ТРАНСФОРМАЦІЯ ПАРАДИГМ ПРОГРАМНОЇ АРХІТЕКТУРИ: ПОРІВНЯЛЬНИЙ АНАЛІЗ МОНОЛІТНИХ, МІКРОСЕРВІСНИХ ТА БЕЗСЕРВЕРНИХ РІШЕНЬ

Мельник Анна Віталіївна
кандидат педагогічних наук,
доцент кафедри комп'ютерних наук та інформаційних технологій
ЖДУ імені Івана Франка

Еволюція архітектурних парадигм у розробці програмного забезпечення (ПЗ) відображає постійний пошук балансу між швидкістю виходу на ринок (Time-to-Market), масштабованістю та вартістю експлуатації. Станом на 2025–2026 роки індустрія перебуває у стані критичного переосмислення попереднього десятиліття, яке пройшло під диктатурою мікросервісного підходу. Сучасний етап характеризується переходом від догматичного впровадження розподілених систем до прагматичного вибору архітектури, що базується на об'єктивних даних про продуктивність та Total Cost of Ownership (TCO) [1]. Трансформація від класичних монолітів до мікросервісів, а згодом до безсерверних (serverless) обчислень та зворотна консолідація у модульні моноліти свідчать про досягнення галузевої зрілості, де архітектурні рішення стають стратегічним інструментом бізнесу, а не лише технічним вибором [1].

Історично монолітна архітектура була стандартом де-факто, забезпечуючи цілісність додатку через єдину кодову базу та спільне середовище виконання. Проте зі зростанням складності систем та вимог до їх гнучкості, традиційні моноліти почали демонструвати ознаки «технологічної інертності» [4]. У відповідь на це мікросервісна парадигма запропонувала декомпозицію системи на малі, незалежно розгорнуті компоненти, що дозволило командам працювати автономно та використовувати різні технологічні стеки [6]. Паралельно з цим розвиток хмарних технологій призвів до появи безсерверної архітектури, яка максимально абстрагує управління інфраструктурою, дозволяючи розробникам зосередитися виключно на бізнес-логіці у форматі Function-as-a-Service (FaaS) [8].

Монолітна архітектура базується на принципі об'єднання всіх функціональних рівнів додатку — інтерфейсу користувача, бізнес-логіки та доступу до даних — у межах одного розгортального модуля. Основною перевагою такої структури є простота розробки, тестування та моніторингу на початкових етапах життєвого циклу проекту [9]. Оскільки всі компоненти працюють в одному процесі, взаємодія між ними відбувається через виклики функцій у пам'яті (in-memory calls), що забезпечує мінімально можливу затримку (latency) [1].

Проте з часом моноліти стикаються з проблемою «великої кулі бруду» (Big

Ball of Mud), коли межі між модулями розмиваються через тісну зв'язність (tight coupling) [4]. Це призводить до того, що зміна в одній частині системи вимагає повного перетестування та розгортання всього додатку, що критично сповільнює релізи. Вертикальне масштабування моноліту (додавання ресурсів до одного сервера) має свої фізичні та фінансові межі, а горизонтальне масштабування вимагає реплікації всього стеку, що є неефективним з точки зору споживання ресурсів, особливо якщо високе навантаження припадає лише на окремий функціональний блок [5].

Станом на 2025 рік спостерігається трансформація моноліту в «модульний моноліт». Це архітектурний стиль, який зберігає переваги одного розгортального модуля, але суворо впроваджує логічне розділення компонентів через чітко визначені інтерфейси та доменні кордони (Bounded Contexts) [1]. Модульні моноліти дозволяють командам володіти окремими частинами коду без складності розподілених систем, забезпечуючи високу швидкість налагодження (debugging), яка, за даними DZone (2024), у монолітах на 35% вища, ніж у мікросервісах [1].

Таблиця 1. Еволюція характеристик монолітної архітектури

Характеристика	**Традиційний моноліт (2010)**	**Модульний моноліт (2025)**
Структура коду	"Спагеті-код, тісна зв'язність"	"Чіткі модулі, Hexagonal Architecture"
Управління даними	Спільна БД для всіх функцій	Логічне розділення схем за доменами
Розгортання	"Ризиковане, тривале"	"Автоматизоване, швидке"
Масштабування	Тільки всього додатку	Клонування модуля з параметризацією
Затримка (Latency)	Наносекунди (in-memory)	Наносекунди (in-memory)
Когнітивне навантаження	Високе (все в одному)	Низьке (ізоляція знань)

Мікросервісна архітектура (MSA) декомпозує додаток на набір дрібних сервісів, кожен з яких виконує конкретну бізнес-задачу та має власне сховище даних [6]. Ця парадигма отримала масове поширення завдяки успішним кейсам Netflix та Amazon, які змогли масштабувати свої операції до глобального рівня

[13]. Основною цінністю MSA є автономія: команди можуть незалежно обирати мови програмування (Java, Go, Rust), бази даних (NoSQL, SQL) та графіки релізів [3].

Однак реалізація мікросервісів впроваджує так званий «мікросервісний податок» (Microservices Premium). Основною проблемою стає мережева взаємодія. Виклики між сервісами (через REST, gRPC або повідомлення) тривають мілісекунди, що в 1 000 000 разів довше, ніж виклики в пам'яті процесу [1]. Це створює каскадні затримки, які важко діагностувати. Крім того, децентралізація даних вимагає складних механізмів підтримки консистентності, таких як патерн Saga [15].

Saga представляє собою послідовність локальних транзакцій, де кожен крок оновлює базу даних та генерує подію для наступного кроку. У разі збою система повинна виконати компенсаційні транзакції для відкату попередніх змін [15]. Хоча цей патерн забезпечує консистентність без використання важких розподілених транзакцій (2PC), він значно ускладнює бізнес-логіку та моніторинг станів системи [16].

У 2025 році з’ясувалося, що багато організацій переоцінили свої потреби в мікросервісах. Згідно з опитуванням CNCF (2025), використання сервісних сіток (service mesh) впало з 18% у 2023 році до 8% у 2025 році [1]. Це свідчить про «втому від інфраструктури» та прагнення до спрощення систем [1].

Таблиця 2. Технологічні виклики мікросервісної архітектури

Проблема	**Опис механізму**	**Наслідки для бізнесу**
Мережевий оверхед	Передача даних по HTTP/gRPC замість RAM	Збільшення затримки відповіді (Latency)
Складність транзакцій	Відсутність ACID у розподіленому середовищі	Ризик втрати цілісності даних
Observability	Потреба в розподіленому трасуванні	Висока вартість моніторингу та логування
Залежності	"Розподілений моноліт" при неправильному дизайні	Блокування релізів через інші команди
Безпека	Кожен сервіс — точка атаки (Zero Trust)	Збільшення витрат на кіберзахист

Безсерверні обчислення (Serverless) пропонують модель, де провайдер хмарних послуг (CSP) повністю бере на себе управління життєвим циклом інфраструктури. Основні компоненти Serverless — це Function-as-a-Service (FaaS) та Backend-as-a-Service (BaaS) [8]. У цій моделі код виконується лише у відповідь на подію, а масштабування від нуля до тисяч примірників відбувається автоматично [8].

Економічна модель Serverless базується на принципі Pay-per-use. Компанія сплачує лише за мілісекунди виконання коду, що дозволяє радикально знизити витрати для систем з нерівномірним навантаженням. Наприклад, міграція середніх та великих підприємств на Serverless дозволяє знизити щомісячні витрати на обчислення з $42,000 до $11,340 [19].

Проте Serverless має суттєві обмеження, головним з яких є «холодний старт» (Cold Start) [8]. Коли функція викликається після періоду бездіяльності, хмарному провайдеру потрібно кілька сотень мілісекунд на ініціалізацію середовища [8]. Станом на 2025 рік час холодного старту в Google Cloud Functions та AWS Lambda вдалося стабілізувати на рівні до 400 мс для більшості середовищ виконання [19].

Для складних процесів використовується оркестрація функцій (наприклад, AWS Step Functions). Проте, досвід Amazon Prime Video показав, що при високому масштабі вартість оркестрації може стати непомірною, що робить Serverless менш придатним для потокової обробки даних великого обсягу [21].

Таблиця 3. Порівняння моделей витрат та продуктивності

Параметр	**Мікросервіси (Контейнери)**	**Безсерверні рішення (FaaS)**
Модель оплати	За зарезервовану потужність (Provisioned)	За фактичне споживання (Pay-per-use)
Час ініціалізації	Секунди (запуск пода в K8s)	Мілісекунди (Warm/Cold start)
Масштабування	На основі метрик (CPU/RAM)	Подієво-орієнтоване (автоматичне)
Обмеження часу	Необмежено	Зазвичай до 15 хвилин
Макс. пропускна здатність	Залежить від кластера	До 250,000 запусків за 22с

Період 2023–2025 років увійшов в історію як час «великої консолідації». Провідні компанії почали відмовлятися від надмірної декомпозиції. Найбільш резонансним став кейс Amazon Prime Video, де розподілену систему замінили одним монолітним додатком у контейнерах ECS [23].

Причиною змін у Prime Video стали вартість оркестрації та передачі даних [21]. Перенісши логіку в один процес, команда забезпечила передачу даних через оперативну пам'яті, що знизило витрати на 90% та покращило масштабованість [22]. Аналогічну трансформацію пройшла компанія Segment, яка консолідувала понад 140 мікросервісів у один моноліт [13]. Основною проблемою Segment була «голова лінійного блокування» (Head-of-Line Blocking) [26]. Після переходу до моноліту продуктивність розробки зросла [25]. Ці кейси підтверджують, що для команд менше 20 розробників мікросервісна архітектура рідко є виправданою [1].

Таблиця 4. Аналіз випадків консолідації (Microservices to Monolith)

Компанія / Продукт	**Початкова архітектура**	**Причина консолідації**	**Результат**
Amazon Prime Video	Serverless / Step Functions	Вартість оркестрації та S3	-90% витрат, стабільність
Segment	140+ Мікросервісів	Складність управління чергами	Покращення TTM, спрощення
InVision	Мікросервіси	Латентність комунікацій	Зменшення коорд. пекла
Istio Control Plane	Мікросервісний Control Plane	Операційна складність	Спрощення інсталяції

У моноліті безпека часто базується на периметрі [28]. У мікросервісах та Serverless кожна функція має бути ізольованою (Zero Trust) [3]. Станом на 2024–2025 роки понад 90% контейнерних образів мають відомі вразливості [28]. Це вимагає впровадження DevSecOps-практик [18].

Для забезпечення надійності в розподілених системах критично важливими є патерни:

- **Circuit Breaker (Запобіжник)**: запобігає каскадним збоям [3].
- **Retry (Повторні спроби)**: повторює запит при помилці мережі [15].
- **Timeout (Таймаут)**: обмежує час очікування відповіді [14].

У 2025–2026 роках AI-агенти стають активними учасниками управління мікросервісами, автономно оптимізуючи комунікації та виявляючи аномалії [4]. Іншим напрямком є периферійні обчислення (Edge Computing) [3]. Міграція

функцій на «периферію» мережі зменшує затримку до одиниць мілісекунд [14]. Українські науковці (Шафоренко С., Авріята А., Артеменко О.) підкреслюють потребу у нових моделях планування функцій на кордоні мережі [3].

Висновки

Вибір архітектури більше не є питанням моди. Сучасний архітектор керується принципом «Right Tool for the Right Job» [2]. Монолітна архітектура залишається оптимальною для систем, що потребують максимальної продуктивності на ранніх етапах. Мікросервіси безальтернативні для глобальних систем, але вимагають величезних інвестицій. Безсерверні рішення забезпечують швидкість, проте їх економічна ефективність нівелюється при стабільно високому навантаженні [1].

Список літератури

1. Understanding Modern Software Architecture - From Microservices Consolidation to Modular Monoliths. Softwareseni. URL: https://www.softwareseni.com/understanding-modern-software-architecture-from-microservices-consolidation-to-modular-monoliths/ (дата звернення: 15.05.2026).

2. Kumar S. The Right Architecture at the Right Time: Monolith vs. Microservices vs. Serverless. Medium. 2026. URL: https://skphd.medium.com/the-right-architecture-at-the-right-time-monolith-vs-microservices-vs-serverless-396b9f00d0c6 (дата звернення: 15.05.2026).

3. Microservices, Serverless and Edge-Computing: Architectural Transformation of Software. ResearchGate. URL: https://www.researchgate.net/publication/399946526_MICROSERVICES_SERVERLESS_AND_EDGE-COMPUTING_ARCHITECTURAL_TRANSFORMATION_OF_SOFTWARE (дата звернення: 1.05.2026).

4. The Evolution and Future of Microservices Architecture with AI-Driven Enhancements. Digital Commons @ Lindenwood University. URL: https://digitalcommons.lindenwood.edu/cgi/viewcontent.cgi?article=1725&context=faculty-research-papers (дата звернення: 15.05.2026).

5. Microservices vs. Monoliths: Comparative Analysis for Scalable Software Architecture Design. ResearchGate. URL: https://www.researchgate.net/publication/387645461_Microservices_vs_Monoliths_Comparative_Analysis_for_Scalable_Software_Architecture_Design (дата звернення: 5.05.2026).

6. The Evolution and Future of Microservices Architecture with AI-Driven Enhancements. IJRES Online. URL: https://ijresonline.com/archives/ijres-v12i1p103 (дата звернення: 15.05.2026).

7. Monolithic vs Microservices - Difference Between Software Development Architectures. AWS. URL: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/ (дата звернення: 15.02.2026).

8. Serverless Architecture and Its Current State of the Art: A Systematic Literature Review. Preprints.org. 2025. URL:

https://www.preprints.org/manuscript/202512.0219 (дата звернення: 4.05.2026).

9. Monoliths vs Microservices vs Serverless. Harness Blog. URL: https://www.harness.io/blog/monoliths-vs-microservices-vs-serverless (дата звернення: 12.05.2026).

10. Що краще моноліт чи мікросервіси? Як обрати архітектуру проєкту? IAMPM. URL: https://iampm.club/ua/blog/shho-krashhe-monolit-chi-mikroservisi-yak-obrati-arhitekturu-projektu/ (дата звернення: 7.05.2026).

11. Microservices and Serverless Computing: Architectural Patterns for Modern Software Applications. Sarcouncil Journal of Engineering and Computer Sciences. 2025. URL: https://sarcouncil.com/download-article/SJECS-320-2025-199-205.pdf (дата звернення: 15.05.2026).

12. Evolution of Microservices Patterns for Designing Hyper-Scalable Cloud-Native Architectures. ResearchGate. URL: https://www.researchgate.net/publication/394445303_Evolution_of_Microservices_Patterns_for_Designing_Hyper-_Scalable_Cloud-Native_Architectures (дата звернення: 15.05.2026).

13. Verma P. When Monoliths Triumph: Case Studies of Reversing Microservices Adoption. Medium. URL: https://medium.com/@princevermasrcc/when-monoliths-triumph-case-studies-of-reversing-microservices-adoption-6f0bb3fd055e (дата звернення: 15.05.2026).

14. Architecture of multifunctional application for data protection based on microservice approach. Journals of State University of Telecommunications. URL: https://journals.dut.edu.ua/index.php/dataprotect/article/download/3000/2898/ (дата звернення: 15.05.2026).

15. Saga Design Pattern. Azure Architecture Center | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/saga (дата звернення: 15.05.2026).

16. Pattern: Saga. Microservices.io. URL: https://microservices.io/patterns/data/saga.html (дата звернення: 15.05.2026).

17. Saga pattern is specific to microservices rather than distributed systems. Dev.to. URL: https://dev.to/siy/comment/270gp (дата звернення: 15.05.2026).

18. Serverless vs. microservices: Which architecture is best for your application? IBM. URL: https://www.ibm.com/think/topics/serverless-vs-microservices (дата звернення: 15.05.2026).

19. Serverless computing and advanced security framework integration: From implementation to future trends. IJSRA. 2025. URL: https://ijsra.net/sites/default/files/fulltext_pdf/IJSRA-2025-0267.pdf (дата звернення: 15.05.2026).

20. The Future of Serverless Architectures in Data Engineering. IJAIBD CMS. URL: https://ijaibdcms.org/index.php/ijaibdcms/article/view/360 (дата звернення: 15.05.2026).

21. Prime Video, from serverless to monolith: a retrospective. Blog Devoteam Rebirth. 2023. URL: https://rebirth.devoteam.com/2023/07/13/prime-video-from-serverless-to-monolith/ (дата звернення: 15.05.2026).

22. Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. WUDSN. 2023. URL: https://www.wudsn.com/productions/www/site/news/2023/2023-05-08-microservices-01.pdf (дата звернення: 15.05.2026).

23. Amazon Prime Video Monitoring Service. ByteByteGo. URL: https://bytebytego.com/guides/amazon-prime-video-monitoring-service/ (дата звернення: 15.05.2026).

24. Wimalasuriya I. Amazon Prime Video's 90% Cost Reduction through moving to Monolithic. DEV Community. URL: https://dev.to/indika_wimalasuriya/amazon-prime-videos-90-cost-reduction-throuh-moving-to-monolithic-k4a (дата звернення: 15.05.2026).

25. Why Twilio Segment moved from microservices back to a monolith. Hacker News. URL: https://news.ycombinator.com/item?id=46257714 (дата звернення: 13.05.2026).

26. Back to the Future: From Microservice to Monolith. Smartpoint. URL: https://www.smartpoint.fr/wp-content/uploads/Back-to-the-Future-From-Microservice-to-Monolith.pdf (дата звернення: 12.05.2026).

27. Su. Back to the Future: From Microservice to Monolith. Conference on Microservices. 2023. URL: https://www.conf-micro.services/2023/papers/06-su-back-to-the-monolith.pdf (дата звернення: 15.05.2026).

28. Огляд мікросервісної архітектури та аналіз типових вразливостей. Львівська політехніка. 2025. URL: https://science.lpnu.ua/sites/default/files/journal-paper/2025/nov/40835/vse-116-123.pdf (дата звернення: 6.05.2026).

29. Identifying and Architecting Microservices for Edge Computing. ResearchGate. URL: https://www.researchgate.net/publication/392253438_Identifying_and_Architecting_Microservices_for_Edge_Computing (дата звернення: 6.05.2026).

30. Prime Video Sparks Serverless Debate by Switching to Monolith. Virtualization Review. 2023. URL: https://virtualizationreview.com/articles/2023/05/08/serverless-vs-monolith.aspx (дата звернення: 02.05.2026).