КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ

УДК 004.4:004.72
DOI https://doi.org/10.36994/2788-5518-2025-02-10-04

ЕВОЛЮЦІЯ АРХІТЕКТУР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ: ВІД МОНОЛІТНИХ ДО МІКРОСЕРВІСНИХ

***Бабич С. М.**, к.т.н., доц., Рівненський державний гуманітарний університет, Рівне, Україна. https://orcid.org/0000-0003-2145-6392, stepaniia.babych@rshu.edu.ua*

***Ляшук Т. Г.**, к.ф.-м.н., Рівненський державний гуманітарний університет, Рівне, Україна. https://orcid.org/0000-0002-2242-7537, taras.liashuk@rshu.edu.ua*

***Анотація.** Проведено дослідження історичної еволюції архітектур програмного забезпечення обчислювальних систем – від перших монолітних структур до сучасних мікросервісних рішень. Розглянуто ключові етапи розвитку: ранні обчислювальні системи з простими програмними структурами (1940–1960-ті роки), становлення монолітної архітектури як фундаментального підходу до побудови цілісних систем, перехід до клієнт-серверних та багаторівневих (n-tier) моделей, появу розподілених об'єктних систем (CORBA, RMI) та web-сервісів (SOAP), еволюцію REST як легковагової альтернативи та формування сервіс-орієнтованої архітектури (SOA), що підготувала ґрунт для мікросервісної парадигми. Особливу увагу приділено детальному аналізу мікросервісної архітектури як сучасного стандарту побудови масштабованих, гнучких та відмовостійких програмних систем, підтримуваних екосистемою технологій контейнеризації (Docker), оркестрації (Kubernetes) та Service Mesh.*

Здійснено систематичне порівняння монолітних, перехідних (клієнт-серверних, n-tier, SOA) та мікросервісних архітектур за критеріями масштабованості, гнучкості, складності підтримки, економічної ефективності та безпеки. Виявлено їхні переваги й обмеження та визначено закономірності еволюційного переходу між ними. Показано, що мікросервісний підхід є природним етапом розвитку архітектур програмного забезпечення, який забезпечує високий рівень автономності команд розробки, горизонтальну масштабованість компонентів, технологічну незалежність (поліглотне програмування) та інтеграцію з практиками DevOps і CI/CD. Водночас окреслено виклики мікросервісної архітектури – ускладнення інфраструктури, проблеми управління узгодженістю даних у розподілених системах і забезпечення безпеки міжсервісної комунікації.

Окреслено перспективні напрями розвитку архітектур програмного забезпечення: поширення serverless-підходів та event-driven моделей для реактивних систем, впровадження Service Mesh технологій для надійної комунікації, інтеграцію edge computing для обробки даних на периферії мережі, автоматизацію через Infrastructure as Code (IaC) та використання штучного інтелекту (AIOps) для інтелектуальної оптимізації, моніторингу й створення self-healing систем. Результати дослідження є корисними для архітекторів програмного забезпечення, інженерів і дослідників у галузі розподілених систем.

***Ключові слова:** архітектура програмного забезпечення, монолітна архітектура, мікросервіси, сервіс-орієнтована архітектура (SOA), клієнт-серверна модель, масштабованість, DevOps, контейнеризація, Kubernetes, Infrastructure as Code, еволюція програмних систем.*

THE EVOLUTION OF COMPUTING SYSTEM SOFTWARE ARCHITECTURES: FROM MONOLITHIC TO MICROSERVICES

***Stepaniia Babych**, Ph.D., Associate Professor, Rivne State University of the Humanities, Rivne, Ukraine. https://orcid.org/0000-0003-2145-6392, stepaniia.babych@rshu.edu.ua*

***Taras Liashuk**, Ph.D., Assistant Professor, Rivne State University of the Humanities, Rivne, Ukraine. https://orcid.org/0000-0002-2242-7537, taras.liashuk@rshu.edu.ua*

***Abstract.** The study explores the historical evolution of software architecture in computing systems – from early monolithic structures to modern microservice-based solutions. The research highlights key development stages: early computing systems with simple software structures (1940s–1960s), the establishment of monolithic architecture as a fundamental approach to building integrated systems, the transition to client–server and multi-tier (n-tier) models, the emergence of distributed object systems*

(CORBA, RMI) and web services (SOAP), the evolution of REST as a lightweight alternative, and the rise of service-oriented architecture (SOA) that laid the groundwork for the microservice paradigm. Particular attention is given to the in-depth analysis of microservice architecture as the modern standard for building scalable, flexible, and fault-tolerant software systems supported by containerization (Docker), orchestration (Kubernetes), and Service Mesh technologies.

A systematic comparison of monolithic, transitional (client–server, n-tier, SOA), and microservice architectures was conducted according to scalability, flexibility, maintenance complexity, cost efficiency, and security criteria. Their advantages and limitations were identified, and evolutionary transition patterns between architectural models were defined. The study shows that the microservice approach represents a natural stage in the evolution of software architectures, providing a high degree of team autonomy, horizontal scalability of components, technological independence (polyglot programming), and integration with DevOps and CI/CD practices. At the same time, it emphasizes the challenges of microservices – infrastructure complexity, data consistency in distributed systems, and interservice communication security.

The prospective directions for the development of software architectures are outlined, including the expansion of serverless approaches and event-driven models for reactive systems, the implementation of Service Mesh technologies for reliable communication, the integration of edge computing for data processing at the network periphery, automation through Infrastructure as Code (IaC), and the use of artificial intelligence (AIOps) for intelligent optimization, monitoring, and the creation of self-healing systems. The research findings are valuable for software architects, engineers, and researchers in the field of distributed systems.

Key words: *software architecture, monolithic architecture, microservices, service-oriented architecture (SOA), client–server model, scalability, DevOps, containerization, Kubernetes, Infrastructure as Code, software systems evolution.*

Вступ

Сучасні обчислювальні системи відіграють ключову роль у розвитку цифрової економіки, забезпечуючи ефективне функціонування бізнес-процесів, державного управління та соціальних сервісів. Зростання обсягів даних, підвищення вимог до масштабованості, продуктивності та надійності інформаційних рішень актуалізували потребу в нових архітектурних підходах до проєктування програмного забезпечення. Традиційна монолітна архітектура, яка домінувала в перших поколіннях програмних систем, забезпечувала простоту створення й розгортання, проте виявила низку обмежень за умов високої динамічності IT-ринку, глобалізації та зростання складності обчислювальних процесів.

У відповідь на зазначені виклики відбувся поступовий перехід до клієнт-серверних і багаторівневих архітектур, що відобразив тенденцію до підвищення модульності, розподілу відповідальності та незалежності компонентів системи. Подальший розвиток розподілених об'єктних технологій, web-сервісів і REST-підходів сприяв стандартизації міжкомпонентної взаємодії та уніфікації механізмів обміну даними. Формування сервіс-орієнтованої архітектури (SOA) створило концептуальне підґрунтя для становлення сучасної мікросервісної парадигми, яка поєднує автономність сервісів із можливостями контейнеризації, оркестрації та автоматизації процесів розгортання.

У цьому контексті актуальною є наукова проблема, яка полягає в систематизації знань про еволюцію архітектур програмних систем та розробленні критеріїв вибору оптимальних архітектурних рішень для різних типів обчислювальних систем. Незважаючи на активне впровадження мікросервісних підходів у промисловій розробці, залишається недостатньо дослідженим питання системного аналізу чинників, які зумовили перехід від монолітних до розподілених архітектур, а також узагальнення переваг і обмежень кожного етапу цієї еволюції.

Питання еволюції архітектур програмного забезпечення активно досліджується як в українській, так і в зарубіжній науковій літературі. У праці А. В. Гавриляка та К. І. Ліщук [1] систематизовано основні архітектурні шаблони та стилі проєктування програмних систем, зокрема клієнт/сервер, компонентну, багатошарову, N-рівневу архітектури, архітектуру на базі шини повідомлень, об'єктно-орієнтовану та сервісно-орієнтовану архітектури (SOA). Для кожного стилю описано принципи проєктування, переваги та сфери застосування. Дослідники підкреслюють важливість правильного вибору архітектури для забезпечення масштабованості, гнучкості та зручності обслуговування програмних систем.

У роботі В. Киселевича та співавторів [2] проаналізовано переваги та недоліки мікросервісної архітектури на прикладах конкретних систем різних масштабів. Автори демонструють, що мікросервісна архітектура не є універсальним рішенням і може бути як ефективним підходом, так і джерелом значних проблем. Особливу увагу приділено випадкам нераціонального застосування мікросервісів у системах середньої складності. Дослідники наголошують на необхідності ґрунтовного аналізу вимог до масштабованості, складності управління та витрат на обслуговування перед прийняттям рішення про вибір архітектурного підходу.

Серед зарубіжних досліджень важливе місце займають фундаментальні праці C. Newman [3] та C. Richardson [4], які систематизують принципи проєктування мікросервісних систем і пропонують практичні шаблони їх реалізації. Значний внесок у розуміння питань масштабованості, надійності

та обробки великих обсягів даних зроблено в [5], де аналізуються архітектурні рішення для побудови розподілених систем, орієнтованих на стійкість до збоїв.

Автори в роботі [6] узагальнюють підходи до побудови програмних архітектур, розкриваючи взаємозв'язок між архітектурними рішеннями, якістю програмного продукту та бізнес-вимогами. Дослідження [7] акцентує увагу на інтеграції мікросервісної архітектури з DevOps-підходами, що дозволяє забезпечити безперервну доставку та високу надійність сервісів. У свою чергу, у [8] систематизуються антипатерни мікросервісних систем і пропонується таксономія типових архітектурних помилок, що мають практичну цінність для уникнення ризиків під час реалізації розподілених систем.

Отже, проведений аналіз свідчить, що сучасні дослідження зосереджуються переважно на практичних аспектах мікросервісних архітектур, проте питання комплексного порівняння еволюційних етапів розвитку архітектур програмних систем, а також виявлення закономірностей переходу від монолітних до мікросервісних підходів залишаються недостатньо розробленими. Це зумовлює потребу в подальшій систематизації наукових положень і критеріях оцінювання ефективності архітектурних рішень.

Виконання досліджень

Історичні передумови розвитку архітектур програмного забезпечення.

Ранні етапи розвитку обчислювальних систем (1940–1960-ті роки) характеризувалися обмеженими апаратними ресурсами та відсутністю розвинених засобів програмування. Програмні продукти мали лінійну або процедурну структуру, оскільки основна увага зосереджувалася на ефективному виконанні обчислень за умов мінімальної пам'яті та обчислювальної потужності. Зі зростанням складності задач виникла потреба в структуризації коду, що започаткувало концепцію модульності та поклало початок формуванню архітектурних підходів [1].

Подальший розвиток апаратного забезпечення та мов програмування в 1970–1980-х роках зумовив перехід від неструктурованого коду до організованих програмних систем. Першим системним етапом цього переходу стала поява монолітної архітектури, яка передбачала створення програми як цілісного набору взаємозалежних модулів, що виконувалися в одному процесі і розгорталися як єдине ціле. Такий підхід забезпечував централізоване управління бізнес-логікою та спрощував інтеграцію компонентів у єдину систему, що було ефективним у контексті обмежених обчислювальних середовищ того часу.

Монолітна архітектура характеризується низкою ключових рис, що визначають її внутрішню структуру та принципи організації програмного забезпечення. Основними структурними особливостями моноліту є єдина кодова база, централізоване управління ресурсами та простота розгортання системи як єдиного модуля. Хоча ці характеристики забезпечували швидкий старт і простоту розробки, з часом вони породили суттєві обмеження в масштабованості, супроводі та гнучкості, особливо для великих і динамічних систем. Ці обмеження стали поштовхом до пошуку нових архітектурних рішень, орієнтованих на розподіл функціональності та підвищення адаптивності програмного забезпечення.

Обмеження та проблеми монолітної архітектури

Високий рівень інтеграції компонентів і значний обсяг кодової бази зумовили низку проблем, пов'язаних із підтримкою, супроводом та масштабуванням монолітних систем. У межах такої архітектури всі компоненти об'єднані в єдиний програмний модуль, тому будь-які зміни в одному елементі можуть впливати на роботу інших. Це ускладнює внесення нових функцій, виправлення помилок і проведення тестування, що поступово знижує гнучкість та керованість програмного продукту.

Зі зростанням складності систем і кількості користувачів ці обмеження ставали дедалі помітнішими, породжуючи технічний борг і зменшуючи ефективність розробки. Обмежена масштабованість монолітної архітектури означає, що систему можна масштабувати лише цілком, без можливості гнучкого розподілу навантаження між окремими компонентами. Це призводить до нераціонального використання ресурсів і збільшення витрат на інфраструктуру.

З економічної точки зору, монолітна архітектура демонструє порівняно низькі витрати на початкових етапах, оскільки вся система створюється як єдине ціле без складної інфраструктури. Однак у довгостроковій перспективі витрати на підтримку та оновлення зростають, адже кожна зміна вимагає повторного тестування всього продукту, а накопичення технічного боргу потребує додаткових ресурсів для його усунення.

У результаті монолітна архітектура, яка свого часу забезпечувала простоту створення та розгортання, поступово втратила ефективність у динамічному середовищі сучасних обчислювальних систем. Це зумовило перехід до нових архітектурних підходів, орієнтованих на розподіл функцій між компонентами, – передусім до клієнт-серверної моделі.

Перехідні архітектури

Проблеми монолітних систем, пов'язані з масштабуванням і підтримкою, стимулювали перехід до більш гнучких архітектурних рішень. Клієнт-серверна архітектура стала першим кроком у цьому напрямі, розділивши функціонал на два компоненти: клієнт надсилає запити, а сервер обробляє їх, виконує бізнес-логіку та забезпечує збереження даних [6]. Такий підхід відкрив шлях від централізованих монолітів до розподілених систем, що дало змогу ізолювати частину навантаження та підвищити ефективність використання ресурсів.

Подальший розвиток розподілених систем у 1990-х роках привів до появи розподілених об'єктних технологій, зокрема CORBA та RMI, які забезпечили міжплатформну взаємодію об'єктів у різних середовищах. З поширенням інтернету відбувся перехід до Web-сервісів, де для стандартизованої взаємодії використовувався протокол SOAP – обмін XML-повідомленнями з суворою типізацією та вбудованими механізмами безпеки. Проте складність SOAP і громіздкість XML обмежували його гнучкість.

Це стимулювало розвиток багаторівневих (n-tier) архітектур, які розділили систему на рівні представлення, бізнес-логіки та даних. Така структура підвищила модульність, спростила супровід і створила основу для незалежного масштабування окремих рівнів.

Прагнення до спрощення комунікацій призвело до появи REST-підходу, запропонованого Роєм Філдінгом у 2000 році. REST використовує стандартні можливості протоколу HTTP (GET, POST, PUT, DELETE) і компактні формати обміну даними (переважно JSON), що забезпечує простоту реалізації, високу продуктивність і масштабованість завдяки stateless-взаємодії.

Інтеграція зазначених ідей відбулася в сервіс-орієнтованій архітектурі (SOA), де програмні компоненти реалізуються як незалежні сервіси, що взаємодіють через стандартизовані інтерфейси. SOA забезпечила інтеграцію великомасштабних систем через єдину шину сервісів (ESB) і сприяла повторному використанню компонентів. Водночас значний розмір сервісів і спільне використання ресурсів залишали певні обмеження щодо гнучкості та автономності.

Мікросервісна архітектура стала наступним етапом еволюції, усунувши ці обмеження. Вона розбиває систему на дрібні, автономні сервіси, кожен з яких виконує окрему бізнес-функцію. Мікросервіси використовують легковагові протоколи взаємодії (HTTP/REST, gRPC), мають власні бази даних (підхід database-per-service) і позбавляються централізованих компонентів на кшталт ESB. Це забезпечує гнучкість, прискорює впровадження змін та підвищує автономність команд розробки.

Таким чином, еволюція архітектур програмного забезпечення – від клієнт-серверної моделі до сервіс-орієнтованих підходів – відображає послідовний рух у напрямі підвищення автономності, масштабованості та адаптивності систем. Кульмінацією цього процесу стало формування мікросервісної архітектури, яка поєднує принципи розподіленості з гнучкістю сучасних технологічних рішень.

Мікросервісна архітектура: суть та принципи

Послідовна еволюція архітектур програмного забезпечення привела до появи мікросервісної парадигми, яка поєднує переваги розподілених систем і принципів модульності. Мікросервісна архітектура ґрунтується на ідеї поділу складної системи на набір невеликих, незалежних сервісів, кожен з яких виконує окрему бізнес-функцію згідно з принципом єдиного обов'язку [3]. Такий підхід забезпечує ізоляцію змін, спрощує тестування та розгортання, підвищуючи загальну гнучкість системи.

Основою мікросервісної архітектури є декомпозиція, що передбачає розподіл за бізнес-можливостями (управління замовленнями, оплати, профілі користувачів) або за предметними областями в межах Domain-Driven Design. Використання підходу bounded contexts дозволяє чітко визначити межі відповідальності кожного сервісу. Важливу роль відіграє також подієво-орієнтована взаємодія, що забезпечує асинхронний обмін повідомленнями та мінімізує залежності між компонентами.

Переваги мікросервісів полягають у підвищеній відмовостійкості, можливості незалежного розгортання та швидкому впровадженні оновлень. Разом з тим, така декомпозиція потребує ретельного планування архітектури. Використання шаблонів проєктування – Circuit Breaker, Event Sourcing, CQRS – сприяє забезпеченню надійності та узгодженості розподіленої системи [4].

Ключовим елементом взаємодії між сервісами є інтерфейси програмування застосунків (API). Вони забезпечують стандартизований спосіб комунікації між сервісами, дозволяють реалізувати слабке зв'язування та приховують внутрішню реалізацію компонентів. Найчастіше застосовують RESTful API або gRPC, які забезпечують ефективну передачу даних і підтримують версіонування для сумісності. Безпека комунікації реалізується за допомогою OAuth 2.0, JWT і механізмів обмеження частоти запитів.

Для управління взаємодією на рівні всієї системи використовується API Gateway, який виконує функції балансування навантаження, контролю доступу, маршрутизації запитів і агрегації відповідей. Це спрощує інтеграцію клієнтів і приховує внутрішню складність мікросервісної інфраструктури.

Ефективне розгортання множини мікросервісів потребує застосування контейнеризації та оркестрації. Контейнери, зокрема Docker, забезпечують стандартизовану упаковку сервісів разом із залежностями, що підвищує портативність і швидкість розгортання. Платформа Kubernetes автоматизує керування життєвим циклом контейнерів, включаючи масштабування, балансування навантаження, самовідновлення при збоях та оновлення без простоїв.

У поєднанні технології Docker і Kubernetes формують основу сучасного середовища для розробки й експлуатації мікросервісних систем. Вони забезпечують автономність компонентів, підтримку CI/CD-процесів та високу доступність, реалізуючи на практиці ключові переваги мікросервісної архітектури.

Отже, мікросервісна архітектура формує сучасну парадигму побудови програмних систем, у якій поєднано модульність, розподіленість та гнучкість. Завдяки використанню контейнеризації, оркестрації й стандартизованих API забезпечується висока автономність компонентів, стабільність

і масштабованість системи. Ці характеристики визначають ключові переваги мікросервісного підходу, які стали підґрунтям його широкого впровадження у промислових програмних рішеннях.

Переваги мікросервісів

Мікросервісна архітектура забезпечує високу гнучкість та адаптивність завдяки розподіленню системи на автономні сервіси, кожен із яких можна розробляти, тестувати й оновлювати незалежно [2]. Це дає змогу швидко впроваджувати нові функції, виправляти помилки без зупинки всієї системи та знижувати ризики під час оновлень. Такий підхід суттєво скорочує час виведення продукту на ринок і підвищує стійкість до змін бізнес-вимог. Використання поліглотного програмування дозволяє обирати оптимальний технологічний стек для кожної задачі – наприклад, Python для аналітики та машинного навчання, Go для високопродуктивних сервісів, Node.js для веб-застосунків. Автоматизація процесів через CI/CD та використання контейнерних технологій (Docker, Kubernetes) прискорюють розгортання, а стратегії blue-green deployment і canary releases мінімізують простої, забезпечуючи можливість безпечного відкату змін.

Однією з ключових переваг є гнучка масштабованість. Кожен мікросервіс може масштабуватися окремо, відповідно до рівня навантаження, що забезпечує ефективне горизонтальне масштабування (scale-out). Наприклад, при зростанні кількості транзакцій можна масштабувати лише сервіс обробки платежів, оптимізуючи використання ресурсів та витрати на інфраструктуру. Платформа Kubernetes підтримує динамічне балансування навантаження завдяки автоматичному масштабуванню (Horizontal Pod Autoscaler), що гарантує еластичність системи. Додатково, географічно розподілене розгортання забезпечує низьку затримку обробки запитів та високу доступність навіть у разі регіональних збоїв.

Мікросервісна архітектура також гармонійно поєднується з практиками DevOps [7], сприяючи інтеграції розробки, тестування та експлуатації в єдиний безперервний процес. CI/CD автоматизує всі етапи життєвого циклу програмного забезпечення, що дозволяє впроваджувати зміни за години замість тижнів. Кожен сервіс має власний pipeline розгортання, що підвищує стабільність і передбачуваність оновлень. Використання підходу Infrastructure as Code (Terraform, Ansible) забезпечує відтворюваність і консистентність конфігурацій у різних середовищах. Формування культури “you build it – you run it” стимулює відповідальність команд і сприяє створенню більш надійного та якісного коду.

Попри значні переваги, мікросервісна архітектура має і низку специфічних викликів, пов’язаних із підвищеною складністю управління, координацією сервісів та забезпеченням їхньої узгодженої роботи в розподіленому середовищі.

Виклики та обмеження мікросервісної архітектури

Разом із перевагами мікросервісна архітектура супроводжується суттєвими інфраструктурними та організаційними викликами. На відміну від монолітних систем, мікросервісний підхід передбачає наявність множини незалежних сервісів, кожен із яких необхідно окремо розгортати, моніторити та масштабувати. Це ускладнює загальне управління системою, потребуючи стандартизації процесів, централізованого логування, розподіленого трасування та підтримки численних репозиторіїв коду. Додаткову складність створює різнорідність технологічних стеків і версій фреймворків, що впливає на сумісність компонентів та процес інтеграції.

Ефективне управління розподіленими системами потребує використання оркестраційних платформ (Kubernetes, Helm, Istio), брокерів повідомлень (Kafka, RabbitMQ) і технологій service mesh. Мережева взаємодія між сервісами піддається ризикам затримок (latency) або тимчасової недоступності, що може спричиняти каскадні збої. Тестування множини взаємозалежних сервісів у локальному середовищі є складним завданням, а перехід до мікросервісної архітектури вимагає значних організаційних та технологічних ресурсів. Ці фактори обумовлюють необхідність ретельного планування переходу й інвестицій у навчання персоналу [8]. Серед технічних викликів особливе місце посідає проблема управління даними.

Управління даними є одним із ключових викликів мікросервісної архітектури через її розподілений характер [5]. Кожен сервіс має власну базу даних (database-per-service), що забезпечує автономність, але ускладнює підтримку узгодженості та транзакційної цілісності. Якщо в монолітних системах використовуються ACID-транзакції, то в мікросервісах частіше застосовується підхід eventual consistency – відкладеної узгодженості даних. Для мінімізації ризиків та підтримки консистентності використовуються архітектурні патерни Event Sourcing, CQRS (Command Query Responsibility Segregation) і Saga, які дозволяють координувати розподілені транзакції між сервісами.

Проблемним є також виконання аналітичних запитів, які потребують об’єднання даних із кількох сервісів, оскільки традиційні SQL-операції (JOIN) у такому середовищі непридатні. Це вимагає побудови аналітичних сховищ або реалізації агрегаційних шарів через API. Серед додаткових викликів – забезпечення продуктивності (через мережеві накладні витрати), організація резервного копіювання великої кількості баз даних, дотримання вимог законодавства (наприклад, GDPR) та реалізація наскрізного шифрування.

Ще одним критичним аспектом є безпека мікросервісного середовища. Оскільки кожен сервіс взаємодіє через відкриті мережеві інтерфейси, розширюється поверхня потенційних атак. Основни-

ми завданнями є автентифікація, авторизація та шифрування даних. Для цього застосовуються протоколи OAuth 2.0, JWT, а також mutual TLS (mTLS) для двосторонньої автентифікації. Технології Service Mesh (Istio, Linkerd) забезпечують автоматизацію безпечної маршрутизації трафіку через sidecar proxies.

Забезпечення спостережуваності та відмовостійкості є ще одним важливим напрямом. Централізоване логування із використанням correlation ID дозволяє відстежувати транзакції та діагностувати аномалії за допомогою інструментів типу ELK Stack або Splunk. Підвищення надійності систем досягається впровадженням патернів Circuit Breaker, Retry та Timeout, які мінімізують наслідки збоїв. Платформа Kubernetes додатково забезпечує мережеву ізоляцію через Network Policies, автоматичне відновлення сервісів після збоїв і безпечне управління секретами.

Усвідомлення цих викликів і обмежень стимулює розвиток нових технологій, методологій DevSecOps та архітектурних підходів, спрямованих на підвищення стабільності, безпеки й керованості розподілених систем.

Сучасні тенденції та перспективи розвитку

У 2025 році serverless-архітектури посіли провідне місце серед хмарних технологій, ставши ключовим етапом еволюції розподілених обчислень. У цій моделі інфраструктура повністю управляється хмарним провайдером, тоді як розробники зосереджуються виключно на бізнес-логіці, реалізованій у вигляді функцій (Function as a Service, FaaS). Провідні платформи – AWS Lambda, Azure Functions та Google Cloud Functions – забезпечують автоматичне масштабування, високу доступність і модель оплати pay-per-use. Інтеграція serverless із edge computing сприяє розгортанню функцій ближче до користувача (наприклад, Cloudflare Workers, Lambda@Edge), що мінімізує затримки обробки даних і підвищує ефективність для IoT-та інтерактивних застосунків.

Event-driven архітектури (EDA) поступово стають стандартом для побудови слабко зв'язаних систем, у яких взаємодія компонентів здійснюється через обмін подіями у event bus. Сучасні реалізації базуються на технологіях Apache Kafka, RabbitMQ, а також на хмарних сервісах – AWS EventBridge, Azure Event Grid. Поєднання EDA із патернами Event Sourcing, CQRS та Saga забезпечує масштабованість та стійкість до збоїв, однак породжує виклики, пов'язані з налагодженням систем і підтриманням eventual consistency. Ефективна реалізація таких архітектур потребує стабільної інфраструктури для управління потоками подій і гарантій доставки повідомлень.

Service Mesh формується як окремий інфраструктурний шар, що забезпечує безпечну та керовану комунікацію між мікросервісами за допомогою sidecar proxy. Провідні рішення – Istio (з компонентами Envoy, traffic management, mTLS, телеметрією), Linkerd (легковагове рішення на Rust) та Consul Connect, який підтримує як Kubernetes, так і віртуальні машини. Ключові можливості Service Mesh охоплюють інтелектуальну маршрутизацію, автоматичне шифрування (mTLS), збирання метрик і реалізацію механізмів відмовостійкості. На сьогодні ця технологія стала стандартом для production-ready систем із підвищеними вимогами до безпеки та керованості.

Edge computing активно інтегрується у сферу IoT, автономних транспортних систем, AR/VR та промислової автоматизації. Його метою є обробка даних безпосередньо на периферії мережі, що знижує затримки та зменшує навантаження на центральні сервери. Основні моделі включають Fog Computing, Mobile Edge Computing (5G), CDN Edge і On-Premise Edge. Інтеграція з мікросервісною архітектурою через KubeEdge, AWS Greengrass, Azure IoT Edge сприяє розвитку концепцій Industry 4.0 та розумних міст (Smart Cities).

Паралельно розвивається підхід Infrastructure as Code (IaC), який досяг зрілості завдяки інструментам Terraform, Ansible, Pulumi. Методологія GitOps (ArgoCD, FluxCD) використовує Git як єдине джерело істини (single source of truth), тоді як Policy as Code (OPA, Sentinel) забезпечує автоматичну перевірку дотримання стандартів безпеки та комплаєнсу. Новим напрямом є впровадження штучного інтелекту у сферу DevOps – AIOps – для аналізу операційних даних, прогнозування збоїв і автоматичного масштабування систем. Використання Deep Reinforcement Learning та Graph Neural Networks оптимізує топологію мікросервісів, тоді як машинне навчання у моніторингу дає змогу виявляти аномалії в реальному часі. Великі мовні моделі (наприклад, GitHub Copilot) підвищують продуктивність команд, сприяючи створенню self-healing систем і автоматизації тестування.

Висновок

Аналіз еволюції архітектур програмного забезпечення засвідчив закономірний рух від централізованих та жорстко інтегрованих систем до гнучких, масштабованих і динамічних моделей. Монолітна архітектура, що колись забезпечувала простоту та надійність, у сучасних умовах виявила низку обмежень, які були поступово подолані завдяки появі клієнт-серверних, багаторівневих, розподілених об'єктних систем, web-сервісів, REST та сервіс-орієнтованих підходів. Кожен етап еволюції сприяв формуванню концепцій незалежності компонентів, стандартизації інтерфейсів та розподіленої обробки даних.

Мікросервісна архітектура стала відповіддю на виклики цифрової трансформації, оскільки поєднує незалежність компонентів, можливість гнучкого масштабування та інтеграцію з сучасними практиками розробки (DevOps, CI/CD). Підтримка контейнеризації, оркестрації та Service Mesh технологій

забезпечує ефективне управління складними розподіленими системами. Разом із тим мікросервісна архітектура породжує нові виклики, пов'язані з управлінням інфраструктурою, узгодженістю даних та безпекою міжсервісної комунікації. Проте перехід до мікросервісів забезпечує організаціям конкурентні переваги через швидкість інновацій, автономність команд та здатність до оперативного масштабування відповідно до бізнес-потреб.

Подальший розвиток архітектур програмного забезпечення пов'язаний із поширенням serverless-підходів та event-driven моделей, впровадженням Service Mesh для надійної комунікації, інтеграцією edge computing для обробки даних на периферії мережі, автоматизацією інфраструктури через Infrastructure as Code та застосуванням штучного інтелекту для інтелектуальної оптимізації систем (AIOps). Ці тенденції відкривають нові горизонти для створення високопродуктивних, надійних та адаптивних обчислювальних систем із властивостями self-healing, здатних автономно реагувати на зміни навантаження та вимоги сучасного цифрового бізнес-середовища.

Література

1. Гавриляк А. В., Ліщук К. І. Архітектурні шаблони та стилі програмного забезпечення. *Вісник НТУУ «КПІ». Серія: Інформатика та обчислювальна техніка*. 2020. № 73. С. 47–54.

2. Киселевич В., Усата О., Сікора Я., Вербівський Д., Іванов Д. Мікросервісна архітектура: переваги та недоліки її практичного застосування. *Інформаційні технології та комп'ютерна інженерія*. 2024. № 2 (63). С. 45–54. DOI: https://doi.org/10.32782/IT/2024-2-7.

3. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol, CA : O'Reilly Media, 2021. 612 p.

4. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island, NY : Manning Publications, 2018. 520 p.

5. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol, CA : O'Reilly Media, 2017. 616 p.

6. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston, MA : Addison-Wesley Professional, 2021. 464 p.

7. Waseem M., Liang P., Shahin M. A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software*. 2020. Vol. 170. Article 110798. DOI: https://doi.org/10.1016/j.jss.2020.110798.

8. Taibi D., Lenarduzzi V., Pahl C. Microservices Anti-patterns: A Taxonomy. In: Microservices: Science and Engineering. Cham : Springer, 2020. P. 111–128. DOI: https://doi.org/10.1007/978-3-030-31646-4_5.

References

1. Havryliak A. V., Lishchuk K. I. (2020). Arkhitekturni shablony ta styli prohramnoho zabezpechennia [Architectural patterns and software design styles]. Visnyk NTUU "KPI". *Informatyka ta obchysliuvalna tekhnika [Bulletin of NTUU "KPI". Informatics and Computer Engineering]*. (73). 47–54 [in Ukrainian].

2. Kyselevych V., Usata O., Sikora Ya., Verbivskyi D., Ivanov D. (2024). Mikroservisna arkhitektura: perevahy ta nedoliky yii praktychnoho zastosuvannia [Microservice architecture: Advantages and disadvantages of its practical implementation]. *Informatsiini tekhnolohii ta komp'uterna inzheneriia [Information Technologies and Computer Engineering]*. 2 (63). 45–54. DOI: https://doi.org/10.32782/IT/2024-2-7 [in Ukrainian].

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

4. Richardson C. (2018). Microservices patterns: With examples in Java. Shelter Island, NY : Manning Publications.

5. Kleppmann M. (2017). Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. Sebastopol, CA : O'Reilly Media.

6. Bass L., Clements P., Kazman R. (2021). Software architecture in practice (4th ed.). Boston, MA : Addison-Wesley Professional.

7. Waseem M., Liang P., Shahin M. (2020). A systematic mapping study on microservices architecture in DevOps. J*ournal of Systems and Software*. 170. 110798. DOI: https://doi.org/10.1016/j.jss.2020.110798.

8. Taibi D., Lenarduzzi V., Pahl C. (2020). Microservices anti-patterns: A taxonomy. In Microservices: Science and engineering (pp. 111–128). Cham : Springer. DOI: https://doi.org/10.1007/978-3-030-31646-4_5.

Дата першого надходження статті до видання: 20.10.2025
Дата прийняття статті до друку після рецензування: 21.11.2025
Дата публікації (оприлюднення) статті: 26.12.2025