладнює систему. Єдина точка відмови: хоча це покращує відмовостійкість у мікросервісах, сам балансувальник навантаження може стати єдиною точкою відмови. Для самого балансувальника навантаження потрібні конфігурації високої доступності. Дорого: балансування навантаження створює певні накладні витрати, оскільки потребує прийняття рішень щодо маршрутизації та керування пулами підключень. Обмежена відмовостійкість: тільки балансування навантаження може не впоратися з усіма типами збоїв. Він не вирішує таких проблем, як помилки програмного забезпечення в самому екземплярі мікросервісу.
11, 21	Резервування (англ. *Reservation*)	Дуже висока доступність: резервування мінімізує час простою за рахунок створення резервних копій або копій даних. Якщо основний компонент виходить з ладу, вторинний бере на себе роботу, забезпечуючи безперервність обслуговування. Узгодженість даних: резервування може допомогти підтримувати узгодженість даних у примірниках залежно від вибраної стратегії резервування. Це стає критично важливим для програм, які покладаються на синхронізацію даних у реальному часі	Дуже дорого: резервування вимагає додаткових ресурсів (серверів, сховищ) для підтримки реплік, що призводить до вищих операційних витрат. Складність: керування декількома копіями збільшує складність. Синхронізація даних у репліках вимагає належного впровадження та може призвести до додаткової затримки. Обмежена відмовостійкість: резервування може не вирішити всі збої. Такі проблеми, як помилки програмного забезпечення в самому екземплярі мікросервісу, можуть впливати на всі репліки.

	Моніторинг (англ. *Monitoring*)	Профілактика: дає змогу виявити проблеми до їх виникнення. Швидке реагування: можливість вжити заходів до того, як проблема призведе до збою	Складність: потребує додаткової інфраструктури та інструментів. Витрати: може потребувати значних ресурсів.
12, 13	Функціональне тестування (англ. *Functional testing*)	Профілактика: дає змогу виявити та виправити помилки до їх випуску в продакшн. Підвищення якості: покращує загальну якість програмного забезпечення	Час: потребує значних часових витрат. Вартість: може потребувати додаткових ресурсів.
14	Тестування хаосу (англ. *Chaos testing*)	Покращена відмовостійкість: виявляючи приховані слабкі місця, хаос-тестування допомагає розробникам зміцнити свої системи, щоб більш витончено справлятися з неочікуваними збоями. Раннє виявлення проблем: тестування хаосу може виявити проблеми до того, як вони виникнуть у виробництві, запобігаючи збоям і впливу на користувачів. Підвищена впевненість: успішне проходження випробувань хаосу створює впевненість у здатності системи справлятися з реальними збоями. Покращена спостережність: тестування хаосу допомагає виявити прогалини в системах моніторингу та оповіщення, що призводить до швидшого виявлення та вирішення реальних проблем.	Збої та час простою: тестування хаосу може порушити поточні операції та потенційно призвести до простою для деяких користувачів. Дуже важливо ретельно планувати та проводити тести, щоб мінімізувати вплив. Складність: впровадження тестування хаосу та керування ним вимагає досвіду та планування. Вибір правильних сценаріїв і інструментів може бути складним завданням. Вартість: інвестиції в інструменти для тестування хаосу та кваліфікований персонал збільшують загальну вартість процесу розроблення. Пізнє використання: тестування хаосу виконується на пізньому етапі розроблення програмного забезпечення Обмежений обсяг: тестування хаосу не може змоделювати всі можливі сценарії збою. Важливо поєднувати його з іншими методами тестування
19	Контрольні точки (англ. *Control points*)	Простіше відновлення: у разі збою мікросервіс можна відновити до раніше відомого справного стану, що спрощує процес відновлення. Швидший перезапуск: порівняно з відновленням усього стану мікросервісу, перезапуск із контрольної точки може бути набагато швидшим. Зменшена втрата даних: контрольні точки дають можливість відновити певний момент часу з мінімальною втратою даних, залежно від того, як часто створюються контрольні точки.	Підвищена складність: впровадження логіки контрольних точок ускладнює сам мікросервіс. Накладні витрати на продуктивність: створення контрольних точок може призвести до навантаження на продуктивність мікросервісу. Проблеми узгодженості даних: підтримка узгодженості даних між мікросервісами може бути складною під час використання контрольних точок, особливо якщо контрольні точки не синхронізовані між службами. Можливість втрати даних: втрата даних може статися, якщо контрольні точки не створюються досить часто.

Аналіз отриманих результатів дослідження. Проаналізувавши близько двадцяти наукових та визначивши переваги та недоліки наявних методів забезпечення відмовостійкості, було виявлено декілька ключових проблем, які ще не є вирішені наявними методами:

1. *Потреба в комплексному підході*: Більшість досліджень зосереджені на окремих методах забезпечення відмовостійкості, таких як повторні спроби, вимикачі ланцюгів або резервування. Проте, для досягнення високого рівня відмовостійкості мікросервісної системи, необхідно застосовувати комплексний підхід, який враховує різні типи відмов і особливості конкретної системи [5].
2. *Складність використання*: з аналізу наукових робіт було визначено, що немає чіткого опису, які методи потрібно використовувати в тих чи інших місцях, сервісах та етапах життєвого циклу програмного забезпечення, що не дає змоги побудувати мікросервісну систему з оптимальними показниками відмовостійкості та накладними витратами [18].
3. *Недостатність автоматизації:* Багато ефективних наявних методів вимагають ручного втручання для виявлення та виправлення помилок. Це може бути неефективним та призводити до затримок у відновленні системи. Необхідно розробляти інструменти та підходи, які дають змогу автоматизувати ці процеси.
4. *Специфіка мікросервісної архітектури*: Багато методів забезпечення відмовостійкості було розроблено для монолітних застосунків, вони не враховують специфіку мікросервісної архітектури, такої як розподіленість, незалежність сервісів і складність взаємодій. Необхідні подальші дослідження щодо адаптації та розроблення методів, які враховують особливості мікросервісів.
5. *Ціна:* методи, які забезпечують найкращу відмовостійкість, наприклад, резервування, балансування навантаження, є складними та надзвичайно дорогими в реалізації.

У майбутніх дослідженнях планується пошук шляхів вирішення цих проблем. Потенційними шляхами можуть бути:

1. Розроблення комплексного підходу, який поєднує різні методи забезпечення відмовостійкості, щоб покрити різні типи відмов. Також комплексний підхід повинен дозволяти розробникам обирати оптимальні комбінації методів і параметрів для досягнення найкращого балансу між відмовостійкістю та накладними витратами. Це може містити створення рекомендацій або інструменту для вибору методів на підставі аналізу характеристик мікросервісної системи.
2. Розроблення засобів, які чітко опишуть, які методи потрібно використовувати в тих чи інших місцях, сервісах та етапах життєвого циклу програмного забезпечення. Протокол може містити рекомендації щодо вибору методів залежно від типу сервісу, його критичності, вимог до продуктивності тощо. Це має надати можливість знизити складність використання методів.
3. Автоматизація методів забезпечення відмовостійкості та тестування, в т.ч. й їх в конвеєр CI/CD (англ. *Continuous Integration / Continuous Delivery*). Це може містити автоматичне масштабування сервісів, автоматичне відновлення після збоїв, автоматичне тестування відмовостійкості мікросервісів. Такий підхід має надати можливість пришвидшити процес забезпечення відмовостійкості програмного забезпечення та здешевити його.
4. Покращення методу балансування навантаження, його адаптація до специфіки мікросервісного програмного забезпечення. Це може містити розроблення нового алгоритму балансування навантаження для мікросервісів,

який враховує не тільки навантаження на сервіси, але й їхню доступність, затримку відповіді тощо.

5. Розроблення методу для підвищення відмовостійкості мікросервісного програмного забезпечення шляхом безперервного функціонального тестування та тестування відмовостійкості. Найбільшу увагу потрібно приділити інтеграційному, наскрізному тестуванню та тестуванню відмовостійкості. Це має сприяти виявленню помилок, які будуть призводити до відмов на ранніх етапах та дасть змогу позбутись їх, що зекономить кошти.

Обговорення результатів дослідження. Упродовж останніх декількох років мікросервісна архітектура ПЗ завоювала ринок і стала домінуючою [17]. Проте вона спричинила велику складність в процес забезпечення відмовостійкості мікросервісного програмного забезпечення. До розгляду цього питання долучилась низка науковців.

У роботах [14, 6] автори описують всю різноманітність відмов, які відбуваються в розподіленому програмному забезпеченні. Проводять їх класифікацію на миттєві, періодичні та тривалі. Також автори роботи [14] визначають, що вибір методів забезпечення відмовостійкості залежить від типу відмови і їх можна розділити на реактивні, проактивні та методи тестування.

Більшість авторів притримуються підходу використання одного методу, або однотипних методів, які борються з одним типом відмов. У роботах [3, 20, 7] описано реактивні методи (повторні спроби, вимикачі ланцюга), а в роботах [19, 21, 8] – проактивні методи (балансування навантаження, резервування, контрольні точки). Такий підхід не дає достатнього рівня відмовостійкості мікросервіній системі.

Віднедавна почали з'являтись роботи з використанням різнотипах підходів. Автори роботи [5] запропонували рішення у вигляді гібриду методів повторних спроб, активної та пасивної реплікації. Отже, їм вдалось забезпечити відмовостійкість незалежно від типу помилки, проте їх метод вийшов дуже дорогим в реалізації, без використання методів тестування та адаптованим до безсерверної архітектури, а не мікросервісної.

З аналізу цих робіт та методів стало зрозуміло, що в процесі забезпечення відмовостійкості досі залишаються проблеми, а саме: потреба комплексного підходу, складність використання, неадаптованість до розподіленої природи мікросервісів, недостатність автоматизації, ціна. 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 – отримала подальший розвиток методика аналізу наявних підходів до забезпечення відмовостійкості мікросервісного програмного забезпечення, що дало змогу розробити комплексні підходи, які, на відміну від відомих, адаптують наявні методи до розподіленої природи мікросервісів і дало змогу автоматизувати процес їх тестування.

Практична значущість результатів дослідження – розроблену методику аналізу наявних підходів до забезпечення відмовостійкості мікросервісного програмного забезпечення можна використати для створення більш надійної та стійкої до збоїв системи його тестування у таких галузях, як фінанси, електронна комерція, медицина тощо, що дасть змогу знизити ризики виникнення збоїв, мінімізувати їх вплив на роботу наявної системи та підвищити загальну ефективність мікросервісних застосунків.

Висновки / Conclusions

Проаналізовано наявні публікації, які мають відношення до методів забезпечення відмовостійкості мікросервісного програмного забезпечення, визначено їх переваги та недоліки, що дасть змогу розробити нові чи вдосконалити наявні методи, які б уможливили знизити накладні витрати та підвищити ефективність процесу тестування. За результатами проведеного дослідження можна зробити такі основні висновки.

1. Проведений аналіз літератури виявив основні методи забезпечення відмовостійкості мікросервісів, їх класифікацію та особливості застосування. Встановлено, що наявні підходи, такі як повторні спроби, вимикачі ланцюга та заслони мають свої переваги, зокрема простоту використання та локалізацію відмов, але дають змогу запобігти тільки тимчасовим відмовам та не є ефективними за інших типів збоїв. Інші методи, такі як балансування навантаження та резервування, демонструють високу ефективність у забезпеченні відмовостійкості, проте не адаптовані до розподіленості системи та мають високу вартість запровадження.
2. Виявлено низку проблем у забезпеченні відмовостійкості мікросервісів. Існує потреба в комплексному підході через різноманітність відмов, а використання наявних методів ускладнюється відсутністю чітких рекомендацій щодо їх застосування. Недостатня автоматизація уповільнює виявлення та виправлення відмов, а наявні методи не завжди враховують специфіку мікросервісної архітектури. Деякі методи, зокрема резервування та балансування навантаження, є дорогими у реалізації.
3. Запропоновано для вирішення цих проблем розробити комплексний підхід, який поєднуватиме різні методи забезпечення відмовостійкості, а також протоколи їх застосування. Важливим етапом є автоматизація методів забезпечення відмовостійкості та тестування, в т.ч. й їх в конвеєр CI/CD. Окрім цього, запропоновано покращити метод балансування навантаження, адаптувавши його до специфіки мікросервісів, та розробити новий метод підвищення відмовостійкості шляхом безперервного функціонального тестування та тестування відмовостійкості.
4. Виявлено ключові проблеми та запропоновано шляхи їх вирішення, що може сприяти підвищенню відмовостійкості мікросервісів і покращенню якості програмного забезпечення загалом.

References

1. Akuthota, A. (2023). Chaos Engineering for Microservices. *Culminating Projects in Computer Science and Information Technology*, 42, 13–21. URL: https://repository.stcloudstate.edu/csit_etds/42
2. Amaral, M., Pardal, M. L., Mercier, H., & Matos, M. (2020). FaultSee: Reproducible Fault Injection in Distributed Systems. *2020 16th European Dependable Computing Conference* (*EDCC*), Munich, Germany, 25–32. https://doi.org/10.1109/edcc51268.2020.00014
3. Ataallah, S. M. A., Nassar, S. M., & Hemayed, E. E. (2015). Fault tolerance in cloud computing – survey. *2015 11th International Computer Engineering Conference* (*ICENCO*). https://doi.org/10.1109/icenco.2015.7416355
4. Baboi, M., Iftene, A., & Gîfu, D. (2019). Dynamic Microservices to Create Scalable and Fault Tolerance Architecture. *Procedia*

Computer Science, 159, 1035–1044. https://doi.org/10.1016/j.procs.2019.09.271
5. Bouizem, Y. (2022). Fault tolerance in FaaS environments. Thèse de doctorat de, Université Rennes, 135. URL: https://theses.hal.science/tel-03882666/file/BOUIZEM_Yasmina.pdf
6. Dubois, S. (2011). Tolerating Transient, Permanent, and Intermittent Failures. Masters thesis. Université Pierre et Marie Curie. URL: https://theses.hal.science/tel-00663317/document
7. Falahah, Surendro, K., & Sunindyo, W. D. (2021). Circuit Breaker in Microservices: State of the Art and Future Prospects. *IOP Conference Series: Materials Science and Engineering*, 1077(1). https://doi.org/10.1088/1757-899 x/1077/1/012065
8. Garg, R., & Kumar, P. (2010). A Review of Checkpointing Fault Tolerance Techniques in Distributed Mobile Systems. *International Journal on Computer Science and Engineering* (*IJCSE*), 2(4), 1052–1063. URL: https://www.researchgate.net/publication/49619228_A_Review_of_Checkpointing_Based_Fault_Tolerance_Techniques_in_Mobile_Distributed_Systems
9. Giamattei, L., Guerriero, A., Pietrantuono, R., & Russo, S. (2023, January). Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software*, 207. https://doi.org/10.1016/j.jss.2023.111857
10. 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
11. Khuran, D., & Shukla, A. (2016). A Survey on Fault Injection and Bebugging. *International Journal of Innovations in Engineering and Technology* (*IJIET*), *7*(3), 283–288. URL: https://ijiet.com/wp-content/uploads/2017/01/40.pdf
12. Kozlov, M. (2022). Increasing the reliability of the microservice architecture of the educational platform. Masters thesis. Odesa I.I. Mechnikov National University. URL: https://dspace.onu.edu.ua/items/3de1b788-e1ab-4794-9221-e6ee0edb632 f
13. Kozlov, M., & Petrushuna, T. (2022). Analysis of some methods of increasing the reliability of the microservice architecture of a web application. *Informatics, Information Systems and Technologies. Nineteenth All-Ukrainian Conference of Students and Young Scientists*, 103–104. URL: https://onu.edu.ua/pub/bank/userfiles/files/fmfit/naukova_diyalnist/konferenc/Zbirka_tez_IIS_T-2022.pdf
14. Kumari, P., & Kaur, P. (2018). A survey of fault tolerance in cloud computing. *Journal of King Saud University – Computer and Information Sciences*, 33(10), 1159–1176. https://doi.org/10.1016/j.jksuci.2018.09.021
15. Kuzmych, O. M., Lakhai, V. Y., & Seniv, M. M. (2023). Improved approach to automated software integration testing under the conditions of serverless architecture. *Scientific Bulletin of UNFU*, *33*(3), 97–101. https://doi.org/10.36930/40330314
16. Livora, T. (2016). *Fault Tolerance in Microservices*. Masters thesis, Masaryk University, 76. URL: https://is.muni.cz/th/ubkja/masters-thesis.pdf
17. Loukides, M. (2023). Technology Trends for 2023. What OReilly Learning Platform Usage Tells Us About Where the Industry Is Headed. OReilly. URL: https://www.oreilly.com/radar/technology-trends-for-2023/
18. Miraj, M., & Fajar, A. (2022). Model-based resilience pattern analysis for fault tolerance in reactive microservice. *Journal of Theoretical and Applied Information Technology*, *100*(9), 3075–3093. URL: https://www.jatit.org/volumes/Vol100No9/30Vol100No9.pdf
19. Mohammadian, V., Navimipour, N. J., Hosseinzadeh, M., & Darwesh, A. (2022). Fault-Tolerant Load Balancing in Cloud Computing: A Systematic Literature Review. *Access*, 10, 12714–12731. https://doi.org/10.1109/access.2021.3139730
20. Molchanov, H., & Zhmaiev, A. (2018). CIRCUIT BREAKER IN SYSTEMS BASED ON MICROSERVICES ARCHITECTURE. *Advanced Information Systems*, *2*(4), 74–77. https://doi.org/10.20998/2522-9052.2018.4.13
21. Sharma, P., & Prasad, R. (2023). Techniques for Implementing Fault Tolerance in Modern Software Systems to Enhance Availability, Durability, and Reliability. *Eigenpub Review of Science and Technology*, 7(1), 239–251. URL: https://studies.eigenpub.com/index.php/erst/article/view/33
22. Troubitsyna, E. (2019). Model-Driven Engineering of Fault Tolerant Microservices. *The Fourteenth International Conference on Internet and Web Applications and Services* (*ICIW*). URL: https://personales.upv.es/thinkmind/dl/conferences/iciw/iciw_2019/iciw_2019_1_10_20069.pdf
23. Ugrynovsky, B. (2022). Methods and means of increasing the reliability of software, taking into account the process of its software aging. Thesis paper for achievement of the scientific degree Doctor of Philosophy in the specialty 121 Software Engineering. Lviv Polytechnic National University, 194. URL: https://lpnu.ua/sites/default/files/2022/radaphd/19620/dissertation-uhrynovskyi-rc-6.pdf

O. M. Kuzmych, M.M. Seniv

Lviv Polytechnic National University, Lviv, Ukraine

ANALYSIS OF EXISTING METHODS AND TOOLS FOR ENSURING FAULT TOLERANCE OF MICROSERVICE SOFTWARE

This study analyzes existing literature on methods and tools for ensuring fault tolerance in microservice software. We examine various failure classifications, including transient failures, (short-lived and self-correcting), intermittent failures (recurring sporadically), and permanent failures (requiring intervention to resolve), and also how these classifications influence the choice and efficacy of different fault tolerance strategies. The analysis encompasses reactive methods such as retries, circuit breakers, aimed at recovery after failure, proactive methods (load balancing, reservation) that strive to prevent failures through load distribution and redundancy, and also testing methods (functional testing, chaos engineering) that identify potential issues before they manifest in production. We delve into the strengths, weaknesses, and applicability of each method, revealing that while being effective in specific scenarios, no single method provides a universal solution for all failure types. For instance, retries are simple to implement but may exacerbate issues under sustained failures, while circuit breakers offer isolation but require careful tuning. Load balancing enhances availability and scalability, although it introduces complexity and potential vulnerabilities. Reservation ensures data and service redundancy, however it can be costly and challenging to manage. Consequently, a comprehensive approach considering microservice architecture specifics, system requirements, and potential failure types is essential. Key areas for further research include developing integrated approaches tailored to microservice architectures, automating error detection and correction processes, and reducing the cost of implementing fault tolerance methods. Thus, the study contributes to the field by highlighting promising research directions, such as creating novel methods and tools that address microservice-specific challenges and provide high fault tolerance at optimal cost, as well as investigating early detection methods for potential issues, such as continuous functional testing with error integration. Moreover, the study emphasizes the need for a standardized protocol to guide the selection and implementation of fault tolerance methods throughout the software development lifecycle.

Keywords: microservice architecture; software reliability; distributed systems; testing; methods of ensuring fault tolerance.