УДК 004.4 DOI https://doi.org/10.35546/kntu2078-4481.2024.4.39

М. М. КОТЕНКО
аспірант кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
ORCID: 0009-0002-7839-6538

Т. А. ВАКАЛЮК
доктор педагогічних наук, професор,
завідувач кафедри інженерії програмного забезпечення
Державний університет «Житомирська політехніка»
ORCID: 0000-0001-6825-4697

ВПРОВАДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ: ТЕХНІЧНІ ВИКЛИКИ ТА РІШЕННЯ

Дана робота досліджує технічні виклики, з якими стикаються організації та розробники під час впровадження мікросервісної архітектури та пропонує рішення для їх подолання. Основна увага приділяється таким аспектам, як декомпозиція сервісів, міжсервісна комунікація, забезпечення безпеки та керування даними в розподілених системах. Зокрема, Domain-Driven Design визначається як ключова стратегія для сегментації монолітних додатків на бізнес-орієнтовані мікросервіси, тим самим підвищуючи придатність системи до обслуговування та потреб масштабованості. Дискусія про міжсервісну комунікацію розкриває компроміси між синхронними та асинхронними методами, зауважуючи, як синхронна комунікація, забезпечуючи негайний фідбек, може призвести до затримок та потенційних збоїв, тоді як асинхронні стратегії покращують стійкість, масштабованість та слабозв'язність, але привносять додаткову складність. Дослідження також розглядає механізми безпеки, наголошуючи на ролі Single Sign-On та API Gateway у спрощенні контролю доступу та підвищенні безпеки в мікросервісній архітектурі. Крім того, ця робота надає рекомендації щодо вирішення викликів розподіленого керування даними, рекомендуючи реплікацію, зумовлену подіями, та паттерн Saga для забезпечення консистентності даних між сервісами, разом з використанням GraphQL, API Gateway та паттерну CQRS для ефективної агрегації та отримання даних. Розглядаючи ці аспекти, було підкреслено критичність стратегічного та інформованого підходу до впровадження мікросервісів, наголошуючи на необхідності розуміння наслідків кожної архітектурної опції для ефективної побудови стійких, масштабованих та адаптивних програмних систем.

Ключові слова: *монолітна архітектура, мікросервісна архітектура, декомпозиція сервісів, міжсервісне спілкування, забезпечення безпеки, керування розподіленими даними.*

M. M. KOTENKO
Postgraduate Student at the Software Engineering Department
Zhytomyr Polytechnic State University
ORCID: 0009-0002-7839-6538

T. A. VAKALIUK
Doctor of Pedagogical Sciences, Professor,
Head of the Software Engineering Department
Zhytomyr Polytechnic State University
ORCID: 0000-0001-6825-4697

IMPLEMENTING MICROSERVICES ARCHITECTURE: TECHNICAL CHALLENGES AND SOLUTIONS

This paper examines the technical challenges faced by organizations and developers during the implementation of microservices architecture and offers solutions to overcome them. It focuses on key aspects such as service decomposition, inter-service communication, security assurance, and data management in distributed systems. Domain-Driven Design is identified as a critical strategy for segmenting monolithic applications into business-oriented microservices, thereby enhancing the system's maintainability and scalability. The discussion on inter-service communication reveals trade-offs between synchronous and asynchronous methods, noting that while synchronous communication provides immediate feedback, it can lead to delays and potential failures. In contrast, asynchronous strategies improve resilience, scalability, and loose coupling but introduce additional complexity. The research also explores security mechanisms, emphasizing the role of Single Sign-On and API Gateway in simplifying access control and enhancing security in microservice architectures. Furthermore, the paper provides recommendations for addressing the challenges of distributed data management, advocating for event-driven replication and the Saga pattern to ensure data consistency across services, alongside

the use of GraphQL, API Gateway, and the CQRS pattern for efficient data aggregation and retrieval. By examining these aspects, the paper underscores the critical importance of a strategic and informed approach to implementing microservices, emphasizing the need to understand the implications of each architectural option for building resilient, scalable, and adaptive software systems.

Key words: *monolithic architecture, microservice architecture, service decomposition, inter-service communication, security assurance, distributed data management.*

Постановка проблеми

У процесі впровадження мікросервісної архітектури перед розробниками постають складні технічні виклики, які потребують як високого рівня технічної підготовки, так і стратегічного бачення.

Основними викликами в цьому процесі є розробка ефективних методів декомпозиції монолітних додатків на сервіси, створення механізмів для налагодження якісного міжсервісного спілкування, реалізація комплексних заходів забезпечення безпеки та оптимізація управління даними в умовах їх розподіленості. Ці компоненти колективно формують фундамент для створення високоефективних та адаптивних систем, що здатні задовольняти потреби сучасного бізнесу.

Вирішення зазначених викликів стає ключовою проблемою, що вимагає не лише глибокого технічного аналізу, але й стратегічного планування. Важливим є розуміння того, як кожен аспект впливає на архітектуру та якість системи, що розробляється. Таке всебічне дослідження є необхідним для забезпечення успішного впровадження та подальшої експлуатації системи в новому архітектурному просторі.

Аналіз останніх досліджень і публікацій

Важливим аспектом впровадження мікросервісної архітектури є декомпозиція сервісів, де робота Хюлії Вурал (Hulya Vural) і Мурата Коюнджу (Murat Koyuncu) [1] вказує на роль Domain-Driven Design як ефективного підходу до структурування мікросервісів. Аналогічно, дослідження Флоріана Радемахера (Florian Rademacher), Сабіни Захве (Sabine Sachweh), і Альберта Цюндорфа (Albert Zündorf) [3] підкреслює значення чіткої інтеграції програмного забезпечення з бізнес-процесами.

Що стосується міжсервісної взаємодії, роботи Юстаса Казанавічюса (Justas Kazanavičius) і Далюса Мажейки (Dalius Mažeika) [6], а також Бен'яміна Шафабахша (Benyamin Shafabakhsh), Роберта Лагерстрьома (Lagerström Robert), і Саймона Гакса (Simon Hacks) [7] розглядають вплив синхронних і асинхронних методів комунікації на продуктивність та стійкість системи. Вони вказують на необхідність вибору оптимального способу зв'язку, який відповідає конкретному контексту використання.

Забезпечення безпеки в мікросервісних архітектурах стає особливо важливим, як це демонструють дослідження Муріло Гоеса де Алмейди (Murilo Góes de Almeida) і Едни Діас Канедо (Edna Dias Canedo) [11], а також Нуно Матеуса-Коельо (Nuno Mateus-Coelho) та його колег [12], де розглядаються стратегії аутентифікації та авторизації. Рамасвамі Чандрамоулі (Ramaswamy Chandramouli) [13] наголошує на важливості ретельного планування механізмів безпеки, особливо через використання API Gateway як ефективного інструменту для захисту мікросервісів.

Керування розподіленими даними становить значний виклик, як це підкреслено в роботі Кіндсона Муноні (Kindson Munonye) і Петера Мартінека (Peter Martinek) [14]. Вони розглядають методи забезпечення консистентності даних і управління транзакціями в децентралізованих середовищах. В додаток, матеріали від Microsoft [15] демонструють підходи до оптимізації доступу до даних та їх агрегації, що є критично важливим для забезпечення високої продуктивності мікросервісних систем.

Формулювання мети дослідження

Метою цього дослідження є аналіз технічних викликів та розробка рекомендацій для успішного впровадження мікросервісної архітектури, зокрема у таких аспектах, як декомпозиція сервісів, міжсервісна комунікація, забезпечення безпеки та керування розподіленими даними.

Викладення основного матеріалу дослідження

1. Декомпозиція сервісів

Одним із перших викликів, з яким зіштовхуються розробники при впровадженні архітектури на основі мікросервісів, є процес декомпозиції сервісів. Даний процес передбачає декомпозицію єдиного, централізованого репозиторію коду на множину автономних сервісів, де кожен сервіс несе відповідальність за певну функціональність у рамках комплексної системи. Така трансформація вимагає методичного підходу, щоб забезпечити автономність кожного мікросервісу, при цьому забезпечуючи їхній взаємний вклад у загальні цілі системи. Реалізація цієї складної рівноваги вимагає детального аналізу домену застосунку, ідентифікації ізольованих функціональних сегментів та розробки чітких інтерфейсів комунікації між сервісами. У даному процесі Domain-Driven Design (DDD) пропонує ефективну методологію для декомпозиції системи на сервіси, дозволяючи командам розробників адаптувати архітектуру програмного забезпечення до бізнес-вимог та сприяючи плавному переходу від монолітної до мікросервісної архітектури.

DDD виступає як фундаментальний методологічний каркас для трансформації монолітних архітектур в мікросервісні системи, акцентуючи на тісній інтеграції програмного забезпечення зі складними бізнес-процесами. Цей підхід підкреслює значення колаборації між доменними експертами та розробниками для створення абстрактних моделей, які відтворюють структуру та динаміку бізнес-доменів [1, 2]. Елементи архітектури DDD, такі як агрегати, сутності, об'єкти значення, сервіси та репозиторії, відіграють центральну роль у внесенні доменно-орієнтованих знань у структуру програмного забезпечення. Використання цих елементів дозволяє відтворити і врахувати складнощі та взаємозв'язки в межах бізнес-доменів, сприяючи створенню гнучких, легко адаптованих до змін бізнесу систем [1].

Концепція «обмеженого контексту» в DDD відіграє центральну роль у ефективному переході монолітних систем до мікросервісно-орієнтованих архітектур, пропонуючи стратегію для розділення бізнес-домену на дискретні частини. Кожен «обмежений контекст» ізолює функціональні зони з власною сферою відповідальності та всюдисущою мовою, спрощуючи спільне розуміння бізнес-логіки серед команд. Ця концепція не тільки сприяє точному визначенню мікросервісів, але й забезпечує їх незалежну роботу, що підвищує модульність та масштабованість системи [2, 3].

Слід зазначити, що стратегія декомпозиції в архітектурі програмного забезпечення ефективно покращується завдяки застосуванню принципів, як-от Single Responsibility Principle та Common Closure Principle, що сприяють згуртованості та автономності мікросервісів. Відповідно до цих принципів, програмне забезпечення організовується довкола індивідуальних бізнес-функцій, гарантуючи, що кожен мікросервіс відповідає за певний набір задач. Такий підхід не тільки спрощує розподіл відповідальностей, але й удосконалює процедури розгортання та обслуговування, забезпечуючи архітектурі гнучкість та адаптивність до змін. У результаті, кожен сервіс виконує унікальну функцію, водночас інтегруючись з іншими сервісами для виконання загальних бізнес-цілей [4].

Для методичної декомпозиції монолітних додатків на мікросервіси з використанням підходу DDD, рекомендується дотримуватися наступної структурованої послідовності кроків:

1. Почніть з аналізу основного бізнес-домену та його субдоменів. На цьому етапі важливо залучити доменних експертів для ідентифікації ключових бізнес-викликів та вимог, що є критичними для розробки архітектурних рішень, зорієнтованих на потреби бізнесу.
2. Далі визначте обмежені контексти. Кожен визначений обмежений контекст включає у себе детально розроблену модель предметної області, що відображає специфіку окремого субдомену в ширшому контексті основного домену. Для кожного такого обмеженого контексту необхідно чітко визначити межі, в рамках яких розробляється всюдисуща мова. Це сприяє ясності у комунікації між командами розробників і спрощує розуміння складнощів домену, що є фундаментальним для наступної ізоляції та модульної розробки сервісів [2].
3. У рамках кожного обмеженого контексту проведіть моделювання агрегатів, сутностей та сервісів, які інкапсулюють критично важливу бізнес-логіку та дані. Це дозволяє інкапсулювати критично важливу бізнес-логіку та дані, визначити обсяг та специфікації функціональностей майбутніх мікросервісів, гарантуючи їх відповідність бізнес-вимогам [1].
4. Використовуючи розроблену модель, окресліть потенційні мікросервіси. Кожен мікросервіс має відповідати за певну бізнес-функцію або можливість. Цей процес вимагає узгодження розробки програмного забезпечення з бізнес-стратегіями [3].
5. Завершіть процес картографуванням обмежених контекстів, щоб проаналізувати взаємозв'язки та взаємодії між ними. Цей крок допомагає забезпечити ефективну інтеграцію між мікросервісами, зберігаючи модульність та гнучкість архітектури, а також сприяє функціонуванню системи як єдиного цілого.

2. Міжсервісне спілкування

Іншою технічною проблемою, з якою зіштовхуються команди при впровадженні мікросервісної архітектури, стає потреба в адаптації до значних змін у взаємодії між компонентами системи. У монолітних системах компоненти взаємодіють через внутрішньопроцесні виклики – виклики методів або функцій в межах одного процесу. Такий спосіб комунікації має переваги завдяки ефективності прямих викликів методів та оптимізацій, здійснюваних компіляторами та середовищами виконання, що дозволяє забезпечити швидку, прямолінійну, тісно зв'язану взаємодію [5, 6]. Однак, при зростанні додатка, ця модель стикається з викликами у масштабуванні та гнучкості через залежність від спільного простору пам'яті та необхідності тісної інтеграції компонентів.

У свою чергу, мікросервісна архітектура характеризується як розподілена система слабо зв'язаних сервісів, що передбачає наявність механізмів для міжсервісної взаємодії, які будуть забезпечувати комунікацію між окремими сервісами [6]. Адаптація до такої моделі має на меті заміну внутрішньопроцесних викликів на міжпроцесне спілкування, внаслідок чого виникають додаткові накладні витрати, пов'язані з необхідністю мережевої взаємодії та відсутністю ефективності, притаманної внутрішньопроцесному виконанню [7, 8]. Методи міжпроцесного зв'язку можуть бути поділені на синхронні та асинхронні типи, кожен з яких впливає на архітектурні рішення, продуктивність та здатність системи до масштабування.

Синхронна комунікація відзначається встановленням безпосереднього обміну даними у форматі запит-відповідь, при якому ініціюючий (клієнтський) сервіс залишається в очікуванні поки оброблюючий (серверний) сервіс здійснить обробку запиту та надішле результат. Для реалізації цього патерну часто використовуються такі протоколи, як HTTP/HTTPS або gRPC, що дозволяє забезпечити негайний фідбек, критично важливий для деяких типів операцій. Проте, водночас це може призводити до підвищення затримок та підвищення ризику відмов у роботі сервісів. Попри свою простоту, синхронний механізм впроваджує строгу залежність між учасниками комунікації, що вимагає їх одночасної доступності в момент обміну даними [7, 9].

Асинхронна комунікація, в свою чергу, забезпечує незалежне функціонування сервісів та обмін інформацією між ними через застосування брокерів повідомлень або систем управління потоками подій. Цей підхід забезпечує можливість обміну повідомленнями між сервісами без потреби в синхронному очікуванні відповідей, вдаючись до протоколів, таких як AMQP, та використовуючи технології подібні до RabbitMQ чи Apache Kafka. Основні переваги асинхронної комунікації охоплюють зниження взаємних залежностей між сервісами, покращення масштабованості та збільшення надійності системи, що, у свою чергу, дозволяє сервісам більш ефективно адаптуватися до періодичних перебоїв у роботі та коливань навантаження [8].

Перехід до ефективної міжсервісної комунікації в мікросервісних архітектурах передбачає вирішення таких викликів, як управління розподіленими транзакціями, логування та моніторинг, забезпечення послідовності повідомлень, обробка часткових збоїв, а також покращення продуктивності та часу відгуку. Техніки, такі як «retry/circuit breaker», «load balancing», «distributed tracing» та «service mesh», є ключовими в подоланні цих складнощів. Ці підходи не лише спрощують процес комунікації, але й створюють стійку інфраструктуру, спроможну забезпечити високий рівень безпеки, надійності та моніторингу [5, 9].

Вибір оптимального стилю міжсервісної взаємодії істотно впливає на проєктування архітектури та передбачає знаходження балансу між функціональними та нефункціональними вимогами, такими як продуктивність, масштабованість, стійкість до збоїв та зв'язність сервісів. Вибір між синхронною та асинхронною комунікацією зазвичай базується на тому, що є пріоритетом: чи це потреба в негайному зворотному зв'язку та простоті, що є характеристиками синхронного підходу, або необхідність мати гнучку, масштабовану архітектуру, яка підвищує стійкість системи, її доступність та слабозв’язність, але при цьому може ускладнити інфраструктуру та управління повідомленнями, що є особливостями асинхронного підходу [8, 9].

3. Забезпечення безпеки

Далі, у процесі впровадження мікросервісної архітектури відбувається значне переосмислення стратегій забезпечення інформаційної безпеки, особливо у сферах аутентифікації та авторизації. В контексті монолітних систем, які представляють собою інтегровані, єдині програмні рішення, загальноприйнятою практикою є використання вбудованих механізмів управління користувачами, що надаються програмними фреймворками на кшталт ASP.NET, Django чи Spring. Ці механізми безпосередньо виконують функції аутентифікації та авторизації в рамках одного монолітного програмного продукту [5, 10]. В той же час, архітектури, засновані на мікросервісах, які вирізняються розподіленою структурою з програмними компонентами, розділеними на численні незалежні сервіси, вимагають більш витонченого і глибоко продуманого підходу до реалізації процедур безпеки. Така необхідність обумовлена збільшенням складності управління ідентифікацією та доступом у межах розподіленої системи та її компонентів [5].

Фундаментальним елементом у подоланні викликів, пов'язаних з безпекою в мікросервісних архітектурах, є імплементація механізму Single Sign-On (SSO). Цей механізм дозволяє користувачам пройти процедуру аутентифікації лише один раз, після чого надається доступ до широкого спектра сервісів без необхідності повторної перевірки ідентифікації для кожного з них. Такий підхід сприяє неперервності користувацького досвіду у межах децентралізованих архітектур. Реалізація SSO досягається за допомогою втілення ключових стандартів безпеки, включаючи Security Assertion Markup Language (SAML), OAuth 2.0 та OpenID Connect (OIDC) [10, 5].

Паралельно з SSO, API Gateway слугує ключовим компонентом в архітектурі безпеки, орієнтованій на мікросервіси. Функціонуючи як централізована точка доступу для всіх клієнтських запитів, API Gateway агрегує функції аутентифікації та авторизації, тим самим звільняючи окремі мікросервіси від необхідності виконувати ці завдання. Така консолідація не лише полегшує управління безпекою у мікросервісних архітектурах, але й гарантує уніфіковане впровадження політик безпеки для всіх сервісів. API Gateway може включати різноманітні заходи безпеки, наприклад, шифрування за протоколом SSL/TLS, контроль частоти запитів, фільтрацію за IP-адресами тощо, забезпечуючи додатковий рівень захисту мікросервісів від загальновідомих мережевих вразливостей та кібератак [5, 12].

Забезпечення безпечного обміну інформацією між компонентами в мікросервісній архітектурі є критично важливим для загальної інформаційної безпеки системи. Методи, такі як використання протоколу HTTPS, застосування клієнтських сертифікатів, впровадження API-ключів та Hash-based Message Authentication Code (HMAC) через HTTP, сприяють зміцненню безпеки комунікацій між різними сервісами. Протокол HTTPS допомагає у шифруванні даних під час передачі, запобігаючи їх перехопленню [11]. Клієнтські сертифікати використовуються

для мутуальної аутентифікації між сервісами, забезпечуючи додаткову перевірку ідентичності сторін [11, 12]. Застосування HMAC у контексті HTTP дозволяє перевіряти цілісність та автентичність запитів за допомогою криптографічного підпису, а також сприяє спрощенню кешування трафіку і потенційно знижує навантаження порівняно з обробкою HTTPS-трафіку [5, 10]. В свою чергу, API-ключі слугують механізмом для ідентифікації та авторизації запитів між сервісами, спрощуючи управління правами доступу у складних системах та забезпечуючи доступ до API третіх сторін [12, 13].

Застосування цих технологічних рішень підкреслює необхідність комплексного підходу до забезпечення безпеки в мікросервісних архітектурах, акцентуючи на ролі кожного інструменту в захисті даних та безпечних взаємодіях в децентралізованому середовищі.

4. Керування розподіленими даними

Впровадження мікросервісної архітектури призводить не лише до змін у технологічному стеку, стратегіях міжсервісної взаємодії та забезпеченні безпеки, але й значних змін у підходах до управління даними. Під час цього процесу виникає необхідність у нових методах обробки та збереження даних, що відповідали б децентралізованій та розподіленій природі мікросервісів. На відміну від монолітних систем, де домінує єдина база даних, що спрощує управління транзакціями, обробку запитів і забезпечення консистентності та цілісності даних, мікросервісна архітектура пропонує новий підхід, згідно з яким кожен сервіс оперує власною базою даних. Цей підхід, відомий як "Database per service", сприяє підвищенню масштабованості сервісів та гнучкості їх розробки, оскільки дозволяє кожній службі обирати найбільш підходящу систему баз даних – будь то реляційна, NoSQL або інша. Однак, він також ставить перед розробниками складні завдання, такі як забезпечення консистентності даних між різними сервісами, управління розподіленими транзакціями та ефективне виконання запитів до даних по всій системі [14].

Досягнення консистентності даних у розподіленому середовищі ставить перед собою унікальні виклики через децентралізований характер зберігання та управління даними. Одним з ефективних способів вирішення цієї проблеми є застосування методу реплікації на основі подій (Event-Driven replication). Такий метод використовує події як засіб для одночасного оновлення даних у різних частинах системи, що допомагає зберегти їх актуальність і консистентність у всьому розподіленому середовищі. Це дозволяє різним сервісам залишатися слабо зв'язаними, оскільки їм не потрібно безпосередньо взаємодіяти з базами даних інших сервісів, а лише реагувати на зміни, що транслюються через події. Такий підхід ілюструє модель BASE (Basically Available, Soft-state, Eventually-consistent), яка акцентує увагу на доступності та кінцевій консистентності, надаючи перевагу їй перед негайною консистентністю, яка є більш характерною для централізованих систем [15].

Управління транзакціями, що охоплюють кілька мікросервісів, ускладнюється через самостійне управління власним сховищем даних кожним сервісом. Патерн "Saga" вирішує цю проблематику шляхом розбиття глобальної транзакції на серію локальних транзакцій для кожного залученого сервісу. Застосування цього підходу разом з компенсуючими транзакціями для обробки помилок забезпечує захист цілісності даних у системі. Додатково, стратегії, такі як "Scheduler Agent Supervisor" та "Compensating Transaction", надають додаткові методи для управління розподіленими транзакціями. Ці стратегії забезпечують точне виконання кожної фази транзакції та включають механізми для зворотного виконання операцій за необхідності, тим самим зберігаючи консистентність даних навіть у випадку можливих збоїв.

Ефективне виконання запитів до даних у децентралізованій архітектурі мікросервісів вимагає використання нетрадиційних рішень. В даному контексті, API Gateway виступає не лише як канал для обробки клієнтських запитів, але і як ефективна система агрегації даних, що збирає інформацію з великої кількості мікросервісів [14]. Покращуючи цю функціональність, інтеграція моделі CQRS (Command and Query Responsibility Segregation) з концепцією матеріалізованих представлень пропонує надійне рішення для агрегації даних між множиною мікросервісів. Цей підхід передбачає попереднє створення матеріалізованих представлень, формуючи таблиці, призначені виключно для читання, які об'єднують дані з різноманітних сервісів у формат, що ідеально підходить під потреби клієнтського застосунку. Така стратегія не тільки покращує продуктивність запитів шляхом сегрегації операцій читання та запису, але й дозволяє незалежне масштабування таблиць для читання, тим самим оптимізуючи продуктивність та масштабованість системи [14, 15]. Додатково, GraphQL Federation виступає як ефективна стратегія для агрегування даних та виконання запитів до них, здійснюючи це через створення єдиної схеми GraphQL. Ця схема злиття даних з різних мікросервісів, презентує інформацію клієнтам так, ніби вона має походження з однієї інтегрованої бази даних. Цей метод значно полегшує взаємодію між клієнтом і сервером, знижуючи складність та логістичні витрати, що виникають при запитуванні даних з різних розподілених джерел [15].

Висновки

У ході дослідження було проаналізовано основні технічні виклики, з якими стикаються організації та розробники під час впровадження мікросервісної архітектури, та запропоновано рішення для їх подолання. Одним із ключових аспектів є декомпозиція сервісів за допомогою підходу Domain-Driven Design, що дозволяє сегментувати монолітні додатки на незалежні бізнес-орієнтовані мікросервіси, підвищуючи їхню масштабованість і підтримуваність. Аналіз міжсервісної комунікації вказує на компроміси між синхронними та асинхронними методами, які

впливають на стійкість та продуктивність системи. Дослідження також акцентує увагу на важливості безпеки в мікросервісній архітектурі. Впровадження механізмів Single Sign-On та використання API Gateway сприяють підвищенню безпеки та полегшують управління доступом. Окрім цього, запропоновано ефективні стратегії керування розподіленими даними, включаючи реплікацію на основі подій та паттерн Saga, які допомагають забезпечити консистентність даних у розподілених системах.

З огляду на переваги, які пропонують мікросервіси, та виклики, що супроводжують їхнє впровадження, майбутні дослідження можуть зосередитися на розробці нових методів і інструментів для оптимізації проектування, аналізу та допомозі у подоланні викликів, притаманних мікросервісним архітектурам. Це може включати розробку нових підходів до моделювання та аналізу взаємодії між мікросервісними компонентами та їх адаптації до архітектурних особливостей таких систем, що сприятиме глибшому розумінню та оптимізації роботи розподілених сервісів.

У підсумку, це дослідження підкреслює, що успішне впровадження мікросервісних систем вимагає глибокого розуміння та комплексного підходу до вирішення викликів, враховуючи вплив кожної архітектурної опції на стійкість, масштабованість і продуктивність системи.

Список використаної літератури

1. Vural H., Koyuncu M. Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?. IEEE Access. 2021. Vol. 9. P. 32721–32733. DOI: 10.1109/access.2021.3060895

2. Mihai I. S. A Systematic Evaluation of Microservice Architectures Resulting from Domain-Driven and Dataflow-Driven Decomposition. Bachelor's thesis. University of Twente, 2023. URL: https://essay.utwente.nl/95827/

3. Rademacher F., Sachweh S., Zündorf A. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. Software Engineering and Formal Methods. Cham, 2018. P. 230–245. DOI: 10.1007/978-3-319-74781-1_17

4. Rudrabhatla C. K. Impacts of Decomposition Techniques on Performance and Latency of Microservices. International Journal of Advanced Computer Science and Applications. 2020. Vol. 11, no. 8. DOI: 10.14569/ijacsa.2020.0110803

5. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Incorporated, 2021. 616 p

6. Kazanavičius J., Mažeika D. Evaluation of Microservice Communication While Decomposing Monoliths. Computing and Informatics. 2023. Vol. 42, no. 1. P. 1–36. DOI: 10.31577/cai_2023_1_1

7. Shafabakhsh B., Lagerström R., Hacks S. Evaluating the Impact of Inter Process Communication in Microservice Architectures. QuASoQ@APSEC. 2020. P. 55–63. URL: https://ceur-ws.org/Vol-2767/07-QuASoQ-2020.pdf

8. I. Karabey Aksakalli et al Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software. 2021. Vol. 180. P. 111014. DOI: 10.1016/j.jss.2021.111014

9. Interservice communication. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication

10. Tereshchenko O., Trintina N. Development Principles of Secure Microservices. CPITS II. 2021. P. 11–20. URL: https://ceur-ws.org/Vol-3188/paper2.pdf

11. de Almeida M. G., Canedo E. D. Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. Applied Sciences. 2022. Vol. 12, no. 6. P. 3023. DOI: 10.3390/app12063023

12. Mateus-Coelho N., Cruz-Cunha M., Ferreira L. G. Security in Microservices Architectures. Procedia Computer Science. 2021. Vol. 181. P. 1225–1236. DOI: 10.1016/j.procs.2021.01.320

13. Chandramouli R. Security strategies for microservices-based application systems. Gaithersburg, MD : National Institute of Standards and Technology, 2019. DOI: 10.6028/nist.sp.800-204

14. K. Munonye and P. Martinek. Evaluation of Data Storage Patterns in Microservices Archicture. 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), Budapest, Hungary, 2020. DOI: 10.1109/sose50414.2020.9130516

15. Challenges and solutions for distributed data management. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/distributed-data-management

References

1. Vural, H., & Koyuncu, M. (2021). Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice? IEEE Access, 9, 32721–32733. 10.1109/access.2021.3060895

2. Mihai, I. S. (2023). A Systematic Evaluation of Microservice Architectures Resulting from Domain-Driven and Dataflow-Driven Decomposition (Bachelor's thesis, University of Twente). https://essay.utwente.nl/95827/

3. Rademacher, F., Sachweh, S., & Zündorf, A. (2018). Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In Software Engineering and Formal Methods (pp. 230–245). Springer International Publishing. 10.1007/978-3-319-74781-1_17

4. Rudrabhatla, C. K. (2020). Impacts of Decomposition Techniques on Performance and Latency of Microservices. International Journal of Advanced Computer Science and Applications, 11(8). 10.14569/ijacsa.2020.0110803

5. Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media, Incorporated.

6. Kazanavičius, J., & Mažeika, D. (2023). Evaluation of Microservice Communication While Decomposing Monoliths. Computing and Informatics, 42(1), 1–36. 10.31577/cai_2023_1_1

7. Shafabakhsh, B., Lagerström, R., & Hacks, S. (2020, December). Evaluating the Impact of Inter Process Communication in Microservice Architectures. In QuASoQ@APSEC (pp. 55-63). https://ceur-ws.org/Vol-2767/07-QuASoQ-2020.pdf

8. Karabey Aksakalli, I., Çelik, T., Can, A. B., & Teki̇nerdoğan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software, 180, 111014. 10.1016/j.jss.2021.111014

9. Interservice communication. https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication

10. Tereshchenko, O., & Trintina, N. (2021). Development principles of secure microservices. In CPITS II (Vol. 2, pp. 11-20). https://ceur-ws.org/Vol-3188/paper2.pdf

11. de Almeida, M. G., & Canedo, E. D. (2022). Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. Applied Sciences, 12(6), 3023. 10.3390/app12063023

12. Mateus-Coelho, N., Cruz-Cunha, M., & Ferreira, L. G. (2021). Security in Microservices Architectures. *Procedia Computer Science*, *181*, 1225–1236. 10.1016/j.procs.2021.01.320

13. Chandramouli, R. (2019). *Security strategies for microservices-based application systems*. National Institute of Standards and Technology 10.6028/nist.sp.800-204

14. K. Munonye and P. Martinek (2020). Evaluation of Data Storage Patterns in Microservices Archicture. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. IEEE. 10.1109/sose50414.2020.9130516

15. Challenges and solutions for distributed data management. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/distributed-data-management