software testing, including AI and ML, big data testing, IoT testing, and robotic process automation. Strategies for overcoming challenges and staying ahead of competitors, such as continuous learning, collaboration, pilot projects, agile methodologies, and customer focus, are also explored.

The article concludes that adopting innovative testing practices is essential for improving the quality, efficiency, and competitiveness of software in today's fast-paced and dynamic digital environment. Early defect detection is fundamental to the shift-left strategy, offering substantial benefits to organizations. Identifying and fixing defects early significantly reduces costs, accelerates development cycles, and shortens time-to-market, allowing for quicker responses to market demands. Early defect detection enhances product quality and reliability, leading to a positive user experience, increased customer satisfaction, and loyalty. Organizations that proactively address defects gain customer trust and strengthen their market reputation. Continuous testing is emphasized as a crucial component in modern software development practices, ensuring quality and reliability throughout the software lifecycle.

Software testing, Shift-Left Testing, Continuous Testing, AI and ML, Test Automation, Shift-Right Testing, Test Data Management

Одержано (Received) 16.09.2024 *Прорецензовано (Reviewed) 12.10.2024*
Прийнято до друку (Approved) 28.10.2024

УДК 004.4 DOI: https://doi.org/10.32515/2664-262X.2024.10(41).16-29

О.С. Улічев, канд. техн. наук, **О.П. Доренський,** доц., канд. техн. наук
Центральноукраїнський національний технічний університет м. Кропивницький, Україна
В.П. Кулагін, асп.
Приватний вищий навчальний заклад "Європейський університет", м. Київ, Україна
e-mail: askin79@gmail.com, dorensky@ukr.net, victor@kulagin.com.ua

Інноваційні рішення та переваги мікросервісної архітектури програмних продуктів

У цьому дослідженні розглядається мікросервісна архітектура, як сучасний підхід до розробки програмного забезпечення, який відповідає зростаючим вимогам бізнесу та технологій. Проведено порівняльний аналіз мікросервісної архітектури з іншими архітектурними стилями, такими як монолітна та сервіс-орієнтована архітектури. Розглянуто приклади впровадження мікросервісів в індустрії на прикладі технологічних гігантів. Висвітлено використання інструментів та технологій, зокрема контейнеризації та оркестрації. Проаналізовано стандарти та найкращі практики, що сприяють ефективному впровадженню мікросервісів. Результати дослідження сприятимуть глибшому розумінню мікросервісної архітектури.

мікросервісна архітектура, масштабованість, DevOps, контейнеризація, оркестрація, розподілені системи

Постановка проблеми. У сучасному світі інформаційних технологій традиційні монолітні архітектури програмного забезпечення стають перешкодою для швидкого розвитку та масштабування систем, що не відповідає зростаючим вимогам бізнесу та користувачів. Мікросервісна архітектура виникла як рішення цих проблем, пропонуючи розподіл додатка на незалежні сервіси, але її впровадження пов'язане з новими викликами, такими як складність управління розподіленими системами.

Аналіз останніх досліджень і публікацій. Мартін Фаулер та Джеймс Льюїс у своїй статті "Microservices" [1] формалізували концепцію мікросервісної архітектури. Вони підкреслили переваги цього підходу, зокрема підвищену гнучкість, масштабованість

та можливість незалежного розгортання сервісів, що стало фундаментом для подальших досліджень у цій сфері. Сем Ньюман у книзі "Building Microservices: Designing Fine-Grained Systems" [2] детально розглядає принципи та практики розробки мікросервісів. Автор надає практичні рекомендації щодо проєктування, розгортання та підтримки мікросервісних систем, акцентуючи увагу на викликах, пов'язаних із складністю управління розподіленими системами. Марк Беккерс у статті "Understanding the Potential of Modulith Architecture" [3] досліджує модулітну архітектуру як компромісний підхід між монолітами та мікросервісами. Автор підкреслює, що модулітна архітектура поєднує модульність та простоту розгортання, що робить її привабливою для малих та середніх проєктів, де впровадження мікросервісів може бути надмірно складним або невиправданим з точки зору витрат. Нікола Драгоні та інші у роботі "Microservices: Yesterday, Today, and Tomorrow" [4] аналізують еволюцію мікросервісної архітектури, її переваги та недоліки. Вони досліджують сучасні тенденції та перспективи розвитку мікросервісів, підкреслюючи необхідність стандартів і найкращих практик для успішного впровадження цього підходу. O'Reilly Media в опитуванні "Microservices Adoption in 2020" [5] відзначають успішне впровадження мікросервісної архітектури, що свідчить
про широке прийняття та актуальність мікросервісів у сучасній індустрії програмного забезпечення, підтверджуючи ефективність цього підходу в реальних проєктах. Cloud Native Computing Foundation (CNCF) у своєму звіті "CNCF Survey 2020" [6] вказує, що 92% респондентів використовують контейнери в продакшені. Це демонструє значне поширення контейнеризації як ключової технології для розгортання мікросервісів, що підвищує ефективність та масштабованість систем. International Data Corporation (IDC) у дослідженні "IDC Forecasts a Robust Market for Enterprise Applications as Organizations Pursue Digital Era Strategies" [7] повідомляє, що 46% компаній планують замінити свої поточні системи протягом наступних трьох років. Це підкреслює постійний інтерес до нових архітектурних підходів, таких як мікросервіси, для підтримки цифрової трансформації бізнесу та підвищення конкурентоспроможності.

Постановка завдання. Метою цієї роботи є надання аналізу мікросервісної архітектури, включаючи її історичний розвиток, сучасні тенденції, інструменти та технології. Будуть розглянуті практичні аспекти впровадження мікросервісної архітектури, наведені приклади успішних кейсів та обговорені альтернативні підходи, такі як модуліт. Це сприятиме отриманню повного уявлення про мікросервісну архітектуру та обґрунтованому вибору, щодо її впровадження в проєктах.

Виклад основного матеріалу. У наш час швидкість розвитку бізнесу та технологій постійно зростає. Конкуренція змушує компанії оперативно адаптуватися до змін ринку, впроваджуючи нові функції та сервіси для задоволення потреб користувачів. У таких умовах традиційні монолітні архітектури програмного забезпечення часто стають перешкодою для швидкого розвитку та масштабування [8].

Монолітні системи, у яких додаток є єдиним нерозривним цілим, можуть бути ефективними на початкових етапах розвитку проєкту. Проте з часом, коли система стає більш складною, зміни в одній частині коду можуть впливати на інші, що ускладнює підтримку та розвиток додатка [8]. Масштабування монолітних додатків є неефективним, оскільки зазвичай вимагає масштабування всього додатка, навіть якщо навантаження зросло лише на одну з його частин [1].

Мікросервісна архітектура виникла як відповідь на ці виклики. Вона передбачає розподіл додатка на ряд невеликих, незалежних сервісів, кожен з яких виконує конкретну бізнес-функцію та може розроблятися, розгортатися і масштабуватися незалежно від інших. Такий підхід дозволяє командам розробників працювати

автономно, обирати найкращі технології для вирішення своїх задач та швидше впроваджувати нові функції [2]. Переваги мікросервісної архітектури були продемонстровані технологічними гігантами, такими як Netflix, Amazon та Spotify, які успішно впровадили цей підхід і змогли значно покращити продуктивність та масштабованість своїх систем. Однак впровадження мікросервісів тягне за собою нові виклики, пов'язані зі складністю управління розподіленою системою, забезпеченням безпеки, моніторингом та тестуванням [9]. Також треба враховувати, що не всі організації готові до повного переходу на мікросервісну архітектуру. Для таких випадків існує концепція модулітної архітектури, модуліту (Modulith), яка поєднує переваги модульності мікросервісів з простотою монолітного розгортання. Модуліт дозволяє структурувати додаток на окремі модулі з чіткими інтерфейсами, що спрощує підтримку та розвиток системи без значних змін в інфраструктурі [3].

Мікросервісна архітектура є сучасним підходом в розробці програмного забезпечення, що передбачає розподіл додатка на набір невеликих, автономних сервісів. Кожен мікросервіс виконує конкретну бізнес-функцію та може бути розроблений, розгорнутий і масштабований незалежно від інших. Основними характеристиками мікросервісної архітектури є незалежність сервісів, легка взаємодія, децентралізоване управління даними та автономність розгортання. Кожен мікросервіс є самостійним та відповідає за певну бізнес-можливість або функцію. Це дозволяє командам розробників працювати автономно, що прискорює процес розробки та впровадження. Сервіси взаємодіють між собою через прості механізми зв'язку, зазвичай HTTP/REST, що спрощує інтеграцію та зменшує тісну пов'язаність між компонентами [10]. Кожен мікросервіс управляє власною базою даних або моделлю даних, що дозволяє уникнути конфліктів при доступі до спільних ресурсів і покращує продуктивність [11]. Мікросервіси можуть бути розгорнуті незалежно один від одного, що дозволяє швидше впроваджувати зміни та нові функції без впливу на всю систему [12]. Мікросервісна архітектура є результатом еволюції програмних архітектур та прагнення до більш гнучких, масштабованих і стійких систем. Розуміння її витоків вимагає аналізу попередніх архітектурних підходів та технологічних тенденцій, які вплинули на її формування.

У перші десятиліття розвитку програмного забезпечення більшість програмних продуктів будувалися як монолітні системи, де весь функціонал був зосереджений в одному великому кодовому масиві. Такий підхід був ефективним для невеликих проєктів, але зі зростанням складності проєктів виникали проблеми. Зміни в одній частині коду могли впливати на інші його частини. Це ускладнювало тестування та розгортання. Масштабування вимагало додавання ресурсів до всього додатка, навіть якщо лише одна його частина відчувала навантаження. Будь-яка зміна потребувала повного перерозгортання додатка, що уповільнювало цикл розробки [4].

У 2000-х роках з'явилася сервіс-орієнтована архітектура (SOA) як відповідь на обмеження монолітних систем. До переваг SOA можна віднести те, що сервіси могли бути використані в різних додатках, що сприяло повторному використанню компонентів. Також використання SOA полегшувало підтримку та розвиток окремих компонентів завдяки модульності. Проте SOA мала недоліки, зокрема складність впровадження через використання важких протоколів та складних стандартів [14].

З появою агільних методологій (Agile Manifesto) у 2001 році розробники почали шукати способи більш швидкого та більш адаптивного розвитку програмного забезпечення [15]. Рух DevOps, який виник у кінці 2000-х років, підкреслив важливість інтеграції між командами розробки та операцій для швидкого та надійного розгортання.

Ці тенденції вимагали архітектур, які підтримують швидкі цикли розробки, незалежність команд та автоматизацію процесів.

Поняття мікросервісів почало формуватися у великих технологічних компаніях, які стикалися з проблемами масштабування та гнучкості. Компанія Netflix після серйозного збою у 2008 році переосмислила свою архітектуру, перейшовши до мікросервісів для покращення стійкості та масштабованості. У 2014 році Джеймс Льюїс та Мартін Фаулер опублікували статтю, в якій формалізували концепцію мікросервісної архітектури [1].

Контейнеризація стала ключовим фактором у розвитку мікросервісів. Docker, випущений у 2013 році, дозволив легко упаковувати додатки та їх залежності в контейнери, що спростило розгортання та масштабування [16]. Інструменти оркестрації, наприклад Kubernetes, випущений Google у 2014 році, автоматизували розгортання, управління та масштабування контейнеризованих додатків [17]. Перехід від важких протоколів, таких як SOAP, до легких, наприклад RESTful API, спростив комунікацію між сервісами.

Підхід Domain-Driven Design (DDD), запропонований Еріком Евансом у 2003 році, вплинув на структурування мікросервісів навколо бізнес-доменів [15]. DDD підкреслює важливість моделювання програмного забезпечення відповідно до бізнес-процесів та концепцій.

Мікросервіси дозволяють структурувати системи навколо бізнес-потреб, що відповідає принципу закону Конвея, який стверджує, що системи проєктуються відповідно до комунікаційних структур організацій [7]. Це забезпечує можливість командам фокусуватися на конкретних бізнес-цілях та швидше реагувати на зміни ринку. Кожен мікросервіс може бути розроблений з використанням різних технологій та мов програмування, що дозволяє вибирати найкращі інструменти для вирішення конкретних задач [1]. Мікросервісна архітектура підкреслює ключові переваги: незалежність сервісів, гнучкість, масштабованість та відповідність бізнес-потребам. Вона стала відповіддю на обмеження традиційних архітектур і відкрила нові можливості для розвитку складних програмних систем.

Мікросервісна архітектура є результатом еволюції попередніх архітектурних підходів та реакцією на обмеження, з якими стикалися розробники в минулому. Далі буде проведено порівняння мікросервісної архітектури з іншими популярними архітектурними стилями, такими як монолітна архітектура, сервіс-орієнтована архітектура (SOA), модульна монолітна архітектура та безсерверна архітектура (Serverless). Це допоможе визначити, коли і чому варто обирати той чи інший підхід.

Монолітна архітектура є традиційним підходом до розробки програмного забезпечення, де весь додаток реалізований як єдиний нерозривний блок. Усі компоненти системи інтегровані в один додаток, який розгортається як єдиний артефакт [2]. Вона має ряд переваг. На початкових етапах проєкту монолітні додатки легше розробляти, тестувати та розгортати, оскільки вся логіка знаходиться в одному місці [2]. Відсутність розподіленої системи спрощує процес тестування, оскільки немає необхідності враховувати мережеві взаємодії між сервісами. Уся логіка зосереджена в одному репозиторії, що може спростити управління версіями та залежностями. Але також треба враховувати і недоліки монолітної архітектури. Масштабування моноліту часто означає копіювання всього додатку на декілька серверів, що може бути неефективним, якщо лише частина системи потребує додаткових ресурсів [2]. Будь-які зміни в коді вимагають повторного розгортання всього додатку, що сповільнює впровадження нових функцій [4]. Використання однієї технології для всього додатку може обмежувати можливості команди вибирати найкращі інструменти для конкретних

задач [10]. Зі зростанням розміру коду монолітні додатки стають важкими для розуміння та підтримки.

Сервіс-орієнтована архітектура (SOA) – це підхід, у якому додаток складається з набору взаємодіючих сервісів, що надають функціональність через стандартизовані інтерфейси та протоколи [18]. SOA була популярною у 2000-х роках і стала попередником мікросервісної архітектури. Вона має багато спільного з мікросервісною архітектурою. Обидва підходи базуються на розподілі додатку на окремі сервіси [18]. Сервіси можуть бути використані в різних контекстах та додатках. Фокус на бізнес-процеси. Обидві архітектури орієнтовані на моделювання бізнес-процесів через сервіси. Але SOA має і відмінності від мікросервісної архітектури. У SOA сервіси часто більші та виконують більш комплексні функції, тоді як мікросервіси орієнтовані на конкретні задачі [13]. SOA зазвичай використовує важкі протоколи, такі як SOAP, які можуть бути складними у налаштуванні та підтримці. Мікросервіси надають перевагу легким протоколам, таким як REST або gRPC [10]. SOA часто передбачає наявність центральної шини сервісів (ESB), що може стати "вузьким місцем" та єдиною точкою відмови. Мікросервіси прагнуть до децентралізації та незалежності. У мікросервісах команди мають більше свободи у виборі технологій, тоді як SOA часто обмежена корпоративними стандартами. Вона добре підходить для великих підприємств, які потребують інтеграції різних систем [18]. Використання загальноприйнятих стандартів сприяє сумісності та взаємодії. До недоліків можна віднести те, що провадження SOA може бути складним та дорогим, вимагаючи значних зусиль на налаштування та підтримку [13]. Крім того, використання важких протоколів може впливати на продуктивність системи.

Модульна монолітна архітектура поєднує підхід моноліту з модульністю. Додаток розбивається на окремі модулі з чітко визначеними інтерфейсами, але все ще розгортається як єдиний артефакт [10]. Модульність сприяє кращому розумінню коду та спрощує навігацію. Характеризується відсутністю складної інфраструктури для управління розподіленими сервісами. Надає можливість тестувати модулі окремо в контексті єдиного додатку. З недоліків можна виділити неможливість масштабувати окремі модулі незалежно. Також існує ризик створення тісно пов'язаних модулів, що може призвести до складнощів у підтримці. Крім того, зміни в одному модулі можуть вимагати повторного розгортання всього додатку.

Безсерверна архітектура (Serverless) або Функції як Сервіс (FaaS), — це модель, де розробники пишуть невеликі фрагменти коду (функції), які виконуються у відповідь на певні події, а управління серверами здійснюється провайдером хмарних послуг [13]. До переваг можна віднести зниження операційних витрат, так як плата здійснюється лише за фактичний час виконання функцій, швидке розгортання та автоматичне масштабування. Хмарний провайдер автоматично масштабує ресурси залежно від навантаження. Із недоліків виділемо обмеження продуктивності ("холодний старт" функцій може призводити до затримок) та залежність від провайдера. Можлива прив'язка до конкретної хмарної платформи, що ускладнює міграцію. Також багато провайдерів встановлюють обмеження на час виконання функцій.

З вищесказаного можна зробити висновок, що вибір архітектурного стилю залежить від багатьох факторів: розміру проєкту, досвіду команди, вимог до масштабованості, бюджетних обмежень тощо. Мікросервісна архітектура забезпечує високу гнучкість та масштабованість, але вимагає складної інфраструктури та досвідчених команд [3, 9]. Монолітна архітектура підходить для невеликих проєктів або стартапів, де швидкість розробки важливіша за масштабованість [10]. SOA добре підходить для великих підприємств, що потребують інтеграції різних систем, але її

складність може бути перешкодою [18]. Модульна монолітна архітектура може бути компромісом між монолітом та мікросервісами, дозволяючи структурувати код без складнощів розподілених систем. Безсерверна архітектура ідеальна для подійно-орієнтованих додатків та може знизити витрати, але погано підходить для довготривалих процесів [13].

Таблиця 1 - Порівняльна таблиця архітектурних стилів проектування програмного забезпечення

Параметр	**Монолітна Архітектура**	**Сервіс-Орієнтована Архітектура (SOA)**	**Мікросервісна Архітектура**	**Модульна Монолітна Архітектура**	**Безсерверна Архітектура (Serverless)**
Розгортання	Єдиний артефакт	Розподілені сервіси	Розподілені мікросервіси	Єдиний артефакт	Функції на хмарній платформі
Масштабованість	Обмежена	Складна	Висока	Обмежена	Автоматична
Управління Інфраструктурою	Просте	Складне	Складне	Просте	Мінімальне
Гнучкість Розробки	Низька	Середня	Висока	Середня	Висока
Вартість	Низька на старті	Висока	Середня	Низька	Залежить від використання
Технологічна Незалежність	Обмежена	Обмежена корпоративними стандартами	Висока	Обмежена	Висока

Джерело: розроблено авторами

Протягом останніх десяти років популярність мікросервісної архітектури значно зросла серед розробників програмного забезпечення. Це пов'язано з потребою в масштабованих, гнучких та легко підтримуваних системах. Згідно з опитуванням [5], проведеним O'Reilly Media у 2020 році серед своїх читачів, 61% респондентів використовують мікросервіси більше року, ще 28% мінімум 3 роки. Про успішність впровадження мікросервісів кажуть дані, що 54% компаній вважають впровадження мікросервісів «переважно успішним», тоді як 10% абсолютно успішним, а загалом 92% опитаних відмічають хоча б мінімальний успіх.

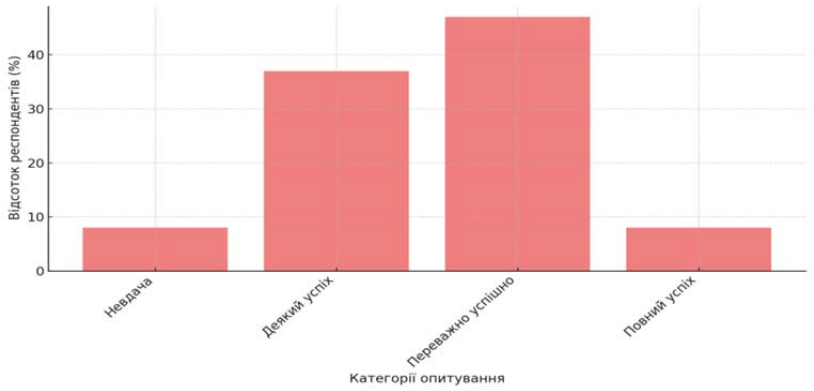

Рисунок 1 – Результати опитування щодо впровадження мікросервісів

Джерело: розроблено авторами на підставі [5]

Cloud Native Computing Foundation (CNCF) у своєму звіті за 2020 рік зазначає [6], що 92% респондентів використовують контейнери у продакшені, що є ключовим показником впровадження мікросервісів, оскільки контейнери часто використовуються для їх розгортання.

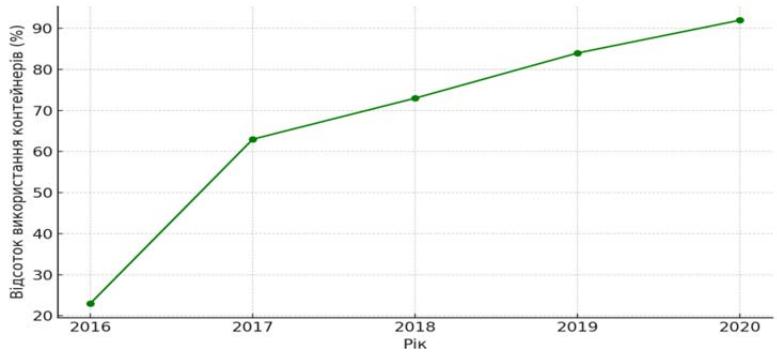

Рисунок 2 – Динаміка росту використання контейнерів в продакшені
Джерело: розроблено авторами на підставі [6]

Багато компаній активно працюють над переходом від монолітних архітектур до більш модульних і гнучких підходів. Це пов'язано з необхідністю швидшого виведення продуктів на ринок та легшого масштабування. IDC SaaSPath Survey 2023 [7] показує, що 46% компаній планують заміну своїх поточних систем протягом наступних трьох років, що свідчить про постійний інтерес до нових архітектур, таких як хмарні рішення та мікросервіси. Це підкреслює важливість модульності та гнучкості для підприємств у цифрову епоху.

Спираючись на тенденції та дослідження попередніх років, можна зробити висновки, що мікросервісна архітектура продовжує залишатися одним із найпопулярніших підходів. Багато організацій переводять свої монолітні додатки на мікросервіси для підвищення масштабованості та гнучкості. Безсерверні (Serverless) та подієво-орієнтовані (Event-Driven) архітектури також набувають все більшої популярності, особливо в контексті хмарних обчислень та Інтернету речей (IoT). Контейнеризація та оркестрація за допомогою таких інструментів, як Docker і Kubernetes, стали стандартом для розгортання та управління додатками, що свідчить про стійкий тренд до зростання використання мікросервісної архітектури у продакшн середовищах.

Сучасна мікросервісна архітектура значною мірою залежить від різноманітних інструментів та технологій, які підтримують розробку, розгортання та управління розподіленими системами.

Впровадження мікросервісів стало можливим завдяки розвитку контейнеризації, оркестрації, сервісних сіток та інших ключових технологій. Контейнеризація є фундаментальною технологією для мікросервісної архітектури, яка дозволяє ізолювати додатки та їх залежності в окремих контейнерах. Docker є однією з найпопулярніших платформ контейнеризації, що забезпечує консистентність середовища розробки та розгортання, спрощує перенесення додатків між різними середовищами та полегшує масштабування [16]. Використання контейнерів дозволяє запускати кожен мікросервіс у власному ізольованому середовищі, що підвищує безпеку та стабільність системи. Зі зростанням кількості мікросервісів управління контейнерами стає складним завданням. Системи оркестрації контейнерів, такі як Kubernetes, автоматизують розгортання, управління та масштабування контейнеризованих додатків [17]. Kubernetes забезпечує автоматичне розгортання та відновлення контейнерів, балансування навантаження та управління конфігураціями, що дозволяє підтримувати високу доступність та продуктивність системи. Компанія Airbnb перейшла на використання Kubernetes для

управління своїми мікросервісами, що дозволило покращити стійкість та масштабованість їх системи [19].

Сервісні сітки (Service Mesh) є інфраструктурним шаром, який відповідає за безпечну та надійну комунікацію між сервісами в мікросервісній архітектурі. Інструменти на кшталт Istio та Linkerd надають можливості для балансування навантаження, маршрутизації трафіку, забезпечення безпеки та моніторингу між сервісами [20]. Використання сервісних сіток спрощує управління складними мережевими взаємодіями та підвищує спостережуваність системи. Компанія PayPal використовує Istio для управління комунікацією між своїми мікросервісами, що покращує безпеку та моніторинг їх системи [21].

Ефективне управління мікросервісами вимагає потужних інструментів для моніторингу та логування. Системи на кшталт Prometheus для збору метрик та Grafana для їх візуалізації дозволяють відстежувати стан сервісів та швидко реагувати на проблеми. ELK Stack (Elasticsearch, Logstash, Kibana) використовується для централізованого збору, аналізу та візуалізації логів, що спрощує діагностику та налагодження системи [22].

Автоматизація процесів розробки та розгортання є критично важливою для мікросервісної архітектури. Інструменти для безперервної інтеграції та доставки (CI/CD), такі як Jenkins, GitLab CI/CD та CircleCI, дозволяють автоматизувати тестування та розгортання сервісів, що прискорює цикл розробки та підвищує якість програмного забезпечення.

Інструменти та технології, що підтримують контейнеризацію, оркестрацію, сервісні сітки, моніторинг та автоматизацію CI/CD, є невід'ємною частиною сучасної мікросервісної архітектури. Вони забезпечують необхідну інфраструктуру для розробки, розгортання та управління розподіленими системами, дозволяючи компаніям ефективно масштабувати та підтримувати свої сервіси.

Одним із найвідоміших прикладів успішного впровадження мікросервісної архітектури є компанія Netflix. Переходячи від монолітної архітектури, Netflix зміг масштабувати свій сервіс для обслуговування понад 200 мільйонів користувачів по всьому світу [23]. Мікросервіси дозволили компанії підвищити стійкість системи, оскільки збій одного сервісу не впливає на роботу всієї платформи. Крім того, незалежні команди можуть розробляти та розгортати сервіси без залежності від інших команд, що прискорює впровадження змін [2]. Можливість масштабувати окремі сервіси забезпечує ефективне використання ресурсів.

Компанія Amazon також перейшла до мікросервісної архітектури для вирішення проблем масштабованості та продуктивност. Розбивши свою велику кодову базу на дрібні сервіси, вона досягла покращення продуктивності та спростила процеси розробки та підтримки.

Компанія Uber зіткнулася з проблемами швидкого зростання та необхідністю масштабувати свій сервіс у різних країнах і містах. Перехід на мікросервіси дозволив Uber швидко адаптуватися до локальних вимог та регуляцій, забезпечуючи гнучкість і масштабованість системи [2].

Банки та фінансові установи активно впроваджують мікросервісну архітектуру для покращення своїх цифрових сервісів. Наприклад, Goldman Sachs перейшов до мікросервісів для свого платіжного сервісу, що дозволило швидше впроваджувати нові функції та забезпечувати високу доступність системи. ING Bank успішно впровадив мікросервіси та хмарні технології, що дозволило зменшити час виходу на ринок нових продуктів та покращити клієнтський досвід.

Компанії електронної комерції, такі як eBay та Alibaba, використовують мікросервіси для обробки великої кількості транзакцій та забезпечення безперебійної роботи платформи. Alibaba завдяки мікросервісам змогла обробляти пікові навантаження під час великих розпродажів, таких як День самотніх, забезпечуючи високу продуктивність системи.

Телекомунікаційні компанії також впроваджують мікросервісну архітектуру для покращення своїх цифрових сервісів. Наприклад, Telefónica використовує мікросервіси для підвищення масштабованості та швидкості впровадження нових послуг, що дозволяє швидше реагувати на потреби ринку.

У автомобільній промисловості компанії Volkswagen та Tesla застосовують мікросервіси для своїх інформаційно-розважальних систем та сервісів автономного водіння. Мікросервісна архітектура дозволяє швидко оновлювати програмне забезпечення та додавати нові функції, що є критичним у контексті швидкого розвитку технологій [24].

У сфері охорони здоров'я мікросервіси застосовуються для створення електронних медичних записів та телемедицини. Компанія Philips Healthcare використовує мікросервісну архітектуру для своїх цифрових рішень, що покращує інтеграцію та обмін даними між різними системами, забезпечуючи більш ефективне надання медичних послуг.

Платформи дистанційного навчання, такі як Coursera та Udemy, використовують мікросервіси для забезпечення масштабованості та персоналізації навчального досвіду. Це дозволяє обслуговувати велику кількість користувачів та адаптувати навчальні матеріали під індивідуальні потреби.

Впровадження мікросервісної архітектури приносить значні переваги для індустрії. Зокрема, незалежні сервіси дозволяють швидше впроваджувати нові функції, що скорочує час виходу на ринок [12]. Можливість масштабувати лише ті сервіси, які цього потребують, підвищує ефективність використання ресурсів. Стійкість системи забезпечується тим, що збій одного сервісу не призводить до зупинки всього додатку. Крім того, кожен сервіс може бути розроблений з використанням найкращих технологій для конкретної задачі, що підвищує гнучкість технологій [25].

Попри численні переваги, компанії стикаються з певними викликами при впровадженні мікросервісної архітектури. Складність управління розподіленою системою вимагає розвинених систем моніторингу та оркестрації. Розподілена архітектура збільшує поверхню атаки, що підвищує вимоги до безпеки системи. Крім того, необхідні культурні зміни та трансформація організаційної структури та процесів для успішного впровадження мікросервісів [2].

Мікросервісна архітектура, як складна та розподілена система, потребує дотримання певних стандартів та найкращих практик для забезпечення її ефективності, надійності та масштабованості. Cloud Native Computing Foundation (CNCF) є однією з провідних організацій, що сприяє розвитку хмарних технологій та мікросервісної архітектури. CNCF об'єднує проєкти з відкритим кодом, які допомагають розробникам створювати хмарні додатки. Основною метою CNCF є стандартизація та підтримка стандартів для хмарних і мікросервісних технологій. Організація підтримує екосистему з відкритим кодом, включаючи такі проєкти, як Kubernetes, Prometheus та Envoy. Також CNCF об'єднує розробників та організації для обміну досвідом і найкращими практиками. Серед внесків CNCF у мікросервіси можна виділити Kubernetes, який став стандартом для оркестрації контейнерів [26]. Prometheus став стандартом для моніторингу та оповіщення в мікросервісних системах, а Envoy функціонує як проксі-сервер і сервісна сітка для управління трафіком.

Методологія Twelve-Factor App, розроблена інженерами компанії Heroku, описує набір найкращих практик для побудови сучасних, масштабованих веб-додатків. Хоча вона не була створена спеціально для мікросервісів, її принципи широко застосовуються в цій архітектурі. Дванадцять факторів включають використання єдиної кодової бази, явне декларування та ізоляцію залежностей, зберігання конфігурації в середовищі, трактування сторонніх сервісів як приєднаних ресурсів та чітке розділення стадій побудови, запуску та виконання. Також важливими є запуск додатка як одного або більше безстанних процесів, експортування сервісів через портове зв'язування, масштабування за допомогою процесів моделі, швидкий старт та граційне завершення процесів, збереження паритету між середовищами розробки та продакшену, трактування логів як потоків подій та виконання адміністративних завдань як однократних процесів.

Впровадження практик DevOps та Continuous Integration/Continuous Deployment (CI/CD) є критичним для успішної роботи з мікросервісами. Принципи DevOps включають співпрацю між командами розробників та операційних інженерів, автоматизацію процесів розробки, тестування та розгортання, а також постійний моніторинг системи з швидким реагуванням на проблеми [27]. Практики CI/CD передбачають регулярне інтегрування коду в спільну кодову базу з автоматичним тестуванням (Continuous Integration) та автоматичне розгортання перевіреного коду в продакшен-середовище (Continuous Deployment).

Правильний дизайн API та вибір протоколів комунікації є ключовими для ефективної взаємодії між мікросервісами. RESTful API використовує стандартизовані методи HTTP, такі як GET, POST, PUT, DELETE, та ресурсно-орієнтований підхід для відображення бізнес-об'єктів на URL [28]. Протокол gRPC забезпечує високу продуктивність завдяки використанню HTTP/2 та бінарного формату даних, а також підтримує схемну комунікацію через Protocol Buffers. Архітектура, що керується подіями (Event-Driven Architecture), використовує асинхронну комунікацію за допомогою повідомлень та черг для передачі даних. Популярними платформами для обробки подій є Apache Kafka та RabbitMQ.

Безпека є критичним аспектом у розподілених системах. Практики безпеки включають аутентифікацію та авторизацію з використанням протоколів OAuth 2.0 та OpenID Connect [29], шифрування комунікацій за допомогою TLS для захисту трафіку, а також безпечне зберігання секретів із застосуванням інструментів на кшталт HashiCorp Vault.

Спостережуваність системи (observability) — це її здатність надавати інформацію про внутрішній стан. Практики включають збір та аналіз ключових метрик, відстеження запитів через різні сервіси за допомогою трасування (наприклад, OpenTracing, Jaeger), а також централізоване зберігання та аналіз логів.

Тестування в мікросервісній архітектурі є складним через розподіленість системи. Практики включають контрактне тестування для перевірки взаємодії між сервісами, автоматизоване тестування з використанням спеціалізованих інструментів та тестування стійкості шляхом інжекції збоїв для перевірки реакції системи (Chaos Engineering).

Дотримання стандартів та найкращих практик є критичним для успішного впровадження та підтримки мікросервісної архітектури. Організації, такі як CNCF, та спільнота розробників постійно працюють над розробкою та вдосконаленням цих практик, що допомагає компаніям ефективно будувати та масштабувати свої системи.

Висновки. Мікросервісна архітектура стала ключовою технологією у сучасній розробці програмного забезпечення, пропонуючи нові можливості для масштабованості, гнучкості та швидкості впровадження. Протягом цього дослідження ми розглянули різні аспекти мікросервісів, їх порівняння з іншими архітектурними стилями, впровадження в

індустрії, інструменти та технології, стандарти та найкращі практики. Окрім того, ми розглянули практичні приклади успішного впровадження мікросервісів.

Порівняння з іншими архітектурними стилями показало, що мікросервісна архітектура відрізняється від монолітних та сервіс-орієнтованих архітектур своєю гранулярністю та незалежністю сервісів. Монолітні системи часто страждають від складності підтримки та обмеженої масштабованості, тоді як мікросервіси дозволяють розробникам створювати автономні сервіси з чітко визначеними інтерфейсами. Порівняно з SOA, мікросервіси мають більш легковагу природу та не залежать від складних корпоративних сервісних автобусів.

Впровадження в індустрії продемонструвало, що мікросервіси стали стандартом для багатьох технологічних гігантів. Компанії, такі як Netflix, Amazon, Uber, Spotify та Walmart, успішно перейшли на мікросервісну архітектуру, що дозволило їм масштабувати свої системи, підвищити стійкість та прискорити впровадження нових функцій.

Інструменти та технології відіграють критичну роль у підтримці мікросервісної архітектури. Контейнеризація з використанням Docker дозволяє ізолювати сервіси та забезпечує консистентність середовища. Оркестрація за допомогою Kubernetes спрощує управління контейнерами, автоматизуючи розгортання та масштабування. Сервісні сітки, такі як Istio та Linkerd, забезпечують керування комунікацією між сервісами, підвищуючи безпеку та спостережуваність.

Стандарти та найкращі практики були розроблені для полегшення впровадження та експлуатації мікросервісів. Cloud Native Computing Foundation (CNCF) просуває стандарти та інструменти з відкритим кодом для хмарних додатків. Методологія Twelve-Factor App надає рекомендації щодо побудови масштабованих та підтримуваних додатків. DevOps та CI/CD практики сприяють автоматизації процесів розробки та розгортання, підвищуючи ефективність команд.

Дослідження показало, що мікросервісна архітектура стала важливим інструментом для організацій, які прагнуть підвищити масштабованість, гнучкість та швидкість інновацій. Однак її успішне впровадження вимагає ретельного аналізу переваг та викликів, готовності до культурних та організаційних змін, а також впровадження сучасних інструментів та практик. Майбутнє мікросервісної архітектури виглядає перспективним, з новими технологічними розробками та практиками, що сприятимуть ще більш ефективному управлінню та розробці складних систем. Організації, які активно адаптуються до цих змін та впроваджують інновації, зможуть залишатися конкурентоспроможними та успішно реагувати на виклики сучасного технологічного середовища.

Список літератури

1. M. Fowler and J. Lewis, "Microservices," martinfowler.com, Mar. 2014. Accessed Oct. 1, 2023. [Online]. URL: https://martinfowler.com/articles/microservices.html
2. S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
3. M. Beckers, "Understanding the potential of Modulith architecture," Worldline Tech Blog, Jan. 2024. Accessed Oct. 2, 2023. [Online]. URL: https://blog.worldline.tech/2024/01/23/modulith.html
4. N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017. P. 195–216, doi: 10.1007/978-3-319-67425-4_12.
5. O'Reilly Media, "Microservices Adoption in 2020," O'Reilly Radar, 2020. Accessed Oct. 1, 2023. [Online]. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/
6. Cloud Native Computing Foundation, "CNCF Survey 2020," CNCF, Nov. 2020. Accessed Oct. 3, 2023. [Online]. URL: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

7. International Data Corporation (IDC), "IDC Forecasts a Robust Market for Enterprise Applications as Organizations Pursue Digital Era Strategies," IDC, Sep. 2023. Accessed Oct. 1, 2023. [Online]. URL: https://www.idc.com/getdoc.jsp?containerId=prUS51107323
8. M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley, 2012.
9. I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture. Sebastopol, CA, USA: O'Reilly Media, 2016.
10. C. Richardson, Microservices Patterns: With Examples in Java. Shelter Island, NY, USA: Manning Publications, 2018.
11. M. E. Conway, "How Do Committees Invent?," Datamation, vol. 14, no. 4. P. 28–31, Apr. 1968.
12. L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," IEEE Softw., vol. 32, no. 2. P. 50–54, Mar./Apr. 2015, doi: 10.1109/MS.2015.27.
13. A. E. Eivy, "Be Wary of the Economics of 'Serverless' Cloud Computing," IEEE Cloud Comput., vol. 4, no. 2. P. 6–12, Mar./Apr. 2017, doi: 10.1109/MCC.2017.32.
14. E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley, 2003.
15. J. Humble and J. Molesky, "Why Enterprises Must Adopt DevOps to Enable Continuous Delivery," Cutter IT J., vol. 24, no. 8. P. 6–12, Aug. 2011.
16. D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, Mar. 2014. Accessed Oct. 5, 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/2600239.2600241
17. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Commun. ACM, vol. 59, no. 5. P. 50–57, May 2016, doi: 10.1145/2890784.
18. T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ, USA: Prentice Hall, 2005.
19. Airbnb Engineering, "Dynamic Kubernetes Cluster Scaling at Airbnb," Airbnb Engineering & Data Science Blog, May 23, 2022. Accessed Oct. 4, 2023. [Online]. Available: https://medium.com/airbnb-engineering/dynamic-kubernetes-cluster-scaling-at-airbnb-d79ae3afa132
20. W. Morgan, "What Is a Service Mesh?," Buoyant, 2017. Accessed Oct. 1, 2023. [Online]. Available: https://buoyant.io/what-is-a-service-mesh
21. R. Hincapie, "The Ultimate Guide to Becoming an Istio Certified Administrator," Medium, May 19, 2023. Accessed Oct. 2, 2023. [Online]. URL: https://richincapie.medium.com/the-ultimate-guide-to-becoming-an-istio-certified-administrator-160b845e7490
22. C. Gormley and Z. Tong, Elasticsearch: The Definitive Guide. Sebastopol, CA, USA: O'Reilly Media, 2015.
23. A. Cockcroft, "Migrating to Microservices," Nginx, 2015. Accessed Oct. 10, 2023. [Online]. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
24. M. Broy, I. H. Krüger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," Proc. IEEE, vol. 95, no. 2. P. 356–373, Feb. 2007, doi: 10.1109/JPROC.2006.888386.
25. S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," IEEE Internet Comput., vol. 14, no. 6. P. 80–83, Nov./Dec. 2010, doi: 10.1109/MIC.2010.145.
26. B. Burns, J. Beda, and K. Hightower, Kubernetes: Up and Running, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2019.
27. C. Ebert and G. Gallardo, "DevOps," IEEE Softw., vol. 33, no. 3. P. 94–100, May/Jun. 2016, doi: 10.1109/MS.2016.68.
28. R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," ACM Trans. Internet Technol., vol. 2, no. 2. P. 115–150, May 2002, doi: 10.1145/514183.514185.
29. D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012, doi: 10.17487/RFC6749.

References

1. M. Fowler and J. Lewis, "Microservices," martinfowler.com, Mar. 2014. Accessed Oct. 1, 2023. [Online]. Retrieved from https://martinfowler.com/articles/microservices.html
2. S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
3. M. Beckers, "Understanding the potential of Modulith architecture," Worldline Tech Blog, Jan. 2024. Accessed Oct. 2, 2023. [Online]. Retrieved from https://blog.worldline.tech/2024/01/23/modulith.html
4. N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017. P. 195–216, doi: 10.1007/978-3-319-67425-4_12.

5. O'Reilly Media, "Microservices Adoption in 2020," O'Reilly Radar, 2020. Accessed Oct. 1, 2023. [Online]. Retrieved from https://www.oreilly.com/radar/microservices-adoption-in-2020/
6. Cloud Native Computing Foundation, "CNCF Survey 2020," CNCF, Nov. 2020. Accessed Oct. 3, 2023. [Online]. Retrieved from https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf
7. International Data Corporation (IDC), "IDC Forecasts a Robust Market for Enterprise Applications as Organizations Pursue Digital Era Strategies," IDC, Sep. 2023. Accessed Oct. 1, 2023. [Online]. Retrieved from https://www.idc.com/getdoc.jsp?containerId=prUS51107323
8. M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley, 2012.
9. I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, Microservice Architecture: Aligning Principles, Practices, and Culture. Sebastopol, CA, USA: O'Reilly Media, 2016.
10. C. Richardson, Microservices Patterns: With Examples in Java. Shelter Island, NY, USA: Manning Publications, 2018.
11. M. E. Conway, "How Do Committees Invent?," Datamation, vol. 14, no. 4. P. 28–31, Apr. 1968.
12. L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," IEEE Softw., vol. 32, no. 2. P. 50–54, Mar./Apr. 2015, doi: 10.1109/MS.2015.27.
13. A. E. Eivy, "Be Wary of the Economics of 'Serverless' Cloud Computing," IEEE Cloud Comput., vol. 4, no. 2. P. 6–12, Mar./Apr. 2017, doi: 10.1109/MCC.2017.32.
14. E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, MA, USA: Addison-Wesley, 2003.
15. J. Humble and J. Molesky, "Why Enterprises Must Adopt DevOps to Enable Continuous Delivery," Cutter IT J., vol. 24, no. 8. P. 6–12, Aug. 2011.
16. D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," Linux Journal, vol. 2014, no. 239, Mar. 2014. Accessed Oct. 5, 2023. [Online]. Retrieved from https://dl.acm.org/doi/10.5555/2600239.2600241
17. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Commun. ACM, vol. 59, no. 5. P. 50–57, May 2016, doi: 10.1145/2890784.
18. T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ, USA: Prentice Hall, 2005.
19. Airbnb Engineering, "Dynamic Kubernetes Cluster Scaling at Airbnb," Airbnb Engineering & Data Science Blog, May 23, 2022. Accessed Oct. 4, 2023. [Online]. Retrieved from https://medium.com/airbnb-engineering/dynamic-kubernetes-cluster-scaling-at-airbnb-d79ae3afa132
20. W. Morgan, "What Is a Service Mesh?," Buoyant, 2017. Accessed Oct. 1, 2023. [Online]. Available: https://buoyant.io/what-is-a-service-mesh
21. R. Hincapie, "The Ultimate Guide to Becoming an Istio Certified Administrator," Medium, May 19, 2023. Accessed Oct. 2, 2023. [Online]. Retrieved from https://richincapie.medium.com/the-ultimate-guide-to-becoming-an-istio-certified-administrator-160b845e7490
22. C. Gormley and Z. Tong, Elasticsearch: The Definitive Guide. Sebastopol, CA, USA: O'Reilly Media, 2015.
23. A. Cockcroft, "Migrating to Microservices," Nginx, 2015. Accessed Oct. 10, 2023. [Online]. Retrieved from https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
24. M. Broy, I. H. Krüger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," Proc. IEEE, vol. 95, no. 2. P. 356–373, Feb. 2007, doi: 10.1109/JPROC.2006.888386.
25. S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," IEEE Internet Comput., vol. 14, no. 6. P. 80–83, Nov./Dec. 2010, doi: 10.1109/MIC.2010.145.
26. B. Burns, J. Beda, and K. Hightower, Kubernetes: Up and Running, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2019.
27. C. Ebert and G. Gallardo, "DevOps," IEEE Softw., vol. 33, no. 3. P. 94–100, May/Jun. 2016, doi: 10.1109/MS.2016.68.
28. R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," ACM Trans. Internet Technol., vol. 2, no. 2. P. 115–150, May 2002, doi: 10.1145/514183.514185.
29. D. Hardt, "The OAuth 2.0 Authorization Framework," RFC 6749, Oct. 2012, doi: 10.17487/RFC6749.

Oleksandr Ulichev, PhD tech. sci., **Oleksandr Dorenskyi,** Assoc. Prof., PhD tech. sci.
Central Ukrainian National Technical University, Kropyvnytskyi, Ukraine
Victor Kulahin, post-graduate
Private Higher Education Establishment "European University" Kyiv, Ukraine

Innovative Solutions and Benefits of Microservice Architecture for Software Products

The rapid advancement of technology and increasing market competition compel businesses to adapt swiftly by implementing new features and services to meet user demands. Traditional monolithic software architectures often hinder this agility due to challenges in scalability and maintenance. This article aims to

analyze microservice architecture to solve these challenges, exploring its historical development, current trends, practical implementation aspects, and comparison with alternative architectural styles such as modulith architecture.

The study examines the limitations of monolithic architectures in handling growing complexity and scaling requirements. It explores the emergence of microservice architecture, highlighting core characteristics like independent services, decentralized data management, and autonomous deployment. The evolution influenced by agile methodologies and DevOps practices is discussed. A comparative analysis with other architectural styles—including monolithic, service-oriented, modular monolithic, and serverless architectures—identifies contexts where microservices are most beneficial. The research reviews essential tools and technologies for implementing microservices, such as Docker for containerization, Kubernetes for orchestration, and service meshes like Istio and Linkerd. Practical cases from industry leaders like Netflix and Amazon illustrate successful adoption and the challenges faced during implementation.

Findings indicate that while microservice architecture offers significant scalability, flexibility, and rapid deployment advantages, it also introduces complexities related to distributed system management and security. The study emphasizes the importance of adopting best practices and standards, such as those promoted by the Cloud Native Computing Foundation and utilizing modern tools to mitigate these challenges. For organizations where full microservices adoption may be impractical, modulith architecture is a viable alternative that combines modularity with deployment simplicity. The article concludes that the choice of architecture should be carefully aligned with the project's specific needs, resources, and long-term strategic goals.

microservice architecture, containerization, scalability, orchestration, distributed systems, DevOps

Одержано (Received) 16.09..2024 *Прорецензовано (Reviewed) 03.10.2024*
Прийнято до друку (Approved) 28.10.2024

УДК 629.083 DOI: https://doi.org/10.32515/2664-262X.2024.10(41).1.29-39

О.Л. Ляшук, проф., д-р техн. наук, **В.А. Готович,** доц., канд. техн. наук, **В.О. Бонар,** асп.
Тернопільський національний технічний університет імені Івана Пулюя, м. Тернопіль, Україна
e-mail: oleglashuk@ukr.net
В.В. Аулін, проф., д-р техн. наук, **А.В. Гриньків,** ст. дослідник, канд. техн. наук
Центральноукраїнський національний технічний університет, м. Кропивницький, Україна
e-mail: AulinVV@gmail.com
Л.П. Матійчук, доц., д-р екон. наук
Тернопільський національний технічний університет імені Івана Пулюя, м. Тернопіль, Україна

Концепція дистанційної діагностики технічного стану транспортних засобів в процесі їх експлуатації

У статті розглянуто концепцію дистанційної діагностики транспортних засобів в процесі їх експлуатації та основні аспекти автоматичного зчитування даних за допомогою протоколу OBD2 з ECU в режимі реального часу з подальшою їх передачею на загальний сервер. Продемонстровано як отримані дані тестуються на наявність в них аномалій за допомогою методів реконструкції та виявленням аномалій у групах даних з подальшим сповіщенням користувача. Представлено архітектуру комунікації мобільного та веб додатків з серверною частиною при діагностиці транспортних засобів.

транспортний засіб, дистанційна діагностика, технічний стан, самодіагностика OBD2, код помилки, аномалія, в наборі даних