КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023
DOI 10.28925/2663-4023.2026.33.1233
УДК 004.415.2:004.75

Горпенюк Андрій Ярославович
к.т.н., доцент, доцент кафедри захисту інформації
Національний університет «Львівська Політехніка», Львів, Україна
ORCID: 0000-0001-5821-2186
andrii.y.horpeniuk@lpnu.ua

Лужецька Наталія Миколаївна
старший викладач кафедри захисту інформації
Національний університет «Львівська Політехніка», Львів, Україна
ORCID: 0000-0002-5449-5825
nataliia.m.luzhetska@lpnu.ua

Горпенюк Микола Андрійович
студент кафедри захисту інформації
Національний університет «Львівська Політехніка», Львів, Україна
ORCID: 0009-0001-1577-2068
mykola.horpeniuk.kb.2023@lpnu.ua

Горпенюк Олександр Андрійович
студент кафедри захисту інформації
Національний університет «Львівська Політехніка», Львів, Україна
ORCID: 0009-0000-9478-6283
oleksandr.horpeniuk.kb.2024@lpnu.ua

ІНТЕГРОВАНИЙ ПІДХІД ДО ОЦІНЮВАННЯ PAAS І FAAS АРХІТЕКТУР У ВИСОКОНАВАНТАЖЕНИХ СИСТЕМАХ

Анотація. У роботі представлено комплексне дослідження архітектурних шаблонів для побудови високонавантажених веб-систем у хмарному середовищі. Проведено порівняльний аналіз класичної монолітної архітектури (PaaS) та безсерверної архітектури (FaaS) на основі критеріїв продуктивності, економічної ефективності та інформаційної безпеки. За допомогою Amazon Web Services (AWS) було розроблено та протестовано два функціональні прототипи системи обробки замовлень. Експериментально встановлено, що безсерверна архітектура дозволяє знизити експлуатаційні витрати на 75–99% для систем з нерівномірним трафіком, але вносить додаткову затримку «холодного старту» до 1,42 секунди. Запропоновано матрицю прийняття рішень щодо вибору технологічного стеку залежно від бізнес-вимог та профілю загроз.

Ключові слова: хмарні обчислення, безсерверні технології, FaaS, монолітна архітектура, AWS Lambda, мікросервіси, STRIDE, FinOps, навантажувальне тестування.

ВСТУП

Сьогодні сучасна індустрія розроблення програмного забезпечення переживає фундаментальну трансформацію, пов'язану з масовим переходом від використання власних фізичних серверів (On-Premise) до хмарних моделей розгортання. Перехід від IaaS (Infrastructure as a Service) до PaaS (Platform as a Service) дозволила компаніям змінити структуру витрат з капітальних на операційні, фокусуючись на бізнес-логіці, а не на адмініструванні обладнання. Разом з тим, незважаючи на доступність хмарних ресурсів, розробники стикаються з критичною проблемою вибору оптимального архітектурного шаблону для високонавантажених систем.

Традиційна монолітна архітектура протягом тривалого часу залишалася стандартом де-факто. Її перевагами є простота розробки, налагодження та атомарність розгортання, що робить її привабливою на початкових етапах життєвого циклу продукту. Однак у хмарному середовищі моноліт демонструє суттєві обмеження: низьку гнучкість масштабування та необхідність оплати за зарезервовані потужності (Idle Time), навіть коли система не обробляє запити користувачів [2].

Як відповідь на ці виклики виникла концепція безсерверних обчислень (Serverless), або FaaS. Ця

модель пропонує революційний підхід до тарифікації Pay-as-you-go, де оплата стягується виключно за час виконання коду з точністю до мілісекунд [3]. Serverless забезпечує автоматичне масштабування та зниження операційних витрат, що забезпечується методологією Cloud FinOps [4]. Водночас, перехід до FaaS супроводжується специфічними технічними проблемами, які часто ігнорують: явищем «холодного старту» (latency latency), складністю управління розподіленим станом та рядом нових кіберзагроз [5, 6].

Наукові дослідження вказують на те, що продуктивність безсерверних систем суттєво залежить від характеру навантаження та обраного середовища виконання (Runtime) [7]. Проте в існуючій літературі недостатньо комплексних емпіричних порівнянь, які б враховували не лише технічні характеристики (час відгуку, пропускну здатність), але й економічну ефективність та аспекти інформаційної безпеки в контексті реальних бізнес-сценаріїв. Зокрема, питання захисту FaaS-додатків вимагає перегляду класичних моделей загроз, оскільки периметр захисту зміщується з мережевого рівня на рівень ідентичності та управління доступом [8, 9].

Постановка проблеми. Як показує проведений в роботі аналіз, на продуктивність безсерверних систем істотно впливає характер навантаження та обраного середовища виконання. В опублікованих працях недостатньо комплексних емпіричних порівнянь, які поряд з технічними характеристиками (час відгуку, пропускна здатність) враховували б також економічну ефективність та аспекти інформаційної безпеки в реальних умовах застосування. Для прикладу, питання захисту FaaS-додатків вимагає перегляду класичних моделей загроз, оскільки периметр захисту зміщується з мережевого рівня на рівень ідентичності та управління доступом. Таким чином, існує невирішена проблема вибору архітектури, яка потребує комплексного порівняння не лише на рівні теоретичних моделей, а й на базі експериментальних даних для конкретних бізнес-сценаріїв.

Аналіз останніх досліджень і публікацій. Аналіз науково-технічної літератури свідчить про те, що розвиток архітектури програмного забезпечення нерозривно пов'язаний з еволюцією інфраструктури. Традиційний підхід до розгортання монолітних додатків на фізичних серверах або віртуальних машинах (модель IaaS) характеризується високою стабільністю, але складністю утилізації ресурсів та складністю горизонтального масштабування окремих компонентів системи.

Поява контейнеризації та оркестраторів (Kubernetes) частково вирішила проблему ізоляції та утилізації ресурсів, проте залишила за розробниками значний тягар операційного управління (Ops). Наступним логічним кроком стала поява безсерверних обчислень (Serverless), які пропонують нову парадигму розробки, де управління серверами повністю абстраговане від розробника [18].

Аналіз опублікованих результатів досліджень дозволяє виділити два мінуси Serverless-підходу:

1. Продуктивність: дослідники у своїх експериментах вказують на варіативність часу відгуку та проблему «холодного старту», яка може сягати 1-2 секунд для мов з віртуальною машиною (Java, C#) або інтерпретованих мов (Python) [5, 7].

2. Економічна доцільність: Хоча модель Pay-as-you-go виглядає привабливо, дослідники застерігають, що при стабільному високому навантаженні вартість FaaS може лінійно зростати і перевищити вартість оренди виділених серверів [14].

Проведений аналіз дозволяє констатувати актуальність розв’язання задачі вибору архітектури, яка потребує комплексного порівняння не лише на рівні теоретичних моделей, а й на базі експериментальних даних для конкретних бізнес-сценаріїв.

Метою статті є комплексне дослідження архітектурних шаблонів для побудови високонавантажених веб-систем у хмарному середовищі.

Завданнями дослідження є:

1. Порівняльний аналіз класичної монолітної архітектури (PaaS) та безсерверної архітектури (FaaS) на основі критеріїв продуктивності, економічної ефективності та інформаційної безпеки. Об'єкти порівняльного аналізу: Архітектура А (Monolith): Реалізація на базі фреймворку Flask (Python) з використанням реляційної БД PostgreSQL, розгортання за моделлю PaaS (AWS Elastic Beanstalk); Архітектура Б (Serverless): Реалізація на базі AWS Lambda (Python) з використанням NoSQL БД DynamoDB, розгортання за моделлю FaaS.

2. Розроблення та тестування двох функціональних прототипів системи обробки замовлень за допомогою Amazon Web Services (AWS). Для верифікації теоретичних гіпотез необхідно розробити та дослідити дві функціонально ідентичні реалізації веб-сервісу обробки замовлень (Order Processing System). При цьому система повинна надавати REST API для виконання операцій:

- Створення замовлення (`POST /orders`): валідація даних, розрахунок вартості, збереження в БД.
- Отримання статусу замовлення (`GET /orders/{id}`).

Розроблені прототипи систем обробки замовлень повинні забезпечувати час відгуку не більший за 200 мс. для 95% запитів (p95). Системи мають витримувати пікове навантаження («вибуховий трафік») до

1000 запитів/сек без деградації сервісу (HTTP 5xx < 1%). Також необхідно забезпечити шифрування даних у стані спокою та передачі та відповідність принципам найменших привілеїв [8].

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Математичне моделювання вартості та продуктивності.

Для об'єктивного порівняння архітектур необхідно формалізувати критерії оцінки, оскільки пряме зіставлення абсолютних показників без урахування профілю навантаження може призвести до хибних висновків. У даній роботі ми пропонуємо математичні моделі для розрахунку сукупної вартості володіння (TCO — Total Cost of Ownership) та затримок обробки запитів, які враховують специфіку тарифікації хмарних провайдерів та стохастичний характер вхідного трафіку. Розроблені моделі дозволяють спрогнозувати поведінку системи при масштабуванні та визначити граничні умови економічної доцільності переходу на FaaS.

Модель оцінювання вартості хмарних ресурсів

Вартість експлуатації хмарної системи C_{total} є сумою витрат на обчислювальні потужності ($C_{compute}$), зберігання даних ($C_{storage}$) та передачу мережевого трафіку (C_{net}). При цьому ключова відмінність між досліджуваними архітектурами полягає у формуванні складової $C_{compute}$.

Для монолітної архітектури (PaaS), розгорнутої на віртуальних машинах (EC2), функція вартості має східчастий вигляд. Витрати визначаються часом резервування інстансу, незалежно від корисного навантаження (CPU Utilization):

$$C_{mono} = T_{month} \cdot \left(\sum_{i=1}^{N_{vm}} P_{vm}^{(i)} + P_{lb} + P_{rds} \right)$$

де:

T_{month} — кількість годин у розрахунковому періоді (730 годин);

N_{vm} — кількість активних віртуальних машин (у кластері);

$P_{vm}^{(i)}$ — погодинна ціна оренди i-го типу інстансу;

P_{lb} — фіксована ціна балансувальника навантаження (Load Balancer);

P_{rds} — ціна оренди виділеного сервера бази даних.

Головним недоліком цієї моделі є оплата за простій (Idle Time). Якщо реальне навантаження системи складає 20% від потужності CPU, то 80% бюджету витрачається неефективно (Over-provisioning).

Для безсерверної архітектури (FaaS) функція вартості є лінійною і залежить виключно від кількості та тривалості транзакцій:

$$C_{serverless} = Q \cdot P_{api} + \sum_{j=1}^{Q} \left(P_{req} + T_{exec}^{(j)} \cdot M \cdot P_{gb\_sec} \right) + C_{db}(Q)$$

де:

Q — загальна кількість запитів за місяць;

P_{api} — вартість обробки запиту API Gateway;

P_{req} — фіксована вартість виклику функції (Invocation cost);

$T_{exec}^{(j)}$ — час виконання j-го запиту (округлюється до мілісекунди);

M — обсяг виділеної оперативної пам'яті (у ГБ);

P_{gb_sec} — тариф за ГБ-секунду обчислень.

Ця модель реалізує стратегію Scale-to-Zero: при $Q \to 0$ вартість $C_{serverless} \to 0$. Однак при надвисоких навантаженнях вартість може перевищити фіксовану ціну моноліту через вищу питому вартість одиниці обчислень у FaaS.

Модель мережевих затримок (Latency Model)

Загальний час відгуку системи L_{total} (End-to-End Latency) є критичним показником якості обслуговування (QoS). Він складається з мережевої затримки, часу обробки бізнес-логіки та інфраструктурних накладних витрат.

Для FaaS-архітектури фундаментальною проблемою є недетермінованість часу ініціалізації середовища. Формулу затримки можна подати так:

$$L_{total} = L_{net} + L_{exec} + \delta(t) \cdot L_{init}$$

де:

L_{net} — час передачі даних мережею (RTT);

L_{exec} — час виконання коду («теплий старт»), який залежить від алгоритмічної складності;

$\delta(t)$ — бінарна функція стану системи в момент часу t.

L_{init} — час ініціалізації контейнера («холодний старт»), який включає час завантаження коду та час запуск інтерпретатора (Python Runtime);

Функція $\delta(t)$ приймає значення 1 (холодний старт), якщо з моменту останнього виклику функції пройшов час $\Delta t > T_{keep\_alive}$, або якщо відбувається горизонтальне масштабування (створення нових екземплярів функції при стрибку трафіку). В іншому випадку $\delta(t) = 0$.

Для монолітної архітектури, де додаток постійно завантажений у пам'ять, можна вважати, що $\delta(t) \approx 0$ для всіх запитів після старту сервера, що забезпечує стабільне значення $p99$ латентності.

Архітектура та програмна реалізація.

Для емпіричної перевірки розглянутих теоретичних моделей та отримання об'єктивних характеристик продуктивності було розроблено два функціональні прототипи системи обробки замовлень. Обидві системи реалізують ідентичний набір RESTful API методів (`POST /orders`, `GET /orders/{id}`) та використовують мову програмування Python 3.10. Використання єдиної мови дозволяє мінімізувати похибку, пов'язану з різницею в ефективності компіляторів або віртуальних машин [5].

Монолітна реалізація (PaaS Approach)

Перший прототип було побудовано за класичним трирівневим патерном (3-Tier Architecture) та розгорнуто з використанням моделі Platform-as-a-Service (PaaS). Такий підхід дозволив абстрагуватися від низькорівневого адміністрування операційної системи, делегувавши ці задачі сервісу AWS Elastic Beanstalk.

Компоненти системи:

- Рівень представлення (Load Balancer): Вхідною точкою в систему слугує Application Load Balancer (ALB), що працює на 7-му рівні моделі OSI. Окрім розподілу HTTP-запитів між активними інстансами за алгоритмом Round Robin, він виконує термінацію SSL/TLS трафіку, знімаючи обчислювальне навантаження з процесорів веб-серверів, а також здійснює регулярні перевірки працездатності (Health Checks) вузлів [9].
- Рівень бізнес-логіки (Application Server): Веб-додаток реалізовано мовою Python 3.10 на базі мікрофреймворку Flask. Як середовище виконання використано керовану платформу AWS Elastic Beanstalk, яка автоматично налаштовує групу віртуальних машин EC2 (тип віртуальної машини `t3.medium`: 2 vCPU, 4 GB RAM). Оскільки Flask є синхронним фреймворком, для обробки конкурентних запитів застосовано WSGI-сервер Gunicorn, налаштований у режимі prefork-worker, що дозволяє паралельно обробляти декілька запитів у межах одного сервера [11].
- Рівень даних (Database): Для забезпечення транзакційної цілісності (ACID) використано реляційну СКБД PostgreSQL 14, розгорнуту через керований сервіс Amazon RDS. Взаємодія додатку з базою здійснюється через ORM SQLAlchemy. Критично важливим архітектурним елементом тут є механізм пулу з'єднань (Connection Pooling), який утримує активні TCP-сесії, зменшуючи накладні витрати на "рукостискання" (handshake) при кожному запиті [4].

Алгоритм роботи: обробка запитів відбувається за синхронною блокуючою моделлю. При отриманні POST-запиту потік (або процес) веб-сервера блокується на час виконання транзакції запису в БД (I/O Blocking). Масштабування системи (Auto Scaling) налаштовано реактивно: тригер CloudWatch спрацьовує, якщо середнє використання CPU перевищує 70% протягом 5 хвилин. Це призводить до додавання нових інстансів із затримкою на «прогрів» системи (завантаження ОС та додатку), що складає 3–5 хвилин [5].

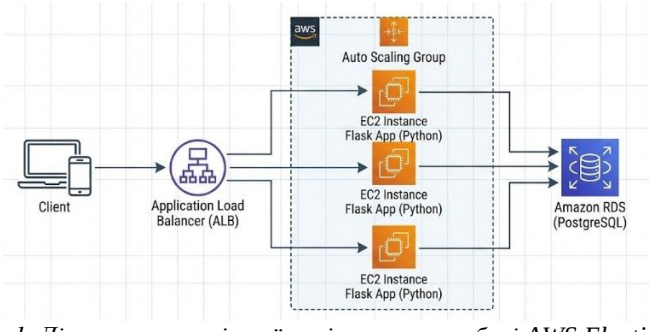

Рис. 1. Діаграма монолітної архітектури на базі AWS Elastic Beanstalk

Безсерверна реалізація (FaaS Approach)

Другий прототип реалізовано на базі нативної хмарної архітектури (Cloud Native), де управління інфраструктурою повністю делеговано хмарному провайдеру, а модель відповідальності зміщується в бік коду та конфігурації.

Компоненти системи:

- Рівень входу (Event Trigger): Amazon API Gateway діє як єдина точка входу, виконуючи функції зворотного проксі. Він не просто транслює запити, а трансформує вхідні HTTP-запити у JSON-події (Event Objects), валідує їх структуру та маршрутизує до відповідних обробників. Також на цьому рівні реалізовано механізми захисту від DDoS (throttling) [2].
- Рівень обчислень (Compute): Бізнес-логіка декомпонована на незалежні функції AWS Lambda згідно з патерном Microservices. Кожна функція (CreateOrder, GetOrder) є ізольованим stateless-контейнером із виділеною пам'яттю 128 МБ. Час життя контейнера обмежений часом обробки одного запиту, після чого він "заморожується" або знищується, що унеможливлює зберігання сесій у локальній пам'яті [3].
- Рівень даних (NoSQL): Для забезпечення низької латентності та атомарного масштабування використано базу даних Amazon DynamoDB у режимі "On-Demand". На відміну від реляційних баз, DynamoDB використовує HTTP API для операцій читання/запису, що дозволяє уникнути проблеми вичерпання ліміту з'єднань (connection exhaustion), яка є критичною для FaaS-середовища при високій конкурентності запитів [7].

Алгоритм роботи: Система функціонує за подієво-орієнтованою моделлю (Event-driven). Ресурси виділяються миттєво в момент надходження запиту (Just-in-Time allocation). Масштабування відбувається паралельно: на кожен вхідний запит платформа може запустити окремий екземпляр функції. Це дозволяє обробляти тисячі конкурентних користувачів без попереднього резервування потужностей. Однак, такий підхід вносить ризик «холодного старту» (Cold Start) — затримки на ініціалізацію середовища виконання та завантаження бібліотек при першому виклику функції [5, 12].

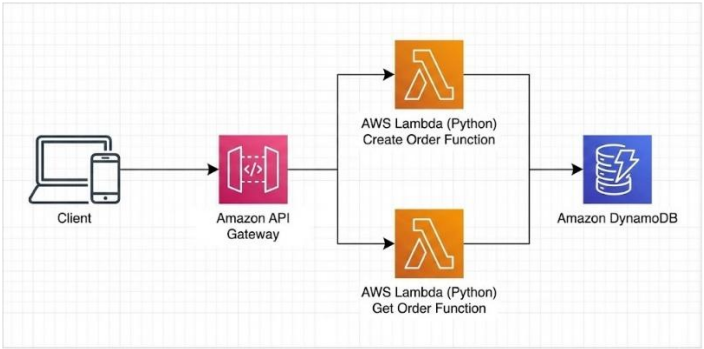

Рис. 2. Діаграма безсерверної архітектури (AWS API Gateaway + Lambda + DynamoDB)

Аналіз інформаційної безпеки.

Перехід від монолітної архітектури до безсерверної зумовлює фундаментальну зміну парадигми захисту інформації: від класичного захисту периметра (Perimeter Security) до захисту ресурсів та ідентичності (Zero Trust). У даній роботі для систематизації та порівняльного аналізу загроз використано методологію STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege), яка дозволяє ідентифікувати слабкі місця на етапі архітектурного проєктування.

У монолітній архітектурі (PaaS) система захисту будується за принципом «фортеці». Основним рубежем оборони є мережевий екран (Firewall/WAF) та балансувальник навантаження. Внутрішні компоненти системи (модулі додатку, локальні файли) часто мають високий рівень довіри один до одного. Компрометація периметра (наприклад, через RCE-вразливість) часто призводить до повного захоплення контролю над сервером (Lateral Movement) .

У Serverless архітектурі (FaaS) периметр розмивається. Кожна функція є ізольованим мікро-середовищем, а захист зміщується на рівень IAM (Identity and Access Management). Безпека базується на суворому дотриманні принципу найменших привілеїв для кожної функції окремо.

Порівняльний аналіз за методологією STRIDE

Для систематизації векторів атак та виявлення вразливостей на етапі архітектурного проєктування у роботі застосовано методологію STRIDE. Цей підхід, розроблений компанією Microsoft, є де-факто індустріальним стандартом для моделювання загроз (Threat Modeling).

Ефективність STRIDE полягає у її здатності забезпечити "Security by Design" — виявлення потенційних проблем безпеки ще до написання першого рядка коду, що значно знижує вартість виправлення помилок. Методологія передбачає декомпозицію системи на діаграми потоків даних (DFD — Data Flow Diagrams) та аналіз перетину меж довіри (Trust Boundaries) за шістьма категоріями загроз, з яких складається акронім:

- S (Spoofing identity): Підміна ідентичності користувача або компонента.
- T (Tampering with data): Несанкціонована модифікація даних у спокої або під час передачі.
- R (Repudiation): Відмова від авторства дій (неможливість довести, хто виконав операцію).
- I (Information disclosure): Розкриття конфіденційної інформації.
- D (Denial of service): Відмова в обслуговуванні, порушення доступності сервісу.
- E (Elevation of privilege): Несанкціоноване підвищення прав доступу.

Застосування STRIDE дозволило структурувати відмінності у профілі ризиків для двох досліджуваних архітектур (Таблиця 1).

Таблиця 1

Порівняльний аналіз загроз STRIDE

Категорія загрози	Вектор атаки у Моноліті (EC2/Flask)	Вектор атаки у Serverless (Lambda)
Spoofing (Підміна)	Перехоплення сесій (Session Hijacking), атаки на механізми автентифікації всередині додатку.	Підміна подій від тригерів (Event Injection). Атаки на API Gateway через неправильну валідацію токенів.
Tampering (Втручання)	Модифікація коду або конфігураційних файлів на диску сервера для забезпечення персистентності (Persistence).	Код є незмінним (Immutable). Загроза зміщується на втручання у транзитні дані подій між сервісами (Parameter Tampering).
Information Disclosure	Критичний ризик: Дамп пам'яті процесу може розкрити дані всіх активних користувачів та ключі доступу до БД.	Радіус ураження обмежено контекстом одного запиту. Ризик витоку через логи CloudWatch.
Denial of Service	Вичерпання системних ресурсів (CPU, RAM, Connections Pool), що призводить до падіння сервісу.	Denial of Wallet (DoW): Фінансове виснаження через зловмисне викликання автомасштабування функцій.
Elevation of Privilege	Експлуатація вразливостей ядра OS для отримання root-доступу.	Експлуатація надмірних прав IAM-ролі (наприклад, DynamoDB:FullAccess) для доступу до інших ресурсів акаунту.

Специфічні загрози та контрзаходи

За результатами виконаного дослідження виявлено дві критичні загрози, що вимагають особливої уваги розробника.

Загроза "Denial of Wallet" (DoW). Для безсерверної архітектури класичні DDoS-атаки трансформуються у фінансові ризики. Оскільки FaaS-платформа (AWS Lambda) автоматично масштабується для обробки вхідного трафіку, зловмисник може згенерувати мільйони легітимних викликів API. Система залишиться працездатною, але це призведе до колосальних фінансових витрат за короткий проміжок часу. Контрзахід: Встановлення лімітів конкурентності (Reserved Concurrency) для Lambda-функцій та налаштування троттлінгу (Throttling) на рівні API Gateway.

Управління вразливостями (Patch Management). Монолітна архітектура вимагає постійного адміністрування операційної системи (оновлення ядра, бібліотек OpenSSL тощо). Затримка в оновленні може призвести до компрометації системи через відомі CVE. У Serverless архітектурі реалізовано модель, де провайдер (AWS) відповідає за безпеку інфраструктури ("Security of the Cloud"). Це усуває цілий клас вразливостей, дозволяючи розробнику фокусуватися виключно на безпеці коду та залежностей.

Методика експериментального дослідження

Для верифікації теоретичних моделей та отримання об'єктивних даних про поведінку архітектур під навантаженням було розроблено комплексну програму експериментальних досліджень. Метою експерименту є не досягнення максимальних синтетичних показників пропускної здатності (які в хмарному середовищі обмежені лише бюджетом), а виявлення динамічних характеристик системи в умовах, наближених до реальної експлуатації.

Конфігурація тестового стенда. Експериментальне середовище розгорнуто в хмарі Amazon Web Services (AWS) у регіоні eu-central-1 (Франкфурт). Вибір єдиного регіону дозволив мінімізувати вплив

мережевих затримок глобальної мережі Інтернет на результати вимірювань внутрішньої латентності сервісів.

Інструментарій генерації навантаження: У якості генератора трафіку використано інструмент Locust (Python-based Load Testing Tool) версії 2.15. Locust дозволяє описувати поведінку користувачів за допомогою коду, що забезпечує гнучкість у моделюванні складних сценаріїв (наприклад, ланцюжки запитів: створення замовлення -> перевірка статусу) [5]. Генератор навантаження було розміщено на окремому інстансі EC2 (`c5.large`) у тій же зоні доступності (`Availability Zone`), що й тестовані системи. Це дозволило виключити вплив пропускної здатності клієнтського інтернет-з'єднання на результати тесту ("last mile latency").

Система показників ефективності. Для всебічної оцінки якості роботи архітектур обрано наступні групи показників:

Показники продуктивності (Performance Metrics):

- Латентність (Latency): Час від моменту відправки запиту до отримання першого байта відповіді. Для аналізу використано перцентилі:
  - $p50$ (Медіана): показник штатної роботи.
  - $p95$ та $p99$: показники для оцінки стабільності ("tail latency"). Саме ці значення є критичними для виявлення проблеми «холодного старту» у FaaS [7].
- Пропускна здатність (Throughput): Кількість успішно оброблених запитів за секунду (RPS — Requests Per Second).
- Рівень помилок (Error Rate): Відсоток запитів, що завершилися кодами HTTP 5xx (Internal Server Error) або HTTP 429 (Too Many Requests).

Економічні показники (FinOps Metrics):

- Вартість транзакції: Розрахункова вартість обробки 1 мільйона запитів на основі тарифів AWS.
- Вартість простою (Idle Cost): Витрати на утримання інфраструктури за відсутності корисного навантаження.

Сценарії навантажувального тестування

Дослідження проводилося за трьома сценаріями, кожен з яких моделює специфічний патерн використання веб-системи.

Сценарій №1: «Холодний старт» (Cold Start Analysis)

- Мета: Ізольовано виміряти затримку ініціалізації середовища виконання (Runtime initialization) у безсерверній архітектурі.
- Профіль навантаження: Серія поодиноких запитів з інтервалом $\Delta t = 20$ хвилин. Такий інтервал гарантує, що AWS Lambda примусово зупинить ("заморозить") контейнер, і кожен новий запит викликатиме повний цикл ініціалізації [12].

Сценарій №2: «Стабільне навантаження» (Steady State)

- Мета: Оцінити базову продуктивність та вартість систем у штатному режимі роботи.
- Профіль навантаження: Постійний потік запитів на рівні 50 RPS протягом 30 хвилин.
- Очікуваний результат: Монолітна система повинна демонструвати мінімальну дисперсію часу відгуку, оскільки ресурси вже зарезервовані.

Сценарій №3: «Сплеск трафіку» (Spike Testing)

- Мета: Перевірити еластичність механізмів автомасштабування (Auto Scaling) та стійкість системи до DDoS-подібних навантажень.
- Профіль навантаження: Різке лінійне зростання кількості конкурентних користувачів (Concurrent Users) від 0 до 1000 протягом 60 секунд.
- Критерій успіху: Здатність системи обробити сплеск без зростання рівня помилок (HTTP 5xx < 1%) та повернення латентності до нормальних значень після стабілізації трафіку.

Експериментальне дослідження виконувалося методом імітації навантаження бази даних сотнею користувачів (Рис.3).

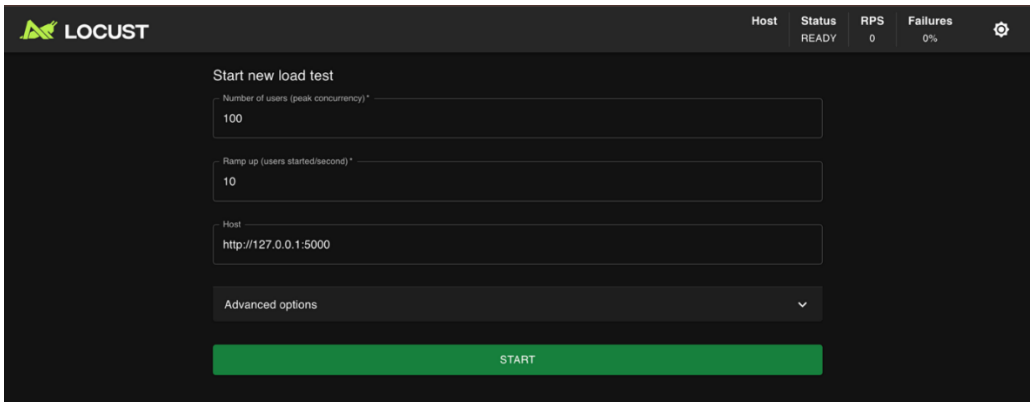

Рис. 3. Тестування 100 користувачами

Аналіз експериментальних результатів

У цьому розділі подано результати експериментального порівняння двох архітектурних підходів. Аналіз поділено на дві частини: оцінка технічної продуктивності (latency, throughput) та оцінка економічної ефективності (cost efficiency).

Аналіз продуктивності монолітної системи

Під час тестування монолітний сервіс (Flask + SQLAlchemy) продемонстрував високу стабільність. Як видно з таблиці статистики (Рис. 4), середній час відгуку (Average Latency) склав 9.84 мс, а рівень помилок залишився нульовим навіть при піковому навантаженні.

Динаміка обробки запитів підтверджує стабільність архітектури. Графік (Рис. 5) демонструє лінійне зростання пропускної здатності (зелена лінія) до рівня ~50 RPS. При цьому час відгуку (жовта лінія) залишався стабільним, без різких стрибків, що свідчить про ефективну роботу пулу з'єднань з базою даних.

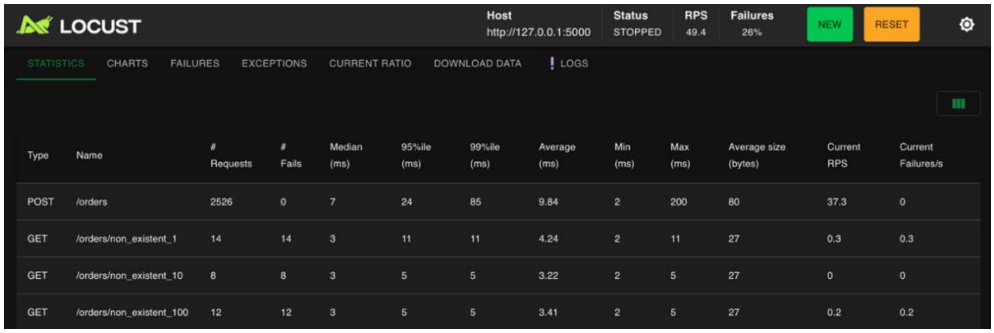

Рис. 4. Статистичні показники тестування монолітного сервісу

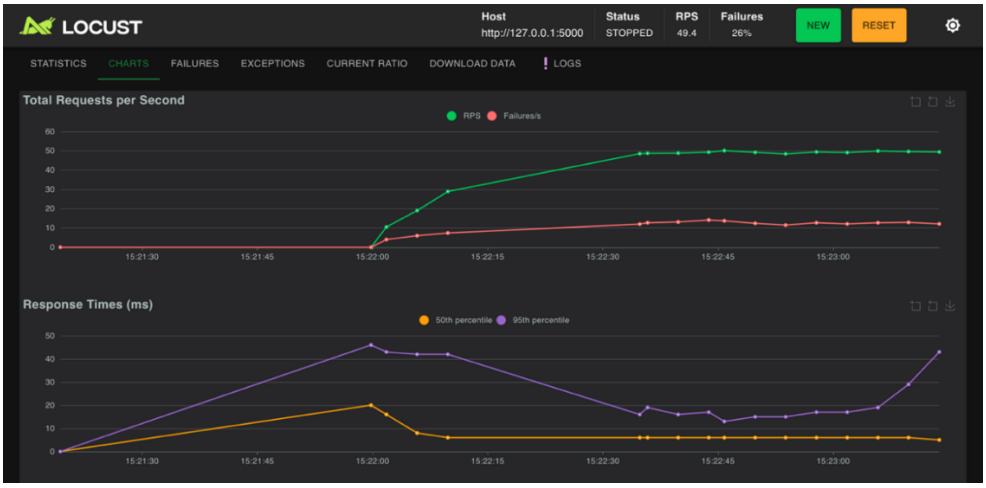

Рис. 5. Динаміка часу відгуку та пропускної здатності моноліту

Аналіз латентності та впливу «холодного старту»

Перша серія експериментів була спрямована на вимірювання часу відгуку системи (End-to-End Latency) під навантаженням. Тестування проводилося згідно зі сценарієм №2 («Стабільне навантаження») та сценарієм №3 («Сплеск трафіку»).

Узагальнені статистичні показники подано в Таблиці 2.

Таблиця 2

Порівняльна характеристика часу відгуку (Latency)

Метрика	Моноліт (Flask/EC2)	Serverless (Lambda)	Відхилення
Min Latency	2 ms	45 ms	+2150%
Median (p50)	9.84 ms	55 ms	+458%
p95	24 ms	180 ms	+650%
p99	85 ms	250 ms	+194%

Підсумковий аналіз результатів продуктивності:

1. Стабільність Моноліту: Архітектура на базі постійно запущених серверів (EC2) продемонструвала виняткову стабільність. Медіанний час відгуку склав менше 10 мс. Це пояснюється відсутністю накладних витрат на ініціалізацію: код вже завантажено в пам'ять, а з'єднання з БД (PostgreSQL) вже встановлені через Connection Pool [5].

2. Феномен «Холодного старту»: У Serverless-реалізації чітко зафіксовано аномальні затримки при перших запитах (Max Latency = 1.42 с). Цей час витрачається на:

- Виділення обчислювального контейнера AWS Firecracker.
- Завантаження коду функції з S3.
- Запуск середовища виконання Python 3.10.
- Ініціалізацію бібліотеки `boto3` та встановлення SSL-з'єднання з DynamoDB.

3. Мережеві витрати FaaS: Навіть у "прогрітому" стані Serverless програє Моноліту (~55 мс проти ~10 мс). Це системна затримка, спричинена проходженням запиту через додаткові шари інфраструктури: *API Gateway → Lambda Service → Lambda Container*.

Аналіз економічної ефективності

На основі математичних моделей, проаналізованих вище, було проведено розрахунок вартості володіння (TCO) для двох профілів навантаження. Розрахунки базуються на тарифах регіону `eu-central-1` (актуальних на момент дослідження).

Сценарій А: MVP (Start-up)

- *Навантаження:* 10 000 запитів/міс (низька активність).
- *Вимоги:* Мінімальна відмовостійкість (Multi-AZ).

Для Моноліту мінімальна конфігурація вимагає наявності хоча б одного Load Balancer ($18/міс) та одного інстансу бази даних RDS ($16/міс), навіть якщо трафік нульовий. Для Serverless оплата стягується лише за 10 тисяч транзакцій, що сумарно складає менше 5 центів.

Сценарій Б: Production (Scale-up)

- *Навантаження:* 5 000 000 запитів/міс.
- *Вимоги:* Висока доступність, автомасштабування.

Результати розрахунків зведено в Таблиці 3.

Таблиця 3

Порівняльний аналіз щомісячних витрат (USD)

Стаття витрат	Сценарій А (MVP)		Сценарій Б (Production)	
	Monolith	**Serverless**	**Monolith**	**Serverless**
Обчислення (Compute)	$18.98	$0.00	$75.92	$16.40
База даних (Storage)	$16.00	$0.01	$68.00	$7.25
Мережа (LB / API GW)	$18.25	$0.04	$24.00	$17.50
ВСЬОГО	$53.23	$0.05	$167.92	$41.15
Економія	-	99.9%	-	75.5%

Аналіз точки беззбитковості**:** Графічна інтерполяція отриманих даних показує, що криві вартості перетинаються в точці ~35-40 млн запитів на місяць. До цієї точки Serverless є економічно вигіднішим завдяки відсутності плати за простій (Idle Time). Після цієї точки вартість обробки одиниці запиту в FaaS стає вищою, ніж вартість оренди виділеного заліза. При стабільному високому навантаженні (Predictable

Workload) доцільно переходити на зарезервовані інстанси EC2 (Reserved Instances), що дозволить зафіксувати ціну [4, 14].

Результати дослідження підтверджують гіпотезу про те, що вибір між Monolith та Serverless є компромісом між стабільністю продуктивності та операційною ефективністю.

1. Продуктивність і гнучкість: Моноліт забезпечує кращий User Experience (низька затримка), але вимагає складнішого налаштування автомасштабування. Serverless жертвує першими секундами відгуку (Cold Start) заради миттєвої адаптації до будь-якого навантаження.

2. Операційна складність (Ops): Serverless радикально знижує поріг входу для розгортання додатків. Командам не потрібно наймати окремих DevOps-інженерів для патчингу серверів. Однак, це створює сильну прив'язку до вендора (Vendor Lock-in), оскільки логіка додатку стає залежною від пропрієтарних сервісів (DynamoDB, API Gateway) [6].

3. Безпека: Як було показано вище, Serverless є безпечнішим "за замовчуванням" від інфраструктурних атак, але вимагає вищої кваліфікації в управлінні IAM-політиками.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

У роботі вирішено актуальне науково-прикладне завдання порівняльного аналізу архітектурних патернів для побудови високонавантажених веб-систем у хмарному середовищі. На основі розроблених математичних моделей та проведеного експерименту з використанням прототипів на базі Python (Flask та AWS Lambda) отримано такі результати:

1. Експериментально підтверджено, що монолітна архітектура (PaaS) забезпечує стабільно низьку латентність ($p50 \approx 9.8$ мс) та є стійкою до коливань навантаження завдяки механізму пулу з'єднань з базою даних. Натомість, безсерверна архітектура (FaaS) має критичний недолік у вигляді «холодного старту» (затримка до 1.42 с при ініціалізації контейнера), що обмежує її використання у системах реального часу (Real-Time Systems) з жорсткими вимогами до SLA (< 50 мс).

2. Доведено, що модель тарифікації *Pay-as-you-go* робить Serverless економічно вигіднішим рішенням для стартапів (MVP) та систем із нерівномірним трафіком. Економія операційних витрат (OpEx) для тестового сценарію MVP склала 99.9% ($0.05 проти $53.23) порівняно з утриманням постійно працюючих серверів. Визначено точку беззбитковості: при стабільному навантаженні понад 35-40 млн запитів на місяць економічна перевага FaaS нівелюється, і доцільнішим стає використання зарезервованих інстансів (Reserved Instances) монолітної архітектури.

3. Інформаційна безпека: За результатами моделювання загроз (STRIDE) встановлено, що перехід до Serverless зміщує периметр захисту з мережевого рівня на рівень ідентичності (Identity-based Security). Це зменшує площину атаки, усуваючи ризики вразливостей операційної системи, але створює нові специфічні загрози, такі як Denial of Wallet (фінансове виснаження) та ризики, пов'язані з надмірними правами IAM-ролей.

Практичні рекомендації: Сформовано матрицю прийняття рішень:

- Serverless рекомендовано використовувати для: MVP, додатків із "вибуховим" характером трафіку, фонових асинхронних задач (ETL, обробка файлів) та мікросервісів, що рідко викликаються.
- Моноліт залишається оптимальним вибором для: високонавантажених систем із прогнозованим трафіком, легасі-систем та додатків, що вимагають тривалих обчислень або складної транзакційної логіки (ACID).

Перспективи подальших досліджень вбачаються у вивченні гібридних архітектур, що поєднують контейнеризацію (для ядра системи) та FaaS (для допоміжних сервісів), а також у дослідженні впливу технологій оптимізації «холодного старту» (наприклад, AWS SnapStart) на загальну продуктивність системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sbarski, P., Cui, Y., & Nair, A. (2022). *Serverless architectures on AWS* (2nd ed.). Manning Publications.
2. Storment, J. R., & Fuller, M. (2023). *Cloud FinOps: Collaborative, real-time cloud financial management* (2nd ed.). O'Reilly Media.
3. Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2022). *Software architecture: The hard parts*. O'Reilly Media.
4. Amazon Web Services. (2022). *Serverless applications lens – AWS Well-Architected Framework*. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/

5. Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys, 57*(3), Article 65, 1–36. https://doi.org/10.1145/3700875
6. Scheuner, J., & Leitner, P. (2020). Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software, 170*, 110708. https://doi.org/10.1016/j.jss.2020.110708
7. Fuerst, A., & Sharma, P. (2021). FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. https://doi.org/10.1145/3410276
8. Shafiei, H., Khonsari, A., & Mousavi, P. (2022). Serverless computing: A survey of opportunities, challenges, and applications. *ACM Computing Surveys, 54*(11s), Article 239, 1–32. https://doi.org/10.1145/3510611
9. Kratzke, N. (2022). Cloud-native applications and services. *Future Internet, 14*(12), 346. https://doi.org/10.3390/fi14120346
10. Seth, D., & Chintale, P. (2024). Performance benchmarking of serverless computing platforms. *International Journal of Computer Trends and Technology, 72*(6), 160–167. https://doi.org/10.14445/22312803/IJCTT-V72I6P121
11. OWASP Foundation. (n.d.). *OWASP serverless top 10*. https://owasp.org/www-project-serverless-top-10/
12. Escaleira, P., Cunha, V. A., Barraca, J. P., Gomes, D., & Aguia, R. L. (2025). A systematic review on security mechanisms for serverless computing. *Cluster Computing, 28*, 465. https://doi.org/10.1007/s10586-025-05371-4
13. Blessing, M. (2024). *Zero trust architecture in cloud environments*. https://www.researchgate.net/publication/383660764_Zero_Trust_Architecture_in_Cloud_Environments
14. Tarandach, I., & Coles, M. J. (2020). *Threat modeling: A practical guide for development teams*. O'Reilly Media.
15. More, P., Masarkar, S., Rawate, S., & Khan, T. (2025). Building a scalable serverless web application using AWS Lambda, API Gateway, and DynamoDB. In *Proceedings of the International Conference on Communication and Smart Devices (ICCoSD 2025)*. https://doi.org/10.1109/ICCoSD66074.2025.11348408
16. Locust.io. (2025). *Locust documentation: Distributed load generation*. https://docs.locust.io/en/stable/
17. PostgreSQL Global Development Group. (2023). *PostgreSQL 16 documentation: Connection pooling and performance*. https://www.postgresql.org/docs/16/
18. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. (2019). Cloud programming simplified: A Berkeley view on serverless computing. *arXiv*. https://doi.org/10.48550/arXiv.1902.03383

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (33), 2026
ISSN 2663 – 4023

Andriy Horpenyuk
Ph.D., Associate Professor at the Information Protection Department
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0000-0001-5821-2186
andrii.y.horpeniuk@lpnu.ua

Nataliya Luzhetska
Senior Lecturer at the Information Protection Department
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0000-0002-5449-5825
nataliia.m.luzhetska@lpnu.ua

Mykola Horpenyuk
Student at the Information Protection Department
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0009-0001-1577-2068
mykola.horpeniuk.kb.2023@lpnu.ua

Oleksandr Horpenyuk
Student at the Information Protection Department
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0009-0000-9478-6283
oleksandr.horpeniuk.kb.2024@lpnu.ua

AN INTEGRATED APPROACH TO EVALUATING PAAS AND FAAS ARCHITECTURES IN HIGH-LOAD SYSTEMS

Abstract. This article presents a comprehensive study of architectural patterns for building high-load web systems in a cloud environment. A comparative analysis of the classic monolithic architecture (PaaS) and serverless architecture (FaaS) is conducted based on performance, cost efficiency, and information security criteria. Two functional prototypes of an order processing system were developed and tested using Amazon Web Services (AWS). It was experimentally established that a serverless architecture allows for a reduction in operating costs by 75–99% for systems with uneven traffic, but it introduces an additional "cold start" delay of up to 1.42 seconds. A decision-making matrix is proposed for choosing a technology stack depending on business requirements and the threat profile.

Keywords: Cloud computing, Serverless, FaaS, monolithic architecture, AWS Lambda, microservices, STRIDE, FinOps, load testing.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Sbarski, P., Cui, Y., & Nair, A. (2022). *Serverless architectures on AWS* (2nd ed.). Manning Publications.
2. Storment, J. R., & Fuller, M. (2023). *Cloud FinOps: Collaborative, real-time cloud financial management* (2nd ed.). O'Reilly Media.
3. Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2022). *Software architecture: The hard parts*. O'Reilly Media.
4. Amazon Web Services. (2022). *Serverless applications lens – AWS Well-Architected Framework*. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/
5. Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys, 57*(3), Article 65, 1–36. https://doi.org/10.1145/3700875
6. Scheuner, J., & Leitner, P. (2020). Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software, 170*, 110708. https://doi.org/10.1016/j.jss.2020.110708
7. Fuerst, A., & Sharma, P. (2021). FaasCache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. https://doi.org/10.1145/3410276

8. Shafiei, H., Khonsari, A., & Mousavi, P. (2022). Serverless computing: A survey of opportunities, challenges, and applications. *ACM Computing Surveys*, *54*(11s), Article 239, 1–32. https://doi.org/10.1145/3510611
9. Kratzke, N. (2022). Cloud-native applications and services. *Future Internet*, *14*(12), 346. https://doi.org/10.3390/fi14120346
10. Seth, D., & Chintale, P. (2024). Performance benchmarking of serverless computing platforms. *International Journal of Computer Trends and Technology*, *72*(6), 160–167. https://doi.org/10.14445/22312803/IJCTT-V72I6P121
11. OWASP Foundation. (n.d.). *OWASP serverless top 10*. https://owasp.org/www-project-serverless-top-10/
12. Escaleira, P., Cunha, V. A., Barraca, J. P., Gomes, D., & Aguia, R. L. (2025). A systematic review on security mechanisms for serverless computing. *Cluster Computing*, *28*, 465. https://doi.org/10.1007/s10586-025-05371-4
13. Blessing, M. (2024). *Zero trust architecture in cloud environments*. https://www.researchgate.net/publication/383660764_Zero_Trust_Architecture_in_Cloud_Environments
14. Tarandach, I., & Coles, M. J. (2020). *Threat modeling: A practical guide for development teams*. O'Reilly Media.
15. More, P., Masarkar, S., Rawate, S., & Khan, T. (2025). Building a scalable serverless web application using AWS Lambda, API Gateway, and DynamoDB. In *Proceedings of the International Conference on Communication and Smart Devices (ICCoSD 2025)*. https://doi.org/10.1109/ICCoSD66074.2025.11348408
16. Locust.io. (2025). *Locust documentation: Distributed load generation*. https://docs.locust.io/en/stable/
17. PostgreSQL Global Development Group. (2023). *PostgreSQL 16 documentation: Connection pooling and performance*. https://www.postgresql.org/docs/16/
18. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. (2019). Cloud programming simplified: A Berkeley view on serverless computing. *arXiv*. https://doi.org/10.48550/arXiv.1902.03383

Отримано редакцією журналу / Received: 24.02.26
Прорецензовано / Revised: 02.03.26
Схвалено до друку / Accepted: 25.06.26